Лекція 1

Середовище виконання програми

Апаратне середовище виконання

Комп’ютер надає програмі, яка скомпільована в машинний код, апаратне середовище виконання (hardware execution environment). Така програма виконується безпосередньо комп’ютером. Процесор виконує програму, яка складається з команд процесора та даних. Команди процесора працюють із регістрами процесора, звертаються до пам’яті комп’ютера, впливають на поточний стан пристроїв комп’ютера і т. ін. Тобто процесор виконує команди програми відповідно до специфікації набору команд архітектури комп’ютера.

Для використання Unix-подібної операційної системи (далі «Unix») на деякому комп’ютері необхідне підтримування з його боку такого: рівні привілеїв процесора, віртуальна пам’ять або сегментація пам’яті, апаратні переривання і таймер, програмні переривання або схожа команда процесора, виняткові ситуації. Теоретично можна частково портувати деяку реалізацію Unix на комп’ютер з архітектурою, яка не підтримує чогось із вищепереліченого, але використовувати таку систему для розв’язування реальних задач можна навряд чи.

Рівні привілеїв процесора (CPU privilege levels) – це обмеження, які команди може виконувати процесор. Рівень привілеїв процесора визначає які команди процесор може виконати та що конкретно дозволено робити командам. Для більшості реалізацій Unix достатньо двох рівнів привілеїв процесора: найвищий та найнижчий. Процесор може підтримувати кілька рівнів привілеїв, але зазвичай використовують тільки два з них. Наприклад, архітектура процесора x86 має чотири рівні привілеїв процесора.

Процесор використовує поточний рівень привілеїв (current privilege level, CPL) під час виконання кожної команди для визначення чи дозволено виконувати поточну команду та що конкретно їй дозволено робити. Якщо CPL найвищий, процесор може виконати будь-яку команду. Якщо CPL найнижчий, процесор може виконати тільки команди, які для системи є безпечними та не можуть вплинути на всю систему. Комп’ютер надає апаратні та програмні механізми для зміни CPL, інакше сенсу в рівнях привілеїв процесора не було б.

Рівні привілеїв процесора – це основа для реальної можливості використання віртуальної пам’яті, апаратних переривань, програмних переривань та виняткових ситуацій. Віртуальну пам’ять, апаратні переривання, програмні

переривання або виняткові ситуації можна реалізувати в комп'ютері, в якому не реалізовано рівні привілеїв процесора, але якщо можна несанкціоновано змінити дані в пам'яті комп'ютері, то сенсу в їхньому використанні нема.

Віртуальна пам'ять (virtual memory) – це технологія управління пам'яттю, в якій апаратура надає широкі можливості контролю виконання команд процесора, які працюють із пам'яттю. Віртуальна пам'ять складається з трьох компонентів:

1. Віртуальний адресний простір (virtual address space) – це множина адрес пам'яті, які можуть бути залучені в командах процесора, які працюють із пам'яттю програми. Ці адреси називають віртуальні адреси.
2. Фізичний адресний простір (physical address space) – це множина адрес оперативної пам'яті (random-access memory, RAM, фізичної пам'яті, physical memory). Ці адреси називають фізичні адреси.
3. Механізм відображення (mapping mechanism) для з'єднання поточного віртуального адресного простору (простір пам'яті, який не відповідає пам'яті жодного фізичного пристрою комп'ютера, це абстракція) з фізичним адресним простором (оперативна пам'ять, яка є фізичним пристроєм

комп’ютера). MMU транслює віртуальні адреси у фізичні адреси, використовуючи механізм відображення, визначений архітектурою комп’ютера, та машинозалежні дані, які визначають поточний віртуальний адресний простір. Механізми відображення можуть бути різними. Якщо транслювання віртуальної адреси не вдалося, буде згенеровано відповідну виняткову ситуацію (пояснено далі).

Зазвичай віртуальний адресний простір та фізичний адресний простір є плоскими адресними просторами, або їх можна привести до плоских адресних просторів.

Для відображення віртуального адресного простору у фізичний адресний простір потрібно, щоб у відповідному пристрої комп’ютера були дані, які визначають відображення. Що це за пристрій та які там мають бути дані залежить від механізму відображення, який використовується в конкретній архітектурі комп’ютера.

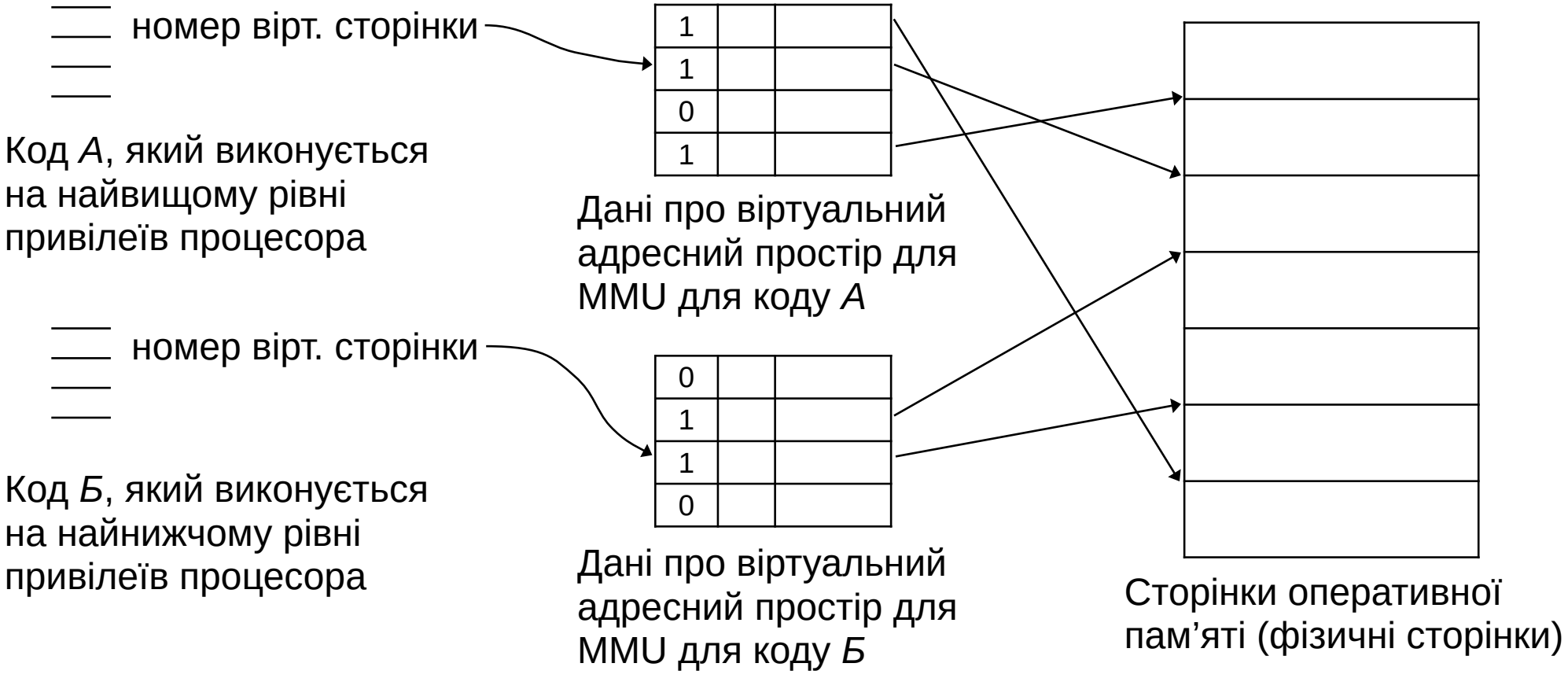
Віртуальний та фізичний адресні простори зазвичай складаються з віртуальних та фізичних сторінок відповідно (сторінка – це масив байт), а такий варіант

віртуальної пам’яті називають сторінкова пам’ять (paged memory). Адреса початку сторінки повинна бути вирівнена на її розмір. Машинозалежні дані, які визначають віртуальний адресний простір, містять властивості віртуальних сторінок, ці властивості називають атрибути віртуальної сторінки. Найбільш важливі атрибути віртуальної сторінки такі: біт присутності, якщо він встановлений, віртуальна сторінка відображена у фізичну сторінку; біт права модифікації вмісту фізичної сторінки; біт модифікації вмісту фізичної сторінки; номер фізичної сторінки, в яку відображена віртуальна сторінка.

Усі фізичні сторінки в комп’ютері мають однаковий розмір. Віртуальна сторінка може бути відображена в одну фізичну сторінку (звичайна віртуальна сторінка) або в масив із кількох фізичних сторінок (суперсторінка), тому віртуальні сторінки в комп’ютері можуть мати різні розміри. Розміри віртуальних сторінок та розмір фізичної сторінки визначає архітектура комп’ютера. Наприклад, в архітектурі процесора x86_64 розмір фізичної сторінки дорівнює 4 Кб, розміри віртуальних сторінок дорівнюють 4 Кб, 2 Мб та 1 Гб. Віртуальна адреса складається з номера віртуальної сторінки та зміщення у віртуальній сторінці.

Комп’ютер надає можливість змінювати дані про відображення, так можна змінювати атрибути віртуальних сторінок та змінювати дані про поточний віртуальний адресний простір. Коду, який виконується на найнижчому рівні привілеїв процесора, не дозволено змінювати поточний віртуальний адресний простір та зазвичай не дозволено або не надано можливість змінювати дані про відображення його власного віртуального адресного простору (не дозволено на апаратному рівні). Віртуальна пам’ять надає розробникам програмного забезпечення абстракцію, використовуючи яку можна виконувати код програми в його власному віртуальному адресному просторі. Тобто програма не може звернутися до віртуальних адрес іншого віртуального адресного простору.

Деякі Unix можна використовувати на комп’ютерах, де реалізована сегментація пам’яті (memory segmentation), яка є варіантом віртуальної пам’яті. Сегментація пам’яті має менший функціонал ніж сторінкова пам’ять.



Код *А* може змінити будь-які дані, які визначають відображення, та змінити поточний віртуальний адресний простір. Код *Б* таке робити не може (це заборонено на апаратному рівні або йому не надана можливість таке робити).

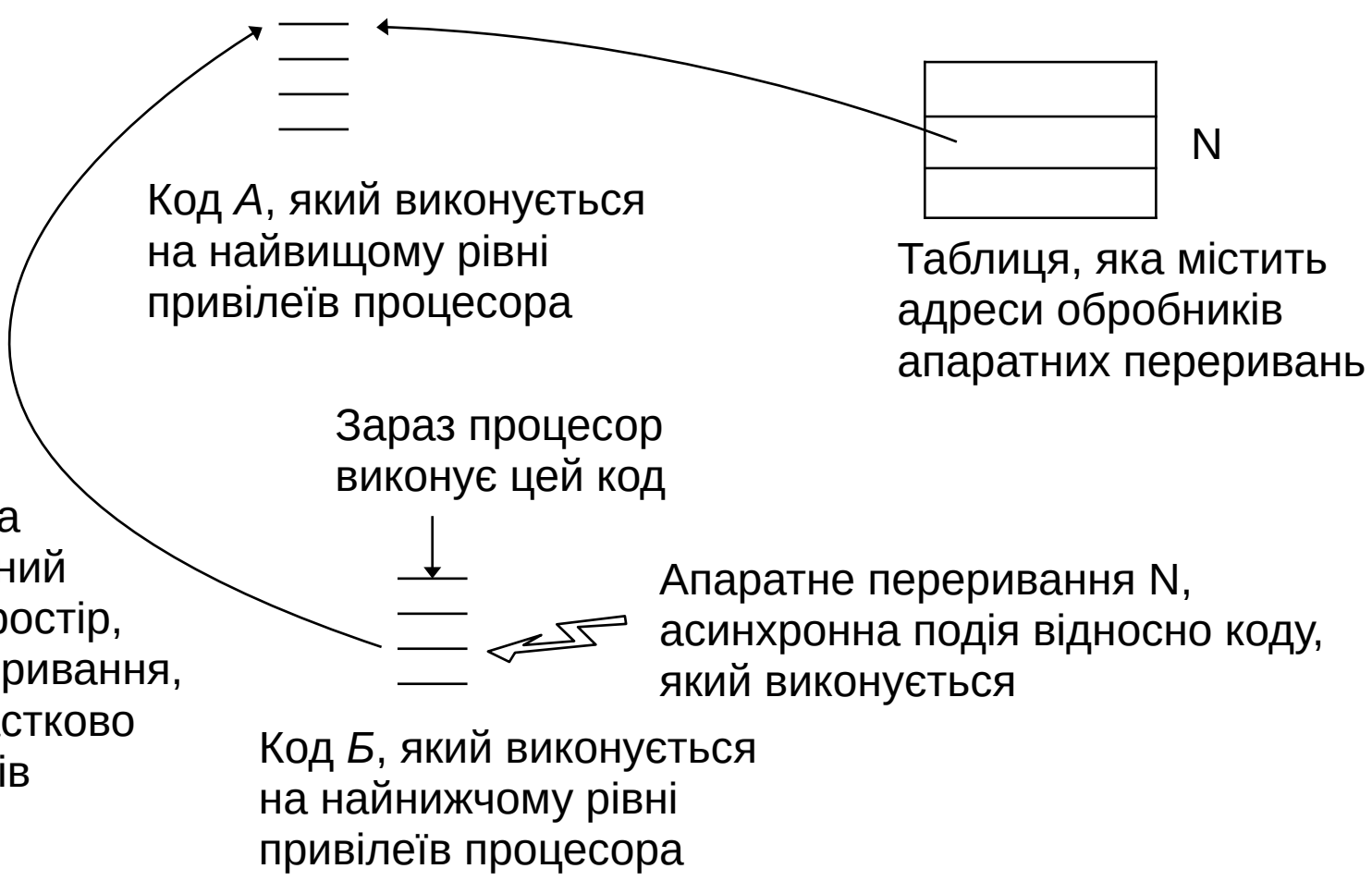
Апаратне переривання (hardware interrupt) – це спосіб, за допомогою якого пристрій комп’ютера інформує систему про зміну свого стану. Апаратні переривання пронумеровані, кожний пристрій зазвичай має власний номер апаратного переривання.

У разі генерації апаратного переривання комп’ютер змінює CPL на найвищий, змінює поточний віртуальний адресний простір (якщо CPL є найнижчий), маскує (блокує) усі апаратні переривання, зберігає повністю або частково поточні значення регістрів процесора та починає виконувати код відповідного обробника апаратного переривання. Зазвичай є можливість зареєструвати різні обробники апаратних переривань для всіх можливих апаратних переривань у комп’ютері. Приклади дій, які призводять до зміни стану в пристрої: користувач натиснув клавішу на клавіатурі й клавіатура згенерувала апаратне переривання, мережева карта відправила дані в мережу і згенерувала апаратне переривання.

Для того, щоб використовувати Unix, комп’ютер повинен мати пристрій, який називають таймер (timer) або годинник (clock). Цей пристрій можна запрограмувати на генерацію апаратних переривань через фіксовані проміжки

часу або на генерацію одного апаратного переривання після завершення вказаного проміжку часу.

Апаратне переривання є асинхронною подією відносно коду, який виконується (генерація апаратного переривання не має відношення до цього коду).



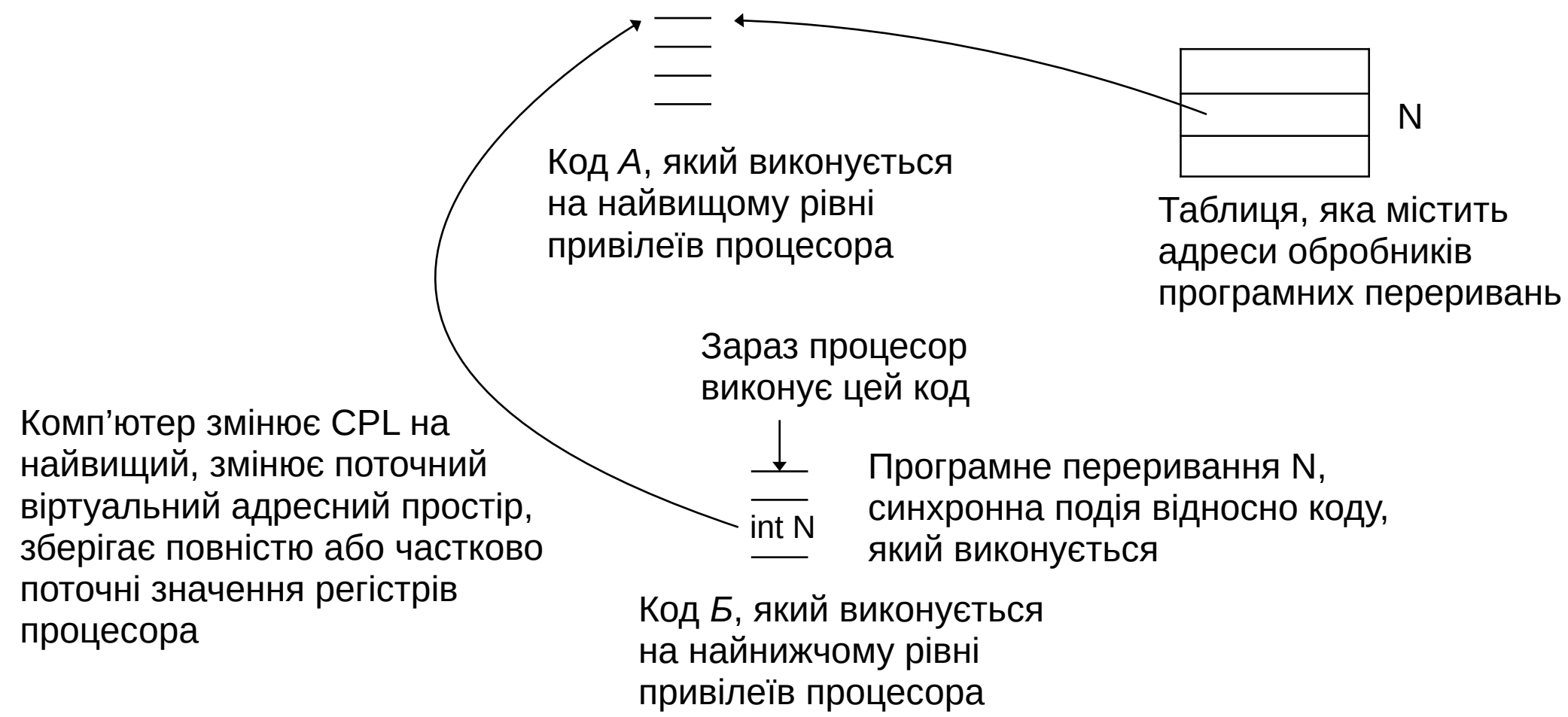
Схематичне пояснення виклику обробника апаратного переривання в разі генерації апаратного переривання під час виконання коду на найнижчому рівні привілеїв процесора.

Програмне переривання (software interrupt) – це команда процесора, виконуючи яку комп’ютер змінює CPL на найвищий, змінює поточний віртуальний адресний простір (якщо CPL є найнижчий), зберігає повністю або частково поточні значення регістрів процесора та починає виконувати код відповідного обробника програмного переривання. Комп’ютер також може маскувати всі переривання під час виклику обробника програмного переривання.

Програмне переривання зазвичай використовують у коді, який виконується на найнижчому рівні привілеїв процесора, для виклику обробника програмного переривання в іншому віртуальному адресному просторі з найвищим рівнем привілеїв процесора.

Замість програмного переривання процесор може мати схожу команду з меншим часом виконання.

Програмне переривання є синхронною подією відносно коду, який виконується (команда програмного переривання присутня безпосередньо в цьому коді).

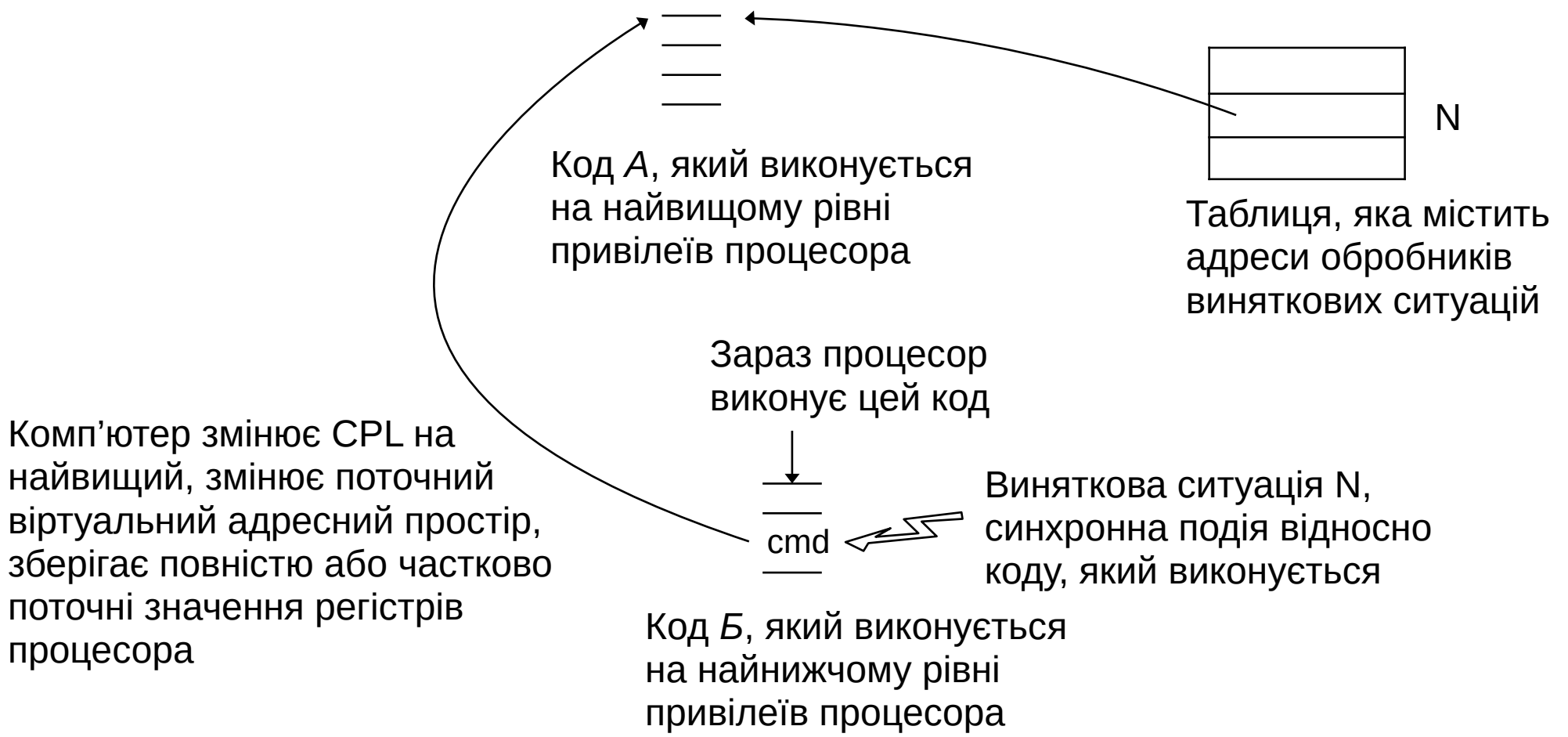


Схематичне пояснення виклику обробника програмного переривання під час виконання коду на найнижчому рівні привілеїв процесора.

Виняткова ситуація (exception) генерується, якщо процесор не може виконати поточну команду. Виняткові ситуації пронумеровані, кожна причина неможливості виконати поточну команду процесором має власний номер.

У разі генерації виняткової ситуації комп’ютер змінює CPL на найвищий, змінює поточний віртуальний адресний простір (якщо CPL є найнижчий), зберігає повністю або частково поточні значення регістрів процесора та починає виконувати код відповідного обробника виняткової ситуації. Зазвичай є можливість зареєструвати різні обробники виняткових ситуацій для всіх можливих виняткових ситуацій у комп’ютері. Приклади дій, які призводять до генерації виняткових ситуацій: ділення на нуль, звернення до не відображеної віртуальної сторінки.

Виняткова ситуація є синхронною подією відносно коду, який виконується (процесор не зміг виконати команду в цьому коді).



Схематичне пояснення виклику обробника виняткової ситуації в разі генерації виняткової ситуації під час виконання коду на найнижчому рівні привілеїв процесора.

Адреси обробників апаратних переривань, програмних переривань, виняткових ситуацій та інформація про віртуальний адресний простір, на який треба змінити поточний віртуальний адресний простір, вказують у машинозалежних структурах даних в оперативній пам’яті. Формати цих структур даних визначає архітектура комп’ютера. Наприклад, в архітектурі процесора x86 – це регістри процесора та таблиці. Коду, який виконується на найнижчому рівні привілеїв процесора, не дозволено на апаратному рівні або не надано можливість змінювати ці машинозалежні структури даних.

Програмне середовище виконання

Unix налаштовує апаратне середовище виконання та надає програмне середовище виконання (software execution environment) програмі, яка скомпільована в машинний код. Таку програму називають програма користувача (user program). Програма користувача не взаємодіє з апаратурою комп'ютера безпосередньо, вона послуговується програмним середовищем виконання, яке надає операційна система (ОС), для взаємодії з апаратурою та програмними абстракціями ОС.

Насправді програма користувача, яка скомпільована в машинний код, усе ж таки взаємодіє з апаратурою безпосередньо. Процесор виконує її код, тому можна сказати, що програма користувача використовує процесор безпосередньо. Під час виконання команд процесор звертається до пам'яті, тому програма користувача також використовує пам'ять безпосередньо. Можливо програма користувача використовує ще іншу апаратуру (залежить від системи). Якби було необхідно повністю відокремити виконання програми користувача від апаратури, тоді її код треба було б інтерпретувати, але тоді має бути програма, яка буде виконувати цю інтерпретацію. Тобто отримуємо

рекурсію, яка призводить до того, що повинна бути програма, яка має бути скомпільованою в машинний код.

ОС складається з таких компонентів:

1. Ядро (kernel) створює та підтримує програмне середовище виконання та налаштовує апаратне середовище виконання для себе та програм користувача. Ядро – це найскладніша частина ОС.
2. Системні програми допомагають створювати програмне середовище виконання для програм користувача (можуть не використовуватися). Також до системних програм належать базові (стандартні) програми з дистрибутиву ОС.
3. Системні бібліотеки допомагають програмам користувача взаємодіяти з ядром або з апаратурою (можуть не використовуватися). Іноді наданий варіант взаємодії в системній бібліотеці кращий за будь-який варіант, який програма користувача може реалізувати сама.

Усі компоненти ОС, які перелічені вище, є програмами. Системні програми є також програмами користувача, але такі, які допомагають створювати програмне середовище виконання для програм користувача, та не потребують інших системних програм. Системні бібліотеки не є окремими програмами, але їхній код використовується в системних програмах та у звичайних програмах користувача. Ядро також є програмою, але ця програма відрізняється від будь-якої програми користувача, оскільки ядро створює програмне середовище виконання та налаштовує апаратне середовище виконання для себе та програм користувача.

Можна виокремити такі завдання ядра:

1. Абстрагування програмних та апаратних інтерфейсів для програм користувача.
2. Робота з периферійними пристроями.
3. Підтримування виконання програм користувача та самого ядра.
4. Управління пам’яттю.

5. Підтримування файлових систем.
6. Підтримування різних механізмів міжпроцесної взаємодії.
7. Підтримування мереж.
8. Безпека.

Код ядра Unix виконується на найвищому рівні привілеїв процесора, система працює в режимі ядра (kernel mode). Це дає змогу ядру використати будь-яку команду процесора для налаштування середовища виконання для себе та програм користувача та для виконання будь-яких дій на комп’ютері.

Код програми користувача виконується на найнижчому рівні привілеїв процесора, система працює в режимі користувача (user mode). Це дає змогу обмежити дії програми користувача на апаратному рівні, оскільки коли процесор виконує якусь команду, тоді тільки він може контролювати свої дії.

Режим ядра та режим користувача – це режими виконання (execution mode) у системі.

Раніше були надані компоненти ОС. Але також можна сказати, що ОС – це тільки ядро, оскільки комп’ютер, в якому виконується тільки код ядра, уже може вирішувати реальні завдання. Наприклад, мережеві інтерфейси можна сконфігурувати в опціях ядра під час його завантаження і ядро може виконувати завдання маршрутизації мережевого трафіку. Жодної програми користувача для виконання цього завдання не потрібно.

Адресний простір ядра називають простір ядра (kernel space). Адресний простір процесу користувача називають простір користувача (user space). Комбінацію режим користувача та простір користувача називають userland. Іноді під термінами простір користувача та userland розуміють одне й те ж саме. Код ядра може виконуватися в кількох адресних просторах (пам’ять ядра та пам’ять програми користувача). Код програми користувача виконується в його власному адресному просторі (пам’ять програми користувача).

Процес – це сутність (entity), яка виконує програму та надає середовище виконання для неї. Програма може бути програмою користувача або частиною ядра (функція ядра), тому є процеси користувача (user processes) та процеси ядра (kernel processes). Ядро для кожного процесу має об’єкт у своїй пам’яті,

який називають *структура процесу* (*process structure*). Ця структура містить машинонезалежну інформацію про процес та покажчик на машинозалежну інформацію про процес.

Процес, який виконується безпосередньо в цей момент часу, називається поточний процес (current process) або процес, що виконується (running process).

Комбінація найнижчого рівня привілеїв процесора та окремого адресного простору під час виконання коду програми користувача дає змогу: ізолювати всю систему та інші процеси користувача від несанкціонованого впливу на них із боку поточного процесу, ізолювати поточний процес від несанкціонованого впливу на нього з боку інших процесів користувача. Ця ізоляція потрібна, оскільки процес користувача може випадково або навмисно спробувати вплинути на всю систему або інші процеси через використання відповідних команд процесора.

Обробники апаратних переривань, програмних переривань та виняткових ситуацій є частинами коду (функціями) ядра, оскільки опрацювання цих подій у

комп’ютері виконуються з найвищим рівнем привілеїв процесора та пов’язані з даними ядра.

Якщо код програми користувача спробує виконати недозволену на апаратному рівні дію, буде згенеровано відповідну виняткову ситуацію і якщо її обробник не зможе її вирішити (resolve), ядро завершить цей процес користувача. Якщо код ядра спробує виконати недозволену на апаратному рівні дію (таке також може бути в режимі ядра), буде згенеровано відповідну виняткову ситуацію і якщо її обробник не зможе її вирішити, ядро зупинить своє виконання або перезавантажить комп’ютер. На відміну від процесу користувача, помилка в якому є локальною для нього, помилка в коді ядра може впливати на само ядро, на будь-який процес користувача та на всю систему.

Програма користувача може мати один або кілька контекстів виконання, які логічно виконуються одночасно (паралельно), але завжди має один адресний простір. Один контекст виконання в програмі користувача називають потік (thread). Програму користувача з кількома контекстами виконання називають багатопотокова програма (multithreaded program). Частиною контексту виконання є апаратний контекст, який складається зі значень регістрів

процесора (регістри загального призначення, регістри з рухомою комою та спеціальні регістри).

Підтримування багатопотокових програм користувача може бути реалізоване в різні способи, але спосіб підтримування є прозорим для багатопотокової програми користувача. Ці способи зазвичай мають назви режим N:1, режим 1:1, режим N:M. Перше число в назвах цих режимів визначає виконання скількох потоків користувача (потоки відомі в режимі користувача) мультиплексується в скількох об’єктах планування (scheduling entity) ядра. Ядро планує виконання своїх об’єктів планування, тобто воно визначає виконання якого об’єкта планування продовжити (resume), виконання якого об’єкта планування призупинити (suspend), виконання якого об’єкта планування зупинити (stop), виконання якого об’єкта планування заблокувати (block). Відповідно режим N:1 реалізований повністю в режимі користувача (об’єкт планування в ядрі – це процес, виконання потоків користувача мультиплексується асиметрично), режим 1:1 реалізований повністю в режимі ядра (об’єкт планування в ядрі – це потік ядра, виконання потоків користувача не мультиплексується), режим N:M реалізований у режимі ядра та режимі користувача (об’єкт планування в ядрі –

це потік ядра, виконання потоків користувача мультиплексується симетрично або асиметрично).

Далі в тексті використано слово «процес», оскільки конкретне ядро може не підтримувати потоки (ядро може підтримувати тільки один контекст виконання в одному процесі). Якщо конкретне ядро підтримує потоки, то під словом «процес» треба розуміти слово «потік».

Зазвичай процес користувача не може виконати жодного реального завдання використовуючи виключно команди процесора, які йому доступні на найнижчому рівні привілеїв процесора. Процесу користувача може бути потрібно прочитати дані з файлу, відправити дані в мережу, створити новий процес, змінити образ процесу, отримати додаткову пам’ять у свій адресний простір і т. ін. Такі дії можна виконати тільки в режимі ядра, тому програмі користувача потрібно запитувати виконання цих дій в ядра, тому ядро надає програмі користувача механізм виклику необхідних функцій ядра. Ці функції ядра називають системні виклики (system call, syscall).

Для переходу в режим ядра програма користувача використовує спеціальну команду процесора, виконуючи яку комп’ютер змінює CPL на найвищий, змінює поточний віртуальний адресний простір на віртуальний адресний простір ядра, зберігає повністю або частково поточні значення регістрів процесора та починає виконувати раніше вказаний код ядра (функцію). Це такі саме дії, які виконуються в разі генерації виняткової ситуації.

Для переходу в режим ядра використовують команду програмного переривання або спеціально призначену для цього команду з меншим часом виконання. Нема принципової різниці як процес користувача викликає функцію ядра.

Набір системних викликів та формат їхніх аргументів визначено в API (application programming interface) між програмою користувача та ядром. Набір системних викликів у кожного ядра свій, але Unix-подібні ОС зазвичай використовують однаковий набір «стандартних» системних викликів («стандартних», оскільки офіційно нема такого стандарту, який визначає які системні виклики мають бути реалізовані, але практика показує, що де-факто він є). Зазвичай аргументи системного виклику – це числа або покажчики.

Зазвичай програма користувача не викликає системні виклики безпосередньо. ОС надає системну бібліотеку, використовуючи яку кожний системний виклик можна викликати за допомогою виклику відповідної функції бібліотеки (звичайний локальний виклик функції), яка в разі помилки повертає наперед визначене значення, яке позначає помилку та встановлює номер помилки в `errno` або повертає номер помилки.

Стандарти *Portable Operating System Interface* (POSIX) визначають функції та їхні аргументи, які мають бути реалізовані в POSIX-сумісній системі. Стандарти POSIX є частиною стандарту *Single UNIX Specification* (SUS). Стандарту SUS призначають номер версії (часто використовують скорочення SUSv3, SUSv4). POSIX не визначає яка функція має бути реалізована як системний виклик, а яка – як функція бібліотеки. Ось переклади цитат з SUSv4: «Різниця між “системним викликом” і “функцією бібліотеки” є деталлю реалізації, яка може відрізнятися в різних реалізаціях, тому була виключена з POSIX.1» та «Інтерфейс, а Не Реалізація – POSIX.1-2017 визначає інтерфейс, а не реалізацію. Не робиться різниці між функціями бібліотеки та системними

викликами; обидві названі функціями. Ніяких подробиць реалізації будь-якої функції не наведено».

Не всі ОС мають реалізацію всіх або точні реалізації всіх функцій, які визначені в POSIX. Це треба враховувати під час розроблення програми, яку будуть виконувати на різних системах (переносний вихідний код, portable source code). Тобто система може бути POSIX-майже-сумісною.

У лекціях більшість функцій з POSIX названі системними викликами, оскільки зазвичай вони реалізовані як системні виклики в більшості Unix-подібних ОС або їхні виклики завжди призводять до викликів відповідних системних викликів без використання додаткових програмних абстракцій.

Системний виклик, для виконання якого завжди достатньо запитаного ресурсу, називають швидким (fast), інакше його називають повільним (slow). Під ресурсом тут розуміють будь-який можливий ресурс, крім оперативної пам’яті (наявність вільної пам’яті) та об’єктів механізмів синхронізації (можливість заволодіти об’єктом механізму синхронізації). Тут слова «швидкий» та «повільний» не визначають час виконання системного виклику. Виконання

повільного системного виклику може бути заблоковано ядром через нестачу запитаного ресурсу для його виконання і це блокування можливо перервати сигналом. Якщо для завершення повільного системного виклику достатньо запитаного ресурсу, ядро виконує його якнайшвидше. Тобто швидкості виконання повільного системного виклику та швидкого системного виклику нічім не відрізняються, як і швидкість виконання будь-якого іншого коду.

Таке визначення типу системного виклику залежить як від системного виклику, так і від його аргументів. Приклад. Системний виклик `read()` є повільний, якщо його застосовано для сокета (ядро може не мати достатньо даних, отриманих із мережевого з'єднання, для його завершення). Цей системний виклик є швидкий, якщо його застосовано для звичайного файлу, навіть якщо цей файл належить мережевій ФС. Тому виконання швидкого системного виклику також може бути заблоковано ядром через нестачу запитаного ресурсу для його завершення, але це блокування неможливо перервати сигналом.

Unix створює таке програмне середовище виконання та налаштовує так апаратне середовище, щоб «одночасно» виконувати кілька процесів. Кількість процесів готових до роботи може бути більша за кількість наявних процесорів у

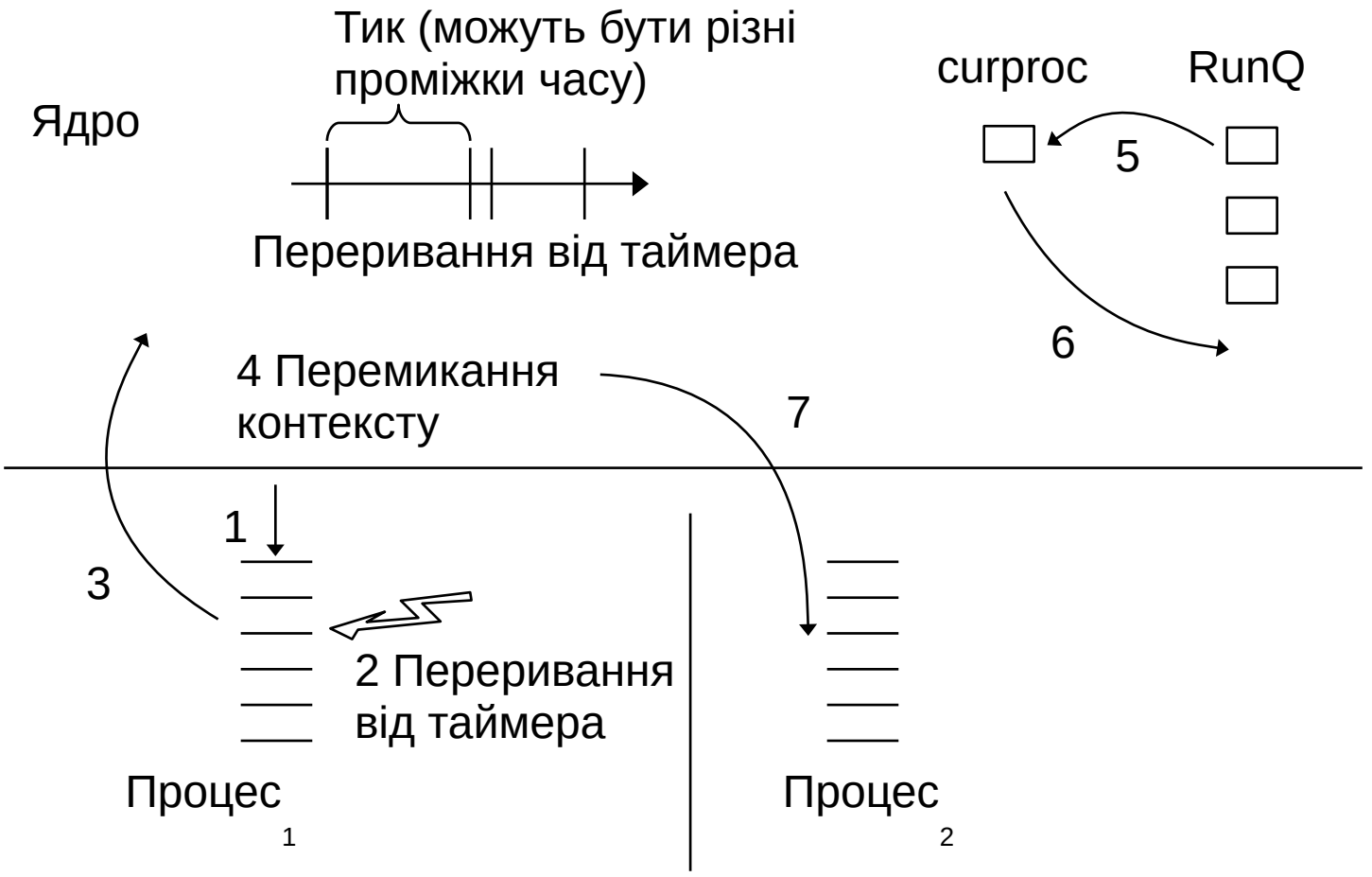
комп’ютері, тому виконувати всі готові до роботи процеси на такому комп’ютері фізично неможливо. Тому Unix імітує одночасне виконання процесів через надання почергово кожному готовому до роботи процесу невеликої частки (дещиці) часу для його виконання, яку називають квант часу (time quantum). Значення кванта часу вибирають таким, щоб користувач не відчув порушення інтерактивності системи (наприклад, час відгуку процесу користувача на зовнішній вплив) і всі готові до роботи процеси постійно виконувалися. Після завершення кванта часу, ядро призупиняє виконання поточного процесу і поновлює виконання іншого готового до роботи процесу. Цю дію називають примусове перемикання контексту (involuntary context switching), тобто примусове перемикання з контексту поточного процесу на контекст іншого готового до роботи процесу.

Готовий до роботи процес (ready-to-run) – це процес, для виконання якого є всі ресурси, він чекає на процесорний час, його виконання призупинено. Структури процесів готових до роботи процесів містяться в *черзі готових до роботи процесів* або *Run queue* (RunQ).

Для реалізації примусового перемикання контексту ядро використовує апаратне переривання від таймера. Апаратне переривання від таймера дають змогу викликати код відповідного обробника через певний проміжок часу. Цей проміжок часу називають тик (tick), це значення менше або дорівнює значенню кванта часу. Обробник переривань від таймера підтримує поточне значення часу й після обробки переривання ядро приймає рішення про примусове перемикання контексту на інший готовий до роботи процес. Переривання від таймера – це гарантія для ядра, що процесор виконуватиме код ядра через вказаний проміжок часу.

Під час виклику обробника переривання від таймера, як і під час виклику обробника будь-якого іншого апаратного переривання, комп’ютер зберігає повністю або частково поточні значення регістрів процесора, далі обробник переривання зберігає поточні значення решти регістрів процесора (зазвичай поточні значення регістрів процесора будуть збережені в стеку віртуального адресного простору, на який відбувається перемикання під час отримання апаратного переривання). Для того, щоб примусово перемикнути контекст на інший готовий до роботи процес треба завантажити в процесор значення

регістрів процесора цього іншого процесу, змінити поточний віртуальний адресний простір та змінити CPL (якщо необхідно). Зазвичай усі ці дії виконують за допомогою відповідної команди процесора.



Процесор почергово виконує кожний готовий до роботи процес. Користувачу здається, що ці процеси виконуються одночасно.

Для продовження виконання процесу може не вистачати деякого ресурсу. Процес може очікувати на наявність ресурсу у два способи.

Програма може очікувати на наявність ресурсу активно. Активне очікування – це очікування, в якому процесор постійно перевіряє в циклі наявність ресурсу. У такому очікуванні процесор навантажено і відповідний пристрій, в якому перевіряють наявність ресурсу, навантажено також (наприклад, процесор перевіряє якесь значення в пам’яті). Активне очікування значить, що програма інтенсивно використовує процесор, до моменту його звільнення самою програмою або до примусового перемикання контексту на іншій готовий до роботи процес.

Програма може очікувати на наявність ресурсу пасивно. Пасивне очікування – це очікування, в якому процесор не використовують для постійного опитування наявності ресурсу в циклі. Програма повідомляє систему, що вона потребує якийсь ресурс та звільняє процесор. Інша програма має зробити цей ресурс доступним та повідомити систему про наявність ресурсу для того, щоб система продовжила виконання програми, яка очікує наявності цього ресурсу.

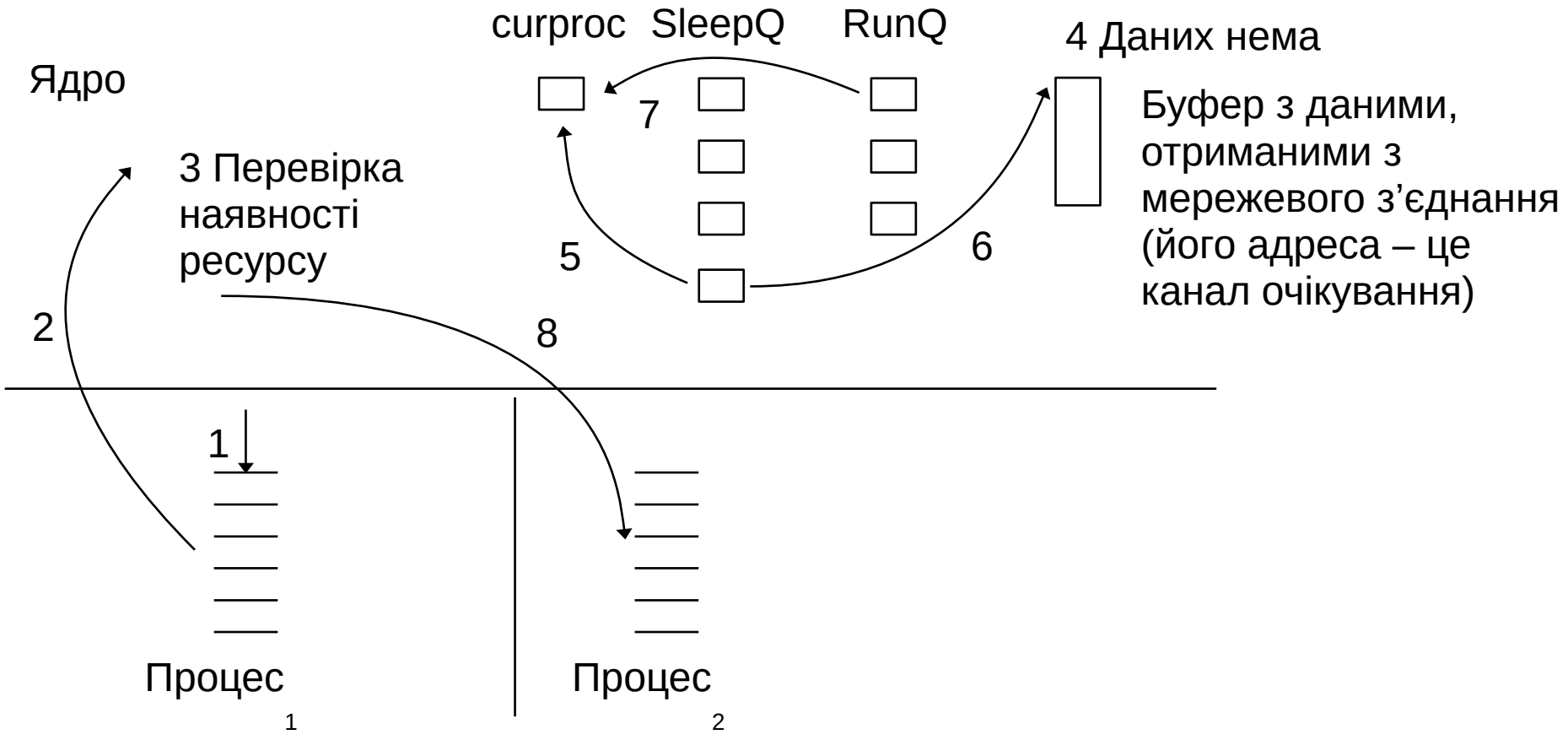
Звільнення процесора під час пасивного очікування наявності ресурсу називають добровільне перемикання контексту (voluntary context switching), тобто добровільне перемикання з контексту поточного процесу, якому не вистачає необхідного ресурсу для продовження його виконання, на контекст іншого готового до роботи процесу. Перемикання контексту на інший готовий до роботи процес можливо тільки в режимі ядра, тому наявність ресурсу в разі добровільного перемикання контексту перевіряє ядро. Процес, для якого було виконано добровільне перемикання контексту, називають заблокований (blocked) або сплячий (asleep). Цей процес отримає процесорний час тільки тоді, коли очікуваний ресурс буде наявним. Структури процесів заблокованих процесів містяться в *черзі сплячих процесів* або *Sleep queue* (SleepQ).

Ядро під час блокування процесу реєструє причину його блокування, тобто на наявність якого ресурсу очікує процес. Це потрібно для того, щоб у разі появи ресурсу ядро могло визначити чи є процеси (кілька процесів можуть очікувати на наявність одного й того ж ресурсу), які чекають на наявність цього ресурсу.

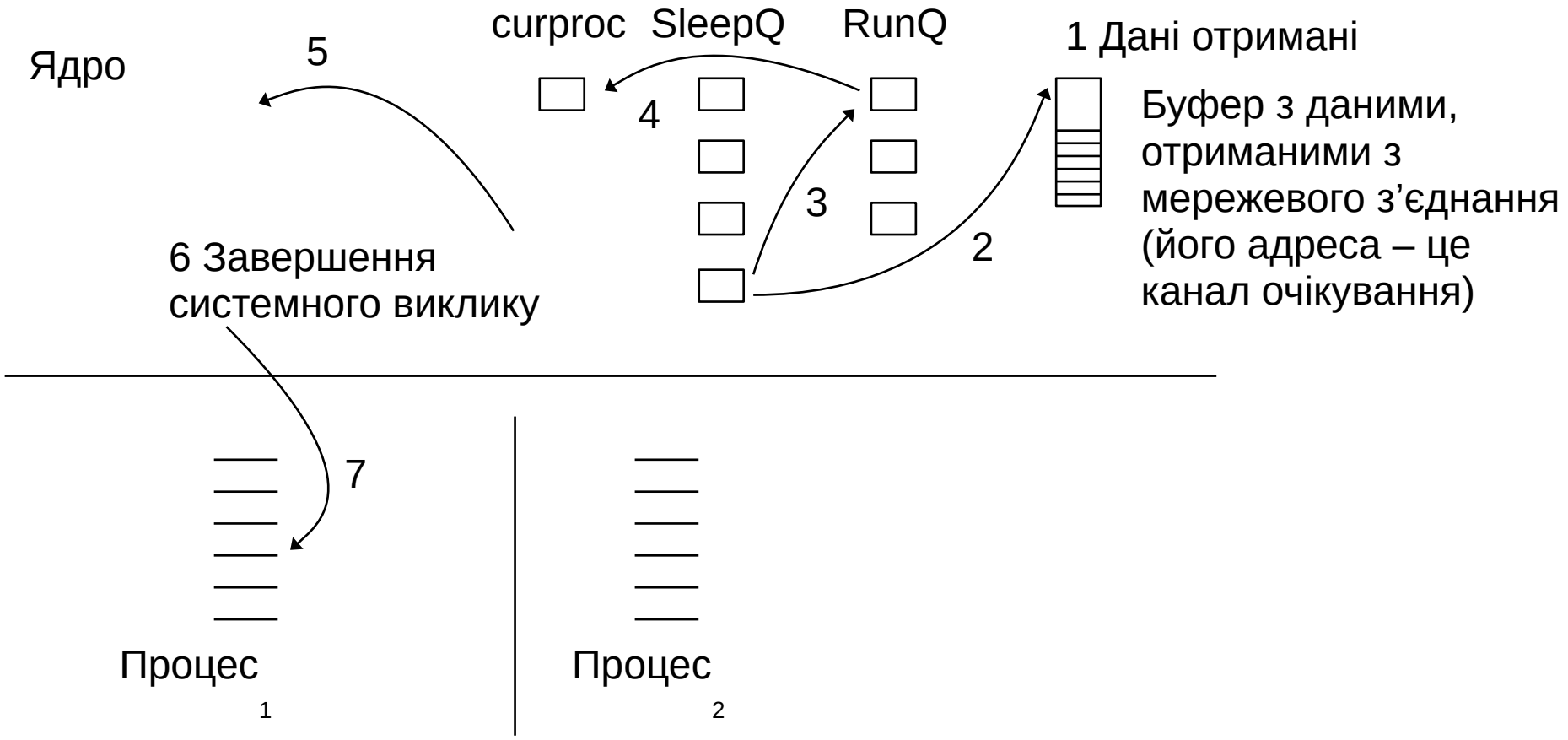
Добровільне перемикання контексту відбувається в разі пасивного очікування будь-якого ресурсу, не тільки ресурсу, який запитано в повільному системному

виклику. Тому навіть під час виконання швидкого системного виклику ядро може виконати добровільне перемикання контексту. Добровільне перемикання контексту також може відбутися під час опрацювання виняткової ситуації.

Код ядра, який відповідає за вибір готового до роботи процесу для перемикання контексту, називають планувальник (scheduler). Насправді це кілька функцій ядра.



Перший процес викликає системний виклик `read()` для мережевого з’єднання. Даних у буфері з даними, отриманими з мережевого з’єднання, нема. Ядро виконує перемикання контексту на готовий до роботи другий процес.



У разі отримання ресурсу, ядро пробуджує заблокований процес (переміщує його структуру процесу з SleepQ до RunQ), коли відбудеться перемикання контексту на цей процес (позначено числом 5), він продовжить виконувати системний виклик `read()`.

Кодування даних

Мережеве програмування майже завжди пов’язано з мережевою комунікацією. Якщо мережева комунікація відбувається між програмами, які виконуються на комп’ютерах із різною архітектурою, виникає необхідність кодування даних, які відправляють через мережеве з’єднання. Це кодування даних має дати змогу програмі, яка виконується в довільному середовищі виконання, однозначно інтерпретувати отримані дані, інакше сенсу в такій мережевій комунікації взагалі нема. Кодування даних не повинно залежати від мови програмування, оскільки програми, які використовують мережеву комунікацію, можуть бути написані різними мовами програмування. Кодування даних потрібно не тільки для мережевої комунікації, воно також потрібно для представлення вмісту звичайного файлу, з яким може працювати програма, яка виконується на комп’ютері з довільною архітектурою.

Є два принципово різних способи кодування даних. Перший спосіб полягає в кодуванні даних у текстове представлення. Це текстове представлення може складатися із символів ASCII та/або символів Unicode. Таке кодування завжди однозначне та може бути декодовано програмою, яка виконується на

комп’ютері з довільною архітектурою. Протокол рівня застосунку може використовувати власні або стандартизовані текстові представлення даних. Другий спосіб полягає в кодуванні даних у бінарне представлення. Бінарне представлення має займати менший розмір, ніж текстове представлення тих самих даних. Парсинг бінарного представлення має бути швидшим, ніж парсинг текстового представлення тих самих даних.

Інформація в мережевих протоколах складається з октетів. Октет (octet) – це 8-ми бітове беззнакове цілочисельне значення, відповідно можна кодувати 256 різних значень в одному октеті. Програми працюють із даними, які складаються з байтів. Байт у програмі – це найменша комірка пам’яті, яку може адресувати програма. Байт і октет не є тотожними термінами, оскільки байт не обов’язково складається з 8 бітів. Тому треба визначити яким може бути байт, оскільки будь-які дані (у тому числі дані, отримані через мережеве з’єднання), з яким працює програма, складаються з байтів.

Стандарти мови програмування C мають макрос `CHAR_BIT`, визначений у `<limits.h>`, стандарти мови програмування C++ мають макрос `CHAR_BIT`, визначений у `<climits>` (файл, який визначає такі самі макроси як у `<limits.h>` у C). Значення `CHAR_BIT`

дорівнює кількості бітів в одному байті, це значення повинно бути принаймні 8. POSIX визначає, що байт складається з 8-ми біт, а значення `CHAR_BIT` дорівнює 8. Якщо ОС є POSIX-сумісною, то байт у ній складається з 8-ми біт. Розмір типу `char` у C та C++ дорівнює 1 байту за визначенням. Оскільки тип `char` може бути знаковим або беззнаковим, треба використовувати типи `signed char` та `unsigned char` за необхідності. Ширина цих типів 8 біт, тобто в представленнях значень цих типів нема padding bits (біти, які програма не використовує та не інтерпретує), є тільки value bits (біти зі значеннями) та sign bit (знаковий біт) для знакового типу.

Порядок байтів (byte order, endianness) – це метод запису байтів (правило розташування байтів у послідовності байтів), які містять відповідні біти, у представленні багатобайтового значення в пам’яті або в комунікації даних. Мережевий порядок байтів (network byte order) визначено так: багатобайтове цілочисельне значення складається з октетів, перший октет значення (октет із найменшою адресою в пам’яті) містить значення старших бітів, наступні октети містять значення наступних бітів підряд. Мережевий порядок байтів називають *big-endian*, зворотний порядок байтів називають *little-endian*. Порядок байтів у

системі (host byte order, переклад українською неточний), може бути різний, він може бути big-endian, little-endian, або якимось іншим, який називають *middle-endian* або *mixed-endian* (це загальна назва для будь-якого іншого порядку байтів, тобто бітів, які містяться в цих байтах). Є системи, які дають змогу вибрати один з наперед визначених порядків байтів (зазвичай big-endian або little-endian), їх називають *bi-endian*. Слово «мережевий» у терміні мережевий порядок байтів не значить, що в мережевому протоколі обов'язково треба використовувати цей порядок байтів.

POSIX визначає, що біти можуть бути розподілені між байтами цілочисельного багатобайтового значення у довільному порядку (можуть бути розподілені не в очевидному порядку). Октет у пам'яті або в комунікації даних має деякий визначений порядок бітів. Програма не може адресувати біт, тому порядок бітів у байті для програми не важливий. Якщо десь потрібно конвертувати порядок бітів в октеті, то це конвертування виконується прозоро для програми. Відповідно порядок байтів визначає розподілення бітів між байтами багатобайтового цілочисельного значення.

Стандарти мов програмування C та C++ визначають обов’язкові мінімальні діапазони значень типів `unsigned short`, `unsigned int`, `unsigned long` та `unsigned long long`, які має підтримувати компілятор, але дозволяють реалізовувати ширші діапазони значень для цих типів. Тому розміри цих беззнакових цілочисельних типів може бути різними. У представленні значень цих типів можуть бути padding bits. Через це та через можливе використання різних порядків байтів у значеннях цих типів, представлення значень цих типів використовувати безпосередньо в мережевій комунікації в загальному випадку неможливо (передане представлення значення неможливо декодувати, якщо невідомий формат цього представлення).

Значення типів `unsigned short`, `unsigned int`, `unsigned long` та `unsigned long long` у C та C++ можна кодувати та декодувати байт за байтом у визначеному порядку, використовуючи дані типу `unsigned char` для кожного байта значення, вказуючи кількість байт у значенні. Якщо програма, яка декодуватиме ці дані, не має можливості зберігати декодоване значення в об’єкті наявного беззнакового цілочисельного типу, то в цій програмі треба реалізувати підтримування великих беззнакових цілочисельних типів (великих для неї). Кодування повинно

бути реалізовано за допомогою оператора побітового зсуву праворуч або відповідного для отримання значень складових байтів оригінального значення. Декодування повинно бути реалізовано за допомогою оператора побітового зсуву ліворуч або відповідного для отримання оригінального значення зі значень складових байтів. Використання операторів побітового зсуву або відповідних дає змогу працювати зі значеннями складових байтів значення, а не зі значеннями складових байтів представлення значення. Якщо програма працює з байтами цілочисельного багатобайтового значення за допомогою покажчиків на `unsigned char`, то це є помилкою, оскільки програма буде працювати зі значеннями байтів представлення значення, а формат цього представлення програма не знає в загальному випадку.

Стандарти мов програмування C та C++ визначають опціональні типи `uint`*`N`*`_t` (*`N`* може бути `8`, `16`, `32`, `64`). Ширина цих типів *`N`* бітів, тобто в представленнях значень цих типів нема padding bits. Тип `uint8_t` має такі самі характеристики як тип `unsigned char`. Значення цих типів використовувати в мережевих протоколах можна безпосередньо, якщо вказати порядок байтів багатобайтових значень.

Залежно від версій стандартів мов програмування C та C++ у представленні від'ємного значення цілочисельного типу може бути використаний (якщо не вказано інше) прямий код (signed magnitude), обернений код (ones' complement), доповняльний код (two's complement). Найбільш розповсюджений спосіб представлення від'ємних цілочисельних значень у сучасних системах є доповняльний код. Через це програми, які некоректно працюють з бітам значень знакових цілочисельних типів складно налагоджувати.

Стандарти мов програмування C та C++ визначають характеристики типів `short`, `int`, `long` та `long long`. Ідея цих характеристик така сама як для відповідних беззнакових типів, але ще визначені способи представлення від'ємних значень.

Стандарти мов програмування C та C++ визначають опціональні типи `intN_t` (*N* може бути `8`, `16`, `32`, `64`). Ширина цих типів *N* бітів, тобто в представленнях значень цих типів нема padding bits. Від'ємні значення представлені доповняльним кодом. POSIX-сумісна система повинна підтримувати типи `intN_t` та `uintN_t` для всіх *N* з множини `8`, `16`, `32`.

Стандарти мов програмування C та C++ визначають обов'язковий мінімальний діапазон значень для типів `signed char` та `unsigned char`, але через вимогу POSIX до значення макросу `CHAR_BIT` ширина цих типів заздалегідь відома.

Функції `htonl()` та `htons()` конвертують представлення беззнакового цілочисельного значення з порядку байтів у системі в мережевий порядок байтів. Функції `ntohl()` та `ntohs()` конвертують представлення беззнакового цілочисельного значення з мережевого порядку байтів у порядок байтів у системі.

```c
#include <arpa/inet.h>

uint32_t htonl(uint32_t host32);
uint16_t htons(uint16_t host16);
uint32_t ntohl(uint32_t net32);
uint16_t ntohs(uint16_t net16);
```

Літера `h` позначає «host», літера `n` позначає «network», літера `s` позначає «short», літера `l` позначає «long» у назвах цих функцій. Літери `s` та `l` у назвах цих функцій є історичними та мають відношення до розмірів відповідних типів даних у деякій системі.

Представлення значення типу `uint8_t` не потребує конвертації, воно має однакове представлення в мережевому порядку байтів та в порядку байтів у системі.

Ще є такий варіант кодування представлень беззнакових багатобайтових цілих значень. Якщо дві програми мають однаковий спосіб представлення цих значень, то кодування цих значень не потрібне, інакше ці значення кодують. Тому програми повідомляють одна одній про свої способи представлення значень. Якщо треба кодувати та декодувати велику кількість значень, то така оптимізація може підвищити продуктивність програм.

Найпростіший спосіб представлення знакового цілочисельного значення – це прямий код (абсолютне значення числа зі знаковим бітом у старшому біті).

Відправлення вмісту структури C або POD структури в C++ (закодованого належним чином) має сенс тільки тоді, коли ця структура упакована, інакше в структурі можуть бути padding bytes (байти, які програма не використовує та не інтерпретує).