Конспект лекцій курсу

ОПЕРАЦІЙНІ СИСТЕМИ

Всеволод Дьомкін
КПІ, Київ, 2016

Вступ

ОС

ОС - це комплекс управляючих і обробляючих програм, які, з одного боку, виступають як інтерфейс між пристроями обчислювальної системи і прикладними програмами, а з іншого боку - призначені для управління пристроями і обчислювальними процесами, ефективного розподілу ресурсів між обчислювальними процесами і організації надійних обчислень.

Рис. 0.1. Місце ОС

Завдання ОС:

- Управління апаратною частиною (менеджер ресурсів)
- Абстракція апаратної частини (віртуальна машина)
- Ізоляція додатків від апаратної частини (щоб уникнути псування)

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Програма в пам'яті

Програма Hello World:

```
#include <stdio.h>
#include <unistd.h>
int main ()
{
    printf("Hello World");
    exit(0);
}
```

Один з варіантів дизасемблювання:

```
.section .rodata
.LC0:
    .string “Hello World”
.text
.globl  main
.type   main,@function
main:
    pushl    %ebp
    movl     %esp, %ebp
    subl     $8, %esp
    subl     $12, %esp
    pushl    $.LC0
    call     printf
    addl     $16, %esp
    subl     $12, %esp
    pushl    $0
    call     exit
```


* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

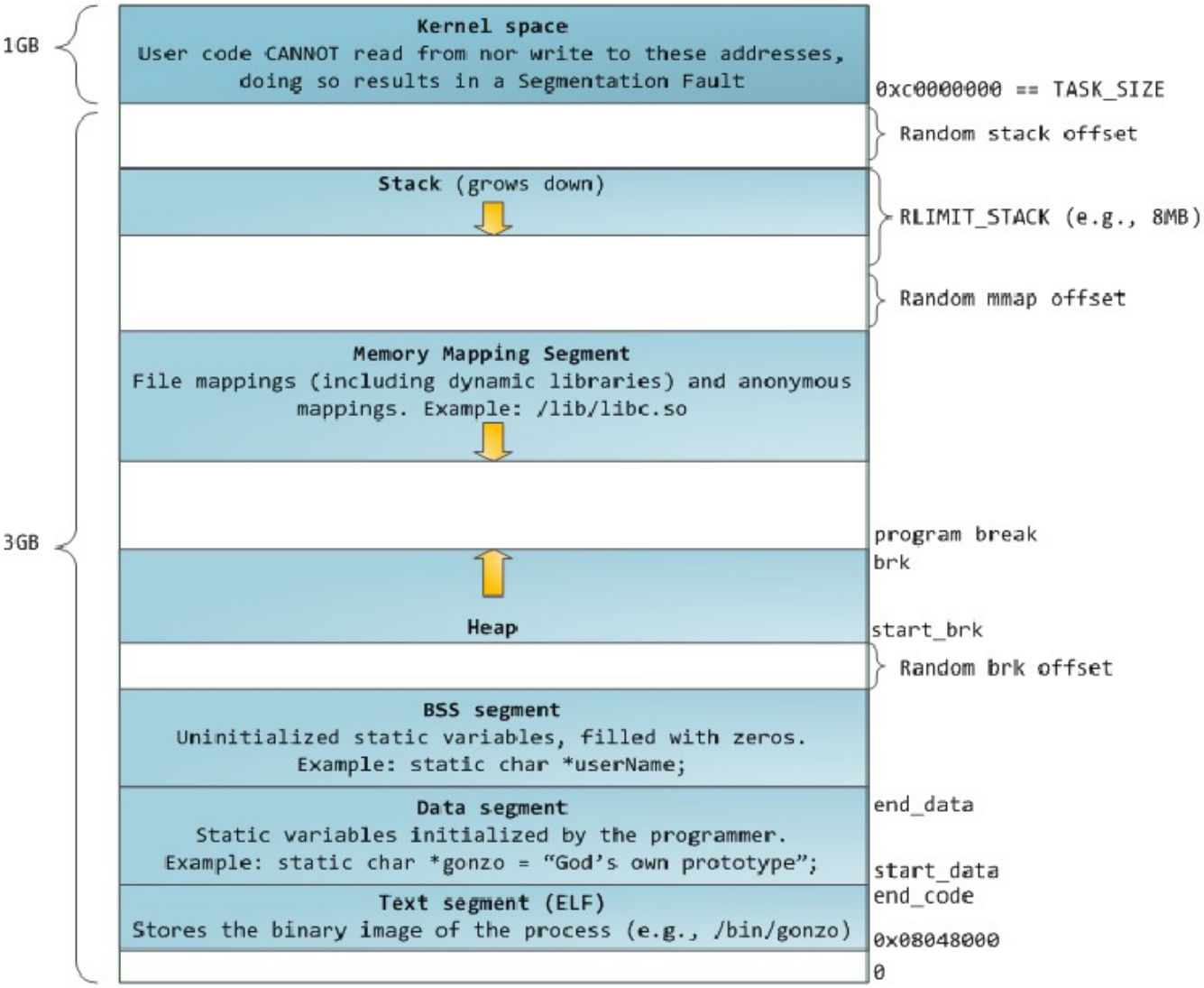

Рис. 0.2. Програма в пам'яті

Ядро ОС

Ядро ОС - це центральна частина операційної системи, що забезпечує додаткам координований доступ до ресурсів комп'ютера, таким як процесорний час, пам'ять, зовнішнє апаратне забезпечення, зовнішній пристрій введення і виведення інформації. Також зазвичай ядро   надає сервіси файлової системи і мережевих протоколів.

Ядро - теж програма.

Варіанти реалізації ядра:

- Монолітне: одна монолітна програма в пам'яті - +простота, +швидкість, -помилки, -перекомпіляція

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

- Модульне: монолітна програма, що надає інтерфейс завантаження і вивантаження доп.модулей - знімає проблему перекомпіляції
- Мікроядро: кілька програм, які взаємодіють через передачу повідомлень - +ізоляція, +слабка зв'язність, -складність, -швидкість
- Наноядро: ядро тільки управляє ресурсами (обробка переривань)
- Екзоядро: наноядро з координацією роботи процесів
- Гібридне

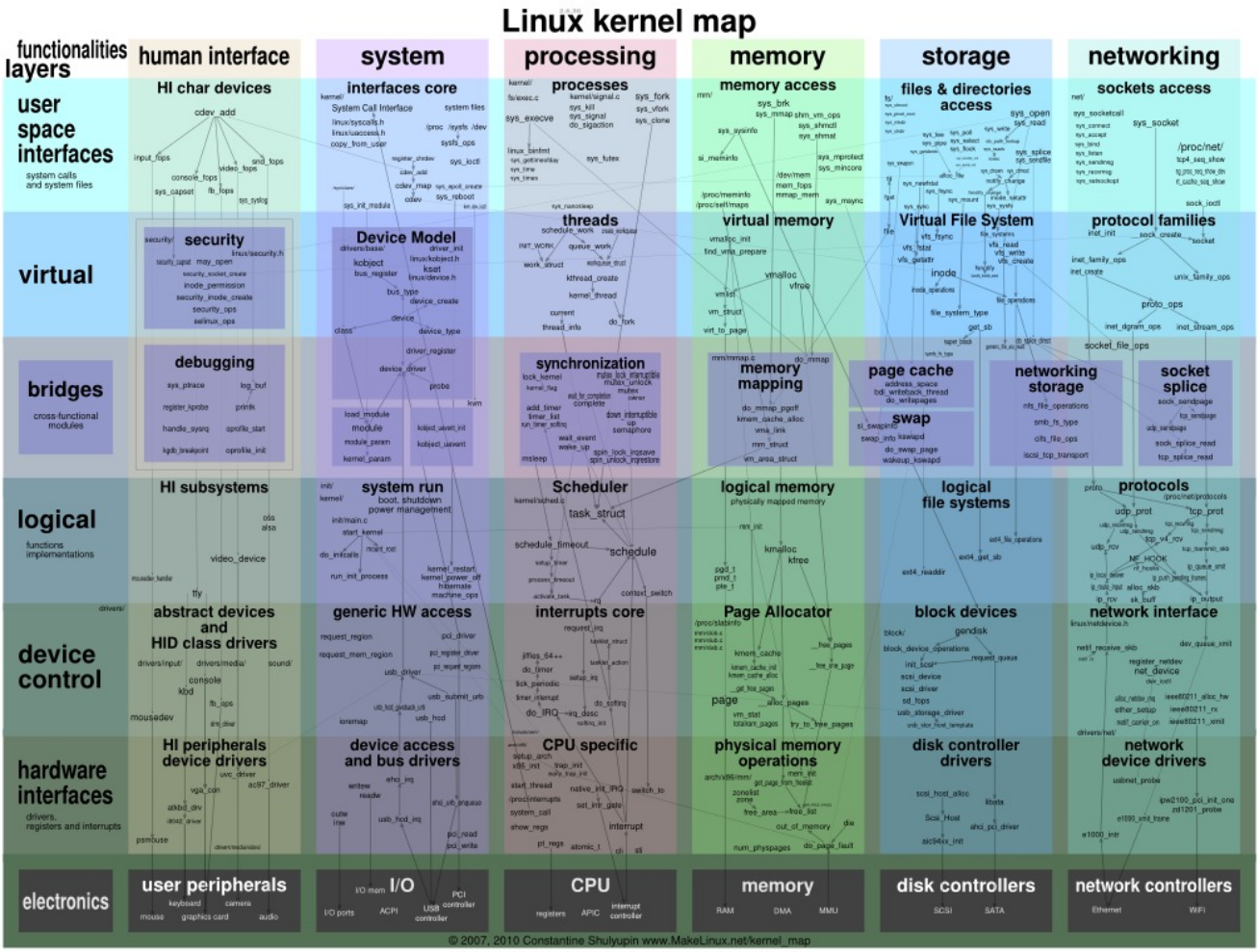

Рис. 0.3. Ядро Linux, http://www.makelinux.net/kernel_map/

Апаратна архітектура

Різні архітектури:

- Фон Неймана
- Гарвардська
- Стекові машини

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

- Lisp Machine
- FPGA
- та інші

Архітектура фон Неймана

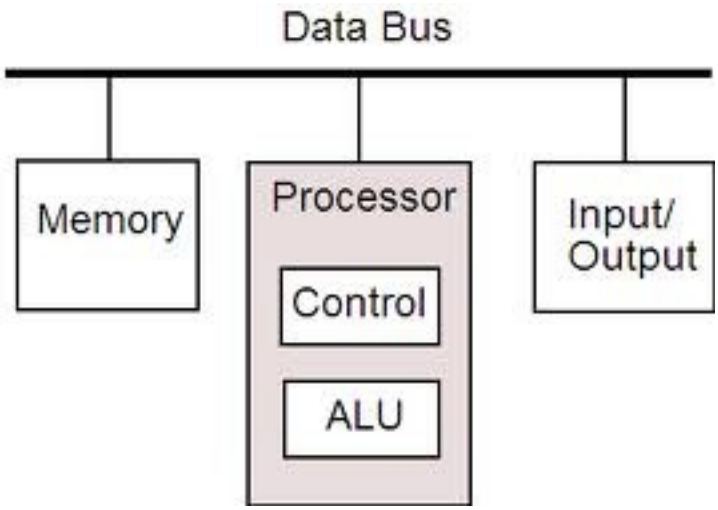

Рис. 0.4. Архітектура фон Неймана

Принципи фон Неймана:

.

Двійкове кодування

Згідно з цим принципом, вся інформація, що надходить в ЕОМ, кодується за допомогою двійкових сигналів (двійкових цифр, бітів) і розділяється на одиниці, звані словами.

.

Однорідність пам'яті

Програми та дані зберігаються в одній і тій же пам'яті. Тому ЕОМ не розрізняє, що зберігається в даній комірці пам'яті - число, текст або команда. Над командами можна виконувати такі ж дії, як і над даними.

.

Адресованість пам'яті

Структурно основна пам'ять складається з пронумерованих осередків; процесору в довільний момент часу доступний будь-який осередок. Звідси випливає можливість давати імена областям пам'яті, так, щоб до значень, які в них знаходяться, можна було б згодом звертатися або міняти їх в

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

процесі виконання програми з використанням привласнених імен.

.

Послідовного програмного керування

Програма складається з набору команд, які виконуються процесором автоматично один за одним у певній послідовності.

.

Жорсткості архітектури

Незмінюваність в процесі роботи топології, архітектури, списку команд.

Von Neumann Bottleneck:

Спільне використання шини для пам'яті програм і пам'яті даних призводить до появи вузького місця архітектури фон Неймана, а саме обмеження пропускної спроможності каналу між процесором і пам'яттю в порівнянні з об'ємом пам'яті. Через те, що пам'ять програм і пам'ять даних не можуть бути доступні в один і той же час, пропускна здатність є значно меншою, ніж швидкість, з якою процесор може працювати. Це серйозно обмежує ефективну швидкодію при використанні процесорів, необхідних для виконання мінімальної обробки на великих обсягах даних. Процесор постійно змушений чекати необхідних даних, які будуть передані в пам'ять або з пам'яті. Так як швидкість процесора і об'єм пам'яті збільшувалися набагато швидше, ніж пропускна здатність між ними, вузьке місце стало великою проблемою, серйозність якої зростає з кожним новим поколінням процесорів.

Рис. 0.5. Ієрархія пам'яті в комп'ютері

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Архітектура x86

http://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture

Основні особливості:

- Набір інструкцій процесора (CICS), інструкції двох типів: арифметико-логічні (ADD, AND, ...) і керуючі (MOV, JMP, ...)
- Обмежена кількість регістрів: кілька регістрів загального призначення (A, B, C, D), розмір яких дорівнює довжині машинного слова, і кілька спеціальних регістрів (IP, FLAGS, ...). АЛП процесора може працювати тільки з регістрами загального призначення. ПУ процесора займається модифікацією значень регістрів або переміщенням даних між регістрами й пам'яттю

Режими роботи процесора:

- Реальний: пряма адресація пам'яті
- Захищений: непряма адресація пам'яті з використанням модуля управління пам'яттю (MMU)
- та інші допоміжні

Робота процесора:

```
while (1) {
    execute_instruction(read_memory(IP));
    // IP - регістр-вказівник інструкції
}
```

Віртуальна машина

Віртуальна машина - це (ефективний) ізольований дуплікат реальної машини.

Критерії ефективної віртуалізації Попека-Голдберга

Можливі завдання:

- Емуляція різних архітектур
- Реалізація мови програмування
- Збільшення переносимості коду

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

- Дослідження продуктивності ПО або нової комп'ютерної архітектури
- Оптимізації використання ресурсів комп'ютерів
- Захист інформації та обмеження можливостей програм (пісочниця)
- Впровадження шкідливого коду для управління інфікованої системою
- Моделювання інформаційних систем різних архітектур на одному компьютері
- Спрощення адміністрування

Типи:

- Системна (гіпервізор)
- Процесна

Типи гіпервізорів:

- Автономний
- На основі базової ОС
- Гібридний

POSIX

POSIX - Portable Operating System Interface for Unix - Переносимий інтерфейс операційних систем Unix - набір стандартів, що описують інтерфейси між операційною системою і прикладною програмою.

Відкриті стандарти - це стандарти, які публікуються у відкритих джерелах і, як правило, мають одну або кілька (часто обов'язковою є наявність мінімум двох) еталонних реалізацій (reference implementation). Також, як правило, такі стандарти розробляються в рамках чітко визначеного процесу.

Інтерфейс - сукупність правил (описів, угод, протоколів), що забезпечують взаємодію пристроїв і програм в обчислювальній системі або сполучення між системами. Це зовнішнє подання, абстракція якогось інформаційного об'єкта. Інтерфейс розділяє методи зовнішньої взаємодії і внутрішньої роботи. Один об'єкт може мати декілька інтерфейсів для різних "споживачів". Інтерфейс - це засіб трансляції між сутностями зовнішньої і внутрішньої для об'єкта середовища. Інтерфейс - це форма непрямої взаємодії. Пов'язаний з концепцією кібернетики "чорний ящик".

Протокол - набір угод інтерфейсу логічного рівня, які визначають обмін даними між різними програмами. Ці угоди задають однаковий спосіб передачі

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

повідомлень і обробки помилок при взаємодії програмного забезпечення рознесеною в просторі апаратури, з'єднаної тим чи іншим інтерфейсом. Це набір правил взаємодії між об'єктами. Ці правила визначають синтаксис, семантику і синхронізацію взаємодії. Протокол може існувати у формі конвенції (неформального) або стандарту (формального набору правил).

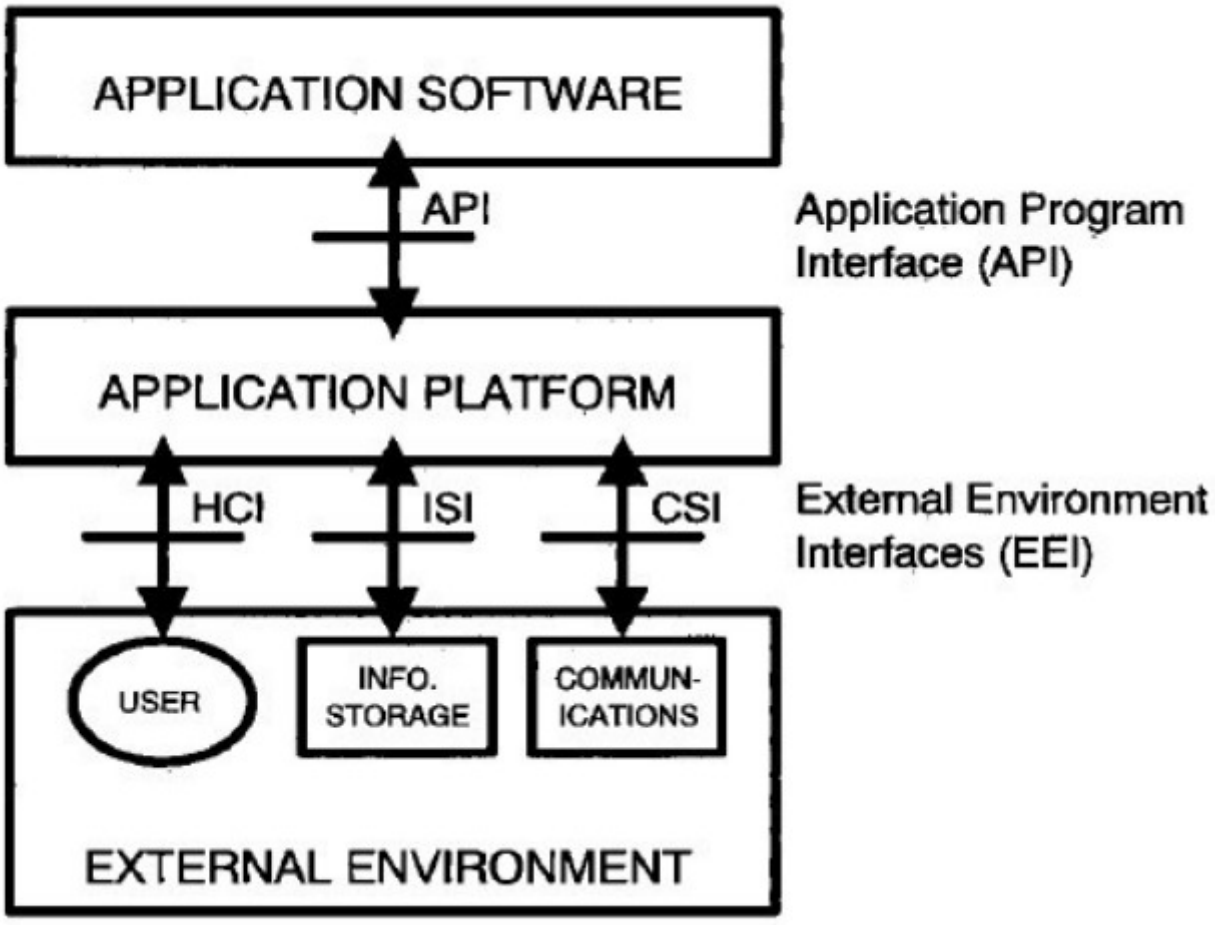

Рис. 0.6. Модель POSIX

Завдання POSIX:

- Сприяти полегшенню перенесення коду прикладних програм на інші платформи
- Сприяти визначенню та уніфікації інтерфейсів заздалегідь при проектуванні, а не в процесі їх реалізації
- Зберегти по можливості і враховувати всі головні, створені раніше і використовувані прикладні програми
- Визначати необхідний мінімум інтерфейсів прикладних програм, для прискорення створення, схвалення і затвердження документів

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

- Розвивати стандарти в напрямку забезпечення комунікаційних мереж, розподіленої обробки даних і захисту інформації
- Рекомендувати обмеження використання бінарного (об'єктного) коду для додатків в простих системах

Принципи відкритої системи:

- Переносимість додатків (на рівнях: коду і програми), даних і персоналу
- Інтероперабельність
- Розширюваність
- Масштабованість

Фольклор

Системне програмування - одна з найстаріших областей комп'ютерної інженерії. Тому вона включає в себе не тільки формальні знання, такі як алгоритми, стандарти і результати досліджень, але також і накопичені неформальні, культурні та соціальні знання.

Приклади:

- Закон Постела (принцип здоровості) - відноситься до організації взаємодії між системами. Один з приципів, що лежать в основі Інтернету

  > Будьте консервативні в тому, що відправляєте, і ліберальні в тому, що приймаєте.

- Закон конвертації програм Завінського - описує розвиток будь-якої складної програмної системи

  > Кожна програма намагається розширитися до тих пір, поки не зможе читати пошту. Ті програми, які не можуть цього зробити, замінюються тими, які можуть.

- Десяте правило програмування Грінспена - описує розвиток будь-якої складної програмної системи, заснованої на статичних мовах

  > Будь-яка достатньо складна програма на C або Fortran містить

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

реалізацію половини Common Lisp, яка є ad hoc, наформально-специфікованої, повної багів і повільною.

Література

- A Crash Course in Modern Hardware
- The C Programming Language або Learn C The Hard Way
- The Unix Programming Environment
- Tanenbaum–Torvalds debate

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Взаємодія з апаратною частиною

Огляд роботи ОС з апаратною частиною

ОС реалізується поверх апаратної архітектури, яка визначає спосіб взаємодії і набір обмежень. Взаємодія відбувається безпосередньо — через видачу інструкцій процесору,— і опосередковано — через обробку переривань пристроїв. З моменту початку завантаження ОС покажчик на поточну інструкцію процесора (регістр IP) встановлюється в початкову інструкцію виконуваного коду ОС, після чого управління роботою процесора переходить до ОС.

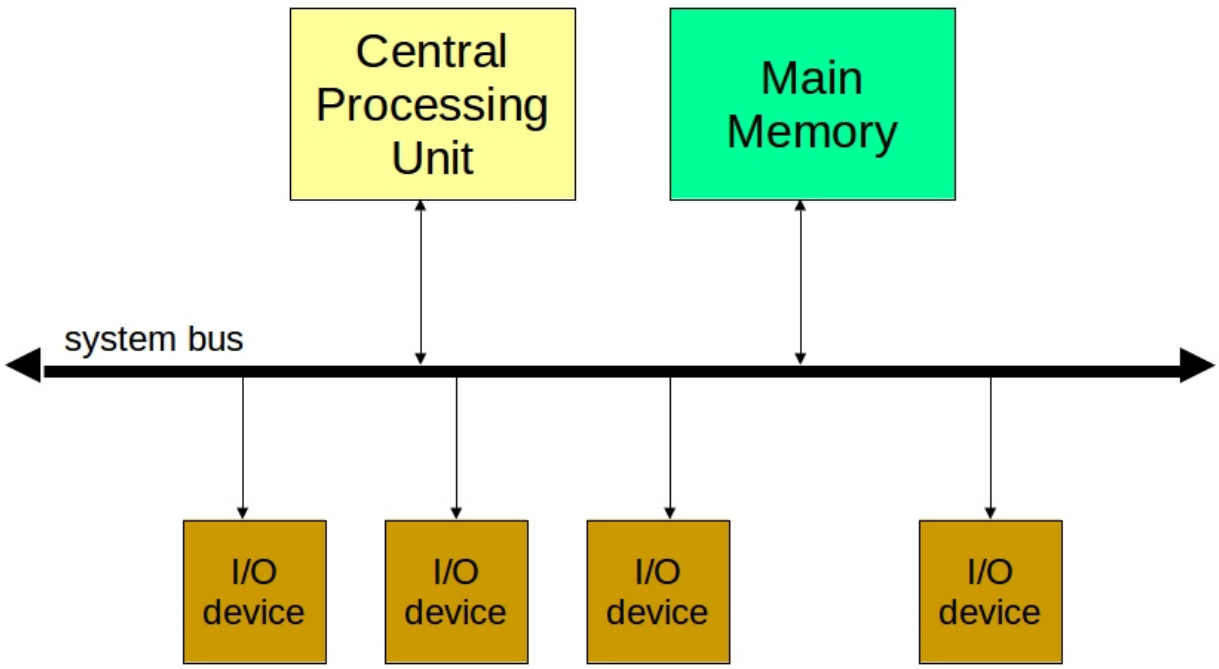

Рис. 1.1. Загальний вид апаратної архітектури

Взаємодія процесора з зовнішніми пристроями (також зване введенням-виведенням) можливо тільки через адресуєму пам'ять. Існують 2 схеми передачі даних між пам'яттю і пристроями: PIO (Programmed IO - введення-виведення через процесор) і DMA (Direct Memory Access - прямий доступ до пам'яті). В основному використовується другий варіант, який покладається на окремий контролер, що дозволяє розвантажити процесор від управління введенням-виводом і таким чином прискорити роботу системи в цілому.

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

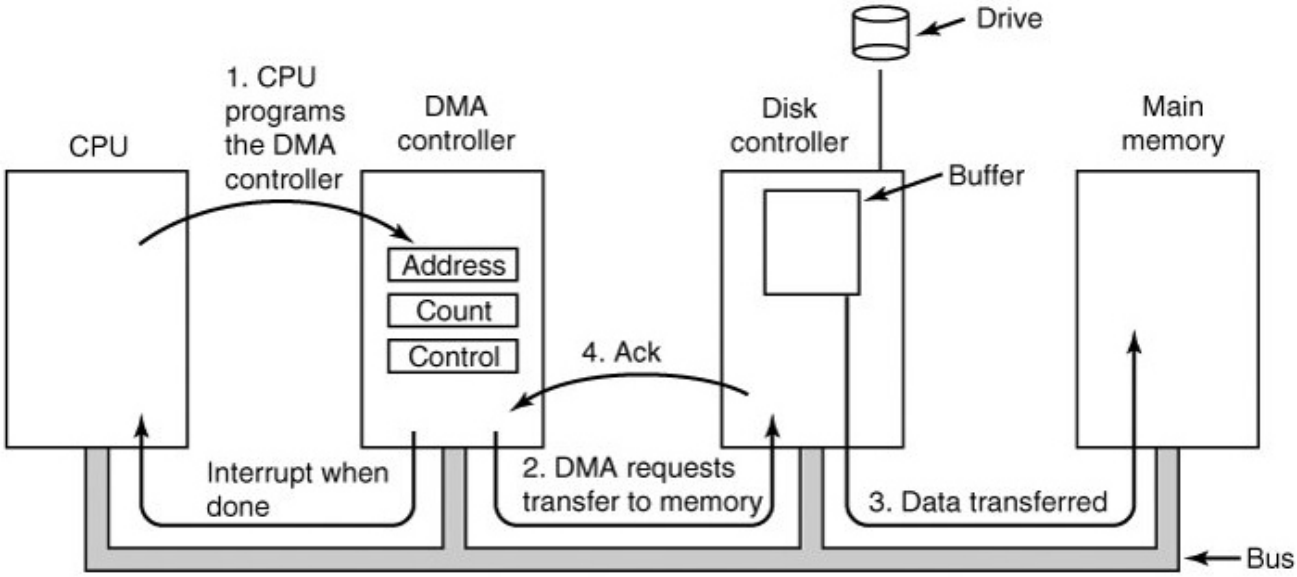

Рис. 1.2. Схема введення-виведення через DMA

Драйвери пристроїв

Драйвер пристрою - це комп'ютерна програма, яка реалізує механізм управління пристроєм і дозволяє програмам більш високого рівня взаємодіяти з конкретним пристроєм, не знаючи його команд та інших параметрів функціонування. Драйвер здійснює свої функції за допомогою команд контролера пристрою і обробки переривань, що приходять від нього. Як правило, драйвер реалізується як частина (модуль) ядра ОС, так як:

- обробка переривань драйвером вимагає задіювання функцій ОС
- справедлива і ефективна утилізація пристрої вимагає координації ОС
- посилка невірних команд або їх послідовностей, а також недотримання інших умов роботи з пристроєм може вивести його з ладу

Драйвери пристроїв деляться на 3 основні класи:

- Символьні - працюють з пристроями, що дозволяють передавати дані по 1 символу (байту): як правило, різні консолі, модеми і т.п.
- Блокові - працюють з пристроями, що дозволяють здійснювати буферизоване введення-виведення: наприклад, різними дисковими накопичувачами
- Мережеві

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

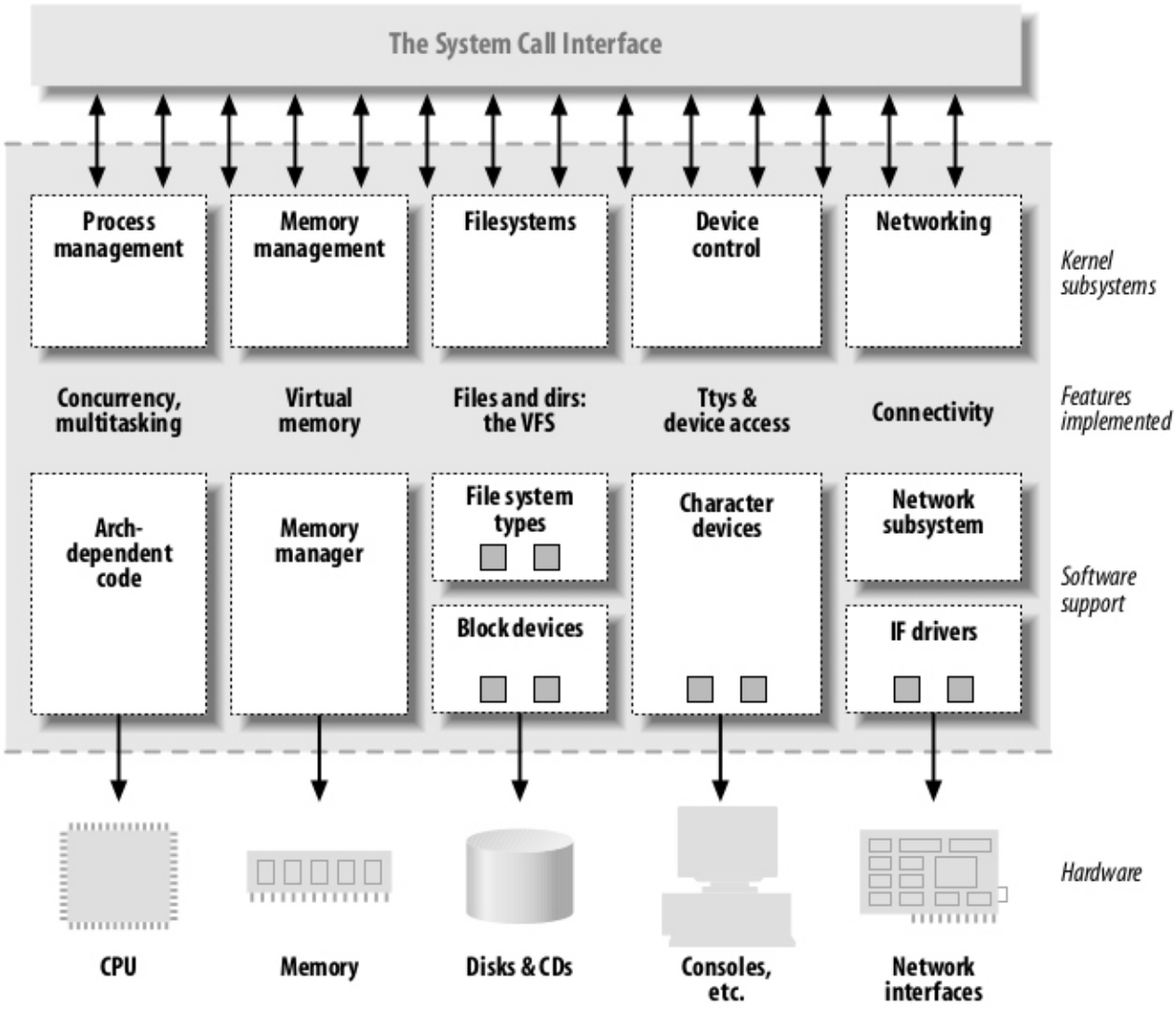

Рис. 1.3. Класи драйверів

Драйвери окремих пристроїв об'єднуються ОС в класи, які надають однаковий абстрактний інтерфейс програмам більш високого рівня. Загалом цей інтерфейс називається Рівнем абстракції обладнання (Hardware absctraction layer, HAL).

Час в комп'ютері

Для роботи ОС використовує апаратний таймер, який працює на заданій тактовій частоті (на даний момент, як правило 100 Гц). У Linux 1 цикл такого таймера називається `jiffies`. При цьому сучасні процесори працюють на тактовій частоті порядку ГГц, взаємодія з пам'яттю відбувається з частотою порядку десятків МГц, з диском і мережею - порядку сотень КГц. Загалом, це створює певну ієрархію операцій в компьютері за порядком часу, який необхідний для їх виконання.

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Переривання

Апаратні переривання

Всі операції вводу-виводу вимагають тривалого часу на своє виконання, тому виконуються в асинхронному режимі. Тобто після виконання інструкції, що викликає введення-виведення, процесор не чекає його завершення, а перемикається на виконання інших інструкцій. Коли введення-виведення завершується, пристрій сигналізує про це за допомогою переривання. Таке переривання називається апаратним (жорстким) або ж асинхронним.

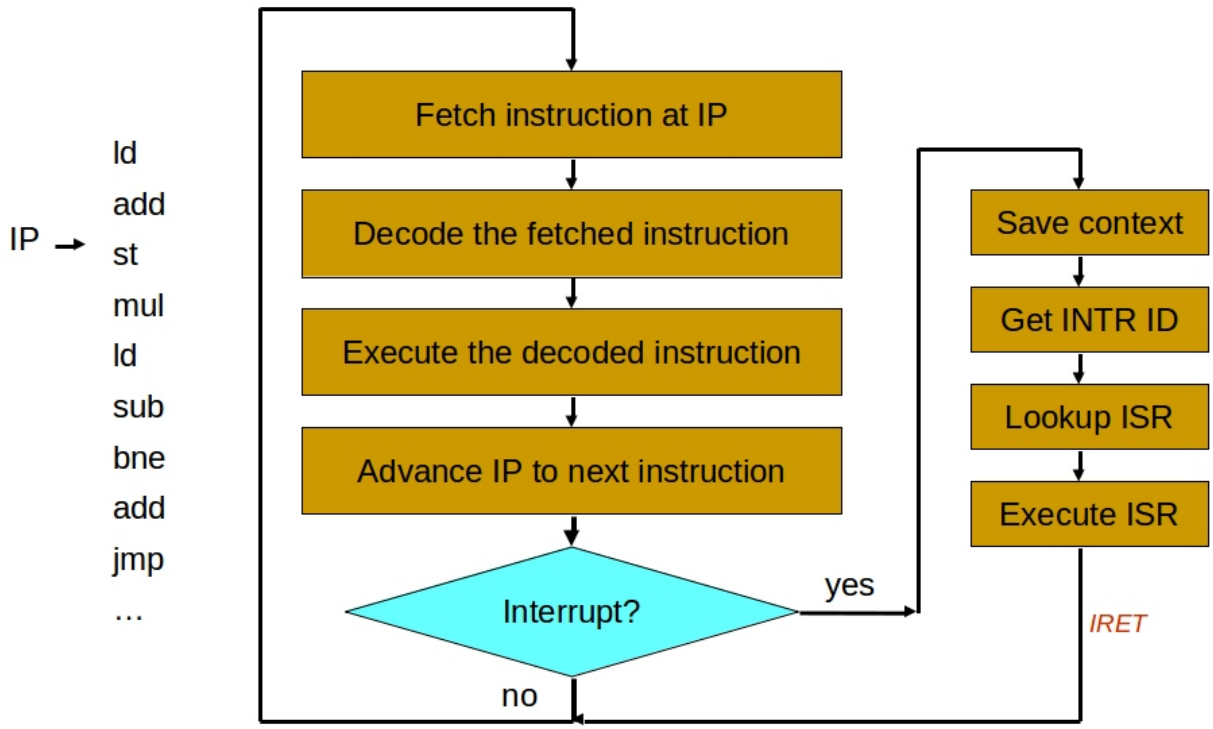

Рис. 1.4. Алгоритм роботи процесора

Загалом, **переривання** — це сигнал, що повідомляє процесору про настання якої-небудь події. При цьому виконання поточної послідовності команд припиняється і керування передається обробнику переривання, який реагує на подію і обслуговує її, після чого повертає керування в перерваний код.

PIC (Programmable Interrupt Controller — програмований контролер переривань) - це спеціальний пристрій, який забезпечує сигналізацію про перериваннях процесору.

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

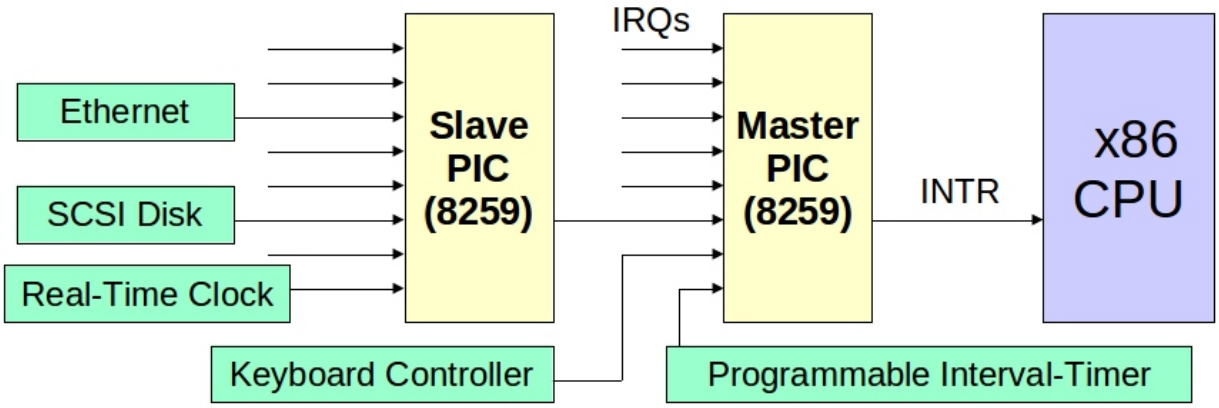

Рис. 1.5. Приклад реализації PIC

Програмні переривання

Крім асинхронних переривань процесори підтримують також синхронні переривання двох типів:

- Виключення (Exceptions): помилки (fault) — припускають можливість відновлення, пастки (trap) — сигнали, які надсилаються після виконання команди і використовуються для зупинки роботи процесора (наприклад, при відлагодженні), і збої (abort) — не припускають відновлення
- Програмовані переривання

В архітектурі x86 передбачено 256 синхронних переривань, з яких перші 32 - це зарезервовані виключення, інші можуть бути довільно призначені ОС. Приклади стандартних виключень в архітектурі x86 з їх номерами:

```
0: divide-overflow fault
6: Undefined Opcode
7: Coprocessor Not Available
11: Segment-Not-Present fault
12: Stack fault
13: General Protection Exception
14: Page-Fault Exception
```


* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Схема обробки переривань

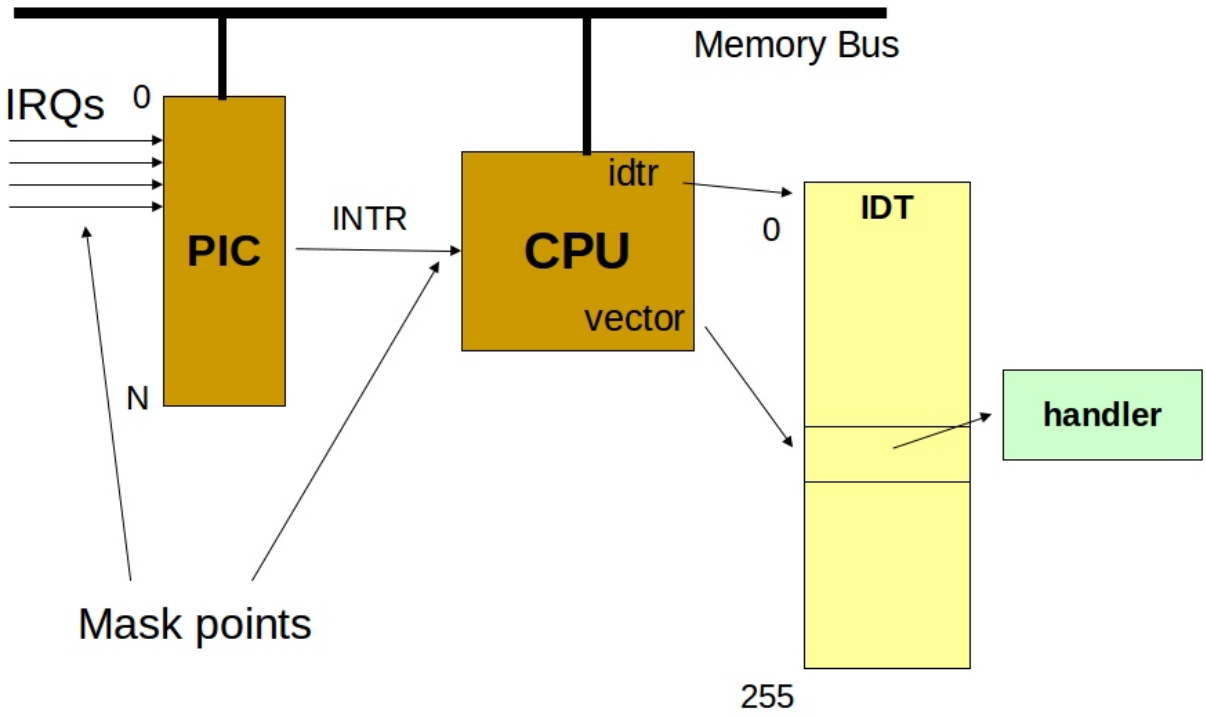

Рис. 1.6. Загальна схема системи обробки переривань

Кожне переривання має унікальний номер, який використовується як зсув в таблиці обробників переривань. Ця таблиця зберігається в пам'яті комп'ютера і на її початок вказує спеціальний регістр процесора — IDT (Interrupt Descriptor Table).

При надходженні сигналу про переривання його потрібно обробити якомога швидше, для того щоб дати можливість проводити введення-виведення інших пристроїв. Тому процесор відразу перемикається в режим обробки переривання. За номером переривання процесор знаходить процедуру-обробник в таблиці IDT і передає їй управління. Обробник переривання, як правило, розбитий на 2 частини: верхню (top half) і нижню (bottom half).

Верхня частина виконує тільки той мінімальний набір операцій, який необхідний, щоб передати управління далі. Цей набір включає:

- Підтвердження переривання (яке дозволяє прихід нових переривань)
- Точне визначення пристрою, від якого прийшло переривання
- Ініціалізація процедури обробки нижньої частини і постановка її в чергу на виконання

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Процедура нижній частині обробника виконує копіювання даних з буфера пристрою в пам'ять.

Контексти

Обробники переривань працюють в т.зв. **атомарному контексті**. Перемикання контексту — це процес збереження стану регістрів процесора в пам'ять і установки нових значень для регістрів. Можна виділити як мінімум 3 контексти:

- Атомарний, який в свою чергу часто розбитий на контекст обробки апаратних переривань і програмних. У атомарному контексті у процесора немає інформації про те, яка програма виконувалася до цього, тобто немає зв'язку з користувацьким середовищем. У атомарному контексті не можна викликати блокують операції, в тому числі `sleep`.
- Ядерний - контекст роботи функцій самого ядра ОС.
- Користувацький - контекст роботи функцій користувацької програми, з якого не можна отримати доступ до пам'яті ядра.

Домени безпеки

Рис. 1.7. Кільця процесора

Для підтримки розмежування доступу до критичних ресурсів більшість архітектур реалізують концепцію **доменів безпеки** або ж **кілець процесора** (CPU Rings). Вся пам'ять ком'ютера промаркована номером кільця безпеки, до якого вона належить. Інструкції, що знаходяться в пам'яті, що відноситься до того чи іншого кільця, в якості операндів можуть використовувати тільки адреси, які відносяться до кілець за номером не більше даного. Тобто програми в кільці 0 мають максимальні привілеї, а в найбільшому за номером кільці —

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

мінімальні.

На x86 архітектурі кілець 4: від 0 до 3. ОС з монолітним або модульним ядром (такі як Linux) завантажуються в кільце 0, а користувацькі програми — в кільце 3. Решта кілець у них не задіяні. У випадку мікроядерних архітектур деякі підсистеми ОС можуть завантажуватися в кільця 1 та/або 2.

У відповідності з концепцією доменів безпеки всі операції роботи з пристроями можуть виконуватися тільки з кільця 0, тобто вони реалізовані в коді ОС і надаються користувацьким програмам через інтерфейс системних викликів.

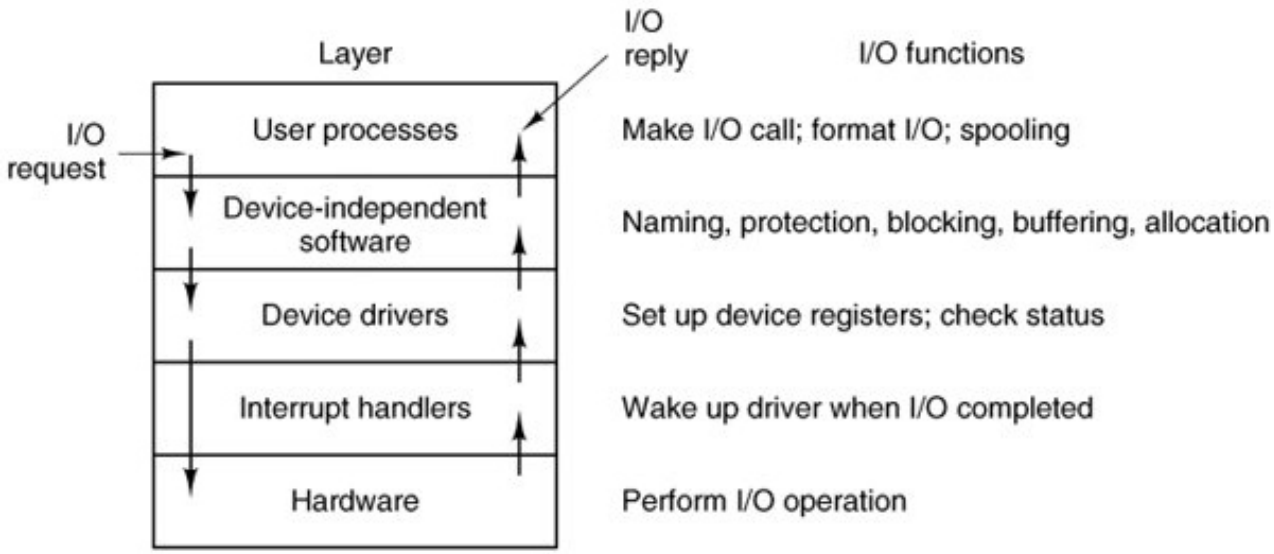

Рис. 1.8. Рівні обрабки IO-запиту

Введення-виведення є тривалою операцією за шкалою процесорного часу. Тому воно блокує викликавший його процес для того, щоб не викликати простою процесора.

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

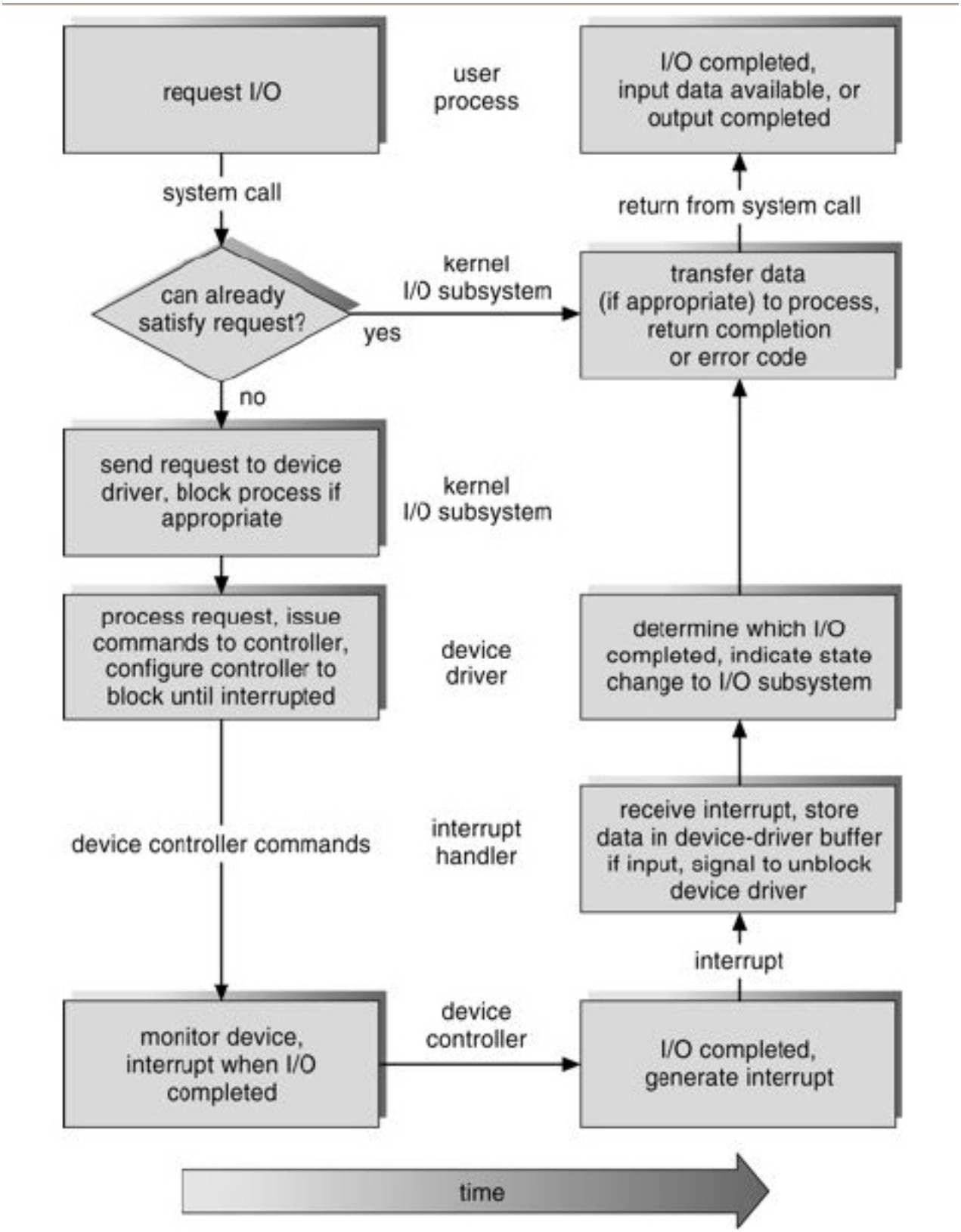

Рис. 1.9. Алгоритм ввода-виводу

Завантаження

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Завантаження (bootstrapping) — букв. витягування себе за власні шнурки — процес багатоступеневої ініціалізації ОС в пам'яті комп'ютера, який проходить через такі етапи:

1. Ініціалізація прошивки (firmware).
2. Вибір ОС-кандидата на завантаження.
3. Завантаження ядра ОС. На комп'ютерах з архітектурою x86 цей етап складається з двох підетапів:
    1. Завантаження в реальному режимі процесора.
    2. Завантаження в захищеному режимі.
4. Завантаження компонентів системи в користувацькому оточенні.

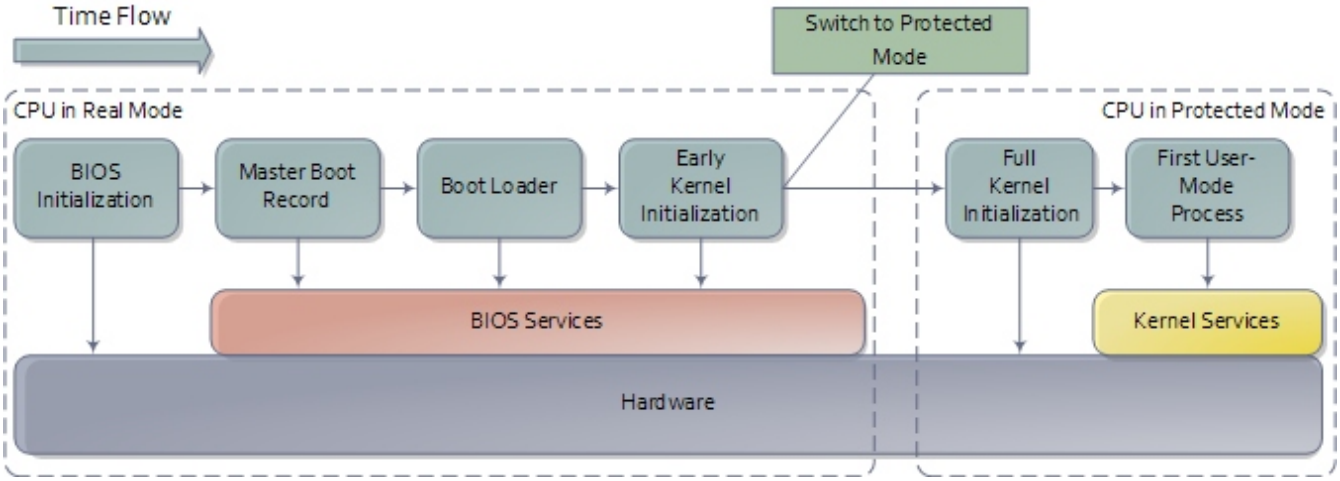

Рис. 1.10. Процесс заваниаження ОС на архітектурі x86

Прошивка

Прошивка (Firmware) - це програма, записана в ROM-пам'ять комп'ютера. На комп'ютерах загального призначення прошивка виконує функцію ініціалізації апаратної частини і передачі управління ОС. Поширеними інтерфейсами прошивок є варіанти BIOS, OpenBootProm і EFI.

BIOS - це стандартний для x86 інтерфейс прошивки. Він має безліч історичних обмежень.

Алгоритм завантаження з BIOS:

1. Power-On Self-Test (POST) — тест працездатності процесора і пам'яті після включення живлення.
2. Перевірка дисків і вибір завантажувального диска.

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

3. Завантаження Головного завантажувального запису (Master Boot Record, MBR) завантажувального диска в пам'ять і передача управління цій програмі. MBR може містити від 1 до 4 записів про розділи диска.
4. Вибір завантажувального розділу. Завантаження завантажувальної програми (т.зв. 1-й стадії завантажувача) з завантажувального запису вибраного розділу.
5. Вибір ОС для завантаження. Завантаження завантажувальної програми самої ОС (т.зв. 2-й стадії завантажувача).
6. Завантаження ядра ОС в реальному режимі роботи процесора. Більшість сучасних ОС не можуть повністю завантажитися в реальному режимі (через жорсткі обмеження по пам'яті: для ОС доступно менше 1МБ пам'яті). Тому завантажувач реального режиму завантажує в пам'ять частина коду ОС і перемикає процесор в захищений режим.
7. Остаточне завантаження ядра ОС в захищеному режимі.

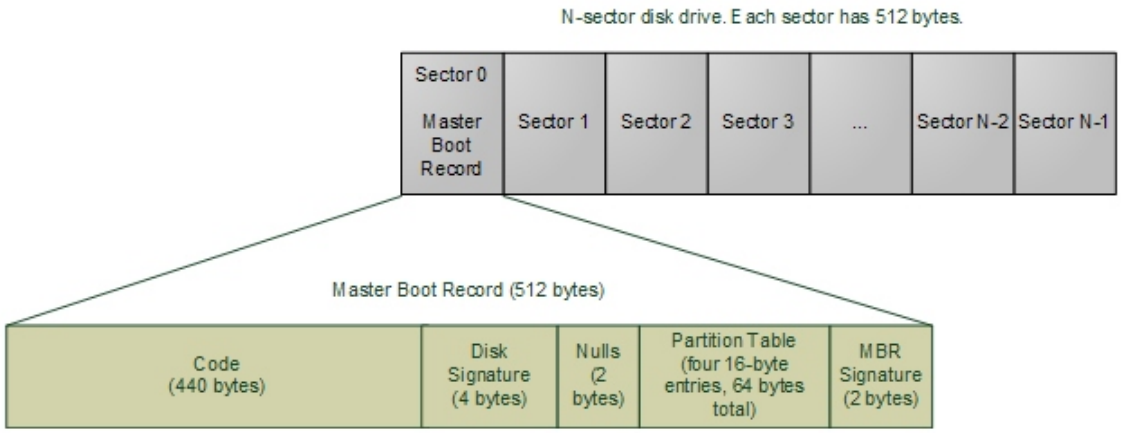

Рис. 1.11. Головний завантажувальний запис (MBR)

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

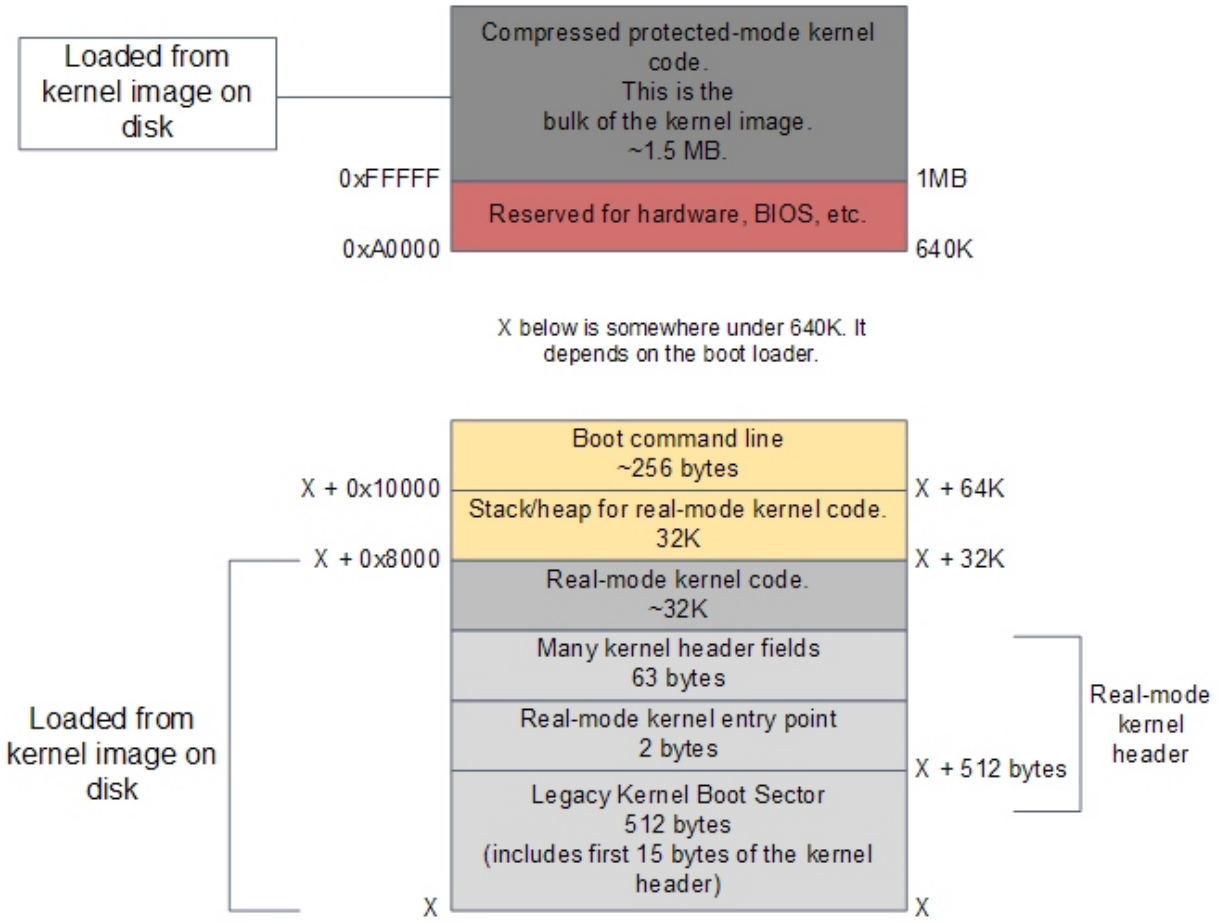

Рис. 1.12. Память комп'ютера після завантаження ОС

Процес init

Після завершення завантаження ядра ОС, запускається перша програма в користувацькому оточенні. В заснованих на Unix системах це процес номер 0, який називається `idle`. Концептуально цей процес працює так:

```
while (1) {
    ; // do nothing
}
```

Такий процес потрібен, тому що процесором повинні постійно виконуватися якісь інструкції, він не може просто чекати.

Процесом номер 1 в Unix-системах є процес `init`, який запускає всі сервіси ОС, які працюють у користувацькому оточенні. Це такі сервіси, як графічна оболонка, різні мережеві сервіси, сервіс періодичного виконання завдань (`cron`) і ін. Конфігурація цих сервісів і послідовності їх завантаження виконується за

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

допомогою shell-скриптів, що знаходяться в директоріях `/etc/init.d/`, `/etc/rc.d/` та ін.

Література

- PC Architecture
- What's New in CPUs Since the 80s and How Does It Affect Programmers?
- Interrupts and Exceptions
- Interrupt Handling Contexts
- Linux Insides series
  - Interrupts and Interrupt Handling
  - Timers and time management
  - Kernel boot process
  - Kernel initialization process
- Jiffies
- Software Illustrated series by Gustavo Duarte:
  - CPU Rings, Privilege, and Protection
  - How Computers Boot Up
  - The Kernel Boot Process
- The Linux/x86 Boot Protocol- Latency Numbers Every Programmer Should Know
- Linux Device Drivers, Third Edition
- The Linux Kernel Driver Model: The Benefits of Working Together- Writing Device Drivers in Linux
- Another Level of Indirection
- x86 DOS Boot Sector Written in C

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Бінарний інтерфейс

Бінарний інтерфейс додатків (ABI)

Бінарний інтерфейс додатків — набір формальних специфікацій і неформальних угод в рамках програмної платформи, що забезпечують виконання програмам на платформі. Бінарному інтерфейсу мають слідувати всі програми, однак вся робота по його реалізації лягає на компілятор і, як правило, прозора для прикладних програмістів.

Бінарний інтерфейс включає:

- специфікацію типів даних: розміри, формат, послідовність байт (endiannes)
- формати виконуваних файлів
- угоди про виклики
- формат і номери системних викликів
- та ін.

Асемблер

Асемблер — це низькорівнева мова, яка дозволяє безпосередньо кодувати інструкції програмної платформи (ОС або віртуальної машини).

Відмінності асемблера від мов більш високого рівня:

- відсутність єдиного стандарту, тобто у кожної архітектури свій асемблер
- програма на Асемблері досить прямо відображається в бінарний код об'єктного (виконуваного) файлу; зворотнє перетворення — з об'єктного файлу в асемблерний код називається дизасемблювання
- в асемблері немає поняття змінних, а також керуючих конструкцій: виконання програми відбувається за рахунок маніпуляції даними в регістрах і пам'яті безпосередньо, а також виклику інших інструкцій процесора
- відсутність будь-яких перевірок цілісності даних (наприклад, перевірки типів)

Синтаксис асемблера складають:

- літерали — константні значення, які представляють самі себе (числа, адреси, рядки, регістри)

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

- інструкції — мнемоніки для запису відповідних інструкцій процесора
- мітки — імена для адрес пам'яті
- директиви компілятора — інтсрукції компілятору, що описують різні аспекти створення з програми виконуваного файлу (розрядність і секції програми, експортовані символи і т.д.) — не записуються безпосередньо у виконувану програму
- макроси — конструкції для запису послідовності блоків коду, в результаті виконання яких на етапі компіляції виконується підстановка цих шматків коду замість імені макросу

Загальноприйнятими є 2 синтаксису асемблера:

- AT&T
- Intel

Найбільш поширені асемблери для архітектури x86: NASM, GAS, MASM, TASM. Частина з них є крос-платформеними, тобто працюють в різних ОС, а частина — тільки в якій-небудь одній ОС або групі ОС.

Адресація пам'яти

Асемблером підтримуються різні способи адресації пам'яті:

- безпосередня
- пряма (абсолютна)
- непряма (базова)
- автоінкрементна/автодекрементна
- регістрова
- відносна (у випадку використання сегментной організації віртуальної пам'яті)

Порядок байтів (endianness) в машинному слові визначає послідовність запису байтів: від старшого до молодшого (**big-endian**) або від молодшого до старшого (**little-endian**).

Регістри процесора

Команди асемблера дозволяють безпосередньо маніпулювати регістрами процесора. Регістр — це невеликий обсяг дуже швидкої пам'яті (як правило, розміром в 1 машинне слово), розміщеної на процесорі. Він призначений для зберігання результатів проміжних обчислень, а також деякої інформації для

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

управління роботою процесора. Так як регістри розміщені безпосередньо на процесорі, доступ до даних, що зберігаються в них, набагато швидше доступу до даних в оперативній пам'яті.

Всі регістри можна розділити на дві групи: **користувацькі** і **системні**. Користувацькі регістри використовуються при написанні "звичайних" програм. У їх число входять основні програмні регістри, а також регістри математичного співпроцесора, регістри MMX, XMM (SSE, SSE2, SSE3) і т.п. До системних регістрів належать регістри управління, регістри управління пам'яттю, регістри відлагодження, машинно-специфічні регістри MSR та інші.

Стек

(Більш правильна назва використовуваної структури даних — **стопка** або **магазин**. Однак, історично прижилося запозичена назва стек).

Стек (stack) — це частина динамічної пам'яті, яка використовується при виклику функцій для зберігання їх аргументів і локальних змінних. В архітектурі x86 стек росте вниз, тобто вершина стека має найменший адресу. Регістр `SP` (Stack Pointer) вказує на поточну вершину стека, а регістр `BP` (Base Pointer) вказує на т.зв. базу, яка використовується для розділення стека на логічні частини, що відносяться до однієї функції — **фрейми** (кадри). Крім операцій звернення до пам'яті безпосередньо, які можуть застосуються в тому числі для роботи зі стеком, додатково для нього також введені інструкції `push` і `pop`, які записують дані на вершину стека і зчитують дані з вершини, після чого видаляють. Ці операції здійснюють зміну регістру `SP`.

Як правило, програми на високорівневих мовах програмування не працюють зі стеком безпосередньо, а це робить за них компілятор, реалізуючи певні угоди про виклики функцій і способи зберігання локальних змінних. Однак стандартна функція `alloca` бібліотеки `stdlib` дозволяє програмно виділяти пам'ять на стеку.

Виклик функції високорівневої мови створює на стеку новий фрейм, який містить аргументи функції, адресу повернення з функції, покажчик на початок попереднього фрейму, а також місце під локальні змінні.

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

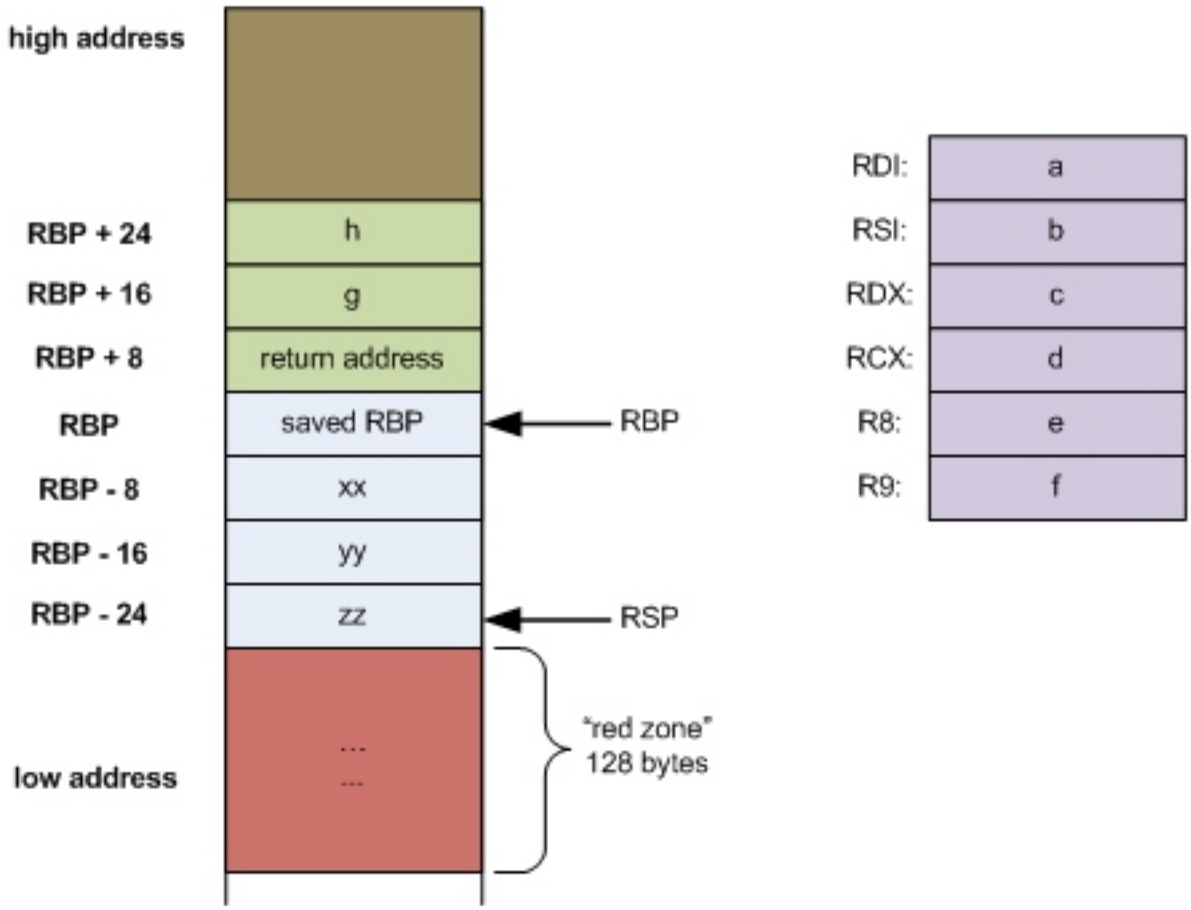

Рис. 2.1. Вид фрейму стека при виклику в рамках AMD64 ABI

На початку роботи програми в стеку виділений тільки 1 фрейм для функції `main` і її аргументів — числового значення `argc` і масиву покажчиків змінної довжини `argv`, кожен з яких записується на стек окремо, а також змінних оточення.

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

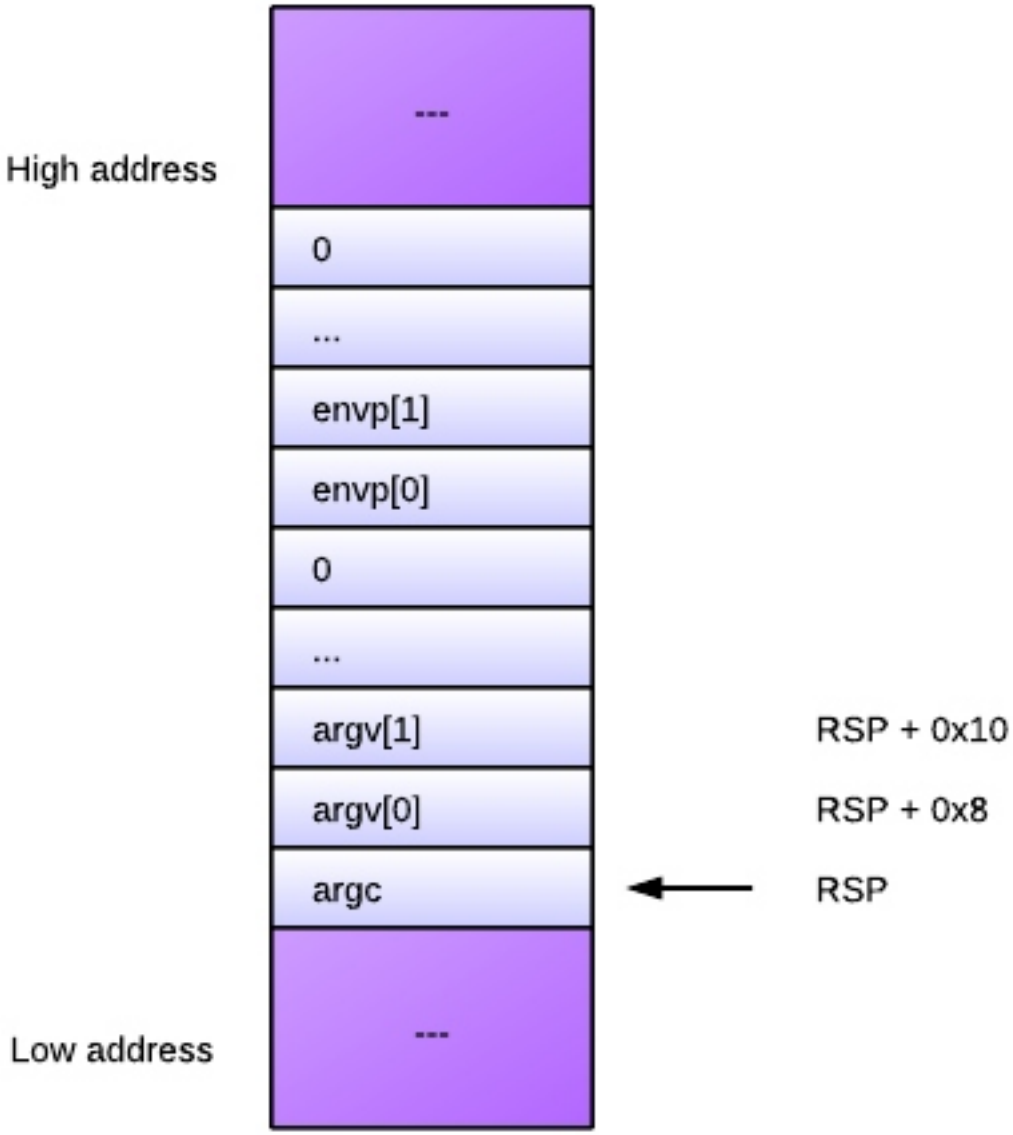

Рис. 2.2. Вид стека після виклику функції main

Угоди про виклики

Угода про виклики — це схема, у відповідності з якою виклик функції високорівневої мови програмування реалізується в виконуваному коді. Угода про виклики відповідає на питання:

- Як передаються аргументи і повертаються значення? Вони можуть передаватися або в регістрах, або через стек, або і так, і так.
- Як розподіляється робота (по маніпуляції стеком викликів) між викликаючою і викликаної стороною?

В принципі, угоди не є жорстким стандартом, і програма не зобов'язана

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

слідувати тій чи іншій угоді для власних функцій (тому програми на асемблері не завжди слідують йому), однак компілятори створюють виконувані файли у відповідності з тою або іншою угодою. Крім того, угоди про виклики для бібліотечних функцій можуть відрізнятися від угоди для системних викликів: наприклад аргументи системних викликів можуть передаватися через регістри, а бібліотечних функцій — через стек.

Поширені угоди:

- `cdecl` — загальноприйнята угода для програм на мові C в архітектурі IA32: параметри кладуться на стек справа-наліво, викликаюча функція відповідає за очищення стека після повернення з викликаної функції
- `stdcall` — стандартна угоди для Win32: параметри кладуться на стек справа-наліво, функція може використовувати регістри `EAX`, `ECX` і `EDX`, викликана функція відповідає за очищення стека перед поверненням
- `fastcall` — нестандартні угоди, в яких передача одного або більше параметрів відбувається через регістри для прискорення виклику
- `pascal` — параметри кладуться на стек зліва-направо (протилежно `cdecl`) і викликана функція відповідає за очищення стека перед поверненням
- `thiscall` — угода для C++
- `safecall` — угода для COM/OLE
- `syscall`
- `optlink`
- AMD64 ABI
- Microsoft x86 calling convention

Приклад виклику згідно до угоди `cdecl`:

Код на мові C:

```
// викликана функція
int callee(int a, int b, int c) {
   int d;
   d = a + b + c;
   ...
   return d;
}
// викликаюча функція
int caller(void) {
    int rez = callee(1, 2, 3);
    return rez + 5;
```


* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

```
}
```

Згенерований компілятором асемблерний код:

```
// в викликаючій функції
pushl   %ebp  // збереження покажчика на попередній фрейм
movl    %esp,%ebp
// запис аргументів у стек справа-наліво
pushl   $3
pushl   $2
pushl   $1
call    callee
addl    $12,%esp // очистка стека
addl    $5,%eax // результат виклика — в регістрі EAX
leave
ret
// у викликаній функції calee(1, 2, 3)
subl $4,%esp // виділення месця під змінну d
movl %eax,4(%ebp) // дістаємо аргумент a
movl %ecx,8(%ebp) // дістаємо аргумент b
addl %eax,%ecx // результат додавання залишається в %eax
movl %ecx,12(%ebp) // дістаємо аргумент c
addl %eax,%ecx // результат додавання залишається в %eax
movl (%esp),%eax // присвоюємо значення d
// ...
movl %eax,(%esp) // записуємо d в %eax
leave // эквівалент movl $ebp,$esp; pop $ebp
ret
```

Системні виклики

Загалом, **системний виклик** — це довільна функція, яка реалізується ядром ОС і доступна для виклику з програми користувача. При виконанні системного виклику відбувається переключення контексту з користувацького в ядерний. Тому на рівні команд процесора системний виклик виконується не як звичайний виклик функції (інструкція `CALL`), а за допомогою програмного переривання (в Linux це переривання номер 80) або ж за допомогою інструкції `SYSENTER` (більш сучасний варіант).

Приклад виконання системного виклика `write` за допомогою програмного переривання:

```
mov eax, 4        ; specify the sys_write function code
                  ; (from OS vector table)
```


* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

```
mov ebx, 1        ; specify file descriptor stdout(1)
mov ecx, str      ; move start address of string message
                  ; to ecx register
mov edx, str_len  ; move length of message (in bytes)
int 80h           ; tell kernel to perform the syscall
```

Приклад виконання системного виклика `write` за допомогою програмного інструкції SYSENTER:

```
push str_len
push str
push 1
push 4
push ebp
mov ebp, esp
sysenter
```

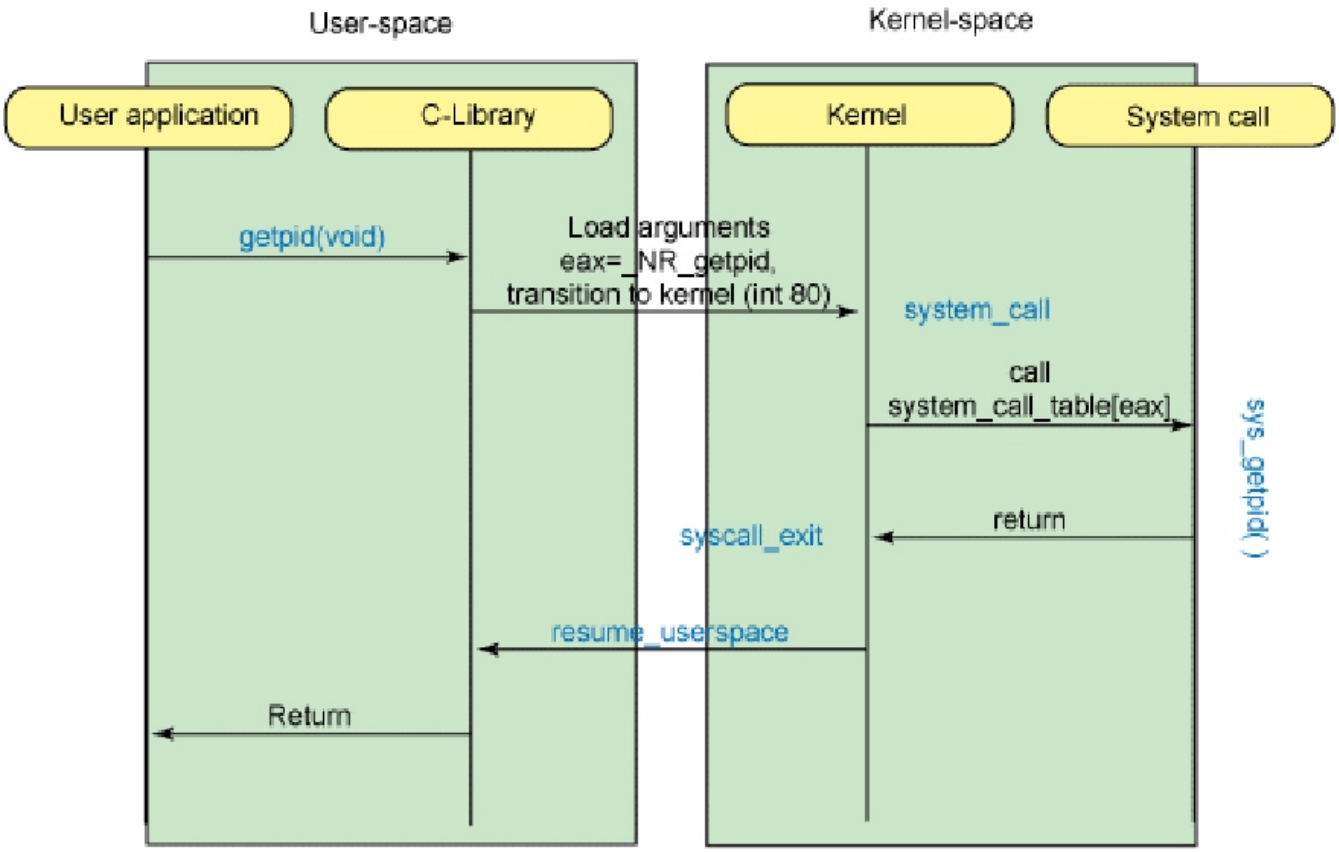

Рис. 2.3. Схема виконання системного виклику

Стандартна бібліотека `libc` реалізує свої функції понад системними викликами.

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Література

- Ассемблер в Linux для программистов C
- Ассемблеры для Linux: Сравнение GAS и NASM
- Why Registers Are Fast and RAM Is Slow
- x86 Registers
- Kernel command using Linux system calls
- The Linux Kernel: System Calls
- Sysenter Based System Call Mechanism in Linux 2.6
- x86 Instruction Encoding
- Reverse Engineering for Beginners

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Управління памяттю

Апаратне управління пам'яттю

Більшість комп'ютерів використовують велику кількість різних запам'ятовуючих пристроїв, таких як: ПЗУ, ОЗУ, жорсткі диски, магнітні носії і т.д. Всі вони являють собою види пам'яті, які доступні через різні інтерфейси. Два основних інтерфейсу — це пряма адресація процесором і файлові системи. Пряма адресація означає, що адреса комірки з даними може бути аргументом інструкцій процесора.

Режими роботи процесора x86:

- Реальний — прямий доступ до пам'яті з фізичного адресою
- Захищений — використання віртуальної пам'яті і кілець процесора для розмежування доступу до неї

Віртуальна пам'ять

Віртуальна пам'ять — це підхід до управління пам'яттю комп'ютером, який приховує фізичну пам'ять (у різних формах, таких як: оперативна пам'ять, ПЗУ або жорсткі диски) за єдиним інтерфейсом, дозволяючи створювати програми, які працюють з ними як з єдиним безперервним масивом пам'яті з довільним доступом.

Нею вирішуються наступні завдання:

- підтримка ізоляції процесів і захисту пам'яті шляхом створення свого власного віртуального адресного простору для кожного процесу
- підтримка ізоляції області ядра від коду користувацького режиму
- підтримка пам'яті тільки для читання та з забороною на виконання
- підтримка вивантаження не використовуваних ділянок пам'яті в область підкачування на диску (свопінг)
- підтримка відображених в пам'ять файлів, в тому числі завантажувальних модулів
- підтримка розділяємої між процесами пам'яті, в тому числі з копіюванням-при-запису для економії фізичних сторінок

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

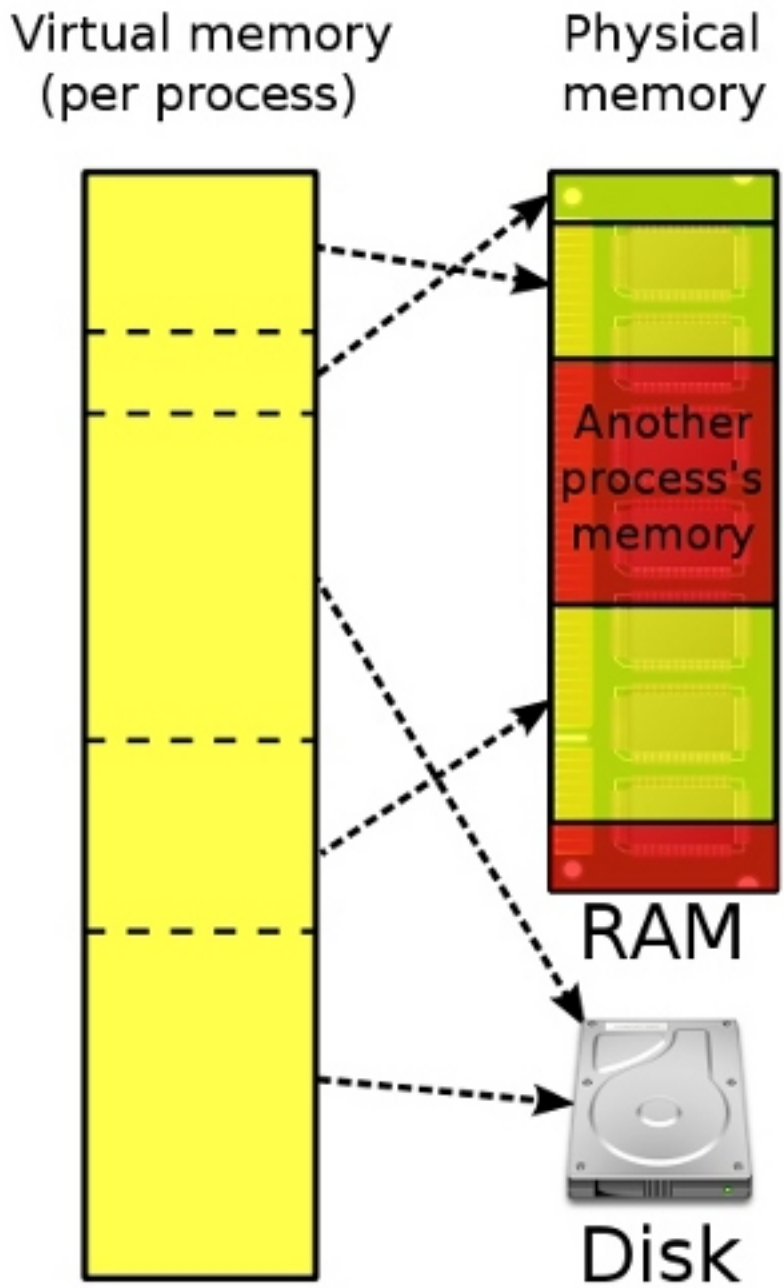

Рис. 3.1. Абстрактне представлення віртуальної пам'яті

Види адрес пам'яті:

- фізична - адреса апаратної комірки пам'яті
- логічна - віртуальна адреса, якою оперує додаток

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

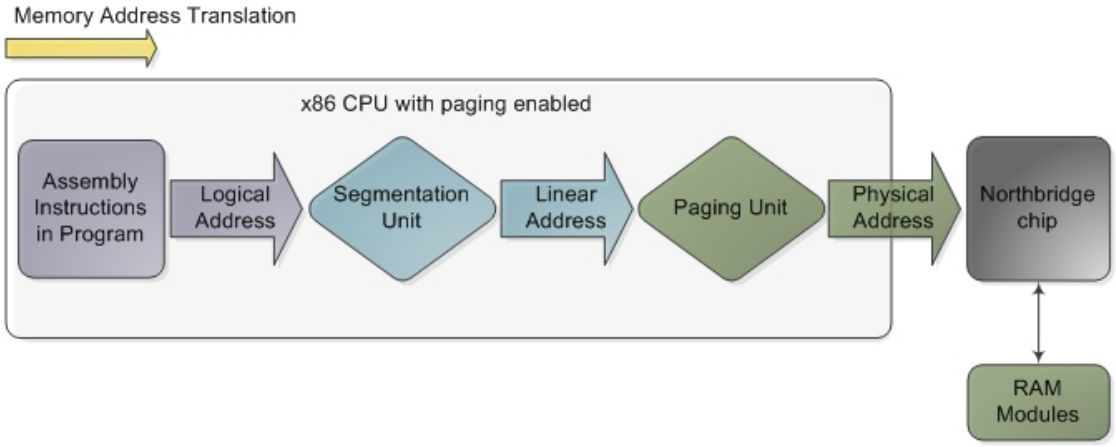

Рис. 3.2. Трансляція логічної адреси у фізичну

За рахунок наявності механізму віртуальної пам'яті компілятори прикладних програм можуть генерувати виконуваний код в рамках спрощеної абстрактної лінійної моделі пам'яті, в якій вся доступна пам'ять представляється у вигляді безперервного масиву машинних слів, що адресується з 0 до максимально можливої адреси для даної розрядності (2^N, де N — кількість біт, тобто для 32-розрядної архітектури максимальна адреса — 2^32 = #FFFFFFFF). Це означає що результуючі програми не прив'язані до конкретних параметрах запам'ятовуючих пристроїв, таких як їх обсяг, режим адресації і т.д.

Крім того, цей додатковий рівень дозволяє через той же самий інтерфейс звернення до даних за адресою в пам'яті реалізувати інші функції, такі як звернення до даних у файлі (через механізм mmap) і т.д. Нарешті, він дозволяє забезпечити більш гнучке, ефективне і безпечне управління пам'яттю комп'ютера, ніж при використанні фізичної пам'яті безпосередньо.

На апаратному рівні віртуальна пам'ять, як правило, підтримується спеціальним пристроєм — Модулем управління памяттю.

Сторінкова організація пам'яті

Сторінкова пам'ять — спосіб організації віртуальної пам'яті, при якому одиницею відображення віртуальних адрес на фізичні є регіон постійного розміру — сторінка.

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

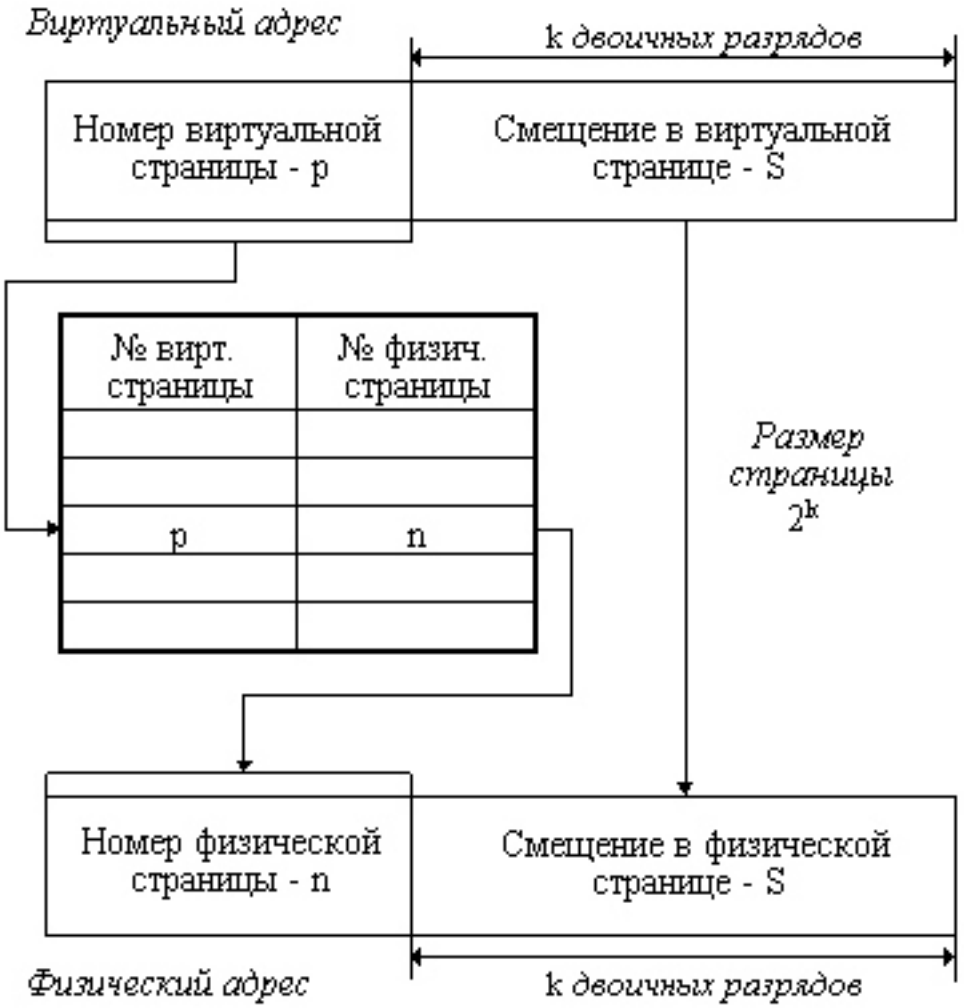

Рис. 3.3. Трансляція адреси в сторінкової моделі

При використанні сторінкової моделі вся віртуальна пам'ять ділиться на N сторінок таким чином, що частина віртуального адреси інтерпретується як номер сторінки, а частина — як зміщення всередині сторінки. Вся фізична пам'ять також поділяється на блоки такого ж розміру — **фрейми**. Таким чином в один фрейм може бути завантажена одна сторінка. **Свопінг** — це вивантаження сторінки з пам'яті на диск (або інший носій більшого обсягу), який використовується тоді, коли всі фрейми зайняті. При цьому під свопінг потрапляють сторінки пам'яті неактивних на даний момент процесів.

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

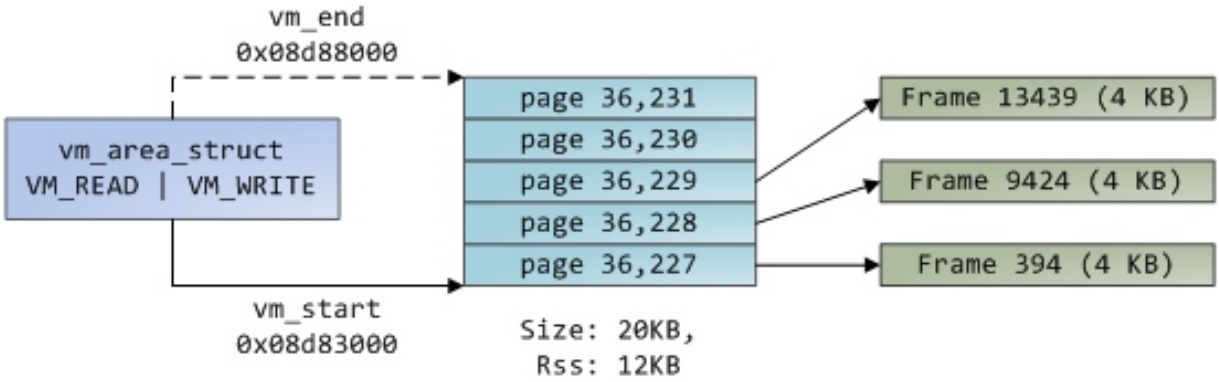

Рис. 3.4. Пам'ять процесу в сторінкової моделі

Таблиця відповідності фреймів і сторінок називається таблицею сторінок. Вона одна для всієї системи. Запис у таблиці сторінок містить службову інформацію, таку як: індикатори доступу тільки на читання або на читання/запис, чи знаходиться сторінка в пам'яті, чи проводився в неї запис і т.д. Сторінка може знаходиться в трьох станах: завантажена в пам'ять, вивантажена в своп, ще не завантажена в пам'ять (при початковому виділенні сторінки вона не завжди відразу розміщується в пам'яті).

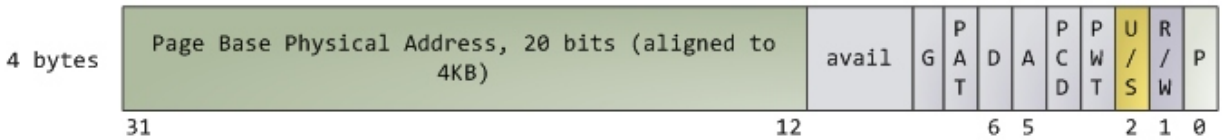

Рис. 3.5. Запис в таблиці сторінок

Розмір сторінки і кількість сторінок залежить від того, яка частина адреси виділяється на номер сторінки, а яка на зміщення. Приміром, якщо в 32-розрядній системі розбити адресу на дві рівні половини, то кількість сторінок буде становити 2^16, тобто 65536, і розмір сторінки в байтах буде таким же, тобто 64 КБ. Якщо зменшити кількість сторінок до 2^12, то в системі буде 4096 сторінки по 1МБ, а якщо збільшити до 2^20, то 1 мільйон сторінок за 4КБ. Чим більше в системі сторінок, тим більше займає пам'яті таблиця сторінок, відповідно робота процесора з нею сповільнюється. А оскільки кожне звернення до пам'яті вимагає звернення до таблиці сторінок для трансляції віртуального адреси, таке уповільнення дуже небажане. З іншого боку, чим менше сторінок і, відповідно, чим вони більші за обсягом — тим більше втрати пам'яті, викликані внутрішньою фрагментацією сторінок, оскільки сторінка є одиницею виділення пам'яті. У цьому полягає дилема оптимізації сторінкової пам'яті. Вона особливо актуальна при переході до 64-розрядних архітектур.

Для оптимізації сторінкової пам'яті використовуються наступні підходи:

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

- спеціальний кеш — TLB (translation lookaside buffer) — в якому зберігається дуже невелике число (поярдка 64) найбільш часто використовуваних адрес сторінок (основні сторінки, до яких постійно звертається ОС)
- багаторівнева (2, 3 рівня) таблиця сторінок — в цьому випадку віртуальна адреса розбивається не на 2, а на 3 (4, ...) частини. Остання частина залишається зміщенням всередині сторінки, а кожна з решти задає номер сторінки в таблиці сторінок 1-го, 2-го і т.д. рівнів. У цій схемі для трансляції адрес потрібно виконати не 1 звернення до таблиці сторінок, а 2 і більше. З іншого боку, це дозволяє свопити таблицю сторінок 2-го і т.д. рівнів, і довантажувати в пам'ять лише ті таблиці, які потрібні поточному процесу в поточний момент часу або ж навіть кешувати їх. А кожна з таблиць окремого рівня має суттєво менший розмір, ніж мала б одна таблиця, якби рівень був один

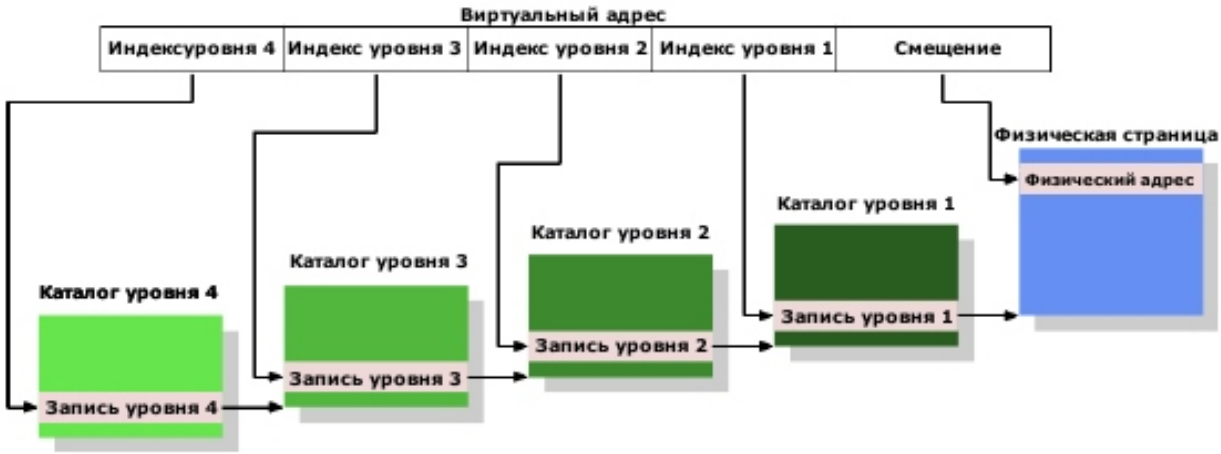

Рис. 3.6. Багаторівнева система сторінок

- інвертована таблиця сторінок — в ній стільки записів, скільки в системі фреймів, а не сторінок, і індексом є номер фрейму: а число фреймів в 64- і більше розрядних архітектурах істотно менше теоретично можливого числа сторінок. Проблема такого підходу — довгий пошук віртуального адреси. Вона вирішується за допомогою таких механізмів: хеш-таблиць або кластерних таблиць сторінок

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Сегментна організація пам'яті

Сегментна організація віртуальної пам'яті реалізує наступний механізм: вся пам'ять ділитися на сегменти фіксованою або довільної довжини, кожний з яких характеризується своєю початковою адресою — **базою** або **селектором**. Віртуальна адреса в такій системі складається з двох компонент: **бази** сегмента, до якого ми хочемо звернутися, і **зміщення** всередині сегмента. Фізична адреса обчислюється за формулою:

```
addr = base + offset
```

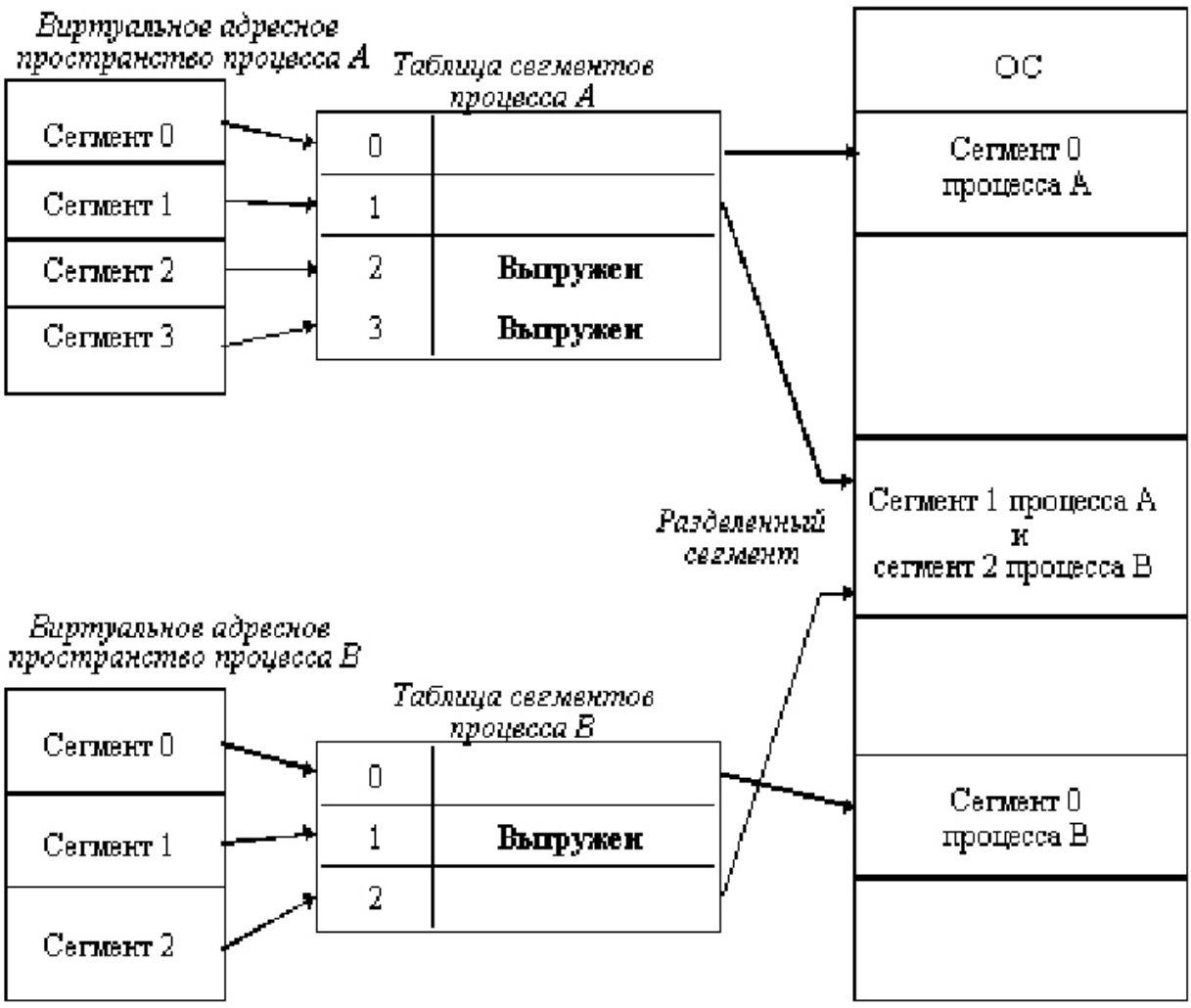

Рис. 3.7. Представлення сегментної моделі віртуальної пам'яті

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Історична модель сегментації в архітектурі x86

В архітектурі x86 сегментна модель пам'яті була вперше реалізована на 16-розрядних процесорах 8086. Використання тільки 16 розрядів для адреси давало можливість адресувати тільки 2^16 байт, тобто 64КБ пам'яті. У той же час стандартний розмір фізичної пам'яті для цих процесорів був 1МБ. Для того, щоб мати можливість працювати з усім доступним обсягом пам'яті і була використана сегментна модель. В ній у процесора було виділено 4 спеціалізованих регістра CS (сегмент коду), SS (сегмент стека), DS (сегмент даних), ES (розширений сегмент) для зберігання бази поточного сегмента (для коду, стека і даних програми — 2 регістри — відповідно).

Фізична адреса в такій системі розраховується за формулою:

```
addr = base << 4 + offset
```

Це призводило до можливості адресувати більші адреси, ніж 1МБ — т.зв. Gate A20.

Див. також: http://en.wikipedia.org/wiki/X86_memory_segmentation

Плоска модель сегментації

32-розрядний процесор 386 міг адресувати 2^32 байт пам'яті, тобто 4ГБ, що більш ніж перекривало доступні на той момент розміри фізичної пам'яті, тому початкова причина для використання сегментної організації пам'яті відпала.

Однак, крім особливого способу адресації сегментна модель також надає механізм захисту пам'яті через **кільця безпеки процесора**: для кожного сегмента в таблиці сегментів задається значення допустимого рівня привілеїв (DPL), а при зверненні до сегменту передається рівень привілеїв поточної програми (запитаний рівень привілеїв, RPL) і, якщо `RPL> DPL` доступ до пам'яті заборонений. Таким чином забезпечується захист сегментів пам'яті ядра ОС, які мають `DPL = 0`. Також в таблиці сегментів задаються інші атрибути сегментів, такі як можливість запису в пам'ять, можливість виконання коду з неї і т.д.

Таблиця сегментів кожного процесу знаходиться в пам'яті, а її початкова адреса завантажується в регістр `LDTR` процесора. У регістрі `GDTR` процесора зберігається покажчик на глобальну таблицю сегментів.

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

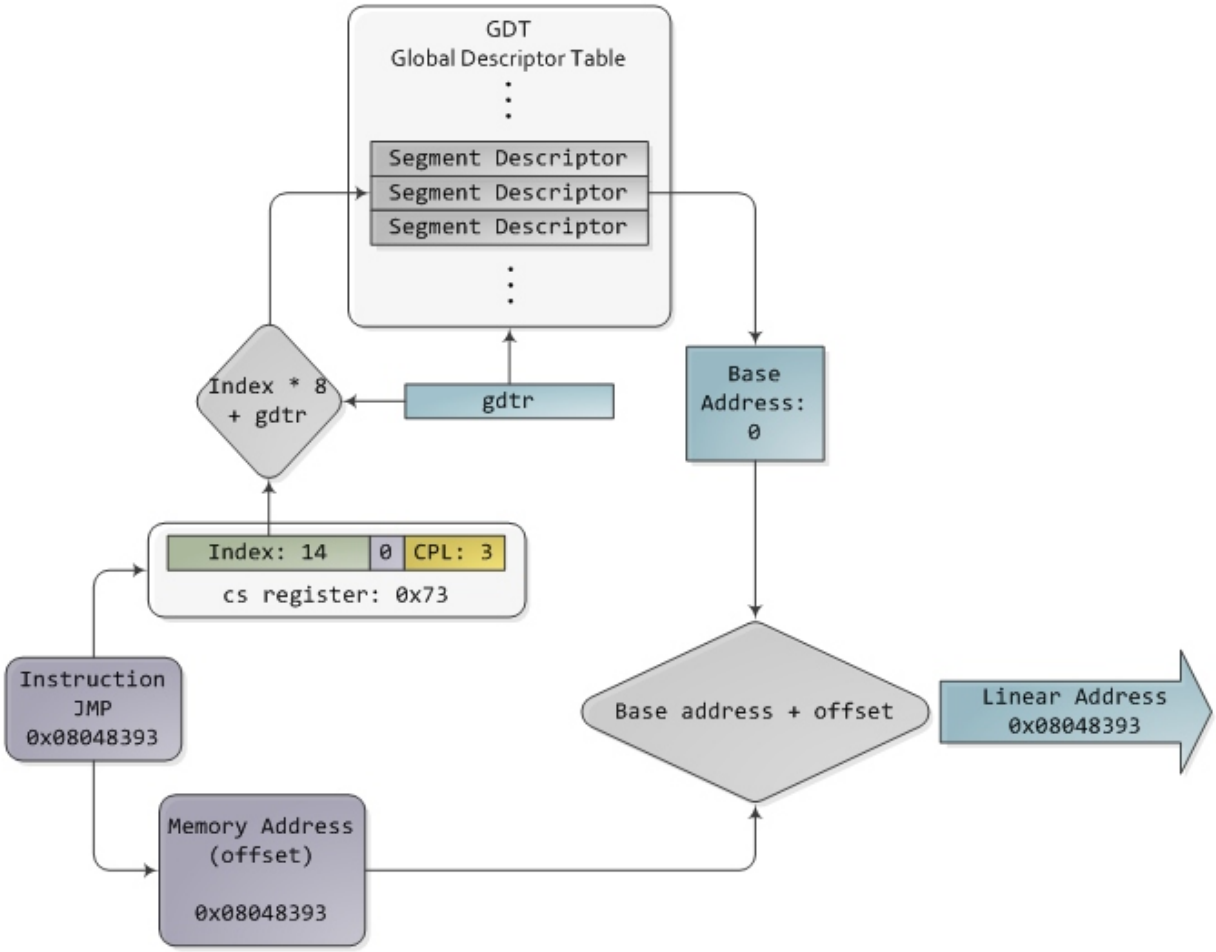

Рис. 3.8. Плоска модель сегментації

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Віртуальна пам'ять в архітектурі x86

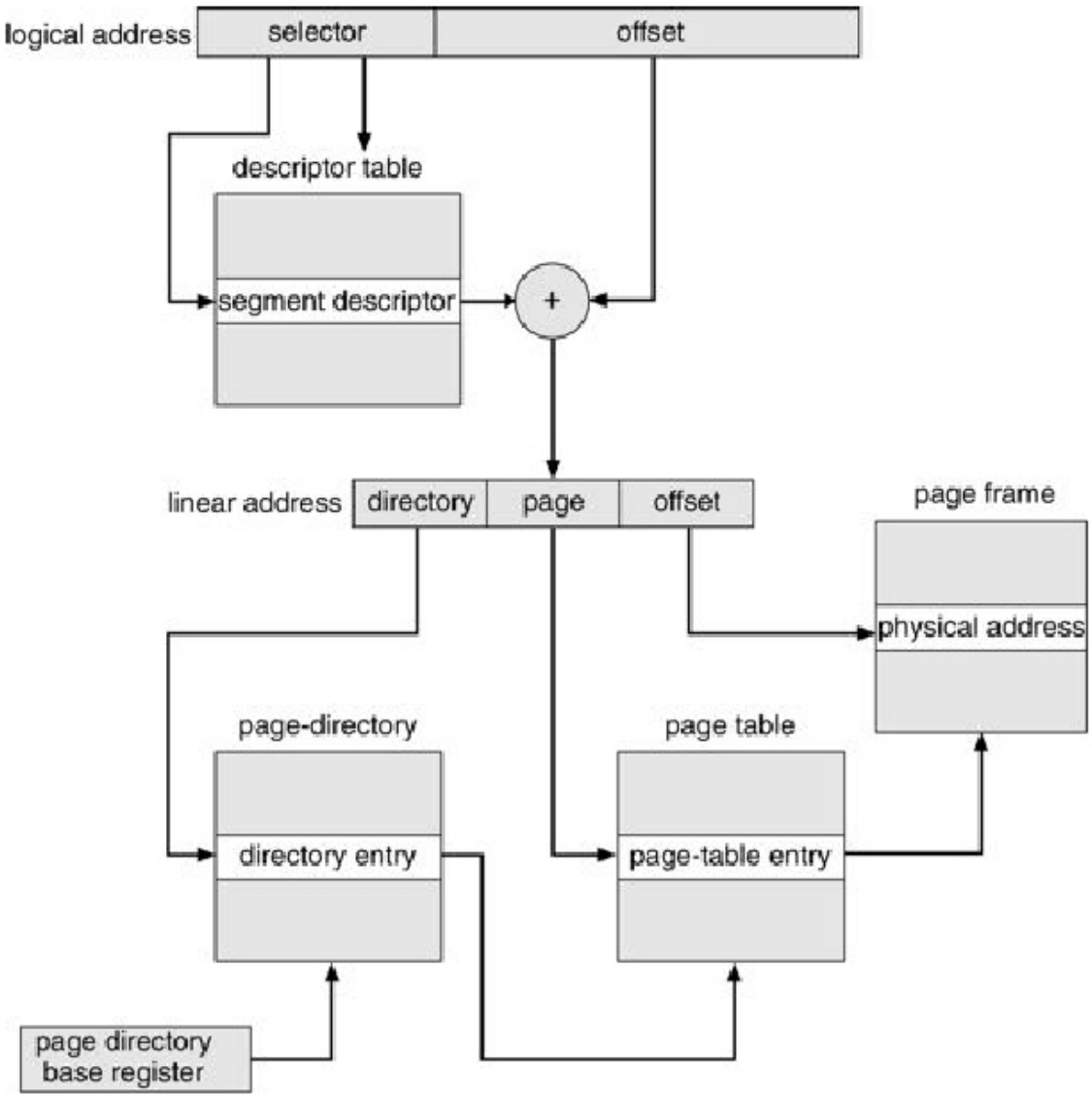

Рис. 3.9. Трансляція адреси в архітектурі x86

Системні виклики для взаємодії з підсистемою віртуальної пам'яті:

- `brk`, `sbrk` — для збільшення сегменту пам'яті, виділеного для даних програми
- `mmap`, `mremap`, `munmap` — для відображення файлу чи пристрою в пам'ять
- `mprotect` — зміна прав доступу до областей пам'яті процесу

Приклад виділення пам'яті процесу:

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

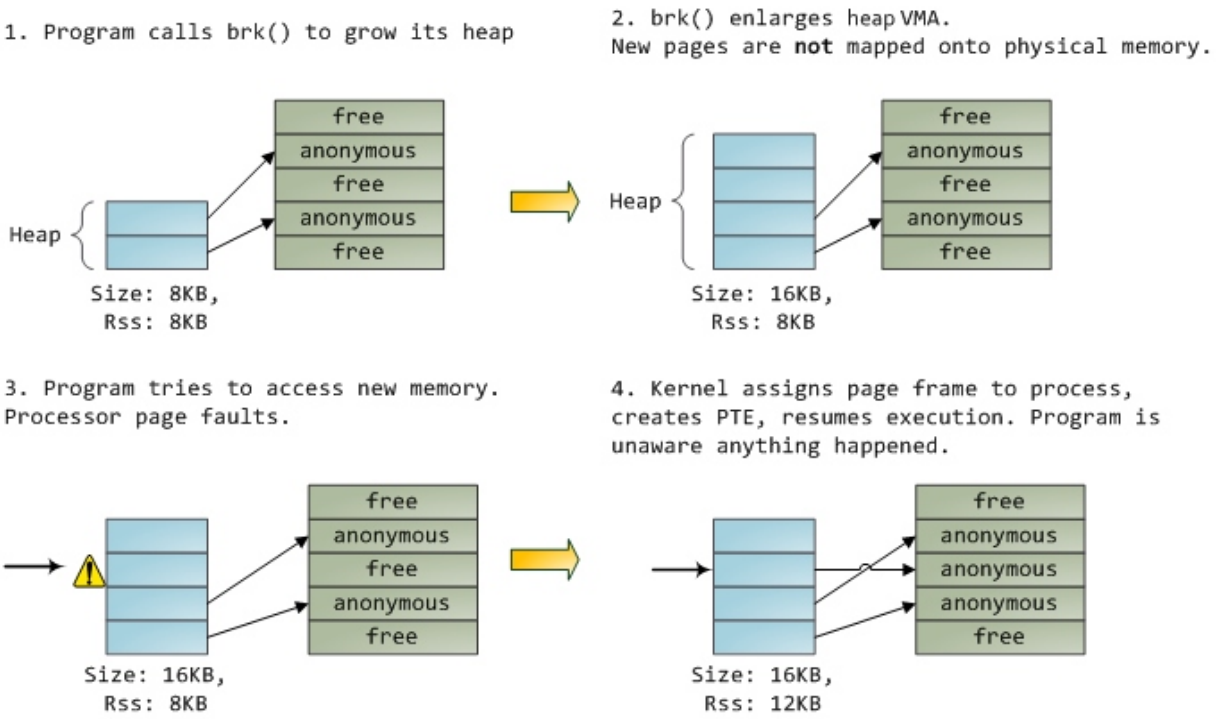

Рис. 3.10. Ледаче виділення пам'яті при виклику brk

Алгоритми виділення пам'яті

Ефективне виділення пам'яті потребує швидкого (за 1 або декілька операцій) знаходження вільної ділянки пам'яті потрібного розміру.

Способи обліку вільних ділянок:

- бітова карта (bitmap) — кожному блоку пам'яті (наприклад, сторінці) ставиться у відповідність 1 біт, який має значення зайнятий/вільний
- зв'язний список — кожному безперервному набору блоків пам'яті одного типу (зайнятий/вільний) ставиться у відповідність 1 запис в зв'язковому списку блоків, в якому вказується початок і розмір ділянки
- використання декількох зв'язних списків для ділянок різних розмірів — див. алгоритм Buddy allocation

Кешування

Кеш - це компонент комп'ютерної системи, який прозоро зберігає дані так, щоб наступні запити до них могли бути задоволені швидше. Наявність кеша означає також наявність запам'ятовуючого пристрою (набагато) більшого розміру, в якому дані зберігаються первісно. Запити на отримання даних з цього пристрою

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

прозоро проходять через кеш в тому сенсі, що якщо цих даних немає в кеші, то вони запитуються з основного пристрою і паралельно записуються в кеш. Відповідно, при подальшому зверненні дані можуть бути прочитані вже з кешу. За рахунок набагато меншого розміру кеш може бути зроблено набагато швидшим і в цьому основна мета його існування.

За принципом запису даних в кеш виділяють:

- наскрізний (write-through) — дані записуються синхронно і в кеш, і безпосередньо в запам'ятовуючий пристрій
- зі зворотним записом (write-back, write-behind) — дані записуються в кещ і іноді синхронізуються з запам'ятовуючим пристроєм

За принципом зберігання даних виділяють:

- повністю асоціативні
- множинно-асоціативні
- прямої відповідності

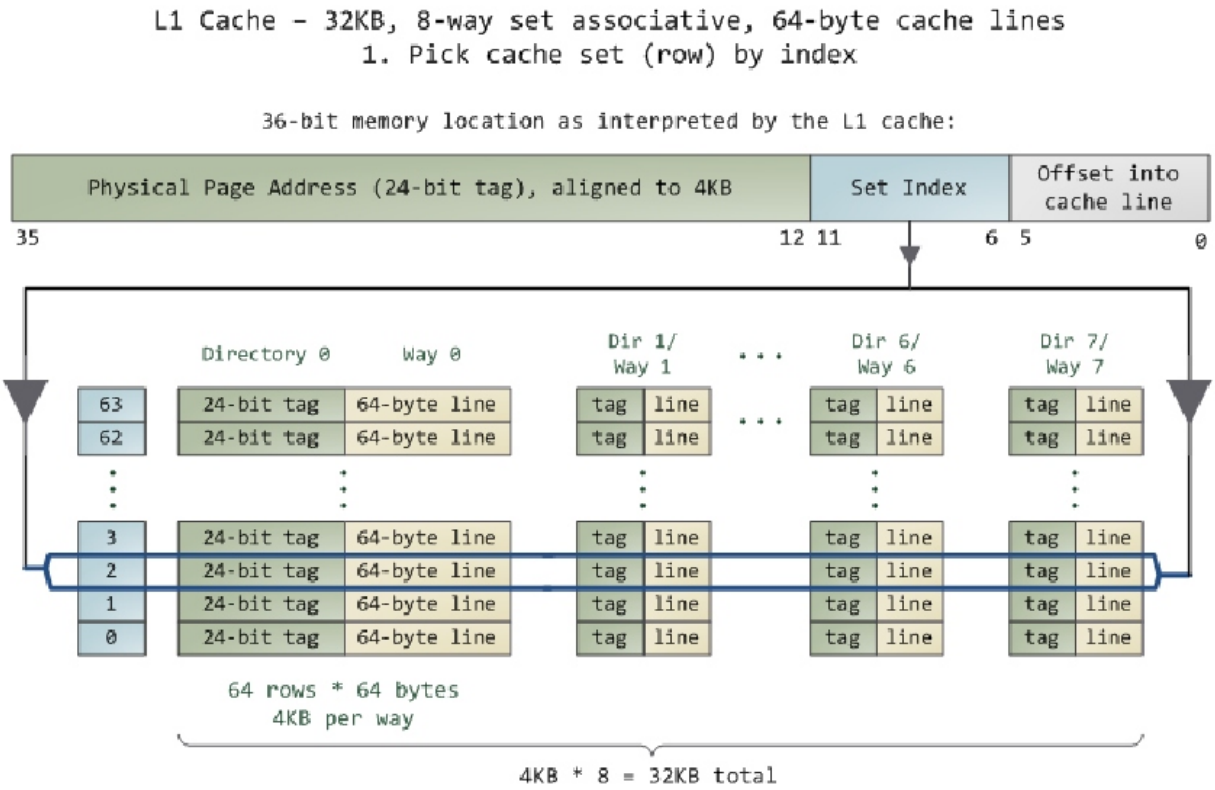

Рис. 3.11. Приклад множинно-асоціативного кеша в архітектурі x86

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

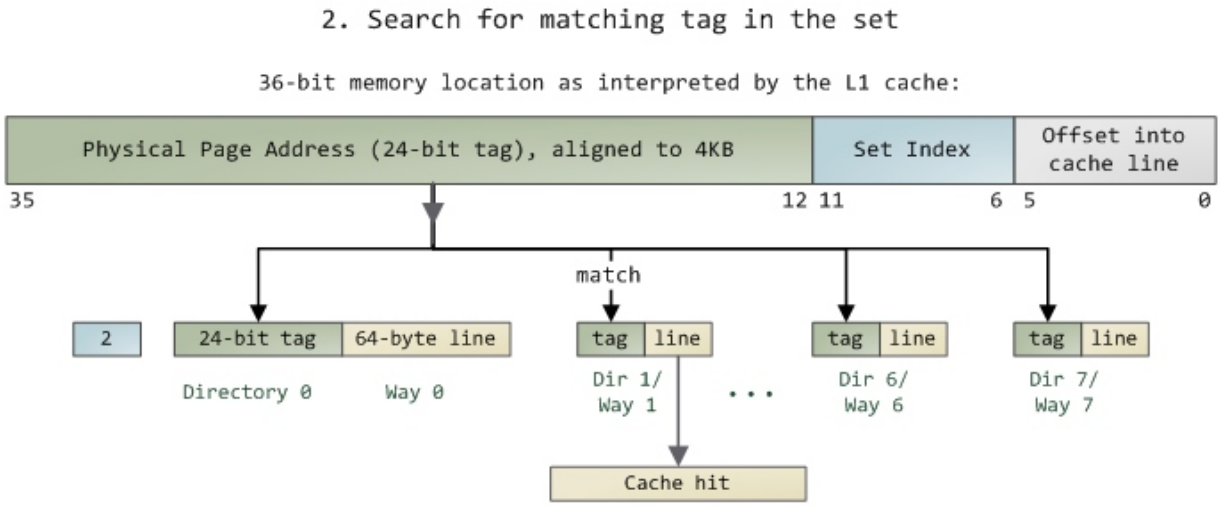

Рис. 3.12. Пошук в множинно-асоціативному кеші

Алгоритми заміщення записів в кеші

Оскільки будь-якмй кеш завжди менше запам'ятовуючого пристрою, завжди виникає необхідність для запису нових даних в кеш видаляти з нього раніше записані. Ефективне видалення даних з кеша має на увазі видалення найменш затребуваних даних. У загальному випадку не можна сказати, які дані є найменш затребуваними, тому для цього використовуються евристики. Наприклад, можна видаляти дані, до яких відбувалося найменше число звернень з моменту їх завантаження в кеш (least frequently used, **LFU**) або ж дані, до яких зверталися найменш нещодавно (least recently used, **LRU**), або ж комбінація цих двох підходів (**LRFU**).

Крім того, апаратні обмеження щодо реалізації кеша часто вимагають мінімальних витрат на облік службової інформації про комірки, якою є також і інформація про використання даних в них. Найбільш простим способом обліку звернень є встановлення 1 біта: було звернення або не було. У такому випадку для видалення з кеша може використовуватися алгоритм **годинник** (або **другого шансу**), який по колу проходить по всім коміркам, і вивантажує комірку, якщо у неї біт дорівнює 0, а якщо 1 — скидає його в 0.

Більш складним варіантом є використання апаратного лічильника для кожної комірки. Якщо цей лічильник фіксує число звернень до комірки, то це простий варіант алгоритму LFU. Він володіє наступними недоліками:

- може статися переповнення лічильника (а він, як правило, має дуже невелику розрядність) — в результаті буде втрачена вся інформація про звернення до комірки

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

- дані, до яких робилось багато звернень в минулому, будуть мати високе значення лічильника навіть якщо за останній час до них не було звернень

Для вирішення цих проблем використовується механізм **старіння**, який передбачає періодичний зсув вправо одночасно лічильників для всіх комірок. У цьому випадку їх значення будуть зменшуватися (у 2 рази), зберігаючи пропорцію між собою. Це можна вважати варіантом алгоритму LRFU.

Література

- Управление памятью
- Виртуальная память
- What Every Programmer Should Know About Memory
- The Memory Management Reference
- Software Illustrated series by Gustavo Duarte:
  - How The Kernel Manages Your Memory
  - Memory Translation and Segmentation
  - Getting Physical With Memory
  - What Your Computer Does While You Wait
  - Cache: a place for concealment and safekeeping
  - Page Cache, the Affair Between Memory and Files
- Memory Allocators 101
- Doug Lea's malloc
- How tcmalloc Works
- Visualizing Garbage Collection Algorithms
- How Bad Can 1GB Pages Be?
- How Misaligning Data Can Increase Performance 12x by Reducing Cache Misses
- Real Mode Memory Management
- Memory Testing from Userspace Programs

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

- How L1 and L2 CPU caches work
- Redis latency spikes and the Linux kernel
- Kernel-bypass Networking Illustrated

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Виконувані файли

Програма в пам'яті

Виконання програми починається з системного виклику `exec`, якому передається шлях до файлу з бінарним кодом програми. `exec` — це інтерфейс до завантажувача ОС, який завантажує секції програми в пам'ять в залежності від формату виконуваного файлу, в який скомпільована програма, а також виділяє додаткові секції динамічної пам'яті. Після завантаження пам'ять програми продовжує бути розділеною на окремі секції. Вказівники на початок/кінець і інші властивості кожної секції знаходяться в структурі `mm_struct` поточного процесу.

Для завантаження окремих сегментів в пам'ять використовується системний виклик `mmap`.

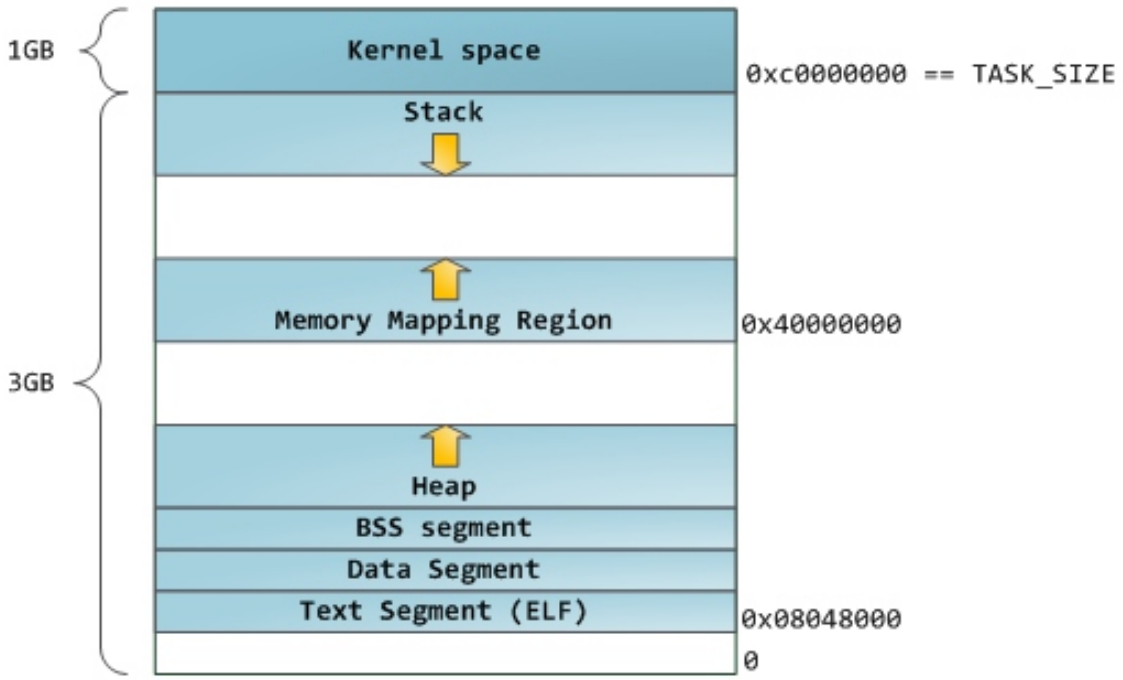

Рис. 4.1. Програма в пам'яті

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

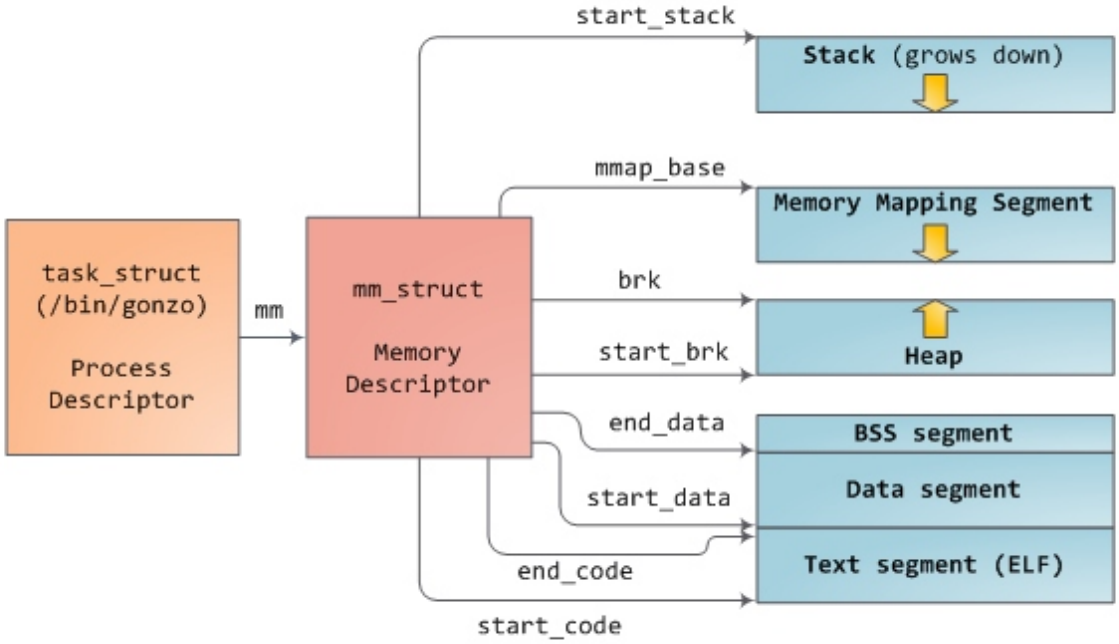

Рис. 4.2. Сегменти пам'яті процесу

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

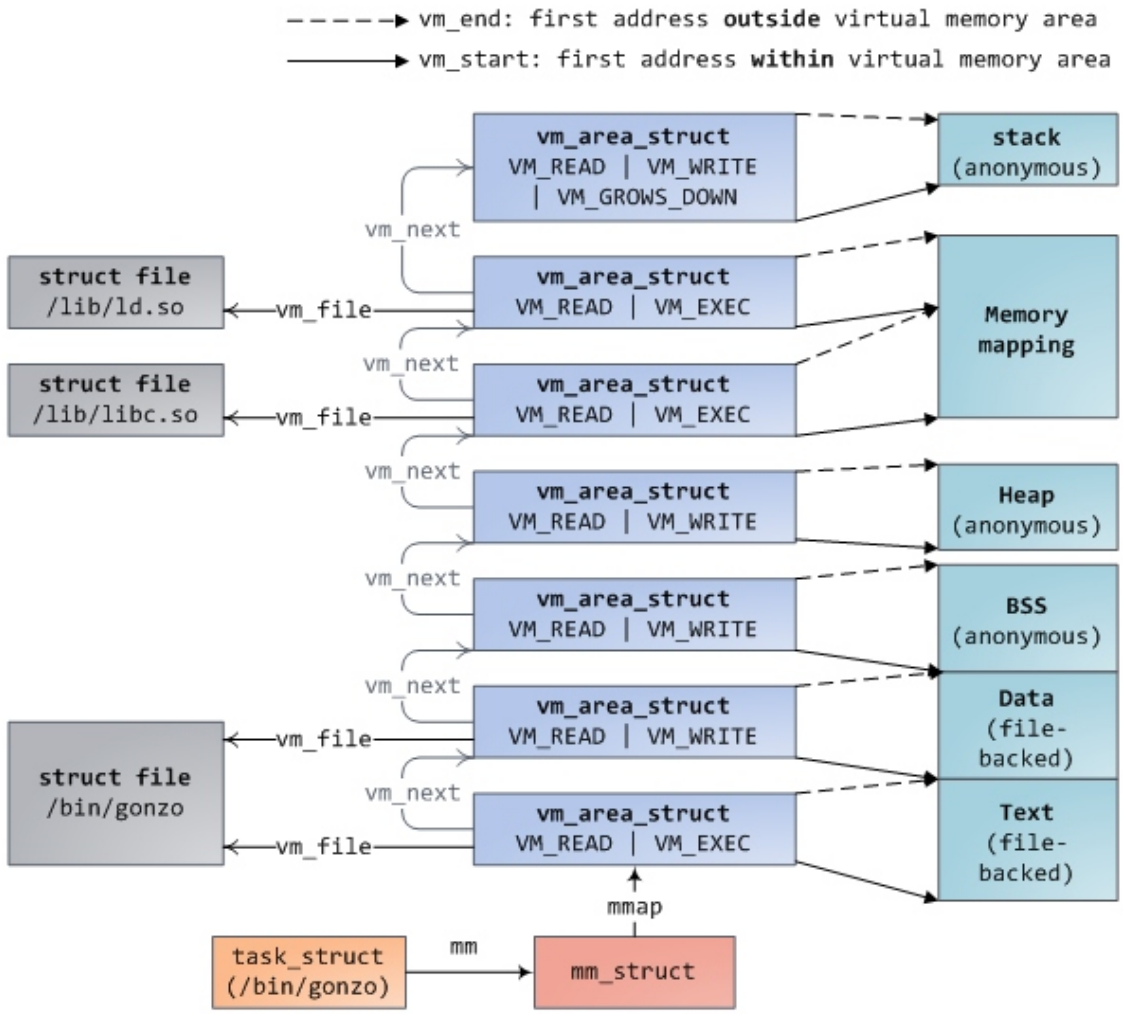

Рис. 4.3. Більш докладна схема сегментів пам'яті процесу

Статична пам'ять програми

Статична пам'ять програми — це частина пам'яті, яка є відображенням коду об'єктного файлу програми. Вона ініціалізується завантажувачем програм ОС з виконуваного файлу (спосіб ініціалізації залежить від конкретного формату виконуваного файлу).

Вона включає декілька секцій, серед яких загальнопоширеними є:

- Секція `text` — секція пам'яті, в яку записуються самі інструкції програми
- Секція `data` — секція пам'яті, в яку записуються значення статичних змінних програми

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

- Секція `bss` — секція пам'яті, в якій виділяється місце для запису значень оголошених, але не ініціалізований в програмі статичних змінних
- Секція `rodata` — секція пам'яті, в яку записуються значення констант програми
- Секція таблиці символів — секція, в якій записані всі зовнішні (експортовані) символи програми з адресами їх місцезнаходження в секціях `text` або `data` програми

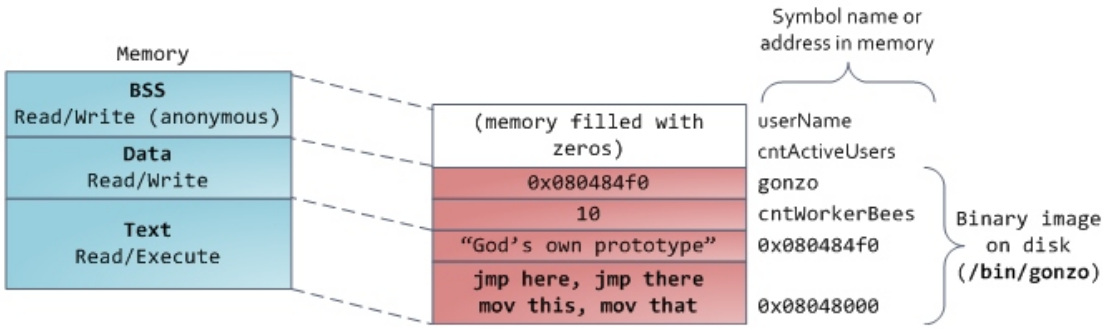

Рис. 4.4. Статична пам'ять програми

Динамічна пам'ять програми

Динамічна пам'ять виділяється програмі в момент її створення, але її вміст створюється програмою по мірі її виконання. В області динамічної пам'яті використовується 3 стандартні секції, крім яких можуть бути і інші.

Стандартні секції включають:

- стек (stack)
- купа (heap)
- сегмент відображуваної пам'яті (memory map segment)

Для виділення додаткового обсягу динамічної пам'яті використовується системний виклик `brk`.

Стек

(Більш правильна назва використовуваної структури даних — **стопка** або **магазин**. Однак, історично прижилося запозичена назва стек).

Стек (stack) — це частина динамічної пам'яті, яка використовується при виклику функцій для зберігання їх аргументів і локальних змінних. В архітектурі x86 стек росте вниз, тобто вершина стека має найменший адресу. Регістр `SP`

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

(Stack Pointer) вказує на поточну вершину стека, а регістр BP (Base Pointer) вказує на т.зв. базу, яка використовується для розділення стека на логічні частини, що відносяться до однієї функції — **фрейми** (кадри). Крім операцій звернення до пам'яті безпосередньо, які можуть застосуються в тому числі для роботи зі стеком, додатково для нього також введені інструкції `push` і `pop`, які записують дані на вершину стека і зчитують дані з вершини, після чого видаляють. Ці операції здійснюють зміну регістру SP.

Як правило, програми на високорівневих мовах програмування не працюють зі стеком безпосередньо, а це робить за них компілятор, реалізуючи певні угоди про виклики функцій і способи зберігання локальних змінних. Однак стандартна функція `alloca` дозволяє динамічно виділяти пам'ять на стеку.

Виклик функції високорівневого мови створює на стеку новий фрейм, який містить аргументи функції, адресу повернення з функції, покажчик на початок попереднього фрейму, а також місце під локальні змінні.

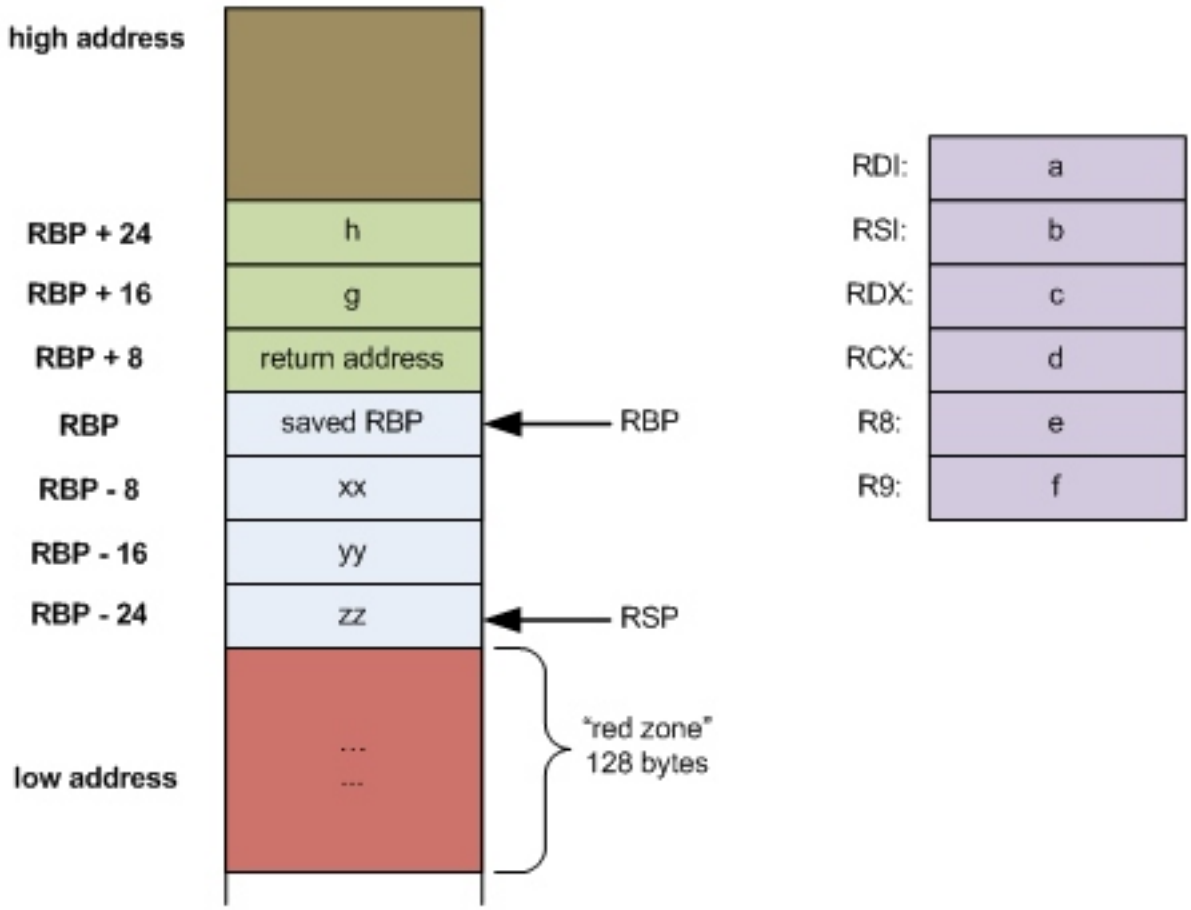

Рис. 4.5. Вид фрейму стека при виклику в рамках AMD64 ABI

На початку роботи програми в стеку виділений тільки 1 фрейм для функції `main` і її аргументів — числового значення `argc` і масиву покажчиків змінної довжини

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

`argv`, кожен з яких записується на стек окремо, а також змінних оточення.

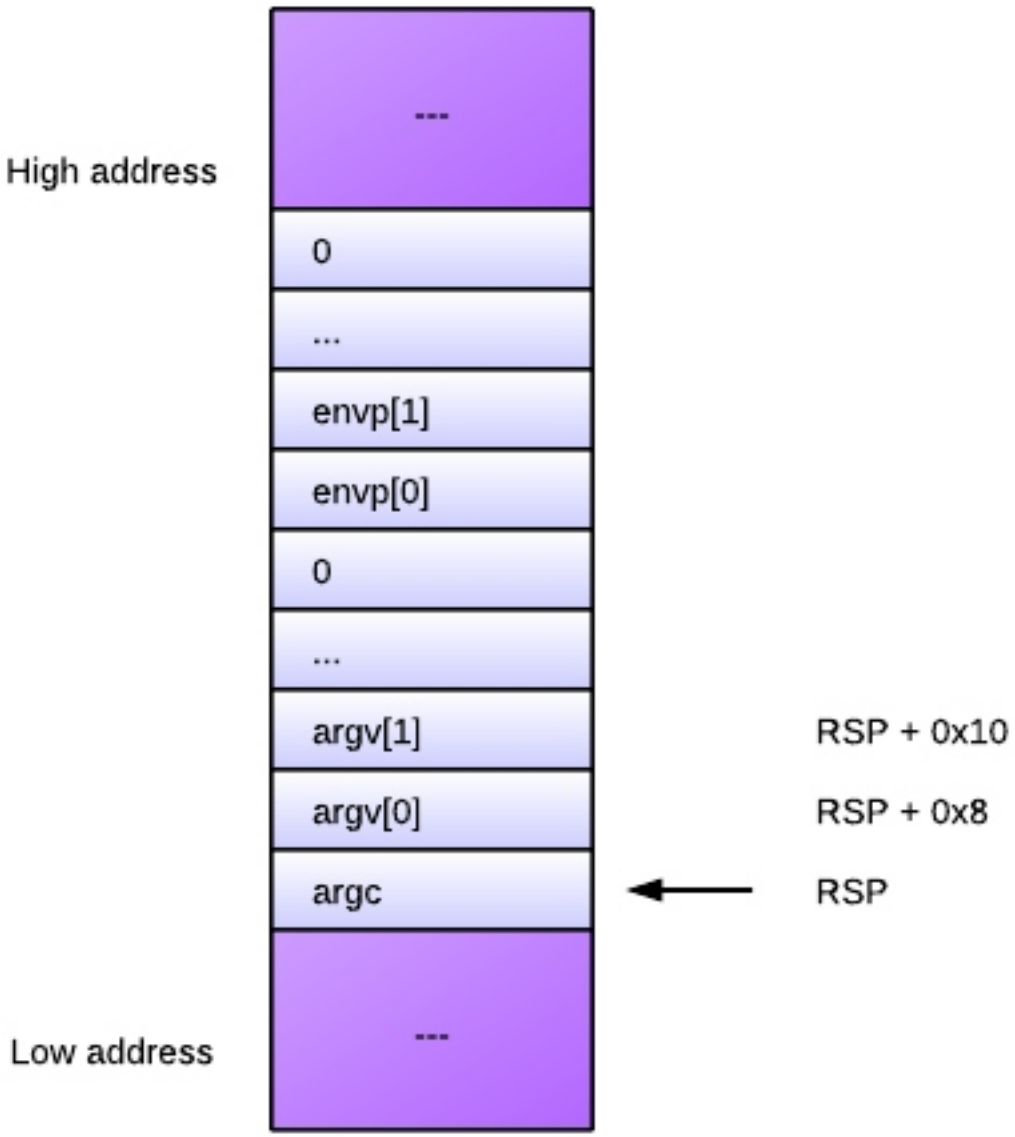

Рис. 4.6. Вид стека після виклику функції main

Купа

Купа (heap) — це частина динамічної пам'яті, призначена для виділення ділянок пам'яті довільного розміру. Вона в першу чергу використовується для роботи з масивами невідомої заздалегідь довжини (буферами), структурами і об'єктами.

Для управління купою використовується підсистема виділення пам'яті (memory allocator), інтерфейс до якого — це функції `malloc`/`calloc` в C, а також `free`.

Основні вимоги до аллокатору пам'яті:

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

- мінімальне використовуваний простір, фрагментація
- мінімальний час роботи
- максимальна **локальність** пам'яті
- максимальна настроюваність
- максимальна сумісність зі стандартами
- максимальна переносимість
- виявлення найбільшого числа помилок
- мінімальні аномалії

Багато мов високого рівня реалізують більш високорівневий механізм управління пам'яттю понад системним аллокатором — автоматичне виділення пам'яті зі збирачем сміття. У цьому випадку у програми немає безпосереднього інтерфейсу до аллокатору і керування пам'яттю здійснює середовище виконання програми.

Варіанти реалізації збирання сміття:

- підрахунок посилань
- трасування/з виставленням прапорів (Mark and Sweep)

Сегмент файлів, що відображаються в пам'ять

Сегмент файлів, що відображаються в пам'ять — це окрема область динамічної пам'яті, яка використовується для ефективно роботи з файлами, а також для підключення ділянок пам'яті інших програм за допомогою виклику `mmap`.

Виконувані файли

В результаті компіляції програми на асемблері в машинний код створюється виконуваний файл, тобто файл, що містить безпосередньо інструкції процесора.

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

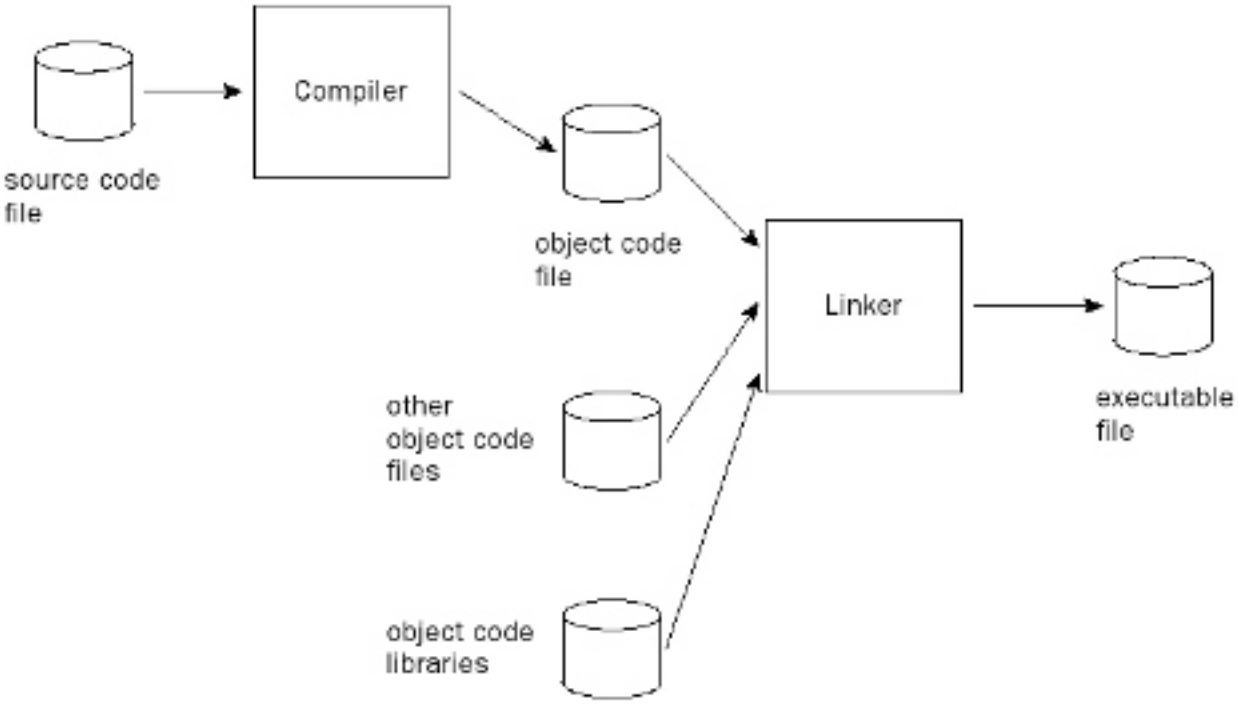

Рис. 4.7. Етапи створення виконуваного файлу

Типи виконуваних файлів:

- об'єктний файл (object file) — файл, перетворений компілятором, але не приведений остаточно до виду виконуваного файлу в одному з форматів виконуваних файлів
- виконувана програма (executable) — файл в одному з форматів виконуваних файлів, який може бути запущений завантажувачем програм ОС
- розділяєма бібліотека (shared library) — програма, яка не може бути запущена самостійно, а підключається (компілятором) як частина інших програм
- знімок вмісту пам'яті (core dump) — знімок стану програми в момент її виконання — може дозволити продовжити виконання програми з того місця, на якому він був зроблений

Формати виконуваних файлів

Формат виконуваного файлу — це визначена структура бінарного файлу, створюваного компілятором і збирачем програми і споживана завантажувачем програм ОС.

В рамках формату виконуваних файлів описується:

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

- спосіб задання секцій файлу, їх кількість і порядок
- метадані, їх типи та розміщення у файлі
- яким чином файл буде завантажуватися: за якою адресою в пам'яті, в якій послідовності
- спосіб опису імпортованих і експортованих символів
- обмеження на розмір файлу і т.п.

Поширені формати:

- .COM
- A.out
- COFF
- DOS MZ Executable
- Windows PE
- Windows NE
- ELF

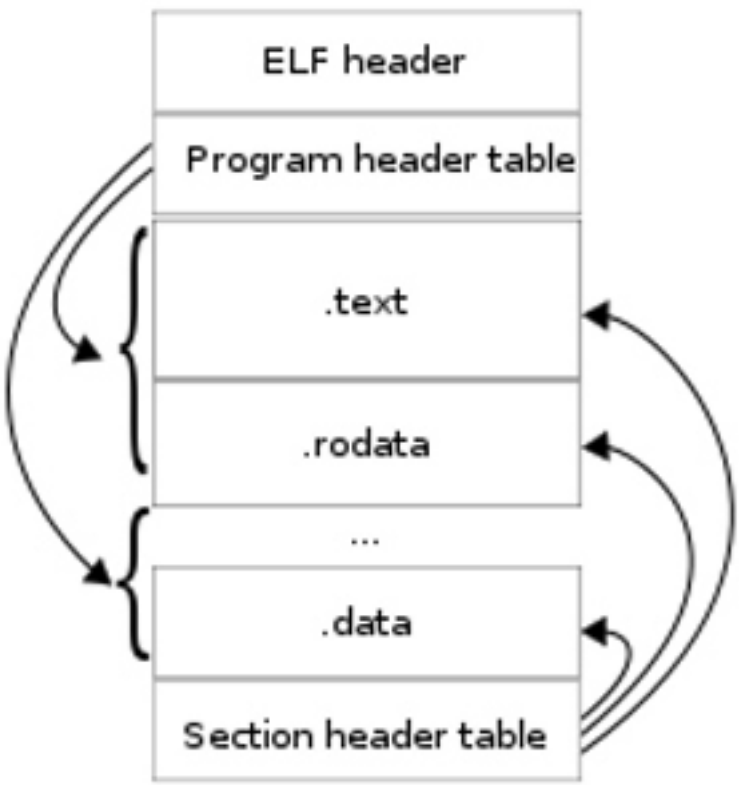

Рис. 4.8. Формат ELF

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Формат ELF (Executable and Linkable Format) — стандартний формат виконуваних файлів в Linux. Файл в цьому форматі містить:

- заголовок файлу
- таблицю заголовків сегментів програми
- таблицю заголовків секцій програми
- блоки даних

Сегменти програми містять інформацію, використовувану завантажувачем програми, а секції — використовувану компонувальником. Ця інформація включає ввідні дані для релокації.

Окреме питання — це запис відлагоджувальної інформації в виконуваний файл. Це може бути специфікованим як самим форматом, так і додатковими форматами, такими як:

- stabs
- DWARF

Бібліотеки

Бібліотеки містять функції, які виконують типові дії, які можуть використовуватися іншими програмами. На відміну від виконуваної програми бібліотека не має точки входу (функції `main`) і призначена для підключення до інших програм або бібліотек. Стандартна бібліотека C (`libc`) — перша і основна бібліотека будь-якої програми на C.

Бібліотеки можуть підключатися до програми в момент:

- збирання — build time (такі бібліотеки називаються статичними)
- завантаження — load time
- виконання — run time

Розділювані бібліотеки — це бібліотеки, які підключаються в момент завантаження або виконання програми і можуть розділятися між декількома програмами в пам'яті для економії пам'яті. Крім того вони не включаються в код програми і таким чином не збільшують його обсяг. З іншого боку, вони в більшій мірі страждають від проблеми конфлікту версій залежностей різних компонент (у застосуванні до бібліотек вона має назву DLL hell).

Способи підключення розділюваних бібліотек в Unix:

- релокації часу завантаження програми

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

· позиційно-незалежний код (PIC)

Релокації часу завантаження програми використовують спеціальну секцію виконуваного файлу — таблицю релокації, в якій записуються перетворення, які потрібно провести з кодом бібліотеки при її завантаженні. Недоліки цього способу — збільшення часу завантаження програми через необхідність переписування коду бібліотеки для застосування всіх релокацій на цьому етапі, а також неможливість зробити секцію коду бібліотеки розділеною в пам'яті через те, що релокації для кожної програми застосовуватися по-різному, адже бібліотека завантажується в пам'ять за різними віртуальним адресами.

Позиційно-незалежний код використовує таблицю глобальних відступів (Global Offset Table, GOT), в якій записуються адреси всіх експортованих символів бібліотеки. Його недолік — це уповільнення усіх звернень до символів бібліотеки через необхідність виконувати додаткове звернення до GOT.

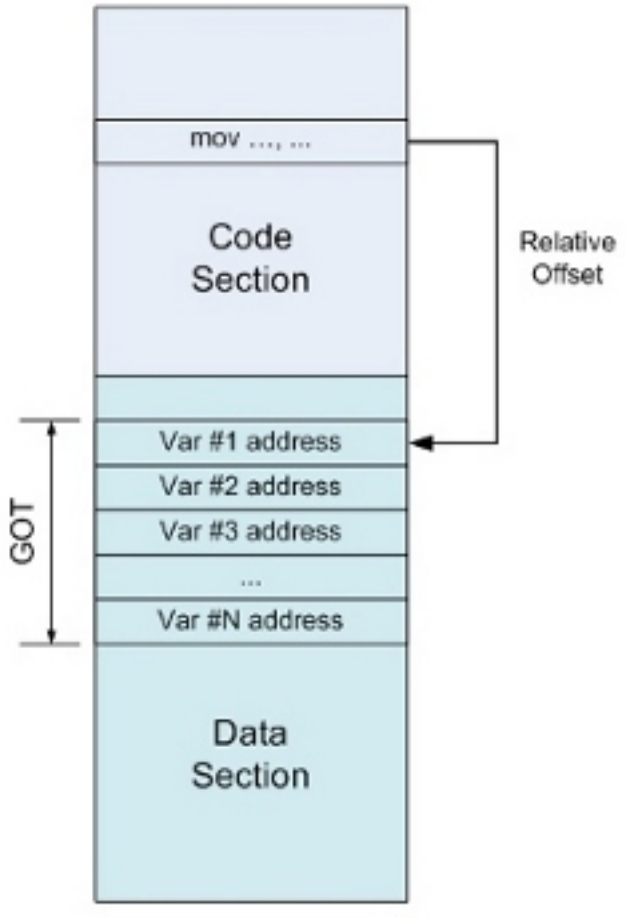

Рис. 4.9. Таблиця глобальних відступів

Для підтримки пізнього зв'язування функцій через механізм "трампліну" також застосовується таблиця компонування процедур (Procedure Linkage Table, PLT).

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

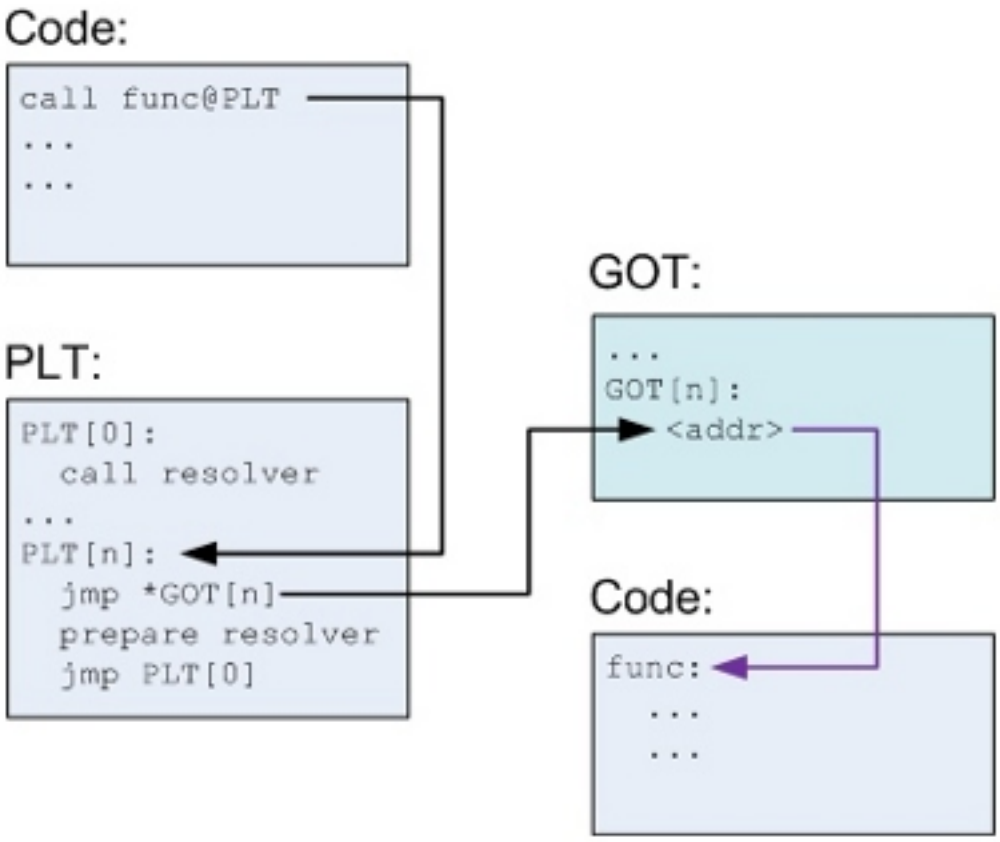

Рис. 4.10. Реалізація трампліну при виклику функції за допомою таблиць компоновки процедур

Віртуальні машини

Віртуальна машина — це програмна реалізація реального комп'ютера, яка виконує програми.

Застосування віртуальних машин:

- збільшення переносимості коду
- дослідження та оптимізація програм
- емуляція
- пісочниця
- віртуалізація
- платформа для R&D мов програміірованія
- платформа для R&D різних комп'ютерних систем
- приховування програм (віруси)

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Типи:

- системна — повна емуляція комп'ютера
- процесна — часткова емуляція комп'ютера для одного з процесів ОС

Системні ВМ

Види системних ВМ:

- гіпервізор/монітор віртуальних машин: тип 1 (на голому залізі) і тип 2 (на ОС-хазяїні)
- паравіртуалізації

Вимоги Попека і Голдберга для ефективної віртуалізації:

- усі чутливі інструкції апаратної архітектури є привілейованими
- не має тимчасових обмежень на виконання інструкцій (рекурсивна віртуалізація)

Приклади: VMWare, VirtualBox, Xen, KVM, Quemu, Linux LXC containers, Solaris zones

Процесні ВМ

Процесні ВМ функціонують за принципом 1 процес — 1 примірник ВМ і, як правило, надають інтерфейс більш високого рівня, ніж апаратна платформа.

Код програми для таких ВМ компілюється в проміжне представлення (**байт-код**), який потім інтерпретується ВМ. Часто в них також використовується JIT-компіляція байт-коду в рідній код.

Варіанти реалізації:

- Стек-машина (0 - операнд)
- Акумуляторна машина (1- операнд)
- Регістрова машина (2- або 3-операнд)

Приклади: JVM, .Net CLR, Parrot, LLVM, Smalltalk VM, V8

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Література

- Anatomy of a Program in Memory
- How is a binary executable organized
- Inside Memory Management
- Stack frame layout on x86-64
- Eli Bendrski on Static and Dynamic Object Code in Linux:
  - How Statically Linked Programs Run on Linux
  - Load-time relocation of shared libraries
  - Position Independent Code (PIC) in shared libraries
  - Position Independent Code (PIC) in shared libraries on x64
- How To Write Shared Libraries
- Executable and Linkable Format (ELF)
- Why do Windows functions all begin with a pointless MOV EDI, EDI instruction?

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Управління процесами

Процес

Процес - це адресний простір і єдина нитка управління. (Застаріле визначення)

Більш точно поняття процесу включає в себе:

- Програму, яка виконується
- Її динамічний стан (регістровий контекст, стан пам'яті і т.д.)
- Доступні ресурси (як індивідуальні для процесу, такі як дескриптори файлів, так і що розділяються з іншими)

В ОС структура Процес (Process control block) - одна з ключових структур даних. Вона містить всю інформації про процес, необхідну різним підсистемам ОС. Ця інформація включає:

- PID (ID процесу)
- PPID (ID процесу-батька)
- Шлях і аргументи, з яким запущений процес
- Програмний лічильник
- Покажчик на стек
- і т.і.

Нижче наведена невелика частина цієї структури в ОС Linux:

```
#include<sched.h>
struct task_struct {
        /* Стан:
         * -1 - заблокований,
         *  0 - готовність,
         * >0 - зупинений */
        volatile long state;
        void *stack;
        unsigned long flags;
        int prio, static_prio, normal_prio;
        struct list_head run_list;
        const struct sched_class *sched_class;
        ...
/* task state */
        struct linux_binfmt *binfmt;
```


* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

```
        long exit_state;
        int exit_code, exit_signal;
        int pdeath_signal;
        pid_t pid;
        pid_t tgid;
        struct task_struct *real_parent;
        struct task_struct *parent;
        struct list_head children;
        struct list_head sibling;
        struct task_struct *group_leader;
        struct timespec start_time;
        struct timespec real_start_time;
        ...
/* process credentials */
        uid_t uid,euid,suid,fsuid;
        gid_t gid,egid,sgid,fsgid;
        struct group_info *group_info;
        kernel_cap_t cap_effective,
            cap_inheritable, cap_permitted;
        unsigned keep_capabilities:1;
        struct user_struct *user;
        ...
/* open file information */
        struct files_struct *files;
/* namespace */
        struct nsproxy *nsproxy;
/* signal handlers */
        struct signal_struct *signal;
        struct sighand_struct *sighand;
        ...
};
```

Нитка управління

Нитка управління (thread) — це один логічний ланцюжок виконання команд. В одному процесі може бути як одна нитка управління, так і декілька (в системах, що підтримують багатопоточність — multithreading).

ОС надає інтерфейс для створення ниток управління і в цьому випадку бере на себе їх планування на рівні з плануванням процесів. У стандарті POSIX описаний подібний інтерфейс, який реалізований в бібліотеці **PTHREADS**. Нитки, що надаються ОС, називаються **рідними** (native). Однак будь-який процес може організувати управління нитками усередині себе незалежно від ОС (фактично, в рамках однієї рідної нитки ОС). Такий підхід називають **зеленими**

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

або **легковагими** нитками.

Волокно (fiber) - легковага нитка, яка працює в системі кооперативної багатозадачності (див. нижче).

Переваги рідних ниток:

- Не вимагають додаткових зусиль по реалізації
- Використовують стандартні механізми планування ОС
- Блокування і реакція на сигнали ОС відбувається в рамках нитки, а не всього процесу

Переваги зелених ниток:

- Потенційно менші накладні витрати на створення і підтримку
- Не вимагають перемикання контексту при системних викликах, що дає потенційно більшу швидкодію
- Гнучкість: процес може реалізувати будь-яку стратегію планування таких ниток

Види процесів

Процеси можуть запускатися для різних цілей:

- виконання якихось єдиноразових дій (наприклад, скрипти)
- виконання завдань під управлінням користувача (інтерактивні процеси, такі як редактор)
- безперервної роботи у фоновому режимі (сервіси або демони, такі як сервіс термінала або поштовый сервер)

Процес-демон — це процес, який запускається для довгострокової роботи у фоновому режимі, відключається від терміналу, який його запустив (його стандартні потоки вводу-виводу закриваються або перенаправляються у лог-файл), і змінює свої права доступу на мінімально необхідні. Управління таким процесом, звичайно, здійснюється за допомогою механизму сигналів ОС.

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Життєвий цикл процесу

Рис. 5.1. Життєвий цикл процесу

Породження процесу

Всі процеси ОС, за винятком першого процесу, який запускається після завантаження ядра, мають батька. Створення нового процесу вимагає ініціалізації структури PCB і запуску (постановки на планування) нитки управління процесу. Основною вимогою до цих операцій є швидкість виконання. Ініціалізація структури PCB з нуля є витратною операцією, крім того породженому процесу, як правило, потрібен доступ до деяких ресурсів (таких як потоки вводу-виводу) батьківського процесу. Тому зазвичай структура нового процесу створюється методом **клонування** структури батька. Альтернативою є завантаження попередньо ініціалізованої структури з файлу і її модифікація.

Модель **fork/exec** - це модель двоступеневого породження процесу в Unix-системах. На першому ступені за допомогою системного виклику `fork` створюється ідентична копія поточного процесу (для забезпечення швидкодії, як правило, через механізм копіювання-при-запису - copy-on-write, COW). На другому етапі за допомогою операції `exec` в пам'ять створеного процесу завантажується нова програма. У цій моделі процес-батько має можливість дочекатися завершення дочірнього процесу за допомогою системних викликів сімейства `wait`. Розбивка цієї операції на два етапи дає можливість легко породжувати ідентичні копії процесу (наприклад, для масштабування програми — такий спосіб застосовується в мережевих серверах), а також гнучко управляти ресурсами, доступними дочірньому процесу.

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Завершення процесу

По завершенню процес повертає цілочисельний код повернення (exit code) — результат виконання функції `main`. У Unix-системах код повернення, рівний 0, сигналізує про успіх, всі інші говорять про помилку (розробник програми може довільно зіставляти помилки їх кодам повернення).

Процес може завершитися наступним чином:

- нормально: викликавши системний виклик `exit` або виконавши `return` з функції `main` (що призводить до виклику `exit` у функції `libc`, яка запустила `main`)
- помилково: якщо виконання процесу викликає критичну помилку (`Segmentation Fault`, `General Protection Exception`, `Divizion by zero` або інше аппартание виключення)
- примусово: якщо процес завершується ОС, наприклад, при нестачі пам'яті, а також, якщо він не обробляє будь-який з надісланих йому сигналів (в тому числі, сигнал `KILL`, який неможливо обробити, через що відправка цього сигналу завжди приводить до завершення процесу)

Використовуючи функції сімейства `wait`, один процес може очікувати завершення іншого. Це часто використовується в батьківських процесах, яким потрібно отримати інформацію про завершення своїх нащадків, щоб продовжити роботу. Виклики `wait` є блокуючими — функція не завершиться, поки не завершиться процес, якого вона чекає.

Якщо нащадок завершується, але батьківський процес не викликає `wait`, нащадок стає т.зв. процесом зомбі. Це завршені процеси, інформація про завершення яких ніким не запрошена. Втім, після завершення батьківського процесу всі його нащадки переходять до процесу з PID 1, тобто `init`. Він самостійно очищає інформацію, що залишилася після зомбі.

Приклад програми, що породжує новий процес і очікує його завершення:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main() {
    /* Клонуємо поточний процес */
    pid_t childpid = fork();
    /* Змінна childpid буде існувати
     * і в оригінальному процесі, і в його клоні,
     * але у дочірньому процесі вона буде 0 */
```


* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

```
    if (!childpid) {
        // Це дочірній процес
        char* command[3] = {"echo", "Hello, world!", 0};
        execvp(command[0], command);
        /* Якщо не відбулося ніяких помилок,
         * execvp() не завершується, і програма
         * ніколи не досягнемо цієї ділянки коду */
        exit(EXIT_FAILURE);
    } else if (childpid == -1) {
        // fork() повертає -1 у випадку помилки
        fprintf(stderr, "Can't fork, exiting...\n");
        exit(EXIT_FAILURE);
    } else {
        // Це батьківський процес
        exit(EXIT_SUCCESS);
    }
    return 0;
}
```

Робота процесу

У мультіпроцессних системах всі процеси виконуються на процесорі не весь час своєї роботи, а тільки його частину. Відповідно, можна виділити:

- загальний час знаходження процесу в системі: від запуску до завершення
- (чистий) час виконання процесу
- час очікування

У стан очікування процес може перейти або при надходженні переривання від процесора, або після виклику самим процесом блокуючої операції, або після добровільної передачі процесом управління ОС (виклик планувальника `schedule` при витісняючий багатозадачності або ж операція `yield` при кооперативної багатозадачності).

При переході процесу в стан очікування відбувається **перемикання контексту** і запуск на процесорі коду ядра ОС (коду обробки переривань або коду планувальника). Перемикання контексту вимагає збереження в пам'яті змісту пов'язаних з виконуваним процесом регістрів і завантаження в регістри значень для наступного процесу.

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

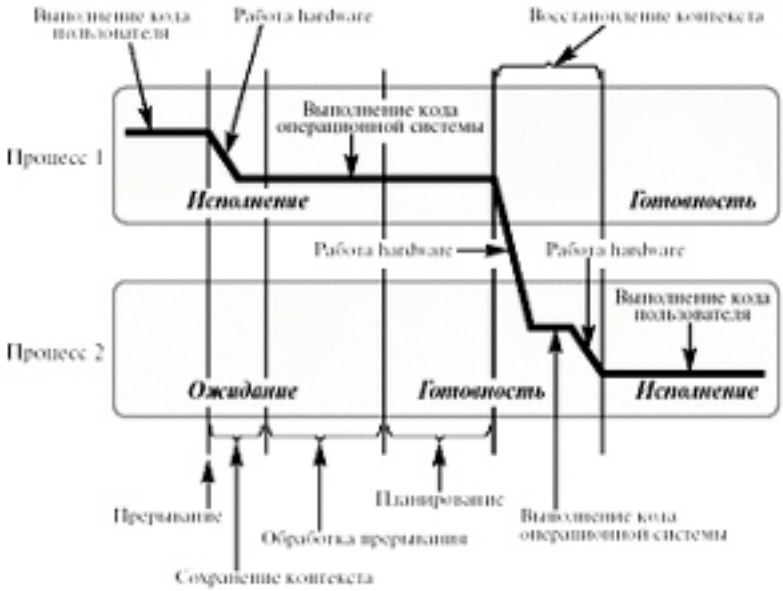

Рис. 5.2. Перемикання контексту

У системах із витісняючою багатозадачності переривання `CLOCK INTERRUPT` викликає планувальник ОС, який переводить процес в стан очікування, якщо відведений йому на виконання час минув.

Завершення блокуючих операцій знаменується перериванням, при обробці якого ОС переводить заблокований процес в стан готовності до роботи.

Планування процесів

Багатозадачність — це властивість операційної системи або середовища програмування забезпечувати можливість паралельної (або псевдопараллельной) обробки декількох процесів.

Види багатозадачності:

- Витісняюча
- Невитісняюча
- Кооперативна (підвид невитісняючої)

Витісняюча багатозадачність потребує наявність в ОС спеціальної програми-планувальника процесів, який приймає рішення про те, який процес повинен виконуватися і скільки часу відвести йому на виконання. Після завершення відведеного часу процес примусово переривається і керування передається іншому процесу.

При невитісняючий багатозадачності процеси працюють по черзі, причому

перемикання відбувається по завершенню всього процесу або логічного блоку в його рамках. Кооперативна багатозадачність - це варіант невитісняючої багатозадачності, в якій тільки сам процес може сигналізувати ОС про готовність передати управління.

Алгоритми планування процесів

Планування процесів застосовується в системах з витісняючою багатозадачністю.

Вимоги до алгоритмів планування:

- Справедливість
- Ефективність (в сенсі утилізації ресурсів)
- Стабільність
- Масштабованість
- Мінімізація часу: виконання, очікування, відгуку

Алгоритми планування для вибору наступного процесу на виконання, як правило, використовують **пріоритет** процесу. Пріоритет може визначатися статично (один раз для процесу) або ж динамічно (перераховуватися на кожному кроці планування).

Алгоритм Перший прийшов — перший обслуговий (FCFS)

Рис. 5.3. Алгоритм FCFS

Це простий алгоритм з найменшими накладними витратами. Це алгоритм зі статичним пріоритетом, в якості якого виступає час приходу процесу. Це найменш стабільний алгоритм, який не може гарантувати прийнятний час відгуку в інтерактивних системах, тому він застосовується тільки в системах

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

batch-обробки.

Алгоритм Карусель (Round Robin)

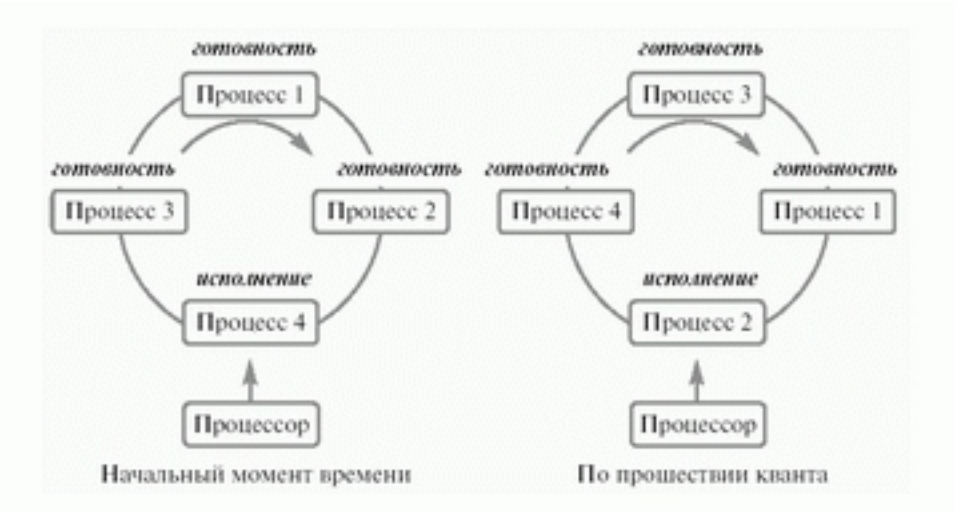

Рис. 5.4. Алгоритм Карусель

Цей алгоритм визначає поперемінне виконання всіх процесів протягом однакового кванту часу, після завершення якого незалежно від стану процесу він переривається і керування переходить до наступного процесу. Цей алгоритм є найбільш стабільним і простим. Він взагалі не використовує пріоритет процесу.

Алгоритм справедливого планування

В основі цього алгоритму лежить принцип: з усіх кандидатів на виконання повинен вибиратися той, у якого відношення чистого часу фактичної роботи до загального часу перебування в системі найменше. Іншими словами, цей алгоритм використовує динамічний пріоритет, який обчислюється за формулою `p = t / T` (де `t` - чистий час виконання, `T` - час, що минув від запуску процесу; чим менше значення `p`, тим пріоритет вище).

Алгоритм Багаторівнева черга зі зворотнім зв'язком

Рис. 5.5. Алгоритм Багаторівнева черга зі зворотнім зв'язком

Як приклад більш складного адаптивного алгоритму можуть служити багаторівневі черги із зворотним зв'язком, які приймають рішення про пріоритет процесу на основі часу, який необхідний йому для завершення роботи або поточного логічного блоку.

Реальні алгоритми

Алгоритми, що застосовуються в реальних системах, діляться на алгоритми для інтерактивних систем і алгоритми для систем реального часу. Алгоритми для систем реального часу завжди мають обмеження на час завершення окремих операцій, тому їм для планування необхідна додаткова інформація, якої, як правило, немає в інтерактивних системах — наприклад, час до завершення процесу.

В основі реальних алгоритмів лежать базові алгоритми, перераховані вище, але також вони використовують деякі додаткові парамаетри такі як:

- Епохи (часові інтервали, в кінці яких накопичена для планування

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

інформація обнуляється)

- Угрупування процесів по класах (м'якого реального часу, процеси ядра, інтерактивні, фонові і т.д.) для забезпечення кращого часу відгуку системи

Крім того, в таких системах враховуються технології Симетричний мультіпроцесінг (Symmetrical Multiprocessing, SMP) і Одночасна багатопоточність (Symulteneous Multithreading, SMT), при яких кілька ядер процесора або кілька логічних потоків виконання в процесорі працюють із загальною пам'яттю.

Міжпроцесна взаємодія (IPC)

Цілі взаємодії:

- модульність (шлях Unix: маленькі шматочки, слабко пов'язані між собою, які роблять щось одне і роблять це дуже добре)
- масштабування
- спільне використання даних
- поділ привілеїв
- зручність

Типи взаємодії:

- Через поділювану пам'ять
- Обмін повідомленнями
  - Сигнальний
  - Канальний
- Широкомовний

Взаємодія через поділювану пам'ять

Найшвидший і найпростігий спосіб взаємодії, при якому процеси записують і зчитують дані із загальної області пам'яті. Він не вимагає жодних накладних витрат, але потребує домовленості про формат записуваних даних. Проблеми цього підходу:

- необхідність блокуючої синхронізації для забезпечення неконфліктного доступу до спільної пам'яті
- збільшення логічної зв'язності між окремими процесами

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

- неможливість масштабуватися за рамками пам'яті одного комп'ютера

Обмін повідомленнями

Передача повідомлень володіє прямо протилежними властивостями і вважається кращим способом організації взаємодії в загальному випадку. Повідомлення можуть передаватися як індивідуально, так і в рамках виділеної сесії обміну повідомленнями.

Сигнальный спосіб взаємодії

Сигнальний спосіб — це варіант взаємодії через передачу повідомлень, який припускає можливість відправки тільки заздалегідь відомих сигналів, які не мають ніякого навантаження у вигляді даних. Таким чином сигнали можуть передавати інформацію тільки про заздалегідь заданий набір подій. Така система є простою, але не здатна обслуговувати всі варіанти взаємодії. Тому вона часто застосовується для обслуговування критичних сценаріїв роботи.

Системний виклик `kill` дозволяє посилати сигнали процесам Unix. Серед них є зарезервовані сигнали, такі як:

- `TERM` — запит на завершення процесу
- `HUP` — запит на перезапуск процесу
- `ABRT` — запит на скасування поточної операції
- `PIPE` — сигнал про закриття конвеєра іншим процесом
- `KILL` — сигнал про примусове завершення процесу
- та ін.

Процес в Unix зобов'язаний обробити сигнал, який надішов до нього, інакше ОС примусово завершує його роботу.

Канальный спосіб взаємодії

Канальний спосіб — це варіант взаємодії через передачу повідомлень, при якому між процесами встановлюється канал з'єднання, в рамках якого передаються повідомлення. Цей канал може бути як одностороннім (повідомлення йдуть тільки від одного процесу до іншого), так і двостороннім.

Pipe (конвеєр, анонімний канал) — односторонній канал, який дозволяє процесам передавати дані у вигляді потоку байт. Це найпростіший спосіб взаємодії в Unix, що має спеціальний синтаксис в командних оболонках (`proc1 | proc2`, що означає, що дані з процесу `proc1` передаються в `proc2`).

Анонімний канал створюється системним викликом `pipe`, який приймає на вхід масив з двох чисел і записує в них два дескриптора (один з них відкритий на запис, а інший — на читання).

Особливості анонімних каналів:

- дані передаються порядково
- не заданий формат повідомлень, тобто процеси самі повинні "домовлятися" про нього
- помилка в каналі призводить до посилки сигналу `PIPE` до процесу, який намагався виконати читання або запис в нього

Іменований канал (named pipe) створюється за допомогою системного виклику `mkfifo`. Фактично, він є правильним замінником для обміну даними через тимчасові файли, оскільки той володіє наступними недоліками:

- використання повільного диска замість більш швидкої пам'яті
- витрата місця на диску (в той час як при обміні даними через FIFO після зчитування вони стираються); більш того, місце на диску може закінчитися
- у процесу може не бути прав створити файл або ж файл може бути зіпсований/видалений іншим процесом

Модель Акторів

Модель акторів Х'ювіта — це теоретична модель, що досліджує взаємодію незалежних програмних агентів.

Актор — це незалежний легковагий процес, який взаємодіє з іншими процесами тільки через передачу повідомлень. У цій моделі процес не використовує поділювану пам'ять.

Ця модель лежить в основі мови програмування Erlang, а також бібліотека для організації розподіленої роботи Java-додатків Akka.

Література

- POSIX: командная оболочка и системные вызовы
- Advanced Linux Programming: Processes
- Daemons, Signals, and Killing Processes
- Taxonomy of UNIX IPC Methods

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

- Understanding the Linux Kernel - 10. Process Scheduling
- The Linux Process Scheduler
- Linux Kernel 2.4 Internals - 2. Process and Interrupt Management
- ULE: A Modern Scheduler For FreeBSD
- How Linux 3.6 nearly broke PostgreSQL
- How Erlang does scheduling
- Daemons, Signals, and Killing Processes

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Синхронізація

Проблема синхронізації

Синхронізація в комп'ютерних системах — це координація роботи процесів таким чином, щоб послідовність їх операцій була передбачуваною. Як правило, синхронізація необхідна при спільному доступі до ресурсів, що розділяються. **Критична секція** — частина програми, в якій є звернення до спільно використовуваних даних. При знаходженні в критичній секції двох (або більше) процесів, виникає стан гонки/змагання за ресурси. **Стан гонки** — це стан системи, при якому результат виконання операцій залежить від того, в якій послідовності виконуватимуться окремі процеси в цій системі, але керувати цією послідовністю немає можливості. Іншими словами, це протилежність правильної синхронізації.

Для уникнення гонок необхідне виконання таких умов:

- **Взаємного виключення**: два процеси не повинні одночасно перебувати в одній критичній області
- **Прогресу**: процес, що знаходиться поза критичною областю, не може блокувати інші процеси
- **Обмеженого очікування**: неможлива ситуація, в якій процес вічно чекає попадання в критичну область
- Також в програмі, в загальному випадку, не повинно бути припущень про швидкість або кількості процесорів

Приклад умови гонок: i++

```
movl i, $eax  // i - мітка адреси змінної i в пам'яти
incl $eax
movl $eax, i
```

Оскільки процесор не може модифікувати значення в пам'яті безпосередньо, для цього йому потрібно завантажити значення в регістр, змінити його, а потім знову вивантажити в пам'ять. Якщо в процесі виконання цих трьох інструкцій процес буде перерваний, може виникнути непередбачуваний результат, тобто має місце умова гонки.

Класичні задачі синхронізації

Класичні задачі синхронізації — це модельні задачі, на яких досліджуються різні

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

ситуації, які можуть виникати в системах з розділюємим доступом та конкуренцією за спільні ресурси. До них відносяться задачі: Виробник-споживач, Читачі-письменники, Обідаючі філософи, Сплячий перукар, Курці сигарет, Проблема Санта-Клауса та ін.

Задача **Виробник-споживач** (також відома як задача обмеженого буфера): 2 процеси — виробник і споживач — працюють із загальним ресурсом (буфером), що має максимальний розмір N. Виробник записує в буфер дані послідовно в чарунки 0,1,2, ..., поки він не заповниться, а споживач читає дані з буфера у зворотному порядку, поки він не спорожніє. Запис і зчитування не можуть відбуватися одночасно.

Наївний розв'язок

```
int buf[N];
int count = 0;
void producer() {
    while (1) {
        int item = produce_item();
        while (count == N - 1)
            /* do nothing */ ;
        buf[count] = item;
        count++;
    }
}
void consumer() {
    while (1) {
        while (count == 0)
            /* do nothing */ ;
        int item = buf[count - 1];
        count--;
        consume_item(item);
    }
}
int main() {
    make_thread(&producer);
    make_thread(&consumer);
}
```

Проблема синхронізації в цьому варіанті: якщо виробник буде перерваний споживачем після того, як запише дані в буфер `buf[count] = item`, але до того, як збільшить лічильник, то споживач зчитає з буфера елемент перед тільки що записаним, тобто в буфері утворюється дірка. Після того як виробник таки збільшить лічильник, лічильник якраз буде вказувати на цю дірку. Симетрична

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

проблема є і у споживача.

Також у цієї задачі може бути багато модифікацій: наприклад, кількість виробників і споживачів може бути більше 1. У цьому випадку додаються нові проблеми синхронізації.

Ще одна проблема цього рішення — це безглузда витрата обчислювальних ресурсів: цикли `while (count == 0) / * do nothing * /;` - т.зв. **зайняте очікування** (busy loop, busy waiting) або ж **поллінг** (pooling) — це ситуація, коли процес не виконує ніякої корисної роботи, але займає процесор і не дає в цей час працювати на ньому іншим процесам. Таких ситуацій треба по можливості уникати.

Алгоритми програмної синхронізації

Програмний алгоритм синхронізації — це алгоритм взаємного виключення, який не заснований на використанні спеціальних команд процесора для заборони переривань, блокування шини пам'яті і т.д. У ньому використовуються тільки загальні змінні пам'яті та цикл для очікування входу в критичну секцію виконуваного коду. У більшості випадків такі алгоритми не ефективні, тому що використовують поллінг для перевірки умов синхронізації.

Приклад програмного адгоритма — алгоритм Петерсона:

```
int interested[2];
int turn;
void enter_section(int process_id) {
    int other = 1 - process_id;
    interested[process_id] = 1;
    turn = other;
    while (turn == other && interested[other])
    /* busy waiting */;
}
void leave_section(int process_id) {
    interested[process_id] = 0;
}
```

Див. також алгоритм Деккера, алгоритм пекарні Лемпорта та ін.

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Апаратні інструкції синхронізації

Апаратні інструкції синхронізації реалізують **атомарні** примітивні операції, на основі яких можна будувати механізми синхронізації більш високого рівня. Атомарність означає, що вся операція виконується як ціле і не може бути перерваною посередині. Атомарні примітивні операції для синхронізації, як правило, виконують разом 2 дії: запис значення і перевірку попереднього значення. Це дає можливість перевірити умову і відразу записати таке значення, яке гарантує, що умова більше не буде виконуватися.

Try-and-set lock (TSL)

Інструкції типу `try-and-set` записують в регістр значення з пам'яті, а в пам'ять — значення 1. Потім вони порівнюють значення в регістрі з 0. Якщо в пам'яті і був 0 (тобто доступ до критичної області був відкритий), то порівняння пройде успішно, і в той же час в пам'ять буде записаний 1, що гарантує, що наступного разу порівняння вже не буде успішним, тобто доступ закриється.

Реалізація критичної секції за допомогою `TSL`:

```
enter_region:
        TSL REGISTER, LOCK
        CMP REGISTER, 0
        JNE ENTER_REGION
        RET
leave_region:
        MOV LOCK, 0
        RET
```

Те ж саме на C:

```
void lock(int *lock) {
    while (!test_and_set(lock))
        /* busy waiting */;
}
```

Compare-and-swap (CAS)

Инструкції типу `compare-and-swap` записують у регістр нове значення і при цьому перевіряють, що старе значення в регістрі рівне значенню, запам'ятованому раніше.

В x86 називається `CMPXCHG`.

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Аналог на C:

```
int compare_and_swap(int* reg, int oldval, int newval) {
    int old_reg_val = *reg;
    if (old_reg_val == oldval)
        *reg = newval;
    return old_reg_val;
}
```

Проблема ABA (ABBA): CAS інструкції не можуть відстежити ситуацію, коли значення в регістрі було змінено на нове, а потім знову було повернуто до попереднього значення. У більшості випадків це не впливає на роботу алгоритму, а в тих випадках, коли впливає, необхідно використовувати інструкції з перевіркою на таку ситуацію, такі як `LL/SC`.

Інші аппаратні инструкції

- Подвійний CAS
- Fetch-and-add
- Load-link/store-conditional (LL/SC)

Системні механізми синхронізації

За допомогою апаратних інструкцій можна реалізувати більш високорівневі конструкції, які можуть обмежувати доступ в критичну область, а також сигналізувати про якісь події.

Найпростішим варіантом обмеження доступу в критичну область є **змінна-замок**: якщо її значення дорівнює 0, то доступ відкритий, а якщо 1 — то закритий. У неї є 2 атомарні операції:

- заблокувати (**lock**) — перевірити, що значення дорівнює 0, і встановлює його в 1 або ж чекати, поки воно не стане 0
- розблокувати (**unlock**), яка встановлює значення в 1

Також корисною може бути операція спробувати заблокувати (**trylock**), яка не чекає поки значення замку стане 0, а відразу повертає відповідь про неможливість заблокувати замок.

Спінлок

Спінлок — це замок, очікування при блокуванні якого реалізовано у вигляді зайнятого очікування, тобто потік "крутиться" в циклі, очікуючи розблокування

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

замка.

Реалізація на асемблері за допомогою CAS:

```
lock:  # 1 = locked, 0 = unlocked.
    dd 0
spin_lock:
    mov eax, 1
    loop:
        xchg eax, [lock]
        # Atomically swap the EAX register with
        # the lock variable.
        # This will always store 1 to the lock,
        # leaving previous value in the EAX register
        test eax, eax
        # Test EAX with itself. Among other things, this
        # sets the processor's Zero Flag if EAX is 0.
        # If EAX is 0, then the lock was unlocked and
        # we just locked it. Otherwise, EAX is 1
        # and we didn't acquire the lock.
        jnz loop
        # Jump back to the XCHG instruction if Zero Flag
        # is not set, the lock was locked,
        # and we need to spin.
        ret
spin_unlock:
    mov eax, 0
    xchg eax, [lock]
    # Atomically swap the EAX register with
    # the lock variable.
    ret
```

Використання спінлока доцільно тільки в тих областях коду, які не можуть викликати блокування, інакше весь час, відведений планувальником потоку, що очікує на спінлоці, буде витрачено на очікування і при цьому інші потоки не будуть працювати.

Семафори

Семафор — це примітив синхронізації, що дозволяє обмежити доступ до критичної секції тільки для N потоків. При цьому, як правило, семафор дозволяє реалізувати це без використання зайнятого очікування.

Концептуально семафор включає в себе лічильник і чергу очікування для

потоків. Інтерфейс семафора складається з двох основних операцій: опустити (**down**) і підняти (**up**). Операція опустити атомарно перевіряє, що лічильник більше 0 і зменшує його. Якщо лічильник дорівнює 0, потік блокується і ставитися в чергу очікування. Операція підняти збільшує лічильник і посилає чекаючим потокам сигнал прокинутися, після чого один з цих потоків зможе повторити операцію опустити.

Бінарний семафор — це семафор з N = 1.

Мьютекс (mutex)

Мьютекс — від словосполучення mutual exclusion, тобто взаємне виключення — це примітив синхронізації, що нагадує бінарний семафор з додатковою умовою: розблокувати його повинен той же потік, який і заблокував.

Реалізація мьютекса за допомогою примітиву CAS:

```
void acquire_mutex(int *mutex) {
    while (cas(mutex, 1, 0)) /* busy waiting */;
}
void release_mutex(int *mutex) {
    *mutex =  1;
}
```

Варто відзначити, що не всі системи надають гарантії того, що мьютекс буде розблоковано саме потоком, що його заблокував. У наведеному вище прикладі ця умова якраз не перевіряється.

Мьютекс з можливістю повторного входу (re-entrant mutex) — це мьютекс, який дозволяє потоку кілька разів блокувати його.

RW lock

RW lock — це особливий вид замку, який дозволяє розмежувати потоки, які виконують тільки читання даних, і які виконують їх модифікацію. Він має операції заблокувати на читання (**rdlock**), яка може одночасно виконуватися декількома потоками, і заблокувати на запис (**wrlock**), яка може виконуватися тільки 1 потоком. Також правильні реалізації RW-замку дозволяють уникнути проблеми інверсії пріоритетів (див. нижче).

Змінні умови і монітори

Монітор — це механізм синхронізації в об'єктно-орієнтованому програмуванні, при використанні якого об'єкт позначається як синхронізований і компілятор

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

додає до викликів всіх його методів (або тільки виділених синхронізованих методів) блокування за допомогою мьютекса. При цьому код, що використовує цей об'єкт, не повинен піклуватися про синхронізацію. У цьому сенсі монітор є більш високорівневої конструкцією, ніж семафори і мьютекси.

Змінна умови (condition variable) — примітив синхронізації, що дозволяє реалізувати очікування якоїсь події і оповіщення про неї. Над нею можна виконувати такі дії:

- очікувати (**wait**) повідомлення про якусь подію
- сигналізувати подію всім потокам, що очікують на даній змінній. Сигналізація може бути блокуючою (**signal**) — у цьому випадку управління переходить до чекаючого потоку, - і неблокуючою (**notify**) — у цьому випадку управління залишається у сигналізуючого потоку

Більшість моніторів підтримують усередині себе використання змінних умови. Це дозволяє декільком потоком заходити в монітор і передавати керування один одному через цю змінну.

Див. http://en.wikipedia.org/wiki/Monitor_(synchronization)

Інтерфейс синхронізації

POSIX Threads (Pthreads) — це частина стандарту POSIX, яка описує базові примітиви синхронізації, підтримувані в Unix системах. Ці примітиви включають семафор, мьютекс і змінні умови.

Futex (швидкий замок у просторі користувача) — це реалізація мьютекса в Unix-системах, яка оптимізована для мінімального використання функцій ядра ОС, за рахунок чого досягається більш швидка робота з ним. За допомогою фьютексів в Linux реалізовані семафори і змінні умови.

Над фьютексамі можна робити такі базові операції:

- очікувати - `wait (addr, val)` - перевіряє, що значення за адресою `addr` рівне `val`, і, якщо це так, то переводить потік в стан очікування, а інакше продовжує роботу (тобто входить в критичну область)
- розбудити - `wake (addr, n)` - відправляє n потокам, які очікують на фьютексе за адресою `addr`, повідомлення про необхідність пробудитися

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Проблеми синхронізації

Крім умов гонки і зайнятого очікування неправильна синхронізація може призвести до наступних проблем:

Тупик/взаємне блокування (deadlock) — ситуація, коли 2 або більше потоків очікують розблокування замків один від одного і не можуть просунутися. Найпростіший приклад коду, який може викликати взаємне блокування:

```
void thread1() {
    acquire(mutex1);
    do_something1();
    acquire(mutex2);
    do_something_else1();
    release(mutex2);
    release(mutex1);
}
void thread2() {
    acquire(mutex2);
    do_something2();
    acquire(mutex1);
    do_something_else2();
    release(mutex1);
    release(mutex2);
}
```

Живий блок (livelock) — ситуація з більш, ніж двома потоками, при якій потоки чекають розблокування один від одного, при цьому можуть змінювати свій стан, але не можуть просунутися глобально в своїй роботі. Така ситуація більш складна, ніж тупик і виникає значно рідше: як правило, вона пов'язана з часовими особливостями роботи програми.

Інверсія пріоритету — ситуація, коли потік з більштм пріоритетом змушений чекати потоки з меншим пріоритетом через неправильну синхронізацію

Голодування — ситуація, коли потік не може отримати доступ до загального ресурсу і не може просунутися. Така ситуація може бути наслідком як тупика, так і інверсії пріоритету.

Способи запобігання тупикових ситуацій

Всі способи боротьби з тупиками не є універсальними і можуть працювати тільки за певних умов. До них відносяться:

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

- монітор тупиків, який стежить за тим, щоб очікування на якомусь із замків не тривало занадто довго, і рестартує чекаючий потік в такому випадку
- нумерація замків і їх блокування тільки в монотонному порядку
- алгоритм банкіра
- та ін.

Неблокуюча синхронізація

Неблокуюча синхронізація — це група підходів, які ставлять своєю метою вирішити проблеми синхронізації альтернативним шляхом без явного використання замків і заснованих на них механізмів. Ці підходи створюють нову **конкурентную парадигму** програмування, що можна порівняти з появою структурної парадигми як запереченням підходу до написання програм з використанням низькорівневої конструкції `goto` і переходу до використання структурних блоків: ітерація, умовний вираз, цикл.

Нічого спільного (shared-nothing)

Архітектури програм без загального стану розглядають питання побудови систем з взаємодіючих компонент, які не мають поділюваних ресурсів та обмінюються інформацією тільки через передачу повідомлень. Такі системи, як правило, є набагато менш зв'язними, і тому краще піддаються масштабуванню і є менш чутливими до відмови окремих компонент.

Теоретичні роботи на цей рахунок: Взаємодіючі паралельні процеси (Communicating Parallel Processes, CPP) та модель Акторів. Практична реалізація цієї концепції — мова Erlang. У цій моделі одиницею обчислення є легковагий процес, який має "поштову скриньку", на яку йому можуть відправлятися повідомлення від інших процесів, якщо вони знають його ID в системі. Відправлення повідомлень є неблокуючим (асинхронної), а прийом є синхронним: тобто процес може заблокуватися в очікуванні повідомлення. При цьому час блокування може бути обмеженим програмно.

CSP

Взаємодіючі послідовні процеси (Communicating Sequential Processes, CSP) — це ще один підхід до організації взаємодії без використання замків. Одиницями взаємодії в цій моделі є процеси і канали. На відміну від моделі CPP, пересилання даних через канал в цій моделі відбувається, як правило, синхронно, що дає можливість встановити певну послідовність виконання процесів. Дана концепція реалізована в мові програмування Go.

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Програмна транзакційна пам'ять

Транзакція — це група послідовних операцій, яка являє собою логічну одиницю роботи з даними. Транзакція може бути виконана або цілком і успішно, дотримуючись цілісність даних і незалежно від паралельно виконуваних інших транзакцій, або не виконана взагалі і тоді вона не повинна призвести до жодного ефекту. У теорії баз даних існує концепція `ACID`, яка описує умови, які накладаються на транзакції, щоб вони мали корисну семантику. **A** означає **атомарність** — транзакція повинна виконуватися як єдине ціле. **C** означає **цілісність** (consistency) — в результаті транзакції будуть або змінені всі дані, з якими працює транзакція, або ніякі з них. **I** означає **ізоляція** — зміни даних, які виготовляються під час транзакції, стануть доступні іншим процесам тільки після її завершення. **D** означає **збереження** — після завершення транзакції система БД гарантує, що її результати зберігатися в довготривалому сховищі даних. Ці властивості, за винятком, хіба що, останнього можуть бути застосовні не тільки до БД, але і до будь-яких операцій роботи з даними. Програмна система, яка реалізує транзакційні механізми для роботи з пам'яттю — це Програмна транзакційна пам'ять (Software Transactional Memory, STM). STM лежить в основі мови програмування Clojure, а також доступна в мовах Haskell, Python (в реалізації PyPy) та інших.

Література

- A Beautiful Race Condition
- Java's Atomic and volatile, under the hood on x86
- Mutexes and Condition Variables using Futexes
- Common Pitfalls in Writing Lock-Free Algorithms
- Values and Change
- Beautiful Concurrency
- Programming Erlang: 8. Concurrent Programming

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Файлова система

Файлова система

Файлова система (ФС) — це компонент ОС, що відповідає за постійне зберігання даних.

Задачі:

- зберігання даних в потенційно необмежених об'ємах
- довгострокове зберігання даних (persistence)
- одночасна доступність даних багатьом процесам

На відміну від оперативної пам'яті ФС є постійною пам'яттю, тому на перше місце для них виходять збереження і доступність даних, а потім вже стоїть швидкодія.

Деякі з видів файлових систем:

- Дискові
- Віртуальні (у пам'яті і не тільки)
- Мережеві
- Розподілені
- Мета ФС (ФС, які використовують інші ФС для організації зберігання даних, а самі додають особливу логіку роботи)

Список ФС

Файловий інтерфейс — це простий і зручний спосіб роботи з даними, тому багато ОС поширюють його на інші об'єкти, крім файлів даних. Наприклад, в Unix всі пристрої підключаються в системі як спеціальні файли (символьно-специфічні — для пристроїв з послідовним введенням-виводом; і блочно-специфічні — для пристроїв з буферизованим введенням-виводом). Крім того, в Linux є спеціальні віртуальні файлові системи: `sysfs`, які оперує файлами, що відображають системні структури даних з пам'яті ядра, а також `tmpfs`, яка дозволяє створювати файли в оперативній пам'яті. Такий підхід призвів до того , що Unix став відомий як **файл-центрична** ОС, хоча є інші ОС, які ще далі просунулися в цьому. Наприклад, такі об'єкти, як сокети (див. лекцію про роботу з мережею) в Unix мають власні системні виклики, в той час як в системі Plan 9 (яка стала розвитком ідей Unix) вони доступні через той же файловий інтерфейс.

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Файл

Файл — це іменована область диска. (Невірне і застаріле визначення). Правильне визначення: файл — це об'єкт файлової системи, що містить інформацію про розміщення даних в сховищі. При цьому сховищі може бути як фізичним запам'ятовуючим пристроєм (диском, магнітною стрічкою і т.д.), так і віртуальним пристроєм, за яким стоїть оперативна пам'ять, пристрій введення-виведення, мережеве з'єднання або ж інша файлова система.

Основні операції над файлами:

- `create` — створення
- `delete` — видалення
- `open` — відкриття (на запис, читання або ж на те й інше разом)
- `close` — закриття
- `read` — читання
- `write` — запис
- `append` — запис в кінець
- `seek` — перехід на задану позицію для подальшого читання/запису
- `getattr` — отримання аттрибутів файлу
- `setattr` — встановлення аттрибутів файлу
- `lock` — блокування файлу для ексклюзивної роботи з ним (у більшості ОС блокування файлів реалізується у вигляді рекомендаційних, а не обов'язкових замків)

Файловый дескриптор — це унікальний для процесу номер в таблиці

дескрипторів процесу (яка зберігається в ядрі ОС) , який операційна система надає йому, щоб виконувати зазначені операції з файлом. Системи Windows оперують схожою концепцією — описувач файлу (file handle) .

Крім зазначених вище операцій в Unix використовуються наступні важливі системні виклики для роботи з файловими дескрипторами:

- `dup` — створення копії запису в таблиці дескрипторів з новим індексом, що вказує на той же файл
- `dup2` — "перенаправлення" одного запису на інший
- `fcntl` — управління нестандартними атрибутами та властивостями файлів, які можуть підтримуватися тими чи іншими файловими системами

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Директорія

Директорія — це об'єкт файлової системи, що містить інформацію про структуру та розміщенні файлів. Часто це теж файл, тільки особливий.

У більшості ФС директорії об'єднуються в дерево директорій, таким чином створюючи **ієрархічну** систему зберігання інформації. Це дерево має корінь, який в Unix системах називається `/`. Положення файлу в цьому дереві — це **шлях** до нього від кореня — т.зв. **абсолютний шлях**. Крім того, можна говорити про **відносний шлях** від обраної директорії до іншого об'єкта ФС. Відносний шлях може містити особливу позначення `..`, яке вказує на предка (батька) директорії в дереві директорій.

Логічне дерево директорій може об'єднувати більше однієї ФС. Включення ФС в це дерево називається **монтуванням**. В результаті монтування корінь ФС прив'язується до якоїсь директорії всередині дерева. Для першої монтованої системи її корінь прив'язується до кореня самого дерева.

У Unix також реалізована операція `chroot`, яка дозволяє змінити корінь поточного дерева на одну з директорій всередині нього. У рамках одного сеансу роботи, виконавши її, вже не можна повернутися назад, тобто отримати доступ до всіх вузлів дерева, які не є нащадками нового кореня ФС.

Існує два підходи до прив'язки файлів до директорій: у більшості ФС ця прив'язка є непрямою — через використання **посилання** на окремий об'єкт, що зберігає метадані файлу. Такі посилання називаються **жорсткими** (hard link). У таких системах один файл може мати кілька посилань на себе з різних директорій. У такій системі створення нового файлу відбувається в 2 етапи: спочатку виконується операція `create` , яка створює сам файловий об'єкт, а потім `link`, яка прив'язує його до конкретної директорії. Операція `link` може виконуватися кілька разів, і кожен файл зберігає кількість посилань на себе з різних директорій. Видалення файлу відбувається з точністю до навпаки: виконується операція `unlink`, після чого файл перестає бути прив'язаним до директорії. Якщо при цьому його лічильник посилань стає рівним 0, то система виконує над файловим об'єктом операцію `delete`. Однак є і більш примітивні системи, в яких метадані про файлі зберігаються прямо в директорії. Фактично, в таких ФС реалізована однозначна прив'язка файлу до єдиної директорії.

В обох видах систем часто підтримується також концепція **символічних посилань** (реалізованих у вигляді спеціальних файлів), які посилаються НЕ безпосередньо на файловий об'єкт ФС, а на об'єкти, що знаходиться по певному шляху. У цьому випадку ФС вже не може забезпечити перевірку існування такого об'єкта і керування іншими його властивостями, як це

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

робиться для жорстких посилань, але ,з іншого боку, символічні посилання дозволяють посилатися на файли в одному логічному дереві директорій, але за межами поточної ФС.

Операції над директоріями:

- `create` — створити
- `delete` — видалити
- `readdir` (`list`) — отримати список безпосередніх нащадків даної директорії (файлів і інших директорій)
- `opendir` — відкрити директорію для змін информації в ній
- `closedir` — закрити директорію
- `link`/`unlink`

Схеми розміщення файлів

Схема розміщення файлів — це загальний підхід до зберігання даних і метаданих файлу на пристрої зберігання даних. Основні критерії ефективності тієї чи іншої схеми — це утилізація диска, швидкодія операцій читання, запису і пошуку, а також стійкість до збоїв .

Фрагментація — це відсутність технічної можливості використовувати частину простору сховища даних через те чи інше розміщення даних в ньому. Види фрагментації:

- Зовнішня — неможливість використання цілих блоків
- Внутрішня — неможливість використання частин окремих блоків

Внутрішня фрагментація неминуча в будь-якому сховищі, яке оперує блоками більшого розміру, ніж одиниця зберігання (наприклад, жорсткий диск зберігає дані побайтно, але файли можуть починатися тільки з початку логічного блоку, який як правило, має розміри близько кілобайт). Можливість появи зовнішньої фрагментація залежить від схеми розміщення даних.

Неперервна/послідовна схема

У цій схемі файл зберігається як одна неперервна послідовність блоків. Для вказання розміщення файлу досить задати адресу його початкового блоку і загальний розмір.

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

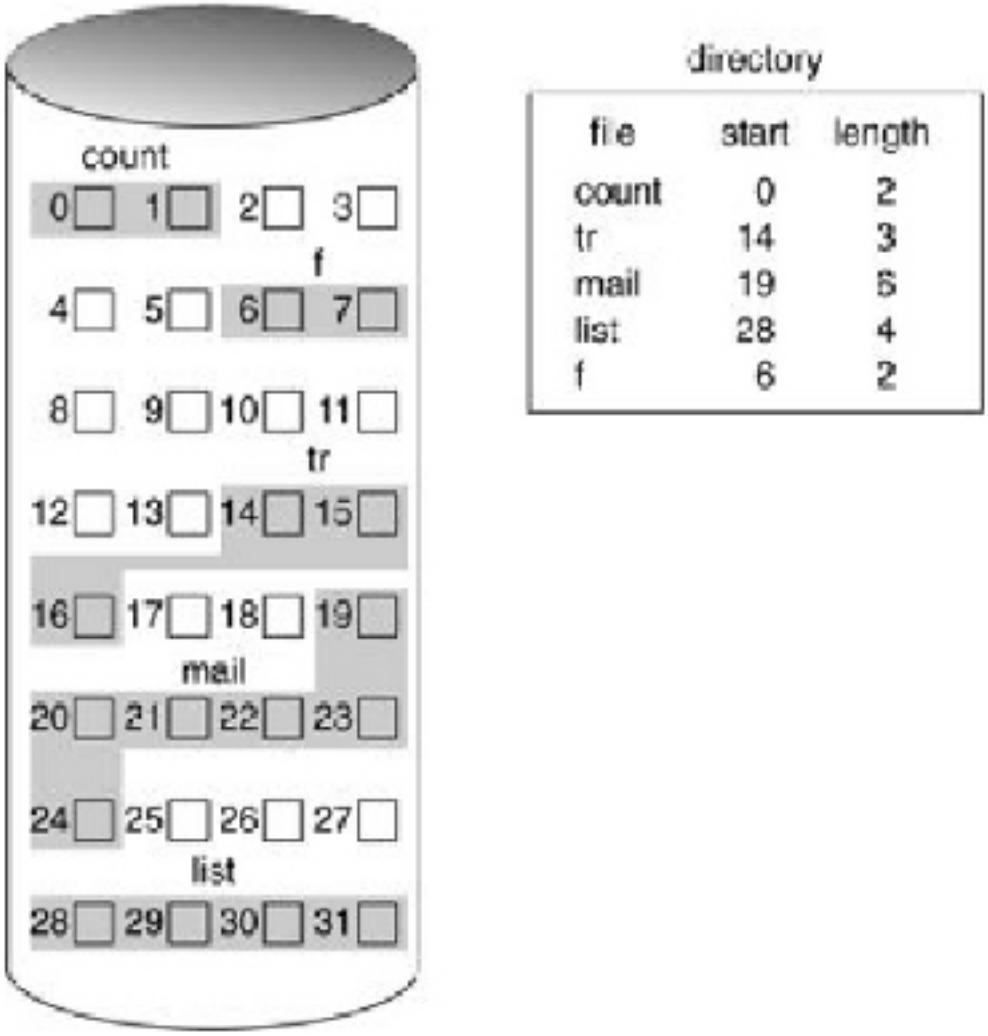

Рис. 7.1. Неперервна схема розміщення файлів

Переваги:

- легко реалізувати
- найкраща продуктивність

Недоліки:

- зовнішня фрагментація
- необхідність реалізувати облік дірок
- проблеми з ростом розміру файлу

Це найпростіша схема розміщення. Вона ідеально підходить для будь-яких носіїв, які розраховані на використання тільки в режимі для читання, або ж істотне (на порядки) перевищення числа операцій читання над записом.

Схема розміщення зв'язковим списком

У цій схемі блоки файлу можуть знаходиться в будь-якому місці диска і не бути впорядкованими якимось чином, але кожен блок повинен зберігати посилання на подальший за ним. Для вказання розміщення файлу досить задати адресу його першого блоку.

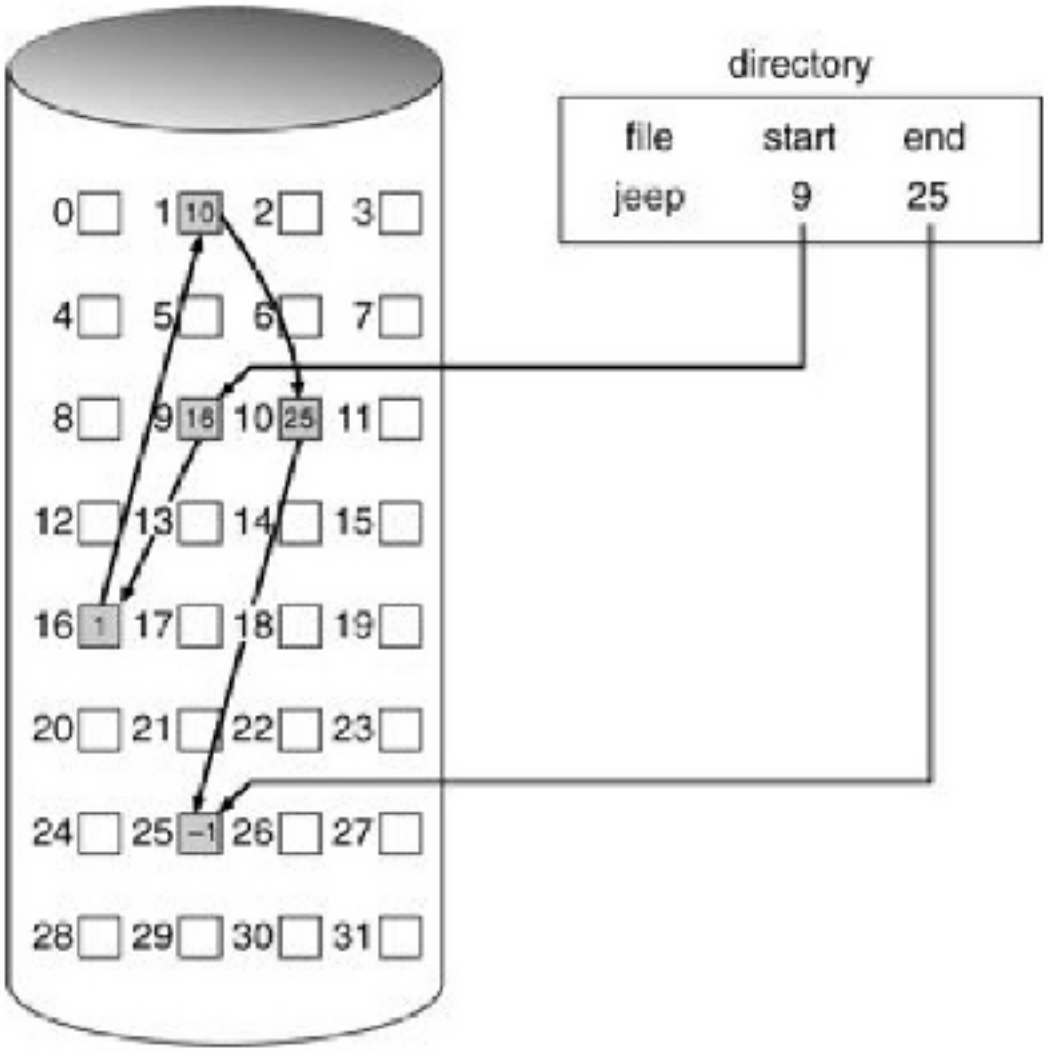

Рис. 7.2. Схема розміщення файлів зв'язковим списком

Переваги:

- немає зовнішньої фрагментації
- простота реалізації

Недоліки:

- найгірша продуктивність (особливо на магнітних дисках)
- погана стійкість до помилок: пошкодження одного блоку в середині файлу

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

призведе до того, що всі наступні за ним блоки будуть недоступні

- не кругла цифра (в ступенях 2) доступного місця в блоці через використання частини простору блоку для зберігання вказы°ника на наступний блок

Тому в чистому вигляді така схема рідко застосовується. Перераховані недоліки в основному усуває **таблична** реалізація цієї схеми. У ній інформація про наступні блокі зв'язного списку виноситься з самих блоків в окрему таблицю, яка розміщується в заздалегідь заданому місці диска. Кожен запис у таблиці зберігає номер наступного блоку для даного файлу (або ж індикатор того, що даний блок не зайнятий або ж зіпсований). Така технологія називається **Таблицею розміщення файлів** (див. FAT). Недоліком табличній схеми є централізація всіх метаданих в таблиці, що призводить до небезпеки втрати всієї ФС у разі непоправного пошкодження таблиці, а також до того, що для великих дисків ця таблиця буде мати великий об'єм, а вона повинна весь час бути повністю доступною в пам'яті.

Індексна схема

У цій схемі блоки поділяють на 2 типи: ті, які використовуються для зберігання даних файлу, і ті, які зберігають метадані — т.зв. індексні вузли (**inode**). Таким чином, на відміну від попередньої схеми, метадані про файли зберігаються розподілено в ФС, що робить цей підхід більш ефективним і відмовостійким. Крім того, ця схема відповідає власне ієрархічної моделі ФС і тривіально підтримує операції link/unlink: директорія зберігає посилання на індексні вузли прив'язаних до неї файлів. Оскільки індексні вузли — це звичайні блоки диска, до них застосовуються ті ж методи роботи, що і для звичайних блоків (наприклад , кешування).

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

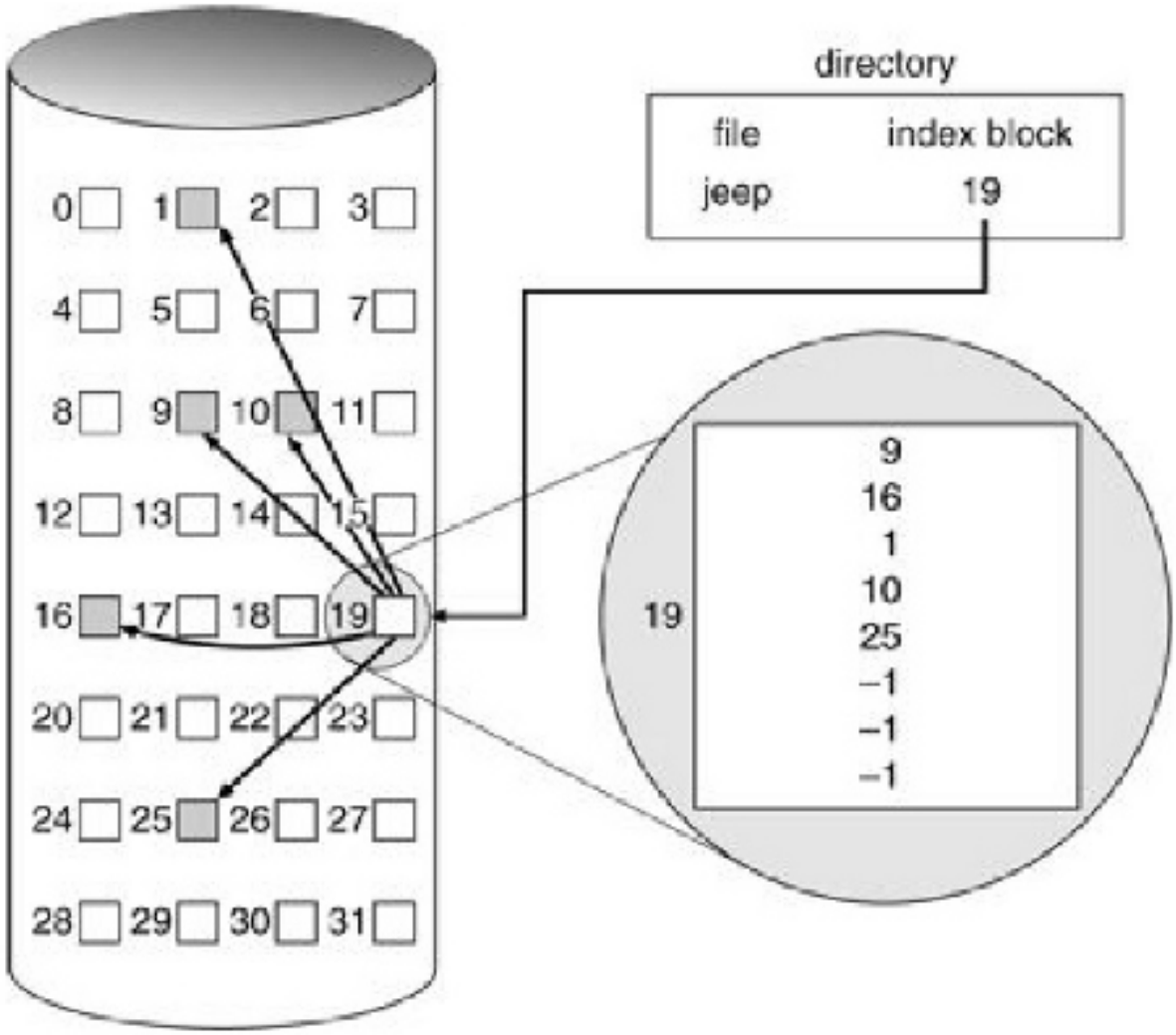

Рис. 7.3. Індексна схема розміщення файлів

Переваги:

- концептуальне відповідність між логічної і фізичної схемою зберігання
- велика ефективність і швидкодія
- велика відмовостійкість

Недоліки:

- фіксований розмір індексного вузла, що накладає обмеження на розмір файлу (для вирішення цієї проблеми використовуються непрямі inod'и, які зберігають посилання не на блоки даних, а на inod'и наступного рівня)

Оптимізація роботи ФС

Важливим параметром, що впливає на оптимізацію ФС є середній розмір файлу. Він сильно залежить від сценаріїв використання системи, але

проводяться дослідження, які заміряють це число для типової файлової системи. У 90-х роках середній розмір файлу становив 1кб, в 2000-них — 2 КБ, на даний момент — 4КБ і більше.

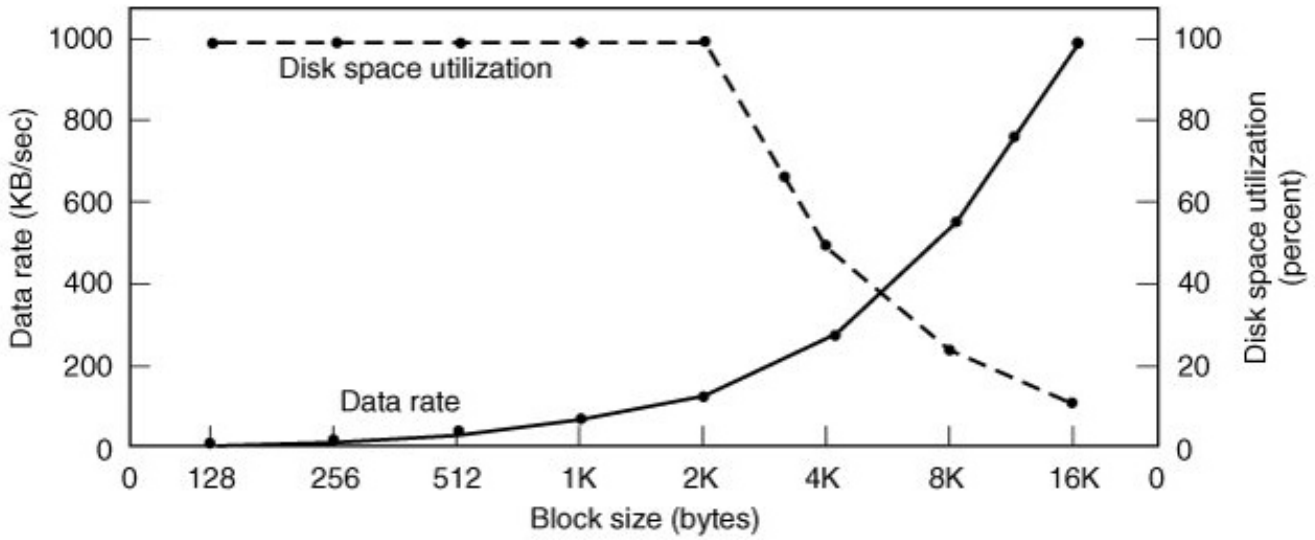

Рис. 7.4. Зв'язок розміру блоку з швидкодією і утилізацією диска при середньому розмірі файлу в системі 2 КБ

Важливий метод оптимізації ФС — це кешування. На відміну від кешування процесора у випадку дискових сховищ точний LRU можливий, але іноді він може бути навіть шкідливий. Тому для дисків часто використовується **наскрізний кеш** (зміна даних у ньому відразу ж викликає зміну даних на диску), хоча він менш ефективний. Хоча в Unix системах для збільшення швидкодії історично прийнято було використовувати операцію `sync`, яка періодично, а не постійно, зберігала зміни даних в дисковому кеші на носій.

Також оптимізація дуже істотно залежить від фізичних механізмів зберігання даних, адже профіль навантаження, наприклад, для магнітного і твердотільних дисків зовсім різні, не кажучи про ФС, що працюють по мережі.

Підходи до оптимізації ФС, що використовують жорсткий диск:

- читання блоків наперед (проте залежить від шаблону використання: послідовний або довільний доступ до даних у файлі)
- зменшення вільного ходу дискової головки за рахунок приміщення блоків в один Циллиндр
- використання ФС на основі журналу

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

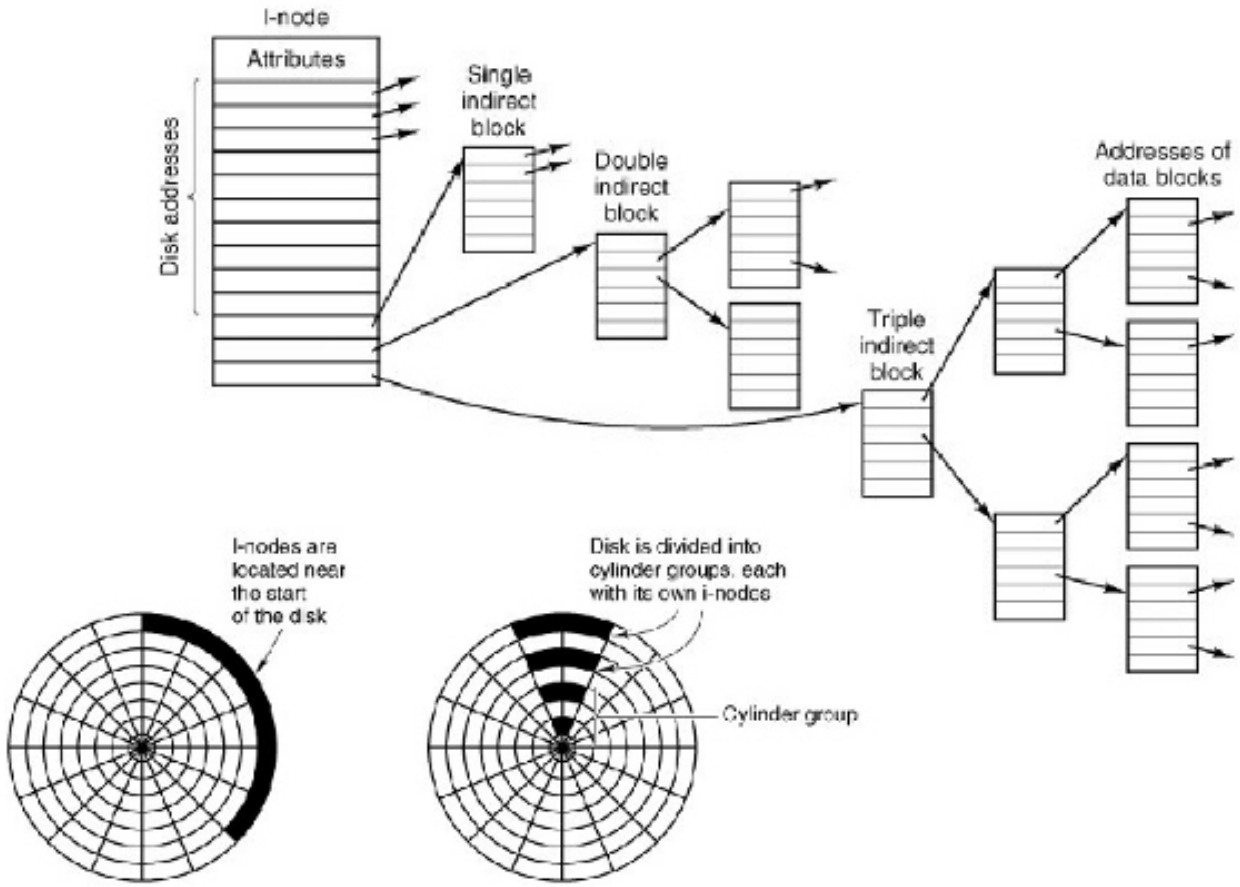

Рис. 7.5. Підходи до оптимізації індексного схеми

Література

- Another Level of Indirection
- Inferno OS Namespaces
- The Google File System
- Everything you never wanted to know about file locking
- Building the next generation file system for Windows: ReFS

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Взаємодія з мережею

> "Мережа — це комп'ютер" (гасло корпорації Sun)

Підтримка роботи з мережею на перший погляд не є функцією ядра ОС, однак на практиці більшість ОС реалізують її в ядрі. Для цього є кілька причин:

- Робота з мережею потрібна переважній більшості нетривіальних програм, тому логічно, що ОС повинна надати для них мережевий сервіс, абстрагований від різнорідних апаратних засобів і низькорівневих протоколів підтримки з'єднання

  > Будь-яка програма прагне розширитися до тих пір, поки з її допомогою не стане можливо читати пошту. Ті програми, які не розширюються настільки, замінюються тими, які розширюються. (Закон огортання софта Jamie Zawinski)

- Робота з мережею повинна бути швидкою
- Обмеження безпеки, пов'язані з роботою з мережею
- Відносна простота реалізації: невеликий набір стандартних протоколів, серед яких основні — це IP, TCP і UDP

Також в основі реализації комп'ютерних мереж лежить Принцип стійкості (Закон Постела):

> Будьте консервативним у тому, що відправляєте, і ліберальним у тому, що приймаєте від інших.

"Хибні" уявлення програмістів про мережу

Розробка розподілених програм, що використовують мережу, відрізняється від розробки програм, що працюють на одному комп'ютері. Ці відмінності виражені в наступному списку т.зв. "оман" програмістів про мережу:

- Мережа надійна
- Витрати на транспорт нульові
- Затримка нульова
- Годинники синхронізувати
- Пропускна здатність необмежена
- Топологія мережі незмінна

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

- Мережа гомогенна
- Є тільки один адміністратор
- Мережа безпечна

Модель OSI

Мережева модель OSI — це теоретична еталонна модель мережевої взаємодії відкритих систем. У ній реалізований принцип поділу турбот (separation of concerns), який виражений в тому, що взаємодія відбувається на 7 різних рівнях, кожен із яких відповідає за вирішення однієї проблеми:

- 7й - Прикладний (application) — доступ до мережних служб для прикладних додатків, дані представляються у вигляді "запитів" (requests)
- 6й - Представлення (presentation) — кодування і шифрування даних
- 5й - Сеансовий (session) — управління сеансом зв'язку
- 4й - Транспортний (transport) — зв'язок між кінцевими пунктами (які не обов'язково пов'язані безпосередньо) і надійність, дані представляються у вигляді "сегментів" (datagrams)
- 3й - Мережевий (network) — визначення маршруту і логічна адресація, забезпечення зв'язку в рамках мережі, дані представляються у вигляді "пакетів" (packets)
- 2й - Канальний (data link) — фізична адресація, забезпечення зв'язку точка-точка, дані представляються у вигляді "кадрів" (frames)
- 1й - Фізичний (physical) — робота із середовищем передачі, сигналами та двійковими даними (бітами)

При забезпеченні зв'язку між вузлами (хостами) дані проходять процес "занурення" з прикладного рівня на фізичний на відправнику і зворотний процес на одержувачу.

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Стек протоколів TCP/IP

На практиці домінуючою моделлю мережевої взаємодії є стек протоколів TCP/IP, який в цілому відповідає моделі OSI, однак не регламентує обов'язкову наявність усіх рівнів в ній. Як випливає з назви, обов'язковими протоколами в ній є TCP (або його альтернатива UDP), а також IP, які реалізують транспортний і мережевий рівень моделі OSI.

Рівні TCP/IP стека:

- 4й - Прикладний рівень (Process/Application) — відповідає трьом верхнім рівням моделі OSI (проте, не обов'язково реалізує функціональність їх всіх)
- 3й - Транспортний рівень (Transport) — відповідає транспортному рівню моделі OSI
- 2й - Міжмережевий рівень (Internet) — відповідає мережному рівню моделі OSI
- 1й - Рівень мережевого доступу (Network Access) — відповідає двом нижнім рівням моделі OSI

У цій моделі верхній і нижній рівні включають в себе декілька рівнів моделі OSI і в різних випадках вони можуть бути реалізовані як одним протоколом взаємодії, так і декількома (відповідними окремим рівням). Наприклад, протокол HTTP реалізує рівні прикладної та представлення, а протокол TLS — сеансовий і представлення, а в поєднанні між собою вони можуть покрити всі 3 верхніх рівня. При цьому протокол HTTP працює і самостійно, і в цьому випадку, оскільки він не реалізує сеансовий рівень, HTTP-з'єднання називають "stateless", тобто не мають стану.

Модель TCP/IP також називають пісочним годинником, оскільки посередині в ній знаходиться один протокол, IP, а протоколи під ним і над ним є дуже різноманітними і покривають різні сценарії використання. Стандартизація протоколу посередині дає велику гнучкість низькорівневим протоколам (яка потрібна через наявність різних способів з'єднання) і високорівневим (потрібну через наявність різних сценаріїв роботи).

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

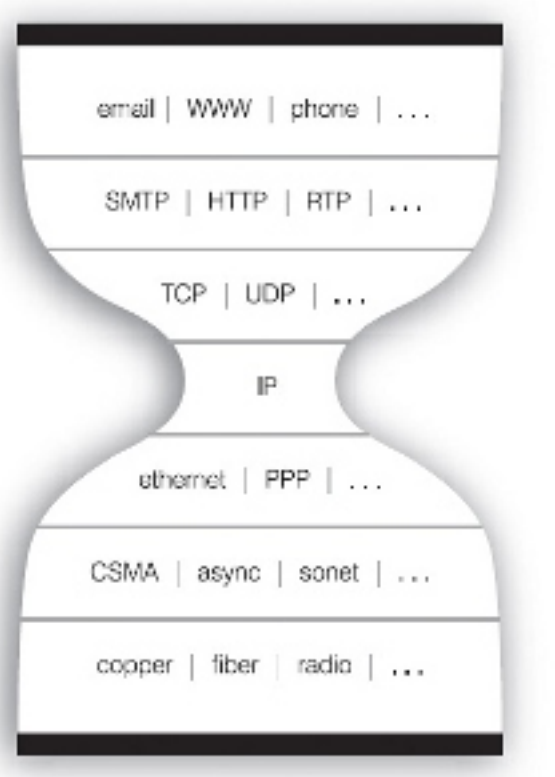

Рис. 8.1. TCP/IP стек як пісочний годинник

Інтерфейс BSD сокетів

Інтерфейс сокетів — це де-факто стандарт взаємодії прикладної програми з ядром ОС — точка входу в мережу для додатка. Він з'єднує прикладний рівень стека TCP/IP, який реалізується в просторі користувача, з нижнім рівнями, які, як правило, реалізуються в ядрі ОС.

Сокети розраховані на роботу в клієнт-серверній парадигмі взаємодії: активний клієнт підключається до пасивного сервера, який здатний одночасно обробляти багато клієнтських з'єднань. Для ідентифікації сервера при сокетних з'єднанні використовується пара IP-адреса—порт. **Порт** — це унікальне в рамках одного хоста число, як правило, обмежене в діапазоні 1-65535. Порти деляться на привілейовані (1-1024), які виділяються для програм з дозволу адміністратора системи, і всі інші — доступні для будь-яких додатків без обмежень. Більшість стандартних прикладних протоколів мають стандартні номери портів: 80 — HTTP, 25 — SMTP, 22 — SSH, 21 — FTP, 53 — DNS. Один порт може одночасно використовувати тільки один процес ОС.

Сокет — це файлоподібний об'єкт, що підтримує наступні операції:

- Створення — в результаті у програми з'являється відповідний файловий дескриптор

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

- Підключення — виконується по-різному для клієнта і сервера
- Відключення
- Читання/запис
- Конфігурація

Основні системні виклики для роботи із сокетами:

- `socket` — створення сокета
- `connect` — ініціація клієнтського з'єднання
- `bind` — прив'язка сокета до порту
- `listen` — переведення сокета в пасивний режим прослуховування порту (актуально тільки для TCP з'єднань)
- `accept` — прийняття з'єднання від клієнта (який викликав операцію `connect`) — це блокуюча операци, яка чекає надходження нового з'єднання
- `read`/`write` або ж `send`/`recv` — читання/запис даних в сокет
- `recvfrom`/`sendto` — аналогічні операції для UDP сокетів
- `setsockopt` — установка параметрів сокета
- `close` — закриття сокета

Оскільки сокети — це, фактично, інтерфейс для занурення на третій рівень TCP/IP-стека, сокети не надають механізмів для управління кодуванням даних і сеансами роботи додатків — вони просто дозволяють передати "сирий" потік байт.

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Загальна схема взаємодії через сокет

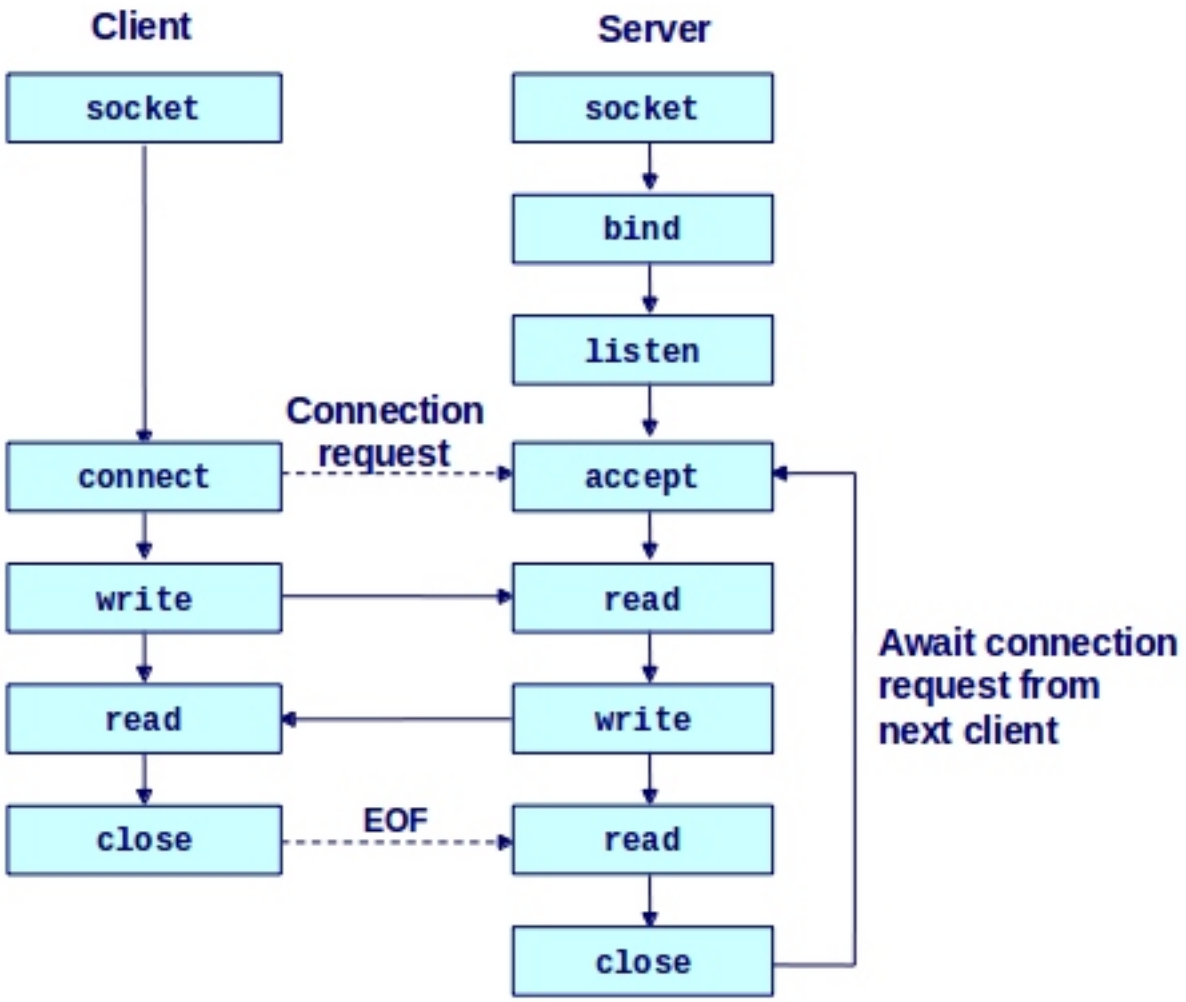

Рис. 8.2. Загальна схема взаємодії через сокет для TCP з'єднань

Як видно зі схеми, на сервері для встановлення TCP з'єднання потрібно виконати 3 операції:

- `bind` захоплює порт, після чого інші процеси не зможуть зайняти його для себе
- `listen` (для TCP з'єднання) переводить його в режим прослуховування, після чого клієнти можуть ініціювати підключення до нього
- однак, поки на сервері на виконаний `accept`, клієнтські з'єднання будуть чекати в черзі (backlog) сокета, обмеження на яку можуть бути задані у виклику `listen`

Виконання `accept` приводить до появи ще одного об'єкта сокета, який відповідає поточному клієнтському з'єднанню. При цьому серверний сокет

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

може приймати нові з'єднання.

Після виконання `accept` сервер може реалізувати кілька сценаріїв обслуговування клієнта:

- ексклюзивний варіант — обслуговування відбувається в тому ж потоці, який виконав `accept`, тому інші клієнти чекають завершення з'єднання у черзі
- 1 потік на з'єднання — відразу ж після виконання `accept` створюється новий потік, куди передається новоутворений сокет, і подальша комунікація відбувається в цьому потоці, який закривається по завершенню з'єднання. Тим часом сервер може приймати нові з'єднання. Така схема є найбільш поширеною. Її основний недолік — це великі накладні витрати на кожне з'єднання (окремий потік ОС)
- неблокуюче введення-виведення — при цьому в одному потоці сервер приймає з'єднання, а в іншому потоці працює т.зв. цикл подій (event loop), в якому відбувається асинхронна обробка всіх прийнятих клієнтських з'єднань

Неблокуюче (асинхронне) введення-виведення

Сокети підтримують як синхронне, так і асинхронне введення-виведення. Асинхронне IO є критичною функцією для створення ефективних мережевих серверів. Для підтримки асинхронного введення-виведення у сокетів (як і у інших файлових дескрипторів) є параметр `O_NONBLOCK`, який можна встановити за допомогою системного виклику `fcntl`. Після переведення файлового дескриптора в неблокуючий режим, з сокетом можна працювати за допомогою системних викликів `select` і `poll`, які дозволяють для групи файлових дескрипторів дізнатися, які з них готові до читання/запису. Альтернативою `poll` є специфічні для окремих систем операції, які реалізовані більш ефективно, але не є портабельними: `epoll` в Linux, `kqueue` у FreeBSD та ін.

ZeroMQ (0MQ)

Розвитком парадигми сокетних з'єднань за рамки моделі взаємодії клієнт-сервер є технологія ZeroMQ, яка надає вдосконалений інтерфейс сокетів з підтримкою більшої кількості протоколів взаємодії, а також з підтримкою інших схем роботи:

- публікація-підписка (pub-sub)
- тягни-штовхай (push-pull)
- дилер-маршрутизатор (dealer-router)
- екскюзівная пара

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

- і нарешті, схема запит-відповідь (req-rep) — це класична клієнт-серверна схема з'єднання

Див. ZeroMQ - Super Sockets

RPC і мережеві архітектури

Розподілена програма використовує для взаємодії певний протокол, який також можна розглядати як інтерфейс виклику процедур віддалено (RPC — remote procedure call). Фактично, інтерфейс RPC реалізує прикладний рівень моделі OSI, але для його підтримки також необхідно в тій чи іншій мірі реалізувати протоколи рівня представлення і, іноді, сеансового рівня. Рівень представлення вирішує задачу передачі даних в рамках гетерогенної (тобто складеної з різних компонент) мережі в "зрозумілій" формі. Для цього потрібно враховувати такі аспекти, як старшинство байт (endianness), кодування для текстових даних, подання композитних даних (колекцій, структур) і т.д. Ще одним завданням RPC-рівня часто є знаходження сервісів (service discovery).

Реалізація RPC може бути заснована на власному (ad hoc) або ж якомусь із стандартних протоколів прикладного рівня і представлення. Наприклад, реалізація RPC по методолгіі REST використовує стандартні протоколи HTTP в якості транспортного і JSON/XML для представлення (сериалізації). XML/RPC або JSON/RPC — це ad hoc RPC, які використовує XML або JSON для представлення даних. Протоколи ASN.1 і Thrift — це бінарні протоколи, які визначають реалізацію всіх 3-х рівнів.

У форматів серіалізації існує 2 дихотомії: бінарні і текстові формати, а також статичні (що використовують схему) і динамічні (без схеми, schema-less).

Поширені формати серіалізації включають:

- JSON — текстовий динамічний формат
- XML — тестовий формат з опціональною схемою
- Protocol Buffers — бінарний статичний формат
- MessagePack — заснований на JSON бінарний формат
- Avro — заснований на JSON формат зі схемою
- EDN (extensible data notation) — текстовий динамічний формат

Мережеві програми можуть бути реалізовані у вигляді різних мережевих архітектур. Ключовим параметром для кожної архітектури є рівень централізації: від повністю централізованих — **клієнт-сервер** — до повністю децентралізованих — **peer-to-peer**/**P2P**. Важливими моделями між цими двома

крайнощами є модель сервісно-орієнтованої архітектури (**SOA**), а також модель **клієнт-черга-клієнт**.

Література

- Tour of the Black Holes of Computing: Network Programming
- Beej's Guide to Network Programming
- Socket System Calls
- How TCP backlog works in Linux
- TCP in 30 instructions
- Мультиплексирование ввода-вывода

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Безпека

Загальні принципи безпеки

Інформаційна безпека — це безперервний процес захисту інформаційних систем від погроз трьох видів:

- несанкціонований доступ (НСД) і використання
- порушення цілісності, конфіденційності, автентичності та інших характеристик даних
- порушення доступності та/або повноцінного функціонування інформаційної системи

Підсистема безпеки не є виділеним компонентом ОС і її практично неможливо додати в систему пост-фактум, тобто вона повинна бути вбудована із самого початку.

Принципи створення безпечної системи:

- принцип безпечних налаштувань за замовчуванням
- принцип валідації даних, що надходять від інших учасників системи
- принцип повної медації — завжди перевірка поточних прав
- принцип найменших привілеїв — видавати права, достатні для виконання тільки необхідних операцій і не більше того
- принцип поділу привілеїв — по можливості, вимагати згоди декількох учасників для виконання операцій
- принцип економії на механізмі — механізми захисту мають бути максимально простими з можливих і реалізовуватися на найнижчому з можливих рівні
- принцип мінімального спільного між різними учасниками системи
- прицип відкритого дизайну — архітектура, реалізація і використовувані алгоритми в системі повинні бути відомі, а в секреті можуть триматися тільки обмежені за обсягом авторизаційні дані (ключі, паролі і т.п.)
- принцип психологічної прийнятності

Основні сервіси системи безпеки описуються абревіатурою AAA:

- аутентифікація (Authentication) — встановлення "особистості" сторони, з якої відбувається взаємодія

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

- авторизація (Authorization) — перевірка прав на виконання будь-яких операцій у системі
- облік (Accounting) — облік операцій, пов'язаних з системою безпеки (для можливості подальшого розслідування та встановлення причин проблеми)

Способи (фактори) аутентифікації:

- за паролем
- за питанням безпеки
- за одноразовим паролем
- за жетоном (унікальним числом)
- за допомогою сертифіката ЕЦП

Аутентифікація може бути як однофакторноб, так і багатофакторною.

Механізми роботи системи безпеки

В системі безпеки розглядаються 3 сутності: учасники системи (суб'єкти, користувачі), ресурси та права доступу. Одиницею управління є **домен захисту** — це пара об'єкт і право доступу. Домен може відповідати одному суб'єкту або їх групі.

Матриця контролю доступу — це теоретична модель, яка описує матрицю, яка приводить у відповідність всі ресурси системи з усіма її суб'єктами. В комірках цієї матриці знаходяться права доступу конкретного користувача/ролі/групи до конкретного ресурсу. Така матриця дозволяє описати всі права доступу в системі, однак її практичне застосування не ефективно.

На практиці використовуються 2 наступних підходу:

- списки контролю доступу (Access Control Lists, ACL), які реалізують зберігання та облік прав на рівні ресурсів, тобто, фактично, по стовпцях цієї матриці
- мандатні системи (Capability або C-list), які реалізують зберігання прав доступу у суб'єктів системи, тобто по рядках матрицям

У системі, заснованій на ACL, для кожного ресурсу визначений список суб'єктів з їх правами. Наприклад, у ФС Unix в кожної директорії і файлу визначені 3 типи суб'єктів: користувач-власник, група-власник і всі інші,— а також 3 типи прав: читання, запис і виконання. Іншим прикладом ACL є список правил міжмережевого екрану, ресурсами в яких є хости/підмережі та/або можливість звернення до певних портів/використання певних протоколів. У такій системі

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

замість прав доступу встановлюються дії екрану при зверненні: дозволити, заборонити, обмежити і т.д. При цьому, враховуючи потенційну необмеженість різних суб'єктів-учасників мережі, в такій системі в основному використовуються узагальнені суб'єкти: всі хости, всі зовнішні хости, всі хости з певної підмережі і т.д.

У системах на основі мандатів мандат видається окремо для кожного суб'єкта на кожне право доступу. Мандат, як правило, реалізується як числовий жетон (token), який видається суб'єкту системою безпеки. Цей жетон може бути:

- просто унікальним числом, яке записується в базу даних для кортежу користувач, домен безпеки. Проблема такого способу в потенційно необмеженій кількості записів в системі з великою кількістю ресурсів та/або суб'єктів. У такій системі аутентифікованому суб'єкту достатньо надати жетон для того, щоб ідентифікувати ресурс, до якого він хоче отримати доступ, і отримати доступ
- криптографічного величиною, отриманою застосуванням односторонньої функції до кортежу, що включає домен безпеки і секретний ключ, відомий тільки ядру системи, `c = f(domain, key)`. У цьому випадку для перевірки мандата суб'єкт повинен надати не тільки сам жетон, але й ідентифікатор ресурсу і права доступу. Перевірка буде проводиться повторним обчисленням значення функції над тими ж аргументами. Безпека системи заснована на неможливості отримати те ж значення жетона без знання секретного ключа. У цій системі утруднений відкликання окремих мандатів, оскільки для відкликання мандата потрібно або змінити ідентифікатор ресурсу, що вплине на всіх суб'єктів, що мають до нього доступ, або змінити ключ, який, як правило, унікальний для користувача, але відкликання ключа призведе до відкликання усіх мандатів цього користувача. Для вирішення цієї проблеми використовуються т.зв. непрямі об'єкти.

Хоча з точки зору моделі Матриці прав доступу в обох системах зберігається одна і та ж інформація, ця модель описує тільки статичні характеристики системи і не описує її поведінки в динаміці.

При розгляді динаміки роботи системи безпеки на основі мандатів володіють наступними перевагами перед списками контролю доступу:

- відсутність необхідності використання загального простору імен ресурсів, відомого всім суб'єктам системи
- можливість більш гранулярного обліку прав: в системі на основі списку контролю доступу адміністраторам системи необхідно вичерпне знання

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

про всіх суб'єктів системи — оскільки це психологічно неприйнятно, суб'єкти зазвичай агрегуються більш загальними сутностями — **принципалами** безпеки

В той же час в мандатних системах важче вирішити наступні проблеми:

- обмежити передачу мандату від одного суб'єкта іншому (така можливість також може бути і корисною властивістю системи)
- відкликання мандатів, особливо вибіркове відкликання окремих прав, а не всіх мандатів для якогось суб'єкта

У багатьох реальних системах використовується комбінація обох підходів: наприклад, у файловій системі Unix ACL використовуються для первинного контролю прав, а для перевірки поточних прав використовуються мандат, який видається після первинної авторизації, перевірка якого набагато ефективніше.

Системи на основі мандата часто використовуються для створення т.зв. "пісочниць", наприклад для виконання коду, отриманого з недовірених джерел — оскільки в такій системі простіше реалізувати принцип "по-замовчуванню без доступу".

Реализація системи безпеки

Апаратна платформа надає такі базові механізми для підтримки системи безпеки:

- кільця процесора (CPU rings), які використовуються в сегментной організації пам'яті
- привілейовані інструкції (в деяких платформах)
- рандомізація адресного простору програми, захист стеку і т.п.

Використовуючи ці примітиви ОС вибудовує систему безпеки. Основа цієї системи в більшості ОС:

- виконання всіх критичних операцій в ядрі ОС та надання обмеженого інтерфейсу до них через механізм системних викликів
- разделение пользователей на администраторов (в Unix-системах: особый пользователь `root`) и обычных пользователей

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Література

- Secure Systems Design Principles
- Defensive Programming
- CWE/SANS Top 25 Most Dangerous Software Errors
- Capability Myths Demolished
- Classic Buffer Overflow Explained
- Priviledge Escalation Bug in Linux
- How to Exploit an XSS
- Server compromised due to publicly accessible Redis

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Unix

Історія Unix

UNIX з'явився в 1973 (почав розроблятися в 1969) в Bell Labs. Перша цільова платформа — мінікомп'ютери DEC (PDP-7).

В Unix були створені такі технології, як: мова C, оператор `pipe` (`|`) для взаємодії між процесами, інтерфейс сокетів BSD та багато інших.

UNIX спочатку був умовно відкритою системою, досить зручною для портування на інші архітектури. Тому досить швидко з'явилися різні гілки (варіанти) Unix'ів. Першою такою гілкою (fork'ом) став Берклевскій дистрибутив (BSD) в 1977 році. У той же час, ліцензія UNIX не давала можливості необмеженої зміни і модифікації системи, з чим були пов'язані багато юридичних конфліктів. У решті решт, на даний момент сформувалося кілька закритих комерційних версій Unix'а, кілька відкритих версій, а також ряд Unix-подібних систем, створених з нуля (насамперед, GNU/Linux).

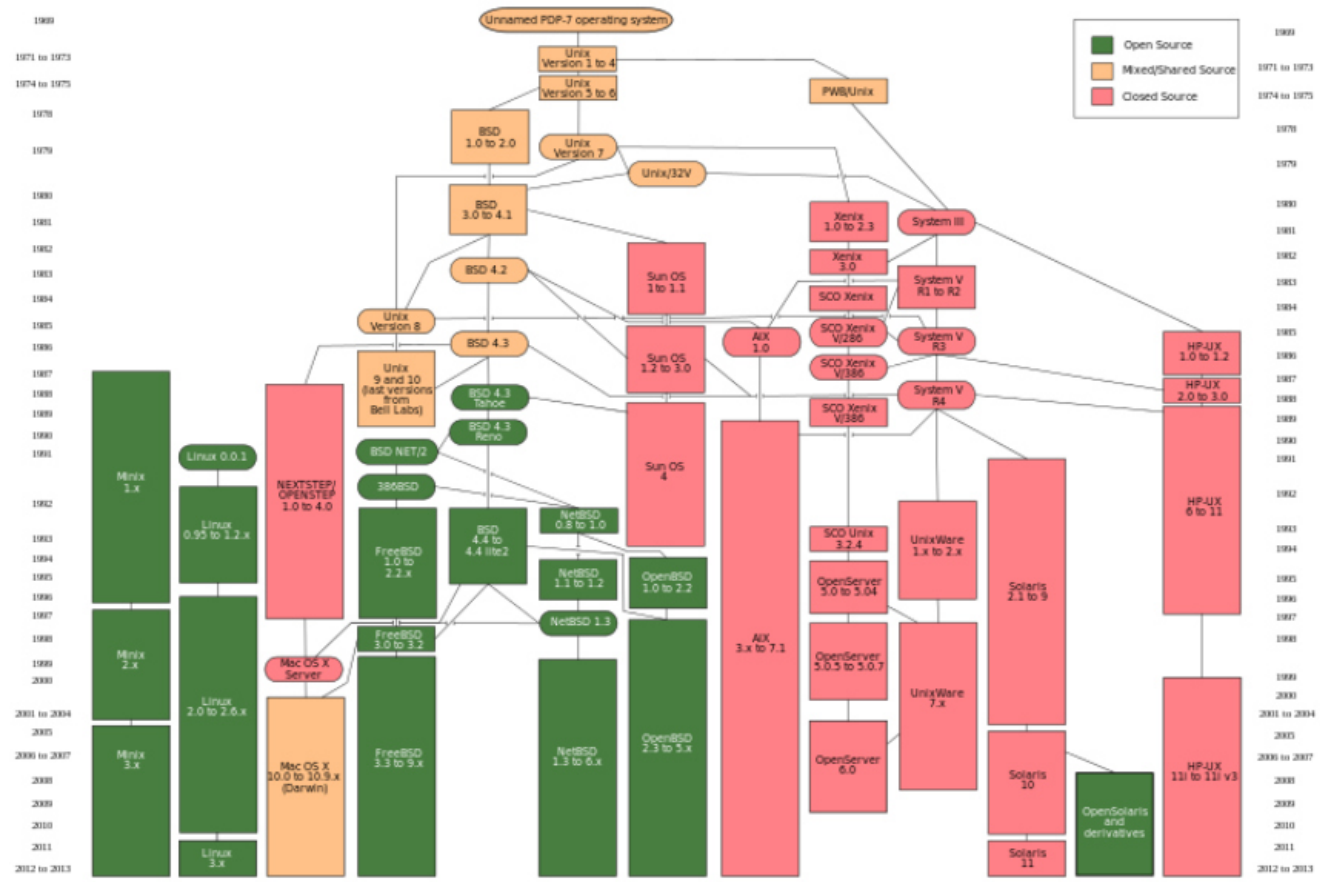

Рис. 10.1. Сімейство нащадків Unix і Unix-подібних ОС

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Основні варіації Unix'ів

Комерційні варіації

Комерційні версії Unix ведуть свій родовід від т.зв. Unix System V. Практично всі вони припинили свій розвиток. Основні представники:

- SCO UnixWare (припинений)
- Sun Solaris (припинений)
- IBM AIX
- HP-UX

На даний момент розвивається в якості ОС для дата-центрів тільки illumos — відкритий спадкоємець Sun Solars, який був найпросунутішої і активно змінюємою версією Unix'а.

Ключові технології illumos:

- ФС ZFS
- Технолония ізоляції та створення т.зв. пісочниць Solaris Zones
- Система інтроспекції DTrace
- Технологія віртуалізації Kernel Virtual Machine

Варіанти BSD

Берклевскій дистрибутив розвивався як альтернатива System V Unix і з моменту виходу в світ версії 4.3 Tahoe став повністю відкритим і безкоштовним. З дистрибутивами BSD пов'язана одна з найпоширеніших open-source ліцензій — ліцензія BSD. BSD-версії Unix продовжують активно розвиватися.

Представники BSD сімейства:

- FreeBSD
- OpenBSD
- NetBSD
- DragonflyBSD

На основі дистрибутива FreeBSD було створено ядро Darwin, яке використовується в MacOS X.

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

GNU/Linux

Linux — це Unix і не Unix — повністю нова система, створена на основі принципів Unix, як їх розумів Лінус Торвальдс. Поштовхом для написання системи стали поширення навчального варіанту Unix'а MINIX, створеного Таненбаумом, а також поява процесора Intel 386 з підтримкою плоскою сегментной моделі, що радикально спростило написання ядра ОС.

Linux обійшов інші варіанти Unix'а на хвилі open-source за рахунок використання напрацювань руху GNU (компілятор gcc, утиліти core-utils, редактор Emacs та ін.) та ліцензії GPL, яка вимагає відкриття доробок системи на тих же умовах, що і оригінальної системи.

Linux — це ядро, на основі якого створюється дистрибутив — набір системних утиліт та інших програм, необхідних для роботи системи, таких як графічна оболонка, офісні додатки і т.п. Існують десятки дистрибутивів GNU/Linux, серед яких найбільш поширені гілки Red Hat (Red Hat Enterprise Linux, Fedora, Suse, CentOS) і Debian (Debian, Ubuntu, Mint).

Специфічним дистрибутивом Linux є ОС Google Android.

Plan 9

Plan 9 була академічної спробою в рамках лабораторії Bell Labs створити спадкоємця Unix, в якому б були виправлені його основні недоліки. Вона виявилася класичним прикладом синдрому "другої системи", і не набула поширення.

Ключові рішення:

- Все — дійсно файл
- Окремі простори імен
- Скасування суперкористувача

Ключові рішення Unix

- відкрита система (man, POSIX, open source)
- файл-центричність і текст-центричність
  - "Small pieces, loosely joined"
  - розвинені засоби IPC
- ієрархічна файлова система з примітивною моделлю безпеки

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

- плоске API системних викликів, засноване на C
  - додаткові можливості заховані в ioctl, fnctl, і т.п.
- користувач root
- модель запуску процесів fork / exec
- розвинена система управління залежностями

Підтримка GUI в Unix

Історично ядро Unix розроблялося без підтримки графічного інтерфейсу, який на той момент ще не був розвинений. По міру розвитку різних варіантів графічної апаратної частини в Unix було створено рішення для підтримки і роботи з ними — протокол X11, розроблений робочою групою в університеті MIT.

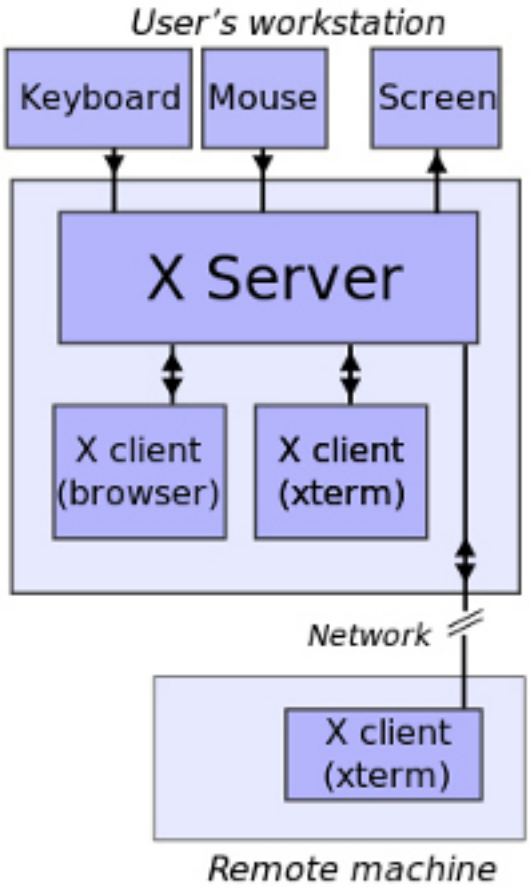

Рис. 10.2. Схема взаємодії в рамках протоколу X11

X11 протокол має різні реалізації:

- xfree86
- X.org

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

- Wayland

На базі X11 протоколу, як правило, будуються більш високоуровенвие графічні фреймворки (GUI toolkits). Найпоширеніші фреймворки в Unix-середовищі:

- GTK
- Qt

Віконні менеджери вирішують задачу управління інтерфейсом додатків в рамках єдиної концепції (на даний момент прийнята концепція Робочого столу або WIMP). Як правило, менеджери вікон будуються на основі графічних фреймворків. Наприклад, найпоширеніший менеджер Gnome використовує GTK, KDE побудований на основі Qt.

Інші менеджери вікон:

- звичайні: Xfce, Sawfish
- плиткові: ion3, XMonad

Управління залежностями

В умовах створення нових додатків з використанням створеної раніше бази програмних бібліотек, важливим і складним завданням, яке вирішує будь-який дистрибутив Unix (а також, на даний момент і середовища всіх мов програмування), є управління залежностями. Для цього (майже) кожен дистрибутив Unix має спеціальну програму — пактений менеджер, який відповідає за ведення бази існуючих бібліотек і їх версій, залежностей і сумісності між версіями, а також місць зберігання вихідного чи об'єктного коду, з яких можна завантажити ту чи іншу версію.

Розрізняють менеджери залежностей, які працюють на рівні вихідного коду (з його наступною збіркою) і на рівні бінарних файлів.

Найпоширеніші пакетні менеджери:

- менеджери на основі формату APT в Debian
- менеджери на основі формату RPM в Red Hat
- BSD ports і система Portage Gentoo Linux

Також, більшість середовищ мов програмування надають свій особливий менеджер пакетів. Наприклад:

- Maven в Java

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

- RubyGems в Ruby
- NPM в Node.js

Принципи розробки під Unix

У книзі "Мистецтво програмування під Unix" (TAOUP) Ерік Реймонд

сформулював наступні принципи хорошого тону, які переважно застосовуються при розробці як самих частин Unix систем, так і програм для Unix:

- Модульності (Modularity): створювати прості частини, що зв'язуються чистими інтерфейсами
- Ясності (Clarity): ясні рішення краще хитрих і розумних
- Композиції (Composition): створювати програми, які можуть бути пов'язаними з іншими програмами
- Поділу (Separation): розділяти політику і механізм, інтерфейс і реалізацію
- Простоти (Simplicity): придумувати найпростіше рішення, додавати складність тільки по необхідності
- Ощадливості (Parsimony): створювати великі програми, тільки коли продемонстровано, що інші не впораються
- Прозорості (Transparency): продумувати програми з тим, щоб зробити інтроспецкцію і відлагодження можливими і максимально простими
- Міцності (Robustness): міцність — це дитя прозорості та простоти
- Представлень (Representation): вкладати знання в дані, щоб логіка програми була тупою і надійною
- Найменшого подиву (Least Surprise): при проектуванні інтерфейсів завжди потрібно вибирати найменш несподіваний варіант
- Тиші (Silence): коли у програми немає нічого дивного, щоб повідомити, вона не має повідомляти нічого
- Полагодження (Repair): намагатися полагодити все, що можна, але коли це не вдається — падати, падати голосно і якомога раніше
- Економії (Economy): час програміста дорого коштує, його краще зберегти і витратити машинний час
- Генерації (Generation): по можливості уникати кодування, коли можна написати програму, яка буде писати інші програми
- Оптимізації (Optimization): прототипувати перед шліфуванням, домогтися роботи програми перед її оптимізацією
- Розмаїття (Diversity): не довіряти ніяким твердженнями про єдиний вірноий

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

шлях

- Розширюваності (Extensibility): проектувати на майбутнє — воно настане раніше, ніж ми думаємо

Критика Unix

Поряд з фанатами Unix існують і Unix-ненависники, які навіть написали власну книгу — Керівництво Unix-ненависників.

Ось деякі типові претензії до Unix систем:

- слабка підтримка GUI
- слабка підтримка сценаріїв роботи звичайного користувача (т.зв. Desktop сценарії)
- недостатнє дотримання моделі "все — файл"
- примітивна модель безпеки: простий ACL, користувач root
- примітивна файлова система
- додаткові можливості заховані в ioctl, fnctl, ...
- застаріла системна мова (C)

Література

- The Art of Unix Programming
- The UNIX-HATERS Handbook
- The Daemon, the GNU & the Penguin

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Windows

Історія Windows

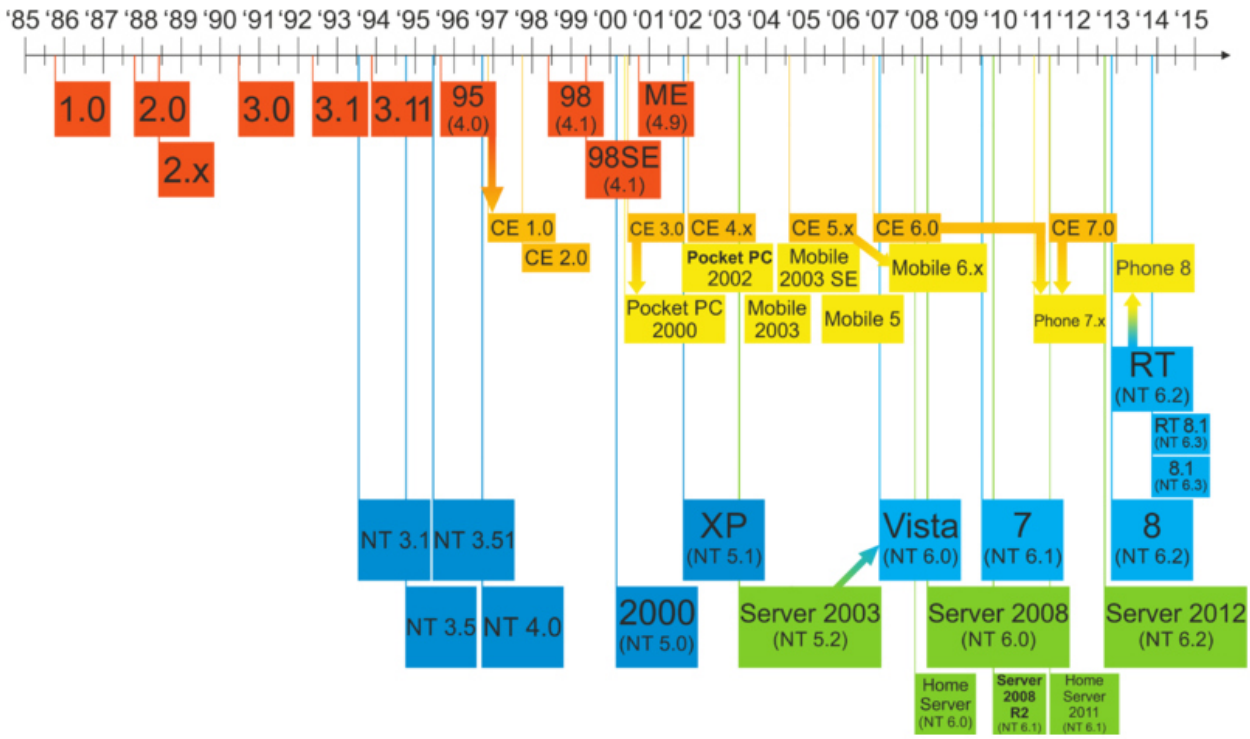

Рис. 11.1. Хронологія розвитку Windows

Windows — це сімейство ОС, яке розвивалося поступово, починаючи з переробки ОС CP/M в ОС MS-DOS. MS-DOS, як і CP/M, був примітивною однокористуацькою ОС для персональних комп'ютерів, які тільки з'являлися. Він був створений в рамках існуючих на той момент апаратних обмежень цих пристроїв (16-бітна архітектура, малі доступні ресурси, такі як потужність процесора, обсяги пам'яті і постійних сховищ). Спочатку ця система повинна була завантажуватися з дискет ємністю близько 750 Кб, працювати в текстовому графічному режимі і управляти пам'яттю до 1 МБ. Система Windows спочатку була графічною оболонкою над MS-DOS.

Разом з швидким розвитком можливостей ПК почала швидко розвиватися і Windows, незабаром впершись у обмеження своєї архітектури. Windows 95/98 стала 32-розрядною версією системи, яка ще не використала такі стандартні технології підтримки багатокористувацької роботи, як поділ на режим ядра і користувача, багатозадачність, ФС з поділом прав доступу і т.п.

Реалізацією сучасних концепцій ОС стало ядро Windows NT, яке послугувало

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

основою систем Windows 2000/XP/7/8 і Windows Server.

Windows NT

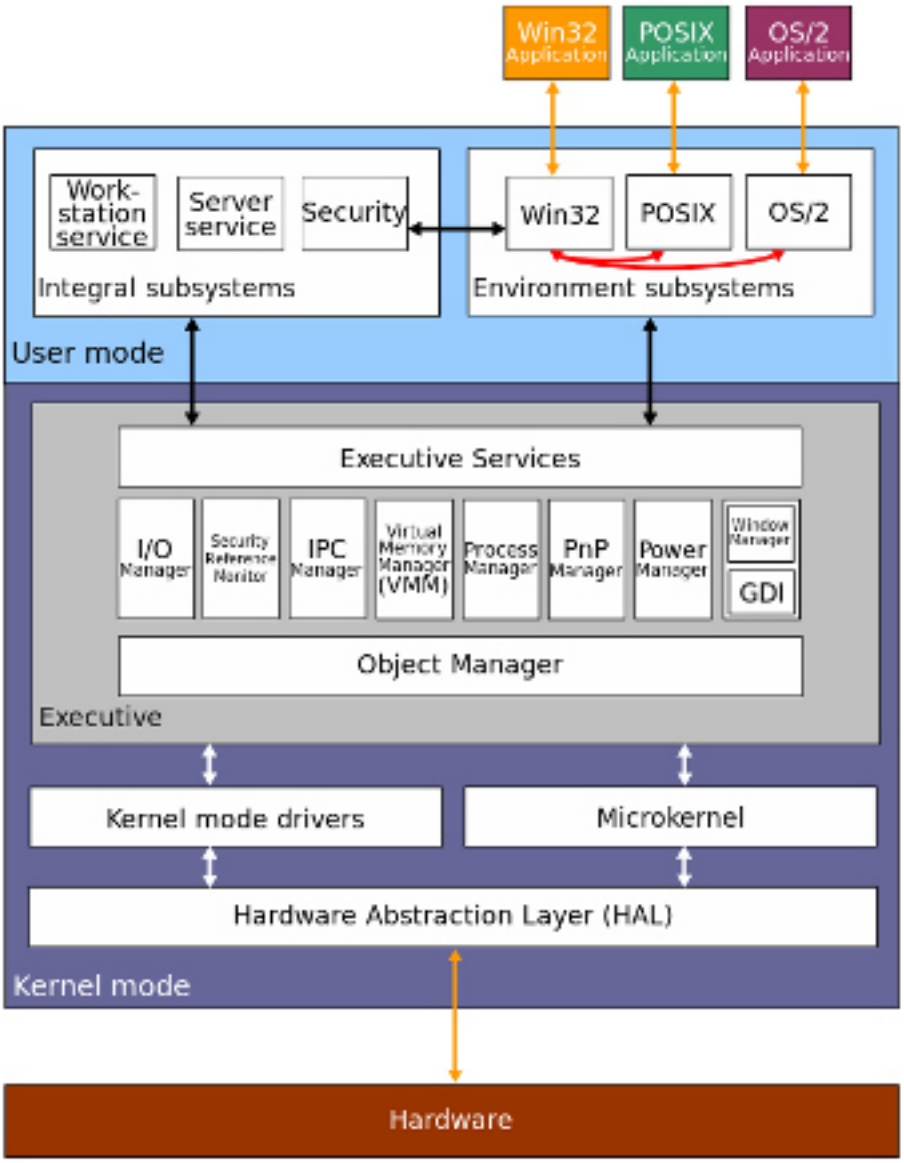

Рис. 11.2. Архітектура Windows NT

Windows NT — це назва сучасного ядра Windows.

Його основні характеристики:

- розділення на режим ядра і користувача
- гібридне ядро, що включає мікроядро, рівень абстракції пристроїв і драйверів (HAL) і сервіси ОС, що працюють в ядерному режимі (Executive), а також сервіси ОС, що працюють в користувацькому режимі: оточення (Environment) та інтеграції (Integral)

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

- підтримка витісняючої багатозадачності
- підтримка SMT/SMP
- виведення на основі пакетного та асинхронного режимів

Мікроядро виконує такі функції:

- синхронізація
- планування виконання ниток і обробки переривань
- перехоплення виключень
- ініціалізація драйверів при запуску системи

Виконавчі сервіси ОС (Executive) включають:

- введення-виведення (в т.ч. взаємодію з графічними пристроями через інтерфейс GDI)
- управління пристроями (в т.ч. живленням, підтримка Plug-n-play пристроїв)
- управління об'єктами ядерного режиму
- управління процесами і міжпроцесною взаємодією
- управління пам'яттю і кешами
- управління конфігурацією (через системний реєстр)
- безпека

Сервіси оточення ядра WinNT підтрумують 3 режими роботи:

- Win32-сумісний (з підтримкою MSDOS і Win16 додатків), що включає також віконний менеджер — сервіс csrss.exe
- Posix-сумісний
- OS/2-сумісний

Ключові рішення Windows

- прив'язка до архітектури x86 (Windows не підтримує інших архітектур, крім ARM)
- GUI-центричність
- Windows API
- управління конфігурацією через реєстр
- велика увага зворотній сумісності
- системна мова C++, потім C#

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting

Критика Windows

- закрита система
- проблеми з композицією програм
- проблеми з безпекою
- непридатність для багатьох сценаріїв роботи (насамперед, як високопродуктивної серверної ОС)

Література

- Mark Russinovich & David Solomon - Windows Internals
- The Infamous Windows "Hello World" Program
- MS-DOS: A Brief Introduction
- How Microsoft Lost the API War

* pdf generated on 2016-01-01 from markdown sources with SBCL 1.2.16 on Linux 3.16.0-38-generic with 3bmd, cl-pdf & cl-typesetting