МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ ТЕХНОЛОГІЧНИЙ УНІВЕРСИТЕТ

І. М. Федотова-Півень, І. В. Миронець,

О. Б. Півень, С. В. Сисоєнко, Т. В. Миронюк

ОПЕРАЦІЙНІ СИСТЕМИ

Навчальний посібник

За редакцією проф. В. М. Рудницького

2019

УДК 004.451
ББК 32.973-018.2я7
34

Автори:

Федотова-Півень Ірина Миколаївна – кандидат технічних наук, доцент, доцент кафедри інформаційної безпеки та комп'ютерної інженерії;

Миронець Ірина Валеріївна – кандидат технічних наук, доцент, доцент кафедри інформаційної безпеки та комп'ютерної інженерії;

Півень Олег Борисович – кандидат фіз.-мат. наук, доцент, професор кафедри інформаційної безпеки та комп'ютерної інженерії;

Сисоєнко Світлана Володимирівна – кандидат технічних наук**,** старший викладач кафедри інформаційної безпеки та комп'ютерної інженерії;

Миронюк Титяна Василівна – кандидат технічних наук, доцент кафедри інформаційної безпеки та комп'ютерної інженерії

Рецензенти:

Семенов Сергій Геннадійович – доктор технічних наук, старший науковий співробітник, завідувач кафедри обчислювальної техніки та програмування Національного технічного університету «Харківський політехнічний інститут» Міністерства освіти і науки України;

Можаєв Олександр Олександрович – доктор технічних наук, професор, професор кафедри інформаційних технологій факультету № 4 Харківського національного університету внутрішніх справ Міністерства внутрішніх справ України;

Смірнов Олексій Анатолійович – доктор технічних наук, професор, завідувач кафедри програмування та захисту інформації Кіровоградського національного технічного університету

Рекомендовано до друку Науково-технічною радою
Черкаського державного технологічного університету,
протокол № 7 від 14 грудня 2018 р.

Федотова-Півень І. М.

34 Операційні системи : навчальний посібник. [за ред. В. М. Рудницького] / І. М. Федотова-Півень, І. В. Миронець, О. Б. Півень, С. В. Сисоєнко, Т. В. Миронюк; Черкаський державний технологічний університет. – Харків : ТОВ «ДІСА ПЛЮС», 2019. – 216 с.

ISBN 978-617-7645-93-0

В даному навчальному посібнику висвітлені основні питання щодо розвитку сучасних операційних систем, їх основних складових, архітектури, принципів і концепцій побудови. Посібник містить теоретичні та практичні матеріали, матеріали для самостійної роботи студентів, що подані у стислій, зручній для сприйняття формі.

Даний навчальний посібник може бути використаний в навчальному процесі підготовки здобувачів вищої освіти галузі знань 12 – «Інформаційні технології».

ISBN 978-617-7645-93-0

ЗМІСТ

ПЕРЕДМОВА

Освоєння принципів і алгоритмів, покладених в основу розробки операційних систем, вивчення їх внутрішньої будови, є важливою складовою частиною базової комп'ютерної підготовки у закладах вищої освіти.

Сучасна обчислювальна система складається з одного або декількох процесорів, оперативної пам'яті (ОП) і великої кількості різноманітних пристроїв для введення/виведення (ПВВ) – дисків, клавіатури, монітору, принтера, мережевого інтерфейсу, тобто є складною системою. Створення програм, які коректно і оптимально працюють з усіма цими компонентами, являє собою складне завдання. Тому комп'ютери оснащують спеціальним рівнем програмного забезпечення (ПЗ), що називається операційною системою (ОС). ОС відповідає за керування усіма, вище зазначеними пристроями і забезпечує користувача простим і зручним інтерфейсом для роботи з апаратурою.

Основне призначення даного навчального посібника полягає у системному відображенні основних понять, щодо складу, основного призначення і функцій сучасних операційних систем. Беручи до уваги те, що розвиток операційних систем не стоїть на місці і в сучасній літературі досить широко і глибоко представлені питання, що стосуються розвитку, складу, функцій і основних процесів в операційних системах (автори Е. Тоненбаум, В. Столлингс, В. А. Шеховцов та ін.), основною метою даного посібника є подання існуючої обширної інформації в стислому і доступному для швидкого сприйняття вигляді.

Зміст посібника структуровано на дві частини: теоретичну і практичну, а також додатки.

Перша частина посібника присвячена короткому теоретичному курсу з основних питань дисципліни операційні системи. Теоретичний матеріал поданий в стислому вигляді, але охоплює всі питання, що вивчаються в даному курсі. Метою першої частини є допомогти студентам впорядкувати знання і інформацію, що отримані під час лекційного курсу.

Друга частина посібника містить курс практичних робіт, що спрямовані на практичне підкріплення теоретичних питань, які розглядаються в курсі дисципліни Операційні системи.

Більшість розроблених і запропонованих практичних робіт спрямовані на вивчення операційної системи LINUX.

LINUX – це UNIX-подібна ОС. На сьогодні всі найважливіші програмні пакети адаптовані під LINUX, тобто для LINUX тепер доступні комерційні розробки.

Переваги LINUX:

багатоплатформенність; 2) багатопроцесорність; 3) багатокористувацький режим; 4) багатозадачний режим; 5) підтримка всіх основних мережевих протоколів; 6) підтримка квотування дискового простору; 7) надання повного вихідного тексту; 8) багатомовна підтримка для шрифтів і клавіатур; 9) керування завданнями; 10) емуляція математичного сопроцесора; 11) захист пам’яті; 12) роздільчі бібліотеки; 13) підтримка більше 10 файлових систем; 14) віртуальні консолі; 15) віртуальна пам’ять.

LINUX може працювати і на мережевих серверах, і на настільних системах.

додатках посібника подано тестові завдання для перевірки рівня засвоєння курсу, теми реферативних робіт для самостійної роботи студентів.

ЧАСТИНА І

РОЗДІЛ 1[1]

АРХІТЕКТУРА, ПРИЗНАЧЕННЯ І ФУНКЦІЇ ОПЕРАЦІЙНИХ СИСТЕМ

1.1 Функції, класифікації і складові ОС

Операційна система (ОС) – це програма, що керує роботою інших програм, як прикладних, так і системних, і є інтерфейсом між програмами і апаратним забезпеченням комп'ютера. Тому, можемо стверджувати, що найголовнішими функціями, які виконує операційна система є:

Інтерфейс між користувачем і комп'ютером.

Керування ресурсами.

Також виділяють і інші функції операційних систем.

ФУНКЦІЇ ОС

ОСНОВНІ	***ДОДАТКОВІ***
Виконання на вимогу програм користувача тих елементарних (низькорівневих) дій, які є спільними для більшості програмного забезпечення і часто зустрічаються майже у всіх програмах (введення та виведення даних, запуск і зупинка інших програм, виділення та вивільнення додаткової пам'яті тощо).	Паралельне або псевдопаралельне виконання задач (багатозадачність).
Стандартизований доступ до периферійних пристроїв (пристрої вводу-виводу).	Розподіл ресурсів обчислювальної системи між процесами.
Завантаження програм в оперативну пам'ять і їх виконання.	Організація надійних обчислень, що полягає у розмежуванні доступу до ресурсів. Тобто унеможливлювання впливу процесу на перебіг інших процесів та на сам ресурс.
Керування оперативною пам'яттю (розподіл між процесами, організація віртуальної пам'яті).	Взаємодія між процесами: обмін даними, синхронізація.
Керування доступом до даних енергонезалежних носіїв (жорсткий диск, оптичні диски, тощо), організованим у тій чи іншій файловій системі.	Захист самої системи, а також користувацьких даних і програм від дій користувача або програм.
Забезпечення користувацького інтерфейсу.	Багатокористувацький режим роботи та розділення прав доступу (автентифікація, авторизація).

[1] При підготовці даного розділу були використані літературні джерела [1–4], [9–11], [12], [15], [17], [21], [23] та ін.

Пересічним користувачем комп'ютерна система сприймається, як набір додатків (прикладних програм) і його не цікавлять подробиці апаратного забезпечення комп’ютерної системи. Додатки створюють програмісти на одній мов програмування і для спрощення програмування функцій управління комп'ютером розроблено цілий набір системних програм (деякі з них називаються утилітами), які реалізують часто використовувані функції (роботу файлами, управління пристроями введення/виведення і т. д.). Програмісти використовують ці засоби при розробці власних програм. Зрозуміло, що найбільш важливою частиною системних програм є ОС, що безпосередньо взаємодіє з апаратурою. Отже, маємо сприймати ОС як посередника, що забезпечує і спрощує користувачеві (програмісту), а також програмним додаткам доступ до різних служб, зменшує залежність від конкретної реалізації апаратного забезпечення. Також ОС управляє ресурсами, розподіляючи їх між програмами, або у часі (розподіл доступу до процесора, принтеру), або в просторі (використання оперативної пам'яті, дискового простору).

На рисунку 1.1 подана ієрархічна структура програмного забезпечення.

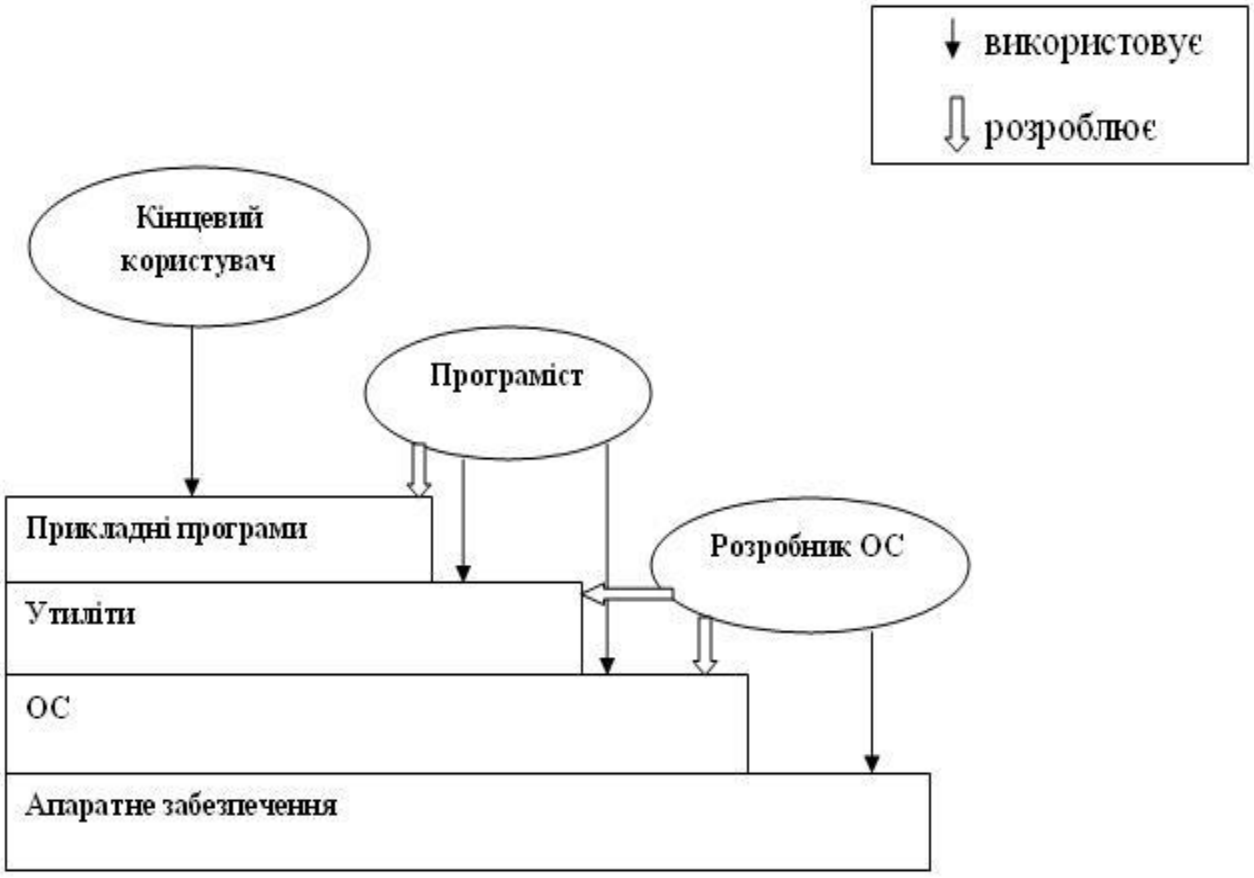


Рисунок 1.1 – Ієрархічна структура програмного забезпечення

На рисунку 1.2 подано більш докладну схему ієрархічної структури програмно-апаратного забезпечення комп’ютера.

Самий нижній рівень містить різні пристрої комп'ютера, що складаються мікросхем, провідників, джерел живлення і т.п. Цей рівень можна розділити на підрівні, наприклад контролери пристроїв, а потім самі пристрої. Можливий розподіл і на більше число рівнів.

Вище розташований мікроархітектурний рівень, на якому фізичні пристрої розглядаються як окремі функціональні одиниці.

На мікроархітектурному рівні знаходяться внутрішні регістри центрального процесора (їх може бути декілька) і арифметико-логічні пристрої (АЛП) з засобами управління ними. На цьому рівні реалізується виконання машинних команд. У процесі виконання команд використовуються регістри процесора і пристроїв, а також інші можливості апаратури.

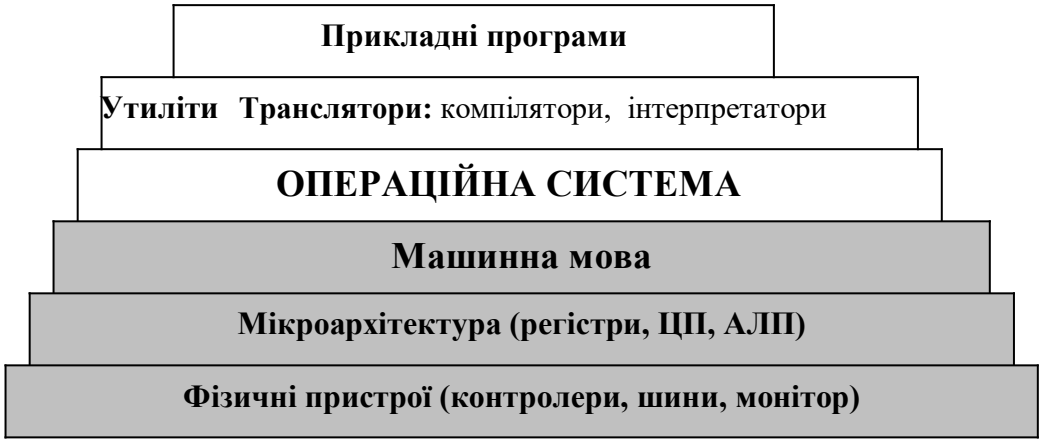


Рисунок 1.2 – Ієрархічна структура програмно-апаратних засобів комп’ютера:

- рівні доступні для розробників ОС;
рівні доступні для користувача і програміста.

Команди, видимі для працюючого на асемблері програміста, формують рівень ISA (Instruction Set Architecture – архітектура системи команд), так звана машинна мова.

Операційна система призначена для того, щоб приховати всі ці складності. Таким чином, операційна система – це набір програм, які контролюють роботу прикладних програм і системних додатків, і виконує роль інтерфейсу між користувачами, програмістами, прикладними програмами, системними додатками і апаратним забезпеченням комп'ютера.

Сервіси, що надають типові операційні системи:

Розробка програм.

Виконання програм.

Доступ до пристроїв введення-виведення.

Доступ до файлів.

Розподіл ресурсів.

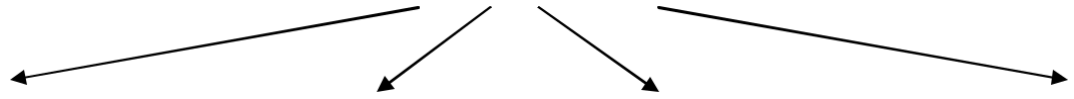
Визначення і обробка помилок.

Облік використання ресурсів.

Засоби комунікації.

Захист даних і адміністрування.

Класифікація ОС



За режимом обробки задач ***За способом взаємодії з користувачем*** ***За призначенням*** ***За принципом побудови***

За режимом обробки задач.

Однопрограмні (наприклад, MS-DOS).

Мультипрограмні (більшість сучасних ОС).

За способом взаємодії з користувачем.

Діалогові ОС.

Системи пакетної обробки.

За призначенням.

ОС мейнфреймів (орієнтовані на одночасну обробку великої кількості завдань, що потребують багато операцій введення/виведення; можуть виконувати три види обслуговування (часто – всі разом): пакетну обробку, обробку транзакцій і роботу в режимі поділу часу. Приклад: OS/390, що створена на основі OS/360).

Серверні ОС (UNIX, Windows 2000, Linux).

Мережеві ОС (при роботі з такою ОС користувачі знають про існування в мережі безлічі ЕОМ, можуть реєструватися на віддалених

комп'ютерах і копіювати файли з однієї ЕОМ на іншу. Кожен комп'ютер при цьому працює під управлінням локальної ОС і має свого власного локального користувача (користувачів)).

Розподілені ОС (розподілені ОС повинні мати в своєму складі спеціальні програми для оптимізації розпаралелювання задач. Приклади розподілених ОС: Mach, Chorus, Sprite).

ОС для персональних комп'ютерів (Windows, Mac OS, Linux).

ОС реального часу (обробляє завдання, що надходять протягом заданих інтервалів часу, які не можна перевищувати; використовуються для керування технологічними процесами і лабораторними експериментами, в роботах систем управління повітряним рухом, в телекомунікаціях).

Вбудовані ОС (для забезпечення роботи мікрокомп'ютерів, вбудованих в різні побутові пристрої, мають такі ж характеристики, що і ОС реального часу, повинні займати невелике місце в пам'яті з урахуванням обмеження по обчислювальній потужності мікрокомп'ютерів. Приклади вбудованих ОС: PalmOS, Windows CE.

За принципом побудови.

Макроядерні (Монолітні).

Мікроядерні.

Ієрархічні.

1.2 Структура операційних систем

Для більш докладного роз'яснення класифікації ОС за принципом побудови з'ясуємо з чого складається операційна система.

До складу операційної системи входять:

ядро операційної системи, що забезпечує розподіл та управління ресурсами обчислювальної системи;

базовий набір прикладного програмного забезпечення;

системні бібліотеки та програми обслуговування.

Ядро операційної системи — це набір функцій, структур даних та окремих програмних модулів, які завантажуються в пам'ять комп'ютера при завантаженні операційної системи та забезпечують три типи системних сервісів:

– управління вводом-виводом інформації (підсистема введення/виведення ОС);
– управління оперативною пам’яттю (підсистема управління оперативною пам’яттю ОС);
управління процесами (підсистема управління процесами ОС).

Типова ОС складається з таких компонентів:
керування процесами;
керування пам’яттю;
підсистема вводу - виводу;
файлова система;
підтримка мережі;
користувацький інтерфейс.

Структура ОС визначає, як пов’язані між собою її компоненти.

Монолітна структура (структура макроядра). Всі програми ОС монолітної структури зібрані в один модуль. Така ОС являє собою набір процедур, кожна з яких при необхідності може викликати інші процедури, так як в монолітній ОС всі процедури відтрансльовані і скомпоновані в єдиний виконуваний файл. У порівнянні з іншими структурами, монолітну ОС важко розробляти, відлагоджувати, розширювати, і як наслідок у неї низька надійність, але більш висока швидкодія. Приклади таких ОС: MS-DOS, ранній Unix.

Ієрархічна структура (багатошарове ядро). Основні принципи побудови ОС ієрархічної структури:

кожний рівень контролює свій ряд структур даних і надає функції доступу до них;

кожний рівень нічого «не знає» про структуру рівнів, що знаходяться нижче, доступ до сервісів, яких відбувається через певний інтерфейс;

кожний рівень нічого «не знає» про існування рівнів, що знаходяться вище.

Переваги ієрархічної ОС: модульність, простота відладки і підтримки, можливість заміни або модифікації окремого шару.

Структура мікроядра. В такій ОС ядро виконує невелику кількість самих важливих функцій: робота з адресним простором, міжпроцесна взаємодія, основне планування. Інші функції ОС виконуються процесами або серверами. Ці процеси запускаються в режимі користувача, а мікроядро працює ними таким же чином, як і з іншими додатками. При цьому спрощується розробка ОС, підвищується її гнучкість, але разом з тим відбувається зменшення швидкодії обчислювальних процесів.

1.3 Системні виклики

Функції ядра операційної системи можуть бути виконані внаслідок виконання в прикладних програмах спеціальних функцій – *системних викликів*.

Більшість ОС можуть працювати в двох режимах:

в режимі користувача (*user mode*), в якому не дозволені привілейовані операції, прямий доступ до пристроїв введення/виведення і деяким керуючим регістрам;

в системному режимі (*system mode*), в якому дозволені привілейовані операції, прямий доступ до пристроїв введення/виведення і до всіх керуючих регістрів. Системний режим також називається режимом ядра, або режимом супервізора.

Необхідність системних викликів виникає тому, що прикладні програми не в змозі самотужки визначити, за якими адресами знаходяться функції ядра.

Системний виклик в один з машинно-залежних способів реалізує механізм отримання адрес функцій ядра та передачу в ці функції необхідних параметрів системного виклику, а також отримання результату системного виклику. Найчастіше системні виклики забезпечуються через систему переривань, завдяки чому адреса функції ядра не тільки обраховується апаратно (в процесі обробки переривання), але й забезпечується захист інформаційних ресурсів ядра. Системні виклики найчастіше мають синтаксис функції мови програмування, якою написано ядро ОС.

1.4 Поняття операційного і програмного середовища

Образно можна сказати, що апаратура комп'ютера надає «сиру» обчислювальну потужність, а завдання операційної системи полягає в тому, щоб зробити використання цієї обчислювальної потужності доступним і, по можливості, зручним для користувача. Програміст може не знати деталі управління конкретними ресурсами (наприклад, диском) комп'ютера і повинен звертатися до операційної системи з відповідними викликами, щоб отримати від неї необхідні сервіси та функції. Цей набір сервісів і функцій і є операційним середовищем, в якому виконуються прикладні програми.

Таким чином, о***перацiйне середовище*** – це інтерфейси, що необхідні програмам і користувачам для отримання сервісів ОС. Операційне середовище включає в себе ***програмне середовище*** та інтерфейс користувача.

Програмне середовище, що безпосередньо утворене кодом ОС, називається основним. Окрім нього, в ОС можуть бути організовані додаткові програмні середовища шляхом емуляції іншої операційної системи. Таким чином можливо забезпечити виконання програм, що створені для інших операційних систем.

Звернення користувачів (програм) до функцій ОС відбувається за допомогою систем запитів або викликів.

Для кожної ОС характерним є свій набір системних викликів, який являє собою сукупність системних викликів і правила їх застосування (***API***, Application Programming Interface – ***Інтерфейс прикладного програмування***).

Програма, що створена для однієї ОС, скоріш за все не буде працювати в іншій, оскільки ці ОС мають різні API. Для подолання такого обмеження розробляються програмні середовища.

Програмне середовище – це системне програмне оточення, що дозволяє виконувати системні запити від прикладних програм.

1.5 Архітектура операційних систем

Під архітектурою операційної системи розуміють структурну і функціональну організацію ОС на основі деякої сукупності програмних модулів. До складу ОС входять виконувані і об'єктні модулі стандартних для

даної ОС форматів, програмні модулі спеціального формату (наприклад, завантажувач ОС, драйвери введення/виведення), конфігураційні файли, файли документації, модулі довідкової системи і т.д.

На архітектуру ранніх операційних систем зверталося мало уваги: по-перше, ні в кого не було досвіду в розробці великих програмних систем, а по-друге, проблема взаємозалежності та взаємодії модулів недооцінювалася.

монолітних ОС майже всі процедури могли викликати одна іншу. Така відсутність структури стала несумісною з розширенням операційних систем. Перша версія ОС OS/360 була створена колективом з 5000 робітників за 5 років і містила понад 1 млн рядків коду. Розроблена дещо пізніше операційна система Mastics містила до 1975 року вже 20 млн рядків. Стало зрозумілим, що розробка таких систем повинна вестися на основі модульного програмування.

Більшість сучасних ОС являють собою добре структуровані модульні системи, здатні до розвитку, розширення та переносу на нові платформи. Якої-небудь єдиної уніфікованої архітектури ОС не існує, але відомі універсальні підходи до структурування ОС. Принципово важливими універсальними підходами до розробки архітектури ОС є:

модульна організація;

функціональна надлишковість;

функціональна вибірковість;

параметрична універсальність;

концепція багаторівневої ієрархічної обчислювальної системи, за якою ОС є багатошаровою структурою;

поділ модулів на дві групи за функціями: ядро – модулі, що виконують основні функції ОС, і модулі, що виконують допоміжні функції ОС;

поділ модулів ОС на дві групи за розміщенням в пам'яті обчислювальної системи: *резидентні*, які постійно перебувають в оперативній пам'яті, і *транзитні*, що завантажуються в оперативну пам'ять тільки на час виконання своїх функцій;

реалізація двох режимів роботи обчислювальної системи: привілейованого режиму (режиму ядра – *Kernel mode*), або режиму супервізора (*supervisor mode*), і режиму користувача (*user mode*), або режиму завдання (*task mode*);

обмеження функцій ядра (а отже, і кількості модулів ядра) до мінімальної кількості необхідних найважливіших функцій.

Перші ОС розроблялися як монолітні системи без чітко вираженої структури (рисунок 1.3).

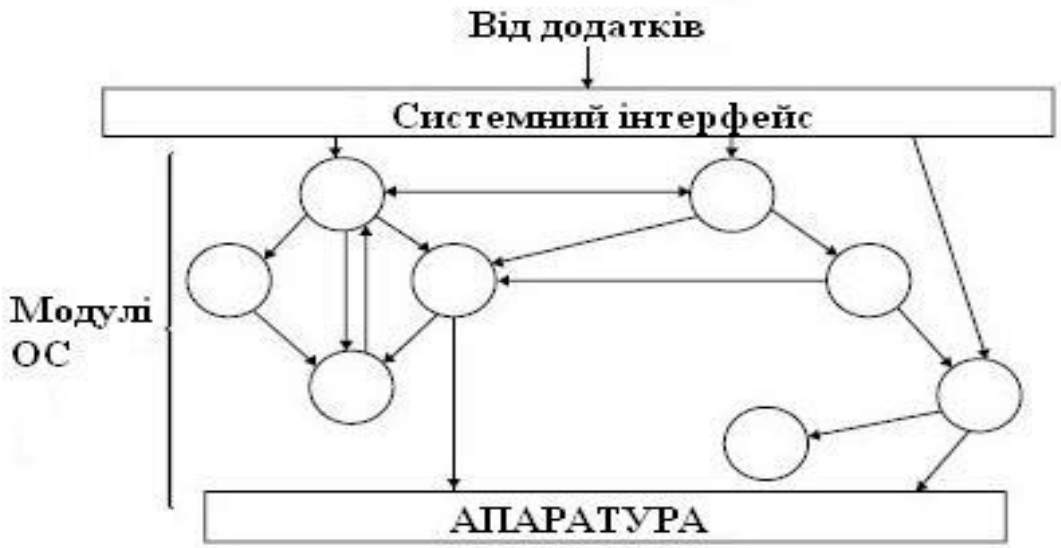


Рисунок 1.3 – Монолітна архітектура

Для побудови монолітної системи необхідно скомпілювати всі окремі процедури, а потім зв'язати їх разом в єдиний об'єктний файл за допомогою компонувальника (прикладами можуть служити ранні версії ядра UNIX або Novell NetWare). Кожна процедура бачить будь-яку іншу процедуру (на відміну від структури, що містить модулі, в якій більша частина інформації є локальною для модуля, і процедури модуля можна викликати тільки через спеціально визначені точки входу).

Але навіть такі монолітні системи можуть бути трохи структурованими. При зверненні до системних викликів, що підтримуються ОС, параметри поміщуються в строго визначені місця, такі як регістри або стек, а потім виконується спеціальна команда переривання, відома як виклик ядра чи виклик супервізора. Ця команда перемикає машину з режиму користувача в режим ядра, званий також режимом супервізора і передає управління ОС. Потім ОС перевіряє параметри виклику, для того щоб визначити, який системний виклик повинен бути виконаний. Після цього ОС індексує таблицю, яка містить посилання на процедури, і викликає відповідну процедуру. Схема структурованої архітектури подана на рисунку 1.4.

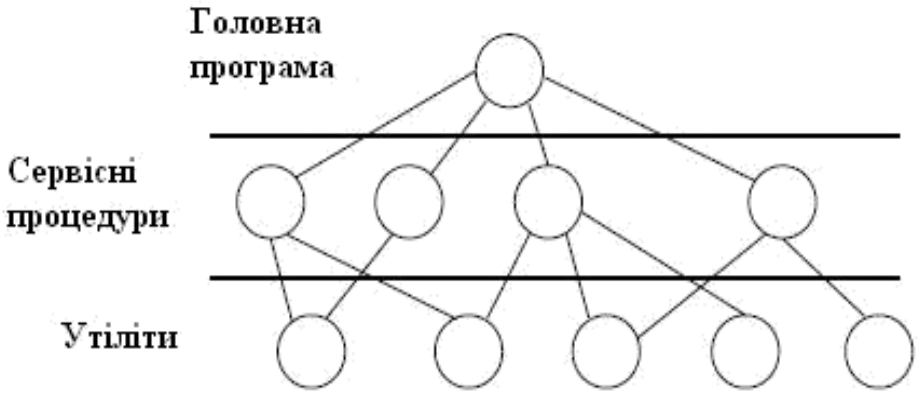


Рисунок 1.4 – Структурована архітектура

цій моделі для кожного системного виклику є одна сервісна процедура. Утиліти виконують функції, які потрібні декільком сервісним процедурам.

Класичною вважається архітектура ОС, заснована на концепції ієрархічної багаторівневої машини, привілейованому ядрі і режимі користувача роботи транзитних модулів. Модулі ядра виконують базові функції ОС: управління процесами, пам'яттю, пристроями введення/виведення і т.п. Ядро складає серцевину ОС, без якої вона є повністю непрацездатною і не може виконати ні одну зі своїх функцій. У ядрі вирішуються внутрішньосистемні задачі організації обчислювального процесу, недоступні для додатку.

Особливий клас функцій ядра служить для підтримки додатків, створюючи для них так зване прикладне програмне середовище. Додатки можуть звертатися до ядра з запитами - системними викликами - для виконання тих чи інших дій, наприклад, відкриття та читання файлу, отримання системного часу, виведення інформації на дисплей і т.д. Функції ядра, які можуть викликатися додатками, утворюють інтерфейс прикладного програмування – API (Application Programming Interface).

Для забезпечення високої швидкості роботи ОС модулі ядра (принаймні, більша їх частина) є резидентними і працюють в привілейованому режимі (Kernel mode). Цей режим, по-перше, повинен забезпечити безпечну роботу самої ОС від втручання додатків, і, по-друге, повинен забезпечити можливість роботи модулів ядра з повним набором машинних інструкцій, що дозволяють власне ядру виконувати управління ресурсами комп'ютера, зокрема, перемикання процесора з задачі на задачу, управління пристроями введення/виведення, розподілом і захистом пам'яті та ін.

Решта модулі ОС виконують не настільки важливі функції, як ядро, є транзитними. Наприклад, це можуть бути програми архівування даних, дефрагментації диска, стиснення дисків, очищення дисків і т.п.

Допоміжні модулі звичайно поділяються на групи:

утиліти – програми, які виконують окремі завдання управління і супроводу обчислювальної системи;

системні опрацьовуючі програми – текстові та графічні редактори (Paint, Imaging в Windows), компілятори та ін;

програми надання користувачу додаткових послуг (спеціальний варіант для користувача інтерфейсу, калькулятор, ігри, засоби мультимедіа Windows);

бібліотеки процедур різного призначення, що спрощують розробки додатків, наприклад, бібліотека функцій введення/виведення, бібліотека математичних функцій і т.п.

Ці модулі ОС оформляються як звичайні додатки, звертаються до функцій ядра за допомогою системних викликів і виконуються в режимі користувача (user mode). У цьому режимі забороняється виконання деяких команд, які пов'язані з функціями ядра ОС (управління ресурсами, розподіл і захист пам'яті і т.п.).

концепції багаторівневої (багатошарової) ієрархічної машини структура ОС також представляється рядом шарів. При такій організації кожен шар обслуговує вищерозміщений шар, виконуючи для нього деякий набір функцій, які утворюють міжшаровий інтерфейс. На основі цих функцій наступний верхній за ієрархією шар будує свої функції – більш складні і більш потужні і т.д. Така організація системи істотно спрощує її розробку, тому дозволяє спочатку «зверху вниз» визначити функції шарів і міжшарові інтерфейси, а при детальній реалізації, рухаючись «знизу вгору», – нарощувати потужність функцій шарів. Крім того, модулі кожного шару можна змінювати без необхідності змін в інших шарах (але не змінюючи міжшарових інтерфейсів!).

Багатошарова структура ядра ОС може бути представлена, наприклад, варіантом, показаним на рисунку 1.5.

Рисунок 1.5 – Багатошарова структура ОС

У даній схемі виділені наступні шари.

Засоби апаратної підтримки ОС. Як правило, в сучасних системах завжди є засоби апаратної підтримки ОС, які прямо беруть участь в організації обчислювальних процесів. До них відносяться: система переривань, засоби підтримки привілейованого режиму, засоби підтримки віртуальної пам'яті, системний таймер, засоби перемикання контекстів процесів (інформація про стан процесу в момент його припинення), засоби захисту пам'яті та ін.

Машинно-залежні модулі ОС. Відображається специфіка апаратної платформи комп'ютера. Призначення цього шару - "екранування" верхніх шарів ОС від особливостей апаратури (наприклад, у Windows - це шар HAL (Hardware Abstraction Layer), рівень апаратних абстракцій).

Базові механізми ядра. Цей шар модулів виконує найбільш примітивні операції ядра: програмне переключення контекстів процесів, диспетчеризацію переривань, переміщення сторінок між основною пам'яттю і диском і т.п. Модулі цього шару не приймають рішень про розподіл ресурсів, а тільки обробляють рішення, прийняті модулями вищерозміщених рівнів.

Менеджери ресурсів. Модулі цього шару виконують стратегічні завдання управління ресурсами обчислювальної системи. Це менеджери (диспетчери) процесів введення/виведення, оперативної пам'яті і файлової системи. Кожен менеджер веде облік вільних і використовуваних ресурсів і планує їх розподіл відповідно запитам додатків.

Інтерфейс системних викликів. Це верхній шар ядра ОС, який взаємодіє з додатками і системними утилітами, він утворює прикладний програмний інтерфейс ОС. Функції API, що обслуговують системні виклики, надають доступ до ресурсів системи в зручній компактній формі, без зазначення деталей їхнього фізичного розташування.

Для підвищення стійкості і надійності ОС в сучасних системах передбачений привілейований режим ядра, при якому відбувається деяке уповільнення виконання системних викликів. Системний виклик привілейованого ядра ініціює перемикання процесора з режиму користувача в привілейований, а при поверненні до додатка – зворотнє перемикання. Недоліком такого підходу є виникнення додаткової затримки в обробці системного виклику, але таке рішення стало класичним і використовується в багатьох ОС (UNIX, VAX, VMS, IBM OS/390, OS / 2 та ін.)

Недоліки багатошарової класичної багаторівневої архітектури ОС: значні зміни одного з рівнів можуть призвести до непередбачуваного впливу на суміжні рівні; 2) численні взаємодії між сусідніми рівнями ускладнюють забезпечення безпеки.

Тому, як альтернатива класичному варіанту архітектури ОС, часто використовується ***мікроядерна архітектура ОС***. При такій архітектурі ОС у привілейованому режимі залишається працювати тільки дуже невелика частина ОС, що називається *мікроядром.*

Мікроядро захищене від інших частин ОС і додатків. До його складу входять машинно-залежні модулі, а також модулі, що виконують базові механізми звичайного ядра. Всі інші більш високорівневі функції ядра оформляються як модулі, що працюють в режимі користувача. Так, менеджери ресурсів, які є невід'ємною частиною звичайного ядра, стають «периферійними» модулями, що працюють в режимі користувача. Таким чином, в архітектурі з мікроядром традиційне розташування рівнів по вертикалі замінюється горизонтальним (рисунок 1.6).

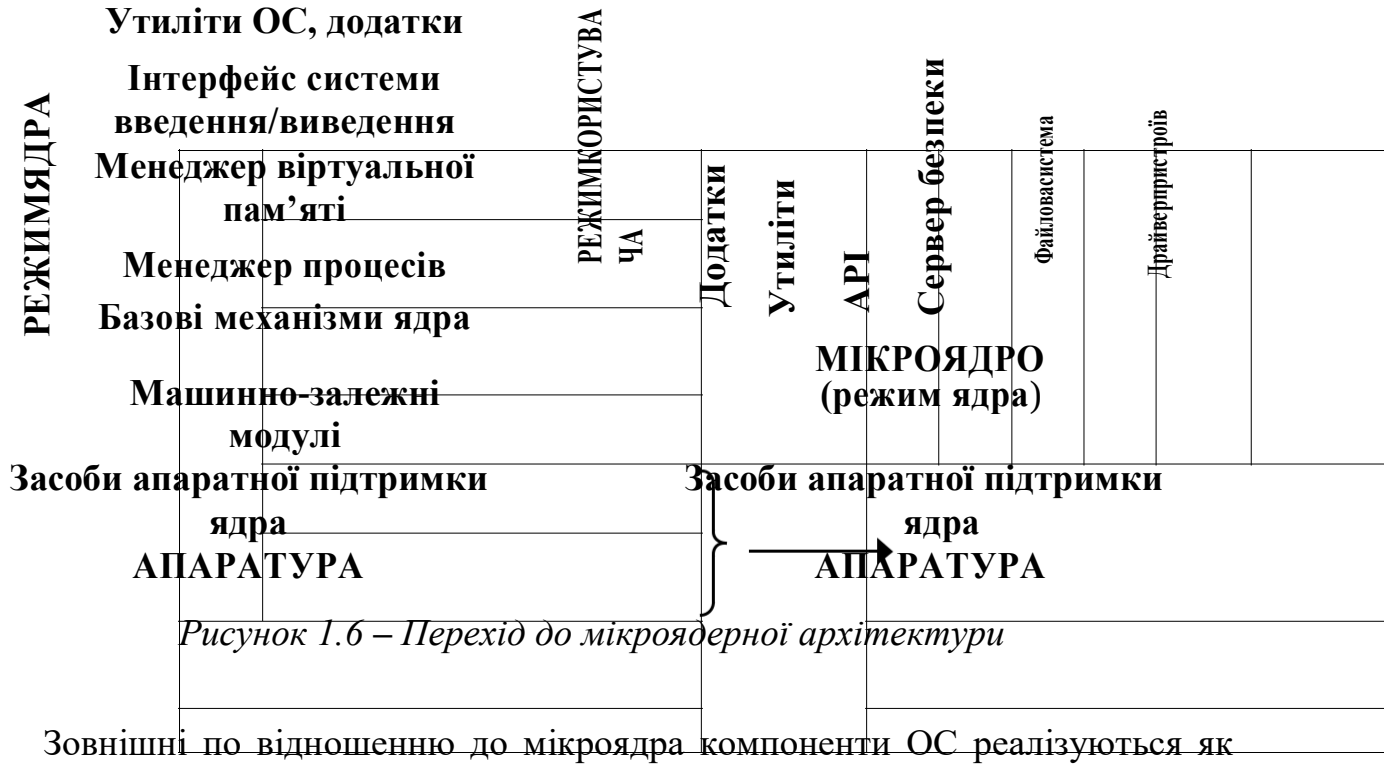


Рисунок 1.6 – Перехід до мікроядерної архітектури

Зовнішні по відношенню до мікроядра компоненти ОС реалізуються як обслуговуючі процеси. Між собою вони взаємодіють як рівноправні партнери за допомогою обміну повідомленнями, які передаються через мікроядро. Оскільки призначенням цих компонентів ОС є обслуговування запитів додатків користувачів, утиліт і системних обробляючих програм, менеджери ресурсів, винесені в користувацький режим, називаються серверами ОС, тобто модулями, основним призначенням яких є обслуговування запитів локальних додатків і інших модулів ОС.

Механізм звернення до функцій ОС у вигляді серверів формує клієнт-серверну архітектуру (рисунок 1.7).

Рисунок 1.7 – Клієнт-серверна архітектура

багатьох джерелах визнані переваги мікроядерної архітектури: однотипні інтерфейси, простота розширюваності, висока гнучкість, висока надійність, підтримка розподілених систем, підтримка об'єктно-орієнтованих ОС.

В сучасних операційних системах розрізняють наступні ***види ядер***.

Наноядро. Украй спрощене і мінімальне ядро, виконує лише одну задачу – обробку апаратних переривань, що генеруються пристроями комп'ютера. Після обробки посилає інформацію про результати обробки вищерозміщеному програмному забезпеченню. Наноядро використовується для віртуалізації апаратного забезпечення реальних комп'ютерів.

Мікроядро надає тільки елементарні функції управління процесами і мінімальний набір абстракцій для роботи з обладнанням. Велика частина роботи здійснюється за допомогою спеціальних користувацьких процесів – сервісів. У микроядерній операційній системі можна, не перериваючи її роботу, завантажувати і вивантажувати нові драйвери, файлові системи і т.д. Мікроядерними є ядра ОС Minix та GNU Hurd і ядро систем сімейства BSD. Класичним прикладом мікроядерної системи є Symbian OS. Це приклад поширеної і відпрацьованої микроядерної (починаючи с версії Symbian OS v8.1, наноядерної) операційної системи.

Екзоядро надає лише набір сервісів для взаємодії між додатками, а також необхідний мінімум функцій, пов'язаних із захистом: виділення і вивільнення ресурсів, контроль прав доступу і т.д. Екзоядро не займається наданням абстракцій для фізичних ресурсів – ці функції виносяться в бібліотеку користувацького рівня (так звану libOS). Мікроядра ОС, що базуються на екзоядрі, забезпечують більшу ефективність за рахунок відсутності необхідності перемиканні між процесами при кожному зверненні до обладнання.

Монолітне ядро надає широкий набір абстракцій обладнання. Всі частини ядра працюють в одному адресному просторі. Монолітне ядро вимагає перекомпіляції при зміні складу обладнання. Компоненти операційної системи не самостійними модулями, а складовими частинами однієї програми. Таке ядро більш продуктивне, ніж мікроядро, оскільки працює як один великий процес. Монолітним ядром є більшість Unix-систем і Linux. Монолітність ядер ускладнює налагодження, розуміння коду ядра, додавання нових функцій і можливостей, видалення непотрібного, успадкованого від попередніх версій

коду. «Розбухання» коду монолітних ядер також підвищує вимоги до обсягу оперативної пам'яті.

Модульне ядро – сучасна, вдосконалена модифікація архітектури мікроядра. На відміну від «класичних» ОС з архітектурою мікроядра, модульні ядра не вимагають повної перекомпіляції ядра при зміні складу апаратного забезпечення комп'ютера. Замість цього вони надають той чи інший механізм підвантаження модулів, що підтримують те чи інше апаратне забезпечення (наприклад, драйверів). Підвантаження модулів може бути як динамічним, так і статичним (при перезавантаженні ОС після переконфігурації системи). Модульне ядро зручніше для розробки, ніж традиційні монолітні ядра. Вони надають програмний інтерфейс (API) для зв'язування модулів з ядром, для забезпечення динамічного підвантаження і вивантаження модулів. Не всі частини ядра можуть бути зроблені модулями. Деякі частини ядра завжди зобов'язані бути присутніми в оперативній пам'яті і повинні бути жорстко «вшиті» в ядро.

Гібридне ядро - модифіковані мікроядра, що дозволяють для прискорення роботи запускати "несуттєві" частини в просторі ядра. Мають «гібридні» переваги і недоліки. Прикладом змішаного підходу може служити можливість запуску операційної системи з монолітним ядром під управлінням мікроядра. Мікроядро забезпечує управління віртуальною пам'яттю і роботу низькорівневих драйверів. Всі інші функції, в тому числі взаємодія з прикладними програмами, здійснюються монолітним ядром. Даний підхід сформувався в результаті спроб використати переваги мікроядерної архітектури, зберігаючи по можливості добре налагоджений код монолітного ядра.

Найбільш тісно елементи мікроядерної архітектури та елементи монолітного ядра переплетені в ядрі Windows NT. Хоча Windows NT часто називають микроядерною операційною системою, це не зовсім так. Мікроядро NT надто велике (більше 1 Мбайт), щоб носити приставку «мікро». Компоненти ядра Windows NT розташовуються в пам'яті і взаємодіють один з одним шляхом передачі повідомлень, як і належить в мікроядерних операційних системах. У той же час всі компоненти ядра працюють в одному адресному просторі і активно використовують загальні структури даних, що властиво операційним системам з монолітним ядром.

1.5.1 Архітектура ОС Windows

На рисунку 1.8 представлена загальна структура Windows.[2]

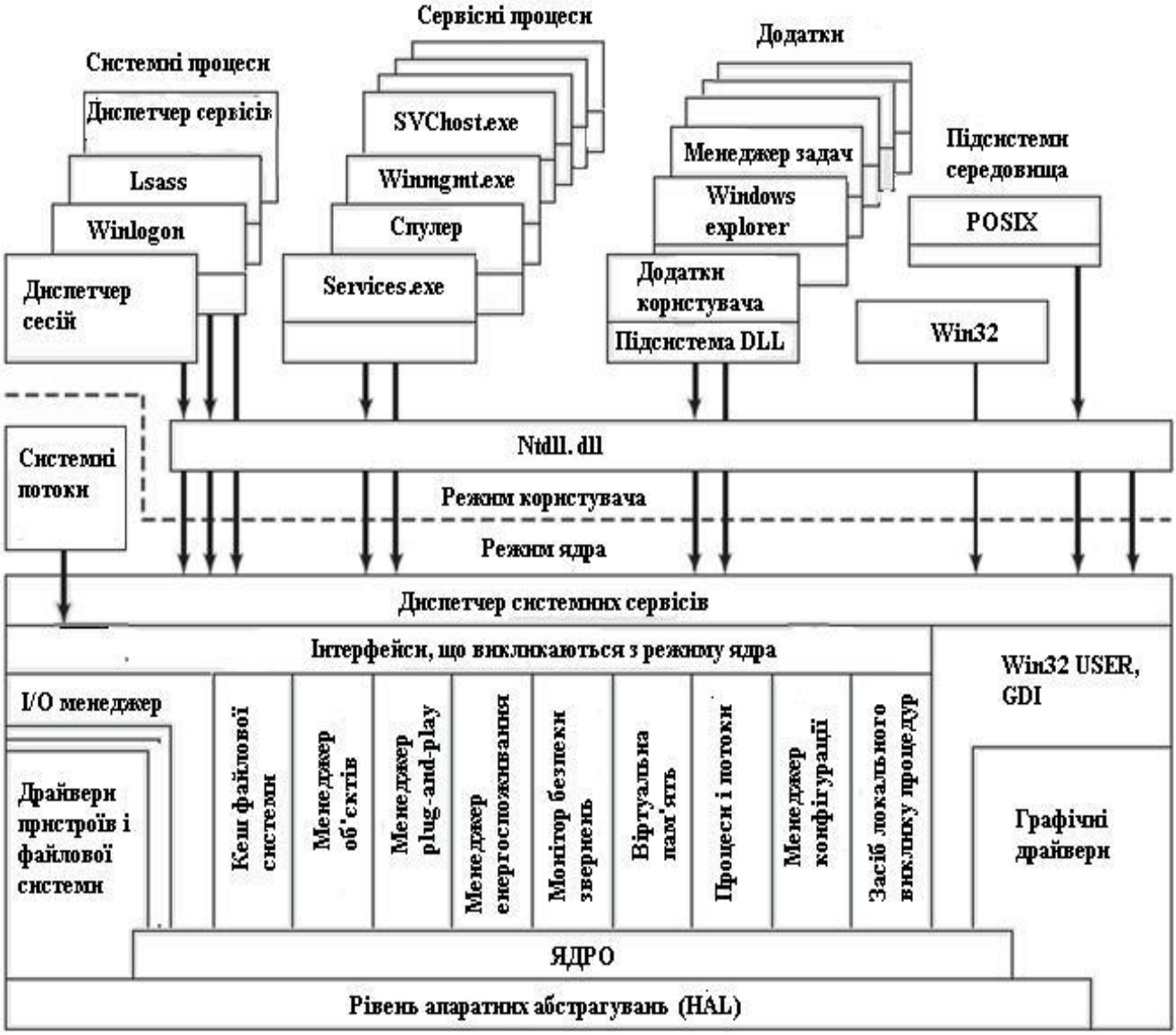


Рисунок 1.8 – Архітектура Windows

Всі версії Windows основані на NT версії і мають в головному схожу структуру на цьому рівні деталізації. У Windows відділено програмне забезпечення, що орієнтоване на користувача ОС від програмного забезпечення ядра ОС. Останнє, яке включає виконавчу систему (Windows Executive), ядро, драйвери пристроїв, рівень абстрагування від устаткування (HAL, hardware abstraction layer), запускається в режимі ядра. Програмне забезпечення режиму

Stallings, William. Operating systems: internals and design principles / William Stallings. – 7th ed. Prentice Hall, New Jersey, 2012. p.769.
ISBN-13:978-0-13-230998-1

ядра ОС має доступ до системних даних і до устаткування комп’ютера. Все інше програмне забезпечення, яке запускається в режимі користувача, має обмежений доступ до системних даних.

Організація ОС Windows має високорозвинену модульну архітектуру, якій кожна системна функція є керованою тільки одним компонентом ОС, а вся інша частина ОС і всі додатки отримують доступ до цієї функції лише через дублюючий компонент, який використовує стандартні інтерфейси. Доступ до ключових системних даних може бути отримано лише через відповідні функції.

принципі, будь-який модуль може бути вилучено, модифіковано чи переміщено без перезапису всієї системи, або її стандартних програмних інтерфейсів додатків (API, Application Programming Interface).

Як бачимо із схеми на рисунку 1.8 до компонентів режиму ядра відносяться:

виконавча система, що складається з шару сервісів ОС, таких як управління пам’яттю, управління процесами і потоками, безпеки, системи введення/виведення;

рівень апаратних абстракцій, який відокремлює ОС від особливостей апаратної платформи і тому системна шина, контролер прямого доступу до пам’яті, контролер переривань, системні таймери і пам’ять з точки зору ядра виглядають однаково;

ядро Windows відповідає за розподіл ресурсів між процесами, їх перемикання і синхронізацію; основним завданням ядра є якомога ефективніше завантаження процесорів системи. Ядро постійно перебуває в пам'яті, послідовність виконання його інструкцій може порушити тільки переривання (під час виконання коду ядра багатозадачність не підтримується);

драйвери пристроїв можуть бути драйвери не тільки апаратних пристроїв, але і застосування, якому потрібні засоби, доступні в режимі ядра, і які завжди варто оформляти як драйвер;

віконна і графічна підсистеми відповідають за інтерфейс користувача – роботу з вікнами, елементами керування і графічним виведенням.

Виконавча система Windows (Windows Executive) - це набір компонентів, відповідальних за найважливіші служби ОС (керування пам'яттю, процесами і потоками, вводом-виводом тощо).

Компонентами Windows Executive є передусім базові засоби підтримки. Ці засоби використовують у всій системі.

Менеджер об'єктів – відповідає за розподіл ресурсів у системі, підтримуючи їхнє універсальне подання через об'єкти.

Засіб локального виклику процедур (ALPC, advanced local procedure call) – забезпечує механізм зв'язку між процесами і підсистемами на одному комп'ютері.

Менеджер процесів і потоків – створює та завершує процеси і потоки, також розподіляє для них ресурси.

Менеджер віртуальної пам'яті – реалізує керування пам'яттю в системі, насамперед підтримку віртуальної пам'яті.

Менеджер введення/виведення – керує периферійними пристроями, надаючи іншим компонентам апаратно-незалежні засоби введення/виведення. Цей менеджер реалізує єдиний інтерфейс для драйверів пристроїв.

Менеджер кеша – керує кешуванням для системи введення/виведення. Часто використовувані блоки диска тимчасово зберігаються в пам'яті, наступні операції введення/виведення звертаються до цієї пам'яті, внаслідок чого підвищується продуктивність.

Менеджер конфігурації – відповідає за підтримку роботи із системним реєстром (registry) – ієрархічно організованим сховищем інформації про налаштування системи і прикладних програм.

Довідковий монітор захисту – забезпечує політику безпеки на ізольованому комп'ютері, тобто захищає системні ресурси.

- *Менеджер plug-and-play* – менеджер підключення, відключення, налаштування пристроїв, які підтримують технологію Plug-and-Play.

1.5.2 Особливості сучасних архітектур ОС UNIX і Linux

Архітектура ОС UNIX. Сучасна ОС UNIX має порівняно просту архітектуру. В ядрі знаходиться найбільш вживані функції, що забезпечують роботу ОС і користувача. Ядро в свою чергу взаємодіє з прикладними програмами за допомогою системних викликів. Сучасна структура ядра UNIX може бути представлена, як зображено на рисунку 1.9. В архітектурі сучасних

UNIX-систем виділено окремий менеджер пам'яті, відповідальний за підтримку віртуальної пам'яті.

Рисунок 1.9 – Ядро сучасної системи UNIX

Стандартом для реалізації інтерфейсу файлової системи є віртуальна файлова система, що абстрагує цей інтерфейс і дає змогу організувати підтримку різних типів файлових систем. У цих системах підтримується багатопроцесорна обробка, а також багатопотоковість. Базові архітектурні рішення, такі як доступ до всіх пристроїв введення/виведення через інтерфейс файлової системи або організація системних викликів, залишаються незмінними в усіх реалізаціях UNIX.

Архітектура Linux. Зауважимо, що архітектура Linux організована за технологією монолітного ядра, тобто структура даних ядра і код знаходяться в єдиному адресному просторі.

На рисунку 1.10 показані головні компоненти ядра Linux, які впроваджені IA-64 архітектуру (тобто, Intel Itanium). Прямокутники з хвилястими лініями, що у верхній частині схеми - це окремі процеси з потоками в них.

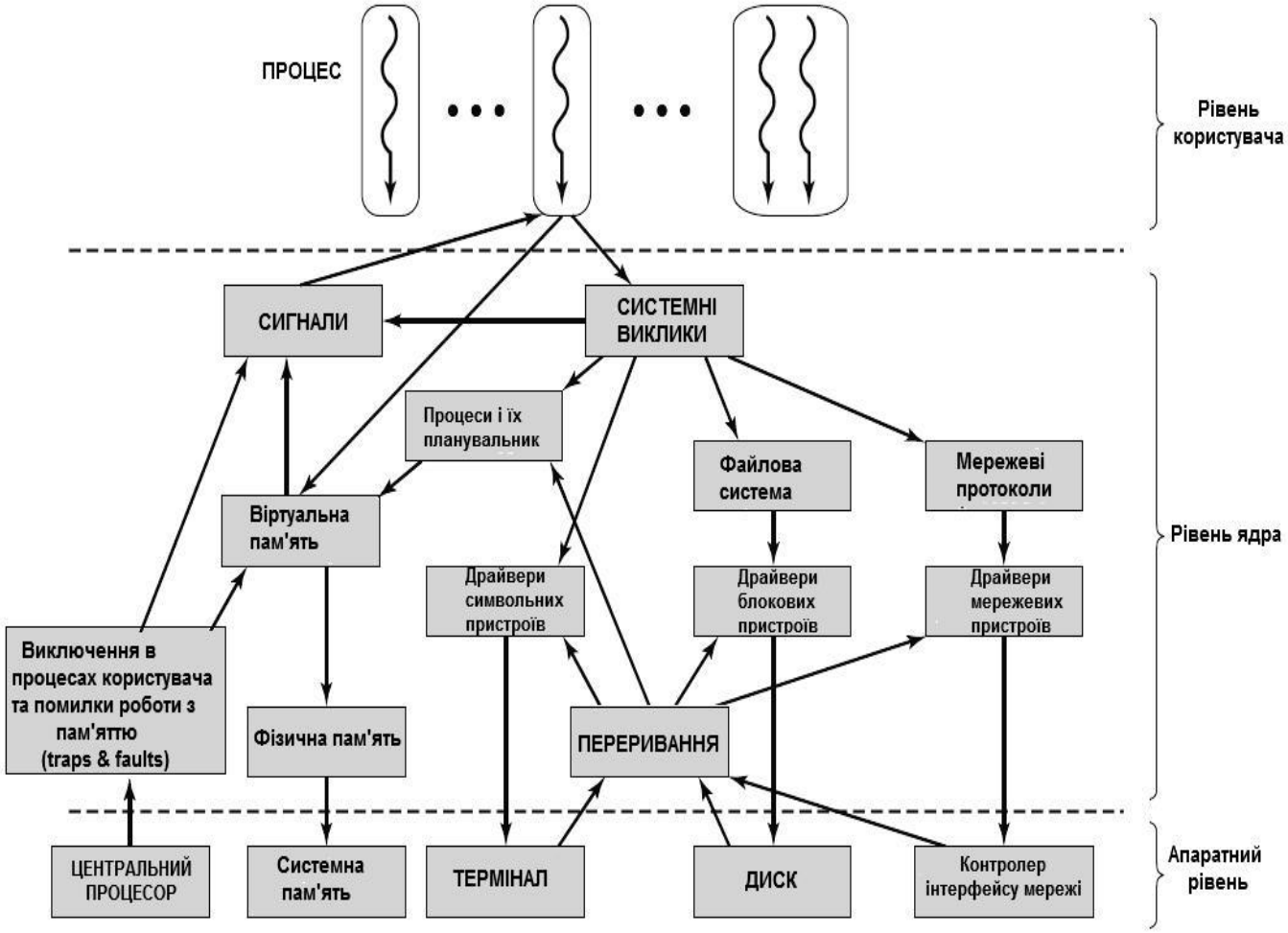


Рисунок 1.10 – Головні компоненти ядра Linux

Саме ядро складається з набору взаємодіючих компонентів між якими показано основні взаємодії. Нижче розміщена апаратна частина, що представлена як набір компонентів. Стрілками показано, які компоненти ядра використовують або керують компонентами апаратури. Всі компоненти ядра виконуються процесором, але це на схемі не показано.

Основні функціональні компоненти ядра:

Менеджер пам'яті – організація адресного простору і віртуальної пам'яті.

Планувальник процесів – реалізує підтримку багатозадачності в операційній системі: робота з таймером, створення процесів, завершення процесів і ін.

Міжпроцесова взаємодія – механізми, що забезпечують обмін даними між процесами.

Віртуальна файлова система – інтерфейс взаємодії з різними файловими системами та пристроями введення/виведення. Забезпечує доступ до д*райверів пристроїв, що* забезпечують безпосередню роботу з периферійними пристроями.

Мережний інтерфейс – забезпечує доступ до реалізації мережних протоколів і драйверів мережних пристроїв.

В Linux підтримується так звана концепція *модулів ядра*, тобто комплекс незалежних блоків. Ці блоки володіють, по-перше, *динамічним зв'язуванням*, при якому на вимогу завантажуються у пам'ять і вивантажуються з неї окремі секції коду. Такі секції (модулі) виконуються у привілейованому режимі. І, по-друге, с*тековою організацією,* при якій модулі організовані у вигляді певної ієрархічної структури. Окремі модулі виконують роль бібліотек при зверненні до них модулів більш високих рівнів в межах цієї структури, і вони самі можуть звертатися до модулів на більш низьких рівнях.

Для підтримки модулів у Linux використовують певні засоби: засоби, що завантажують модулі у пам'ять із здійсненням обміну даними між модулями та іншою частиною ядра; реєстрація драйверів для попередження іншої частини ядра про те, що новий драйвер став доступним; резервування апаратних ресурсів для захисту драйверів пристроїв від випадкового використання їх іншими драйверами.

1.6 Операційна система Android

Операційна система Android – це система на основі Linux, спочатку розроблена для мобільних телефонів. Це найпопулярніша мобільна ОС з великим відривом: телефони Android перевершують iPhone від Apple в світі приблизно в 4 рази [Morra, J. "Google Rolls Out New Version of Android Operating System."Electronic Design, August 24, 2016.]. Але це тільки один з елементів зростаючого домінування Android. Все частіше ця ОС стоїть практично за будь-яким пристроєм з комп'ютерним чіпом, крім серверів і ПК. Android – широко використовувана ОС для Інтернету речей.

Первісна розробка ОС Android була здійснена компанією Android, Inc., яка була куплена Google в 2005 році. Перша комерційна версія, Android 1.0,

була випущена в 2008 році. На момент написання цієї статті самою останньою версією є Android 7.0 (Nougat). Android має активне співтовариство розробників ентузіастів, які використовують вихідний код Android Open Source Project (AOSP) для розробки і поширення власних модифікованих версій операційної системи. Відкритий характер Android був ключем до його успіху.

1.6.1 Архітектура програмного забезпечення Android

Android визначається як програмний стек, який включає модифіковану версію ядра Linux, проміжне програмне забезпечення і ключові програми. На рисунку 1.11 подано архітектуру програмного забезпечення Android в деяких деталях. Таким чином, Android слід розглядати як повний програмний стек, а не просто як ОС.

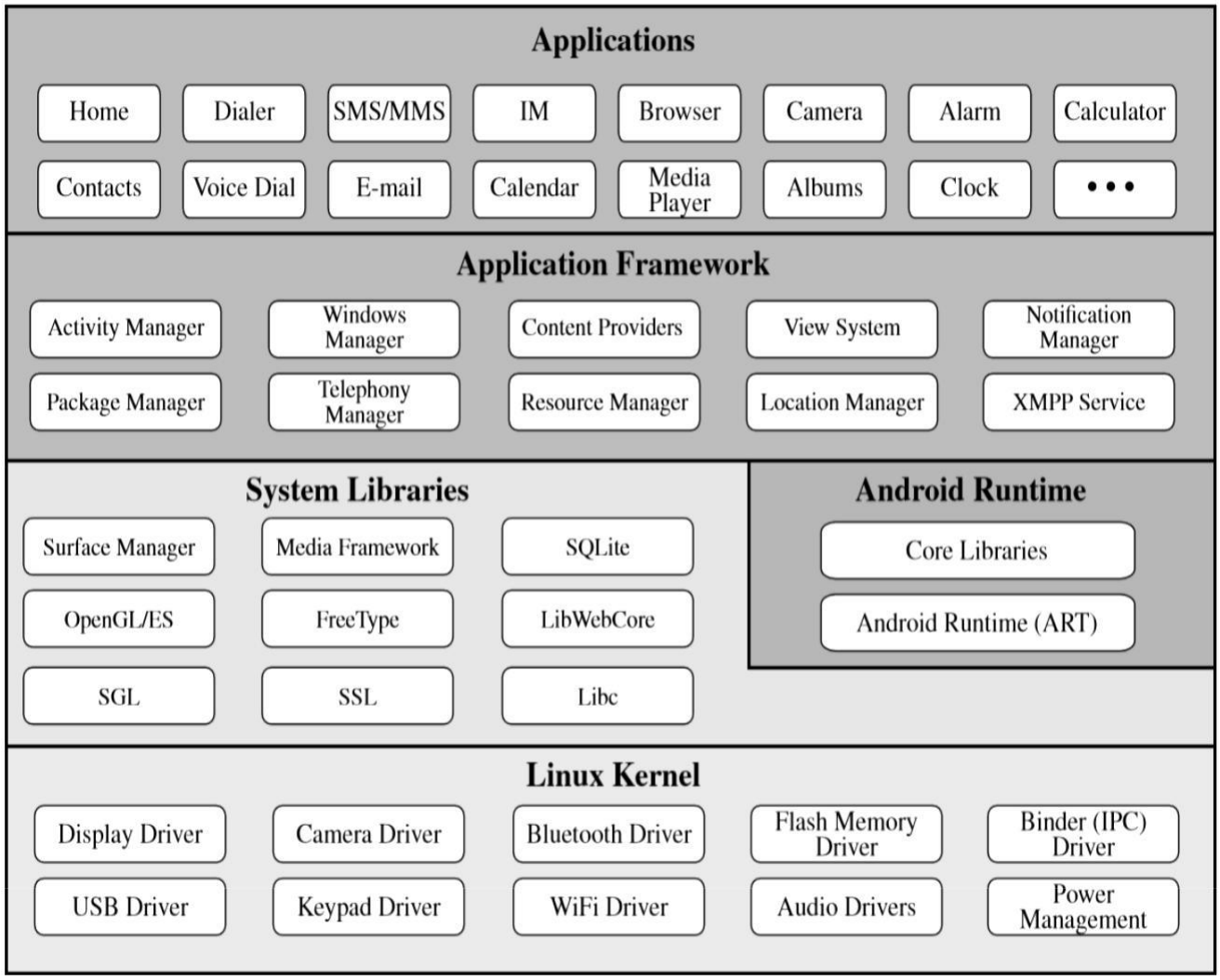


Рисунок 1.11 – Архітектура програмного забезпечення Android

Пояснення до рисунку 1.11:

Applications – рівень додатків;

Applications Framework – рівень спеціалізованих бібліотек для додатків; System Libraries – системні бібліотекі;

Android Runtime – середовище виконання Android-додатків з їх компіляцією під час установки;

Linux Kernel – ядро ОС Linux.

Додатки. Всі додатки, з якими користувач взаємодіє безпосередньо, частиною рівня додатків. Сюди входить основний набір додатків загального призначення, таких як поштовий клієнт, програма SMS, календар, карти, браузер, контакти та інші додатки, які зазвичай використовуються на будь-якому мобільному пристрої. Додатки зазвичай реалізуються на Java. Основна мета архітектури Android з відкритим вихідним кодом полягає в тому, щоб полегшити розробникам реалізацію нових додатків для конкретних пристроїв і конкретних вимог кінцевого користувача. Використання Java дозволяє розробникам позбутися специфічних для апаратного забезпечення міркувань і особливостей, а також використовувати більш широкі можливості мови Java, такі як зумовлені класи. Приклади типів базових додатків на платформі Android: Google Keep – нотатки та списки; Facebook, WhatsApp Messenger, Instagram; Google Новини; Microsoft Word: правка документів та загальний доступ; Google Документи та ін.

1.7 Основні етапи розвитку операційних систем

До середини 1950-х років перші ЕОМ ще не були забезпечені операційними системами.

На початку 1960-х вони лише комплектувались набором інструментів для розробки, планування та виконання завдань і до кінця 1960-х було розроблено цілий ряд операційних систем. Найбільш розвинуті ОС того часу, такі як «OS/360» (компанія «IBM»), «SCOPE» (компанія «CDC») та завершений вже в 1970-х роках «MULTICS» (MTI та компанія «Bell Labs»), передбачали можливість використання багатопроцесорних систем.

Спонтанний характер розробки ОС призвів до наростання кризових явищ, пов'язаних, перш за все, зі складністю та великими розмірами розроблюваних

систем. ОС погано масштабувались (простіші не використовували всіх можливостей потужних обчислювальних машин; складніші неоптимально виконувались або взагалі не виконувались на менш потужних системах) і були повністю несумісними між собою.

1969 році співробітники МТІ Кен Томпсон, Деніс Рітчі та Браян Керніган з колегами розробили та реалізували ОС «Юнікс» («Unix»; первинно «UNICS», на противагу «MULTICS»). Нова ОС увібрала в себе багато рис попередниць, але на противагу їм мала цілий ряд переваг:

- проста метафорика (два ключових поняття - процес та файл);
- компонентна архітектура (принцип «одна програма - одна функція», або інакше «кожна програма має робити лише одну роботу, але робити її добре» плюс потужні засоби об'єднання цих програм для розв'язання конкретних задач);
  мінімізація ядра та кількості системних викликів;
  незалежність від апаратної архітектури і реалізація на машино незалежній мові програмування (для цього була розроблена мова програмування «C»);
  уніфікація файлів (будь-що у системі є файлом, до якого можна отримати доступ за спільними для всіх правилами).

Завдяки зручності перш за все в якості інструментального середовища Unix дуже тепло зустріли в університетах, а потім і в галузі. В цілому і незабаром вона стала прототипом єдиної ОС, котру можна було використовувати у найрізноманітніших обчислювальних системах, і - більше того - швидко та з мінімумом зусиль перенести на іншу апаратну архітектуру.

кінці 1970-х років співробітники Каліфорнійського університету в Берклі внесли ряд суттєвих вдосконалень у джерельні коди Unix, включно з реалізацією стеку мережевих протоколів TCP/IP. Їх розробка стала відомою під іменем BSD (англ. *Berkeley Software Distribution*).

Через конфлікт з «Bell Labs» Річард Столмен поставив задачу реалізувати повністю незалежну від авторських прав ОС на основі Unix, заснувавши проект «GNU» (англ. *рекурсивсне скорочення «GNU's Not Unix»* – «ГНЮ Не Юнікс»).

Незабаром Unix стала стандартом де-факто, а потім і юридичним - ISO/IEC 9945. ОС, що дотримувались цього стандарту чи опираються на нього, називають «відкритими» або «стандартними». До них належать системи, що

базуються на останній версії Unix, випущеної «Bell Labs» («System V»), на розробках Університету Берклі («FreeBSD», «OpenBSD», «NetBSD»), а також ОС «Linux», розроблена спільнотою на чолі з Лінусом Торвальдсом та в межах проекту «GNU» (основні системні інструменти).

Нижче наведені ***основні етапи в розвитку операційних систем.***

Прості пакетні ОС. При роботі з такими ЕОМ завдання на машинних носіях збирали в пакети і поміщали в пристрій вводу даних. Після цього ОС, або монітор, зчитував з пристрою введення/виведення ці завдання по одному і передавала управління наступному зчитаному завданню. По завершенні завдання або при виникненні помилки управління поверталося монітору, який починав зчитувати наступне завдання. Приклади пакетних ОС: FMS (Fortran Monitor System), IBSYS (ОС, створена фірмою IBM для ЕОМ IBM 7094).

Багатозадачні пакетні ОС. Для підвищення ефективності використання процесора необхідно було реалізувати принцип багатозадачності (наприклад, поки одне завдання чекає завершення виконання операції введення/виведення, інше може використовувати процесор). Це можливо, якщо

ОП є достатньо місця і для самої ОС, і для двох (або більше) програм користувача. Для багатозадачних ОС обов'язковим є

наявність контролерів зовнішніх пристроїв, що працюють незалежно від процесора і дозволяють виконувати операції введення/виведення одночасно з командами процесора;

наявність системи переривань, за допомогою яких такі контролери взаємодіють з процесором.

При роботі багатозадачних ОС виникає необхідність в управлінню пам'яттю, в плануванні щодо виконання задач, в диспетчеризації процесів, в захисті завдань від впливу один одного. Приклад багатозадачної пакетної ОС: OS/360 фірми IBM.

ОС з розподілом часу. У цьому випадку декілька користувачів одночасно отримують доступ до системи за допомогою терміналів (термінал - робоче місце користувача: дисплей і клавіатура), а ОС забезпечує почергове виконання програм кожного користувача через малі проміжки часу. Таким чином, кожному користувачеві надається частина процесорного часу ЕОМ, не враховуючи витрат часу на роботу ОС. Приклади ОС з разподілом часу: CTSS (Compatible Time-Sharing System), MULTICS.

Контрольні питання

Що таке операційна система? Які основні функції ОС?

На якому рівні ієрархічної структури програмно-апаратних засобів комп'ютера реалізується виконання машинних команд?

Назвіть сервіси, що надає ОС?

Яким чином можна класифікувати ОС?

Як класифікуються ОС за призначенням? Відповідь деталізуйте.

Які основні складові ОС?

Що являє собою ядро ОС?

Що визначає структура ОС? Назвіть і опишіть основні типи структур ОС.

В яких режимах працюють сучасні ОС?

Що таке операційне і програмне середовище ОС?

Що таке *API*?

Що розуміють під архітектурою ОС? Назвіть універсальні підходи до розробки архітектури ОС?

Який основний недолік монолітної архітектури ОС?

На якій концепції заснована класична архітектура ОС?

Що являє собою багатошарова архітектура ОС?

Які особливості мікроядерної архітектури?

Назвіть компоненти режиму ядра ОС Windows?

Що називається виконавчою системою Windows (Windows Executive)?

Яку технологію ядра реалізує архітектура Linux?

Назвіть і опишіть основні етапи розвитку операційних систем?

Назвіть основні елементи архітектури програмного забезпечення Android.

РОЗДІЛ 2

ПРОЦЕСИ. ПОТОКИ. СИГНАЛИ[1]

2.1 Основні відомості про процеси. Управління процессами

Для розуміння структури операційних систем застосовують концепцію процесів. Цей термін уперше був застосований у 60-х роках минулого століття розробниками операційної системи Multics. Існує декілька визначень терміну «процес», наприклад

програма, що виконується;

екземпляр програми, що виконується на комп'ютері;

об'єкт, який можливо ідентифікувати і виконувати на процесорі;

одиниця активності, яка характеризується єдиним ланцюжком послідовних дій, поточним станом та зв'заним з ним набором системних ресурсів.

2.1.1 Властивості та класифікація процесів

Більшість сучасних операційних систем є мультипрограмними.[2] Переваги мультипрограмного режиму:

збільшення пропускної спроможності обчислювальної системи завдяки більш оптимальному завантаженню пристроїв;

можливість поєднувати тривалі операції (наприклад, вивід на принтер) з іншою корисною роботою;

нові засоби декомпозиції при розробці складних систем, що обробляють різні асинхронні події.

Оскільки мультипрограмність передбачає виконання декількох програм одночасно, то виникає необхідність керувати ними, розподіляючи між ними ресурси під час виконання і, як наслідок виникає таке поняття, як процес.

---

[1] При підготовці даного розділу були використані літературні джерела [1–4], [15–18], [20–22] та інші із списку літератури.

[2] Мультипрограмування – це такий спосіб організації обчислень, коли на однопроцесорній обчислювальній системі імітується одночасне виконання декількох програм.

Процес – це окрема програма в момент її виконання, а також виділені їй ресурси комп'ютера.

обчислювальних системах завжди виникають причини для створення нових процесів.

Ініціалізація системи. При запуску ОС відразу створюється кілька процесів, що забезпечують взаємодію з користувачем. Інші процеси є фоновими. Їх ще називають демонами (Disk And Execution MONitor).

Створення процесу іншим процесом, що працює. Створений процес дочірнім, а процес, що його створив, – батьківським.

Запит користувача на створення процесу. Процес завжди створюється однаково за допомогою особливого системного виклику для створення нового процесу.

В обчислювальних системах завжди є ситуації при яких процес може бути завершеним:

звичайне або успішне завершення процесу;

перевищення ліміту часу, що відведений для програми;

недостатній об'єм пам'яті;

зайве очікування (процес очікує настання певної події довше ніж задано в параметрах системи);

вихід із процесу при виникненні помилки, яку неможливо усунути;

поступила помилкова команда, або з недосяжними привілеями;

знищення процесу іншим процесом.

При виконанні програм процесором обчислювальної системи визначають наступні стани процесора:

Passive – процес завантаження в пам'ять, але його виконання ще неініційоване.

Ready – процес готовий до виконання (очікує ресурс центрального процесора).

Run – процес виконується на центральному процесорі.

Wait – процес очікує звільнення необхідного ресурсу.

В залежності від типу очікуваного ресурсу розрізняють системи:

з обмеженням по введення/виведення (в таких системах найбільший час очікування витрачається процесом при зверненні до підсистем введення/виведення);

з обмеженням по швидкодії (найбільший час витрачається на доступ до процесора).

Будь – який процес може знаходитись в одному з станів один раз в певний момент часу (рисунок 2.1).

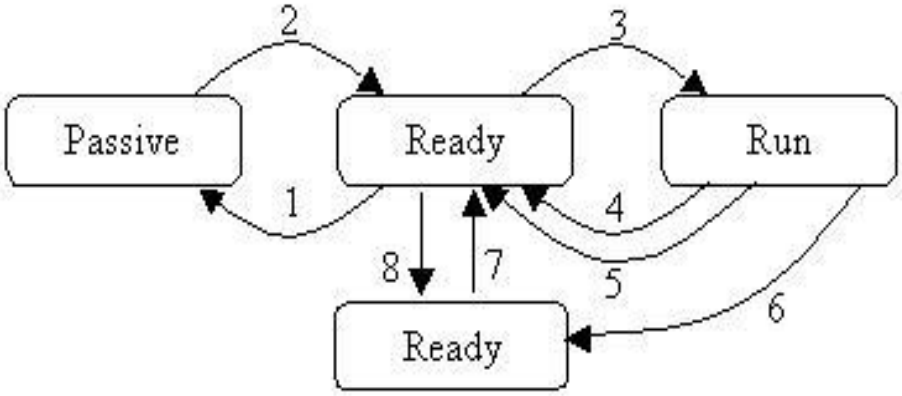


Рисунок 2.1 – Стани процесів

Визначають наступні умови зміни станів процесу:

2 – активізація процесу програмою – диспетчером або ОС;

3 – диспетчеризація[1] процесу;

4 – виникнення переривань;

5 – запит процесом певної системної функції в складі ОС (можливе створення певного процесу);

6 – запит процесом певної системної функції (вводу/виводу);

– завершення виконання системної функції;

– зупинка виконання процесу;

1 – завершення виконання процесу.

2.1.2 Моделі процесу

Модель з двома станами. Основним завданням ОС є управління виконанням процесів (визначення схеми чергування процесів та виділення їм ресурсів).

найпростішій моделі управління процесами в будь-який момент часу процес або виконується, або не виконується. На рисунку 2.2-а представлена

[1] Поняття диспетчеризації буде розглянуто в підрозділі «Планування і диспетчеризація процесів».

модель, процесу з двома станами, а на рисунку 2.2-б – діаграма використання черги.

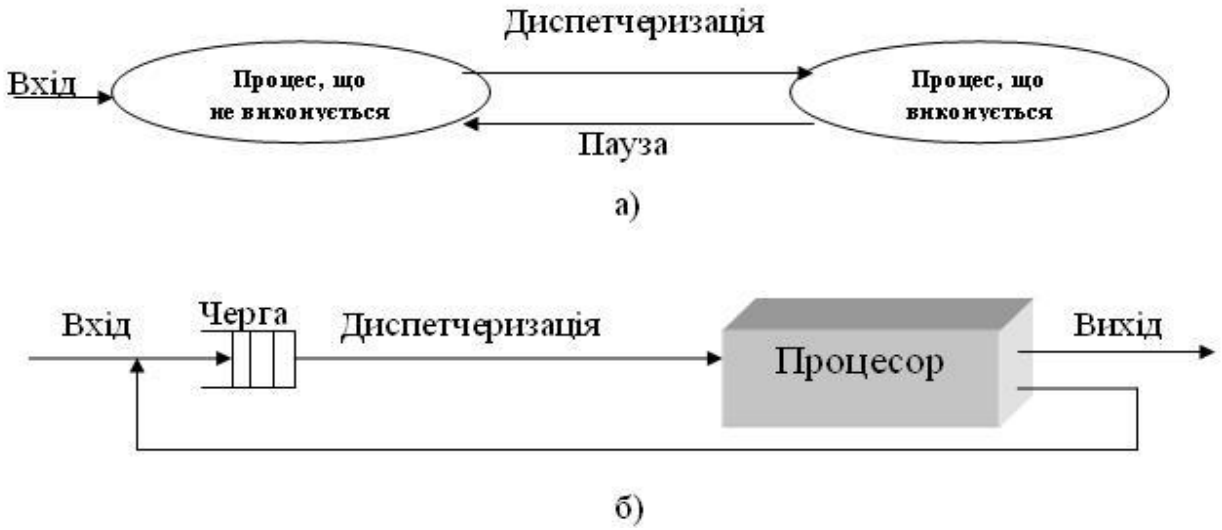


Рисунок 2.2 – Модель процесу з двома станами:
а) діаграма станів; б) використання черги

Створивши новий процес, ОС вводить його в систему в стані, що не виконується. Після цього створений процес чекає, коли він зможе бути запущений. Час від часу процеси, що виконуються будуть перериватися, а диспетчер буде вибирати інший процес для виконання .

Процеси, що не виконуються організовані в чергу (*пул*), де вони чекають свого виконання. Елементи такої черги - покажчики на процеси. Процес, робота якого була перервана, переходить в чергу процесів, що очікують виконання.

Якщо процес завершено, то він виводиться з системи. Далі для виконання диспетчер вибирає з черги наступний процес.

Модель з п'ятьма станами. В моделі з двома станами можливі дві некоректні ситуації, коли процес не виконується:

ОС вирішила на деякий час надати процесор іншому процесу;

процес блокований, так як з точки зору своєї внутрішньої логіки він не може виконуватися далі (наприклад, очікує вхідні дані).

На рисунок 2.3-а, представлені три стани процесу: 1) як такий, що виконується (використовує процесор); 2) готовий до виконання (тимчасово призупинений, щоб дозволити виконуватися іншому процесу); 3) блокований (процес не може бути запущений, поки не відбудеться деяка зовнішня подія). Тут також добавлено ще два стани: новий (тільки що створений процес, який

ще не поміщений операційною системою в чергу процесів, що виконуються і ще не завантажений в ОП) та завершений (процес, вже вилучений з пулу виконаних процесів).

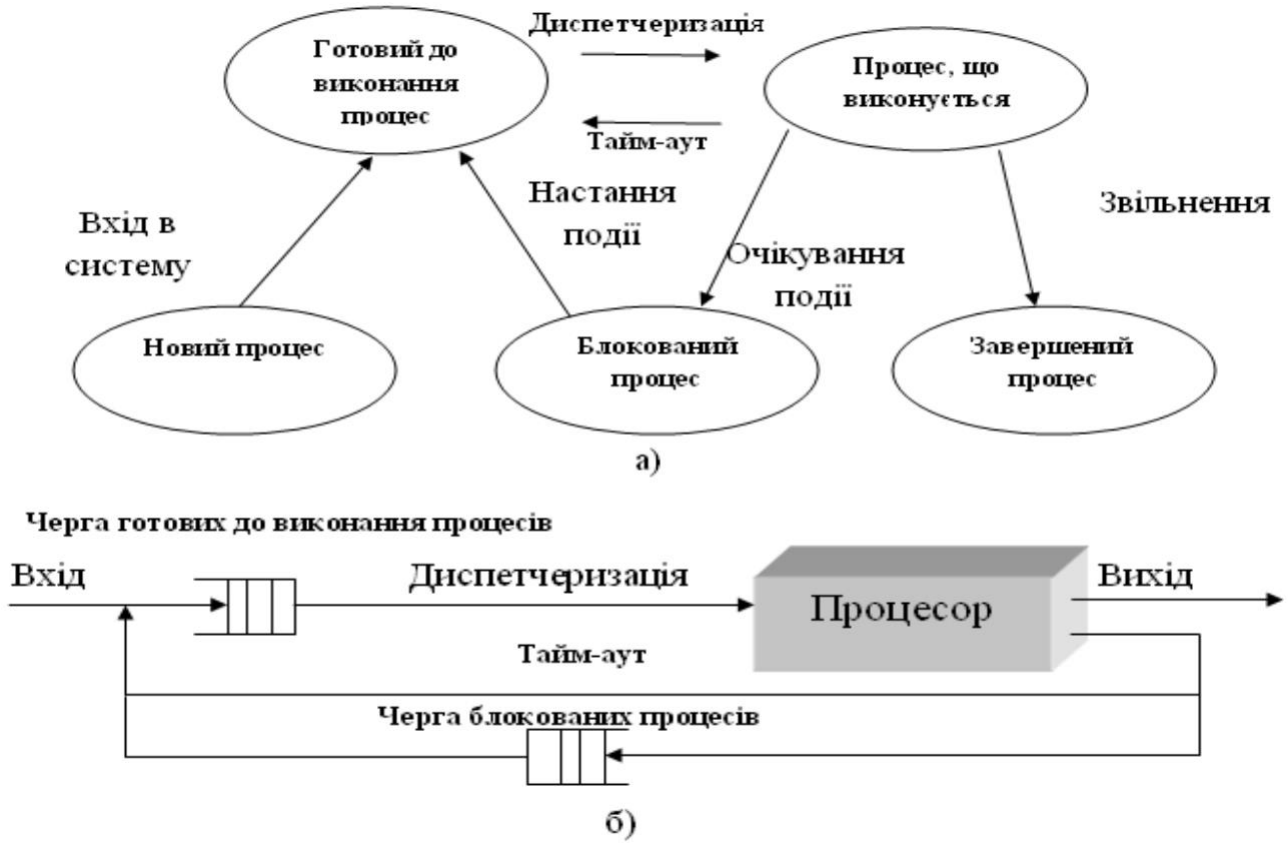


Рисунок 2.3 - Модель з п'ятьма станами:
а) діаграма станів; б) використання черги

Виходячи з діаграми станів, можливі наступні переходи між станами процесу:

нульовий стан → новий – причини переходу між цими станами збігаються з причинами створення процесів;

новий → готовий до виконання – цей перехід відбувається, коли ОС готова до обробки додаткового процесу, так як у більшості ОС є обмеження на кількість процесів або на обсяг виділеної для процесів віртуальної пам'яті;

готовий → виконується – цей перехід відбувається в момент вибору нового процесу для запуску;

виконується → завершений – причини переходу між цими станами збігаються з причинами завершення процесів;

виконується → готовий до виконання – цей перехід відбувається найчастіше після закінчення кванта часу, виділеного процесу. У деяких ОС може відбутися витіснення процесу і до закінчення цього кванта часу (наприклад, при появі процесу з більш високим пріоритетом). Витіснення (preemtion) - це ситуація, коли процес позбавляється деякого ресурсу до того, як він завершить роботу з ним;

виконується → блокований – цей перехід виконується, якщо для продовження виконання процесу потрібно дочекатися настання деякої події (звільнення занятого ресурсу, закінчення операції ввід-вивід, повідомлення від іншого процесу і т.д.);

блокований → готовий до виконання – цей перехід відбувається, коли настає очікувана процесом подія.

На рисунку 2.3-б показаний один із способів реалізації почергового виконання процесів. Маємо дві черги: 1 – для готових до виконання і 2 – для заблокованих процесів.

Кожен процес, який надходить в систему для обробки, поміщується в чергу готових до виконання процесів, і ОС вибирає з неї для виконання черговий процес. Якщо схема пріоритетів відсутня, то така черга має тип FIFO («First Input First Output» – «першим увійшов-першим вийшов» – «класичний» варіант черги, послідовність обробки елементів якої на виході та ж, що і послідовність поміщення цих елементів в чергу). Коли виконання процесу переривається, він, в залежності від обставин, може або завершитися, або потрапити в одну з двох згаданих вище черг. Якщо ж станеться якась подія, то всі процеси, що очікують його переміщаються з черги блокованих в чергу готових процесів.

Недоліком такої схеми є те, що після будь-якої події ОС повинна переглядати всю чергу блокованих процесів, відшукуючи серед них ті, які очікують саме цієї події. Тому ефективніше організувати кілька черг – свою для кожної події. Тоді при настанні якої-небудь події всі процеси з відповідної черги можуть бути відразу переведені в чергу готових до виконання процесів.

Якщо ж диспетчеризація здійснюється з використанням пріоритетів[1], то можна організувати кілька черг готових процесів, по одній черзі на кожен рівень пріоритету. У цьому випадку потрібно також передбачити механізм контролю виконання процесів з відповідних черг відповідно до їх пріоритету.

---

[1] Поняття пріоритету процесу буде викладене в підрозділі 2.1.5.

2.1.3 Опис процесів

Для керування процесами ОС створює і підтримує таблиці процесів, які містять покажчики на образ кожного процесу.

Образ процесу містить програму (код), дані, стек і атрибути процесу. Структура даних разом з атрибутами процесу часто називається його ***керуючим блоком*** (рисунку 2.4)**.**

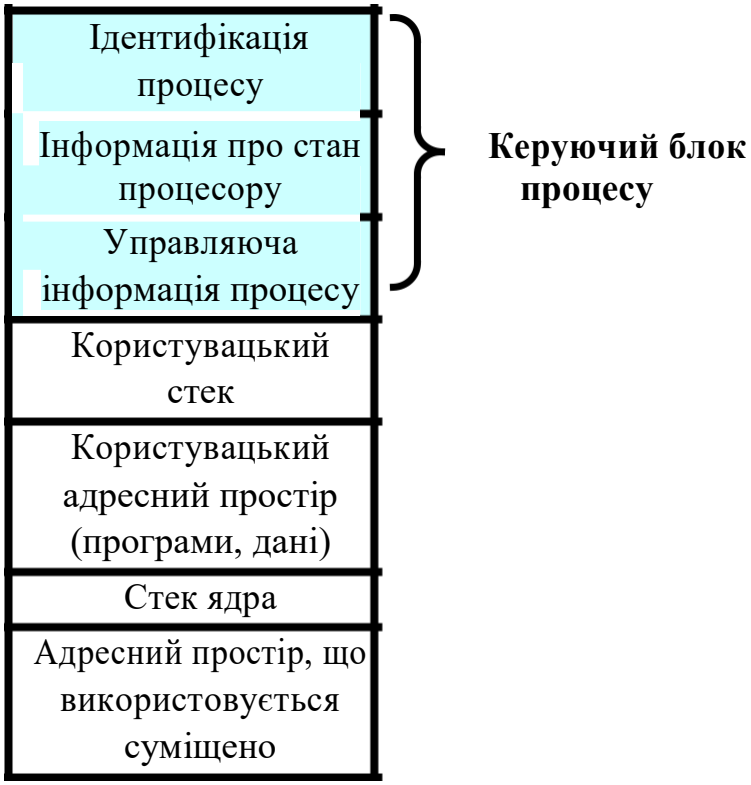


Рисунок 2.4 – Образ процесу

Атрибути процесу включають в себе наступне:

ідентифікатори (ідентифікатор даного процесу, ідентифікатор батьківського процесу, ідентифікатор користувача);

інформацію про стан процесу (вміст регістрів загального призначення, керуючих регістрів і регістрів стану, покажчики на стек);

керуючу інформацію про процес (стан процесу – виконується, готовий до виконання, блокований і т.д.; пріоритет; інформацію, пов'язану з плануванням; інформацію про події; інформацію про зв'язки з іншими процесами (в чергах, родинні зв'язки); відомості про обмін інформацією між процесами (різні прапори, сигнали, повідомлення); привілеї процесу; інформацію про володіння ресурсами та їх використання).

Процеси можна класифікувати наступним чином:

За приналежністю до центрального процесору (ЦП):

внутрішні (в операційних системах прийнято розрізняти не тільки час існування процесу, але й час його народження. Такою точкою відліку прийнято вважати ЦП. Процеси виконані на ньому називаються програмним або внутрішніми);

зовнішні (це процеси, розвиток яких проходить під контролем не ЦП, а інших).

За приналежністю до ОС:

системні;

користувацькі.

За генеалогічним типом:

породжуючі (це процеси, які можуть породжувати існування інших процесів);

породжені (процеси, які починають існувати в результаті існування іншого процесу).

За результативністю:

еквівалентні (два процеси, які мають один і той же результат обробки даних в одній і тій же програмі на одному і тому ж, або на різних процесорах);

тотожні (якщо в кожному з еквівалентних процесів обробка даних проходить в одній і тій же програмі, але траси не співпадають);

рівні (при співпаданні слідів тотожних процесів).

За динамічним принципом:

- *послідовні* (якщо інтервали двох процесів не пересікаються в часі);
- *паралельні* (якщо на певному інтервалі часу існують одночасно два процеси);

За зв’язаністю:

взаємозв’язані (якщо між ними створюються зв’язки за допомогою системи управління процесів);

ізольовані;

інформаційно-незалежні (якщо два взаємозв’язані процеси використовують одні і ті ж ресурси, але не обмінюються між собою інформацією);

взаємодіючі (якщо між двома процесами є інформаційні зв’язки).

2.1.4 Управління процесами

Створення процесів. При створенні нового процесу ОС повинна виконати наступні дії:

присвоїти новому процесу унікальний ідентифікатор, при цьому в таблицю процесів вноситься новий запис;

виділити простір для процесу – для образу процесу і для його керуючого блоку;

ініціалізувати керуючий блок процесу;

встановити необхідні зв'язки – наприклад, включити новий процес в чергу готових процесів;

створити або розширити інші структури даних.

Перемикання процесів може відбутися в будь-який момент, коли управління передається операційній системі: у разі апаратного переривання, в тому числі переривання від таймера, спрацьовування пастки (виключення), при системному виклику. Всі перераховані причини реалізуються *через механізм переривань.*

Дії ОС при перемиканні процесів:

збереження контексту процесора (вмісту регістрів загального призначення і керуючих регістрів) в керуючому блоці процесу;

оновлення керуючого блоку виконується в даний момент процесу – потрібно змінити стан цього процесу і зберегти інформацію з обліку використання ним ресурсів;

переміщення керуючого блоку даного процесу у відповідну чергу (готових, заблокованих процесів та ін);

вибір необхідного процесу для виконання;

оновлення керуючого блоку обраного процесу - необхідно встановити для нього стан виконання;

оновлення структур даних по управлінню пам'яттю;

відновлення контексту процесора для вибраного процесу.

Вважається, що ОС теж є програмою, яка виконується процесором, тобто виникає питання: чи є ОС процесом? Якщо так, то має здійснюватися певним чином управління таким процесом. При розв’язанні цієї проблеми виникли різні підходи до розробки ОС:

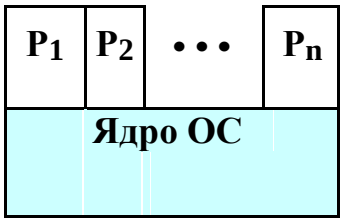
автономне ядро;

ядро в складі користувацьких процесів;

ОС на основі процесів.

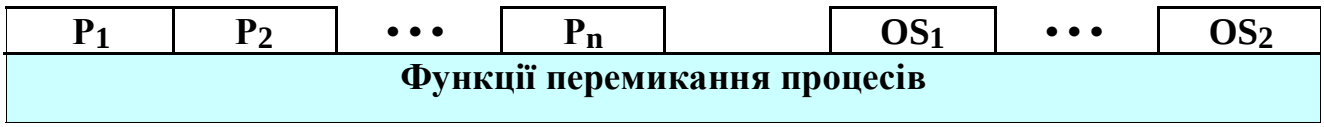
Ядро поза процесами (автономне ядро) – використовується в багатьох ранніх ОС. Концепція процесу тут застосовується тільки до користувацьких програмам, тоді як код ОС виконується як окремий об'єкт, що працює в привілейованому режимі (рисунок 2.5-а).

Ядро в складі користувацьких процесів – використовується в невеликих ЕОМ (в ПК і робочих станціях). Майже всі програми такої ОС виконуються в контексті користувача процесу (рисунок 2.5-б).



а) Автономне ядро

б) Функції операційної системи, що виконуються в користувацьких системах



в) Функції операційної системи, що виконуються як окремі процеси.

Рисунок 2.5 – Виконання коду ОС по відношенню до процесів: а) ядро поза процесами; б) ядро в складі користувацьких процесів; в) ОС на основі процесів

При виклику системних процедур, що працюють в режимі ядра, і поверненні з них використовується окремий стек ядра. Код та дані ОС знаходяться в спільно використовуваному адресному просторі і доступні для використання всіма користувацькими процесами. Тому при перериванні, включаючи системні виклики, перемикання процесів не відбувається –

перемикається тільки режим роботи процесора (користувацький ←→ ядро) в рамках одного і того ж процесу. При цьому в рамках одного і того ж процесу можуть виконуватися і користувацькі програми, і програми ОС. Програми ОС, виконувані в різних користувацьких процесах, є ідентичними.

ОС на основі процесів. На рисунку 2.5-в показана структура ОС на основі процесів.

Переваги такого підходу:

він сприяє розробці модульних ОС з простими міжмодульними інтерфейсами;

деякі другорядні функції ОС зручно реалізувати у вигляді окремих процесів; ці функції може викликати тільки ОС, і такі процеси можуть чергуватися з користувацькими;

в багатопроцесорних і багатокомп'ютерних обчислювальних системах окремі служби ОС можуть виконуватися на різних процесорах, що, відповідно, підвищує продуктивність системи.

2.1.5 Стратегії планування і диспетчеризація процесів

Планування – це системний процес, який здійснює установку користувацьких процесів в чергу та визначення атрибутів їх виконання в рамках використовуваної обчислювальної системи.

загальному випадку стратегія планування визначає які процеси повинні бути заплановані на виконання з метою досягнення результатів вирішуваної задачі.

Стратегії можуть бути наступними:

• закінчувати обчислення в тому самому порядку, в якому вони були розпочаті;

• перевагу надавати більш коротким процесам;

• надавати всім процесам однакові послуги, в тому числі і однаковий час очікування.

Розрізняють три типи планування:

– *короткострокове планування* – **диспетчеризація –** рішення про додавання процесу в пул процесів, що виконуються;

середньострокове планування – рішення про додавання процесу до числа процесів, що повністю або частково розміщені в оперативній пам'яті;

довгострокове планування – рішення про те, який з доступних процесів буде виконуватися процесором (полягає у виборі таких обчислювальних процесів, які менше всього б конкурували між собою в процесі досягнення мети обчислень).

Системний процес який здійснює планування може бути реалізований двома способами:

цілісний планувальник – це програмний модуль, який є частиною операційної системи, його робота ініціюється перериваннями і він виконується як окремий системний процес;

розподілений планувальник – програмний модуль планувальника або його частина записується в кожний процес при його завантаженні в оперативну пам'ять.

У загальному випадку планувальник здійснює наступні функції:

1) запис копії процесу в пам'ять обчислювальної системи;

2) аналіз атрибутів при виконанні обчислювального процесу та формування набору робочих атрибутів в залежності від характеристик обчислювальної системи;

3) запис процесу в чергу процесів у випадку, якщо є вільне місце в черзі, або перевід процесу в пасивний стан у випадку, якщо всі елементи черги зайняті; 4) періодичний перегляд черги процесів, запис або знищення процесів в черзі у відповідності з часом функціонування процесів.

Відома велика кількість правил (дисциплін диспетчеризації), у відповідності до яких формується список (черга) готових до виконання задач.

Розрізняють два великих класи дисциплін диспетчеризації: безпріоритетні і пріоритетні.

При безпріоритетній організації задачі або процеси вибираються згідно з певним, раніше установленим, порядком без врахування важливості цих процесів та необхідного часу обслуговування.

При реалізації пріоритетних дисциплін певним окремим задачам чи процесам надаються виключне право або набір прав для установки такого процесу в чергу.

На рисунку 2.6 представлена структура дисциплін диспетчеризації.

Далі опишемо принципи роботи деяких дисциплін диспетчеризації:

Дисципліна FIFO (first input first output) – є реалізацією безпріоритетної черги. Якщо в черзі звільняється елемент і він не останній, то всі після нього посуваються вперед і в останній вільний елемент черги записується ім'я нового процесу. Першим з черги вибирається елемент, який стоїть на початку черги.

Дисципліна LIFO (last input first output) – першим з черги вибирається елемент, який прийшов останнім.

Рисунок 2.6 – Структура диспетчеризації

Дисципліна FCFS (first come first server) – є подібна до FIFO. Ця дисципліна враховує перебування процесів в станах блокування, наприклад при операціях вводу/виводу. Ті, які були заблоковані в процесі роботи після переходу в стан готовності поступають в чергу готовності перед новими задачами, які ще не обслуговувались. При такій дисципліні організовуються дві черги: черга готових до обслуговування процесів і черга нових процесів (рисунок 2.7).

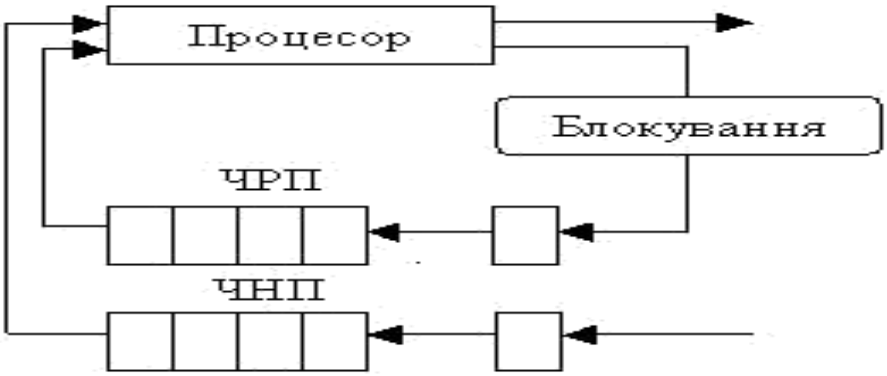


Рисунок 2.7 – Дисципліна FCFS (ЧНП – черга нових процесів; ЧРП – черга робочих процесів)

Ця дисципліна не потребує втручання ззовні в хід обчислювального процесу. Не відбувається розподіл процесорного часу.

Переваги: простота реалізації та невеликий розхід системних ресурсів для організації черги задачі. Однак при збільшенні завантаженості ОС, збільшується середній час обслуговування задачі, коли «короткі» завдання (які потребують невеликих затрат машинного часу) повинні очікувати такий самий час, що й довгі завдання.

Дисципліна SJN (short jump next) – згідно з цією дисципліною ОС вимагає, щоб для кожного процесу була відома оцінка необхідних обчислювальних ресурсів. Для постановки задачі в чергу диспетчер оцінює необхідний час виконання задачі і ставить задачу перед довшою задачею. При цій дисципліні існує одна черга процесів і завдання, що були заблоковані знову поступають в кінець черги як і нові завдання. Це приводить до того, що завдання які потребують мало часу очікують процесу як і довгі процеси.

Дисципліна SRT (short remain time) – розроблена з метою забезпечення більш якісного обслуговування коротких завдань. Виконує спочатку ті процеси, час завершення виконання яких найменший.

Дисципліна RR (round robin) – кожна задача отримує порцію часу (квант часу). Після закінчення цього кванту часу задача знімається з виконання на процесорі і він передається наступній задачі. Задача, яка була знята з черги записується в кінець черги задач, які готові до виконання. Величина кванту часу, як правило, вибирається, як середнє значення між достатнім часом реакції системи на запити користувачів та процесорним часом, який необхідний для перемикання між задачами.

2.1.6 Алгоритми в диспетчеризації з витісненням та без

Диспетчеризація без розподілу процесорного часу (*багатозадачність без витіснення*) – це такий спосіб диспетчеризації, при якому активний процес виконується до тих пір, поки він за власною ініціативою не передасть управління диспетчеру задач для вибору іншого, готового до виконання процесу. До них відносяться дисципліни: FCFS, SJN, SRT.

Диспетчеризація з розподілом процесорного часу (*багатозадачність з витісненням*) – це такий спосіб, де рішення про перемикання з однієї задачі на іншу приймається диспетчером задач, а не власною активною задачею. Процес, що виконується може бути перерваним і переведений операційною системою в стан готовності до виконання. Рішення про витіснення може прийматися при запуску нового процесу про перериванню або періодично, згідно переривань від таймеру. До таких дисциплін відносяться RR та інші, реалізовані на її основі.

Будь-які процеси мають бути гарантовано певним чином обслужені операційною системою. Гарантоване обслуговування може бути досягнуто трьома способами:

ОС виділяє мінімальну частину процесорного часу деякому класу процесів, у випадку, коли хоча б один з них готовий до виконання;

ОС виділяє мінімальну долю процесорного часу певному конкретному процесу якщо він готовий до виконання;

ОС виділяє стільки часу певному процесу, щоб він міг виконати своє обчислення в певний термін.

Для порівняння алгоритмів диспетчеризації можуть бути використані наступні критерії:

використання (завантаження) процесору – відсоток часу протягом якого процесор зайнятий;

пропускна здатність процесора – це кількість процесів, яка виконується процесором за одиницю часу;

час обороту – інтервал часу від моменту появи процесу у вхідній черзі до моменту завершення процесу. Цей час обороту включає в себе час очікування у вхідній черзі, час очікування у черзі готовності, час готовності у чергах до периферійних пристроїв, час виконання на процесорі та час вводу/виводу;

час очікування – це сумарний час знаходження процесу в черзі очікування готових процесів;

час відповіді – це час від моменту поступлення процесу на вхідну чергу до моменту першого звернення процесу на ввід/вивід даних.

Як відомо, при виконанні операційною системою багатьох процесів може відбуватися зниження продуктивності обчислювальної системи.

Існують певні методи підвищення продуктивності системи:

сумісне планування, при якому всі потоки однієї задачі одночасно вибираються для виконання процесором (у випадку наявності мультипроцесорної системи) і одночасно знімається з виконання. В цьому випадку зменшується час перемикання між задачами.

планування, при якому задачі, що знаходяться в критичній області не перериваються, а закінчують виконання критичної секції. Задачі, які чекають на виконання критичної секції коду не виконують її поки не завершиться переривання.

При виконанні програм, які реалізують функції контролю чи управління в системах реального часу може виникнути ситуація, коли одна або декілька задач не можуть бути вирішені протягом довгого проміжку часу через значну завантаженість обчислювальної системи. В цьому випадку втрати, які пов’язані невиконанням цих задач можуть бути більшими ніж втрати від невиконання задач з більш високим пріоритетом. Тому іноді виникає необхідність динамічної зміни пріоритету задачі в процесі її виконання. Це дозволяє реалізувати більш швидку реакцію на запити користувачів і гарантувати виконання будь-яких запитів.

Операційна система може змінити пріоритет задачі за допомогою:

Підвищення пріоритету активної задачі: при завантаженні задачі на виконання її пріоритет автоматично збільшується і при цьому знижується час реакції цієї активної задачі на дії користувачів в порівнянні з іншими фоновими задачами;

Підвищення пріоритету операції вводу/виводу: після завершення операції вводу/виводу задача отримує найвищий пріоритет в певному класі задач, що дозволяє більш швидко закінчити всі незавершені операції вводу/виводу;

Підвищення пріоритету «забутих» задач: якщо задача не отримує часу процесора протягом певного відрізку часу, то диспетчер задач тимчасово присвоює їй більш високий пріоритет, який не перевищує певної заданої межі, і таким чином можливо перемикатися на такі «забуті» задачі більш швидко. Після виконання такої задачі за певний квант часу її пріоритет знижується до попереднього значення.

2.2 Взаємодія процесів

2.2.1 Проблема взаємного виключення і способи її вирішення

Два процеси називаються *паралельними (concurrent)*, якщо їх виконання може перекриватися в часі, тобто, наприклад, другий процес починається раніше, ніж завершується перший. Два процеси називаються *послідовними (serial***),** якщо вони не є паралельними.

мультипроцесорній системі досягається *фізична паралельність.* Якщо виконання декількох процесів чергується в одному-єдиному процесорі, то досягається *логічна паралельність.*

Можна виділити три види взаємодії процесів:

• *обмін інформацією* – якщо процеси безпосередньо проінформовані про наявність один одного;

• *конкуренція за доступ до ресурсів* – якщо процеси не проінформовані про наявність один одного; ОС повинна регулювати такі звернення;

• *узгодження дій процесів* – якщо процеси не прямо обізнані один про одний (наприклад, процес А генерує дані, а процес В виводить їх на друк).

При цьому паралельні процеси взаємодіють за допомогою або *механізму поділюваних змінних*, або *механізму передачі повідомлень*.

При роботі паралельних процесів, як в однопроцесорних, так і в багатопроцесорних системах неможливо передбачити відносну швидкість їх роботи, і тому виникають проблеми при взаємодії таких процесів.

1. *Стан гонок* (змагань), коли два процеси використовують одну і ту ж глобальну змінну і обидва виконують читання і запис цієї змінної. Критичним при цьому виявляється порядок читання і запис цієї змінної різними процесами.

2. *Блокування та взаємоблокування при доступі до ресурсів*[1]. Процес А може зажадати і одержати контроль над деякими пристроєм

[1] Питання взаємоблокуваня винесено в окремий підрозділ 2.4. Взаємне блокування.

введення/виведення (ПВВ), після чого тимчасово призупинити роботу. При цьому небажано, щоб ОС блокувала даний ПВВ і не дозволяла іншим процесам використовувати його, так як це може призвести до взаємоблокування.

Програмні помилки, які важко виявити, так як результат роботи програми перестає бути детермінованим і відтворюваним.

Приклад стану гонок. Нехай у нас є процедура, що виконує читання символу з клавіатури і його відображення на екрані, яка знаходиться в пам'яті, що розділяється і використовується всіма процесами:

```
void echo ()
{
chin = getchar (); chout = chin; putchar (chout);
}
```

Розглянемо таку послідовність подій:

процес А виконує оператор chin = getchar () – введено символ «*x*»;

процес В виконує процедуру echo до кінця – введено і надруковано символ «*y*»;

процес А продовжує своє виконання, але при цьому буде виведений вже символ «*y*», а не «*x*», тобто «*y*» буде виведений двічі, а «*x*» – втрачений.

Така проблема не буде виникати, якщо в кожен момент часу тільки один процес буде входити в процедуру *echo*, і ця процедура обов'язково повинна бути повністю виконана процесом, що увійшов до неї до того, як стане можливим її виконання іншим процесом.

Та ж проблема залишається і в багатопроцесорній системі.

Для уникнення проблем типу гонок, блокувань та взаємоблокувань в концепції процесів існує *механізм взаємного виключення* (mutual exclusion).

Взаємне виключення – механізм, який гарантує, що в будь-який момент часу тільки один процес виконує деяку визначену послідовність дій, і тим самим виключає можливість роботи іншого процесу.

Критичний ресурс (КР) – ресурс, до якого в кожний момент часу можливий доступ тільки одного процесу. Доступ до такого ресурсу здійснюється в ***критичній секції (КС)*** – частини коду, яка в будь-який момент часу може виконуватися тільки одним процесом.

Але взаємне виключення може привести до двох проблем:

взаємного блокування (*deadlock*).

голодування (*starvation*).

Розглянемо як виникає проблема голодування. Нехай є процеси *A, B* і *C*, кожному з яких періодично потрібен доступ до ресурсу *P*. Нехай процес *A* використовує ресурс *P*, а процеси *B* і *C* – очікують. Потім процес *B* використовує ресурс *P*, потім знову процес *A* використовує ресурс *P* і т.д. Тобто, процес *C* може ніколи не отримати доступ до ресурсу *P*, незважаючи на те, що ніякого взаємного блокування немає. Така ситуація і називається *голодуванням*.

Отже, щоб уникнути гонок, ми повинні уникати знаходження двох процесів одночасно в критичних секціях. Але цієї вимоги ще недостатньо! Для правильної спільної роботи паралельних процесів необхідно виконання шести умов для досягнення взаємного виключення:

два процеси не повинні одночасно перебувати в критичних секціях;

у програмі не повинно бути закладено ніяких початкових припущень про швидкість або кількість процесорів;

процес, що перебуває поза критичною секцією, не повинен блокувати інший процес;

повинна бути неможлива ситуація, коли процес вічно чекає попадання критичну секцію (тобто не повинні з'являтися взаємоблокування і голодування);

коли в КС немає жодного процесу, будь-який процес, що запитав можливість входу в цю КС, повинен негайно отримати доступ до неї;

процес залишається в КС лише протягом обмеженого часу.

Існують такі підходи до досягнення взаємного виключення:

програмний – в цьому випадку сам процес є «відповідальним» за досягнення взаємного виключення, тобто процес (як системний, так і користувацький) повинен координувати свої дії з іншими процесами для досягнення взаємного виключення без підтримки з боку ОС або середовища програмування;

апаратний – з використанням спеціальних машинних команд;

примітиви синхронізації – семафори, монітори та інші подібні засоби.

2.2.2 Апаратні способи досягнення взаємного виключення

Існують такі апаратні способи досягнення взаємного виключення:

Заборона переривань.

Спеціальні команди, що виконуються атомарно.

При *забороні переривань* процесор перемикається на інший процес тільки по перериванню. Відповідно, процес забороняє переривання перед входом у критичну секцію і дозволяє переривання відразу після виходу з критичної секції. Взаємне виключення при цьому досягається, тому що процес не переривається всередині критичної секції, і тому інші процеси не можуть увійти в свої критичні секції.

такого підходу є певні недоліки: по-перше, він може бути застосований тільки в однопроцесорних системах, по-друге, важливі події введення/виведення можуть бути не оброблені і, по-третє, вразі краху процесу всередині критичної секції може відбутися порушення роботи ОС.

Спеціальні команди, що виконуються атомарно. Під атомарними операціями маються на увазі такі команди зміни даних в оперативній пам'яті, які виконуються процесором з повним блокуванням доступу до пам'яті, тобто жоден інший процесор або зовнішній пристрій не може отримати доступ до комірок пам'яті, поки процесор-монополіст не завершить операцію з нею.

Розглянемо реалізацію взаємного виключення за допомогою атомарної команди «перевірити і встановити» (TestAndSet). Ця команда виконує атомарно дві операції: читання з комірки пам'яті і запис в неї значення «зайнято».

Нехай нульове значення змінної *lock* відповідає умові, що критична секція вільна, а значення *lock*, відмінне від нуля, відповідає зайнятості критичної секції. Тоді перед входом у критичну секцію процес повинен в циклі перевіряти, чи звільнилася критична секція:

```
while (TestAndSet (& lock));
/ * Критична секція * /
```

Відповідно, після виходу з критичної секції процес повинен встановити змінну в значення «вільно»:

```
lock = 0;
```

2.2.3 Програмні способи досягнення взаємного виключення

Розрізняють наступні програмні способи досягнення взаємного виключення:

Змінні блокування.

Алгоритм Деккера.

Алгоритм Петерсона.

Змінні блокування. Процес для здійснення входу в критичну секцію безперервно перевіряє значення деякої змінної, щоб визначити, чи є вільним ресурс, що розділяється. Ця змінна має назву «*змінна блокування*» і зберігає стан критичної секції. Початкове значення змінної блокування дорівнює *0*. Якщо процес хоче увійти в критичну секцію, то він попередньо зчитує значення змінної блокування, і якщо воно дорівнює 0, то процес змінює його на 1 і входить в критичну секцію. Якщо ж значення цієї змінної вже дорівнює 1, то процес чекає, поки це значення не зміниться на 0. Таким чином, 0 означає, що жодного процесу в критичній секції немає, а 1 – що певний процес вже перебуває в критичній секції.

```
while (busy);
busy = 1;
/ * Критична секція * /
.....
busy = 0;
/ * Решта коду * /
.....
```

При такому підході виникають певні проблеми, а саме:

ті ж проблеми, що і при організації доступу до загальних змінних;

якщо після виходу з критичної секції процес з якоїсь причини не встановить значення змінної блокування рівним 0, то інші процеси не зможуть увійти в критичну секцію, порушується умова про те, що процес, який знаходиться поза критичною секцією не повинен блокувати інший процес;

проблема непродуктивної витрати часу процесору.

Активне очікування (*busy waiting*) – постійна перевірка змінної в очікуванні деякого значення. *Спін-блокування* – блокування, що використовує активне очікування.

Алгоритм Деккера. Голландський математик Т. Деккер був першим, хто розробив програмне рішення проблеми взаємного виключення (1970). Коли процес *P0* намагається увійти в критичну секцію, він встановлює свій прапор *flag [0]* таким, що рівний *true*, а потім перевіряє стан прапора іншого процесу *P1*. Якщо його прапор дорівнює *false*, то процес *P0* може негайно увійти в критичну секцію, інакше *P0* звертається до змінної *turn*.

Якщо *turn = 0*, то це означає, що зараз – черга процесу *P0* на вхід в критичну секцію, і тоді *P0* періодично перевіряє стан прапора процесу *P1* Цей процес, в свою чергу, в певний момент часу виявляє, що зараз не його черга для входу в критичну секцію, і встановлює свій прапор рівним *false*, даючи можливість процесу *P0* увійти в критичну секцію.

після того як *P0* вийде з критичної секції, він встановить свій прапор рівним *false* для звільнення критичної секції і присвоїть змінній *turn* значення *1* для передачі прав на вхід в критичну секцію процесу *P1*:

P0	*P1*
flag[0] = true; *while (flag[l])* *{* *if(turn == 1)* *{* *flag[0] = 1 false;* *while(turn == 1)* */* очікувати */;* *flag[0] = true;* *}* *}* */* критична секція */* *turn = 1;* *flag[0] = false;*	*flag[l] = true;* *while (flag[0])* *{* *if (turn == 2)* *{* *flag[0] = false;* *while(turn щ 1)* */* очікувати */;* *flag [1] = true;* *}* *}* */* критична секція */* *turn =0;* *flag[l] = false;*

Алгоритм Петерсена. Г. Петерсон розробив набагато більш простий алгоритм взаємного виключення (1981 р.). З цього моменту алгоритм Деккера вважається застарілим.

```
int turn; / * 0 або 1 - чия зараз черга? * /
int interested [2];
/ * Всі змінні спочатку рівні 0 * /

void enter_cs (int  proc)
{
      int other = 1 - proc;
      interested [proc] = true;
      turn = proc;
      while (turn == proc && interested [other]);
}
void leave_cs (int proc)
{

interested [proc] = false;
}
```

Перед входом у критичну секцію процес викликає функцію *enter_cs* зі своїм номером (0 або 1) в якості параметра. Після виходу з КС процес викликає функцію *leave_cs* , щоб позначити свій вихід і тим самим дозволити іншому процесу вхід в КС.

Початково обидва процеси знаходяться поза КС. Коли процес 0 викликає функцію *enter_cs*, ця функція повертає керування тому, що процес 1 не зацікавлений в попаданні в КС. Тепер, якщо процес 1 теж викличе функцію *enter_cs* , то йому доведеться чекати, коли *interested [0]* стане рівним *false*, це відбудеться тільки в той момент, коли процес *0* викличе функцію *leave_cs*.

Нехай тепер обидва процеси намагаються викликати функцію *enter_cs* практично одночасно. Тоді обидва вони збережуть свої номери в змінну *turn*. При цьому в ній збережеться номер того процесу, який був другим, а попередній номер буде загублений. Нехай *другим* був процес 1, і відповідно, *turn = 1*. У цьому випадку, коли обидва процеси дійдуть до оператора *while*, процес *0* увійде в КС, а процес *1* залишиться в циклі і буде чекати, поки процес *0* вийде з КС.

2.3 Примітиви синхронізації

2.3.1 Семафори

Семафор – це спеціальна змінна, яка має ціле значення і пов'язану з ним чергу. Над семафором визначено три операції:

семафор може бути ініціалізований ненегативним значенням;

операція *wait* зменшує значення семафора; якщо це значення стає негативним, то процес, що виконує операцію *wait*, блокується;

операція *signal* збільшує значення семафора; якщо це значення не позитивне, то заблокований операцією *wait* процес розблокується.

Існує також більш обмежена версія семафора – *бінарний семафор*, що приймає тільки значення 0 або 1.

Для зберігання процесів, що очікують семафори, використовується *черга*. Якщо вона обслуговується за принципом FIFO, то такий семафор називається *сильним семафором*, інакше – *слабким семафором*.

Можна сказати, що семафори забезпечують рішення для всіх видів проблем синхронізації, але їм також властиві недоліки:

процес, що використовує семафор, повинен бути обізнаний про інші процеси, що використовують цей же семафор, так як операції над семафорами у всіх взаємодіючих процесах повинні бути чітко скоординовані;

операції над семафорами у всіх процесах повинні бути ретельно налагоджені, так як пропуск однієї з таких операцій може призвести до неспроможності (порушення цілісності ресурсу, що розділяється) або до взаємного блокування;

програми, що використовують семафори, дуже важко перевіряти на коректність.

Наведемо приклад застосування семафорів при взаємному виключенні. Розглянемо рішення задачі взаємовиключення з використанням семафора *s*. Нехай є *n* процесів. У кожному з цих процесів перед входом у критичну секцію виконується виклик *wait (s)*. Якщо значення *s* при цьому стає негативним, то відповідний процес призупиняється. Якщо ж це значення дорівнює 1, то воно зменшується до нуля, а процес негайно входить у критичну секцію. І оскільки *s* вже більше не є позитивним, жоден інший процес не може увійти в критичну секцію.

```
semaphore s = 1:
.....
wait (s);
/ * Критична секція * /
signal (s);
/ * Решта коду * /
```

Семафор ініціалізується значенням 1. Отже, перший процес, що виконує виклик *wait (s)*, може негайно потрапити в критичну секцію, встановлюючи при цьому значення семафора рівним *0*. Будь-який інший процес при спробі увійти в критичну секцію виявить, що вона зайнята. Відповідно, відбудеться блокування такого процесу, а значення семафора буде зменшено до *-1*. Намагатися увійти в критичну секцію може будь-яка кількість процесів, і кожна така неуспішна спроба зменшує значення семафору. Після того як процес, який увійшов в критичну секцію першим, покине його, значення *s* збільшується, а один із заблокованих процесів (якщо такі є) віддаляється з черги семафора і активізується. Таким чином, як тільки планувальник ОС надасть цьому процесу можливість виконання, процес тут же зможе увійти в критичну секцію.

Сигналізуючий семафор – це семафор з нульовим початковим значенням. Процес сигналізує про подію, виконуючи операцію *signal (s)*, а всі інші процеси очікують події, виконуючи операцію *wait (s)*.

2.3.1 М’ютекси

М’ютекс - це спрощена версія семафора: змінна, яка може знаходитися в одному з двох станів – блокованому і неблокованому.

Значення м’ютекса встановлюється за допомогою двох процедур. Якщо процес (або потік) збирається увійти в критичну секцію, то він викликає процедуру *mutex_lock*. Якщо м’ютекс не заблокований, тобто вхід в критичну секцію дозволений, то запит виконується, а процес, що викликає входить в критичну секцію. Якщо ж м’ютекс заблокований, то процес блокується до тих пір, поки інший процес, що перебуває в критичній секції, не вийде з неї, викликавши процедуру *mutex_unlock*. Якщо при цьому м’ютекс блокував кілька процесів; то з них вибирається один.

2.3.3 Монітори

Щоб спростити написання програм, був запропонований механізм, синхронізації більш високого рівня.

Монітор – це набір процедур, змінних і інших структур даних, об'єднаних в особливий модуль. Він являє собою високорівневу конструкцію мови програмування, яка забезпечує функціональність, еквівалентну до функціональності семафорів, але при цьому ним набагато легше управляти. Монітори реалізовані в мовах програмування Concurrent Pascal, Pascal-Plus, Modula-2, -3, Java.

Монітор являє собою програмний модуль, який складається з послідовності, що ініціалізується однією або декількома процедурами і локальних даних. *Основні характеристики монітора*:

локальні змінні монітора доступні тільки його процедурам; зовнішні процедури доступу до локальних даних монітора не мають;

процес входить в монітор шляхом виклику однієї з його процедур;

в моніторі в певний момент часу може виконуватися тільки один процес; будь-який інший процес, що викликав монітор, буде припинений в очікуванні доступності монітора.

Перші дві характеристики нагадують нам об'єкти в об'єктно-орієнтованому програмуванні; дотримання ж третьої умови дозволяє монітору забезпечити взаємовиключення. Дані монітора доступні в деякий конкретний момент тільки одному процесу, - отже, захистити спільно використовувані дані можна, помістивши їх в монітор. Якщо дані в моніторі представляють якийсь ресурс, то монітор забезпечує взаємовиключення при зверненні до цього ресурсу.

Для широкого застосування в паралельних обчисленнях монітори повинні включати в себе інструмент синхронізації. Припустимо, наприклад, що якийсь процес використовує монітор і, перебуваючи в цьому моніторі, він повинен бути припинений до виконання певної умови. При цьому потрібно механізм, який не тільки призупиняє процес, але і звільняє монітор, дозволяючи увійти в нього іншому процесу. Пізніше, коли умова виявиться виконаною, а монітор доступним, припинений процес зможе продовжити свою роботу з того місця, де він був призупинений.

Монітор підтримує синхронізацію за допомогою *змінних умови*, що розташовуються в моніторі і доступних тільки в моніторі. Працювати з цими змінними можуть дві функції:

swait (c) – призупиняє виконання процесу, що викликає за умовою *c*; монітор при цьому доступний для використання іншим процесом;

csignal (c) – відновлює виконання процесу, припиненого викликом *cwait (c)*; якщо є декілька таких процесів, то вибирається один з них, а якщо таких процесів немає, то ця функція нічого не робить.

2.4 Взаємні блокування

Взаємне блокування – це блокування групи процесів, що очікують події, яка може ніколи не настати, оскільки дану подію може викликати тільки інший процес з тієї ж групи. Взаємоблокування зазвичай виникають при доступі процесів до ресурсів.

Ресурс – це будь-який об'єкт, який може запитуватися і очікуватися процесом; ресурс може складатися з будь-якої кількості ідентичних одиниць, і процес може запитувати будь-яку кількість одиниць ресурсу.

Існують дві *основні категорії ресурсів*:

Повторно використовувані ресурси (reusable) не виснажуються при використанні (можуть використовуватися знову). Вони мають фіксовану кількість одиниць, що не може створюватися або знищуватися. Процес, що запитує такий ресурс, утримує його при використанні, потім звільняє, а потім ресурс може бути виділений іншим процесам. Приклади повторно використовуваних ресурсів: процесор, пристрій вводу/виводу, основна та вторинна пам'ять, структури даних (файли).

Ресурси, що витрачаються (consumable) можуть бути створені (вироблені) або знищені (спожиті). Якщо процес використав такий ресурс, то останній припиняє своє існування («витрачається»). Зазвичай обмеження на кількість витрачених ресурсів певного типу відсутні, і не заблокований процес-виробник може створити будь-яку кількість таких ресурсів. Приклади ресурсів, що витрачаються: переривання, сигнали, повідомлення, інформація в буферах введення/виведення.

два режими доступу до ресурсу:

Монопольний (exclusive).

Розділюваний (shared).

При *монопольному (exclusive) режимі* доступ до ресурсу в кожний момент часу може отримати тільки один процес. При *розділюваному (shared) режимі* доступ до ресурсу одночасно може отримати будь-яка кількість процесів.

Ці режими несумісні, тобто до ресурсу не може бути реалізований доступ одночасно в монопольному і розділюваному режимі. Якщо ж існує багато одиниць (копій) ресурсу, то доступ до одних з них може здійснюватися в розділюваному режимі для декількох процесів, а доступ до інших – в монопольному режимі. Очевидно, що два цих режиму доступу мають сенс тільки для повторно використовуваних ресурсів, так як споживані ресурси знищуються відразу після використання.

Приклад 1. Взаємне блокування з повторно використовуваними ресурсами.

Процес P1
Запит ресурсу A
Запит ресурсу B

Процес P2
Запит ресурсу B
Запит ресурсу A

Коли обидва ці процеси дійдуть до другого запиту, виникне взаємне блокування . Один зі способів уникнути його – це накласти системні обмеження на порядок запиту ресурсів.

Приклад 2. Взаємне блокування з повторно використовуваними ресурсами.

Нехай нам доступно 200 Кб ОП і виконується наступна послідовність запитів:

Процес P1
Запит 80 Кб
Запит 60 Кб

Процес P2
Запит 70 Кб
Запит 80 Кб

Якщо обидва ці процеси дійдуть до другого запиту, то виникне взаємне блокування – незважаючи на те, що в системі досить пам'яті для виконання кожного з процесів окремо.

Приклад 3. Взаємне блокування з ресурсами, що витрачаються.

Нехай два процеси намагаються одержати повідомлення від іншого процесу, а потім відправити йому повідомлення.

Процес P1
ml = Receive (P2);
Send (P2, m2);

Процес P2
ml = Receive (P1)
Send (P1, m2);

Взаємне блокування при цьому виникне, якщо операція *Receive* є блокуючою.

Умови виникнення взаємного блокування :

взаємне виключення – одночасно використовувати ресурс може тільки один процес;

утримання та очікування – процес може утримувати виділені ресурси під час очікування інших ресурсів;

відсутність перерозподілу – ресурс не може бути примусово забраний процесу, що його утримує;

циклічне очікування – існує замкнутий ланцюг процесів, кожний з яких утримує як мінімум один ресурс, необхідний процесу, наступного у ланцюзі після даного процесу.

Моделювання взаємних блокувань.

Узагальнений ресурсний граф – це двочастковий спрямований граф з непересічними множинами вузлів процесів і ресурсів.

Вектор доступності – невід'ємний цілий вектор (r_1 r_2, ..., r_m), що позначає кількість одиниць ресурсів, доступних в будь-якому стані.

Кожен процес на узагальненому ресурсному графі позначається кружечком, а ресурс – прямокутником, де ресурс, що використовується – це прямокутник з подвійною рамкою. Кружечки всередині прямокутників позначають кількості одиниць даного типу ресурсу. Дуга, спрямована від процесу P до ресурсу R, називається *дугою запиту*. Вона означає, що процесу P призначено одну одиницю ресурсу R. Дуга, спрямована від повторно використовуваного ресурсу R до процесу P, називається дугою призначення. Вона означає, що процесу P призначена одна одиниця ресурсу R. Дуга, спрямована від ресурсу R, що використовується до процесу P, називається дугою виробника. Вона означає, що процес P є виробником ресурсу R.

Приклад (рисунок 2.8). Процеси P_1 і P_2 утримують по одній одиниці ресурсу R_1; при цьому процес P_2 є виробником споживаного ресурсу R_2. Доступно по одній одиниці кожного ресурсу. Процес P_1 активний, а P_2 – заблокований і очікує призначення одиниці ресурсу R_1. Процес P_i є заблокованим в якомусь стані тоді і тільки тоді, коли для деякого ресурсу R_j кількість дуг запиту більше r_j

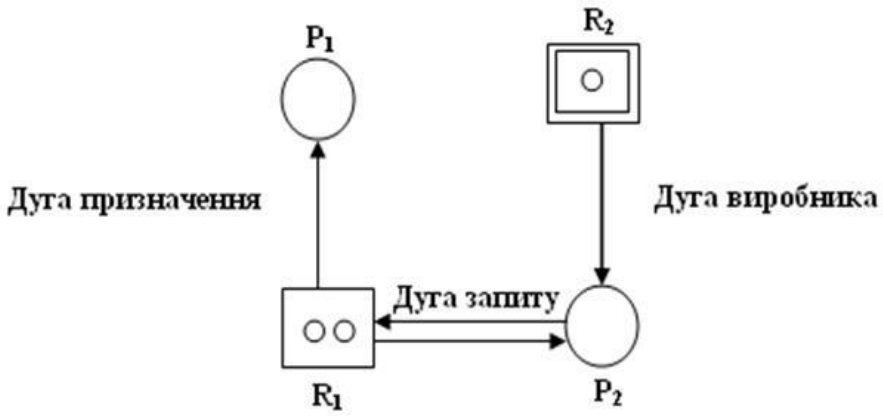


Рисунок 2.8 – Приклад графа ресурсів

Цикл в графі означає наявність взаємоблокування, циклічно включає в себе процеси і ресурси (припускаємо, що в системі є по одному ресурсу кожного виду).

Нехай існує три процеси: A, B, C і три ресурси: R, S, T, а також є наступна послідовність запитів і повернень ресурсу:

Процес A
запит R
запит S
звільнення R
звільнення S

Процес B
запит S
запит T
звільнення S
звільнення T

Процес C
запит T
запит R
звільнення T
звільнення R

ОС може запустити будь-який незаблокований процес в будь-який момент часу. Нехай запити ресурсів відбуваються у такому порядку:

процес A запрошує R;

B запрошує S;

C запрошує E;

A запрошує S;

B запрошує T;

C запрошує R.

Якщо ці шість запитів будуть здійснюватися в зазначеній послідовності, то ми отримаємо граф, показаний на рисунку 2.9, тобто відбудеться взаємне блокування.

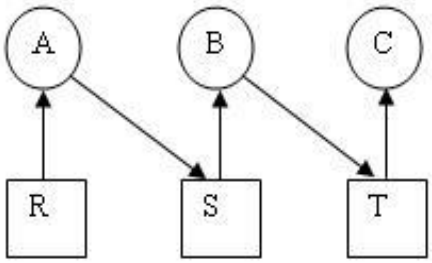


Рис. 2.9. Взаємне блокування.

Однак ОС не зобов'язана запускати процеси в якомусь особливому порядку. Зокрема, якщо виконання якого-небудь запиту спричиняє глухий кут, ОС може призупинити відповідний процес без задоволення запиту (тобто не виконуючи план цього процесу) до тих пір, поки це безпечно.

нашому випадку ОС могла б призупинити процес B, якби вона «знала» про виникнення взаємоблокування, і тим самим виключити це взаємоблокування.

Графи ресурсів є інструментом, що дозволяє побачити, чи може задана послідовність дій стати причиною взаємоблокування. Для цього після кожного кроку граф перевіряється на наявність у ньому циклів.

Важливим питанням є *виявлення взаємних блокувань*. Це можливо за наявності або одного ресурсу кожного типу, або наявності декількох ресурсів кожного типу.

Виявлення взаємних блокувань при наявності одного ресурсу кожного типу. У такій ситуації необхідною і достатньою умовою взаємоблокування є наявність циклу в графі ресурсів.

Виявлення взаємних блокувань при наявності декількох ресурсів кожного типу. Нехай n – кількість процесів P_1, P_2,…,

P_m; m – кількість класів ресурсів R_1, R_2 ,…, R_m;

E_i – кількість одиниць ресурсу класу;

– вектор існуючих ресурсів, що містить загальну кількість наявних в системі одиниць кожного ресурсу;

A – вектор доступних ресурсів, елемент якого A_i є кількість одиниць ресурсу R_i, що доступні в даний момент;

C – матриця поточного розподілу, елемент якої C_{ij} є кількість одиниць ресурсу R_j використовуване процесом P_i;

– матриця запитів, елемент якої R_{ij} є кількість одиниць ресурсу R_j, яку запитує процес P_j.

Останні чотири структури показані на рисунку 2.10.

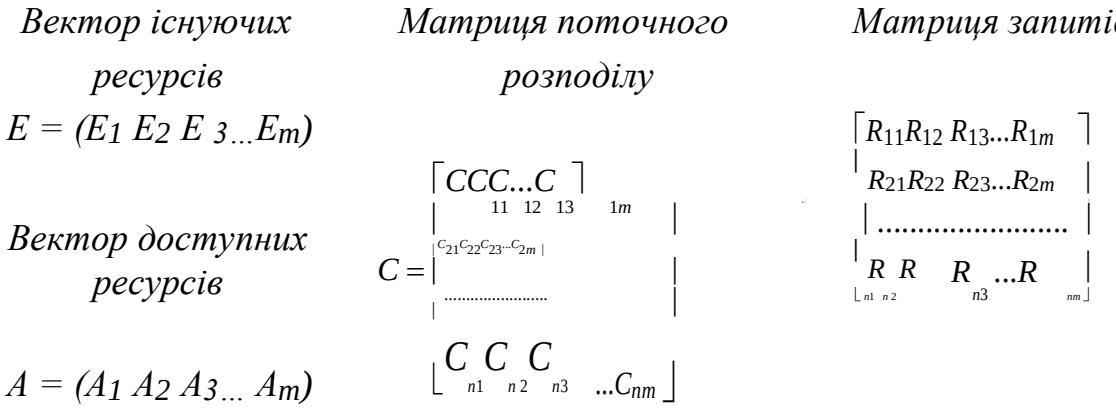


Рисунок 2.10 – Структури даних, що необхідні для алгоритму виявлення взаємних блокувань

Нехай у нас є алгоритм, який ставить на процесах відмітку, і вона є ознакою того, що ці процеси можуть закінчити роботу і, отже, не знаходяться у ситуації взаємного блокування. Тоді після завершення такого алгоритму будь-який немаркований процес знаходиться в тупіку.

Подібний алгоритм виявлення таких станів буде складатися з наступних кроків:

шукаємо немаркований процес P_i ,для якого *i*-й рядок матриці R поелементно не перевищує вектор A, тобто для всіх $1 \le k \le m$ виконується умова: $R_{ik} \le A_k$,

якщо такий процес знайдений, то додаємо *i*-й рядок матриці C до вектора A, маркуємо процес і повертаємося до кроку 1;

якщо таких процесів не існує, то робота алгоритму закінчується. Таким чином, на першому кроці цей алгоритм шукає процес, який може бути виконаний до кінця, тобто всі необхідні для нього ресурси доступні, виконує його і повертає зайняті таким процесом ресурси в загальний фонд ресурсів, а сам процес – маркує як завершений. Якщо ж виявиться, що всі процеси можуть працювати, то жоден з них в даний момент не заблокований.

Для ілюстрації роботи описаного алгоритму розглянемо рисунку 2.11.

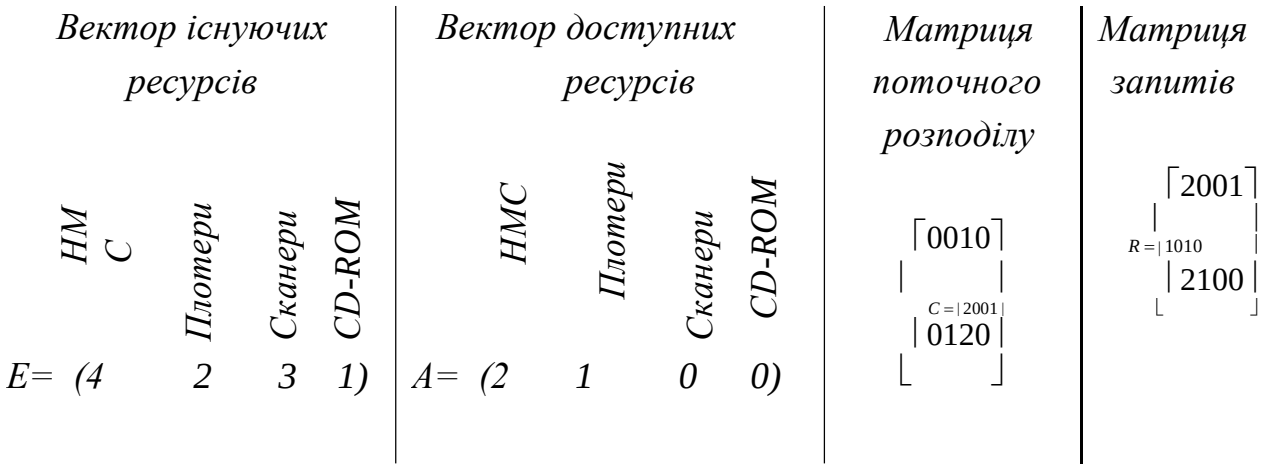


Рисунок 2.11 – Приклад використання алгоритму визначення взаємних блокувань

Тут є три процесу та чотири класи ресурсів (наприклад, накопичувачі на магнітній стрічці – НМС, плотери, сканери і пристрій для читання компакт-дисків). Процес 1 використовує один сканер. Процес 2 займає два НМС і пристрій для читання компакт-дисків. Процес 3 займає плотер і два сканера.

кожен з цих процесів потребує додаткового пристрою, як показує матриця R.

Працюючи з описаним вище алгоритмом виявлення взаємних блокувань, ми шукаємо процес, для якого запит ресурсів може бути виконаний. Вимоги першого процесу не можна виконати, тому що в системі немає доступного пристрою для читання компакт-дисків. Запити другого процесу також не можна задовольнити, так як немає вільних сканерів. Але третій процес може отримати все необхідне, тому він працює, завершується і повертає всі свої ресурси. Отже,

=(2220).

цього моменту може почати виконуватися процес 2, який по завершенні повертає свої ресурси в систему. Отримаємо: A = (4 2 2 1).

Тепер може працювати і процес, який залишився. Таким чином, у розглянутій системі не виникає взаємного блокування.

Далі змінимо варіант, що зображений на рис.2.11. Нехай процесу 3, окрім двох НМС і плотеру знадобиться також і пристрій для читання компакт-дисків. Тоді виникає ситуація, коли вже жоден із запитів не може бути виконаним. Тобто вся система опиниться заблокованою.

Для попередження взаємних блокувань необхідно визначати *безпечні* та *небезпечні стани* взаємодії процесів.

Стан безпечний, якщо він не перебуває в тупіку і якщо існує такий порядок планування, при якому кожен процес може виконатися до завершення, навіть якщо всі процеси захочуть отримати максимальну кількість необхідних їм ресурсів.

Небезпечний стан сам по собі ще не є блокованим. Однак тільки в безпечному стані система може гарантувати, що всі процеси успішно завершать свою роботу, а в небезпечному стані такої гарантії дати не можна.

Нідерландським вченим Е. Дейкстрою був розроблений **«алгоритм банкіра»,** який є розширенням алгоритму виявлення взаємних блокувань. Він перевіряє, чи веде виконання кожного з наявних запитів до небезпечного стану, і якщо так, то такий запит відхиляється.

«Алгоритм банкіра» для одного виду ресурсу. Щоб зрозуміти, чи є стан безпечним, перевіряється, чи можна надати достатньо ресурсів для завершення роботи процесу. Якщо так, то ці запити вважаються «погашеними», після чого перевіряється наступний, найближчий до межі запиту, процес. Якщо в результаті всі запити можуть бути виконані, то такий стан визнається безпечним, а початковий запит можна задовольнити (виконати).

На рисунок 2.12-а представлено чотири процеси, кожен з яких отримав певну кількість одиниць ресурсу. Усього є 10 одиниць ресурсу. В останній колонці міститься значення максимального запиту ресурсу кожним процесом.

Нехай в деякий момент виникає ситуація, що показана на рисунку 2.12-б. Цей стан є безпечним тому, що залишилися дві одиниці вільного ресурсу, і ОС може затримати всі запити, крім запитів процесу С, дозволяючи процесу С завершитися і повернути всі чотири одиниці ресурсу.

Розглянемо тепер, що могло б статися, якби в ситуації на рисунку 2.12-б був виконаний запит ще однієї одиниці ресурсу для процесу В. Ми потрапили б стан, що показаний на рисунку 2.12-в, який є небезпечним. Тоді, якщо б всі процеси запросили свою максимальну кількість одиниць ресурсу, то ОС не змогла б їх забезпечити, і ми потрапили б у глухий кут, тобто відбулося взаємне блокування.

Процес	Отримав	Мах
A	0	6
B	0	5
C	0	4
D	0	7

Вільно: 10

а)

Процес	Отримав	Мах
A	1	6
B	1	5
C	2	4
D	4	7

Вільно: 2

б)

Процес	Отримав	Мах
A	1	6
B	2	5
C	2	4
D	4	7

Вільно: 1

в)

Рисунок 2.12 – Три стани розподілу ресурсів:
а) – початкове, б) – безпечне, в) – небезпечне

«Алгоритм банкіра» для декількох видів ресурсів. Розглянутий вище «алгоритм банкіра» можна узагальнити і для управління системою з декількома видами ресурсів. Для цього:

Шукаємо в матриці R рядок, що відповідає процесу, у якого невиконані потреби ресурсів менше або дорівнюють вектору A. Якщо такого рядка не існує, то система в результаті потрапить в глухий кут, оскільки жоден процес не може пропрацювати до успішного завершення.

Припускаємо, що процес, рядок якого ми обрали в пункті 1, запросив всі необхідні ресурси і закінчив роботу. Відзначаємо цей процес як «завершений» і додаємо всі його ресурси до вектору A.

Повторюємо кроки 1 і 2 до тих пір, поки або всі процеси будуть помічені як «завершені», і стан в цьому випадку буде вважатися безпечним, або відбудеться взаємне блокування, і тоді стан буде вважатися небезпечним.

2.5 Класичні проблеми міжпроцесної взаємодії

Проблема виробника і споживача. «Проблема виробника і споживача», також відома як «проблема обмеженого буфера», полягає в наступному. Нехай два процеси спільно використовують буфер обмеженого розміру. Один з них («виробник»), поміщає дані в цей буфер, а другий, («споживач») – зчитує їх звідти.

Умови синхронізації:

«виробник» повинен чекати, якщо буфер повний;

«споживач» має чекати, якщо буфер порожній;

операції з буфером – це критичні секції, тобто працювати з буфером може тільки один процес.

Нижче наведено вирішення «проблеми виробника і споживача» з використанням семафорів:

```
   define N 100 / * кількість сегментів в буфері *
 / typedef int semaphore;
semaphore mutex = 1; / * контроль доступу в критичну секцію *
/ semaphore empty = N;
                    / * Кількість порожніх сегментів буфера *
/ semaphore full = 0;
                   / * Кількість повних сегментів буфера * /
void producer (void)
{
int item;
while (TRUE) {
item = produce__item ();
                  / * Створити дані, що поміщуються в буфер *
/ wait (& empty);
                 / * Зменшити лічильник порожніх сегментів буферу * /
wait (& mutex); / * вхід у критичну область * /
 insert___itern (item);
                 / * Помістити в буфер новий елемент *
/ signal (& mutex);
                / * Вихід з критичної області * /
signal (full);
               / * Збільшити лічильник повних сегментів буфера * /
       }
}
void consumer (void);
{
      int item;
while (TRUE) {
      wait (& full);
           / * Зменшити кількість повних сегментів буфера * /
    wait (& mutex); / * вхід у критичну область * /
    item = remove__item ();
                       / * Видалити елемент з буфера * /
signal (& mutex);
                         / * Вихід з критичної області * /
signal (& empty);
                      / * Збільшити лічильник порожніх сегментів буфера *
/ consume_item (item); / * обробка елементу * /
         }
}
```

Тут семафор *mutex* використовується для реалізації взаємного виключення, тобто для виключення одночасного звернення до буферу двох процесів. Решта семафорів використані для синхронізації: семафори *full* і *empty* необхідні, щоб гарантувати, що «виробник» припиняє роботу, коли буфер повний, а «споживач» припиняє роботу, коли буфер порожній.

Можна узагальнити цю задачу і на випадок *m* виробників і *n* споживачів.

Задача про «читачів» і «письменників». Це одна із задач синхронізації, що описує доступ процесів до поділюваної бази даних. Процеси при цьому розділяються на дві категорії: «читачі» – ніколи не модифікують базу даних, і «письменники» – можуть читати і модифікувати базу даних.

Правила синхронізації тут наступні:

«читачі» можуть працювати паралельно, якщо ніхто з «письменників» не модифікує базу даних;

модифікувати базу може тільки один «письменник», коли ніхто не читає інформацію з бази даних;

змінні стану може переглядати та змінювати тільки один процес.

Нижче наведена схема вирішення даної задачі:

«Читач»	**«Письменник»**
Чекати, поки закінчить роботу «письменник» (якщо хтось пише)	Чекати, поки закінчать роботу всі «читачі» (якщо хтось читає)
Читати з бази даних	Модифікувати базу даних
За необхідністю – «розбудити» «письменника», що очікує	За необхідністю – «розбудити» «читачів», що очікують

Задача про «філософів, що обідають». Ця задача моделює використання загальних ресурсів декількома процесами. Суть її така. П'ять філософів сидять за круглим столом. На столі – п'ять тарілок з макаронами і п'ять виделок, по одній між тарілками. Передбачається, що треба їсти двома виделками відразу. Поведінка кожного з філософів наступна: думає – хоче їсти (намагається взяти дві виделки) – їсть.

Обмеження:

кожен філософ повинен чекати, поки не звільняться дві виделки поруч з ним;

модифікація змінних стану повинна виконуватися в критичній секції.

Задача про «сплячого перукаря». Ця задача моделює обслуговування потоку запитів по черзі. Клієнти приходять у перукарню і встають в чергу. Перукар запрошує і обслуговує клієнтів по одному.

Обмеження:

клієнт повинен чекати, якщо є черга і / або перукар зайнятий обслуговуванням іншого клієнта;

перукар «спить», якщо немає клієнтів;

перший клієнт «будить» перукаря.

2.6 Потоки

Визначення процесу, яке було описане вище, базується на двох незалежних концепціях: на групуванні ресурсів і на виконанні програм. Якщо ж їх розділити, то з'явиться поняття *потоку*. Ресурсами керувати простіше, об'єднавши їх у формі процесу, однак, з іншого боку, процес можна розглядати як потік виконуваних команд, або просто потік.

Потік – це незалежно планований контекст виконання, що розділяє єдиний адресний простір з іншими потоками свого процесу.

Елементи, що спільно використовуються потоками, – адресний простір, глобальні змінні, відкриті файли, дочірні процеси, сигнали і їх обробники, інформація про використання ресурсів.

Елементи, що окремо використовуються потоками, – лічильник команд, регістри, стек, стан. Лічильник команд при цьому відстежує порядок виконання дій, регістри зберігають поточні значення змінних, стек містить протокол виконання процесу і тимчасові змінні.

Хоча потік виконується всередині процесу, концепції процесу і потоку різні (табл. «Порівняння процесів і потоків»). Концепція потоків, зокрема, додає до моделі процесу можливість одночасного виконання в одному і тому ж середовищі процесу декількох достатньою мірою незалежних команд.

Порівняння процесів і потоків

Можливість	**Процес**	**Потік**
Час для створення	Велике	Мале
Час для перемикання	Велике	Мале
Механізм взаємодії	Складний	Простий
Колективні дані	Немає	Є
Захист	Є	Немає
Положення	Той же самий або на різних комп'ютерах	Той же комп'ютер

Потоки володіють деякими властивостями процесів, тому їх іноді називають «спрощеними процесами» (lightweightprocesses).

Багатопоточність - виконання декількох потоків в одному процесі. При запуску багатопотокового процесу в системі з одним процесором потоки працюють по черзі. Ілюзія ж паралельної роботи потоків створюється шляхом постійного перемикання системи між потоками. І лише в багатопроцесорній або багатоядерній системі може бути реалізовано реальне паралельне виконання команд різних потоків.

Різні потоки в одному процесі не настільки незалежні, як різні процеси. всіх потоків – один адресний простір, що означає спільне використання глобальних змінних. Оскільки будь-який потік має доступ до будь-якої адреси комірки пам'яті в адресному просторі процесу, то один потік може змінювати інформацію в стеку іншого потоку. Захисту від цього не існує, оскільки, по-перше, це неможливо, а по-друге, це й не потрібно, так як потоки зазвичай виконують спільну задачу.

Якщо програма паралельно виконує багато різних дій і деякі з них можуть час від часу блокуватися, то використання декількох потоків дає наступні переваги:

спрощується сам програмний додаток;

полегшується спільне використання даних;

підвищується продуктивність (за наявності операцій введення/виведення);

знижується навантаження на підсистему керування пам'яттю;

з'являється можливість для повноцінного використання багатопроцесорних систем.

2.6.1 Основні операції з потоками та способи реалізації потоків

Створення нового потоку відбувається за допомогою функції *pthread_create* – в UNIX, *CreateThread* – в Windows. Основним параметром цих функцій є ім'я процедури, яку необхідно запустити в новому потоці.

Windows також передбачена функція *CreateRemoteThread*, що дозволяє створити потік в іншому процесі.

Завершити потік можна зсередини самого цього потоку функцією *pthread_exit* в UNIX або *ExitThread* – в Windows. Після цього потік зникає і планувальник його не використовує. Завершити потік можна також з іншого потоку за допомогою функції *pthread_cancel* – в UNIX або *TerminateThread* – в Windows.

Очікування потоком завершення іншого певного потоку виконується за допомогою функції *pthread_join* – в UNIX або *WaitForSingleObject* і *WaitForMultipleObject* -в Windows.

Потік може добровільно надати свою чергу іншому потоку за допомогою функцій *pthread_yield* – в UNIX або *SwitchToThread* і *SuspendThread* – в Windows.

Існують такі способи реалізації потоків:

- *У просторі користувача.*
- *У просторі ядра.*
- *Змішана реалізація.*

випадку реалізації потоків у просторі користувача пакет підтримки потоків цілком розміщується в просторі користувача. Ядро ОС при цьому нічого не знає про потоки і керує звичайними однопоточними процесами. Кожен процес тут має власну таблицю потоків. Коли потік збирається виконати дію, яка може призвести до локального блокування, він викликає процедуру з пакету підтримки потоків. Ця процедура зберігає стан потоку в таблиці потоків, потім шукає в таблиці потік, готовий до запуску, і завантажує в регістри його стан.

Реалізація потоків у просторі користувача забезпечує наступні переваги:

можна реалізувати цей механізм в ОС, що не підтримує потоки;

переключення потоків відбувається на порядок швидше, ніж в режимі ядра;

кожен процес може мати свій власний алгоритм планування;

вище масштабованість додатків.

Проблеми, що виникають при реалізації потоків у просторі користувача:

блокуючий системний виклик в одному з потоків зупинить всі потоки процесу;

при запуску одного потоку жоден інший потік не буде запущений, поки перший потік добровільно не звільнить процесор.

випадку реалізації потоків у просторі ядра таблиця потоків розташовується в ядрі, яке також містить звичайну таблицю процесів. Всі запити, які можуть блокувати потік, тут реалізуються як системні виклики. Коли один потік блокується, ядро ОС запускає інший потік з цього ж процесу або потік з іншого процесу. Недоліком такого підходу є збільшення часу перемикання потоків, оскільки всі операції з потоками реалізуються через системні виклики, а не за допомогою бібліотечних функцій.

випадку змішаної реалізації потоків ядро ОС «знає» тільки про потоки свого рівня і управляє ними. Деякі потоки ядра при цьому можуть містити по декілька потоків користувацького рівня, які управляються так само, як потоки в системі, що не підтримує багатопоточність. У даному підході поєднуються переваги перших двох методів.

2.6.2 Організація процесів і потоків в Linux

UNIX-системах для кожного компонента образу процесу виділяють окрему ділянку пам'яті. Заначимо,що образ процесу UNIX-системах містить такі компоненти:

керуючий блок процесу;

код програми, яку виконує процес;

стек процесу, де зберігаються тимчасові дані (параметри процедур, повернені значення, локальні змінні тощо);

глобальні дані, спільні для всього процесу.

Керуючий блок процесу в Linux відображається структурою даних *task_struct,* основні поля якої містять наступну інформацію:

ідентифікаційні дані (зокрема pid — ідентифікатор процесу);

стан процесу (виконання, очікування тощо);

покажчики на структури предка і нащадків;

час створення процесу та загальний час виконання (так звані таймери процесу);

стан процесора (вміст регістрів і лічильник інструкцій);

атрибути безпеки процесу (uid, gid, euid, egid).

Також *task_struct* має кілька полів спеціалізованого призначення, що необхідні для різних підсистем Linux, а саме

відомості для обробки сигналів;

інформація для планування процесів;

інформація про файли і каталоги, пов'язані із процесом;

структури даних для підсистеми керування пам'яттю.

Структура процесу містить інформацію, необхідну протягом усього часу існування процесу (зокрема, коли він вивантажений на диск). Це такі дані, як pid, ефективний uid, пріоритет, покажчик на структуру користувача. Структури процесу поєднуються в черзі процесів у системі. Структура користувача містить інформацію, необхідну тільки під час виконання процесу (стан процесора, uid, gid, поточний каталог, інформацію про відкриті файли).

Необхідно зазначити, що в операційній системі Linux новий прцес створюється шляхом копіювання атрибутів поточного процесу.

Новий процес може бути клонованим (cloned) і, при цьому такі ресурси, як файли, обробники сигналів і віртуальна пам’ять, використовуються сумісно. І, як наслідок, якщо два процеси використовують одну і туж віртуальну пам’ять, то вони функціонують як потоки в рамках одного і тогож процесу. Але для потоків структури даних окремо не задаються, і таким чином в операційній системі Linux потоки і процеси не розрізняють.

На рисунку 2.13 представлено модель процесів і потоків Linux.

Виконуваний (*executing*). Це стан, при якому поточний процес або виконується, або готовий до виконання.

Підлягає перериванню (interruptible). Це стан блокування, в якому процес очікує настання події, наприклад, завершення операції введення/виведення, звільнення ресурсу або сигналу від іншого процесу.

Не підлягає перериванню (uninterruptible). Це стан блокування, при якому процес в неперервному стані безпосередньо очікує виконання певної апаратної умови, тому він не сприймає ні яких сигналів.

Зупинений (stopped). Процес був зупинений і може бути продовжений тільки при відповідному впливі іншого процесу.

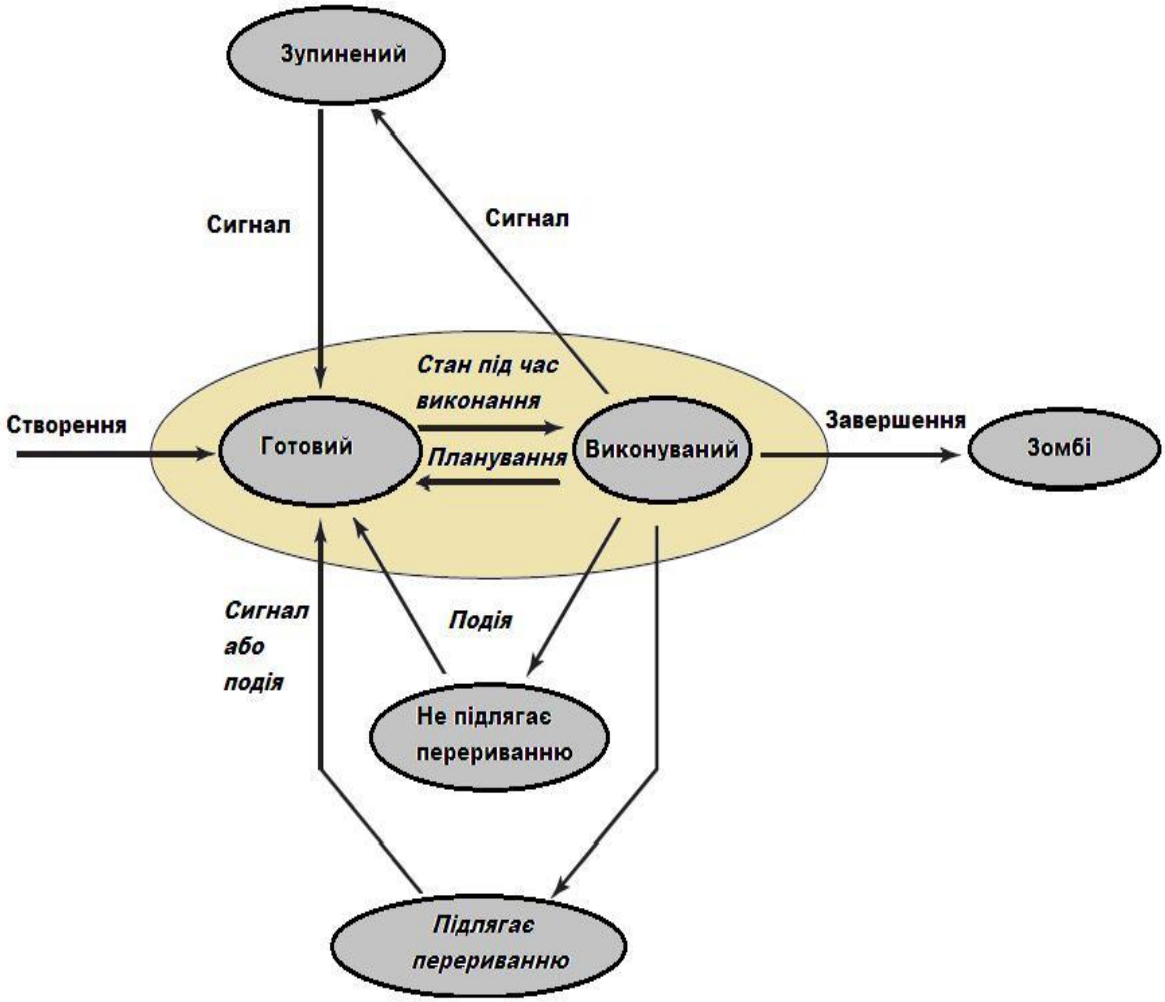


Рисунок 2.13 – Стани виконання процесу в Linux

Зомбі (*zombie*). Процес був завершений, але за якоюсь причиною його структура залишається в таблиці процесів.

2.7 Сигнали ОС UNIX

Сигнал в ОС UNIX - це асинхронне повідомлення процесу про будь-яку подію. Функції сигналів:

оповіщення процесів про виникнення системних подій;

забезпечення механізму для комунікації та синхронізації між прикладними процесами.

Сигнали дають можливість викликати спеціальну функцію – *обробник сигналу* при виникненні певної події. Після обробки сигналу таким обробником

основна програма продовжує роботу з того місця, в якому вона була перервана надходженням сигналу. Події позначаються цілими числами і представляються як символьні константи.

Процес оповіщення про подію складається з двох етапів – *генерації сигналу і обробки сигналу.* При настанні події, пов'язаної з будь-яким сигналом, ядро ОС генерує цей сигнал шляхом встановлення відповідного біта в масці сигналів, що очікують в керуючому блоці процесу. При цьому процес може отримати сигнал лише в певні моменти часу:

при виборці процесу диспетчером на виконання з черги готових;

перед блокуванням або після блокування;

під час деяких блокувань (наприклад, під час очікування введення з клавіатури).

Незалежно від джерела сигналу всі сигнали процесу доставляє ядро ОС.

Обробка сигналів. Кожен сигнал володіє дією, що встановлена за замовчуванням, яка виконується ядром, якщо процес не встановив для цього сигналу іншу дію.

Всього таких дій може бути п'ять:

аварійне завершення зі створенням дампа пам'яті (*Dump*);

аварійне завершення без створення дампа пам'яті (*Abort*);

ігнорування сигналу (*Ignore*);

призупинення процесу (*Stop*);

відновлення роботи призупиненого процесу (*Continue*).

Процес може перевизначити дію за замовчуванням для будь-якого сигналу (крім сигналів *SIGKILL* і *SIGSTOP)* на ігнорування цього сигналу або на свій обробник сигналу. Процес також може в будь-який час вказати нову дію або скинути установки на дію за замовчуванням. Процес може тимчасово блокувати сигнал, і цей сигнал не буде оброблено, доки він не буде розблокований (функція *sigprogmask*).

Щоб вказати спосіб, яким необхідно обробити сигнал, слід виконати наступні дії:

написати функцію обробника сигналу, включивши її в програму, для якої вона призначена; 2) зареєструвати цю функцію, викликавши на початку програми функцію *sigaction*. Виклики *sigaction* можна роити багаторазово, задаючи різні реакції на сигнал.

Нижче наведено приклад обробки сигналів при натисканні комбінації клавіш *Ctrl + C* (дією за замовчуванням для цього сигналу зазвичай є аварійне завершення програми):

```
include <signal.h>
.....
void handler (int sig);
{printf ("Отримано сигнал переривання | n");
}
main ()
{Struct sigaction act;
        act. sa__handler = handler;
        if (sigaction (SIGINT, & act, 0)! = 0) {perror
        (...); exit (1); }
            / * Продовження роботи * /
}
```

таблиці 2.1 «Сигнали, визначені в стандарті POSIX» наведено 20 сигналів, що визначені в стандарті POSIX[1].

Таблиця 2.1 Сигнали, визначені в стандарті POSIX

№	Позначення сигналу	Дія за замовчуванням	Опис
1	*SIGHUP*	*Abort*	Зависання терміналу / аварійне зупинення керуючого процесу
2	*SIGINT*	*Dump*	Переривання від клавіатури (Ctrl + C)
3	*SIGQUIT*	*Dump*	Сигнал виходу з терміналу (Ctrl + \)
4	*SIGILL*	*Dump*	Невірна команда
5	*SIGTRAP*	*Dump*	Пастка трасування / точки зупину
6	*SIGABRT*	*Dump*	Сигнал *Abort* від клавіатури
7	*SIGBUS*	*Dump*	Помилка шини
8	*SIGFPE*	*Dump*	Виключення з плаваючою точкою

Стандарт POSIX розроблено Інститутом інженерів з електротехніки і електроніки (IEEE), для того що б надати можливість писати програми, що працюють в будь-якій *UNIX*-системі. Стандарт POSIX визначає мінімальний інтерфейс системних викликів, який має підтримувати всі системи, що сумісні з *UNIX*. *Деякі ОС,* що відрізняються від *UNIX*, тепер теж можуть підтримувати стандарт POSIX.

№	Позначення сигналу	Дія за замовчуванням	Опис
9	*SIGKILL*	*Abort*	Сигнал *Kill*
10	*SIGUSR1*	*Abort*	Сигнал № 1, визначений користувачем
11	*SIGSEGV*	*Dump*	Невірна адреса пам'яті
12	*SIGUSR2*	*Abort*	Сигнал № 2, визначений користувачем
13	*SIGPIPE*	*Abort*	Запис в канал без його читання
14	*SIGALRM*	*Abort*	Сигнал таймеру від функції *alarm*
15	*SIGTERM*	*Abort*	Сигнал завершення
16	*SIGSTKFLT*	*Abort*	Помилка стека сопроцесора
17	*SIGCHLD*	*Ignore*	Нащадок зупинений або завершений
18	*SIGCONT*	*Continue*	Продовжено зупинений процес
19	*SIGSTOP*	*Stop*	Процес зупинено
20	*SIGTSTP*	*Stop*	Сигнал зупинки терміналу (Ctrl +Z)

Під час обробки сигналу інші сигнали того ж самого типу блокуються. Тому обробники сигналів можуть не бути реєнтерабельними[1], так як кілька копій обробника не можуть виконуватися одночасно. Однак обробник сигналу одного типу може перервати роботу оброблювача сигналу іншого типу.

Звичайні сигнали ядро не запам'ятовує в черзі. Якщо якийсь сигнал вже обробляється, то другий сигнал того ж типу, що надійшов в цей момент, просто губиться. У системах реального часу втрата сигналів є критичною. Тому багато версій UNIX (включаючи Linux) підтримують сигнали реального часу. Такі сигнали ядро поміщає в чергу, і вони не губляться – гарантується їх доставка в порядку FIFO.

Посилка сигнала процесу. Послати сигнал процесу можна за допомогою системного виклику kill:

```
   include <sys/types.h>
   include <signal.h>
int kill (pid_t pid, intsig);
```

Ядро Linux є реентерабельним. Це означає, що одночасно в режимі ядра може виконуватися код кількох процесів. В однопроцесорних системах процесор у конкретний момент виконує код тільки одного процесу, інші перебувають у стані очікування. У багатопроцесорних системах код різних процесів може виконуватися паралельно.

Тут pid – це одержувач сигналу, а sig - номер сигналу. Якщо pid> 0, то сигнал sig надсилається процесові з *PID* = pid.

Якщо *pid* = 0, то сигнал посилається всім процесам у групі, до якої належить поточний процес. Якщо pid = -1, то сигнал посилається всім процесам в системі, крім першого (init), починаючи від найбільших PID в таблиці процесів. Якщо ж pid <-1, то сигнал посилається всім процесам у групі процесів-pid.

Контрольні питання

Що таке процес?

Що таке мультипрограмування?

Назвіть причини для створення процесу в обчислювальних системах?

Які ситуації призводять до завершення процесу?

Які можливі переходи між станами процесів в моделі процесу з п’ятьма станами?

З яких елементів складається образ процесу?

Що входить до управляючого блоку процесу?

Що включають атрибути процесу?

Які дії виконує ОС при створенні процесу?

Опишіть ОС на основі ядра в складі користувацьких процесів?

Що таке планування процесів? Які є типи планування процесів?

В чому відмінність між цілісним і розподіленим планувальником? Які основні функції планувальника процесів?

Що таке безпріорітетна організація процесів? Які недоліки такої організації процесів?

За яким принципом відбувається робота дисципліни диспетчеризації FIFO?

За яким принципом відбувається робота дисципліни диспетчеризації LIFO?

За яким принципом відбувається робота дисципліни диспетчеризації FCFS?

За яким принципом відбувається робота дисципліни диспетчеризації SJN?

За яким принципом відбувається робота дисципліни диспетчеризації SRT?

За яким принципом відбувається робота дисципліни диспетчеризації RR?

Які переваги багатозадачності з витісненням на відміну від багатозадачності без витіснення?

Як може бути досягнуто гарантоване обслуговування процесів?

Які є види взаємодії процесів?

Які можливі проблеми при взаємодії процесів?

Який механізм гарантовано запобігає виникненню взаємних блокувань при взаємодії процесів?

Які апаратні способи досягнення взаємного виключення?

Опишіть алгоритм Деккера? Який його недолік?

Для чого застосовуються примітиви синхронізації? Які існують примітиви синхронізації?

Що таке семафор? У якому випадку семафор називається сильним?

Наведіть приклад взаємного блокування з повторно використовуваними ресурсами?

Дайте визначення безпечних та небезпечних станів взаємодії процесів.

Назвіть і опишіть класичні проблеми міжпроцесної взаємодії.

Що таке потік? Якими елементами характеризується потік? Які переваги при використанні декількох потоків?

Які існують способи реалізації потоків?

Які є стани виконання процесів в Linux?

Що таке сигнал в ОС UNIX? Обробка сигналів UNIX?

РОЗДІЛ 3

УПРАВЛІННЯ ПАМ'ЯТТЮ В ОПЕРАЦІЙНИХ СИСТЕМАХ[1]

Пам'ять в операційних системах - це ресурс комп'ютера, призначений для зберігання програмного коду і даних. Пам'ять зображають як масив машинних слів або байтів з їхніми адресами.

Існують різні види пам'яті, які знаходяться в певній ієрархії. **Жорсткий диск комп'ютера** (допоміжний запам'ятовувальний пристрій – *secondary storage*) є найдешевшим і найповільнішим запам'ятовувальним пристроєм. **Оперативна пам'ять** зберігається в мікросхемах пам'яті, її називають **основною пам'яттю** (*main memory*) – вона є швидшою й дорожчою. Швидшими засобами зберігання даних є **кеші** процесора і обсяг цих кешів обмежений.

Керування пам'яттю в ОС – досить складне завдання. Переважно потрібної за характеристиками пам'яті виявляється недостатньо, і щоб це не заважало роботі користувача, стає необхідним координувати між собою різні види пам'яті.

Управління пам'яттю поряд з управлінням процесами і ресурсами, – одна з найбільш важливих функцій операційної системи.

Будь-яка програма, введена в систему, повинна бути розміщена в пам'яті і оформлена у вигляді процесу для її виконання. Кожна програма при введенні в систему поміщається у вхідну чергу – сукупність процесів на диску, які очікують розміщення в пам'яті для виконання своїх програм. До свого виконання користувацькі програми проходять у системі декілька стадій.

Багатоетапна обробка користувацької програми.

Вихідний код програми (у формі текстового файлу) мовою високого рівня або на асемблері перетворюється компілятором або асемблером в об'єктний модуль, що містить бінарні виконувані машинні команди і таблицю символів, визначених і використаних в даному модулі коду. Розглянута фаза називається часом компіляції.

Однак об'єктний модуль не може безпосередньо виконуватися, оскільки він містить недозволені посилання на зовнішні модулі та їх компоненти. Наступна фаза обробки програми – редагування зв'язків. Редактор зв'язків

[1] При підготовці даного розділу були використані літературні джерела [1–3], [4], [6], [15], [17], [21–23] та інші із списку літературних джерел.

(linker) – системна програма, яка отримує на вхід один або кілька об'єктних модулів, а на виході видає завантажувальний модуль – двійковий код, утворений кодом декількох об'єктних модулів, в якому дозволені всі міжмодульні посилання – для кожного символу, зовнішнього для даного об'єктного модуля.

Завантажувальний модуль може бути завантажений в пам'ять для виконання з допомогою ще однієї системної програми – завантажувача (loader), який отримує на вхід завантажувальний модуль і файли з бінарними кодами системних бібліотек, які використовує програма. Завантажувач, об'єднуючи код програми з кодами системних бібліотек, створює бінарний образ програми в пам'яті.

Логічний та фізичний адресний простір.

Концепція логічного адресного простору, пов'язаного з відповідним фізичним простором, є однією з основних для управління пам'яттю.

Логічною адресою називається адреса, що генерована процесором при виконанні машинної команди.

Фізична адреса – це реальна адреса в пам'яті, яку «бачить» і «розуміє» пристрій управління пам'яттю (Memory Management Unit - MMU).

Логічні адреси збігаються з фізичними при зв'язуванні адрес під час компіляції або під час завантаження (тобто до виконання програми). Однак при зв'язуванні адрес під час виконання логічні адреси відрізняються від фізичних.

Пристрій управління пам'яттю.

Пристрій управління пам'яттю (*Memory Management Unit – MMU*) – це один з модулів апаратури, що відповідає за адресацію пам'яті і пов'язаний з процесором та іншими пристроями системною шиною. З точки зору підтримки описаних концепцій адресації, пристрій управління пам'яттю – це апаратура, перетворююча логічну адресу (отримана по загальній шині від процесора) у фізичний (реальна адреса в пам'яті, за якою і відбувається звернення).

Апаратура MMU використовує значення регістру переміщення, що містить адресу початку області пам'яті, виділеної ОС для програми користувача. MMU додає значення регістра переміщення до (логічної) адреси, що згенерована програмою користувача, отримуючи в результаті фізичну адресу.

Програма користувача працює тільки з логічними адресами і не «бачить» фізичних адрес. Схема адресації і перетворення логічної адреси у фізичну з використанням регістра переміщення зображена на рисунку 3.1.

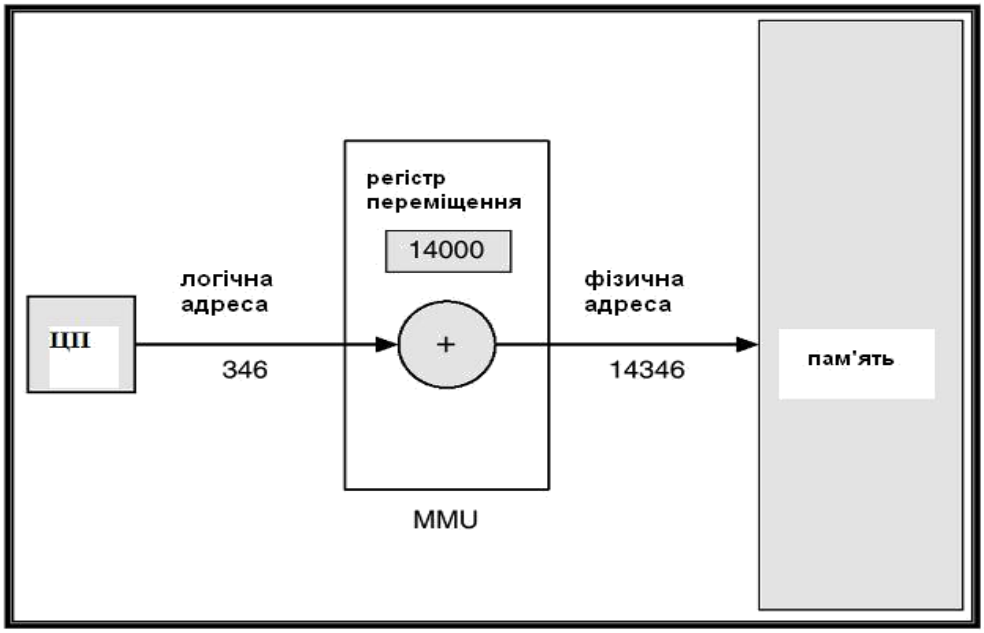


Рисунок 3.1 – Адресація з використанням регістру переміщення

Відкачування і підкачка.

Користувацький процес може перебувати в різних станах під час обробки системою. Зокрема, процес може бути деякий час неактивним, якщо, наприклад, він виконується в режимі поділу часу, і користувач за терміналом обмірковує наступну команду або редагує вихідний код своєї програми.

подібних випадках процес може бути відкачано операційною системою на диск тому, що пам'ять, яку він займає необхідна в даний момент для іншого активного процесу.

Відкачування і підкачка (swapping) – це дії операційної системи з відкачування (запису) образу неактивного процесу на диск або підкачування (зчитування) активного процесу в основну пам'ять. Необхідність виконання подібних дій викликана нестачею основної пам'яті.

Файл відкачування (backing store) – область дискової пам'яті, використовувана операційною системою для зберігання образів відкачаних процесів. Файл відкачування організується максимально ефективно: забезпечується прямий доступ до всіх образів процесів в пам'яті (наприклад, через таблицю за номером процесу).

Популярний різновид стратегії відкачування і підкачки – ***roll out / roll in:*** відкачування і підкачка на базі пріоритетів; більш пріоритетні процеси виконуються, менш пріоритетні – відкачуються на диск.

При відкачуванні найбільше часу витрачається при передачі даних: повний образ процесу може займати велику область пам'яті. Загальний час відкачування пропорційний розміру відкачуваних даних.

поширених ОС – UNIX, Linux, Windows та ін – реалізовані різні стратегії відкачування і підкачки.

Схема відкачування та підкачування зображена на рисунку 3.2.

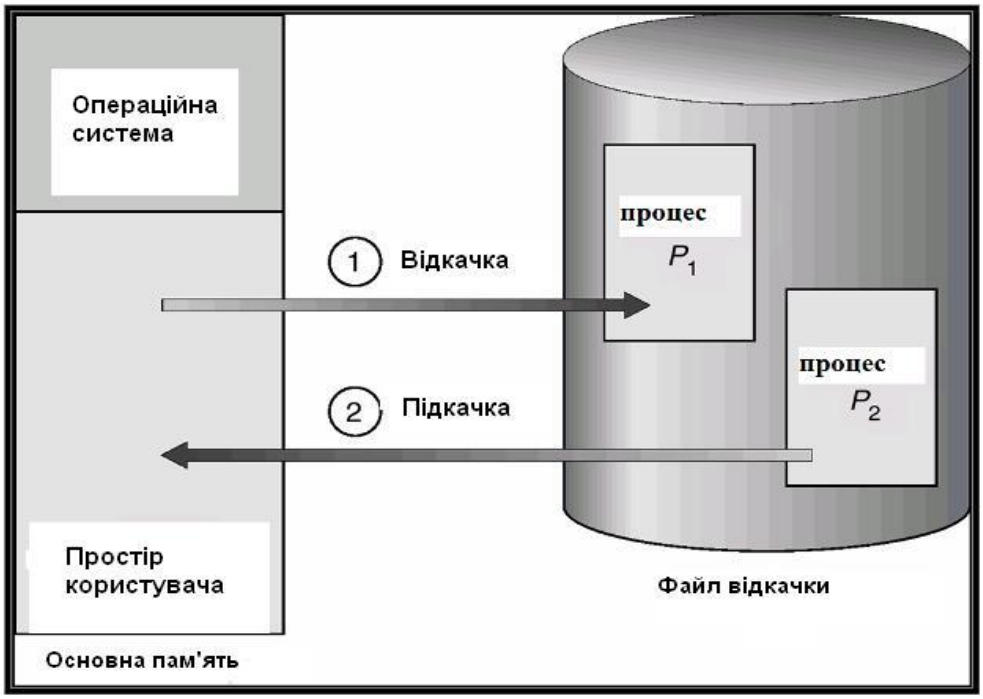


Рисунок 3.2 – Схема підкачки і відкачки

3.1 Технології розподілу пам'яті

Оперативна пам'ять (ОП) – це важливий ресурс ЕОМ. У ранніх ОС управління пам'яттю зводилося до завантаження програми і її даних із зовнішньої пам'яті в ОП.

Функції мультипрограмної ОС по управлінню пам'яттю:

відстеження вільної і зайнятої пам'яті;

виділення пам'яті процесам і звільнення пам'яті по завершенні цих процесів;

захист пам'яті;

настроювання адрес програм;

виділення динамічної пам'яті;

переміщення програм між ОП і зовнішньою пам'яттю. При виконанні останньої функції існує дві стратегії:

1) *свопінг*, або проста підкачка – програми цілком завантажуються в ОП і цілком вивантажуються на диск;

2) програми частково завантажуються в ОП і частково вивантажуються на диск; якщо це прозоро для програміста, то така стратегія називається *віртуальною пам'яттю*, а якщо не прозоро, то *оверлеєм*.

Головна операція по управлінню пам'яттю – це розміщення програми в ОП для її виконання процесором. У сучасних ОС при цьому використовується складна система, що називається *віртуальною пам'яттю*, яка використовує одну чи обидві базові технології – сегменти і сторінки. Але спочатку ми познайомимося з більш простими технологіями.

3.1.1 Фіксований розподіл пам'яті

У цьому випадку пам'ять поділяється на області з фіксованими межами.

Розділи однакового розміру. Будь-який процес, розміри якого не перевищують розміру розділу, може бути завантажений в будь-який доступний розділ. Якщо ж всі розділи зайняті і немає жодного процесу в стані готовності або роботи, то ОС може вивантажити процес з будь-якого розділу і завантажити нього інший процес, забезпечуючи процесор роботою. Прийняття рішення, який процес вивантажити, – це завдання планувальника.

Недолік даного способу – неефективне використання ОП, так як маленька програма цілком займає великий розділ. Це явище називається *внутрішньою фрагментацією.*

Розділи різного розміру. Цей спосіб не усуває повністю недолік, зазначений вище, але зменшує його вплив.

При призначенні розділу процесу існує два підходи:

кожен процес можна розмістити в найменшому розділі, здатному повністю вмістити даний процес; це мінімізує внутрішню фрагментацію, але для кожного розділу потрібна своя черга планувальника;

кожен процес можна розмістити в найменшому доступному розділі, здатному повністю вмістити даний процес; при цьому використовується єдина черга планувальника.

Недоліки методу фіксованого розподілу:

• кількість розділів, що визначена в момент генерації системи, обмежує кількість активних процесів;

• так як розміри розділів встановлюються заздалегідь, це призводить до неефективного використання пам'яті.

Фіксований розподіл пам'яті є не ефективним і тому не застосовується в сучасних ОС.

3.1.2 Динамічний розподіл

цьому випадку в пам'яті формується змінна кількість розділів змінної довжини, а при розміщенні процесу в ОП для нього виділяється строго необхідна кількість ОП. Цьому методу властива *зовнішня фрагментація*, тобто фрагментованою стає пам'ять, зовнішня по відношенню до всіх розділів. Для позбавлення від зовнішньої фрагментації ОС здійснює ущільнення: час від часу процеси переміщаються в пам'яті так, щоб вони займали суміжні області ОП. Для виконання такого ущільнення потрібен додатковий час і здатність програм до динамічного переміщення.

При організації динамічного розподілу використовується три основних алгоритми розміщення процесу в пам'яті:

найкращий підходящий-вибирається блок, розмір якого найбільш близький до необхідного;

перший підходящий – перевіряються всі вільні блоки з початку ОП і вибирається перший з них, достатній за розміром;

наступний підходящий – перевіряються всі вільні блоки, починаючи з того місця, де востаннє був виділений блок, і вибирається перший блок, достатній за розміром.

3.1.3 Система двійників

Як вже було сказано, фіксований розподіл обмежує кількість активних процесів і неефективно використовує ОП при невідповідності розмірів розділів процесів. Динамічний розподіл вимагає більш складної організації і великих накладних витрат на ущільнення пам'яті.

Система двійників є компромісом між цими двома способами. При її використанні ОП розподіляється блоками розміром $2k$, $L \le k \le U$, де $2L$-мінімальний розмір блоку, що виділяється, а $2U$-максимальний блок, що розподіляється (розмір всієї пам'яті, що доступний для розподілу).

Спочатку весь доступний для розподілу простір ОП розглядається як єдиний блок розміру $2U$. При запиті блоку довільного розміру s, якщо $2U-1 \le s \le 2U$, то виділяється весь блок. В іншому випадку цей блок розділяється на два блоки-двійники з розмірами $2U-1$. Якщо тепер $2U-2 \le s \le 2U-1$, то за запитом виділяється один з двох цих двійників, в іншому випадку один з двійників знову ділиться навпіл. Цей процес продовжується до тих пір, поки не буде згенеровано найменший блок, розмір якого не менше s.

При всьому цьому система постійно веде список доступних блоків для кожного розміру $2i$. Доступний блок може бути вилучений зі списку ($2i + 1$) поділом його навпіл і внесенням двох нових доступних блоків розміру $2i$ в список i. Коли ж пара двійників у списку i опиняється звільненою, вони видаляються зі списку і об'єднуються в єдиний блок в списку ($i + 1$).

Модифікована версія системи двійників використовується для розподілу пам'яті ядром ОС Linux.

3.1.4 Проста сторінкова організація

При використанні цього методу ОП розділяється на ряд кадрів рівного розміру.

Кожен процес також розділяється на сторінки рівного розміру, що збігається з розміром кадрів.

Процес завантажується шляхом завантаження всіх його сторінок в доступні, але не обов'язково суміжні кадри. При цьому відсутня зовнішня фрагментація, але є невелика внутрішня фрагментація, що є складовою частиною останньої сторінки процесу.

На рисунку 3.3 показано використання сторінок і кадрів.

№ кадру	Основна пам'ять	№ кадру	Основна пам'ять	№ кадру	Основна пам'ять
0	A.0	0	A.0	0	A.0
1	A.1	1	A.1	1	A.1
2	A.2	2	A.2	2	A.2
3	A.3	3	A.3	3	A.3
4	B.0	4	-	4	Д.0
5	B.1	5	-	5	Д.1
6	B.2	6	-	6	Д.2
7	C.0	7	C.0	7	C.0
8	C.1	8	C.1	8	C.1
9	C.2	9	C.2	9	C.2
10	C.3	10	C.3	10	C.3
11		11		11	Д.3
12		12		12	Д.4
13		13		13	
14		14		14	
а) Завантаження процесів A, B, C.		**б)** Вивантаження процесу B		**с)** Завантаження процесу Д	

Рисунок 3.3 – Завантаження процесів A, B, C, Д

Коли необхідно завантажити процес А (складається із чотирьох сторінок) ОС знаходить вільні кадри і завантажує туди сторінки процесу А.

Далі завантажується процес В (три сторінки) і процес С (чотири сторінки) (рисунок 3.3.-а).

За умовою процес Д має п'ять сторінок, тобто для завантаження йому не вистачає кадрів. Тому, процес В буде призупинено і вивантажено (рисунок 3.3.-б). Процеси блокуються і завантажується процес Д (рисунок 3.3.-с).

Як бачимо, для завантаження процесу при простій сторінковій організації пам'яті не обов'язковим є суміжне розташування кадрів, в які розміщуються сторінки одного процесу.

Для кожного процесу ОС підтримує таблицю сторінок, яка вказує розташування кадрів кожної сторінки процесу (рисунок 3.4). Також формується таблиця вільних кадрів.

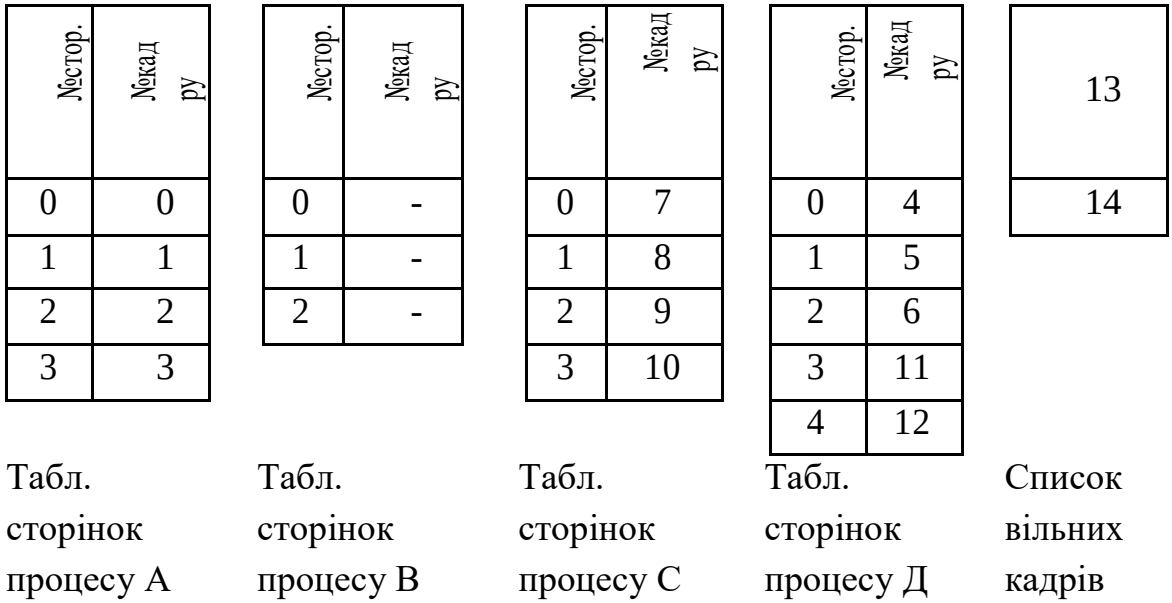


Рисунок 3.4 – Таблиці сторінок процесів

Таблиця сторінок містить по одному запису для кожної сторінки процесу. Це таблиця індексується номером сторінки починаючи з нуля, а кожний запис таблиці містить номер кадру в ОП. Операційна система підтримує єдиний список вільних кадрів.

програмі логічна адреса дорівнює номеру сторінки і зміщення всередині сторінки.

Фізична ж адреса дорівнює номеру кадру і зміщення.

Перетворення логічних адрес у фізичні – це завдання апаратного забезпечення.

На рисунку 3.5 наведено приклад трансляції логічної адреси з використанням простої сторінкової організації пам'яті для 16-бітових адрес.

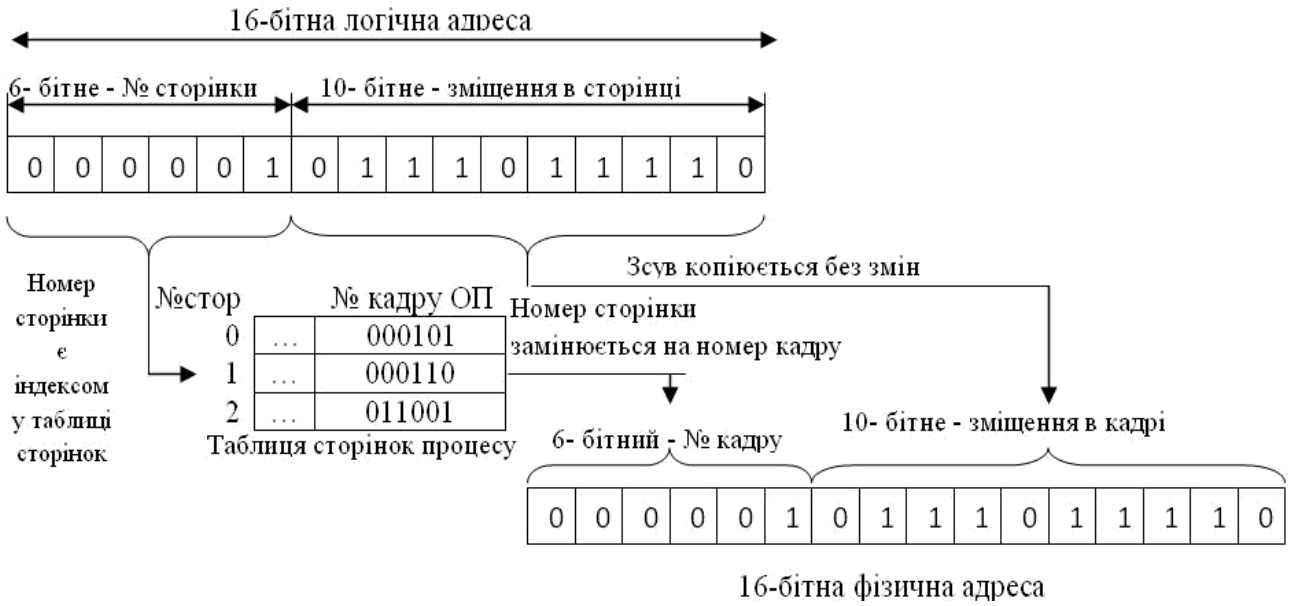


Рисунок 3.5 – Приклад трансляції логічних адрес в фізичну адресу при сторінковій організації пам'яті

Таким чином, проста сторінкова організація подібна фіксованому розподілу. Її відмінність полягає в малому розмірі розділів, які до того ж можуть бути несуміжними. Розмір сторінки повинен бути кратним ступеню числа 2.

3.1.5 Проста сегментна організація

Тут кожен процес поділяється на ряд сегментів і завантажується шляхом завантаження всіх його сегментів в динамічні (не обов'язково суміжні) розділи. Логічна адреса в цьому випадку дорівнює номеру сегмента і зміщення. При цьому, у порівнянні з динамічним розподілом, відсутня внутрішня фрагментація, краще використовується пам'ять і знижені накладні витрати.

Даний спосіб схожий на динамічний розподіл пам'яті. Якщо в ньому не використовуються оверлеї і віртуальна пам'ять, то для виконання програми всі сегменти повинні бути завантажені в ОП. Відмінність же від динамічного розподілу полягає в тому, що сегменти програми можуть займати кілька розділів, до того ж несуміжних.

Цей спосіб усуває внутрішню фрагментацію, але породжує зовнішню фрагментацію. Сегментація при його використанні прозора для програміста і застосовується при розміщенні коду і даних в різних сегментах для організації декількох незалежних адресних просторів. Як і при сторінковій організації, тут

використовується таблиця сегментів для кожного процесу і ведеться список вільних блоків ОП. Кожен запис таблиці сегментів при цьому повинен містити стартову адресу сегмента в ОП і його довжину.

Приклад трансляції адреси з використанням простої сегментної організації пам'яті для 16-бітових адрес показаний на рисунку 3. 6.

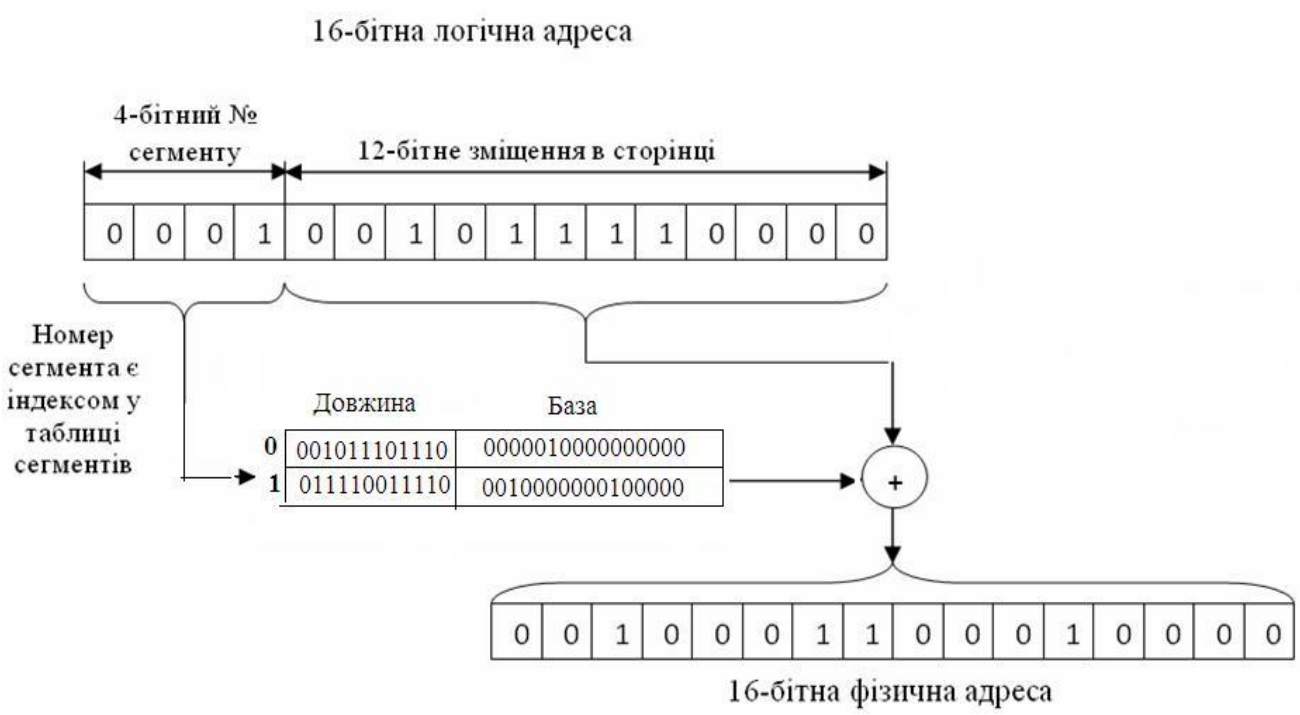


Рисунок 3.6 – Приклад трансляції фізичних адрес в логічну адресу при сегментації

3.2 Віртуальна пам'ять

3.2.1 Передумови для організації віртуальної пам'яті

Оскільки процес виконується тільки в основній пам'яті, ця пам'ять також називається *реальною*. Однак програміст або користувач мають справу з потенційно більшою за обсягом пам'яттю, виділеною на диску. Ця пам'ять називається ***віртуальною***[1].

Вперше сповіщення про віртуальну пам’ять на основі сторінкової організації з’явилося у 1962 році в роботі Kilburn I та ін. "*One-Level Storage System*" і незабаром після цього віртуальна пам’ять стала широко застосовуватись в комерційних системах.

Передумови для організації віртуальної пам'яті.

Всі звернення до пам'яті в рамках процесу являють собою логічні адреси, які динамічно транслюються в фізичні адреси під час виконання. Це означає, що процес може бути вивантажений на диск і знову завантажений в ОП, так що під час роботи він може перебувати в різних місцях ОП.

Процес може бути розбитий на кілька частин (сторінок або сегментів), які не обов'язково повинні розташовуватися в ОП єдиним безперервним блоком. Це забезпечується завдяки використанню динамічної трансляції адрес, таблиць сторінок або сегментів.

При виконанні цих умов одночасна наявність всіх сторінок або сегментів процесу в ОП не є необхідним.

Розглянемо, як це можна здійснити. Припустимо, що настав час завантаження нового процесу в пам'ять. ОС починає таке завантаження з розміщення в пам'яті тільки одного або декількох блоків (тут під блоком ми матимемо на увазі сторінку або сегмент), включаючи блок, що містить початок програми.

Частина процесу, що розташовується в деякий момент часу в ОП, називається *резидентною множиною процесу*. Далі, під час виконання процесу за допомогою таблиці сторінок або сегментів процесор завжди може визначити, чи розташовується блок, до якого потрібно звернення, в ОП. Якщо ні, то процесор генерує переривання, що свідчить про відсутність блоку в пам'яті; тоді ОС переводить перерваний процес в заблокований стан і завантажує в ОП блок, що містить адресу, через яку відбулося переривання.

Після того, як необхідний блок завантажений в ОП, виконується переривання введення/виведення, що передає керування операційній системі, яка, в свою чергу, переводить заблокований процес в стан готовності, і процес продовжує виконання з перерваної команди.

Переваги використання віртуальної пам'яті:

в оперативній пам'яті може підтримуватися більша кількість процесів;

процес може бути більше, ніж обсяг всієї фізичної пам'яті.

таблиці 3.1. порівняно переваги і недоліки всіх видів організації пам'яті.

Таблиця 3.1 – Порівняння різних технологій розподілу пам’яті

Технологія	Опис	Переваги	Недоліки
Фіксований розподіл	Основна пам'ять розділяється на ряд статичних розділів під час генерації системи. Процес може бути завантажений в розділ рівного або більшого розміру	Простота реалізації, малі системні накладні витрати	Неефективне використання пам'яті через внутрішню фрагментацію, фіксована максимальна кількість активних процесів
Динамічний розподіл	Розділи створюються динамічно, кожен процес завантажується в розділ строго необхідного розміру	Відсутня внутрішня фрагментація, більш ефективне використання основної пам'яті	Неефективне використання процесора через необхідність ущільнення для протидії зовнішній фрагментації
Проста сторінкова організація	Основна пам'ять розподілена на ряд кадрів рівного розміру. Кожен процес розподілений на деяку кількість сторінок рівного розміру тієї ж довжини, що і кадри. Процес завантажується шляхом завантаження всіх його сторінок в доступні, але не обов'язково послідовні, кадри	Відсутня зовнішня фрагментація	Є невелика внутрішня фрагментація
Проста сегментація	Кожний процес розподілено на ряд сегментів. Процес завантажується шляхом завантаження всіх своїх сегментів в динамічні (не обов’язково суміжні) розділи	Відсутня внутрішня фрагментація	Покращене використання пам’яті и знижені накладні витрати в порівнянні з динамічним розподіленням
Сторінкова організація віртуальної пам’яті	Все, як при простій сторінковій організації, з тим винятком, що не потрібно одночасно завантажувати всі сторінки процесу. Необхідні нерезиденті сторінки автоматично завантажуються в пам'ять	Немає зовнішньої фрагментації; більш високий ступінь багатозадачності; великий віртуальний адресний простір	Накладні витрати через складність системи управління пам'яттю

Технологія	Опис	Переваги	Недоліки
Сегментація віртуальної пам'яті	Все, як при простій сегментації, з тим винятком, що не потрібно одночасно завантажувати всі сегменти процесу. Необхідні нерезиденті сегменти автоматично завантажуються в пам'ять	Немає внутрішньої фрагментації; більш високий ступінь багатозадачності; великий віртуальний адресний простір; підтримка захисту і спільного використання	Накладні витрати через складність системи управління пам'яттю

Коли використовується віртуальна пам'ять, то віртуальні адреси не передаються напряму шиною пам'яті. Замість цього вони передаються диспетчеру пам'яті (*MMU – Memory Management* Unit), який відображує віртуальні адреси на фізичні адреси пам'яті (рисунок 3.7). Тут диспетчер пам'яті показаний як частина мікросхеми процесора.

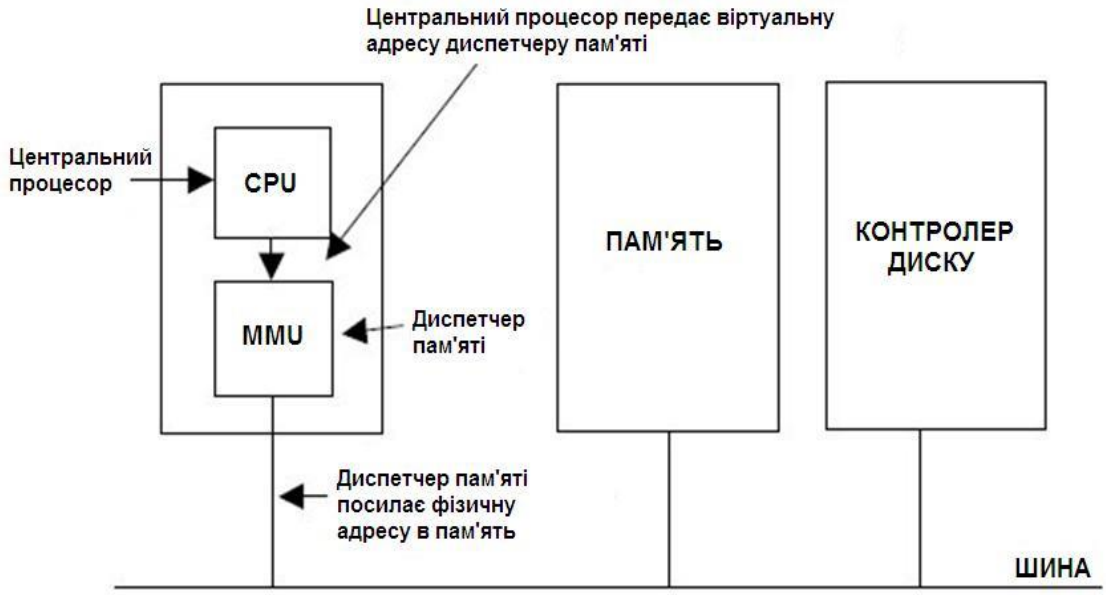


Рисунок 3.7 – Диспетчер пам'яті

теперішній час реалізація віртуальної пам'яті може бути організована різними способами, що відрізняються в основному способом структуризації віртуального адресного простору. Виділяють три методи реалізації віртуальної пам'яті:

1. Сторінкова віртуальна пам'ять – організує переміщення даних між основною пам'яттю і диском сторінками – частинами віртуального адресного простору фіксованого і порівняно невеликого розміру.

Сегментна віртуальна пам'ять – передбачає переміщення даних сегментами - частинами віртуального адресного простору довільного розміру.

Сегментно-сторінкова віртуальна пам'ять – використовує дворівневий поділ, при якому віртуальний адресний простір поділяється на сегменти, а потім сегменти поділяються на сторінки. Одиницею переміщення даних є сторінка.

Для тимчасового зберігання сегментів і сторінок на диску відводиться спеціальна область або спеціальний файл (сторінковий файл або файл підкачки – paging file). Поточний розмір сторінкового файлу є важливим параметром, що впливає на можливості ОС: чим більший сторінковий файл, тим більше додатків може одночасно виконувати ОС (при фіксованому розмірі оперативної пам'яті). Але при збільшенні числа додатків, що одночасно працюють збільшується і розмір сторінкового файлу, а це сповільнює їх роботу, так як значна частина часу витрачається на переміщення даних на диск і в зворотному напрямку.

Розмір сторінкового файлу в сучасних ОС є таким параметром, що налаштовується, і може бути визначений адміністратором системи для досягнення компромісу між рівнем програмування і швидкодією системи.

3.2.2 Сторінкова організація віртуальної пам'яті

Термін «віртуальна пам'ять» зазвичай асоціюється з системами, що використовують сторінкову організацію, хоча використовується і віртуальна пам'ять на основі сегментації.

Структура таблиці сторінок.

При розгляді простої сторінкової організації ми вказували, що кожен процес має свою таблицю сторінок, що створюється операційною системою. Кожен запис в цій таблиці сторінок містить номер кадру відповідної сторінки в пам'яті. Таблиця сторінок розташовується в ОП.

Для організації віртуальної пам'яті кожний запис таблиці сторінок обов'язково має містити *біт присутності P*, що вказує на присутність відповідної сторінки в ОП; якщо ця сторінка розташовується в ОП, то в записі таблиці також міститься номер її кадру.

Іншим керуючим бітом в записі таблиці сторінок є *біт модифікації M*. Він вказує, чи було змінено вміст даної сторінки з часу її останнього завантаження в ОП. Якщо таких змін не було, то при настанні часу заміни сторінки в займаному нею в даний момент кадрі записувати цю сторінку на диск не потрібно, так як на диску вже є її точна копія.

Базовий механізм читання слова з пам'яті включає в себе трансляцію віртуальної, або логічної адреси, що складається з номера сторінки і зміщення у фізичну адресу, яка являє собою номер кадру і зсув, з використанням таблиці сторінок.

принципі, кожна віртуальна адреса викликає звернення до двох фізичних адрес: одне звернення потрібно для вибірки відповідного запису з таблиці сторінок і ще одне - для власне звернення до даних, що адресуються. Отже, проста схема віртуальної пам'яті, по суті, подвоює час звернення до ОП. Для подолання цієї проблеми більшість схем віртуальної пам'яті використовує спеціальний високошвидкісний кеш для записів таблиці сторінок, який зазвичай називається буфером швидкого перетворення адреси, або просто буфером пошуку трансляції (TLB, Translation Lookaside Buffer). Цей кеш функціонує так само, як і звичайний кеш пам'яті.

3.2.3 Сегментація

Сегментація дозволяє програмісту розглядати пам'ять як область, що складається з безлічі адресних просторів, або сегментів, які можуть мати різні (фактично – динамічні) розміри. Звернення до пам'яті використовують адреси, що представляють собою пари: (номер сегмента, зміщення).

Така організація має ряд переваг у порівнянні з несегментованим адресним простором:

спрощується обробка зростаючих структур даних;

спрощується спільне використання коду і даних різними процесами;

покращується захист даних.

Як відомо, при простій сегментній організації кожен процес має власну таблицю сегментів, у кожному записі якої зазначена початкова адреса відповідного сегмента в ОП і його довжина. Та ж таблиця сегментів потрібна і при схемі віртуальної пам'яті, заснованої на сегментації, однак структура записів таблиці сегментів в цьому випадку ускладнюється.

Оскільки в ОП можуть знаходитися не всі сегменти процесу, в кожному записі потрібна наявність *біта присутності*, який вказує на те, чи розташовується даний сегмент в ОП. Якщо сегмент розташований в ОП, то запис також включає його початкову адресу і довжину.

Ще один біт, необхідний в даній схемі, – *біт модифікації*, який вказує, чи було змінено вміст сегменту з часу його останнього завантаження в ОП. Якщо змін не було, то при вивантаженні сегмента немає необхідності в його запису на диск.

Основний механізм читання слова з пам'яті включає в себе перетворення віртуальної, або логічної, адреси, що складається з номера сегмента і зміщення,

фізичну адресу з використанням таблиці сегментів, для зберігання якої використовується ОП. Коли запускається певний процес, в одному з регістрів зберігається стартова адреса його таблиці сегментів. Номер сегмента з віртуальною адресою використовується в якості індексу таблиці, що дозволяє визначити початкову адресу сегменту. Для отримання ж фізичної адреси до початкової адреси сегмента додається зсув з віртуальної адреси.

3.2.4 Комбінація сегментації і сторінкової організації

сторінкова організація, і сегментація мають свої переваги. Сторінкова організація, прозора для програміста, усуває зовнішню фрагментацію і, таким чином, забезпечує ефективне використання ОП. Так як переміщувані в ОП і з неї блоки мають фіксований, однаковий розмір, то полегшується створення ефективних алгоритмів управління пам'яттю. Сегментація ж, є видимою для програміста і має свої переваги.

Деякі обчислювальні системи використовують переваги обох методів.

таких системах адресний простір користувача розбивається на ряд сегментів на розсуд програміста. Кожен сегмент, у свою чергу, розбивається на ряд сторінок фіксованого розміру, відповідного до розміру кадру ОП. Якщо розмір сегмента менше за розмір сторінки, то він займає сторінку цілком. З точки зору програміста, логічна адреса в цьому випадку складається з номера сегмента і зміщення в ньому. З позиції ж ОС зсув в сегменті слід розглядати як номер сторінки певного сегмента і зміщення в ній.

3.2.5 Алгоритми управління віртуальною пам'яттю

Алгоритми (стратегії) операційної системи, пов'язані з управлінням віртуальною пам'яттю, діляться на наступні групи.

Стратегія вибірки. Стратегія вибірки визначає, коли сторінка повинна бути передана в ОП. При цьому можливі два основні варіанти: *на вимогу і попередньо*. При вибірці *за вимогою* сторінка передається в ОП тільки тоді, коли виконується звернення до комірки пам'яті, розташованої на цій сторінці.

разі ж *попередньої вибірки* завантажується не тільки сторінка, що викликала переривання звернення. Якщо сторінки процесу розташовані у вторинній пам'яті послідовно, то набагато більш ефективною буде завантаження в ОП декількох послідовних сторінок за один раз, ніж завантаження цих же сторінок по одній протягом певного проміжку часу.

Стратегія розміщення. Стратегія розміщення визначає, де саме у фізичній пам'яті будуть розташовуватися частини процесу. У разі «чистої» сегментації стратегія розміщення використовує алгоритми першого відповідного, наступного, що підходить та інші. Однак для систем, що використовують тільки сторінкову організацію або сторінкову організацію в поєднанні з сегментацією, стратегія розміщення зазвичай не так важлива, оскільки апаратна трансляція адреси і апаратне звернення до пам'яті однаково результативні при будь-яких поєднаннях «сторінка-кадр».

Стратегія заміщення. Стратегія заміщення відповідає за вибір сторінок

ОП для їх заміщення сторінками, що завантажуються з вторинної пам'яті. Ця стратегія включає в себе ряд взаємопов'язаних питань: 1) яка кількість кадрів має бути виділена кожному активному процесу; 2) чи має безліч сторінок, які потенційно можуть бути заміщені сторінками, що завантажуються обмежуватися одним процесом, або в якості кандидатів на заміщення можуть розглядатися всі кадри ОП; 3) які саме сторінки з розглянутої безлічі слід вибрати для заміщення.

Коли всі кадри ОП зайняті, а нам потрібно розмістити нову сторінку, стратегія заміщення визначає, яка з сторінок, що знаходяться в даний час в ОП повинна бути вивантажена, щоб звільнити кадр для сторінки, що завантажується. Всі варіанти стратегії при цьому спрямовані на те, щоб вивантажити сторінку, звернень до якої в найближчому майбутньому не буде.

відповідності з принципом локалізації, часто спостерігається сильна кореляція між сторінками, до яких останнім часом виконувалися звернення, і сторінками, до яких будуть звернення найближчим часом. Більшість стратегій намагаються визначити майбутню поведінку програми на основі її минулої поведінки. Однак при розгляді різних стратегій слід враховувати, що чим більш досконалий і інтелектуальний алгоритм використовує стратегія, тим вище будуть накладні витрати при його реалізації.

Оптимальний алгоритм полягає у виборі для заміщення тієї сторінки, звернення до якої буде виконуватися через найбільший проміжок часу в порівнянні з усіма іншими сторінками. Цей алгоритм призводить до мінімальної кількості переривань через відсутність сторінок. Зрозуміло, що реалізувати такий алгоритм на практиці неможливо, оскільки для цього системі потрібно знати всі майбутні події. Однак цей алгоритм є стандартом, з яким порівнюються реальні алгоритми.

Стратегія довше за всіх невикористовуваного елементу *(LRU, least Recently Used)* заміщає в пам'яті ту сторінку, звернень до якої не було довше, ніж до інших сторінок: згідно з принципом локалізації, можна очікувати, що ця сторінка не буде використовуватися і в найближчому майбутньому. Ця стратегія близька до оптимальної. Основна ж проблема полягає в складності її реалізації.

Стратегія «першим увійшов – першим вийшов» розглядає кадри сторінок процесу як циклічний буфер з циклічним же видаленням сторінок з нього. Це одна з найпростіших у реалізації стратегій заміщення, яка заміщає сторінку, що знаходиться в ОП довше інших. Однак далеко не завжди ця сторінка використовується дійсно рідко.

Годинникова стратегія є компромісом між стратегією довше за всіх невикористовуваного елемента (близькою до оптимальної, але складною в реалізації) та стратегією «першим увійшов – першим вийшов» (простою в реалізації, але далекою від оптимальної). У найпростішій схемі годинникової стратегії з кожним кадром зв'язується один додатковий біт, що носить назву *«біт використання».* Коли сторінка вперше завантажується в кадр, її біт використання встановлюється рівним 1. При наступних зверненнях до сторінки цей біт також встановлюється рівним 1. При роботі алгоритму заміщення безліч кадрів розглядається як циклічний буфер, з яким пов'язаний покажчик, і при заміщенні сторінки покажчик переміщається до наступного кадру в буфері.

Коли настає час заміщення сторінки, ОС сканує буфер для пошуку кадру, біт використання якого дорівнює 0, і всякий раз, коли в процесі цього пошуку зустрічається кадр з бітом використання, рівним 1, він скидається в 0. Для заміщення вибирається перший, що зустрінеться кадр з нульовим бітом використання. Якщо ж всі кадри мають біт використання, рівний 1, то покажчик робить повний круг і повертається до початкового положення, замінюючи сторінку в цьому кадрі.

3.2.6 Управління пам'яттю в сучасних версіях Windows

Система керування віртуальною пам'яттю Windows контролює розподілення пам'яті і роботу сторінкової організації. Диспетчер пам'яті сконструйований для роботи на багатьох платформах і використовує сторінки розміром від 4 до 64 Кбайт. Платформи Intel і AMD 64 мають сторінки по 4 Кбайт, а платформа Intel Itanium має сторінки по 8 Кбайт.

Кожний користувацький процес Windows отримує окремий 32-бітовий адресний простір, що надає процесу до 4 Гбайт пам'яті. За замовчуванням частина пам'яті зарезервована для операційної системи, так, що користувачу доступні 2 Гбайт віртуального адресного простору, і всі процеси сумісно використовують одні і ті ж самі 2 Гбайт системного простору.

Великі доданки, що інтенсивно використовують пам'ть (як клієнти так і сервери) можуть запускатися більш ефективно в 64-бітному Windows. Окрім нетбуків найбільш сучасні персональні комп'ютери використовують архітектуру процесора AMD 64, який може запускатися і в 32-бітній і в 64-бітній системах.

При створенні процесу в принципі йому повністю доступний простір користувача розміром 2 Гбайта (або 8 Тбайт в 64-бітному Windows). Цей простір розподіляється на сторінки фіксованого розміру, кожна із яких може бути завантажена в ОП.

Схема управлінням віртуальною пам'яттю вимагає як програмної, так і апаратної підтримки. Апаратна підтримка, що забезпечується процесором, включає динамічне перетворення віртуальних адресів в фізичні і генерацію переривання при відсутності сторінки, що адресується або сегменту в ОП. Це переривання обробляється програмним забезпеченням управлінням пам'яттю.

Контрольні питання

Яку пам'ять називають основною пам'яттю (*main memory*)?

Що таке логічна і фізична адреси? Як відбувається перетворення логічної адреси у фізичну?

Що таке Memory Management Unit?

Які функції мультипрограмної ОС по управлінню пам'яттю?

Які є види фіксованого розподілу пам'яті? Чому такий розподіл пам'яті в чистому вигляді не використовується?

Коли виникає зовнішня фрагментація пам'яті? Яким чином ОС зменшує негативний ефект зовнішньої фрагментації?

Що являє собою проста сторінкова організація основної пам'яті? Чому при такій організації пам'яті відсутня зовнішня фрагментація і зведена до мінімуму внутрішня?

Опишіть, як відбувається завантаження сторінок процесів у кадри пам'яті?

Що являє собою таблиця сторінок процесу? Навіщо створюються таблиці сторінок?

З чого складається логічна адреса при сторінковому розподілу пам'яті?

Опишіть схему перетворення логічної адреси у фізичну?

З чого складається логічна адреса при сегментному розподілу пам'яті?

Опишіть схему перетворення логічної адреси у фізичну?

Що таке віртуальна пам'ять? Коли вперше стало відомо про віртуальну пам'ять?

Які передумови виникнення технології віртуальної пам'яті? Які переваги і недоліки використання віртуальної пам'яті?

Які є методи реалізації віртуальної пам'яті? Якому з них надається найбільша перевага і чому?

Які керуючі біти мають бути в кожному запису таблиці сторінок при віртуальній пам'яті?

Які є алгоритми (стратегії) управління віртуальною пам'яттю?

Опишіть алгоритм управління віртуальною пам'яттю «Стратегія вибірки».

Чому «Оптимальний алгоритм» реалізувати на практиці неможливо?

Яке розподілення пам’яті контролює система керування віртуальною пам’яттю Windows?

Який розмір простору користувача в 64-бітному Windows?

РОЗДІЛ 4

СИСТЕМА ВВЕДЕННЯ/ВИВЕДЕННЯ. ФАЙЛОВА СИСТЕМА[1]

4.1 Система введення/виведення

Основні принципи програмного забезпечення введення/виведення.

Незалежність від пристроїв. Цей принцип означає можливість написання програм, здатних отримувати доступ до будь-якого пристрою введення/виведення без попередньої вказівки конкретного пристрою.

Однакове іменування. Ім'я файлу або пристрою повинно являти собою просто текстовий рядок або ціле число і ніяк не повинно залежати від фізичного пристрою.

Способи здійснення введення/виведення.

Програмний ввід-вивід. Якщо, наприклад, потрібно роздрукувати на принтері рядок символів, то операційна система спочатку копіює ці дані в ядро. Потім ОС входить в цикл, на кожній ітерації якого друкує на принтері один символ. Після друку кожного символу процесор в циклі опитує готовність пристрою. Така поведінка процесора називається опитуванням, або активним очікуванням. Програмний ввід-вивід легко реалізується, але його недолік полягає в тому, що центральний процесор зайнятий на весь час операціями введення/виведення.

Ввід-вивід, керований перериваннями. Коли виконується системний виклик друку рядка, буфер копіюється в простір ядра, а перший символ рядка копіюється на принтер. Після цього центральний процесор викликає планувальник, який запускає на виконання якій-небудь інший процес. Після того як принтер надрукує символ і буде готовий прийняти наступний, принтер ініціює відповідне переривання. Воно перехоплюється обробником переривання і сигналізує ОС, що вивід символу завершений. Після цього ОС посилає на друк наступний символ, і т.д. Таким чином, під час друку символу на принтері процесор не зайнятий опитуванням готовності пристрою друку, а виконує іншу роботу.

Ввід-вивід з використанням прямого доступу до пам'яті (*DMA, Direct Memory Access*). Очевидний недолік введення/виведення керованого перериваннями, полягає в тому, що переривання відбуваються при друку

---

[1] При підготовці даного розділу були використані літературні джерела [1–4], [9–11], [12], [15], [17], [21], [22] та ін.

кожного символу. Обробка цих переривань займає певний час, тому така схема не є ефективною. Рішення проблеми полягає у використанні DMA. Контролер DMA поставляє принтеру символи по одному, не займаючи при цьому центральний процесор. По суті, цей метод майже не відрізняється від програмного введення/виведення, з тією лише різницею, що всю роботу замість центрального процесора виконує контролер DMA. Найбільший виграш від використання DMA полягає в зменшенні кількості переривань з одного на кожний друкований символ до одного на весь друкований буфер.

4.1.2 Програмні рівні введення/виведення

Програмне забезпечення введення/виведення зазвичай організовано у вигляді чотирьох рівнів, показаних на рисунку 4.1, де у кожного рівня є свої функції і строго визначений інтерфейс з сусідніми рівнями.

Рисунок 4.1 – Програмні рівні введення/виведення

Обробники переривань. Коли відбувається переривання, починає роботу обробник переривань. Після виконання необхідної роботи він може розблокувати драйвер, що запустив його.

Нижче наведені основні дії, що виконуються програмним забезпеченням після того, як відбулося апаратне переривання:

збереження всіх регістрів, що не збереглися апаратурою;

установка контексту для процедури обробки переривань (виконання цієї дії може включати встановлення TLB і таблиці сторінок);

установка покажчика стека для процедури обробки переривань;

4) видача підтвердження контролеру переривань (якщо ж централізованого контролера переривань немає, то дозвіл переривань);

копіювання вмісту регістрів в таблицю процесів;

запуск процедури обробки переривань, яка видобуває інформацію з регістрів контролера пристрою, що ініціював переривання;

вибір процесу, якому слід передати управління;

установка контексту блоку управління пам'яттю для наступного працюючого процесу;

завантаження регістрів для нового процесу;

запуск на виконання нового процесу.

Драйвери пристроїв. У кожного контролера є набір регістрів, які використовуються, для того щоб надавати команди певному пристрою і зчитувати стан цього пристрою. Команди, що видаються пристрою, залежать від конкретного типу цього пристрою, тому для управління кожним пристроєм введення/виведення потрібна спеціальна програма. Ця програма, називається драйвером пристрою і зазвичай пишеться виробником пристрою, причому для кожної операційної системи потрібні спеціальні драйвери. Кожен драйвер зазвичай підтримує тільки один тип пристроїв або ж клас близьких за типами пристроїв.

Щоб отримати доступ до регістрів контролера, драйвер пристрою має бути частиною ядра ОС. Драйвери зазвичай розташовуються під всією іншою частиною операційної системи.

ОС зазвичай класифікує драйвери по декількох категоріях у відповідності типами пристроїв, що ними обслуговуються. До найбільш загальних категорій відносяться *блокові пристрої* (наприклад, диски, що містять блоки даних), до яких можлива незалежна адресація, і *символьні пристрої*, такі як клавіатура і принтери, які формують або приймають потік символів.

ОС визначені стандартні інтерфейси, підтримувані всіма драйверами. Ці інтерфейси включають в себе набір процедур, які можуть викликатися із операційної системи для звернення до драйверу.

Драйвер пристрою повинен виконувати кілька функцій:

- обробку абстрактних запитів читання і запису, що надходять від незалежного від цього пристрою програмного забезпечення, розташованого над ним;
- перевірку вхідних параметрів;

перетворення абстрактних параметрів в конкретні (наприклад, дисковий драйвер може перетворювати лінійний номер блоку в номери головки, доріжки і сектора);

перевірку готовності і незайнятості пристрою;

власне управління пристроєм – видачу йому необхідної серії команд;

у багатьох випадках драйвер пристрою повинен чекати, поки контролер пристрою не виконає для нього певну роботу, тому він блокується до тих пір, поки переривання від пристрою його не розблокує; в інших випадках операція завершується без затримок, і драйверу не потрібно блокуватися;

по завершенні виконання операції драйвер повинен перевірити, чи виконана ця операція без помилок. Якщо все в порядку, то драйвер передає дані незалежного від пристроїв програмного забезпечення.

Незалежне від пристроїв програмне забезпечення введення/виведення. Незалежне від пристроїв програмне забезпечення введення/виведення забезпечує наступні функції:

однаковий інтерфейс для драйверів пристроїв;

буферизацію;

повідомлення про помилки;

захоплення і звільнення виділених пристроїв;

розмір блоку, не залежний від пристрою.

Програмне забезпечення введення/виведення простору користувача. Програмне забезпечення введення/виведення простору користувача складається бібліотек, приєднаних до програм користувача. Системні виклики вводу зазвичай складаються з бібліотечних процедур. Наприклад, якщо програма на мові С містить виклик *write*, то бібліотечна процедура *write* буде скомпонована з програмою і, відповідно, буде міститися в виконуваному файлі.

Хоча більшість таких процедур в основному здійснюють звернення до системного виклику з відповідними параметрами, але є й ряд процедур введення/виведення, що виконують певну роботу. Зокрема, бібліотечними процедурами виконуються операції форматного вводу і виводу (приклад - функція *printf* мови *C*).

4.2 Завдання файлової системи. Поняття файлу

4.2.1 Фізична організація файлової системи

Файл (file) – це поіменована область зовнішньої пам'яті, в яку можна записувати і з якої можна зчитувати дані (*суміжна область логічного адресного простору*). Основні цілі використання файлу:

довгострокове і надійне зберігання інформації;

спільне використання інформації.

Кожен файл має свій тип, що визначає, яка інформація зберігається в файлі. Основні типи файлів: програма (код) або дані. Дані підрозділяються на числові, символьні (текстові) і двійкові (довільна інформація).

Структура файлу

різних системах прийняті різні точки зору на структуру файлів. У ряді систем структура файлу прив'язувалася до типу пристрою, на якому він знаходиться. У деяких інших системах структура файлу була штучно ускладнена. Однак найбільш просту і уніфіковану точку зору з них запропонували автори системи UNIX: файл – це послідовність слів або байтів. Здавалося б, це очевидно, але перевага даного підходу до файлів в тому, що базове уявлення файлу і базові операції над ним (read, write) не залежать від типу пристрою. Можна сказати, що файли у своєму розвитку пройшли шлях, аналогічний розвитку архітектури комп'ютерів – спочатку в бік значних ускладнень, потім – спрощення та уніфікації.

Файли можна умовно підрозділяти на файли *простої та складної структури* (хоча точка зору на структуру файлу залежить від тієї програми, яка його обробляє).

Файли простої структури складаються з послідовності записів (*records*) – елементарних одиниць, в термінах яких виконуються операції обміну з файлом. Записи можуть бути: рядками, якщо це текстовий файл; двійковими даними фіксованої довжини; двійковими даними змінної довжини.

Файли складної структури можуть бути самого різного виду, наприклад: відформатованим документом Microsoft Office (такий файл, окрім власне тексту, містить керуючі символи перемикання шрифтів, кольорів і т.д.); завантажувальним модулем реального або віртуального двійкового коду, наприклад, portable executable (PE)-файлом для платформи NET; class-файлом для платформи Java; подібні файли складаються з декількох секцій, містять внутрішні посилання і таблиці і т.д.

Складна структура файлу може бути змодельована записами шляхом додавання відповідних керуючих символів.

Файли інтерпретуються операційною системою або програмами їх обробки.

Файлова система (ФС) – це частина операційної системи, що включає в себе:

сукупність всіх файлів на диску;

набори структур даних, використовуваних для управління файлами;

комплекс системних програмних засобів, що реалізують операції над файлами.

ФС грає роль «проміжного шару», що усуває для користувача/програміста всі складнощі фізичної організації довготривалого сховища даних і створює для програм більш просту логічну модель цього сховища, а також надає їм набір зручних у використанні команд для маніпулювання файлами.

Завдання, розв'язувані файловою системою, залежать від способу організації обчислювального процесу в цілому.

Основні функції ФС в однокористувацьких і однопрограмних ОС (типу MS-DOS):

іменування файлів;

програмний інтерфейс для додатків;

відображення логічної моделі файлової системи на фізичну організацію сховища даних;

стійкість файлової системи до збоїв харчування, помилок апаратних і програмних засобів.

однокористувацьких мультипрограмних ОС також мають бути передбачені засоби для блокування файлу і його частин, запобігання гонок, виключення тупиків, узгодження копій і т. п.

багатокористувацьких ОС з'являється ще одна задача – захист файлів одного користувача від несанкціонованого доступу іншого користувача.

Нарешті, ще більш складними стають функції ФС, яка працює у складі мережевої ОС.

Принципи розміщення файлів, каталогів і системної інформації на реальному пристрої описуються фізичною організацією файлової системи. Різні ФС мають різну фізичну організацію.

4.2.2 Будова диску

Основним типом пристрою для зберігання файлів є дискові накопичувачі. Ці пристрої призначені для зчитування і запису даних на жорсткі і гнучкі магнітні диски.

Жорсткий диск складається з однієї або декількох скляних або металевих пластин, кожна з яких покрита з одного або з двох сторін магнітним матеріалом. На кожній стороні кожної пластини шляхом відповідного намагнічування поверхні магнітного шару розмічені тонкі концентричні кільця – доріжки (tracks), на яких зберігаються дані (рисунок 4. 2).

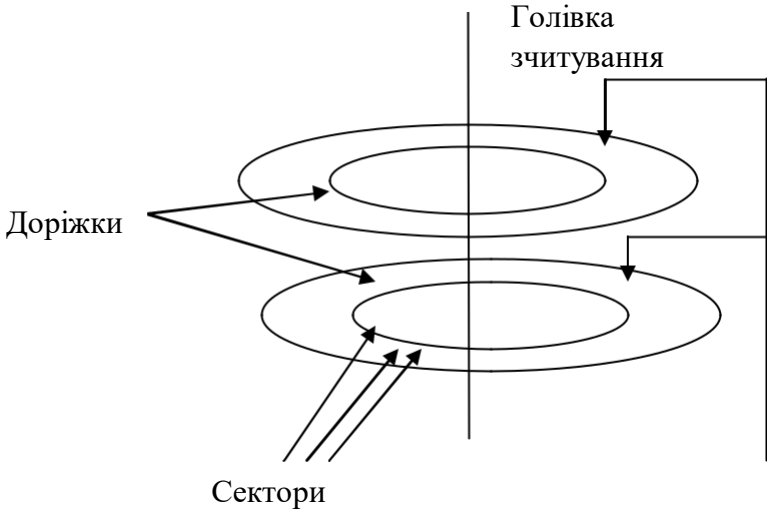


Рисунок 4.2 – Схема устрою накопичуваних жорстких магнітних дисків

Коли диск обертається, магнітна головка накопичувача зчитує двійкові дані з обраної магнітної доріжки або записує їх на магнітну доріжку. Для цього голівка може позиціонуватися над заданою доріжкою. Зазвичай всі головки закріплені на єдиному механізмі, що переміщується і рухаються синхронно. Тому, коли голівка фіксується на заданій доріжці однієї поверхні, всі інші голівки зупиняються над доріжками інших пластин з такими ж номерами. Сукупність доріжок одного і того ж радіусу на всіх поверхнях всіх пластин дискового пакета називається *циліндром*.

Кожна доріжка розбивається на фрагменти, звані секторами, або блоками, так що всі доріжки мають рівну кількість секторів, в які можна максимально записати одну і ту ж кількість байтів. Сектор має фіксований для конкретної ОС розмір, що виражається ступенем числа 2; найчастіше розмір сектора складає 512 байт. Враховуючи ж, що доріжки різного радіусу мають однакову

кількість секторів, щільність запису стає тим більшою, чим ближче розташовується доріжка до центру диска.

Сектор – це найменша адресуєма одиниця обміну даними дискового пристрою з оперативною пам'яттю. Щоб контролер міг знайти на диску потрібний сектор, необхідно задати йому всі складові адреси цього сектора: номер циліндра, номер поверхні, номер доріжки та власне номер сектора на цій доріжці.

ОС при роботі з диском використовує, як правило, власну одиницю дискового простору, звану *кластером* (в UNIX кластер називається блоком). При створенні файлу місце на диску йому виділяється кластерами. Доріжки і сектори створюються в результаті виконання процедури фізичного, або *низькорівневого форматування*, що передує використанню диска. Для визначення меж блоків на диск записується ідентифікаційна інформація. Низькорівневий формат диска не залежить від типу ОС, яка цей диск буде використовувати. Розмітку диска під конкретний тип файлової системи виконують процедури високорівневого, або *логічного форматування*. При високорівневому форматуванні визначається розмір кластера, а на диск записується інформація, необхідна для роботи файлової системи, у тому числі відомості про доступний і невикористаний простір, про межі областей, відведених під файли і каталоги, про пошкоджені області. Крім того, на диск записується *завантажувач операційної системи* – невелика програма, яка починає процес ініціалізації операційної системи після включення живлення або перезавантаження комп'ютера.

Перш ніж форматувати диск під певну файлову систему, його можна розбити на розділи. *Розділ* – це безперервна частина фізичного диска, яку операційна система являє користувачу як окремий логічний пристрій (часто використовуються також назви «логічний диск» і «логічний розділ»). Логічний пристрій функціонує так, ніби то це окремий фізичний диск. ОС різного типу зазвичай використовують єдине для всіх них уявлення про розділи, але створюють на його основі логічні пристрої, специфічні для кожного типу ОС. Так само як файлова система, з якою працює одна ОС, в загальному випадку не може інтерпретуватися ОС іншого типу, одні й ті ж логічні пристрої часто не можуть бути використані операційними системами різного типу. При цьому на кожному логічному пристрої може створюватися лише одна файлова система, на різних логічних пристроях одного і того ж фізичного диска можуть розташовуватися файлові системи різного типу.

Файлова система FAT.

Абревіатура FAT (file allocation table) розшифровується як «таблиця розміщення файлів». Цей термін відноситься до лінійної табличної структури

відомостями про файли — іменами файлів, їхніми атрибутами й іншими даними, що визначають місцезнаходження файлів (чи їхніх фрагментів) у середовищі FAT. Елемент FAT *визначає* фактичну область диска, у якій зберігається початок фізичного файлу.

файловій системі FAT логічний дисковий простір будь-якого логічного диска поділяється на дві області (рисунок 4.3): *системну область* і *область даних.*

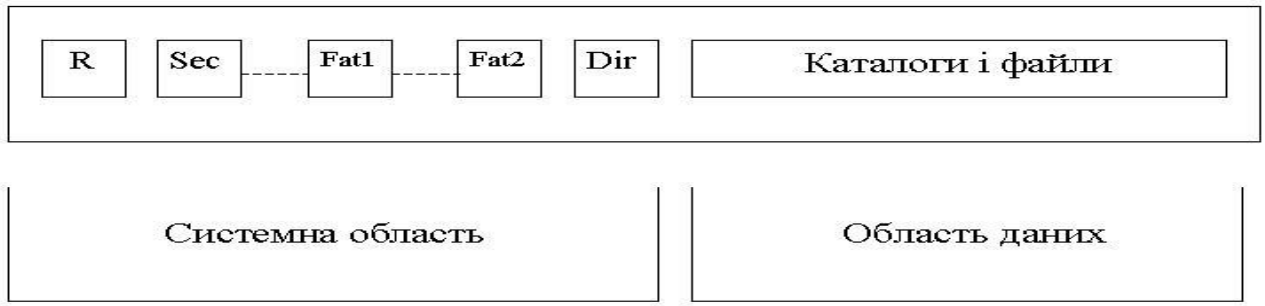


Рисунок 4.3 – Структура логічного диску

Системна область логічного диску створюється і ініціалізується при форматуванні, а потім оновлюється при маніпулюванні файловою структурою. Область даних логічного диску містить файли і каталоги, підпорядковані кореневому. Вона, на відміну від системної області, доступна через користувацький інтерфейс DOS. Системна область складається з наступних компонентів, розміщених у логічному адресному просторі підряд:

завантажуючого запису (*boot record, BR*);

зарезервованих секторів (*reserved sector, ResSecs*);

таблиці розміщення файлів (*file allocation table, FAT*);

кореневого каталогу (*root directory, Rdir*).

Також є файлові системи VFAT (віртуальна FAT) і FAT32, NTFS.

назву файлової системи ***NTFS*** входять слова «*New Technology*», тобто «нова технологія». Дійсно, NTFS містить ряд значних удосконалень і змін, що істотно відрізняють її від інших файлових систем. З погляду користувачів,

файли як і раніше зберігаються в каталогах (часто званих «папками» чи *фолдерами* в середовищі Windows). У NTFS на відміну від FAT робота на дисках великого обсягу відбувається набагато ефективніше; є засоби для обмеження в доступі до файлів і каталогів, введені механізми, що істотно підвищують надійність файлової системи, зняті багато обмежень на максимальну кількість дискових секторів і/чи кластерів.

Основні можливості файлової системи NTFS.

При проектуванні системи NTFS особлива увага була приділена наступним характеристикам:

Надійність. Високопродуктивні комп'ютери і системи спільного користування (сервери) повинні мати підвищену надійність, що є ключовим елементом структури і поводження NTFS. Одним зі способів збільшення надійності є введення механізму транзакцій, при якому здійснюється **журналювання** файлових операцій;

Розширена функціональність. NTFS проектувалася з врахуванням можливого розширення. У ній були втілені багато додаткових можливостей — вдосконалена відмовостійкість, емуляція інших файлових систем, могутня модель безпеки, рівнобіжна обробка потоків даних і створення файлових атрибутів, обумовлених користувачем;

Підтримка POSIX. Оскільки уряд США вимагав, щоб усі закуповувані ним системи хоча б у мінімальному ступені відповідали стандарту POSIX, така можливість була передбачена і в NTFS. До числа базових засобів файлової системи POSIX відноситься необов'язкове використання імен файлів з урахуванням регістра, збереження часу останнього звертання до файлу і механізм так званих «жорстких посилань» — альтернативних імен, що дозволяють посилатися на той самий файл по двох і більше іменах;

Гнучкість. Модель розподілу дискового простору в NTFS відрізняється надзвичайною гнучкістю. Розмір кластера може змінюватися від 512 байт до 64 Кбайт; він являє собою число, кратне внутрішньому кванту розподілу дискового простору. NTFS також підтримує довгі імена файлів, набір символів Unicode і альтернативні імена формату 8.3 для сумісності з
FAT.

4.2.3 Фізична організація і адресація файлу

Фізична організація файлу – це спосіб розміщення файлу на диску. Основними критеріями ефективності фізичної організації файлів є:

швидкість доступу до даних;

об'єм адресної інформації файлу;

ступінь фрагментації дискового простору;

максимально можливий розмір файлу.

Безперервне розміщення - це найпростіший варіант фізичної організації, при якому файлу надається послідовність кластерів диска, що утворюють безперервну ділянку дискової пам'яті. Цей метод має свої переваги: 1) висока швидкість доступу так як витрати на пошук і зчитування кластерів файлу мінімальні; 2) мінімальний об'єм адресної інформації; 3) необмежений максимально можливий розмір файлу.

Однак реалізувати цю схему складно через те, що розмір файлу, як правило, заздалегідь невідомий. Ще більш серйозною проблемою є фрагментація: через деякий час після створення такої файлової системи в результаті виконання численних операцій створення та видалення файлів простір диска неминуче перетворюється на «клаптикову ковдру», що включає в себе велику кількість вільних областей невеликого розміру.

Розміщення файлу в вигляді зв'язаного списку кластерів, коли на початку кожного кластера міститься покажчик на наступний кластер.

На відміну від попереднього способу, кожен кластер тут може бути приєднаний до ланцюжку кластерів якого-небудь файлу, а отже, фрагментація на рівні кластерів відсутня. Файл може змінювати свій розмір під час свого існування, нарощуючи число кластерів. Але маючи певні переваги цей метод також має свої недоліки: 1) складність реалізації доступу до довільного місця файлу; 2) кількість даних файлу, що містяться в одному кластері, не дорівнює ступеню числа 2 (так як одне машинне слово вже витрачено на номер наступного кластера), а багато програм читають дані кластерами, розмір яких дорівнює ступеню числа 2.

Використання зв'язаного списку індексів. Це – модифікація попереднього способу. Тут файлу також виділяється пам'ять у вигляді зв'язаного списку кластерів. При цьому номер першого кластера

запам'ятовується в записі каталогу, де зберігаються характеристики даного файлу, а інша адресна інформація відділена від кластерів файлу. З кожним кластером диска зв'язується особливий елемент – індекс, де всі індекси розташовуються в окремій області диска: в ОС MS-DOS це таблиця FAT, що займає один окремий кластер. Коли пам'ять вільна, всі індекси мають нульове значення. Якщо ж деякий кластер N призначений деякому файлу, то індекс цього кластера стає рівним або номеру M наступного кластера даного файлу, або приймає спеціальне значення, що є ознакою, що даний кластер є для файлу останнім. Індекс ж попереднього кластера файлу приймає значення N, вказуючи на новопризначений кластер.

При такій фізичній організації зберігаються всі переваги попереднього способу – мінімальність адресної інформації, відсутність фрагментації і відсутність проблем при зміні розміру файлу, але з'являються і додаткові переваги:

висока швидкість доступу (для доступу до довільного кластеру файлу не потрібно послідовно зчитувати його кластери – досить прочитати тільки сектори диска, які містять таблицю індексів, відрахувати потрібну кількість кластерів файлу по ланцюжку і визначити номер потрібного кластера);

дані файлу заповнюють кластер цілком, а значить, мають об'єм, рівний ступеню числа 2.

Недолік же цього методу тільки один: вся таблиця FAT повинна постійно знаходитися в ОП чи кожен раз цілком завантажуватися в ОП.

Перелік номерів кластерів, зайнятих файлом (метод і-вузлів). Такий перелік і служить адресою файлу. З кожним файлом зв'язана структура даних «*і-вузол*», що містить також атрибути файлу.

Недолік цього методу: довжина адреси залежить від розміру файлу, і для великого файлу може скласти значну величину. Але і переваги цього методу в тому, що: 1) висока швидкість доступу до довільного кластеру файлу, так як тут застосовується пряма адресація, яка виключає перегляд ланцюжка покажчиків при пошуку адреси довільного кластера файлу; 2) відсутня фрагментація на рівні кластерів; 3) кожен вузол повинен знаходитися в ОП тільки тоді, коли відповідний йому файл відкритий.

Модифікований підхід до переліку номерів кластерів. Даний метод використовується в ОС UNIX в традиційних файлових системах s5 і ufs, в ОС

Linux в найбільш поширених файлових системах ext2fs[1] та ext3fs (остання відрізняється від першої тільки наявністю журналювання). Для скорочення обсягу адресної інформації прямий спосіб адресації поєднується з непрямим.

стандартній для UNIX ФС ufs використовується наступна схема адресації кластерів файлу (рис. 4.4). Для зберігання адреси файлу виділено 15 полів, кожне з яких складається з 4 байтів. Якщо розмір файлу менше або дорівнює 12 кластерам, то номери цих кластерів безпосередньо перераховуються в перших дванадцяти полях адреси.

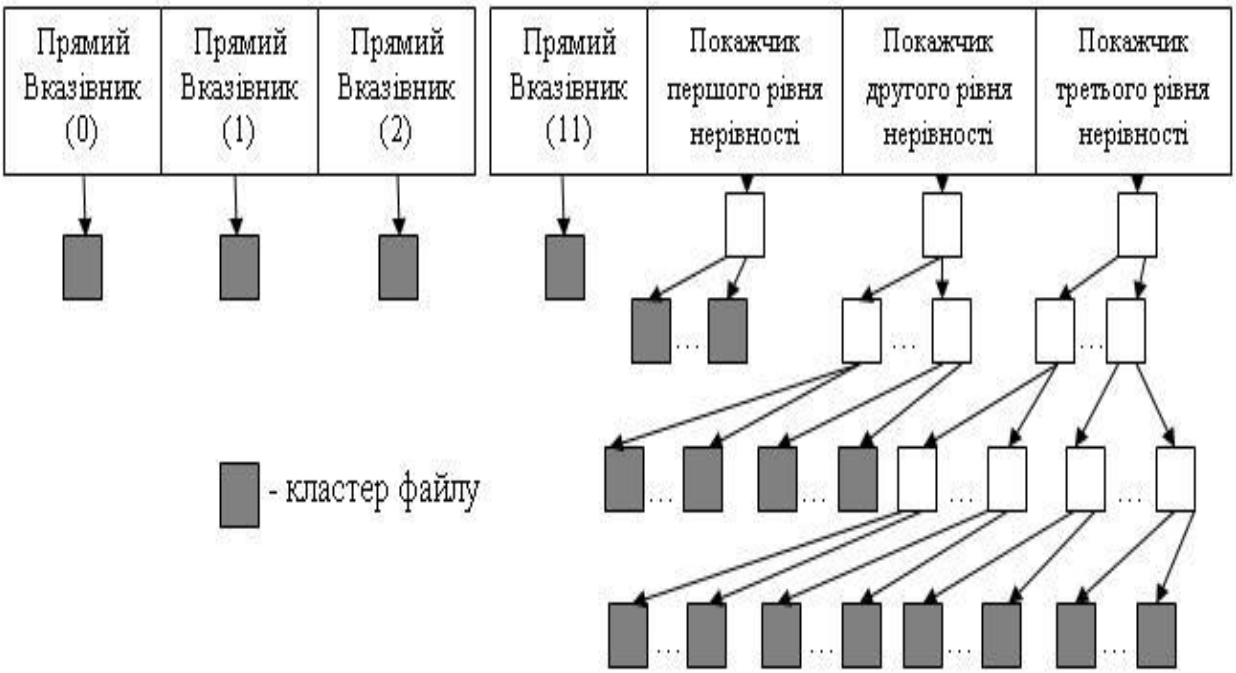


Рисунок 4.4 – Схема адресації кластерів файлу у файловій системі ufs

Якщо розмір файлу перевищує 12 кластерів, то наступне, 13-е поле містить адресу кластера, в якому можуть бути розташовані номери наступних кластерів файлу. Якщо розмір файлу перевищує 12 + 2048 = 2060 кластерів, то використовується 14-е поле, в якому знаходиться номер кластера, що містить 2048 номерів кластерів, кожен з яких зберігає 2048 номерів кластерів даних файлу, – тобто тут застосовується вже подвійна непряма адресація. І нарешті,

ext2fs – стандартна дискова файлова система для Linux, в якій дисковий простір ділять не на групи циліндрів, а просто на групи блоків (block groups), не прив'язані до геометрії диска. Такі групи за структурою схожі на групи циліндрів: кожна група блоків теж є зменшеною копією файлової системи із суперблоком, таблицею індексних дескрипторів тощо.

якщо файл включає більше 12 + 2048 +2048 2048 = 4196364 кластерів, то використовується останнє, 15-е поле для потрійний непрямої адресації.

Метод переліку адрес кластерів файлу задіяний і в файловій системі NTFS, використовуваної в ОС сімейства Microsoft Windows NT. Тут він доповнений досить природним прийомом, що скорочує об'єм адресної інформації: адресуються не кластери файлу, а безперервні області, що складаються з суміжних кластерів диска. Кожна така область, що називається групою (в *ext2fs*) або екстентом (*extent*), описується за допомогою двох чисел – початкового номера кластера та кількості кластерів у відрізку. Так як для скорочення часу операції обміну ОС намагається розмістити файл у послідовних кластерах диска, то в більшості випадків кількість послідовних областей файлу буде менше кількості кластерів файлу, так що обсяг службової адресної інформації в NTFS скорочується порівняно зі схемою адресації файлових систем ufs/s5.

4.3 Логічна організація файлової системи

ОС повинна надавати користувачеві зручності при роботі з даними на дисках. Для цього ОС підміняє фізичну структуру даних зручною для користувача логічною моделлю. При цьому логічна модель файлової системи матеріалізується у вигляді дерева каталогів, в символьних складових імен файлів і в командах роботи з файлами.

4.3.1 Типи файлів

Звичайні файли (або просто файли), містять інформацію довільного характеру, яку заносить в них користувач або яка утворюється в результаті роботи системних і користувальних програм.

Каталоги – це особливий тип файлів, які містять системну довідкову інформацію про набір файлів, згрупованих користувачами за якоюсь ознакою.

каталог можуть входити файли будь-яких типів, в тому числі інші каталоги, за рахунок чого утворюється деревоподібна структура, зручна для пошуку. Каталоги встановлюють відповідність між іменами файлів і їх характеристиками, використовуваними файловою системою для управління файлами.

Спеціальні файли – це файли, асоційовані з пристроями введення/виведення і використовувані для уніфікації механізму доступу до файлів і зовнішніх пристроїв. Спеціальні файли дозволяють користувачеві виконувати операції введення/виведення за допомогою звичайних команд запису у файл або читання з файлу. Сучасні ФС підтримують і інші типи файлів (символьні зв'язки, іменовані конвеєри, відображувані в пам'ять файли та ін.).

4.3.2 Ієрархічна структура файлової системи

Більшість файлових систем має ієрархічну структуру, в якій рівні створюються за рахунок того, що каталог більш низького рівня може входити в каталог більш високого рівня.

Граф, що описує ієрархію каталогів, може представляти собою дерево або мережу. Каталоги утворюють *дерево*, якщо файлу дозволено входити тільки в один каталог, або *мережу*, якщо файл може входити відразу в кілька каталогів. Наприклад, в MS-DOS і Windows каталог утворюють деревовидну структуру, а UNIX – мережеву. У деревоподібній структурі кожен файл є листом. Каталог самого верхнього рівня називається кореневим каталогом, або коренем (*root*).

При такій організації користувач звільнений від запам'ятовування імен всіх файлів; йому досить приблизно уявляти, до якої групи може бути віднесений той або інший файл, щоб шляхом послідовного перегляду каталогів знайти його. Ієрархічна структура зручна і для багатокористувацької роботи: кожен користувач зі своїми файлами локалізується в своєму каталозі або піддереві каталогів, але разом з тим, всі файли в системі логічно пов'язані.

Окремим випадком ієрархічної структури є однорівнева організація, коли всі файли входять в один каталог.

Імена файлів.

Всі типи файлів мають символьні імена. В ієрархічно організованих файлових системах зазвичай використовуються три типи імен файлів: прості і складні, де останні можуть бути абсолютними і відносними.

Просте, або коротке, символьне ім'я ідентифікує файл в межах одного каталогу. Прості імена привласнюють файлам користувачі і програмісти. В ієрархічних файлових системах різним файлам дозволено мати однакові прості символьні імена за умови, що вони належать різним каталогам, – тобто тут працює схема: «багато файлів – одне просте ім'я».

Для однозначної ідентифікації файлу в таких системах використовується так зване *повне ім'я*, яке являє собою ланцюжок простих символьних імен всіх каталогів, через які проходить шлях від кореня до даного файлу, плюс просте ім'я самого цього файлу.

деревоподібної файлової системи між файлом і його повним ім'ям мається взаємно однозначна відповідність: «один файл – одне повне ім'я».

файлових же системах, що мають мережеву структуру, файл може входити в кілька каталогів, а значить мати кілька повних імен; тут справедлива відповідність: «один файл – багато повних імен». Проте в обох цих випадках файл однозначно ідентифікується повним ім'ям.

Файл може бути також ідентифікований відносним ім'ям, яке визначається через поняття «поточний каталог». При цьому файл ідентифікується ланцюжком імен каталогів, через які проходить маршрут від поточного каталогу до даного файлу.

Хоча повне ім'я однозначно визначає файл, операційній системі простіше працювати з файлом, якщо між файлами і їх іменами мається взаємно однозначна відповідність. З цією метою ОС також привласнює файлу унікальне ім'я, так що справедливе співвідношення: «один файл – одне унікальне ім'я». Унікальне ім'я існує поряд з одним або декількома символьними іменами, присвоюється файлу користувачами або додатками, представляє собою числовий ідентифікатор і призначене тільки для ОС. Приклад – номер індексного дескриптора в системі UNIX.

Монтування

В обчислювальній системі може бути декілька дискових пристроїв. При цьому одна фізична пристрій за допомогою засобів ОС може бути представлене як кілька логічних пристроїв (зокрема, шляхом розбиття дискового простору на розділи).

Як організувати зберігання файлів в системі, що має кілька пристроїв зовнішньої пам'яті? Для цього можливо два рішення:

1. На кожному з пристроїв розміщується автономна ФС, тобто файли, що знаходяться на цьому пристрої, описуються деревом каталогів, ніяк не пов'язаним з деревами каталогів на інших пристроях. У такому випадку для однозначної ідентифікації файлу користувач, поряд з складовим символьним ім'ям файлу, повинен вказувати *ідентифікатор логічного пристрою*. Приклад – ОС MS-DOS.

Користувачеві надається можливість об'єднувати файлові системи, що знаходяться на різних пристроях, в єдину ФС, що описується єдиним деревом каталогів. Така операція називається *монтуванням*.

4.3.3 Атрибути файлів

Файл – це не тільки дані, але й атрибути-інформація, що описує властивості файлу. Приклади можливих атрибутів файлу:

тип файлу (звичайний файл, каталог, спеціальний файл і т. п.);
власник файлу;
творець файлу;
пароль для доступу до файлу;
інформація про дозволені операції доступу до файлу;
час створення, останнього доступу і останньої зміни;
поточний розмір файлу;
максимальний розмір файлу;
ознака «тільки для читання»;
ознака «прихований файл»;
ознака «системний файл»;
ознака «архівний файл»;
ознака «двійковий/символьний»;
ознака «тимчасовий» (видаляється після завершення процесу);
ознака блокування;
довжина запису у файлі;
покажчик на ключове поле в записі;
довжина ключа.

Набір атрибутів файлу визначається специфікою файлової системи. Користувач може одержувати доступ до атрибутів, використовуючи засоби, що надані для цих цілей файловою системою. Зазвичай дозволяється читати значення будь-яких атрибутів, а змінювати тільки деякі з них.

Контрольні питання

Назвіть способи здійснення введення/виведення. Який з них найкращий, назвіть його основні переваги?
Які програмні рівні вводу/виводу?

Назвіть основні категорії драйверів пристроїв?

Які функції повинен виконувати драйвер пристрою?

Що таке файл? Для чого він необхідне?

З яких елементів складаються файли простої структури?

З яких елементів складаються файли складної структури?

Що включає в себе файлова система? Яку роль відіграє файлова система в ОС?

Які основні функції файлової системи?

10.Опишіть будову жорсткого диску.

11. Що таке фізичне і логічне форматування диску?

12. Назвіть сучасні, відомі Вам файлові системи?

13.Що являє собою файлова система FAT?

14. Які основні можливості файлової системи NTFS?

Що таке фізична організація файлу? Назвіть основні способи розміщення файлу на диску?

16. Які є типи файлів? Опишіть коротко кожен тип.

17. Які є види ієрархічної структури файлової системи? Для яких ОС вони властиві?

18. Які три типи імен файлів використовуються в ієрархічно організованих файлових системах?

19. Що таке монтування? В яких випадках застосовується?

20. Наведіть приклади можливих атрибутів файлу.

РОЗДІЛ 5

ТЕНДЕНЦІЇ РОЗВИТКУ СУЧАСНИХ ОПЕРАЦІЙНИХ СИСТЕМ

5.1 Сучасні тенденції в розвитку ОС

При стрімкому розвитку сучасних операційних систем всі вони мають відповідати сучасним тенденціям і вимогам аби бути конурентноспроможніми на комп'ютерному ринку. Тому розробники ОС мають дотримуватись наступних основних вимог.

Підтримка нових мережевих технологій і Web-технологій. Мережі та Інтернет активно розвиваються. З'являються нові стандарти і протоколи – IPv6, HTML 5 (для хмарних обчислень) і т.д. Сучасні ОС розвиваються в напрямку підтримки всіх нових мережевих технологій.

Підтримка багатопоточності і багатоядерних процесорів. Всі сучасні ОС мають бібліотеки програм, що підтримують багатопоточність. Саме завдяки багатоядерній архітектурі, стає реально можливим паралельне виконання потоків (threads).

Графічні оболонки. Будь-яка сучасна ОС має графічний користувацький інтерфейс, причому (з цілком зрозумілих причин гострої конкуренції між фірмами-розробниками) графічні оболонки для всіх ОС приблизно однакові за можливостями. Часом користувачеві важко зорієнтуватися, в який саме ОС він працює, хоча для кінцевих користувачів (непрограмістів), мабуть, така уніфікація зручна.

Розвиток файлових систем з метою захисту інформації та значного збільшення розміру файлів (для мультимедіа). Розробляються нові файлові системи, що допускають зберігання дуже великих файлів. Іншою вимогою є забезпечення конфіденційності інформації, яке призводить до необхідності реалізації в файлових системах можливості криптування.

Посилена увага до механізмів безпеки і захисту. Багато в чому завдяки ініціативі Trustworthy Computing, розпочатої фірмою Microsoft в 2002 р.,

також через існування кіберзлочинності, всі сучасні ОС приділяють підвищену увагу безпеці: при перегляді веб-сторінок браузери виконують їх перевірку на відсутність *phishing*; завантаження та інсталяції програм з мережі виконуються тільки з явної згоди користувача і т.д.

Підтримка розподілених і паралельних обчислень. Сучасні ОС мають у своєму складі високорівневі бібліотеки, що дозволяють розробляти паралельні алгоритми вирішення завдань - наприклад, підтримуючі стандарти паралелізму OpenMP і MPI.

Віртуалізація ресурсів і апаратури. Сучасні ОС мають у своєму складі засоби віртуалізації, що дозволяють виконувати додатки для інших платформ в ізольованих віртуальних машинах, в які можуть бути інстальовані інші операційні системи.

5.1.1 Операційні системи різних крупних фірм

IBM – в 1960-х – 1970-х рр. розробила OS IBM 360/370; потім – ОС для персональних комп'ютерів OS / 2; в даний час найбільш сучасними ОС цієї фірми є z / OS і z / VM;

Apple – з початку 1980-х рр. розвиває сімейство ОС MacOS, що характеризується поліпшеним графічним інтерфейсом користувача;

Oracle / Sun – с початку 1980-х рр. фірма Sun розвиває ОС Solaris, розвиток UNIX;

Hewlett-Packard – розвиває власний діалект UNIX – систему HP / UX ;

Novell – одна з провідних фірм в області мережевих технологій; розвиває сімейство мережевих операційних систем: NetWare; нині – Open Enterprise Server (мережева ОС, яка включає всі мережеві можливості NetWare і можливості поширеного діалекту Linux – openSUSE).

Це далеко не повний перелік комерційних і дослідницьких ОС, що включає сотні найменувань.

Solaris - операційна система розробки фірми Oracle / Sun

Вона є розвитком UNIX (лінія Solaris 1.x – розвитком діалекту UNIX BSD, лінія Solaris 2.x – розвитком діалекту UNIX AT & T SVR4). На думку спеціалістів ця ОС є досить зручною і надійною, а також має зручні інструменти розробки програм. Одна з багатьох особливостей Solaris – це підтримка багатопоточності. Апаратні платформи, підтримувані ОС Solaris: SPARC, x86, IA-32 і x64.

У 1990-х роках багато фірм-користувачів воліли використовувати саме Solaris як серверну ОС, а в якості серверів – комп'ютери SPARC фірми Sun. Важливою перевагою Solaris є використовуване в цій системі сімейство

оптимізуючих компіляторів Sun Studio (C, C + +, Fortran). В даний час компілятори Sun Studio працюють також на ОС Linux. Дуже цікавим і потужним інструментом, доступним в системі Solaris, є відладник розподілених додатків DTrace. Нова файлова система ОС Solaris - ZFS (Zetabyte File System, що підтримує великі обсяги файлів і можливість криптування інформації у файлах з метою її захисту. Система Solaris має потужну підтримку багатопоточності і синхронізації потоків, а також зручну підтримку симетричного багатопроцесування (SMP-архітектур). Використовувані в системі Solaris графічні оболонки – Common Desktop Environment (CDE), GNOME, Java Desktop; в 1990-х рр. використовувалася також графічна оболонка OpenWindows.

MacOS

Розробником MacOS є фірма Apple – законодавець мод в області GUI, починаючи з 1980-х рр. Центральною ідеєю MacOS з самого початку є розробка розвиток ОС тільки на основі графічного інтерфейсу користувача – «ОС без командного рядка».

Апаратна платформа MacOS – різні сімейства комп'ютерів Macintosh фірми Apple (найбільш популярні серед робочих станцій у США), а також PowerPC – робоча станція RISC-архітектури, спільно розроблена Apple, IBM і HP.

Версії MacOS розрізняються за своїм підходу до реалізації, хоча для користувача, завдяки зручному графічному інтерфейсу, ці відмінності можуть бути непомітні. Класична MacOS (classic MacOS) – оригінальна розробка фірми Apple; нова лінія MacOSX – розвиток ОС MacOS Classic і ОС NeXTSTEP (UNIX-подібної ОС), тобто вона є UNIX-сумісною.

HP-UX

Розробником ОС HP-UX є компанія Hewlett-Packard. Це UNIX-подібна 64-бітова ОС (одна з перших 64-бітових ОС в історії). Її апаратна платформа – робочі станції фірми HP. В системі HP-UX використовується оригінальна віртуальна файлова система: Veritas File System (Vx-Fs) використовує представлення файлів на основі розширень (extents).

5.1.2 Переваги 64-розрядної версії Windows

Ще пару років тому на більшості домашніх комп'ютерів працювала 32-бітова версія Windows, а 64-бітний варіант використовували виключно професіонали (наприклад, для ресурсоємних програм відеомонтажу). При цьому процесори з 64-розрядної архітектурою випускаються вже з 2003 року, а двома роками пізніше на ринок вийшла Windows XP (не підтримується розробниками) в 64-бітному варіанті. Але на більшості нових комп'ютерів 64-бітові версії Windows почали встановлювати тільки після появи Windows 7.

Переваги 64-розрядної системи.

Збільшена робоча швидкість. З 64-бітовою версією Windows комп'ютер може обробляти за одиницю часу в два рази більше даних, ніж з 32-бітовою. При цьому він використовує розширені можливості 64-розрядних процесорів, які за один такт здатні обробляти 64 біта даних (8 байт). Тому програми, оптимізовані для 64-бітових ОС, вміють працювати швидше, ніж їх аналоги для процесорів, що обробляють всього 32 біта (4 байти) за один такт.

Більше оперативної пам'яті. 32-бітові версії Windows можуть використовувати максимум 4 Гб пам'яті, причому не вся вона доступна програмам. Наприклад, 1 Гб займає сама операційна система, до того ж до 1 Гб (залежно від ПК) резервується для відеопам’яті. Тож для самих програм залишається не більше 2 Гб. Якщо вони «ненажерливі», пам'яті може не вистачити. Для 64-бітових версій Windows подібного обмеження немає. Максимальна робоча пам'ять в Windows 7 Home Premium виросла до 16 Гб; версії Professional, Ultimate і Enterprise можуть працювати навіть з 192 Гб. Завдяки більш доступної пам'яті Windows рідше доводиться вивантажувати розділи програм на порівняно повільний жорсткий диск (у файл підкачки), тому і комп'ютер працює швидше

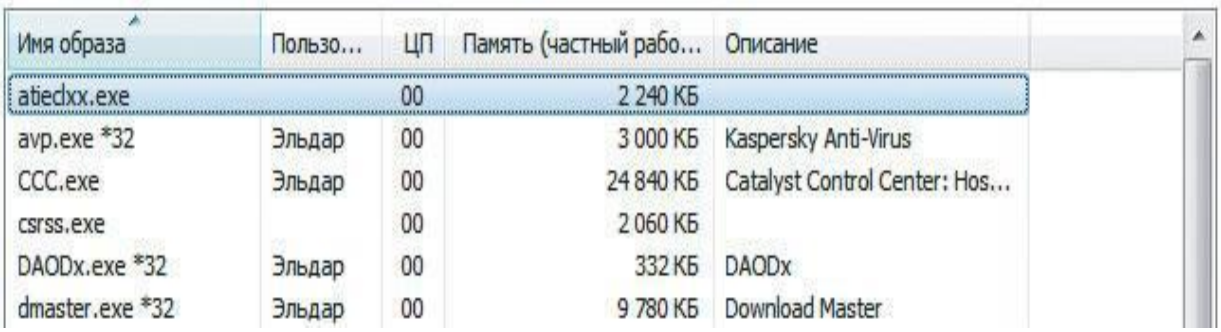
Вимоги до комп'ютера.

Процесор. Найважливішою необхідною умовою для роботи 64-бітної операційної системи є процесор з 64-бітною архітектурою. Виробник процесорів AMD називає цю технологію AMD64, а його конкурент Intel - EM64T.

Процесори з 64-бітною архітектурою вже стали скоріше правилом, ніж винятком. Приблизно з 2006 року більшість нових комп'ютерів і ноутбуків оснащується саме ними, хоча багато ще працюють під 32-бітною Windows. Дізнатися, який саме процесор встановлений на вашому ПК, можна за допомогою програми CPU-Z.

Оперативна пам'ять. 64-бітна Windows має сенс лише в тому випадку, якщо у комп'ютера не менше 4 Гб оперативної пам'яті. В іншому 64-бітові версії ОС не пред'являють особливих вимог до апаратної частини (наприклад, до HDD або відеокарти) в порівнянні з 32-бітним.

Якщо в 64-бітній Windows працюють 32-бітові програми, то в диспетчері завдань вони позначаються відповідним чином (* 32) (Рисунок 5.1). Диспетчер завдань можна запустити, натиснувши комбінацію клавіш Ctrl + Shift + Esc.



Имя образа	Пользо...	ЦП	Память (частный рабо...	Описание
atiedxx.exe		00	2 240 КБ	
avp.exe *32	Эльдар	00	3 000 КБ	Kaspersky Anti-Virus
CCC.exe	Эльдар	00	24 840 КБ	Catalyst Control Center: Hos...
csrss.exe		00	2 060 КБ	
DAODx.exe *32	Эльдар	00	332 КБ	DAODx
dmaster.exe *32	Эльдар	00	9 780 КБ	Download Master

Рисунок 5.1 – Диспетчер завдань

Як дізнатися версію операційної системи? Це дуже просто: клацніть по кнопці «Пуск», потім правою кнопкою миші – по «Комп'ютер» і виберіть «Властивості». У вікні, напроти «Тип системи» побачите версію системи.

Недоліки 64-розрядної системи.

Потрібні нові драйвери. Всі пристрої вимагають для 64-бітної Windows відповідних драйверів (аналоги для 32-бітної ОС працювати не будуть). Для сучасних пристроїв це звичайно не проблема, але для застарілих моделей багато виробників не розробляють 64-бітові версії драйверів. Тому перш ніж переходити на 64-бітову Windows, зайдіть на веб-сторінку виробника пристрою переконайтеся, що для вашого обладнання доступна 64-бітова версія драйвера.

Потрібно більше пам'яті. Багато структури даних в 64-бітових програмах мають розмір 8 байт (64 біт). Тому програми займають на (10–20) % більше місця на жорстких дисках, ніж відповідні 32-бітні версії з 4-байтним структурами. Потреба в оперативній пам'яті для 64-бітного програмного забезпечення також підвищується – приблизно на 15 %.

Краща п'ятірка програм для 64-бітної системи

Ці програми чудово працюють в 64-бітній операційній системи і користуються підвищеним попитом у користувачів.

7-ZIP Програма-архіватор, призначена для стиснення файлів. Може бути корисна, наприклад, коли потрібно скопіювати багато великих файлів на USB-драйв.

Nitro PDF Reader. Ця проста програма дозволяє легко створювати, переглядати та редагувати документи PDF-формату.

nHancer. Утиліта забезпечує доступ до настройок графічної плати NVIDIA. Працює з усіма поширеними моделями відеокарт і навіть з SLI-конфігураціями. Можна вибирати профілі налаштувань для різних ігор.

ProgDVB. Якщо у вашому ПК встановлено цифровий ТВ-тюнер, за допомогою ProgDVB вдасться переглядати телевізійні передачі і записувати їх на жорсткий диск. Програма працює з ТВ-тюнерами різних виробників. Крім безкоштовного варіанту, є розширена платна версія.

Blender. Програма для роботи з тривимірними об'єктами. До її складу входять засоби для моделювання, анімації, постобробки відео, а також створення інтерактивних ігор.

Виникає питання, чи працюють 32-бітові програми під 64-бітної Windows? Підсистема перетворення під назвою Windows-on-Windows 64-bit (WoW64) забезпечує спільну роботу 32-бітного програмного забезпечення в 64-бітній операційній системі. З її допомогою більшість 32-бітних програм працює бездоганно але, на жаль, є винятки. Труднощі зазвичай виникають з програмами наступних категорій.

Резервне копіювання. Програми для створення бекапів безпосередньо працюють із системними папками Windows. Але WoW64 іноді перенаправляє доступ 32-бітного програмного забезпечення в інші папки. Якщо програма цього не враховує, резервне копіювання може виявитися неповним або навіть помилковим.

Оптимізація Windows. Старі, розраховані на 32-бітову Windows програми оптимізації не можна запускати на 64-бітній Windows. Часом вони здатні більше нашкодити, ніж допомогти.

Антивіруси. Якщо потрібен надійний захист для ПК, використовуйте в 64-бітній Windows тільки 64-бітові версії антивірусів і аналогічних програм. Застаріле 32-бітне захисне програмне забезпечення нездатне контролювати всі системні папки, а значить, може пропустити шкідників, розроблених спеціально для атак на 64-бітову Windows. Сучасні антивірусні пакети відомих

виробників, наприклад «Лабораторії Касперського», Norton або Symantec та інші, підходять як для 32 -, так і для 64-бітної Windows.

Якщо програма під 64-бітову Windows працює зі збоями, позбавитися від них нерідко допомагає установка новітньої версії і всіх оновлень з веб-сайту виробника. До того ж у «Центрі сумісності Windows» можна перевірити, чи сумісні ваші пристрої і програми з 64-бітної системою.

64-бітній Windows програми працюють швидше. Деякі 64-бітові програми були всього лише перетворені для застосування в 64-бітній Windows, але по суті залишилися 32-бітними. Наприклад, хоча вони і мають 64-бітні структури даних, однак зберігають у них як і раніше 32 біта даних. Подібні програми, як правило, не в змозі працювати швидше. 64-бітові програми реалізують свої переваги, лише якщо з самого початку розроблялися і оптимізували для застосування в 64-бітових Windows. У тесті ComputerBild такі програми в середньому працювали на 13 % швидше 32-бітних «побратимів». Старе 32-бітне програмне забезпечення в 64-бітній Windows працює навіть повільніше, ніж в 32-бітній системі; причина – перетворення підсистемою WoW64 64-бітових програм. Знайти подібні програми поки не так легко, як хотілося б. У таблиці 5.1 приведені приклади 64-бітових програм, багато з них безкоштовні і без особливих зусиль скачуються з сайту розробника.

Таблиця 5.1 – Приклади 64-бітових програм

Програма	**Версія**	**Категорія**	**Інтернет-адреса**
7-Zip	9.20	Архіватор	www.7-zip.org.ua/ru
AVG Anti-Virus Free	2011	Антивірусна програма	www.freeavg.com
Blender	2.59	Редагування 3D-моделей	www.blender.org
CCleaner	3.10	Очистка системи	www.piriform.com
Defraggler	2.06	Дефрагментатор	www.piriform.com
GIMP	2.7.3	Обробка зображень	www.gimp.org
Image Composite Editor	1.4.4	Створення панорам	www.microsoft.ru
Internet Explorer	9	Інтернет-браузер	www.microsoft.ru
iTunes	10.4	Відтворення музики і відео	www.apple.com/ru/itunes
Mathematics	4.0	Математична програма	www.microsoft.ru

Програма	Версія	Категорія	Інтернет-адреса
MediaCoder	2011 R8	Конвертоване відео	www.mediacoderhq.com
Mixxx	1.9.0	DJ-програма	www.mixxx.org
nHancer	2.5.9	Оптимізація графічних плат	www.nhancer.com
Nitro PDF Reader	2.0	Редагування PDF-файлів	www.nitroreader.com
Office	2010	Офісний пакет	www.microsoft.ru
Paint.NET	3.5.8	Обробка зображень	www.getpaint.net
Partition Manager	11	Управління розділами жорстких дисків	www.paragon.ru
PDF Xchange Viewer	2.5.197	Перегляд PDF	www.tracker-software.com
ProgDVB	6.70.6	Перегляд і запис ТВ-передач	www.progdvb.com
SafeErase	5.0	Надійне видалення даних	www.oo-software.com
Sandboxie	3.58	Захищене оточення	www.sandboxie.com
Total Commander	7.56a	Файловий менеджер	www.wincmd.ru
SyncToy	2.1	Синхронізація	www.microsoft.ru
Thunderbird	6.0.1	Поштова програма	www.mozilla.org/ru/thunderbird
Vegas Pro	10	Програма відеомонтажу	www.sonycreativesoftware.com
Virtual Dub	1.9.11	Конвертування відео	www.virtualdub.org
Virtual PC	6.1	Програма віртуалізації	www.microsoft.ru
WinRAR	4.01	Архіватор	www.rarlab.com

5.1.3 Windows 8, як новий етап у розвитку операційних систем Microsoft

Windows 8 було приділено увагу системі розпізнавання голосу і голосовому управлінню. Серед інших можливих нововведень була поліпшена робота віртуалізації, що дозволяє віртуалізувати всю операційну систему; тісної інтеграції з мобільною версією операційної системи, підтримка розпізнавання жестів. У Windows 8 вдосконалена система глибокого сну та відновлення, а також нові функції забезпечення безпеки, включаючи модифікацію PatchGuard.

Згідно презентації Microsoft, в Windows 8 був істотно скорочений час завантаження і вимкнення в порівнянні з попередніми ОС сімейства Windows. Приділена увага енергоспоживанню та енергоефективності. Заявлена підтримка інтерфейсу USB 3.0, з'єднань Bluetooth 3.0, а також бездротових і стереоскопічних дисплеїв.

Одним з нововведень на той час була система розпізнавання особи користувача за допомогою веб-камери. Ця система може самостійно переводити ПК в різні режими енергоспоживання залежно від того, чи знаходиться користувач перед веб-камерою чи ні. У певному вигляді подібна концепція реалізована в ігровому контролері Kinect для ігрової консолі Xbox 360.

Windows 8 вже присутні нові інструменти для забезпечення безпеки, які спрощують діагностику та усунення несправностей. Є функція «скидання системи», яка дозволить перевстановити Windows 8 із збереженням при цьому всі файли користувача. Також Windows 8 підтримує акселерометри, шифрування жорсткого диску, підтримку GPS в нетбуках і ноутбуках, використання WWAN-модулів.

Особливості Windows 8

Всі традиційні програми для Windows 7 працюватимуть і під Windows 8.

Windows 8 має такі ж системні вимоги, як і попередня версія операційної системи: процесор частотою 1 ГГц, 1 ГБ оперативної пам'яті і не менше 16 ГБ вільного дискового простору.

Windows 8 використовує новий механізм виключення під назвою Hybrid Boot, який прискорює завантаження і вимикання системи в кілька разів. Технологія являє собою гібрид традиційного «холодного» завантаження і відновлення роботи після глибокого сну.

Антивірус включений в систему за замовчуванням. Крім цього, в Windows 8 є поліпшені засоби запобігання атак, деякі з яких вбудовані в ядро системи.

Windows 8 підтримує функцію Near Field Communication, яка дозволяє різним цифровим пристроям взаємодіяти один з одним або, наприклад, зі спеціальними позначками при допомозі простих торкань.

У Windows 8 є вбудована підтримка технології віртуалізації Hyper-V. Це дозволяє одночасно запускати кілька 32-розрядних або 64-розрядних операційних систем x86 на одному комп'ютері.

За допомогою функції Portable Workspace користувач може зробити клон операційної системи на USB-носії. Об'єм носія повинен бути не менше 16 ГБ.

Додані функції оновлення та обнулення стану системи. Refresh використовується в тому випадку, якщо ваш комп'ютер став працювати нестабільно і дозволяє оновити його стан без втрати даних користувача і додатків. Reset можна використовувати для приведення системи в первинний стан на момент установки.

Для входу в систему можуть використовуватися біометричні дані або розпізнавання осіб.

Система має розширену підтримку роботи з декількома моніторами. Тепер можна відображати панель завдань на будь-якому моніторі, керувати шпалерами на різних екранах, відображати з одного боку стартове меню, а з іншого - робочий стіл, і так далі.

5.1.4 Windows 10

Після випуску операційної системи Windows 8 вважається, що пропуск номеру «9» не був пов’язаний з сумісністю. У компанії хотіли, щоб користувач не порівнював нову версію операційної системи з невдалою Windows 8.

Розроблена і представлена Windows 10 – це нова операційна система сімейства Windows, яка створена універсальною для таких гаджетів, як ПК, смартфони, планшети, консолі Xbox та інші, з єдиною платформою розробки і єдиним магазином оригінальних додатків.

1 червня 2015 року користувачі Windows 7 і 8.1 змогли зареєструвати заявку на безкоштовне оновлення на нову версію, тобто на Windows 10. До того ж, це могли зробити як офіційні власники ліцензованого ОС, так і «пірати».

Нова операційна система вийшла у липні та одночасно з'явилася у продажу у 190 країнах, а також стали доступні мобільні версії та адаптації під інші гаджети.

Що стосується найпримітніших оновлень, то у Windows 10 повернулася кнопка «Пуск», а також з'явилася система біометричної ідентифікації Windows Hello. Її завданням є вхід в систему без пароля на мобільних пристроях.

ОС Windows 10 отримала інтелектуальний голосовий помічник Contata, управління яким здійснюється голосом і текстовими командами. Сервіс аналізує запити користувача, вибиваючи найоптимальнші результати.

Windows 10 також з'явився новий інтернет-браузер Microsoft Edge, що прийшов на зміну Internet Explorer. Крім того, із випуском Windows 10 компанія Microsoft перейшла на нову бізнес-модель SaaS (Windows просувається як сервіс). Тобто всі оновлення, актуальні параметри безпеки та сервіси встановлюються в автоматичному режимі.

Системні вимоги:

Процесор: не менше 1 ГГц або SoC.

ОЗП: 1 ГБ (для 32-розрядних систем) або 2 ГБ (для 64-розрядних систем).

Місце на жорсткому диску: 16 ГБ (для 32-розрядних систем) або 20 ГБ (для 64-розрядних систем).

Відеокарта: DirectX версії 9 або вище з драйвером WDDM 1.0.

Дисплей: 800 x 600.

В таблиці 5.2 наведено переваги Windows 10:

Таблиця 5.2 – Переваги і недоліки Windows 10

ПЕРЕВАГИ	**НЕДОЛІКИ**
Новий інтерфейс значків.	Обмежена безкоштовність.
Оновлене меню «Пуск».	Не можна відключити оновлення.
Оновлення – в ОС Windows 10 включено автоматичне завантаження та встановлення оновлень. Тепер відмовитися від них вже буде неможливо.	Не сумісність більшості ігор з новою ОС.
Нова система безпеки, тобто паролі вже не підходять для входу в ПК, тепер це перевірка відбитка пальця, оболонки ока, а також рис обличчя.	З приводу паролів – такі паролі діють лише за наявності спеціального обладнання.
Стабільна робота – ОС Windows 10 працює набагато швидше, збільшилася кількість драйверів.	Відсутність медіаплеєру Windows Media Center.
Браузер Microsoft Edge – новий, універсальний і інтелектуальний браузер, створений спеціально для ОС Windows 10.	Прискорена розрядка деяких ПК (ноутбуків).
Інтелектуальні функції: Cortana – віртуальний помічник, який розрізняє ваші вимоги «на слух».	Помічник Cortana не підтримує російську та українську мови, і в наступних оновленнях ОС не поставлена мета це змінити.

Windows 10 є першою операційною системою компанії Білла Гейтса, яка офіційно стала поширюватися з серверів постачальника, і з ПК її користувачів, заснована на принципі протоколу BitTorrent.

Компагія Microsoft стверджувала, що Windows 10 стане останнім продуктом сімейства Windows.

5.1.5 ОС для хмарних обчислень (cloud computing). Windows Azure

Хмарні обчислення засновані на масштабованих і віртуалізованних ресурсах (даних і програмах), які доступні користувачам через Інтернет і реалізуються на базі потужних центрів обробки даних (data centers). Виділяють доступні «хмари», що надаються різними компаніями, які можна використовувати для доступу до потужних обчислювальних ресурсів, а користувач сплачує абонентську плату за використання хмарних сервісів певної фірми.

«Хмариний» підхід має певні недоліки: по-перше, є повна залежність користувача від використовуваної ним хмари, так як через хмару доступні не тільки програми, але і дані самого користувача; по-друге, виникає багато питань щодо безпеки даних користувача, які зберігаються в хмарі.

хмарних платформ найбільш популярною є Microsoft Windows Azure - операційна система з підтримкою хмарних обчислень. І Microsoft Azure Services Platform – платформа для розробки і використання хмарних сервісів на базі Microsoft.NET.

В даний час багато великих компаній – Microsoft, Google, IBM, Oracle/Sun, Amazon займаються розробкою своїх хмарних сервісів та інструментів для їх створення. Є тенденція до інтеграції «корпоративних хмар» єдину доступну користувачеві хмару.

Концепція хмарних обчислень базується на таких положеннях: інфраструктура як сервіс, платформа як сервіс, програмне забезпечення як сервіс, а також бізнес-додатки доступні через Інтернет. Іншими словами, організація хмарних обчислень докорінно змінює архітектуру системи: в ній необхідно представити всі можливості обробки даних, використання програм, налаштування і т.д. як хмарні сервіси.

Розрізняються наступні рівні архітектури хмарних обчислень.

Рівень клієнта – це клієнтське програмне забезпечення, що використовується для доступу до хмарних сервісів, наприклад, web-браузер.

Рівень сервісів – це самі сервіси, використовувані через хмарну модель.

Рівень додатків – це програми, доступні через хмару і не потребують інсталяції на комп'ютері користувача (в принципі – одна з головних переваг хмарної моделі).

Рівень платформи – це програмна платформа, яка об'єднує повний набір інструментів для розгортання та використання хмарних обчислень на комп'ютері користувача (без додаткових інсталяцій, купівлі обладнання та ін.) Приклад такої платформи: Microsoft.NET Azure Services Platform.

Рівень пам'яті – підтримка зберігання даних користувача і доступу до них через хмару.

Рівень інфраструктури – надання повної віртуалізованої платформи через хмару, наприклад, Amazon EC2.

Наведемо схему архітектури хмарних обчислень:

Сервіси, доступні через хмару;

Інфраструктура для їх розгортання і використання;

Платформа – набір інструментів для використання хмари;

Пам'ять – підтримка зберігання призначених для користувача даних у ЦОД, що реалізує хмару;

Архітектор хмари – це головний розробник його архітектури;

Інтегратор хмари – це його системний адміністратор, що відповідає за додавання компонентів в хмару і їх зміну;

Компоненти хмари – як правило, є Web-сервісами.

Хмара може бути загальнодоступною або приватною (корпоративною).

При використанні хмарних обчислень дещо змінюються і ролі фахівців, що беруть участь у них. Постачальником хмари є центр обробки даних. Користувачами хмари можуть бути будь-які користувачі Інтернету. Виробник устаткування або програмного забезпечення – це компанія, що забезпечує розробку апаратури і програмного забезпечення для центру обробки даних. Модель хмарних обчислень заснована на дотриманні цілого ряду стандартів:

стандарти HTTP (основний Web-протокол) – для взаємодії додатків;

XMPP (Jabber) – стандарт для відправки і отримання миттєвих повідомлень;

SSL (Secure Socket Layer) – рівень безпечних сокетних мережевих з'єднань.

Для роботи клієнтів у хмарі використовуються Web-браузери (з активним використанням технології AJAX, що дозволяє зменшити число перенаправлень

однієї веб-сторінки на іншу і, тим самим, час доступу користувача до необхідної йому інформації) і offline-клієнти, робота яких заснована на HTML 5 (спеціальній версії HTML для хмарних обчислень).

Для реалізації хмари використовуються принципи віртуалізації програм і даних і стандарт OMF. Для взаємодії з сервісами дані передаються у форматі XML.

Windows Azure.

Windows Azure - хмарна платформа, розроблена фірмою Microsoft (по суті, операційна система і набір інструментів «в хмарі»). На рисунку 5.2 ілюструється роль Windows Azure в організації використання хмарних сервісів, що надаються центром обробки даних Microsoft, фірмами та індивідуальними замовниками.

Рисунок 5.2 – Windows Azure, користувачі і центри обробки даних

Зауважимо, що Windows Azure забезпечує зберігання, використання і модифікацію даних і запуск програм тільки на комп'ютерах центрів обробки даних Microsoft. Ніякого програмного забезпечення, крім Інтернет-браузера, на користувацьких комп'ютерах не потрібно.

Перспективи хмарних обчислень, незважаючи на їх критику і поки обережне ставлення до них, дуже великі, так як хмарні обчислення дозволяють отримати доступ до потужних обчислювальних можливостей і даних великого обсягу практично з будь-яких комп'ютерів користувачів (у тому числі – кишенькових, мобільних пристроїв і т. д.), що мають веб-браузер.

5.2 Особливості ОС для мобільних пристроїв

До мобільних пристроїв прийнято відносити мобільні телефони, смартфони і комунікатори. Розробники ОС для мобільних пристроїв працюють над тим, щоб наблизити можливості цих ОС до можливостей ОС для настільних і портативних комп'ютерів. Однак в ОС для мобільних пристроїв є своя специфіка, яку необхідно враховувати при розробці таких ОС.

Необхідно враховувати більш *жорсткі обмеження по пам'яті* мобільних пристроїв. Хоча мобільні пристрої активно розвиваються, за своїми параметрами (обсягом пам'яті, швидкодії процесора) вони все ж поки поступаються настільним комп'ютерам. Тому додатки для мобільних пристроїв, які потребують великого обсягу пам'яті, відтворюються на мобільних пристроях з неповними можливостями. Ряд інструментів, наприклад, Java, також доступні для мобільних пристроїв в спеціальних версіях, розроблених з метою економії пам'яті, з обмеженнями, не властивими класичним версіями: на мобільних пристроях працює Java Micro Edition (JME), а не повна версія – Java Standard Edition (JSE) для настільних комп'ютерів. У ній, наприклад, відсутня дійсна арифметика в Java і ряд інших важливих можливостей.

2. Необхідно враховувати порівняно *невелику швидкість процесору*. порівнянні з настільними комп'ютерами, аналоги настільних додатків на мобільних пристроях працюють помітно повільніше: наприклад, повільно відкривається зображення на екрані мобільного телефону, текстовий файл для перегляду і т.д. Це незручно для користувачів і, мабуть, буде подолано в наступних версіях самих мобільних пристроїв і їх ОС.

Необхідно враховувати *особливості екранів і екранних навігаторів* конкретних моделей мобільних пристроїв. Багато типів мобільних пристроїв різних фірм мають суттєво різні екрани і різні види екранних навігаторів (трекбол та ін.) При розробці ОС і сервісних програм для мобільних пристроїв ці відмінності доводиться враховувати, що ускладнює розробку програмного забезпечення.

Необхідно враховувати *сумісність з основними форматами файлів*: .doc / docx, .ppt / .pptx,. pdf,. jpg та ін При роботі на мобільному пристрої необхідно мати можливість візуалізувати, редагувати і створювати файли тих же звичних форматів, що і на настільних комп'ютерах. Забезпечення такої сумісності – одне з важливих завдань ОС і сервісних програм для мобільних пристроїв.

Мультимедійні можливості: малюнки, відео, аудіо, обмін мультимедійними повідомленнями. Обробка мультимедійної інформації для мобільних пристроїв особливо важлива: користувачі повинні мати можливість зробити фотографії, зняти відео, переглянути їх на мобільному пристрої, опублікувати в Інтернеті, послати на настільний комп'ютер або у вигляді повідомлення своїм кореспондентам.

Підтримка комунікаційних і мережних технологій: Wi-Fi / WiMAX, Bluetooth, GPRS, EVDO, GSM, CDMA. Мобільний пристрій і його ОС повинні забезпечувати для користувачів можливість сучасних видів комунікації в бездротових мережах.

5.2.1 Короткий огляд ОС для мобільних пристроїв

даний час на ринку мобільних пристроїв використовується кілька десятків найбільш поширених ОС. Деякі з них розроблені на основі вільно поширюваного ядра Linux. Провідні фірми-виробники мобільних пристроїв підтримують власні ОС або ОС, придбані разом з їх фірмами-розробниками. Найбільш поширені ОС для мобільних пристроїв наступні:

- Nokia Symbian OS
- Google Android
- Windows Mobile
- Blackberry OS.
- Apple iPhone OS
- Samsung Bada
- PalmOS.

Windows Mobile (з 2011 року не розробляється і не підтримується).

Windows Mobile – сімейство ОС для мобільних пристроїв фірми Microsoft. Воно відноситься до сімейства Windows CE (Consumer and Embedded) – Windows для вбудованих систем. Ядро ОС Windows Mobile заснована на ОС Windows CE.

У США Windows Mobile – третя за популярністю ОС для мобільних пристроїв (після Blackberry OS і iPhone OS). Windows Mobile підтримувала наступні види мобільних пристроїв: PocketPC, смартфони, комунікатори (наприклад, Qtek). Перша версія Windows Mobile була випущена в 1996 р.

Windows Mobile надавала різноманітний набір можливостей і програмне забезпечення:

Office Mobile – аналог Microsoft Office для мобільних пристроїв; повна сумісність за форматами;

Windows Media Player – мультимедійний програвач, аналог програвача для настільної версії Windows;

Internet Explorer Mobile – Web-браузер, аналог Internet Explorer для настільної версії Windows;

Програмне забезпечення для підтримки Bluetooth і Wi-Fi – сучасних видів комунікації;

Програмне забезпечення ***Microsoft ActiveSync*** для синхронізації даних з настільними комп'ютерами.

Windows Mobile підтримувала користувацький інтерфейс з мобільним пристроєм за допомогою торкання екрана стайлусом і пальцями, в тому числі (в сучасних версіях) – multi-touch.

Останні версії Windows Mobile підтримували також .NET Compact Framework, що давало можливість виконання додатків для платформи. .NET на мобільних пристроях.

Версії Windows Mobile 6 (з 2011 року не розробляється і не підтримується)**.** Починаючи з версії 6, Windows Mobile мала три основні версії:

Windows Mobile 6 Classic – для КПК

Windows Mobile 6 Professional – для комунікаторів

Windows Mobile 6 Standard – для смартфонів.

Symbian OS – найбільш поширена в певний час ОС для мобільних пристроїв, розроблена консорціумом Symbian (Nokia, Ericsson, Psion, Motorola), що заснований у 1998 р. Фірма Nokia фінансувала об'єднання Symbian Foundation, метою якого була розробка і підтримка єдиної (для мобільних пристроїв різних компаній) мобільної платформи на основі Symbian OS. Symbian OS розроблена на основі ОС Psion EPOC32 (фірми Psion). Мова реалізації системи – C + +; є також підтримка Java. Найбільш поширені версії – Symbian OS Series 60 2nd edition; 3rd edition. Наприкінці 2012 року закрився веб-сайт «symbian.nokia.com».

Google Android – стек додатків для мобільних пристроїв, який включає операційну систему (на базі ядра Linux), проміжне програмне забезпечення (middleware) і сервісні програми. Система Android розроблена фірмою Android,

Inc., яка придбана компанією Google (2005). Важливою особливістю Google Android є те, що сервісні програми та бібліотеки цієї системи написані на Java.

Можливості Google Android. Google Android має зручний і естетичний інтерфейс користувача, який розроблений з використанням двовимірної і тривимірної графіки (бібліотеки OpenGL). Основні можливості системи наступні:

СУБД SQLite для зберігання даних;

Підтримувані мережеві технології: GSM / EDGE, IDEN, CDMA, EV-DO, UMTS, Bluetooth, Wi-Fi, WiMAX;

Обмін повідомленнями SMS та MMS;

Web-браузер на базі WebKit Application Framework.

Підтримка Java. Фірма Google з принципових міркувань використовує в системі Android власну реалізацію Java – Dalvik Virtual Machine, розроблену спеціально для мобільних пристроїв. На думку фахівців Google, Стандарт Java Micro Edition (JME) застарів, тому що розрахований на застарілі типи мобільних пристроїв і їх технічні можливості. Тому в Google Android стандарт JME не підтримується.

Підтримка мультимедіа. В системі Google Android є кодеки для всіх поширених мультимедійних стандартів, програмне забезпечення для обробки мультимедійних файлів і взаємодії з відео-і аудіо пристроями.

Підтримка розробки додатків. Система Google Android має своє власне інтегроване середовище для розробки додатків – Android SDK, що включає емулятор мобільних пристроїв, засоби налагодження, профілювання, а також plug-in до популярного середовища Eclipse для розробки Java-додатків.

Перспективи розвитку ОС для мобільних пристроїв.

зв'язку з активним розвитком мобільних пристроїв, операційні системи для них мають великі перспективи розвитку за такими напрямками:

• Покращення та спрощення інтерфейсу користувача.

• Покращена графіка.

• Більш широкі мультимедійні можливості.

• Розвиток набору сервісних та ігрових програм.

• Забезпечення повної сумісності з настільними комп'ютерами і з використовуваними на них форматами файлів.

• Продовження і розвиток використання платформи Java для мобільних пристроїв; всі провідні виробники мобільних пристроїв підтримують платформу Java, що є гарантією розвитку самої Java-технології.

Розвиток самих мобільних пристроїв: прискорення процесорів, збільшення обсягу пам'яті, реалізація нових швидких комунікаційних технологій, і підтримка цих нових можливостей в ОС для мобільних пристроїв.

5.2.2 Розквіт ОС в 2000-х роках

даний час ми є свідками небувалого розквіту операційних систем.

Безперечним лідером у цій галузі є корпорація Microsoft, що випустила менш ніж за 10 недавніх років цілу серію ОС сімейства Windows: Windows XP, Windows 2003, Windows Vista (2007), Windows 2008, Windows 2008 High-Performance Computing (HPC), Windows 7, Windows 8, Windows 10 з постійними автоматичними оновленнями.

Розвиваються також діалекти ОС Linux (Red Hat, Fedora, Mandrake, Ubuntu, SuSE та ін. – сотні діалектів). Linux – операційна система типу UNIX, ядро якої вільно поширюється з вихідними кодами.

Фірма Sun (у 2010 р. увійшла до складу фірми Oracle) розробляє і випускає ОС Solaris – одну з найбільш сучасних ОС типу UNIX з розвиненою підтримкою паралельного програмування, новими видами файлових систем, що відрізняється своєю підвищеною надійністю.

Існує також багато інших операційних систем. У США і Канаді, як відомо, найбільш популярні комп'ютери сімейства Macintosh фірми Apple (коротко – Mac) зі своєю операційною системою MacOS, яка є законодавцем мод в області графічних користувацьких інтерфейсів (GUI) і обміну мультимедійною інформацією (наприклад, мовного введення). Назвемо також ОС фірми IBM для суперкомп'ютерів і комп'ютерів загального призначення (mainframes).

Особливо важливо те, що в даний час багато з них (або їх великі частини, наприклад, ядро) доступні з відкритим вихідним кодом.

Корпорація Microsoft поклала початок цьому руху в 2003 р., коли була оголошена академічна програма Windows Embedded Shared Source – був відкритий вихідний код Windows для вбудованих систем. А в 2006 р. відбулася зовсім неймовірна досі в програмістському світі подія – Microsoft відкрила «святу святих», вихідний код ядра ОС Windows сімейства NT (NT/2000/XP/2003/2008/7) і надала в розпорядження університетів та академічних організацій Windows Research Kernel (WRK) -

самодокументований вихідний код «дослідницького» ядра Windows. Тепер кожен студент, викладач і дослідник мають можливість вивчати систему Windows «зсередини» і навіть розвивати її, але тільки для цілей навчання і досліджень, а не для комерції.

Фірма Sun (нині – Oracle) поклала початок аналогічної ініціативи для ОС Solaris – кілька років тому було започатковано проект OpenSolaris. Результати цього академічного проекту використовуються при випуску нових версій комерційної ОС Solaris.

За традицією, ще з початку 1990-х рр., ядро ОС Linux також поширюється вільно, з вихідними кодами, що викликало цілу хвилю робіт зі створення нових діалектів Linux, а також з розробки нових ОС для мобільних пристроїв на базі ядра Linux (наприклад, ОС Google Android).

Також інтенсивно розвиваються ОС для мобільних пристроїв. Ще кілька років тому найбільш використовуваними ОС в цій області були ОС сімейства Symbian. Однак зараз ОС Microsoft Windows Mobile і Google Android активно тіснять Symbian з ринку.

ОС для хмарних обчислень – принципово новий вид ОС, що відображає сучасну тенденцію до організації обчислень, таких як хмарних (cloud computing). Хмара – це метафора Інтернету. При хмарних обчисленнях користувач зі свого комп'ютера одержує платний доступ через Інтернет до Web-сервісів, які працюють на комп'ютерах потужних центрів обробки даних (наприклад, на серверах Microsoft). При цьому не тільки використовуване програмне забезпечення (у вигляді набору Web-сервісів), а й самі оброблювані дані користувача зберігаються на серверах «хмарного» центру обробки даних. На своєму комп'ютері користувач має лише простий, зручний «хмарний» Web-інтерфейс, що не вимагає великих ресурсів. Найбільш поширеною ОС для хмарних обчислень є в даний час Microsoft Windows Azure.

Ось лише дуже короткий огляд розвитку операційних систем в наші дні.

Контрольні питання

В чому полягає вимога до сучасних ОС щодо підтримки багатопоточності і багатоядерних процесорів?

Які основні вимоги висуваються при розробці сучасних ОС?

Які переваги має ОС Solaris?

В чому основні переваги ОС MacOS? Яка фірма розробник цієї ОС?

Назвіть основні переваги 64-розрядних систем?

Які основні вимоги до комп'ютера при використанні 64-бітної операційної системи?

В чому основні недоліки 64-розрядної системи?

Чи працюють 32-бітові програми під 64-бітну Windows?

Яки Ви знаєте найважливіші 64-бітові програми?

Чому Windows 8 вважається новим етапом у розвитку операційних систем Microsoft?

Що являє собою Windows Research Kernel? З якою метою використовується?

На чому засновані хмарні обчислення?

В чому недоліки «хмарного» підходу до ОС?

На яких положеннях базується концепція хмарних обчислень?

Які є рівні архітектури хмарних обчислень?

Що собою являє хмарна платформа Windows Azure?

Що необхідно враховувати при розробці ОС для мобільних пристроїв?

Назвіть найбільш поширені ОС для мобільних пристроїв? Опишіть найвдалішу на Вашу думку?

Які основні різновиди мобільних пристроїв?

Які основні особливості ОС для мобільних пристроїв?

З якими основними форматами файлів необхідно забезпечити сумісність в ОС для мобільних пристроїв?

Які комунікаційні технології підтримані в ОС для мобільних пристроїв?

Що таке Windows Mobile? Ядро якої ОС використовує Windows Mobile?

Які основні офісні і сервісні додатки надає Windows Mobile? Які сучасні версії (види) Windows Mobile використовуються в даний час?

Що таке Symbian OS? Якою компанією і на якій мові розроблена?

Які основні можливості Symbian OS?

Що таке Google Android? Ядро якої ОС використовує Google Android?

Які основні можливості Google Android?

Яку реалізацію Java використовує Google Android?

Які інструменти розробки додатків для Google Android?

ЧАСТИНА ІІ

ПРАКТИКУМ[1]

Практична робота N1.

Робота з файлами з командного рядка LINUX

Мета: навчитися працювати з файлами в Linux.

Завдання. Наступні дії виконати з командного рядка Linux:

Створити з командного рядка Linux каталог.
Створити файл у цьому каталозі.
Переглянути файл. Створити інший каталог.
Скопіювати файл в цей інший каталог.
Перейменувати файл.
Створити інший файл.
Об'єднати два створені файли.
Перемістити файл в попередній каталог.
Переглянути файл.
Змінити права доступу до файлу.
Переглянути каталог.
Переміститись від одного каталогу до попереднього з командного рядка.
Знищити файли в створених вами каталогах.
Знищити створені вами каталоги.

Теоретичні відомості

Всі імена файлів, каталогів і команд чутливі до великих і малих букв, на відміну від MS-DOS. Наприклад, команда `make` відрізняється від `Make` або `MAKE`. Імена файлів можуть бути довші 8 символів і містити точку або кілька точок на будь-якій позиції.

Отримання довідки по LINUX з командного рядка: *#man man* (Enter). Кожна сторінка довідки – це окремий HTML – файл. Всі вони знаходяться в каталозі */usr/man*, мають формат *man№.gz*, де *№* – порядковий номер файлу, *gz* означає, що файл заархівований командою *gzip*.

Команда pwd . Вона виводить робочий каталог і показує, в якому місці файлової системи ви знаходитесь. Наприклад, якщо після команди *# cd /usr/bin*

[1] При підготовці даного розділу були використані літературні джерела [1–4], [21–25] та ін.

набрати # *pwd*, то на екрані з'явиться */usr/bin*. Для відображення лише поточного робочого каталогу набирають # *pwd –help*. Якщо немає впевненості, в якому місці каталогу ви знаходитесь, запустіть # / *bin/pwd –help*.

Команда find. Ця команда шукає файли на диску. Наприклад, для пошуку команди *spell* на диску в каталозі */usr* можна використати таку команду: # find */usr –name spell –print*. Для пошуку програми в каталозі /usr/bin, до якої не зверталися останні 100 днів, набирають: # *find /usr/bin – type f –atime +100 – print*. Якщо треба знайти файли в каталозі, які з'явилися за останню добу, набирають: *# find /usr/bin –type f –mtime -1 –print.* Ця команда працює також з шаблонами файлів * *і ?*. Наприклад, для виведення списку всіх файлів з розширенням *.ps* з каталогу */usr*, набирають: #find /usr –name '*.ps' –print. Разом командою find застосовують опцію –xdev, яка обмежує пошук рамками поточної файлової системи (LINUX), а не Windows і Dos. Наприклад, вам треба знайти лише в вашому розділі windows всі файли з розширенням *.sys* в каталозі */mnt/dos:*

*find /mnt/dos –name *.sys –print –xdev.*

Команда whereis. Теж використовується для пошуку файлів. Вказує на тип файлу (двійковий чи вихідний) і на розміщення сторінки довідки до цього типу файлів. Наприклад, # *whereis find* (Enter) дає результат: *find: /usr/bin/find /usr/man/man1/find.1.gz.* Тобто, відображується інформація про те, що команда *find* є в каталозі */usr/bin*, а її сторінка довідника – в каталозі */usr/man/man1*. Якщо шукають лише двійкову версію програми, то набирають: # *whereis –b find* (Enter). Результат: *find:/usr/bin/find*. (показує місцезнаходження лише файлу без документації). Якщо файл не знайдено, видається *loo:*. *Whereis* швидша за *find*.

Команда locate. Ця команда швидша за *whereis*. Вона не переглядає каталоги, а користується файлом імен всіх файлів. Наприклад: *# locate *.ps*. База даних за об'ємом ~700 000 символів з іменами всіх файлів є в папці */var/state* і оновлюється командою # *updatedb* (для цього користувач повинен увійти в систему як *root* –оператор).

Команди whatis і apropos. Ці команди беруть резюме з сторінки документації за іменем програми. Наприклад, *#apropos search* показує всі програми LINUX, які дозволяють шукати файли за ознакою (в тому числі *find, whatis, apropos*). Якщо набрати *# whatis who* (Enter), то екран стане чистим і з'явиться рядок : *who (1) – show who is logged on.* Для виходу з *whatis* натискають *Q*. Перед користуванням командою *whatis* в режимі *root-*

користувача командою *#makewhatis* оновлюють базу даних резюме. Файл резюме є в */usr/man/man1*.

Команда mkdir. Створює новий каталог. Приклад: `mkdir /home/Peter/test` створює каталог `test` в каталозі `/home/Peter`. Синтаксис: `mkdir <dir1>...<dirN>`, де

`<dir1>...<dirN>` - створювані каталоги.

Команда rmdir. Вилучає пусті каталоги. Поточний робочий каталог повинен знаходитись зовні каталогу, що вилучається. Синтаксис: `rmdir <dir1>...<dirN>`, де `<dir1>...<dirN>` – каталоги, що вилучаються. Приклад: `rmdir /home/Peter/papers` вилучає каталог `/home/Peter/papers`, якщо він пустий.

Команда переміщення по дереву каталогів: cd. Команда змінює поточний робочий каталог. Після відкриття LINUX ви знаходитесь у домашньому каталозі, ім'я якого складається з */home/*, за яким слідує ім'я користувача. Наприклад, команда *# cd /usr/bin* переміщує вас в каталог /usr/bin. Щоб попасти каталог /usr, треба набрати # cd .., а для попадання в кореневий каталог – # cd ../... Повернутися в домашній каталог можливо командою # cd без аргументів або # cd ~. Символ "." посилається на поточний каталог, ".." - на попередній в дереві каталог. Наприклад, #cd ../loo переводить з поточного каталогу в *../loo*.

Команда перегляду вмісту каталогів: ls. Видає інформацію про файли в каталозі.

Синтаксис: `ls <file1>...<fileN>`, де `<file1>...<fileN>` імена файлів або каталогів, інформацію про які треба показати на екрані.

Опції. Найчастіше використовується: *`-F`* (для представлення інформації про типи файлів), та *`-l`* (видає в довгому (*long*) форматі інформацію про розміри файлів, їх власників, права доступу і т.д.).Приклад: *`ls -lF /home/Peter`* видасть вміст каталогу */home/Peter*. Риска зліва від назви означає, що ця назва – каталог. Якщо такої риски немає, то це – файл.

Команда rm. Вилучає файли. В LINUX вилучені файли відновити неможливо. Синтаксис: `rm <file1>...<fileN>`, де `<file1>...<fileN>` імена файлів для вилучення. Опція `-i` вимагає вашої згоди на вилучення файлу. Приклад: `rm -i /home/Peter/joe /home/Peter/frog` вилучає файли `joe` та `frog` в каталозі `/home/Peter`.

Команда cp. Копіює файл(и) в файл або каталог. Синтаксис: `cp <file1>...<fileN> <destination>`, де `<file1>...<fileN>` імена файлів, що копіюються, а `<destination>` – файл або каталог, в який копіюють. Приклад: `cp ../frog joe` копіює файл `../frog` в файл або каталог `joe`. Кілька файлів однією командою *cp* можна скопіювати лише в каталог; в файл можна скопіювати лише один файл. Для посилання на поточний каталог використовують ".". Наприклад:

```
/home/Peter/loo# cp /etc/termcap .
 /home/Peter/loo# cp /etc/shells .
 /home/Peter/loo# ls -F
 shells     termcap
```

***Команда mv*.** Переміщує файл(и) в другий файл або каталог. Ця команда не еквівалентна копіюванню з наступним знищенням оригіналу. Застосовується для перейменування файлів, як команда `RENAME` з MS-DOS. Синтаксис: `mv <file1>... <fileN> <destination>`, де `<file1>...<fileN>` - імена переміщуваних файлів, а `<destination>` - ім'я файлу або каталогу, в який переміщують.

Приклад1: `mv ../frog joe` переміщує файл `../frog` в файл або каталог `joe`.

Приклад2:
```
/home/Peter/loo# mv termcap sells
              /home/Peter/loo# ls -F
        bellssellsshells
```

Видно, що *termcap* більше не існує, а на його місці файл `sells`.

Зауваження. Команди `mv` и `cp` знищують вміст файлу, в який вони пишуть (якщо він є на диску) за замовчуванням. Тому при перенесенні файлу в інший каталог перевірте, чи існує в ньому файл з таким же ім'ям.

Команда more. Видає вміст названих файлів поекранно.

Синтаксис: `more <file1> ... <fileN>`, де `<file1> ... <fileN>` відображувані файли. Приклад: `more papers/final` представляє файл `papers/final`.

Команда cat. Використовується для конкатенації і для видачі повного вмісту файлу за раз. Синтаксис: `cat <file1> ... <fileN>`, де `<file1> ... <fileN>` файли до видачі.

Приклад: `cat letters/mdw` видає на дисплей файл `letters/mdw`.

Команда grep. Видає всі рядки у вказаних файлах, які містять заданий зразок.

Синтаксис: `grep <pattern> <file1> ... <fileN>`, де `<pattern>` - зразок і `<file1> ... <fileN>` - файли, в яких відбувається пошук. Приклад: `grep loom /etc/hosts` видасть всі рядки, в яких файл `/etc/hosts` містить зразок ``loom''.

Права доступу та їх зміна.

Команда `ls` з опцією `-l` видає на екран список файлів певного каталогу в "довгому" форматі з інформацією про права доступу:

```
/home/Peter/loo# ls -l stuff
-rw-r--r-- 1 Peter users 505 Mar 13 19:05 stuff
```

Перше поле в цьому рядку – це права доступу (по порядку вказано права власника, групи і всіх інших). `Peter` – власник файлу, `users` –група, `stuff` – ім'я файлу. Символ «-» - це тип файлу (звичайний файл (не каталог не спеціальний файл якого-небудь пристрою). Наступні 3 символи "`rw-`" – це права доступу `Petera`. Символ «*r*» означає «*read*» (читати), «*w*» – «*write*» (писати). Отже, `Peter` може читати файл `stuff` і писати в нього, але виконувати цей файл йому заборонено (право виконання надається після «*rw*» третім символом «*x*», тобто «*execute*» (виконати). Наступні 3 символи `r--` є правами доступу групи *users* для цього файлу (лише читання). Всім іншим дозволено те ж саме («r-- » – лише читання).

Приклад1: **-rwxr-xr-x (** власник файлу може читати, писати і виконувати файл. Члени групи і всі інші користувачі можуть читати і виконувати файл).

Приклад2: **-rw------- (**власник файлу може читати і писати в файл. Всім іншим доступ до файлу закрито). *Приклад3:* **-rwxrwxrwx** (всі можуть читати, писати і виконувати файл).

Права доступу змінюються командою з таким синтаксисом: `chmod {a,u,g,o}{+,-}{r,w,x} <filenames>`, де параметр *a* означає всі, *u* – користувач, *g* – група, *o* – інші, + означає надати права, - означає забрати права. Наприклад: **chmod a+r stuff** надає всім користувачам право читати файл *stuff*; **chmod og-x stuff** - забирає право на виконання у всіх, крім власника; **chmod u+rwx stuff** - дозволяє власнику всі дії з файлами; **chmod o-rwx stuff** - забороняє всі дії з файлами користувачам категорії *other*.

Контрольні запитання

Дайте визначення операційній системі. Які основні функції операційної системи?

Як класифікуються ОС за призначенням?

Із яких компонентів складається типова ОС?

Які основні принципи побудови ОС ієрархічної структури?

Що таке програмне середовище?

Що таке операційне середовище?

Що таке API ОС?

Якими командами в Linux створюється каталог, вилучається каталог, здійснюється перехід від одного каталогу до іншого, копіюється каталог?

Якими командами в Linux створюється файл, вилучається файл, здійснюється перегляд файлів у каталозі, перейменовується файл, копіюється файл?

Якою командою змінюються атрибути файлу в Linux?

Практична робота № 2

Створення командних файлів в Linux у вигляді Perl-скриптів

Мета: оволодіти навичками написання Perl-скриптів в Linux.

Завдання.

Скласти блок-схему алгоритму роботи Perl-скрипта.

Скласти програму на мові Perl згідно з варіантом завдання.

Вивести результати на екран.

Варіанти.

Написати Perl-скрипт, який вилучає каталог, ім’я якого передається через командний рядок. Якщо скрипт запустили без параметрів, то передбачити видачу сповіщення про правильний синтаксис його виклику. (Вказівка: спочатку вилучити всі файли з каталогів, вкладених в даний каталог (якщо такі каталоги є)).

Написати Perl-скрипт копіювання одного файла в другий. Передбачити введення імен файлів як з командного рядка, так і з екрану монітора.

Написати Perl-скрипт копіювання вмісту одного каталогу в інший каталог. Передбачити введення імен каталогів як з командного рядка, так і з екрану монітора.

Написати Perl-скрипт читання рядка текстового файлу з заданим номером. Передбачити випадки, коли номер заданого рядка більший за число рядків в файлі. Якщо номер рядка – від’ємне число, то прочитати всі рядки, починаючи з рядка з номером, рівним абсолютному значенню введеного від’ємного числа.

Написати Perl-скрипт, який замінює всі файли з розширенням *.mtx* в заданій директорії ОС Linux на файли з розширенням *.txt*. Файлів з розширенням *.xtx* в заданій директорії повинно бути не менше 5.

Написати Perl-скрипт, який перевіряє наявність файлу *index.htm* в директорії з *HTML*-файлами і при його наявності видає його вміст з допомогою браузера на екран. Тобто треба його повністю зчитати і вивести на екран.

Написати Perl-скрипт, який створює, перейменовує і вилучає файл *stroki.txt.*

Написати Perl-скрипт, який виводить вміст файлу *data.txt* на екран у вигляді *html*-файлу: (Вказівка. Для виведення даних з Perl-скрипту у виді html-файла треба вказати на це браузеру таким рядком*: print "Content-type: text/html\n\n*"; Починаючи з нього браузер будує *html*-документ зі змісту команди *print.* Отже, додавання в perl-скрипт рядка *print "<html><head>"* додає *html*-документ, що формується, рядок *<html><head>).*

Написати Perl-скрипт, який здійснює запит і введення імені користувача, порівняння з поточним логічним ім'ям користувача і виведення сповіщення: правильно/неправильно.

Написати Perl-скрипт, який переходить в інший каталог, формує файл з лістингом каталогу і повертається у вихідний каталог.

Написати Perl-скрипт, який виводить все відомості про вказаний файл або сповіщення про помилку, якщо файл не знайдено. Ім'я файлу передається з допомогою форми.

Написати Perl-скрипт, який виводить вказаний рядок файлу з вказаним зміщенням, тобто починає виведення не з початку. Використовуйте функції Seek і Getc для написання скрипта.

Написати Perl-скрипт, який створює в директорії з HTML документами HTML- файл, в якому записано таблицю Піфагора (10x10).

Написати Perl-скрипт, який виводить на екран список всіх файлів заданого каталогу, до яких більше 30 днів ніхто не звертався.

Написати Perl-скрипт, який вилучає всі файли заданого каталогу, до яких більше 30 днів ніхто не звертався.

Теоретична частина

ОС Linux має вбудовану мову програмування Perl (Practical Extraction and Report Language). Perl безкоштовна, структурована, різностороння і гнучка мова програмування, на якій написано більшість скриптів для Web. Perl інтерпретована, тобто Perl-програми не треба компілювати. Perl має широкий спектр інструментів для роботи з файлами.

На Perl знак # (дієз) означає коментар (ігнорується текст після дієзу до кінця рядка).

Перший рядок (#!*c:/usr/bin/perl або #!c:/usr/local/bin/perl*) будь-якого Perl-скрипта дає команду Linux запуску скрипта інтерпретатором Perl,

інстальованим за адресою *c:/usr/bin/perl* чи *c:/usr/local/bin/perl*. Цей перший рядок є коментар, але без нього скрипт може не спрацювати.

Застосування PERL-дебагера. В ньому легко по рядкам перевіряти роботу скриптів. Perl-дебагер вбудовано в Perl і запускається з командного рядка так: *C:\PERL> Perl -d hello.pl <ENTER>*. Perl завантажить скрипт *hello.pl* почне відлагодження. В Linux також можна написати рядок *#!/usr/bin/perl -d* в самому початку Perl-скрипта. Загрузка Perl-дебагера без загрузки скрипта: *C: \> Perl -de 0 <ENTER>* (ключ *-d* вказує Perl запустити дебагер, а аргумент *-e 0* - виконати Perl-скрипт з 0 рядків). При нормальній роботі Perl-дебагера на екран виводиться: *Loading DB routines from $RCSfile: perldb.pl,v $$Revision: 4.0.1.3 $$Date: 04/09/09 13:43:57 $ Emacs support available. Enter h for help. main '(p1000159:1): DB<1>*. Якщо виводиться *"Can't locate perldb.pl @INC"*, то Perl треба перевстановити. Команди дебагера: - *h* – виведення списку команд дебагера, - *n* – виконувати до наступного виразу; - *<CR>* - повторити останню команду *n* або *s*, - *p* вираз – друкувати вираз, - *q* – закінчити роботу, - *r* – виконувати до виходу з підпрограми, - *s* – крок у скрипті (з входом в підпрограму).

Приклади виведення на екран сповіщення *Hello Friends*:

```
1) DB<1> prinf "Hello Friends\n"; <ENTER> Hello Friends DB<2> ;
2) <ENTER>        DB<2> p "Hello Friends\n" <ENTER> Hello Friends  DB<3>.
```

Для введення кількох рядків в дебагер в кінці кожного рядка вставляють символ продовження рядка *</>: DB<3> for ($i = 0; $i < 10; $i++){\<ENTER>cont: print $i;\<ENTER> cont: } <ENTER> 0123456789 DB<4>*.

Скалярні змінні. Тип числових або символьних даних вказують символом$: $ім'я_скаляра = 4; (скалярній змінній $ім'я_скаляра присвоєно число 4). Скалярні змінні можна додавати і віднімати.

Список – це впорядковані скаляри, записані через кому в круглих дужках*:(5,$x,12,2+$y).*

Масив – це змінна, яка містить список. Перед змінною-масивом ставиться знак @:

@ім'я_масиву = *(5, $x, 12, 2+$y);*.

Доступ до елемента масиву отримують так: *$ім'я_масиву [номер_скаляра];* Номер_скаляра відраховується з нуля. Отже, звернення до змінної *$x* в цьому масиві таке: *$ім'я*_масиву [1];

Хеш – це асоціативний масив. Кожний елемент хеша має свій "ключ" доступності. Назва хешу починається з символу %. Додають новий елемент в хеш так: *$хеш {"ключ"} = "значення"*. Тепер %хеш містить "значення". Видобувають елемент "значення" так: $хеш {"ключ"}.

Керуючі структури. Всюди далі скорочення «оп» означає «оператор». Нижче наведено синтаксис керуючих Perl-структур:

Керуюча структура if/else:
if (умова_1) {
оп_1; оп_2; … # ці оператори виконуються, # якщо умова_1 вірна
} else {
 	
оп_3; оп_4; … # ці оператори виконуються,
#якщо умова_1 хибна
}
	Керуюча структура for:
	for (вираз_1; умова_1; вираз_2) {
	оп_1; оп_2; ...
	}
	# вираз_1 виконується 1 раз на початку циклу
	# блок операторів виконується, поки #умова_1 вірна
	# вираз_2 виконується в кінці кожного #циклу.

Керуюча структура while:
while (вираз_1) {
оп_1; оп_2; ...
}
# блок операторів виконується, 	
# поки вираз_1 вірний. 	
	Керуюча структура until:
	until (вираз_1) {
	оп_1; оп_2;
	}
	# блок операторів виконується,
	# поки вираз_1 хибний.

Керуюча структура do/while: do {оп_1; оп_2; } while вираз_1; # блок операторів виконується 1 раз. # далі блок виконується, поки вираз_1 вірний.	***Керуюча структура do/until:*** do {оп_1;оп_2; } until вираз_1; # блок операторів виконується 1 раз. # далі блок виконується поки вираз_1 хибний.
Керуюча структура foreach: foreach $x (@list) {оп_1; оп_2; ...} скалярній змінній $x по черзі присвоюються #значення зі списку @list. З кожним наступним значенням $x виконується #блок операторів.	

Функції користувача. Свою функцію пишуть в кінці скрипта. Її синтаксис такий:

sub ім'я_функції {оп_1; оп_2; ...} . Викликається вона вставкою в скрипт рядка: і'мя_функції();

Зарезервовані змінні

Для використання в Perl довгих імен змінних в заголовку скрипта пишуть *use English;*

В Perl є змінні лише для читання (значення в них не записати не можна):

**$_** – в неї за умовчанням можна вводити, присвоювати, заносити результат пошуку за заданим зразком: *while($_= <>){…}* ідентичне за дією *while(<>){…}.*

**$<digit>** – доступна лише для читання, як і змінні *$&, $`, $' і $+*.

$. – містить номер останнього прочитаного рядка з останнього прочитаного файлу. Доступна лише для читання.

$/ – містить символ розділення записів, що вводяться. За замовчуванням містить символ переведення рядка.

$| – містить 0 за умовчанням. Якщо в ній не 0, очищаються буфери кожний раз після виведення на друк, екран і т.п..

$, – містить символ-розділювач полів для оператора друку.

**$** – містить символ-розділювач записів для оператору друку. Задають *$* замість набирання *n* в кінці друку.

$" – подібна $, використовується при зверненні до списку величин в подвійних лапках. За замовчуванням містить символ пробіл.

**$;** – містить символ-розділювач для емуляції багатовимірних хешів. При зверненні до елементу хеша як *$toy{$a,$b,$c},* реально виконається *$toy{join($;,$a,$b,$c)}.* Не плутати з *@toy{$a,$b,$c},* що таке ж за дією, як *($toy{$a},$toy{$b},$toy{$c}).* За замовчуванням містить значення \034.

$# – формат для друку чисел (містить початкове значення %.20g).

$% – містить номер поточної сторінки, що виводиться.

$= – містить число рядків для друку поточної сторінки (звичайно містить значення 60).

$- – містить число ще не надрукованих рядків сторінки для каналу виведення при друкуванні.

$~ – містить ім’я поточного формату сповіщень (звичайно – ім’я дескриптора файлу).

$^ – містить ім’я поточного формату заголовку сторінки (звичайно – ім’я дескриптора файлу з _TOP в кінці).

$: – містить символи, після яких виведення рядка починається з нового рядка.

$! – якщо це змінна-число, то містить поточне значення *errno* (номер помилки). Якщо це змінна-рядок, то містить відповідне системне сповіщення про помилку.

$@ – містить сповіщення про синтаксичну помилку при виконанні останньої команди *eval()*. Сповіщення в ній не накопичуються. При виконанні *eval()* без помилок змінна містить 0.

$$ – містить ідентифікатор поточного процесу.

$< – містить ідентифікатор користувача (UID), якому належить поточний процес.

$> – містить ефективний UID поточного процесу.

$(– містить ідентифікатор групи (GID) користувача, якому належить поточний процес.

$) – містить ефективний GID поточного процесу.

$0 – містить ім’я файлу скрипта.

$ARGV – містить ім’я поточного файлу, з якого відбувається читання.

@ARGV – містить масив аргументів командного рядка, які були передані скрипту.

Наприклад, з Perl-скрипта можна отримати доступ до аргументів командного рядка шляхом запису в Perl-скрипті аргументів командного рядка в списковy змінну *@ARGV*. Для виведення аргументів командного рядка на екран: *while ($arg = shift @ARGV) { print "$arg\n"; })*

@INC - містить список точок входу в скрипт з конструкціями *do EXPR, require* та *use.*

%INC - містить входи для кожного файлу, який включається з допомогою операторів *do* або *require*. Ключами є імена файлів, а значеннями місця їх розміщення.

%ENV – містить поточне оточення процесу. Зміною вмісту хеша можна змінити оточення породженого (дочірнього) процесу.

%SIG – цей хеш використовується для встановлення обробників різних сигналів.

Дескриптори файлів

Доступ до файлу дає дескриптор – символьне ім’я в Perl-скрипті для посилання на файл, пристрій, сокет, програмний канал (дескриптор програмного каналу зв’язує процеси). Ім’я дескриптора файлу пишеться для зручності пошуку в скрипті лише великими буквами і не може бути ім’ям

зарезервованої Perl-функції. Одному дескриптору може відповідати лише один файл, хоч в різних місцях програми це може бути кожен раз інший файл.

Дескриптор файлу в Perl і в Linux – це два різних дескриптори. Дескриптор файлу в Linux – це 32-бітна адреса системної області для посилань на відкритий файл. Ця область містить інформацію про цей файл. Дескриптор файлу в Perl – це не адреса, а ім'я, придумане програмістом для посилань на файл. Отримати дескриптор файлу в Linux з Perl-програми можна командою *f ileno().*

Створює дескриптор файлу функція *open()* з двома параметрами – ім'ям дескриптора і ім'ям файлу з вказаним режимом доступу: *open(DESCRYPTOR, "> /temp/myfile.txt");* #open створює дескриптор *DESCRYPTOR,* приєднує його до файлу */temp/myfile.txt* і відкриває файл для запису (на це вказує символ ">"). Приклад запису в цей файл значення змінної *var*: *print DESCRYPTOR $var*;. Дескриптор не можна присвоїти, зберегти в змінній або передати як параметр в функцію, оскільки дескриптор – не змінна (його ім'я не містить префікса змінної *Perl ($, @ або %)*). Для цих дій перед ім'ям дескриптора пишуть префікс * – посилання на глобальний тип даних. Наприклад, записати в файл значення змінної можна ще й так:*$name1 = *DESCRYPTOR; print $name1 $var*; #змінна *$name1* заміщує *DESCRYPTOR,* в який виводиться значення змінної *$var*.

будь-якому Perl-скрипті завжди є три наперед заданих дескриптори (*STDIN, STDOUT и STDERR*), де *STDIN* зв'язаний з клавіатурою, *STDOUT і STDERR* – з екраном монітора. *STDIN* використовує функція о, якщо в командному рядку виклику Perl-скрипта немає списку файлів. STDOUT за умовчанням використовують функції print і die, а STDERR – функція *warn*. Можна перенаправляти стандартне введення і виведення в інші файли дописуванням префіксів *'>'*(в файл) і *'<'*(з файлу): *perl script1.pl <in.dat >out.dat*; – при виконанні *script1.pl* дані з файлу *in.dat* запишуться в файл *out.dat* без показу на екрані. Перенаправлення з Perl-скрипта можливе у виді: open(*STDIN, "in.dat"); open*(*STDOUT, ">out.dat"*); *open(STDERR, ">err.dat");*
Тепер все стандартне введення/виведення йтиме крізь вказані в open() файли. Таке перенаправлення можна робити в Perl-скрипті лише раз (функція open() не повертає початкові призначення дескрипторів *STDIN, STDOUT і STDERR).*

Вибрані вбудовані Perl-функції для роботи з файлами і каталогами
binmode FILEDESCR – для читання або запису в файл за його дескриптором в бінарному режимі.

caller EXPR – видає контекст поточного виклику підпрограми. Якщо *EXPR* скаляр, то видає *TRUE* (при знаходженні функції в тілі підпрограми), *eval()* або *require(),* інакше – *FALSE*. Якщо *EXPR* список, то видає: *($package, $filename,$line)=caller*. З аргументом *EXPR* видає інформацію для використання дебагером при друкуванні карти стеку. Значення *EXPR* відмічає глибину стеку до поточного запису: ($package, $filename, $line, $subroutine, $hasargs, $wantargs) = caller($i);

chdir EXPR – змінює поточну директорію на вказану в *EXPR*, якщо остання існує. Без *EXPR* поточною стає домашня директорія. Видає *TRUE* в разі успіху, інакше - *FALSE*.

chmod LIST - змінює права доступу до файлів зі списку *LIST*. Першим аргументом списку є цифрова (за звичай вісімкова) маска доступу. Видає число файлів з успішно зміненими правами доступу. Наприклад: *$cnt = chmod 0700 'loo','tar'; chmod 700 @executables; chmod 0666 'f1', 'f2', 'f3';*

chown LIST – змінює власника або групу, якій належить список файлів. Першими двома аргументами завжди є *uid* та *gid*. Видає число успішних змін.

close FILEDESCR – закриває файл з дескриптором *FILEDESCR*. Наприклад:

open(FD5 '/usr/home/stud'); ... close FD5;

closedir DIRDESCR – закриває каталог, відкритий функцією *opendir*.

dbmopen ASSOC, DBNAME, MODE – Зв'язує *dbm(3)* або *ndbm(3)* файл з асоціативним масивом. *ASSOC* – ім'я асоціативного масиву. *DBNAME* – ім'я бази даних (без *.dir* або *.pag* розширення). Якщо база даних не існує, вона створюється з правами доступу, вказаними в *MODE*: *dbmopen(%HIST,'/usr/lib/news/history', 0600); while (($key, $val) = each %HIST){ print $key, '=', unpack('L',$val),\n;} dbmclose(%HIST);*

dbmclose ASSOC – розриває зв'язок між файлом і асоціативним масивом.

die LIST – за межами *eval()* друкує значення *LIST* в *STDERR* і виходить з програми з поточним значенням *$!*. Якщо в *$!* нуль, то приймає значення *$? >> 8*, а якщо значення *$?>>8* нульове, то *255*. Всередині *eval()* сповіщення про помилку заноситься в змінну *$@* і *eval()* переривається з невизначеним значенням. Приклад:

open(FL, "/root/rm-rf") || die "Can't open file.\n";

do EXPR – вважає *EXPR* ім'ям файлу і запускає вміст цього файлу як програму на *Perl*. Часто нею включають в скрипт бібліотечні підпрограми.

Наприклад: *do 'text.pl'*; ідентичне: *eval 'cat text.pl'*;. Підключати бібліотечні модулі зручніше функціями *use* та *require*.

eof FILEDESCR – видає 1, якщо наступне зчитування видає кінець файлу або якщо *FILEDESCR* не було відкрито. Без аргументу *eof* опрацьовує останній файл, що зчитувався. Функція мало використовується, оскільки в *Perl* оператори читання видають невизначене значення в кінці файлу.

getc FILEDESCR – читає і видає наступний символ з файлу читання, приєднаного до *FILEDESCR* або пустий рядок у випадку кінця файлу. Без *FILEDESCR* зчитує з *STDIN*.

Код нижче читає і виводить 15 байт з файлу *"file.txt": open (file,"file.txt"); while ($u<=15){ print getc(file);} continue{$u++} close file*;

link OLDFILE,NEWFILE – створює файл *NEWFILE*, приєднаний до файлу *OLDFILE*. В *UNIX* для одного файлу можна створити кілька імен. Видає 1 в разі успіху і 0, якщо інакше.

mkdir FILENAME,MODE – створює директорію з іменем *FILENAME* і правами доступу, вказаними в змінній *MODE* (вісімкове число 0777). В разі успіху видає 1, інакше видає 0 і встановлює значення змінної *$!(errno)*. Якщо задано ім’я каталогу без шляху до нього, то він створюється в поточному каталозі.

open FILEDESCR,EXPR – відкриває файл з іменем *EXPR* і ставить йому у відповідність файлову змінну *FILEDESCR* (дескриптор файлу). Без *EXPR* змінна з іменем *FILEDESCR* містить ім’я файлу. Спецсимвол (<,>,>>,+<,+>) перед ім’ям файлу визначає режим доступу до файлу при його відкритті. Наприклад: *open (FILEDESCR,"EXPR");* або *open (FILEDESCR,"<EXPR");* – відкриває файл тільки для читання (файл має існувати);

open (FILEDESCR,">EXPR"); – відкриває файл для запису в його початок зі знищенням попереднього вмісту файлу (якщо файл не існує, то створюється);

open (FILEDESCR, ">>EXPR"); – відкриває файл для дозапису в його кінець без знищення попереднього вмісту файлу (якщо файл не існує, то створюється);

open (FILEDESCR,"+<EXPR"); – відкриває файл для читання і запису без знищення попереднього вмісту файлу;

open (FILEDESCR, "+>EXPR"); – створює файл для читання і запису зі знищенням попереднього вмісту файлу;

open (FILEDESCR,"+>>EXPR"); – створює файл для читання і дозапису кінець без знищення попереднього вмісту файлу;. По закінченні роботи з файлом завжди пишуть *close (FILEDESCR);* (закривається файл *FILEDESCR*).

Спецсимволом також може бути знак ”|”, який може ставитися перед ім’ям або відразу ж після імені файлу. Знак ”|” перед ім’ям файлу означає, що воно є командою створення програмного каналу – вихід з *FILEDESCR* передасться на вхід *EXPR*. Наприклад: *open (FILEDESCR,"|EXPR");* – направити інформацію на вхід програми;. Знак ”|” після імені файлу означає, що вихід з *EXPR* передасться на вхід *FILEDESCR*, тобто з *FILEDESCR* відбуватиметься читання. Наприклад: *open (FILEDESCR,"EXPR|");* – зчитати інформацію з виходу програми. Фрагмент програми показує використання функцій *open* і *close*:

```
    open(InFile, " proba.dat") || die; # відкриваємо для читання proba.dat
      open(OutFile, ">proba.dat") || die;    # створюємо proba.dat
      $AuxFile = ">>proba.dat"; open(Aux, $AuxFile) || die; # відкриваємо для дозапису proba.dat
close(InFile); close(OutFile); close(Aux);
```

open DIRDESCR,EXPR – відкриває директорію з ім’ям *EXPR*, видає *TRUE* в разі успіху.

opendir DIRDESCR,DIRNAME – відкриває директорію з ім’ям DIRNAME. Приклад Perl-скрипта, який перевіряє, чи є всі файли каталогу двійковими (без перевірки вмісту вкладених каталогів): *#! perl –w opendir FDIR, "/usr/prog"; while ($name=readdir FDIR) {next if -d $name; # каталог print("/usr/prog/$name: двійковий\n") if -b $name; #Двійковий файл } closedir FDIR*;

print FILEDESCR,LIST – друкує рядок або кілька рядків, розділених комою. *FILEDESCR* може бути ім’ям скалярної змінної, яка містить дескриптор файлу. Без цієї змінної друк йде в вибраний канал виводу. Без змінної *LIST* друкує змінну *$_* в *STDOUT*.

printf FILEDESCR, LIST – еквівалент print *FILEDESCR, sprintf (LIST)*. Перший аргумент *LIST* інтерпретується як формат друку.

read FILEDESCR,SCALAR,LENGTH,OFFSET – зчитує *LENGTH* байт даних з *FILEDESCR* в змінну *SCALAR*. Видає число зчитаних байт або невизначеність в разі помилки. Для зчитування даних не з початку рядка вводять значення змінної *OFFSET*.

Положення вказівника зберігається. Код нижче читає і виводить 15 байт з файлу "file.txt": *open (file,"file.txt"); read("file",$u,5); print $u,"
"; read("file",$u,10); print $u; close file;*

readdir DIRDESCR, DIRNAME – для відкритого каталогу видає список імен всіх файлів каталогу. В скрипті допускаються дескриптори файлу і каталогу з однаковими іменами: *open FF, "/usr/out.dat" # дескриптор файлу opendir FF, Rusr" # дескриптор каталогу*. Приклад використання функції *readdir* для виведення на екран списку файлів з поточного каталогу: *opendir(Dir, $INC[2]) || die; while ($file = readdir(Dir)) { print "$file \n" } closedir(Dir);*, де змінна *$INC[2]* дає доступ до поточного каталогу. Змінюючи *$INC[2]* на *$ARGV[0]*, скрипт виводить на екран список файлів з каталогу, вказаному в командному рядку.

readlink EXPR – видає значення символьного посилання, якщо посилання існує. Якщо посилання не існує, то видає *fatal error* і встановлює значення змінної *$!*. За умовчанням опрацьовує змінну *$_*.

require EXPR – підключає модулі. Приклад: *require "oraperl.pm";*

rename (старе_ім’я, нове_ім’я_файлу); – перейменовує файл (видає 1 в разі успіху, інакше - 0).

rewinddir DIRDESCR – поточну позицію в каталозі встановлює на початок (можна повторно читати імена файлів каталогу, не закриваючи його). Єдиним параметром цієї функції є дескриптор відкритого каталогу.

rm FILENAME – вилучає файл або директорію з заданим ім’ям. Видає 1 в разі успіху, інакше - 0 і встановлює значення змінної *$!*. За замовчуванням опрацьовує аргумент *$_*.

rmdir DIRNAME – вилучає пустий каталог (без вкладених пустих каталогів). Якщо параметру не задано, то вставляють спеціальну змінну *$_*, яка містить значення «істина» при успішному знищенні каталогу і пояснення помилки при неможливості знищення.

seek FILEDESCR, POSITION, WHENCE – встановлює курсор в файлі, визначеному в змінній *FILEDESCR*, на позицію *POSITION* в режимі, вказаному змінній *WHENCE* (або відлік). Якщо в змінній *WHENCE* 0, то позиція починає відлік від початку файлу, якщо 1 – від поточної позиції і якщо 2, то від кінця файлу. Видає 1 в разі успіху і 0 – інакше.

select FILEDESCR – видає поточний вибраний *FILEDESCR*. Направляє виведення в *FILEDESCR*.

stat FILENAME – видає 13 - елементний масив параметрів вказаного файлу або пустий список в разі помилки. Наприклад: *($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size, $atime,$mtime,$ctime,$blksize,$blocks) = stat($filename)*; *$dev* – ім'я пристрою ;*$ino* - номер і-вузла; *$mode* - права доступу; *$nlink* - число зв'язків; *$uid* - ідентифікатор власника; $gid - ідентифікатор групи; *$rdev* - тип пристрою; *$size* розмір файлу в байтах; *$atime* - дата останнього звернення; *$mtime* - дата останньої модифікації; *$ctime* - дата останньої зміни статусу; *$blksize* - розмір блока на диску; *$blocks* - число блоків в файлі. Код виведення розміру файлу в байтах: *$u = (stat("file.txt"))[7];print $u;}.*

sysopen ДЕСКРИПТОР, ІМ'Я, ПРАПОР [, ДОЗВІЛ]; – може, крім завдання режиму відкриття файлу, задавати права доступу до файлу: режим відкриття файлу задає ПРАПОР – число як результат побітового *АБО* (|) над константами режимів з модуля Fcntl. Константи режиму доступу до файлу: *O_RDONLY* (тільки читання), *O_WRONLY* (тільки запис), *O_RDWR* (читання і запис), *O_CREAT* (створення файлу, якщо він не існує), *O_EXCL* (завершення з помилкою, якщо файл вже створено), *O_APPEND* (додавання в кінець файлу %). Права доступу (параметр ДОЗВІЛ) задаються у вісімковій системі з урахуванням поточного значення маски доступу до процесу з функції *umask()*. Якщо цей параметр не задано, то Perl використовує значення *0666* для звичайних файлів. Для каталогів і виконуваних файлів використовують *0777*. Приклад операцій відкриття файлів функцією *open ()* та *sysopen ():*

```
use Fcntl; open FF, "< file.txt"; sysopen FF, "file.txt", O_RDONLY; # лише читання open FF, "> file.txt";sysopen FF, "file.txt", O_WRONLY | O_CREAT | O_TRUNC; # лише запис (створює новий файл або очищує існуючий)open FF, ">> file.txt";sysopen FF, "file.txt", O_WRONLY | O_CREAT | O_APPEND; # дозапис в кінець (створюється, якщо не існує)open FF, "+< file.txt"; sysopen FF, "file.txt", O_RDWR; # читання/запис існуючого файлу open FF, "+> file.txt"; sysopen FF, "file.txt", O_RDWR | 0_CREAT I O_TRUNC; # читання/запис (файл очищається)
```

tell FILEDESCR – видає поточну позицію курсору у відкритому файлі *FILEDESCR*. Без аргументу опрацьовує файл, який читався останнім. Наприклад, наступний код читає 15 байт з файлу "file.txt" (довжина файлу повинна бути більше 15 байт).

```
print "<pre>"; open (file, "file.txt"); while (tell(file) < 15){
print getc(file);} print "</pre>";
```

truncate FILEDESCR, LENGTH – обрізає файл *FILEDESCR* до заданої довжини.

Приклад запису в файл file.txt рядка "This is a sample file" і вкорочення його до 20 байт: `#!/usr/bin/perl print "Content-type: text/html\n\n"; $string = "This is a sample file"; print "Write to file: $string"; open (file,">file.txt");`

`print file $string; close file; truncate ("file.txt",20); open (file, "file.txt"); $string=<file>; close file; print "Read from file: ",$string;`

unlink LIST – вилучає список файлів і видає число успішно вилучених файлів. Не вилучає каталоги, якщо ви не *root*-користувач. Безпечніше використати функцію *rmdir*.

use Module LIST – приєднує модуль до програми: *use strict qw(subs,vars,refs);*

utime Date1,Date2,список_файлів – змінює дату звернення і модифікації файлів зі списку. Перші два елемента списку повинні вказувати нове значення дати звернення і модифікації. Видає число змінених файлів.

write – створює запис, який може складатися з кількох рядків у відповідному файлі, з використанням формату, що відповідає цьому файлу. Формат для поточного каналу виведення встановлюється присвоєнням змінній *$~{ }* назви формату.

-X – перевірка файлу на можливість дій з ним. Це унарний оператор з одним аргументом - або ім'ям файлу, або вказівником файлу. Перевіряє одну з умов. Без аргументу береться значення змінної *$_*. Аргумент можна писати в круглих дужках. *'X'* має такі значення:

-r – файл дозволено читати ефективним *uid/gid* ; -w – в файл дозволено записувати ефективним *uid/gid*; -x - файл дозволено виконувати ефективним *uid/gid*; -o - файл належить ефективному *uid* (ідентифікатор користувача); -R - файл дозволено читати реальним *uid/gid*; -W –в файл дозволено записувати реальним *uid/gid*; -X - файл дозволено виконувати реальним *uid/gid*; -O – файл належить реальному *uid*; *-e* - файл існує; *-z* – файл пустий; *-s* - файл не пустий; *-f* – звичайний текст; *-d* - директорія; *-l* - символічне посилання; *-p* - pipes (конвеєр); *-S* - socket (сокет); *-b* - спеціальний блочний пристрій*; -c* –

спеціальний символьний пристрій; *-t* – вказівник на пристрій *tty*; *-u* - встановлено біт *setuid*; *-g* - встановлено біт *setgid*; *-k* - встановлено біт *sticky*; *-T* - текстовий файл; *-B* – двійковий файл; *-M* - "вік" файлу в днях на момент старту скрипта; *-A* - днів з останнього читання; *-C* - днів з останньої модифікації *inode*.

Приклад перевірки наявності файлу "file.txt" в поточній директорії і виведення при наявності кількості днів з моменту останнього звернення до нього:

```
if (-e("file.txt")){print (-A("file.txt"))} else {print "Файл не створено"}.
```

Синтаксис регулярних виразів для обробки тексту

Регулярний вираз – це зразок, що складається з символів. Він використовується для аналізу скриптом вхідних даних з урахуванням пробілів, ком, символів табуляції та інших розділювачів. В регулярних виразах Perl для скороченого запису використовує символи:

Символ	Опис
.	Відповідає будь-якому символу (крім символу нового рядка)
(..)	Групує послідовність елементів
+	Задовольняє попередньому зразку один або більшу кількість раз
-	Задовольняє зразку нуль або один раз
*****	Відповідає зразку один або нуль раз
[...]	Відповідає символу з заданої множини
[^...]	Відповідає символу з множини, отриманої запереченням
(...|...|...)	Відповідає одній з альтернатив
^	Відповідає початку рядка
$	Відповідає зразку в кінці рядка
{n,m}	Відповідає зразку від n до m раз
{n}	Відповідає зразку точно n раз
{n,}	Відповідає зразку мінімум n раз
\n\t etc.	Відповідає знаку нової лінії, символу табуляції і т. д.
\b	Відповідає на межі слова
\B	Відповідає всередині меж слова
\d	Відповідає цифрі
\D	Відповідає не цифрі
\s	Відповідає пробілу
\S	Відповідає не пробілу
\w	Відповідає букві або цифрі
\W	Відповідає символу, що не є ні буквою, ні цифрою

Perl регулярні вирази (зразки, шаблони) обмежуються з обох боків слешами (похилими рисками), наприклад, в виді /pattern/.

Наприклад:

```
  наступні регулярні вирази істинні, якщо:
/ig/          #    рядок містить 'ig'
/(b|d|f)ig/      # рядок містить 'big', 'dig' або 'fig'
/[0-9]+/        # рядок містить число
/[A-Za-z][A-Za-a0-9_]*/ # рядок містить ідентифікатор
```

Детальніше з роботою з файлами в Perl можна ознайомитися на сайтах: *http:\\www.PERL.org.ru* або *http:\\www.perl.com*

Приклад. Створити Perl-скрипт, який виводить на екран дату-час, інформацію про вільне місце на дисках та про використання оперативної пам’яті комп’ютера.

Рішення. Створюємо в *Kword* або в *StarOffice* наступний текст програми і зберігаємо його з розширенням *.txt* або *.pl*:

```
use Term::ANSIColor; system clear; print color(“magenta”);
system date;
ptint color(“white”), “\n”; system df; print color
(“yellow”), “\nOperative memory\n”; system free; print
color(“reset”),”\n”;
```

При збереженні програми даємо ім’я файлу: *fds.txt*. Запускається Perl-скрипт на виконання з командного рядка Linux так: *[stud@localhost Perl] $ perl –w fds.txt.*

Контрольні запитання

Які сервіси надає типова операційна система?

Для ЕОМ якого покоління були розроблені перші ОС?

Як класифікуються ОС за режимом обробки задач?

Що таке Perl-скрипт?

Які керуючі структури є в Perl?

Які команди роботи з файлами і каталогами в Perl використано в Perl-скрипті зробленої практичної роботи?

Які регулярні вирази Perl використано в Perl-скрипті зробленої практичної роботи?

Практична робота №3

Робота в пакеті StarOffice

Частина 1. Текстовий процесор

Мета: реалізувати в StarOffice формулу, таблицю вказаного типу, побудувати кругову діаграму.

Завдання.

режимі колонтитулів ввести в верхній частині листа прізвище, дату, групу, номер сторінки.

Вставити в документ формулу:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma}} \cdot \exp\left|\frac{-(x-m)^2}{2\pi\sigma}\right|,$$

де m – математичне сподівання, σ_2 -дисперсія.

Вставити таблицю з 3 колонок і 6 рядків. Назви колонок: *ПІБ, Зарплата, Кількість дітей.* Шостий рядок першого стовпчика – слово *Всього*. Заповнити цей рядок в StarOffice.

Створити кругову діаграму з інформацією з таблиці.

Контрольні запитання

Як увійти в текстовий процесор StarOffice?

Як вставити формулу в текстовий процесор StarOffice?

Як заповнювати шостий рядок в таблиці?

Як створити кругову діаграму?

Частина 2. Електронні таблиці

Мета: реалізувати в StarOffice таблицю вказаного типу, побудувати кругову діаграму.

Завдання.

Створити таблицю:

N п/п	ПІБ	Нараховано, грн	Податок
1.	Сич В.В.	16,00	
2.	Філь К.К.	64,00	
3.	Туз Д.І.	120,00	
4.	Дяк С.П.	683,00	
5.	Кудь О.О.	185,00	
6.	Грек А.А.	920,00	
7.	Рудь І.І.	396,00	
8.	Гук М.М.	1232,00	
9.	Жук Г.Г.	943,00	

Заповнити колонку *Податок*, якщо діють такі умови[1]:

від суми, меншої 17 грн. податок не нараховується;

від суми, що лежить в межах 17 грн.<сума<=85 грн., береться податок так: 10% від суми, що перевищує 17 грн.

від суми, яка лежить в межах 85 грн. < сума <=170 грн., береться податок так: 6,88 грн. + 15% від суми, що перевищує 85 грн.;

від суми, що лежить в межах 170 грн. < сума, береться податок так: 11,98 грн.+20% від суми, яка перевищує 170 грн.

Вказівка. Використати функцію ЕСЛИ( ).

Побудувати кругову діаграму з даними таблиці *ПІБ, Податок* з виносками значень. Роздрукувати результати.

Контрольні питання

Як увійти в StarOffice? Як створити таблицю в StarOffice?
Як увійти в режим використання вбудованих функцій в StarOffice?
Як працювати з вбудованими функціями StarOffice?
Як будувати діаграму в StarOffice?

---

[1] Данне завдання не відображає реальні сучасні нарахуваня податків і є прикладом складної умови для реалізації в електронних таблицях.

Практична робота № 4

Створення, відслідковування і знищення процесів в ОС Linux

Мета: дізнатися, як керувати процесами в Linux. Навчитися переводити процеси із фонового режиму в пріоритетний і навпаки, слідкувати за працюючими процесами, зупиняти процеси і запускати їх так, щоб вони продовжували працювати у вашу відсутність.

Завдання.

Запустити цифровий годинник xclock.

Перевести його в пріоритетний і фоновий режим роботи.

Запустити аналоговий годинник xclock.

Провести моніторинг процесів:

вивести інформацію про статус процесів;

вивести інформацію про статус фонових процесів.

Проаналізувати стан пам’яті комп’ютера.

Вивести інформацію:

про поточний час;

про час безперервної роботи системи;

про кількість підключення користувачів;

про середню завантаженість за останні 1,5 і 15 хвилин.

Вивести список процесів, що постійно оновлюється.

Теоретичні відомості

Команда ps. Назва: ***ps*** - видача інформації про стан процесів.

Синтаксис: *ps [-i] [- d] [- a] [- f] [- 1] [- n* файл з системою] [- t список терміналів] *[-p* список ідентифікаторів процесів]; *[- u* список ідентифікаторів користувачів] *[- g* список ідентифікаторів лідерів груп]

Опис. Команда *ps* видає інформацію про активні процеси. За замовчуванням інформація дається тільки про процеси, що асоціюються з цим терміналом. Виводяться ідентифікатор процесу, ідентифікатор терміналу, витрачений до цього моменту час ЦП і ім'я команди. Якщо потрібна інша інформація, слід користуватися опціями. Деякі опції мають один аргумент або список аргументів. Аргументи в списку можуть бути або відокремлені один від

одного комами, або всі разом поміщені в подвійні лапки і відокремлені пропусками або комами.

Аргументи в списку процесів і в списку груп мають бути числами.

Командою ps обробляються наступні опції: *-i* - вивести інформацію про усі процеси; - *d* - вивести інформацію про усі процеси, окрім лідерів груп; *-a* - вивести інформацію про всі процеси, що найчастіше запрошуються, тобто про усі процеси, окрім лідерів груп і процесів, що не асоціюються з терміналом; - *f* - генерувати повний лістинг (див. нижче роз'яснення сенсу колонок); - *1* - генерувати лістинг в довгому форматі (див. нижче); - *n* – файл з системою (вважати, що операційна система завантажена з файлу з системою, а не з файлу /unix); - *t* - список терміналів; видавати інформацію тільки про процеси, що асоціюються з терміналами із заданого списку терміналів.

Термінал - це або ім'я файлу-пристрою, наприклад номер або console, або просто номер, якщо ім'я файлу починається з tty.

-p - список ідентифікаторів процесів: видавати інформацію тільки про вказані процеси; - *u* - список ідентифікаторів користувачів: видавати інформацію тільки про процеси із заданими ідентифікаторами або вхідними іменами користувачів. Ідентифікатор користувача виводиться в числовому виді, за наявності опції - *f* - в символьному.

g список ідентифікаторів лідерів груп: виводити інформацію тільки про процеси, для яких вказані ідентифікатори лідерів груп. *Лідер групи* - це процес, номер якого ідентичний його ідентифікатору групи. *Shell,* що запускається при вході в систему, є стандартним прикладом лідера групи.

При вказівці опції - *f ps* намагається визначити ім'я команди і аргументи, з якими був створений процес, досліджуючи призначений для користувача блок процесу. Якщо це не вдається, ім'я процесу виводиться так само, як і за відсутності опції - *f*, тільки полягає в квадратних дужках.

Нижче наводяться заголовки колонок видачі і пояснюється значення їх вмісту. Букви *l* або *f* в дужках означають, що ця колонка з'являється відповідно при довгому або повному форматі видачі; відсутність букв означає, що ця колонка виводиться завжди. Відмітимо, що опції - *l* і - *f* впливають тільки на формат видачі, але не на список процесів, інформація про які буде надана.

F(1) - Прапори(шістнадцятирічні), логічна сума яких дає наступні відомості про процес : *00* - процес термінує; елемент таблиці процесів вільний.

01 - системний процес: завжди в основній пам'яті.

02 - процес трасується батьківським процесом.

04 - батьківський сигнал трасування зупинив процес; батьківський процес чекає.

08 - процес не може бути розбуджений сигналом.

- процес в основній пам'яті.

- процес в основній пам'яті; блокований до завершення події.

- поступає сигнал до видаленої системи.

- процес в черзі на введення/виведення.

- *S(l)* - статус процесу : *0* – активний, обробляється процесором; *S* – сплячий, чекає завершення події; *R* – готовий, стоїть в черзі на виконання; *I* - що народжується, процес створюється; *Z* - стан "зомбі", процес завершений, але батьківський процес не чекає цього; *T* – трасований, процес зупинений сигналом, оскільки батьківський процес трасує його; *X* - процес що росте, чекає отримання більшого об'єму основної пам'яті.

UID(f, l) - Ідентифікатор власника процесу; при вказівці опції - *f* видається вхідне ім'я користувача.

PID - Ідентифікатор процесу(потрібний для того, що термінує процесу).

PPID(f, l) - Ідентифікатор батьківського процесу.

C(f, l) - Доля виділеного планувальником часу ЦП.

STIME(f) - Час запуску процесу(години:хвилини:секунди). Якщо процес запущений більш ніж 24 години тому, видається місяць і день запуску.

PRI(l) - Пріоритет процесу; більше число означає менший пріоритет.

NI(l) - Поправка до пріоритету.

ADDR(l) - Адреса процесу в пам'яті.

SZ(l) – Розмір (у блоках по 512 байт) образу процесу в пам'яті.

WCHAN(1) - Адреса події, на яку чекає процес. У активного процесу ця колонка порожня.

TTY - термінал(зазвичай - термінал, з якого був запущений процес; див. нижче СЮРПРИЗИ), що управляє. Якщо такого немає, видається символ «?».

TIME - Витрачений процесом час ЦП.

COMMAND - Ім'я програми; якщо вказана опція - *f,* то виводиться повне ім'я команди і її аргументи.

Процес, що закінчив виконання своєї програми, і батьківський процес, що не дочекався ще завершення, в якості імені програми отримує <defimct>.

Файли. */dev/kmem* - Віртуальна пам'ять ядра.

/dev/mem - Пам'ять.

/etc/passwd - Файл паролів, з якого береться інформація про ідентифікатори користувачів.

/etc/psdata - Внутрішня структура даних. */unix* - Ім'я файлу, звідки була завантажена операційна система. **Сюрпризи.** Під час виконання команди *ps* стан речей може змінитися;

таким чином, *ps* дає лише наближення до дійсності. Деякі відомості, що відносяться до процесів, відмічених як *<defunct>,* не відповідають дійсності.

Якщо не вказана жодна з опцій - *t, -p, - u, - g*, команда *ps* перевіряє стандартне введення, стандартне виведення і стандартний протокол (у вказаному порядку) для визначення терміналу, що управляє і намагається вивести інформацію про процеси, які асоціюються з цим терміналом. За ситуації, коли усі три стандартні напрями введення/виведення перепризначувалися, *ps* не може знайти і видати ім'я терміналу, що управляє.

Системний виклик fork. Назва: *fork* - створення нового процесу. Синтаксис: *int fork( ).*

Опис. Виклик *fork* призводить до створення нового процесу (породженого процесу) - точної копії процесу, що зробив виклик (батьківського процесу). Точніше, породжений процес наслідує у батьківського процесу наступні характеристики:

Оточення.

Прапор "закрити при виконанні виклику exec".

Способи обробки сигналів (тобто *SIG DFL, SIG IGN, SIGHOLD*, адреси функцій обробки сигналів).

Дозвіл встановлювати заново діючий ідентифікатор користувача.

Дозвіл встановлювати заново діючий ідентифікатор групи.

Стан профілізації(включено/вимкнено).

Значення поправки до пріоритету.

Усі приєднані сегменти пам'яті, що розділяються.

Ідентифікатор групи процесів.

Ідентифікатор групи терміналу.

Поточний робочий каталог.

Кореневий каталог.

Маска режиму створення файлів.

Обмеження на розмір файлу.

Породжений процес відрізняється від батьківського процесу наступним.

Породжений процес має свій унікальний ідентифікатор процесу.

Породжений процес має інший ідентифікатор батьківського процесу, рівний ідентифікатору процесу, його породжувача.

Породжений процес має свої власні копії батьківських дескрипторів файлів. Кожен дескриптор файлу породженого процесу розділяє з відповідним батьківським дескриптором файлу загальний покажчик поточної позиції у файлі.

Усі semadj значення скидаються [див. semop(2)].

Породжений процес не наслідує у батьківського процесу ознак утримання в пам'яті сегменту команд, даних або усього процесу цілком [див. plock(2)].

Обнуляються лічильники часу, витраченого для обслуговування цього процесу (tms utime, tms stime, tms cutime, tms cstime). Відміняється запит до будильника.

Хід роботи

Управління пріоритетними і фоновими завданнями

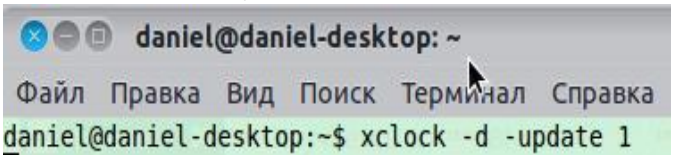
Коли ви запускаєте команду у вікні терміналу, вона виконується в пріоритетному режимі. Багато команд виконуються швидко, але припустимо, що ви працюєте в графічному режимі і хочете бачити на робочому столі цифровий годинник.

Використаємо утиліту xclock. Для того щоб запустити цифровий годинник на вашому робочому столі потрібно виконати в терміналі наступну команду: *$ xclock -d -update 1*

Опція *-update 1* задає оновлення годинника кожної секунди; в іншому випадку оновлення буде виконуватися кожної хвилини. Ви можете побачити годинник, зображений на рисунку 1, а вміст вашого вікна термінала має бути схожим на вміст **лістингу 1**.

Примітка. На момент написання цієї роботи у роботі xclock була присутня програмна помилка, яка виявлялася при ввімкнутих ефектах робочого столу. Вона виявлялася, перш за все, у тому, що панель заголовку не змінювалася навіть при перемиканні на неї. Якщо у вас виникає така помилка, то можна тимчасово вимкнути ефекти робочого столу.

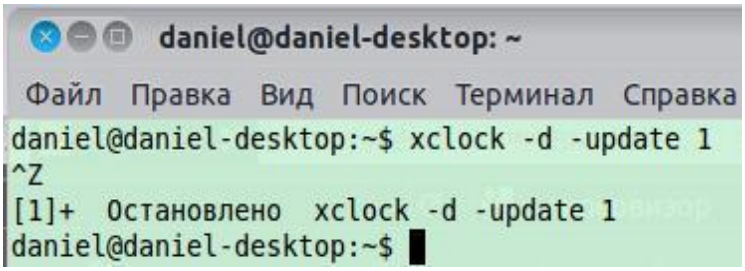
Лістинг 1. Запуск xclock



Тепер у вашому вікні термінала не відображається запрошення, тому необхідно повернути управління.

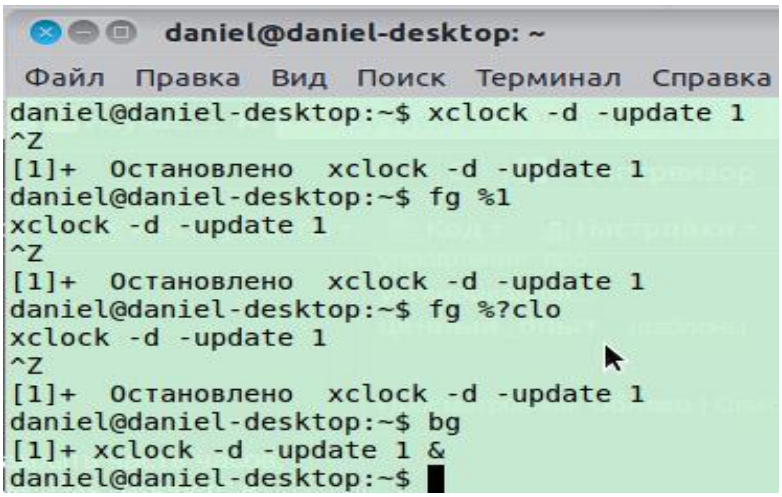
командному інтерпретаторі bash є комбінація клавіш для призупинення Ctrl-z. Натиснувши цю комбінацію клавіш, ви знову побачите запрошення термінала, як показано в ***лістингу 2.***

Лістинг 2. Призупинка xclock за допомогою Ctrl-z.



Годинник все ще знаходиться на вашому робочому столі, але із-за призупинення він не йде. Зверніть увагу на повідомлення “[1]+Остановлено” вікні терміналу. Цифра 1 в цьому повідомлення означає номер завдання. Ви можете перезапустити годинник, набравши команду *fg %1*. Тут ви також можете використати ім'я команди або його частину, набравши, наприклад, *fg %xclock* або *fg %?clo*. Нарешті якщо ви запустите команду *fg* без яких-небудь параметрів, то буде перезапущено останнє зупинене завдання, в нашому випадку це завдання 1. Перезапуск завдання за допомогою *fg* поновлює його роботу в пріоритетному режимі, і ви знову не побачите запрошення. В цій ситуації нам потрібно перевести завдання у фоновий режим. Саме для цього призначена команда *bg*, параметри якої аналогічні параметрам команди *fg*.

лістингу 3 показано, як поновити роботу програми xclock в пріоритетному режимі, призупинити її, і знову запустити, використовуючи дві форми команди fg. Після призупинки ви можете поновити роботу xclock в фоновому режимі — годинник продовжить працювати, а ви зможете продовжити роботу у вікні термінала.



Лістинг 3. Переведення xclock в пріоритетний або фоновий режим роботи.

Ви могли помітити, що при переведенні xclock в фоновий режим в кінці команди використовується &, а повідомлення “Остановлено”

зникає. На справді не обов'язково призупиняти процес для його переведення в фоновий режим; для цього потрібно просто додати & в кінець команди, і інтерпретатор запустить її (або список команд) в фоновому режимі.

Використаємо це для запуску аналогічного годинника. Ви маєте побачити годинник, зображений на рисунку 1, а вміст вашого вікна термінала має бути схожим на вміст лістингу 4.

Рисунок 1 – Аналоговий годинник xclock

Лістинг 4. Запуск аналогового годинника *xclock* в фоновому режимі за допомогою &.

```
daniel@daniel-desktop: ~
Файл  Правка  Вид  Поиск  Терминал  Справка
daniel@daniel-desktop:~$ xclock -bg wheat -update 1&
[2] 2625
daniel@daniel-desktop:~$
```

Цього разу повідомлення дещо відрізняється. Воно містить номер завдання і ідентифікатор процесу *process id (PID).* Давайте запустимо команду *jobs* і подивимось які завдання виконуються в нашій системі. Додайте опцію *-l* для відображення ідентифікаторів *PID*, і ви побачите, що завдання 2 має ідентифікатор *PID 2625*, як показано на лістингу 5. Також зверніть увагу на те, що завдання 2 має знак +, який означає, що це завдання є поточним. Якщо ніяке інше завдання не призначене на виконання командою *fg*, то поточне завдання буде виконуватися в пріоритетному режимі.

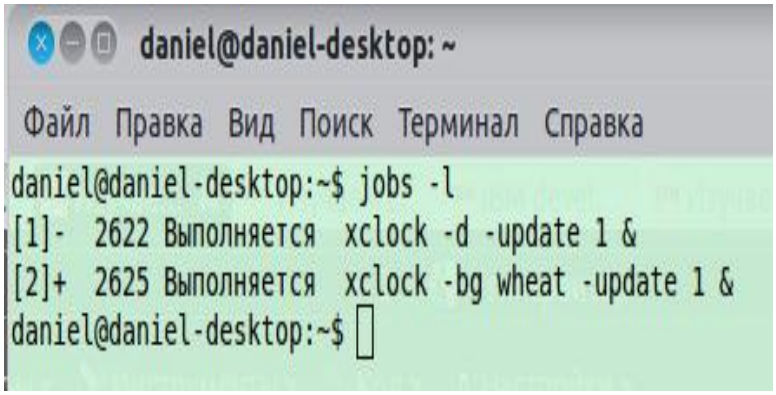
Лістинг 5. Вивід інформації про запущені завдання і процеси, що виконуються.

```
daniel@daniel-desktop: ~
Файл  Правка  Вид  Поиск  Терминал  Справка
daniel@daniel-desktop:~$ jobs -l
[1]-  2622 Выполняется  xclock -d -update 1 &
[2]+  2625 Выполняется  xclock -bg wheat -update 1 &
daniel@daniel-desktop:~$
```

Далі створимо у вікні терміналу простий цифровий годинник. Ми будемо використовувати команду *sleep*

для створення двохсекундної паузи і команду *date* для відображення поточної дати і часу. Ми розташуємо ці дві команди в циклі *while* з блоком *do/done* для створення нескінченого циклу. Цю конструкцію розташуємо в дужках для створення списку команд і запустимо цей список на виконання в фоновому режимі, використовуючи *&*.

Лістинг 6. Простий цифровий годинник.



Наш список команд запускається у вигляді завдання з номером 2 і ідентифікатором *PID 2969*. Кожні дві секунди запускається команда *date*, і на екран виводиться поточна дата і час. Данні, що вводяться нами в лістингах, виділені жирним шрифтом. Якщо ви друкуєте повільно, то перш ніж ви закінчите набирати команду, між символами, що ви вводите можуть з'явитися рядки виводу. Дійсно, ви можете помітити, що символи *'f'* і *'g'*, які ми надрукували для переведення списку в пріоритетний режим, розділені декількома рядками. Коли нарешті команда *fg* надрукована, інтерпретатор *bash* показує поточну команду, що працює в ньому, а саме наш список команд, який продовжує виводити час кожні дві секунди.

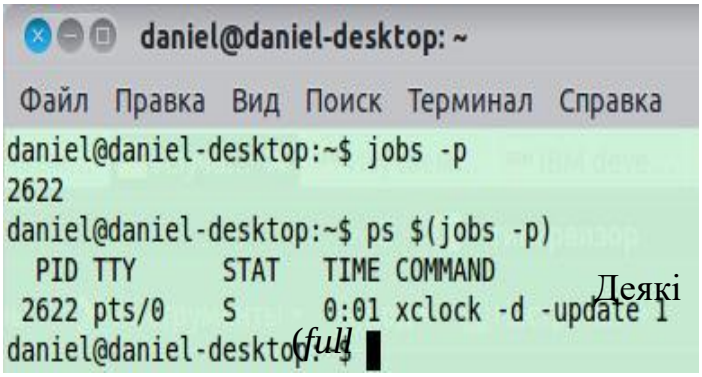
Як тільки ви перевели завдання в пріоритетний режим, ви можете завершити його або виконати над ним яку-небудь іншу дію. У нашому випадку давайте натиснемо *Ctrl-C* і зупинимо наш годинник.

Моніторинг процесів

Команда ps

Існує і інша команда, що відображає різну інформацію про статус процесу — команда *ps (“process status” - статус процесу*). Команда *ps* приймає якості аргументів нуль або більше ідентифікаторів *PID* і відображає статус відповідних процесів. Якщо запустити команду *jobs* з опцією *-p*, то її вивід буде представляти собою ідентифікатор лідера групи процесів кожного завдання. Ми будемо використовувати цей вивід в якості аргументів команди *ps*, як показано лістингу 7.

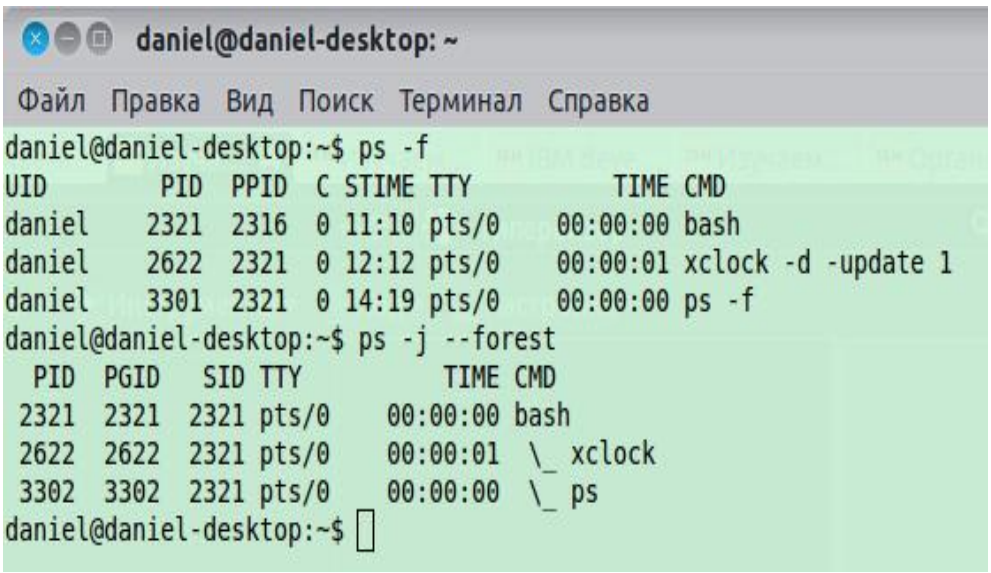
Лістинг 7. Статус фонових процесів.



Якщо ви запустите команду *ps* без яких-небудь опцій, то ви побачите список процесів, для яких ваш термінал є керуючим терміналом.

Деякі опції, включаючи *-f* (*full* — повний), *-j* (*jobs* — завдання) і *-l* (*long* — довгий) дозволяють керувати кількістю інформації, що виводиться. Якщо ви не вказуєте ніякі ідентифікатори *PID*, то іншою корисною опцією є опція *-forest*, що відображає команди у вигляді дерева і показує, які процеси є батьківськими для інших процесів. Зокрема можна побачити, що команди *sleep* з попередніх лістингів є дочірніми процесами сценаріїв, запущених вами в фоновому режимі. Приклади використання деяких опцій наведені в лістингу 8.

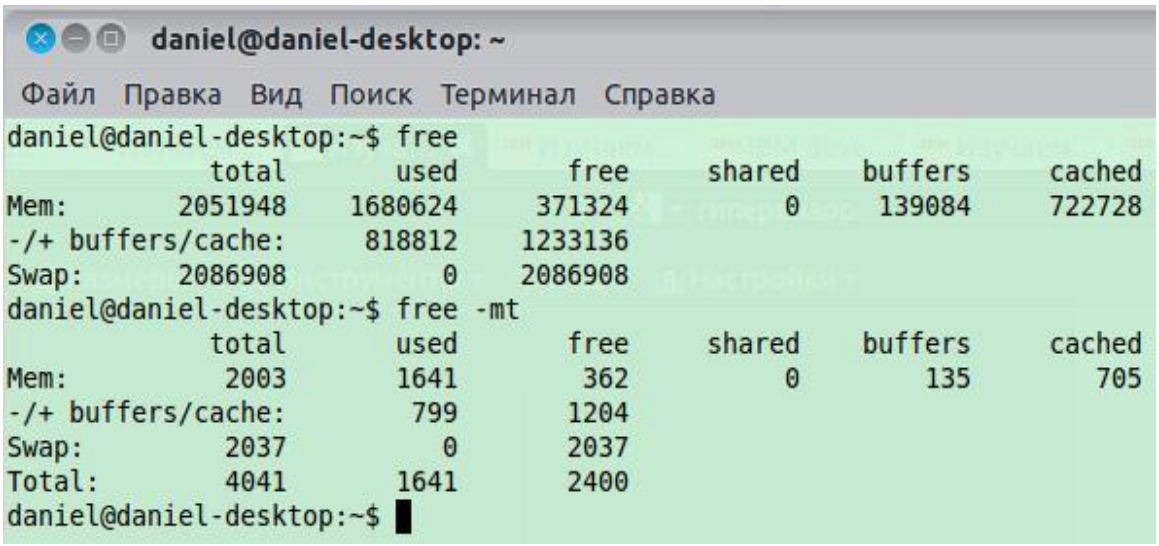
Лістинг 8. Більш детальна інформація про статуси процесів.



Команда free.

Команда *free* відображає кількість вільної пам'яті і пам'яті, що використовується комп'ютером. По замовчуванню значення виводяться в кілобайтах, але можна змінити це за допомогою різних опцій: *-b* (байт) *-k* (кб) *-m* (мб) *-g* (гб). Опція *-t* виводить сумарний рядок, а опція *-s* оновлює інформацію з вказаною періодичністю (час вказується в секундах). В лістингу 9 наведені два приклади.

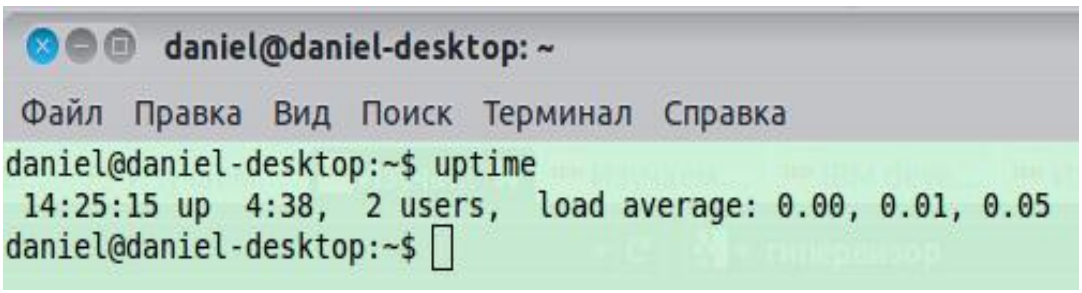
Лістинг 9. Використання команди free.



Команда uptime.

Команда uptime виводить рядок, де вказано поточний час, час безперервної роботи системи, кількість підключених користувачів і середня завантаженість за останні 1, 5 і 15 хвилин. Приклад команди показаний в лістингу 10.

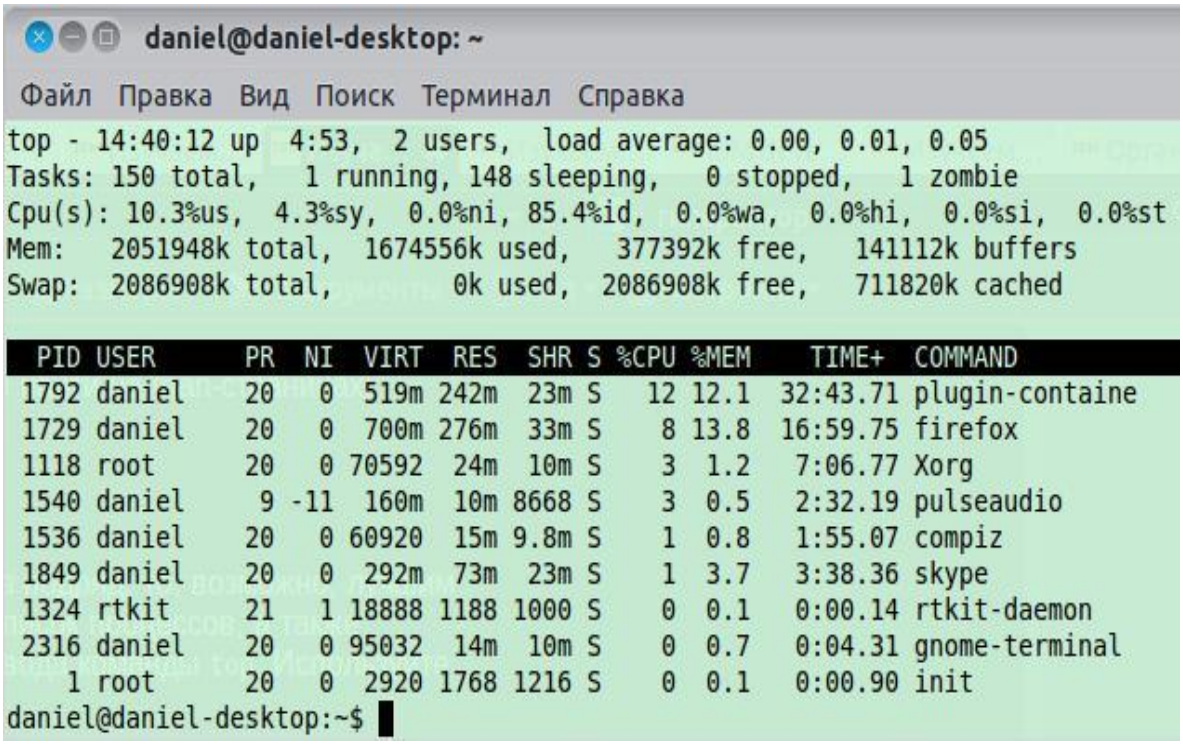
Лістинг 10. Інформація, що виводиться командою uptime.



Використання команди top.

Якщо ви хочете відслідковувати зміни і для цього запускаєте команду ps декілька раз, то, можливо, кращім рішенням буде команда *top*. Ця команда відображає список процесів, що постійно оновлюється, а також корисну сумарну інформацію. В лістингу 11 показані перші декілька рядків виводу команди *top*. Використовуйте підкоманду *q* для закриття *top*.

Лістинг 11. Вивід інформації про процеси за допомогою *top*.



Команда *top* має декілька підкоманд, з яких найбільш корисними є:

h — вивід довідкової інформації; *q* — завершення роботи команди *top*; *f* — додавання і видалення полів, що відображаються; *o* — сортування порядку виводу інформації; *F* — вибір полів, по яким виконується сортування.

Контрольні запитання

Що таке процеси? Які основні причини створення нових процесів у обчислювальних системах?

Що називається образом процесів? Назвіть атрибути процесів?

Які дії виконує ОС при створенні процесів?

Які дії виконує ОС при перемиканні процесів?

Які є види планування процесів?

Назвіть основні критерії короткострокового планування.

Які Ви знаєте стратегії короткострокового планування процесів?

Практична робота № 5

Потоки, програмні канали перенаправлення в ОС Linux

Мета: Дізнатися що таке потоки, як їх перенаправляти і розділяти, як перетворити потік в аргументи певної команди.

Завдання.

Створити директорію і файли для роботи з ними.
Перенаправити в файл стандартний вивід.
Перенаправити стандартний потік помилок.
Перенаправити два потоки в один файл.
Перенаправити вивід з використанням різних концепцій.
Розділити потоки.

Хід роботи

Для виконання прикладів в цій роботі ми будемо використовувати деякі файли. Створимо необхідну директорію і файли.

Для початку перейдіть в домашню директорію.

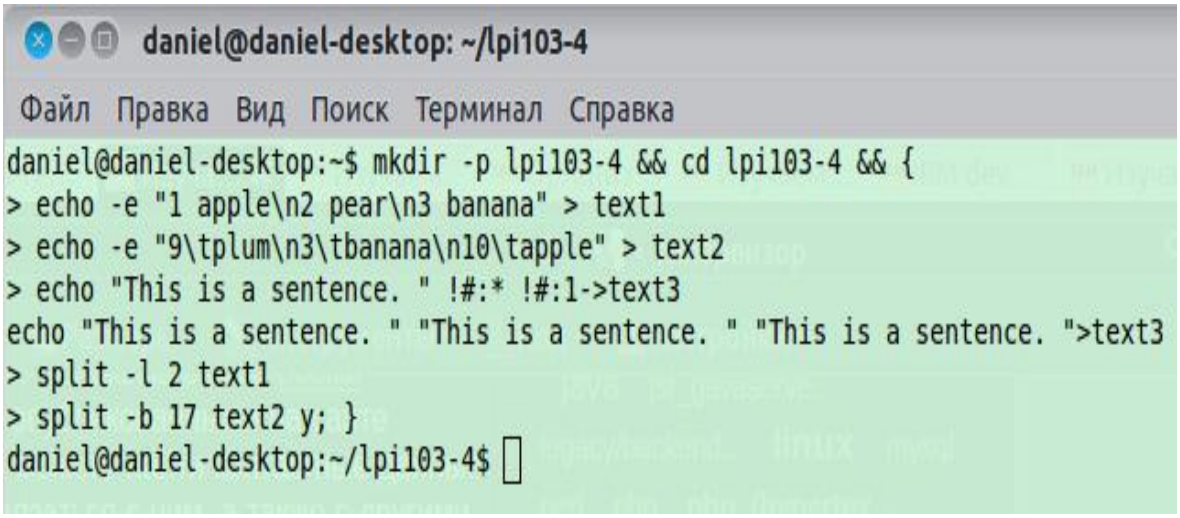
Відкрийте термінал і скопіюйте в нього вміст лістингу 1 (по рядкам, натискаючи *Ente*r після кожного рядка).

Лістинг 1. Створення файлів, необхідних для прикладів цієї роботи.

```
mkdir -p lpi103-4 && cd lpi103-4 && { echo
-e "1 apple\n2 pear\n3 banana" > text1
echo -e "9\tplum\n3\tbanana\n10\tapple" >
text2 echo "This is a sentence. " !#:* !#:1->text3
split -l 2 text1
split -b 17 text2 y; }
```

Ваше вікно має виглядати так, як показано в лістингу 2, а поточною директорією повинна стати щойно створена директорія *lpi103-4*.

Лістинг 2. Результати створення необхідних файлів.



Перенаправлення стандартного вводу/виводу.

Командний інтерпретатор Linux, такий як *bash,* отримує вхідні данні і направляє вихідні данні у вигляді послідовностей або потоків символів. Будь-який символ не залежить ні від попередніх, ні від наступних символів. Символи не впорядковані у вигляді структурованих записів або блоків з фіксованим розміром. Доступ до потоків здійснюється за допомогою механізмів вводу/виводу незалежно від того, звідки надходять і куди передаються потоки символів (файл, клавіатура, вікно, екран або інший пристрій вводу/виводу). Командні інтерпретатори Linux використовують три стандартні потоки вводу/виводу, кожному з яких назначений певний файловий дескриптор.

stdout – стандартний потік виводу, відображає вивід команд і має дескриптор;

stderr – стандартний потік помилок, відображає помилки команд і має дескриптор 2;

stdin – стандартний потік вводу, передає вхідні данні командам і має дескриптор 0.

Потоки вводу забезпечують передачу вхідних даних (зазвичай тих, що надходять з клавіатури) командам. Потоки виводу забезпечують друк текстових символів, як правило, на термінал. Спершу термінал являв собою ASCII друкуючий пристрій або дисплейний термінал, але зараз, як правило, це просто вікно робочого столу комп'ютера.

Перенаправлення виводу.

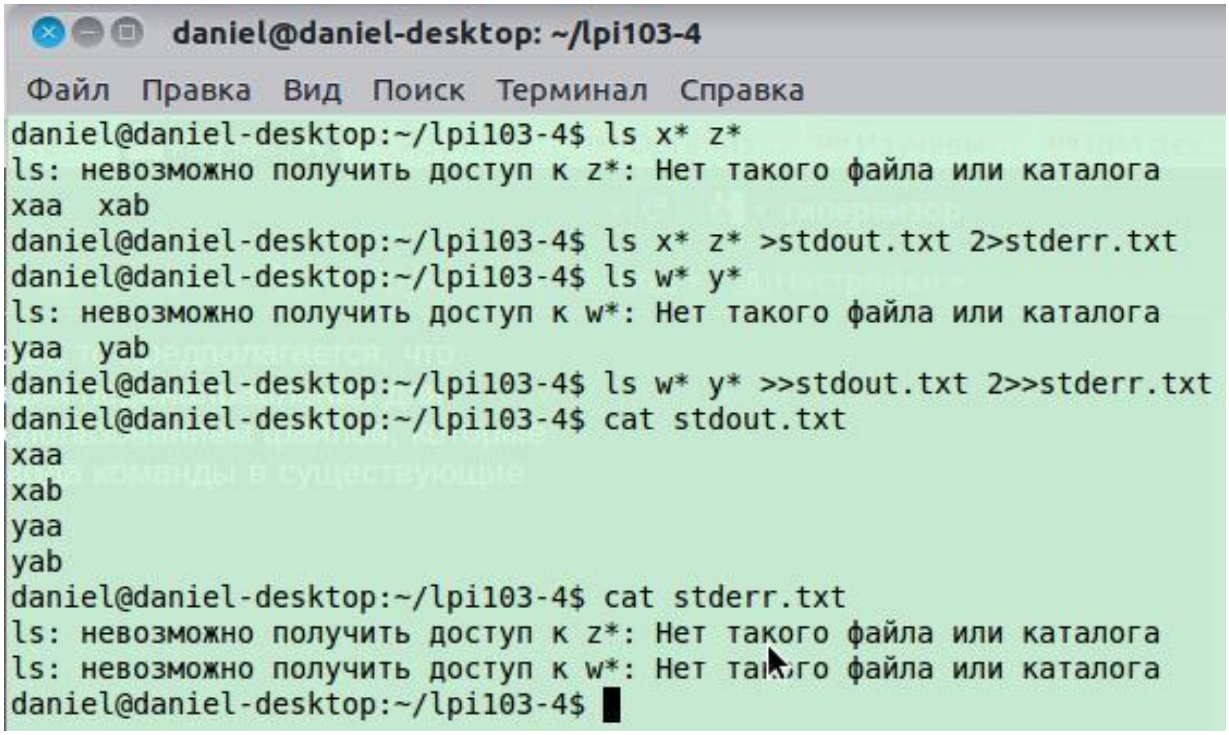
Існує два способи перенаправлення виводів в файл:

n> – перенаправляє вивід із файлового дескриптора n в файл. Ви повинні мати права на запис в файл. Якщо файл не існує, то він буде створений. Якщо файл існує, то весь його вміст, як правило, знищується без яких-небудь попереджень.

n>> – також перенаправляє вивід із файлового дескриптора *n* в файл. Ви також повинні мати права на запис в файл. Якщо файлу не існує, то він буде створений. Якщо файл існує, то вивід додається до його вмісту.

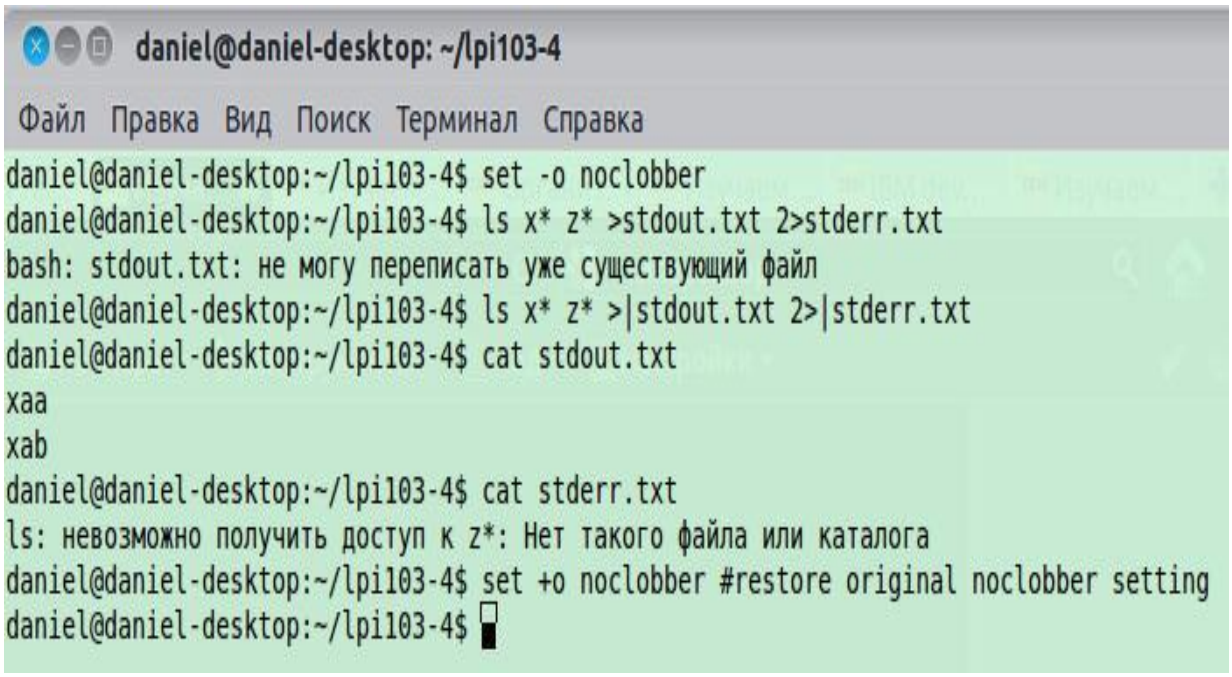
Символ *n* в операторах *n* > або *n* >> є файловим дескриптором. Якщо він не вказаний, то припускається, що використовується стандартний пристрій виводу. В лістингу 3 продемонстрована операція перенаправлення для розділення стандартного потоку виводу і стандартного потоку помилок команди *ls* з використанням файлів, які були створені раніше в директорії *lpi103-4*. Також продемонстровано додавання виводу команди в існуючі файли.

Лістинг 3. Перенаправлення виводу.



Перенаправлення виводу за допомогою оператора *n*> зазвичай призводить до перезапису існуючих файлів. Ви можете керувати цією властивістю за допомогою опції *noclobber* вбудованої команди *set*. Якщо ця опція визначена, то ви можете перевизначити її за допомогою оператора n>|, як показано в лістингу 4.

Лістинг 4. Перенаправлення виводів за допомогою опції noclobber.

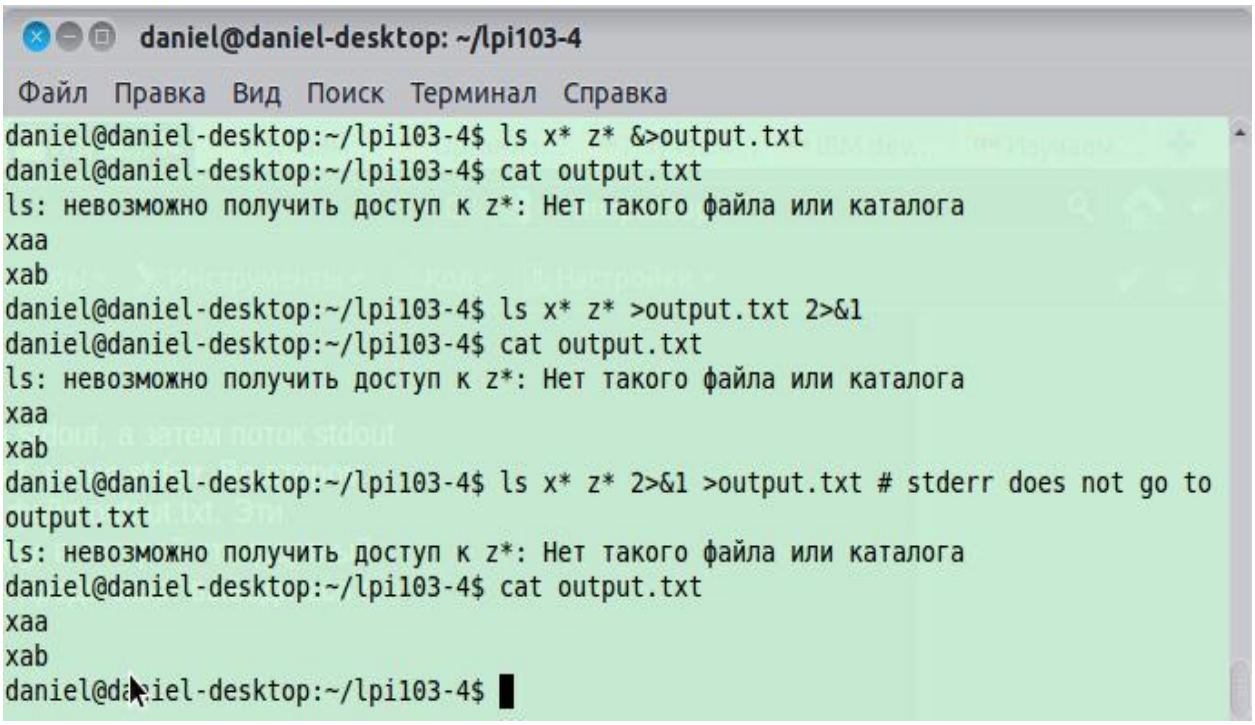


Іноді може знадобитися перенаправити в файл як стандартний вивід, так і стандартний потік помилок. Часто це використовується в автоматизованих процесах або фонових завданнях для того, щоб потім можна було подивитися результати роботи. Щоб перенаправити стандартний вивід і стандартний потік помилок в одне і те ж місце, використовуйте оператор *&>* або *&>>*. Альтернативний варіант – перенаправити файловий дескриптор *n* і потім файловий дескриптор *m* в одне і те ж місце за допомогою конструкції *m>&n* або *m>>&n*. В цьому випадку важливий порядок перенаправлення потоків. Наприклад, команда *command 2>&1 >output.txt* це не те ж саме, що команда *command >output.txt 2>&1*.

першому випадку потік помилок *stderr* перенаправляється в поточне місце розташування потоку *stdout*, а потім потік *stdout* перенаправляється в файл output.txt; однак друге перенаправлення зачіпає лише *stdout*, але не *stderr*.

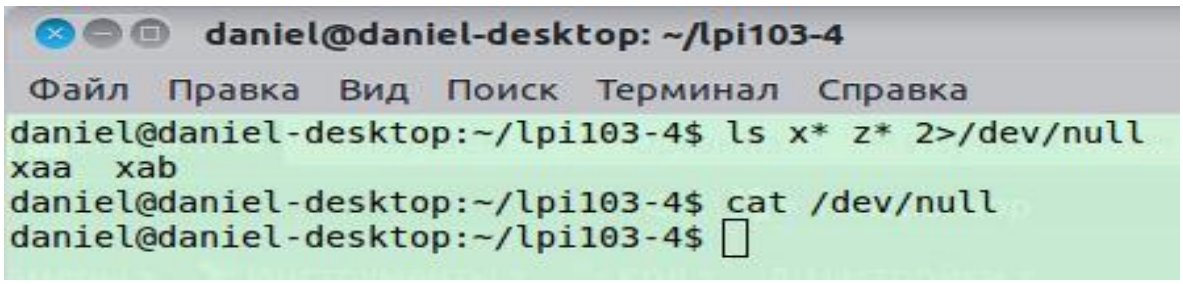
другому випадку потік *stderr* перенаправляється в поточне місце розташування потоку *stdout*, тобто, в файл output.txt. Ці перенаправлення проілюстровані в лістингу 5. Зверніть увагу на останню команду, в якій стандартний вивід був перенаправлений після стандартного потоку помилок, і, як наслідок, потік помилок продовжує виводитися у вікно терміналу.

Лістинг 5. Перенаправлення двох потоків в один файл.



інших випадках вам може знадобитися повністю проігнорувати стандартний вивід або стандартний потік помилок. Для цього слід перенаправити відповідний потік в пустий файл */dev/null*. В лістингу 6 показано, як проігнорувати потік помилок команди ls і як за допомогою команди cat переконатися в тому, що файл */dev/null* насправді пустий.

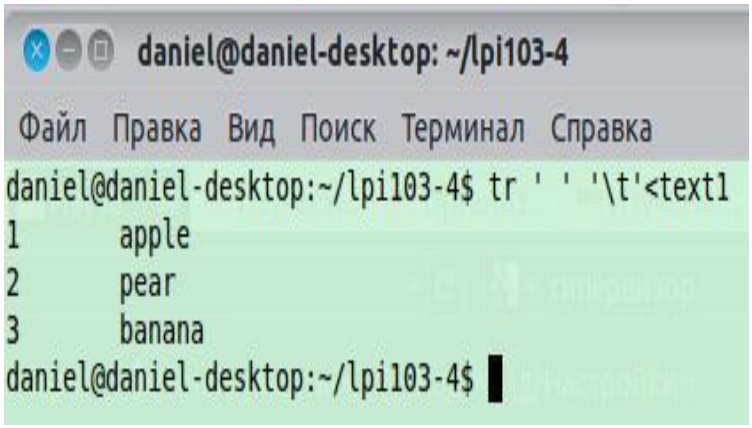
Лістинг 6. Ігнорування стандартного потоку помилок шляхом використання */dev/null.*



Перенаправлення вводу.

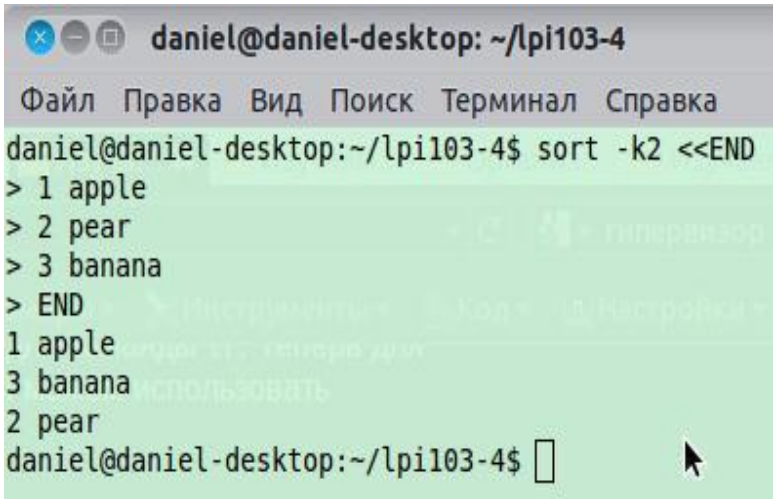
Таким же чином, як ми можемо перенаправляти потоки *stdout* і *stderr*, ми можемо перенаправити потік *stdin* із файлу за допомогою оператора <. Команда *tr* використовується для заміни пробілів в файлі *text1* на символи табуляції. Вивід команди *cat* можна використати для того, щоб створити стандартний потік вводу для команди *tr*. Але для перетворення пробілів в символи табуляції замість даремного виклику команди *cat* ми можемо використати перенаправлення вводу, як показано в лістингу 7.

Лістинг 7. Перенаправлення вводу.



В командних інтерпретаторах, в тому ж числі і в *bash*, реалізована концепція *here-document*, яка є одним із способів перенаправлення вводу. В ній використовується конструкція << і будь-яке слово, наприклад *END*, яке є маркером, або сигнальною міткою, що означає кінець вводу. Ця концепція продемонстрована в лістингу 8.

Лістинг 8. Перенаправлення вводу з використанням концепції here-document.



Але чому просто не можна набрати команду *sort -k2*, ввести данні і натиснути комбінацію *Ctrl-d*, яка позначає кінець вводу? Зрозуміло, що ви могли б виконати цю команду, але тоді ви не дізналися б про концепцію *here-document*, яка дуже часто використовується сценаріях командною оболонкою (в яких не існує іншої можливості вказати, які саме рядки повинні сприйматися в якості вводу). Оскільки для вирівнювання тексту і забезпечення зручності читання в сценаріях використовуються символи табуляції, то існує інший прийом використання концепції *here-document*. При використанні оператора << – замість оператора << початкові символи табуляції видаляються.

лістингу 9 ми використовували підстановку команд для створення символу табуляції, а потім створили невеликий сценарій командної оболонки, що містить дві команди *cat*, кожна з яких зчитує данні з блоку *here-document*. Зауважте, що ми використовували слово *END* в якості сигнальної мітки блоку *here-document*, який ми зчитуємо з терміналу. Якби ми використовували це ж саме слово в нашому сценарії, то наш ввід закінчився би передчасно. Тому замість слова *END* ми використовуємо в сценарії слово EOF. Після того, як наш

сценарій створений, ми використовуємо команду **.** (точка) щоб запустити його в контексті поточного командного інтерпретатора.

Лістинг 9. Перенаправлення вводу з використанням концепції here-document.

Розділення виводу.

Іноді може виникнути необхідність продивлятися вивід на екрані і одночасно зберігати його в файл. Для цього ви могли б перенаправити вивід команди в файл в одному вікні, а потім за допомогою *tail -fn1* відслідковувати виводи в іншому вікні, однак простіше за все використовувати команду *tee*.

Команда *tee* використовується в конвеєрі, а її аргументом є ім'я файлу (або імена декількох файлів), в який буде передаватися стандартний вивід. Опція *-a* дозволяє не заміщати старий вміст файлу новим вмістом, а додавати данні в кінець файлу. Якщо ви хочете зберегти як стандартний вивід, так і потік помилок, то необхідно перенаправити потік *stderr* в потік *stdout* перш, ніж передавати данні на вхід команді *tee*. В лістингу 10 наведений приклад використання команди *tee* для збереження виводу в два файли, *f1 і f2*.

Лістинг 10. Розділення потоку *stdout* за допомогою команди *tee*.

```
daniel@daniel-desktop:~/lpi103-4$ ls text[1-3]|tee f1 f2
text1
text2
text3
daniel@daniel-desktop:~/lpi103-4$ cat f1
text1
text2
text3
daniel@daniel-desktop:~/lpi103-4$ cat f2
text1
text2
text3
daniel@daniel-desktop:~/lpi103-4$
```

Контрольні запитання

Дайте визначення потокам. Що таке багатопотоковість?

Які основні операції з потоками?

Назвіть і опишіть способи реалізації потоків?

Які елементи сумісно використовуються потоками?

Які елементи окремо використовуються потоками?

Практична робота № 6

Робота з сигналами в ОС Linux

Мета: дізнатися про способи посилання, приймання і зміни реакції на сигнали в ОС Linux.

Завдання.

Відправляти сигнали за допомогою сигналу *kill*.

Використати сигнали для зупинки і перезапуску фонових завдань.

Використати сигнали для безумовного завершення процесів.

Хід роботи

Відправка сигналів процесам.

Розглянемо сигнали Linux, які є асинхронним засобом взаємодії з процесами. Головним способом відправки сигналів є команда kill.

Відправка сигналів, використовуючи kill

Команда kill посилає сигнал певному завданню або процесу.

лістингу 1 наведений приклад використання сигналів SIGTSTP і SIGCONT для зупинки і перезапуску фонового завдання. Використання сигналу SIGTSTP еквівалентно використанню команди fg, яка переводить завдання в пріоритетний режим, і подальшому використанню Ctrl-z для його призупинки. Використання сигналу SIGCONT еквівалентно використанню команди bg.

Лістинг 1. Зупинка і перезапуск фонових завдань.

```
daniel@daniel-desktop: ~
Файл  Правка  Вид  Поиск  Терминал  Справка
daniel@daniel-desktop:~$ xclock -d -update 1
^Z
[2]+  Остановлено  xclock -d -update 1
daniel@daniel-desktop:~$ bg
[2]+ xclock -d -update 1 &
daniel@daniel-desktop:~$ jobs -l
[1]-  2622 Выполняется  xclock -d -update 1 &
[2]+  3524 Выполняется  xclock -d -update 1 &
daniel@daniel-desktop:~$ kill -s SIGTSTP %2

[2]+  Остановлено  xclock -d -update 1
daniel@daniel-desktop:~$ jobs -l
[1]-  2622 Выполняется  xclock -d -update 1 &
[2]+  3524 Остановлено  xclock -d -update 1
daniel@daniel-desktop:~$ kill -s SIGCONT 3524
daniel@daniel-desktop:~$ jobs -l
[1]-  2622 Выполняется  xclock -d -update 1 &
[2]+  3524 Выполняется  xclock -d -update 1 &
daniel@daniel-desktop:~$
```

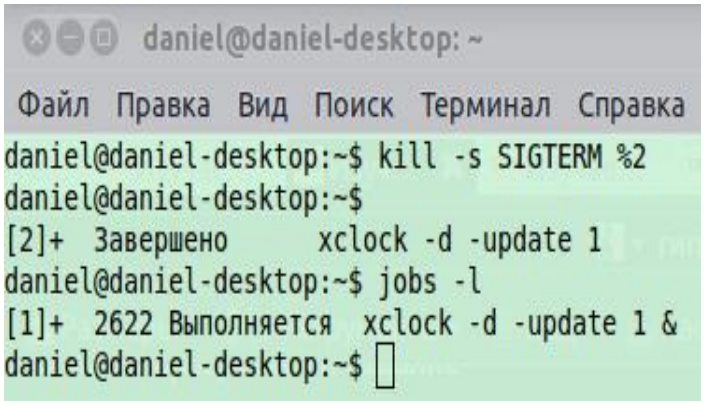
Обробники сигналів і завершення процесів

Ви вже бачили, що натискання Ctrl-c завершує процес. Фактично ця комбінація клавіш посилає процесу сигнал SIGINT (або interrupt — перервати). Якщо ви використовуєте команду kill і не вказуєте ім'я сигналу, то процес посилається сигнал SIGTERM. Для більшості задач ці два сигнали є еквівалентними.

загальному випадку в процесі може бути реалізований обробник сигналів, який перехоплює їх. Таким чином, в процесі може бути реалізований обробник для перехоплення сигналів SIGINT або SIGTERM. Оскільки обробник сигналів знає про те, який сигнал був отриманий, він може проігнорувати сигнал SIGINT і завершити додаток лише при отриманні, наприклад, сигналу SIGTERM. В лістингу 2 наведений приклад, к якому показано, як посилати сигнал SIGTERM завданню %2. Зауважте, що зразу ж після того, як сигнал був відісланий, статус процесу змінився на “Завершено”. Якби ми надіслали сигнал SIGINT, то статус процесу змінився би на “Переривання”. Миттю по тому настає очистка процесів, і завдання більше не відображається в списку.

Лістинг 2. Завершення процесу за допомогою SIGTERM

Обробники сигналів надають процесам значну гнучкість. Процес може виконувати свою звичайну роботу і перериватися по сигналу за для певних цілей. Крім того, що сигнали дозволяють процесам перехоплювати запити на завершення роботи і виконувати певні дії (наприклад, закривати файли або встановлювати контрольні точки у транзакціях, що виконуються), вони часто використовуються для того, щоб повідомити процесам-демонам про необхідність заново зчитати конфігураційний файл або виконати перезавантаження. Наприклад, можна послати певний сигнал процесу inetd, якщо ви змінили параметри мережі, або домену друку (lpd) після того, як ви додали новий принтер.



Безумовне завершення процесу

Деякі сигнали не можуть бути перехоплені (наприклад, деякі виключення апаратного рівня). Сигнал SIGKILL, який ви, найвірогідніше, будете

використовувати, не може бути перехоплений обробником сигналів і безумовно завершує процес. Взагалі, цей сигнал слід використовувати лише в тому випадку, коли всі інші спроби завершити процес виявилися марними.

Контрольні запитання

Дайте визначення сигналу?

Основні функції сигналів?

Із яких етапів складається процес сповіщення про події?

Що таке стандарт POSIX?

Назвіть основні сигнали, що визначені в стандарті POSIX.

Практична робота №7

Створення і видалення процесу в ОС Windows

Мета: Навчитись створювати і видаляти процеси в ОС Windows, використовуючи Assembler для Win 32.

Завдання.

Створити новий процес: розробити програму мовою Assembler для Win 32, яка створює процес, перериває процес, знищує процес.

Теоретичні відомості

Створення і керування процесом в ОС Windows.

Процес – це додаток, що виконується, і який складається із особового віртуального адресного простору, коду даних та інших ресурсів операційної системи, таких як файли, синхронізаційні об'єкти, що відомі для процесу.

Процес має складатися із використовуваного модуля, особистого адресного простору і гілки. У кожного процесу має бути хоча б одна гілка. Фактично гілка – це черга, що виконується.

Коли ОС Windows вперше створює процес, вона створює тільки одну гілку на процес, ця гілка зазвичай починає виконання з першої інструкції в модулі, якщо в подальшому знадобиться більше гілок він може сам створювати його. Коли Windows отримає команду для створення процесу, то вона створює особистий адресний простір для процесу, а потім завантажує виконуваний файл в просторі. Після цього ОС створює основну гілку для процесу.

Далі представлено синтаксис функції створення процесу в ОС Windows CreateProcess:

```
   CreateProcess proto lpAplicationName: DWORD, \
lpComandLine:DWORD, \
   lpProcessAttributes:DWORD, \ lpThreatAttributes:DWORD, \
   bInheritHandles: DWORD, \ dwCreationFlags: DWORD, \
lpEnviroment: DWORD,\
   lpCurrentDirectory: DWORD, \ lpStartupInfo: DWORD, \
lpProcessInformation: DWORD
```

Більшу частину параметрів можна ігнорувати.

lpAplicationName – Ім'я виконуючого файлу з або без шляху. Якщо параметр рівний нулю, маємо представити ім'я виконуючого файлу в параметрі lpComandLine.

lpComandLine – Аргументи командного рядка до програми, яку необхідно запустити.

lpProcessAttributes і *lpThreatAttributes* – Тут необхідно вказати атрибути безпеки для процесу і головної гілки. Якщо вони дорівнюють нулям (NULL), то використовуємо атрибути безпеки за замовчуванням.

bInheritHandles – прапор, який вказує, чи буде новий процес наслідувати всі відкриті *Handle* вашого процесу.

dwCreationFlags – декілька прапорів, які визначають поведінку процесу при його створенні, наприклад процес може бути створений і тут же призупинений, щоб була змога перевірити його або змінити перш ніж він зупиниться.

lpEnviroment – Вказівник на блок пам'яті, який містить декілька змінних оточень для нового процесу.

lpCurrentDirectory – Вказівник на рядок, який вказує на поточний диск і директорія для дочірнього процесу. NULL, якщо дочірній процес має унаслідувати її від батьківського процесу.

lpStartupInfo – Вказує на структуру *STARTUPINFO*, яка визначає як має з'явитися основне вікно нового процесу.

lpProcessInformation – вказує на структуру *PROCESS_INFORMATION*, яка отримує ідентифікаційну інформацію про новий процес.

Структура *PROCESS_INFORMATION* має наступні параметри.

```
PROCESS_INFORMATION STRUCT
hProcess     HANDLE ?;        хендл дочірнього процесу
process
   hThread    HANDLE ?;       хендл основної гілки дочірнього процесу
dwProcessId     DWORD ?;   ID дочірнього процесу
dwThreadId      DWORD ?;     ID основної гілки
PROCESS_INFORMATION ENDS
```

Хендл процесу і ID процесу – це різні речі. **ID процесу** – це унікальний ідентифікатор в системі. **Хендл процесу –** це значення, що повертає Windows для

використання іншими API-функціями, зв’язаними з процесами. Хендл процесу не може використовуватись для ідентифікації процесу, так як він не унікальний.

Після виклику функції *CreateProcess*, створюється новий процес і функція відразу повертається. Можна перевірити, чи є даний процес активним, викликавши функцію *GetExitCodeProcess,* яка має наступний синтаксис:

GetExitCodeProcess proto hProcess:DWORD, lpExitCode:DWORD

Якщо виклик цієї функції успішний, *lpExitCode* міститиме код виходу із процесу, що запитується. Якщо значення *lpExitCod*e дорівнює STILL_ACTIVE, тоді це означає, що процес все ще запущений. Можна примусово перервати процес, викликавши функцію TerminateProcess.

Хід виконання роботи.

Створюємо в блокноті файл MSGBOX.ASM:

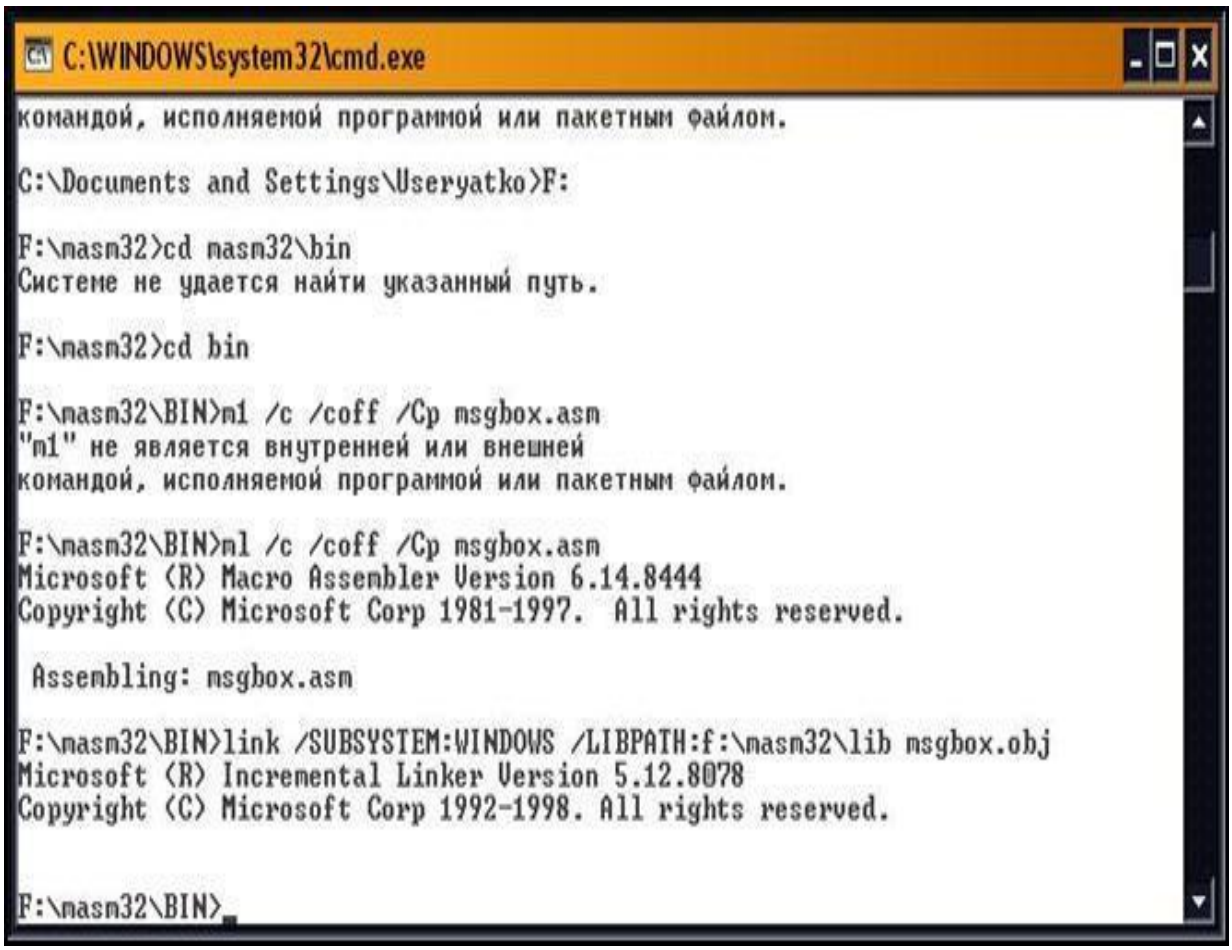
```
.386
.model flat,stdcall
option casemap:none

include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib

.data
MsgCaption db "Нове вікно",0
MsgBoxText db "Створено новий процес. Програма MSGBOX.EXE запущена на виконання і працює.",0

.code
start:
    invoke MessageBox, NULL, addr MsgBoxText, addr MsgCaption,
        MB_OK invoke ExitProcess, NULL
end start
```

Компілюємо його щоб отримати виконуваний файл MSGBOX.EXE

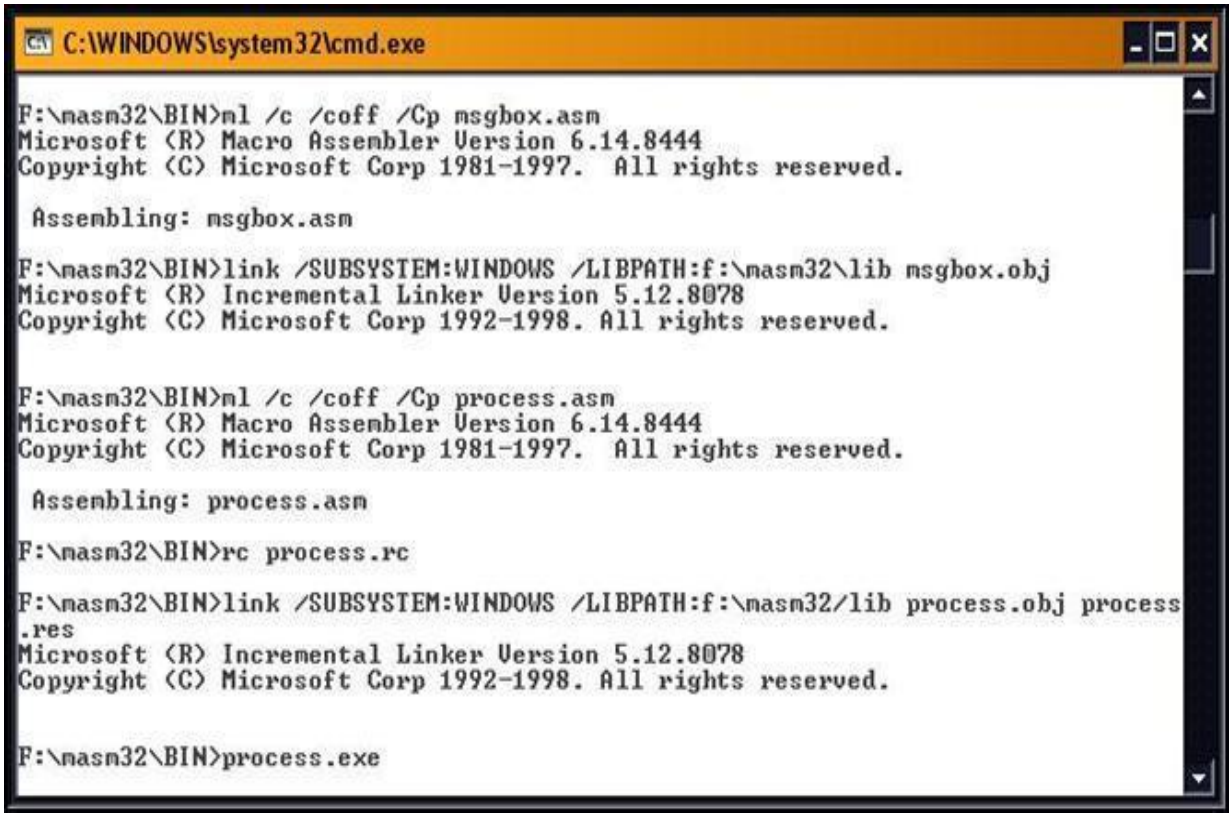


Створюємо у блокноті файл ресурсів PROCESS.RC

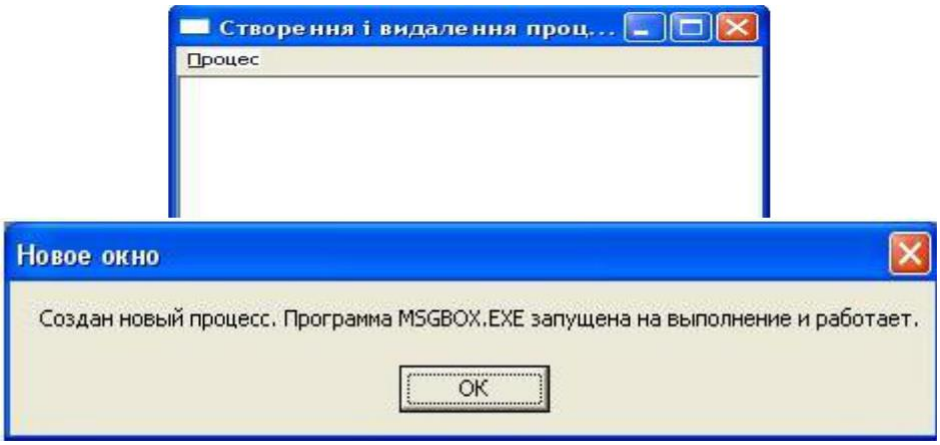
```
#define IDM_CREATE_PROCESS 1
#define IDM_TERMINATE 2
#define IDM_EXIT 3

FirstMenu MENU
{
POPUP "&Процес"
        {
        MENUITEM "&Створити процес",IDM_CREATE_PROCESS
        MENUITEM "&Перервати процес",IDM_TERMINATE
              MENUITEM SEPARATOR
              MENUITEM "&Вихід",IDM_EXIT
        }
}
```

Створюємо у блокноті файл основної програми PROCESS.ASM.
Компілюємо його:



Запускаємо на виконання PROCESS.EXE і отримуємо результати:



Підсвітивши заголовок вікна створення і видалення процесу, отримують доступ до пункту “Перервати процес” спливаючого вікна. Натискання на “Перервати процес” призводить до закривання “Нове вікно”.

Контрольні питання

Що таке процес в ОС?

Що включають в себе атрибути процесу?

Які існують переходи між станами процесів?

Що містить образ процесу?

Які дії виконує ОС при створенні нового процесу?

Які існують підходи до розробки ОС?

Практична робота № 8

Потоки в ОС Windows

Мета: Розробити програму мовою Asm для Win32, яка керує потоками за допомогою функції CreateThread.

Теоретичні відомості.

Потік – це ланцюг інструкцій. Завжди є можливість також створювати додаткові потоки у вашій програмі. Вважається, що мультипотоковість – це багатозадачність всередині однієї програми.

Потік – це функція, яка виконується паралельно з основною програмою. Можливо запускати декілька екземплярів однієї і тієї ж функції, або можливо запускати декілька функцій одночасно в залежності від вимог програміста. Мультипотоковість властива для Win 32. Потоки виконуються у тому ж процесі, тому вони мають доступ до всіх ресурсів процесу, але кожний потік має свій власний стек, тому локальні змінні в кожному потоці приватні. Кожний потік має свій власний набір регістрів, тому коли Windows перемикається на інший потік, попередній, запам'ятовує свій стан і може відновити його коли основа отримує контроль. Це забезпечується внутрішніми засобами Windows.

Потоки поділяються на дві категорії:

потік інтерфейсу користувача: потік такого типу створює своє власне вікно, тому він отримує двійкові сповіщення. Він може відповідати користувачу за допомогою свого вікна.

робочі потоки: цей тип потоків не створює вікна тому він не може приймати будь-які Windows-сповіщення та існує тільки для того, щоб виконувати призначену йому роботу на задньому фоні відповідно до його призначення.

Найкраща стратегія при використанні мультипотокових властивостей Win 32- це дозволити основному потоку робити все, що пов'язане з користувацьким інтерфейсом, а іншим виконувати роботу у фоновому режимі. В цьому випадку основний потік – головний, інші потоки – його помічники.

Ми можемо створити потік за допомогою виклику функції *CreateThread*, яка має наступний синтаксис:

```
    CreateThread proto LpThreadAttributes:DWORD,\
    dwStackSize: DWORD,\ LpStartAddress:
DWORD,\LpParameter: DWORD,\dwCreationFlags: DWORD,\
LpThreadid: DWORD
```

Функція *CreateThread* схожа на *CreateProcess*.

lpThreadAttributes - можливо використовувати NULL, якщо необхідно, щоб у потоку були установки безпеки за замовчуванням.

dwStackSize – вказати розмір стека потоку. Якщо необхідно, щоб потік мав такий же розмір стека, як і у основного, використовують NULL в якості параметру.

lpStartAddress – адреса функції потоку. Ця функція буде виконувати призначену для потоку роботу. Ця функція має отримувати один і тільки один 32-бітний параметр і повертати 32-бітне значення.

lpParametr – параметр, який необхідно передати функції потоку.

dwCreationFlags – 0 означає, що потік починає виконуватися відразу після створення. Для зворотнього можна використовувати прапор *CREATE_SUSPEND*.

lpThreadId – CreateThread – помістити сюди *ID* створеного потоку.

Якщо виклик *CreateThread* пройшов успішно, вона повертає хэндл створеного потоку, в зворотньому випадку вона повертає *NULL*.

Функція потоку запускається так швидко, як тільки закінчується виклик *CreateThread*, якщо тільки не позначено прапор *CREATE_SUSPEND*. В цьому випадку потік буде заморожений до виклику функції *ResumThread*.

Коли функція потоку повертається (за допомогою інструкції *ret*) Windows опосередковано викликає *ExitThreat* для функції потоку.

Можливо отримати код виходу потоку за допомогою функції *GetExitCodeThread*.

Існує три методи комунікації між потоками:

Використання глобальних змінних.

Windows повідомлення.

Події.

Потоки розглядають ресурси процесу, включаючи глобальні змінні, тому потоки можуть використовувати їх для того, щоб взаємодіяти один з одним. Але все таки, цей метод має використовуватись обережно. Синхронізацію потрібно уважно спланувати.

Також можливо використовувати Windows повідомлення, щоб виконувати взаємодію між потоками.

Якщо всі потоки мають користувацький інтерфейс, то немає проблем: цей метод може використовуватися для двохсторонньої комунікації. Все що потрібно зробити – це визначити один або більше додаткових Windows повідомлень, які будуть використовуватися з потоками. Визначаємо повідомлення використовуючи значення

WM_USER як базове, наприклад так:

WM_MYCUSTOMMSG equ WM_USER+100h

Якщо один із потоків має інтерфейс користувача, а інший є робочим ми не можемо використовувати даний метод для двостороннього користування, так як робочого потоку немає свого вікна, а відповідно і черги повідомлень. Можливо використовувати наступні схеми:

Потік з інтерфейсом користувача → глобальна змінна → робочий потік.

Робочий потік → Windows –повідомлення → Потік з користувацьким інтерфейсом.

Фактично ми будемо використовувати цей метод в нашому прикладі.

Хід виконання роботи

Створюємо в блокноті файл potok.asm.

```
386
.model flat,stdcall
option casemap:none

WinMain proto :DWORD,:DWORD,:DWORD,:DWORD

include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib

.const
IDM_CREATE_THREAD equ 1
IDM_EXIT equ 2
WM_FINISH equ WM_USER+100h
```

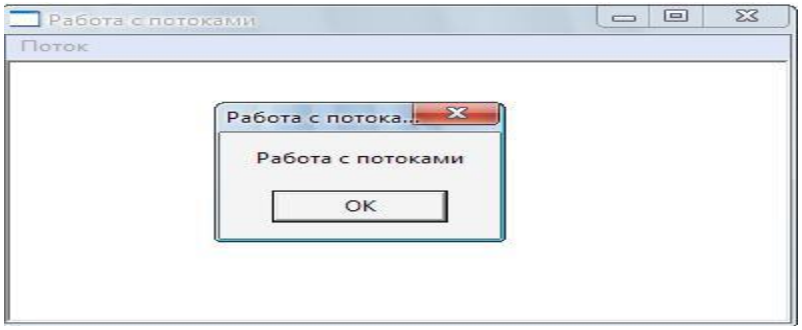
```
.data
……
……
……


ThreadProc PROC USES ecx
        Param:DWORD mov ecx,600000000
Loop1:
        add eax,eax
        dec ecx
        jz Get_out
        jmp Loop1
Get_out:
        invoke
        SendMessage,hwnd,WM_FINISH,NULL,NULL ret
ThreadProc ENDP
end start
```

Створюємо в блокноті файл potok.rc.

```
#define IDM_CREATE_THREAD 1
#define IDM_EXIT 2
FirstMenu MENU
{
POPUP "&Поток"
        {
        MENUITEM "&Створити поток",IDM_CREATE_THREAD
        MENUITEM SEPARATOR
       MENUITEM "&Вихід",IDM_EXIT
        }
}
```

Результат компіляції і запуску програми.



Контрольні питання

Що таке потік?

Назвіть елементи, що використовуються потоками одночасно?

Назвіть елементи, що використовуються потоками нарізно?

Які існують способи реалізації потоків?

Які переваги реалізації потоків в просторі користувача?

В яких ОС можлива реалізація паралельного виконання команд різних потоків?

ДОДАТОК А

ТЕСТОВІ ПИТАННЯ

Попередником якого сучасного сімейства ОС була ОС Minix Ендрю Таненбаума?

Windows

Linux

BSD

Які стани задачі являються загальноприйнятими й достатніми в загальному випадку?

Виконання

Підготовка

Готовність

Очікування

Закінчення

Чи може один екземпляр мікроядерної ОС фізично бути розділеним на декілька комп’ютерів?

Так

Ні

Яка нотація викликів функції прийнята в системних викликах Windows ?

Змішана нотація мов *C* і *Pascal* (Зворотній порядок аргументів, очистка стеку функцією)

Нотація мови *Pascal* (Прямий порядок аргументів, очистка стеку функцією)

Нотація мови *C* (Зворотній порядок аргументів, очистка стеку визиваючим кодом)

Чи можна безпосередньо звертатися до портів вводу/виводу в програмах запущених під управлінням ОС родини Windows 9x ?

Так

Ні

Чи мають ядра NT+ риси мікроядерної архітектури?

Ні, це чистий моноліт

Так, підтримка сервісів

Так, підтримка сервісів і так названа « виконуюча частина » в ядрі

Чи обов’язково в багатозадачній ОС буде присутня підтримка одночасного виконання декількох частин одної і тої ж програми?

Так

Ні

Яка з нижче наведених ОС є найбільш об’єктивно орієнтованою ?

Всі Linux

OS/2

BeOS

Як при мікроядерній архітектурі називаються додатки користувача?

Сервери

Клієнти

Сервіси

Програми

Споживачі

В якій із ОС вперше був реалізований стек протоколів TCP/IP ?

DOS

Linux

BSD

Windows

AmigaOS

В чому головний недолік монолітних ядер?

Вони займають дуже багато операційної пам’яті

Їх неможна модифікувати під час роботи

Із часом вони настільки збільшуються, що різко ускладнюється внесення всяких змін

Укажіть типи повідомлень, котрі можуть використовуватись в мікроядерних ОС.

Синхронне і асинхронне
Тільки асинхронне
Тільки синхронне

Які ядра сучасних ОС підтримують Multiboot Speciation?

Linux
SunOS
MacOS
GNU Hurd
Windows
Всі ядра BSD

Які із перерахованих нижче видів архітектури ядер існують?

Мікроядра
Екзоядра
Монолітні ядра

Укажіть основні функції ОС .

Розділення ресурсів комп'ютера між програмами.
ОС повинна враховувати апаратні особливості.
Захист програм одна від одної і від самих себе.
ОС повинна запропонувати програмам уніфікувати інтерфейс взаємодій з іншими програмами і апаратурою (через драйвери).

Яка оптимальна довжина кванта процесорного часу для ОС загального призначення?

Декілька сотень мілісекунд
Декілька мілісекунд
Чим менше, тим краще

В яких ОС підтримка графічного інтерфейсу користувача (GUI) інтегрована безпосередньо в ядро?

Windows
Linux
MacOS
BeOS
BSD

Чи знайшли екзоядерні ОС широке застосування в сучасній обчислювальній техніці?

Ні

Так

Вкажіть основний засіб міжпроцесної взаємодії в мікроядерних архітектурах.

Віддалені виклики процедур (RPC, Remote Procedure Call)

Сповіщення

Потоки

Загальна (розділювана) пам’ять

В якій операційній системі вперше було реалізовано графічний інтерфейс користувача?

Windows

AmigaOS

BSD

UNIX

21. Що означає абревіатура PIC в контексті ОС?

Portable Incompatible Code

Programmable Interrupt Controller

Position Independent Code

Past Implemented Code

Які існують види багатозадачності?

Кругова багатозадачність.

Кооперативна (невитісняюча) багатозадачність.

Витісняюча багатозадачність.

Симетрична багатозадачність.

23. Який основний недолік мікроядерної архітектури?

Проблеми з переносністю

Проблеми з продуктивністю

Проблеми з безпекою

Де розміщено драйвери по відношенню до ядра в ОС сімейства Linux?

Лише в ядрі

Зовні ядра, лише в окремих процесах

І в ядрі, і в окремих процесах (але останнє лише при увімкненій підтримці в ядрі)

І в ядрі, і в окремих процесах

Як (в ідеалі) повинне проходити перенесення ядра ОС з однієї апаратної платформи на іншу?

Переписуванням яких-небудь його частин

Переписуванням лише однієї, виділеної частини

Повним переписуванням коду

Як при мікроядерній архітектурі називаються додатки користувача?

Програми

Сервіси

Сервери

Споживачі

Клієнти

Вкажіть основний засіб міжпроцесової взаємодії в мікроядерних архітектурах.

Дистанційні виклики процедур (RPC, Remote Procedure Call)

Повідомлення

Потоки

Спільна пам’ять

Чи знайшли екзоядерні ОС широке застосування в сучасній обчислювальній техніці?

Ні

Так

У чому головний недолік монолітних ядер?

Їх не можна модифікувати під час роботи

Вони займають занадто багато оперативної пам'яті

З часом вони настільки розростаються, що різко ускладнюється внесення будь-яких змін

30. В якій з ОС вперше був реалізований стек протоколів TCP / IP?

Linux
DOS
BSD
Windows
AmigaOS

31. Чи мають ядра NT + риси мікроядерної архітектури?

Так, підтримка сервісів і так звана "виконувана частина" в ядрі
Ні, це чистий моноліт
Так, підтримка сервісів

32. Які ядра сучасних ОС підтримують Multiboot Specification?

Windows
Linux
GNU Hurd
Все ядра BSD
SunOS
MacOS

33. Чи обов'язково в багатозадачній ОС буде присутня підтримка одночасного виконання декількох частин однієї і тієї ж програми?

Так
Ні

34. Яка оптимальна тривалість кванта процесорного часу для ОС загального призначення?

Чим менше, тим краще.
Кілька сотень мілісекунд.
Кілька мілісекунд

35. Вибрати всі правильні твердження про відношення DOS до перших версій Windows?

У Windows можна було запускати додатки DOS
Підтримка програм DOS була обмеженою і неповною (при емуляції на VDM, в рамках режиму V86)
Багато функцій Windows делегувалися відповідним функціям DOS (тобто для цього виконувалося перемикання режимів роботи ЦП)

36. Які існують види багатозадачності?

Кругова багатозадачність.

Кооперативна (невитісняюча) багатозадачність.

Витісняюча багатозадачність.

Симетрична багатозадачність

37. Яка з нижче перерахованих ОС є найбільш об'єктно-орієнтованою?

Windows 2k

OS/2

Всі Linux

BeOS

FreeBSD

38. Який основний недолік мікроядерної архітектури?

Проблеми з переносимістю

Проблеми з продуктивністю

Проблеми з безпекою

Де розташовуються драйвери по відношенню до ядра в ОС сімейства Linux?

Тільки в ядрі

Поза ядром, тільки в окремих процесах

І в ядрі, і в окремих процесах (але в останніх тільки при включеній підтримці в ядрі)

І в ядрі, і в окремих процесах

40. Як (в ідеалі) повинне відбуватися перенесення ядра ОС з однієї апаратної платформи на іншу?

Переписуванням певних його частин

Переписуванням тільки однієї, виділеної частини

Повним переписуванням коду

ДОДАТОК Б

ТЕМИ ДЛЯ САМОСТІЙНОЇ РОБОТИ СТУДЕНТІВ

Перші операційні системи: причини виникнення і можливості.
Історія створення і розвитку ОС UNIX.
Історія створення і розвитку ОС Windows.
Особливості ОС Windows 10.
Історія створення і розвитку ОС Linux.
Історія створення і розвитку ОС Solaris (фірма Sun).
Технологія симетричної многопроцесорності (*symmetric multiprocessing - SMP*).
Стани процесів: необхідність свопінгу.
Міжпроцесова взаємодія: базові механізми міжпроцесової взаємодії.
Міжпроцесова взаємодія: технології передавання повідомлень.
Сторінково-сегментна організація віртуальної пам’яті.
Реалізація керування основною пам’яттю в ОС Linux.
Взаємодія з диском під час керування пам’яттю: завантаження сторінок на вимогу; особливості підкачування сторінок.
Взаємодія з диском під час керування пам’яттю: визначення продуктивності завантаження на вимогу.
Огляд концепції та реалізацій хмарних обчислень.
Хмарна платформа Microsoft Windows Azure.
Огляд Google Android.
Аналітичний огляд і порівняння можливостей ОС для мобільних пристроїв.

СЛОВНИК ОСНОВНИХ ТЕРМІНІВ

Адресний простір (*Address space*) – діапазон адрес, що доступні для комп'ютерної програми.

Архітектура операційної системи – структурна і функціональна організація ОС на основі деякої сукупності програмних модулів.

Алгоритм Деккера – голландський математик Т. Деккер був першим, хто розробив програмне рішення проблеми взаємного виключення (1970 р.).

Багатопоточність – виконання декількох потоків в одному процесі.

Відкачування і підкачка (swapping) – це дії операційної системи з відкачування (запису) образу неактивного процесу на диск або підкачування (зчитування) активного процесу в основну пам'ять.

Віртуальна пам'ять *сторінкова* – організує переміщення даних між основною пам'яттю і диском сторінками – частинами віртуального адресного простору фіксованого і порівняно невеликого розміру.

Віртуальна пам'ять *сегментна* – передбачає переміщення даних сегментами – частинами віртуального адресного простору довільного розміру.

Віртуальна пам'ять *сегментно-сторінкова* – використовує дворівневий поділ, при якому віртуальний адресний простір поділяється на сегменти, а потім сегменти поділяються на сторінки. Одиницею переміщення даних є сторінка.

Взаємне блокування – це блокування групи процесів, що очікують події, яка може ніколи не настати, оскільки дану подію може викликати тільки інший процес з тієї ж групи. Взаємоблокування зазвичай виникають при доступі процесів до ресурсів.

Використання (завантаження) процесору – відсоток часу протягом якого процесор зайнятий.

Диспетчер пам'яті (***MMU – Memory Management Unit***) – відображує віртуальні адреси на фізичні адреси пам'яті.

Диспетчеризація без розподілу процесорного часу (*багатозадачність без витіснення*) – це такий спосіб диспетчеризації, при якому активний процес виконується до тих пір, поки він за власною ініціативою не передасть управління диспетчеру задач для вибору іншого, готового до виконання процесу.

Диспетчеризація з розподілом процесорного часу (*багатозадачність з витісненням*) – це такий спосіб, де рішення про перемикання з однієї задачі на іншу приймається диспетчером задач, а не власною активною задачею.

Завантажувач операційної системи – невелика програма, яка починає процес ініціалізації операційної системи після включення живлення або перезавантаження комп'ютера.

Кеш-пам’ять (*Cache memory*) – пам’ять, що менша за розміром і швидша, ніж основна; розташована між процесором і основною пам’яттю. Кеш буфером для комірок пам’яті, звернення до яких були останніми.

Критичний ресурс (КР) – ресурс, до якого в кожний момент часу можливий доступ тільки одного процесу. Доступ до такого ресурсу здійснюється в **критичній секції (КС)** – частині коду, яка в будь-який момент часу може виконуватися тільки одним процесом.

Логічна адреса – адреса, що генерована процесором при виконанні машинної команди.

Макроядро (*Macrokernel*) – велике ядро операційної системи, що забезпечує широкий діапазон сервісів.

Мікроядро (*Microkernel*) – мале привілегійоване ядро операційної системи, що забезпечує планування процесів, управління пам’яттю; для виконання інших функцій, які традиційно пов’язують з ядром операційної системи, використовує окремі процеси.

Монітор – це набір процедур, змінних і інших структур даних, об'єднаних в особливий модуль. Він являє собою високорівневу конструкцію мови програмування, яка забезпечує функціональність, еквівалентну до функціональності семафорів, але при цьому ним набагато легше управляти.

Мультипрограмування – це такий спосіб організації обчислень, коли на однопроцесорній обчислювальній системі імітується одночасне виконання декількох програм.

М’ютекс – це спрощена версія семафора: змінна, яка може знаходитися в одному з двох станів – блокованому і неблокованому.

Операційна система (ОС) – це програма, що керує роботою інших програм, як прикладних, так і системних, і є інтерфейсом між програмами і апаратним забезпеченням комп'ютера.

Операційне середовище – це інтерфейси, що необхідні програмам і користувачам для отримання сервісів ОС. Операційне середовище включає в себе **програмне середовище** та інтерфейс користувача.

Образ процесу (*Process image*) – містить програму (код), дані, стек і атрибути процесу. Структура даних разом з атрибутами процесу часто називається його **керуючим блоком.**

Пам'ять в операційних системах – це ресурс комп'ютера, призначений для зберігання програмного коду і даних.

Паралельні процеси *(concurrent),* якщо їх виконання може перекриватися в часі, тобто, наприклад, другий процес починається раніше, ніж завершується перший.

Пастка (*Trap*) – непрограмуємий умовний перехід до певної адреси, що автоматично активується апаратним забезпеченням; позиція з якої було здійснено перехід записується.

Переривання (*Interrupt*) – призупинка нормального виконання процесу (комп'ютерної програми), що викликана зовнішньою по відношенню до процесу подією і здійснюється таким чином, що по її завершенні виконання процесу може бути продовжено.

Планування – це системний процес, який здійснює установку користувацьких процесів в чергу та визначення атрибутів їх виконання в рамках використовуваної обчислювальної системи.

Планування *короткострокове* – **диспетчеризація –** рішення про додавання процесу в пул процесів, що виконуються.

Планування *середньострокове* – рішення про додавання процесу до числа процесів, що повністю або частково розміщені в оперативній пам'яті.

Планування *довгострокове* – рішення про те, який з доступних процесів буде виконуватися процесором (полягає у виборі таких обчислювальних процесів, які менше всього б конкурували між собою в процесі досягнення мети обчислень).

Послідовні процеси *(serial),* якщо вони не є паралельними.

Потік – це незалежно планований контекст виконання, що розділяє єдиний адресний простір з іншими потоками свого процесу.

Програмне середовище – це системне програмне оточення, що дозволяє виконувати системні запити від прикладних програм.

Процес – це окрема програма в момент її виконання, а також виділені їй ресурси комп’ютера.

Пропускна здатність процесора – це кількість процесів, яка виконується процесором за одиницю часу;

Реєнтерабельна процедура (*Reentrant procedure*) – Підпрограма, вхід в яку може бути здійснено до завершення її попереднього виконання.

Ресурс – це будь-який об'єкт, який може запитуватися і очікуватися процесом; ресурс може складатися з будь-якої кількості ідентичних одиниць, і процес може запитувати будь-яку кількість одиниць ресурсу.

Розділ – це безперервна частина фізичного диска, яку операційна система являє користувачу як окремий логічний пристрій (часто використовуються також назви «логічний диск» і «логічний розділ»).

Розподіл часу (*Time sharing*) – паралельне використання пристрою декількома користувачами.

Розподілена операційна система – спільна операційна система, що сумісно використовується мережею комп’ютерів. Розподілена операційна система забезпечує підтримку для міжпроцесної взаємодії, переміщення процесів, взаємовиключення, а також попередження і виявлення взаємоблокувань.

Семафор – це спеціальна змінна, яка має ціле значення і пов'язану з ним чергу.

Симетрична багатопроцесорність (*Symmetric Multiprocessing, SMP*) – вид багатозадачності, що дозволяє операційній системі працювати на будь-якому із доступних процесорів або на декількох доступних процесорах одночасно.

Система реального часу (*Real-time system*) – операційна система, яка може планувати завдання реального часу і керувати ними.

Спулінг (*Spooling*) – використання вторинної пам’яті в якості буферу для зниження затримок при передаванні даних між периферійними пристроями і процесорами комп’ютера.

Сторінка (*Page*) – У віртуальній пам’яті блок фіксованої довжини, що має віртуальну адресу, яка передається між основною і вторинною пам’яттю.

Утиліти – програми, які виконують окремі завдання управління і супроводу обчислювальної системи.

Файл (*file*) – це поіменована область зовнішньої пам'яті, в яку можна записувати і з якої можна зчитувати дані (суміжна область логічного адресного простору).

Файлова система (ФС) – це частина операційної системи, що включає в себе: сукупність всіх файлів на диску, набори структур даних, використовуваних для управління файлами, комплекс системних програмних засобів, що реалізують операції над файлами.

Файл відкачування *(backing store)* – область дискової пам'яті, використовувана операційною системою для зберігання образів відкачаних процесів. Файл відкачування організується максимально ефективно: забезпечується прямий доступ до всіх образів процесів в пам'яті (наприклад, через таблицю за номером процесу).

Фізична адреса – це реальна адреса в пам'яті, яку "бачить" і "розуміє" пристрій управління пам'яттю (Memory Management Unit – MMU).

Фізична організація файлу – це спосіб розміщення файлу на диску.

Час обороту – інтервал часу від моменту появи процесу у вхідній черзі до моменту завершення процесу. Цей час обороту включає в себе час очікування у вхідній черзі, час очікування у черзі готовності, час готовності у чергах до периферійних пристроїв, час виконання на процесорі та час вводу/виводу;

Час очікування – це сумарний час знаходження процесу в черзі очікування готових процесів;

Час відповіді – це час від моменту надходження процесу на вхідну чергу до моменту першого звернення процесу на ввід/вивід даних.

Ядро (*Kernel*) операційної системи — це набір функцій, структур даних та окремих програмних модулів, які завантажуються в пам’ять комп’ютера при завантаженні операційної системи та забезпечують три типи системних сервісів: управління вводом-виводом інформації; управління оперативною пам’яттю; управління процесами.

Application Programming Interface (***API)*** – інтерфейс прикладного програмування.

Windows Executive (***виконавча система Windows***) – це набір компонентів, відповідальних за найважливіші служби ОС (керування пам'яттю, процесами і потоками, вводом-виводом тощо).

FAT (*file allocation table*) – таблиця розміщення файлів.

Instruction Set Architecture – архітектура системи команд.

Hardware Abstraction Layer (**HAL**) – рівень апаратних абстракцій.

Page fault (*переривання із-за відсутності сторінки*) – відбувається тоді, коли сторінка, що містить слово, до якого здійснюється звернення, відсутня в основній пам’яті. Це викликає генерацію переривання, обробник якого завантажує необхідну сторінку в основну пам’ять.

Symmetric Multiprocessing, SMP – симетрична багатопроцесорність.

Swapping ***(відкачування і підкачка)*** – це дії операційної системи з відкачування (запису) образу неактивного процесу на диск або підкачування (зчитування) активного процесу в основну пам'ять. Необхідність виконання подібних дій викликана нестачею основної пам'яті.

Job control language, **JCL (*мова керування завданнями*) –** мова, що розроблена для опису завдань та їх вимог до операційної системи.

ЛІТЕРАТУРА

Stallings, William. Operating systems: internals and design principles / William Stallings. – 7th ed. Prentice Hall, New Jersey, 2012, p.769. ISBN-13:978-0-13-230998-1

Kusswurm Daniel. Modern X86 Assembly Language Programming/ Daniel Kusswurm. - Apress, 2019. — 604 p.

William Stallings. Operating Systems: Internals and Design Principles, 9th Edition. – Pearson, 2018. ISBN-10: 0-13-467095-7 | ISBN-13: 978-0-13-467095-9.

Бэкон Дж. Операционные системы / Дж. Бэкон, Т Харрис. – К.: Издат. группа BHV; СПб.: Питер, 2004. – 800 с.

Вахалия Ю. UNIX изнутри / Ю. Вахалия. – СПб.: Питер, 2003. – 844 с.

Гордеев А. В. Системное программное обеспечение / А. В. Гордеев, А. Ю. Молчанов. - СПб.: Питер, 2003. – 736 с.

Дейтел П. Дж. Операционные системы. Часть 1. Основы и принципы / П. Дж. Дейтел, Х. М. Дейтел, Д. Р. Чофнес. 3-е изд.: Пер. с англ. – Бином-Пресс, 2011. – 1024 с.

Зыль С. Операционная система реального времени QNX. От теории к практике / С. Зыль. – 2-е изд. - СПб.: БХВ-Петербург, 2004. – 192 с.

Иртегов Д. В. Введение в операционные системы / Д. В. Иртегов. – 2-е изд. СПб.: БХВ-Петербург, 2012. – 1040 с.

10. *Митчелл М.* Программирование для Linux. Профессиональный подход / М. Мітчелл, Д. Оулдем, А. Самьюэл. – М.: Вильямс, 2002. – 288 с.
11. *Моли Б.* UNIX/Linux. Теория и практика программирования / Б. Моли. – М.: Кудиц-Образ, 2004. – 576 с.
12. *Олифер В. Г.* Компьютерные сети. Принципы, технологии, протоколы / В. Г. Олифер, Н. А. Олифер. – 2-е изд. - СПб.: Питер, 2003. – 864 с.
13. *Олифер В. Г.* Сетевые операционные системы / В. Г. Олифер, Н. А. Олифер. – СПб.: Питер, 2001. – 544 с.

14. *Померанц О.* Ядро Linux. Программирование модулей / О. Померанц. - М.: Кудиц-Образ, 2000. – 112 с.
15. *Робачевский А. М.* Операционная система UNIX / А. М. Робачевский, С. Немнюгин, О. Стесик. - СПб.: Петербург, 2010. – 656 с.
16. *Стивенс У. Р.* UNIX: взаимодействие процессов / У. Р. Стивенс. - СПб.: Питер, 2003. – 576 с.
17. *Стивенс У. Р.* UNIX: разработка сетевых приложений / У. Р. Стивенс, Б. Феннер, Э. М. Рудофф. – 3-е изд. - СПб.: Питер, 2007. – 1040 с.
18. *Столлингс В.* Операционные системы / В. Столлингс. - 4-е издание.: Пер. с англ. – М.: Издательский дом «Вильямс», 2002. – 848 с.
19. *Таненбаум Э.* Компьютерные сети / Э. Таненбаум. – 4-е изд. - СПб.: Питер, 2004. – 992 с.
20. *Таненбаум Э.* Операционные системы / Э. Таненбаум. – 3-е изд. - СПб.: Питер, 2010. –1120с.
21. *Фуско Дж.* Linux. Руководство программиста / Дж. Фуско. - СПб.: Питер, 2010. – 448 с.
22. *Харт Дж. В.* Системное программирование в среде Win32 / Дж. В. Харт. – М.: Вильямс, 2001. – 464 с.
23. *Haseman Chris.* Android Essentials / Chris Haseman. – Apress, 2008. – 116 p.
24. Уорд Б. Внутреннее устройство Linux / Б. Уорд – Питер, 2016. – 864 с.
25. Шоттс У. Командная строка Linux. Полное руководство / У. Шоттс - Питер, 2017. – 480 с.

Навчально-методичне видання

Федотова-Півень Ірина Миколаївна,
Миронець Ірина Валеріївна,
Півень Олег Борисович,
Сисоєнко Світлана Володимирівна,
Миронюк Титяна Василівна

ОПЕРАЦІЙНІ СИСТЕМИ

Навчальний посібник

За редакцією проф. В. М. Рудницького

Формат 60 ×84/16. Папір офсетний. Гарнітура TimesNewRoman.
Ум. др. арк. 12,56. Зам № 0317/9-19. Наклад 300 пр.

Видавництво ТОВ «ДІСА ПЛЮС»
Свідоцтво суб'єкта видавничої справи: серія ДК № 4047 від 15.04.2011 р.
61029, м. Харків, шосе Салтівське, буд. 154. Тел. (057) 768-03-15,
e-mail: disadruk@gmail.com

Надруковано з готових оригінал-макетів у друкарні ФОП Петров В. В.
Єдиний державний реєстр юридичних осіб та фізичних осіб-підприємців.
Запис № 2400000000106167 від 08.01.2009 р.
61144, м. Харків, вул. Гв.Широнінців, 79в, к. 137, тел. (057) 78-17-137. e-mail:bookfabrik@mail.ua