Міністерство освіти і науки України
Рівненський державний гуманітарний університет

ЛЯШУК Т. Г.

Операційні системи та системне програмування.

Ч.1. Багатозадачність: управління процесами

Навчальний посібник

Рівне
Видавець О. Зень
2024

УДК 004.4 (075.3)
Л 99

Розглянуто та рекомендовано до друку Вченою радою
Рівненського державного гуманітарного університету
(протокол № 7 від 26 червня 2024 р.)

Рецензенти:
Джунь Й.В. – доктор фізико-математичних наук, професор, завідувач кафедри математичного моделювання Міжнародного економіко-гуманітарного університету імені академіка Степана Дем'янчука;
Войтович І.С. – доктор педагогічних наук, професор, завідувач кафедри інформаційно-комунікаційних технологій та методики викладання інформатики Рівненського державного гуманітарного університету.

Ляшук Т.Г.
Операційні системи та системне програмування. Ч.1. Багатозадачність: управління процесами [навчальний посібник] / Т.Г. Ляшук. – Рівне : О. Зень, 2024. – 164 с.

ISBN 978-617-601-498-0

В посібнику викладено основні принципи побудови системного програмного забезпечення з точки зору багатозадачності. Тематикою даного рукопису слугує питання програмної реалізації процесів, як основного поняття будь-якої операційної системи. Розглядаються питання керування процесами, доступ до їх властивостей, механізми міжпроцесної взаємодії та інші суміжні теми.

При цьому, основний акцент приділяється практичній складовій. Наводиться велика кількість програмних лістингів, які демонструють застосування засобів та методів багатозадачності для вирішення конкретних завдань.

Навчальний посібник призначений для студентів, які навчаються за спеціальностями представленими галуззю знань «Інформаційні технології», а також може бути корисним для викладачів та спеціалістів в області інформатики, кібернетики, обчислювальної техніки, які пов'язані з розробкою багатозадачного системного програмного забезпечення.

ISBN 978-617-601-498-0

ЗМІСТ

ВСТУП

Відомо, що програмне забезпечення (ПЗ) класифікується на прикладне (ППЗ) та системне (СПЗ). Перше із них відповідає за повсякденну взаємодію користувача із таким ПЗ, в той час, як СПЗ прямої взаємодії з користувачем впродовж тривалого часу не вимагає. ППЗ вирішує конкретні проблеми користувача, а СПЗ являється тим компонентом, який забезпечує інфраструктуру, на якій зазвичай працюють прикладні програми. З ППЗ користувач зустрічається повсюди: браузери, медіа-програвачі, офісні пакети тощо, проте, під «капотом» в них знаходиться дещо те, що приховане від рядового користувача – ПЗ системного характеру, яке власне і як було сказано, забезпечує роботу ППЗ.

СПЗ носить двояких характер: з однієї сторони – являється системоорієнтованим на прикладному рівні, тобто підлаштовується під конкретні програмні платформи користувача – операційні системи (ОС), а з іншого – здійснює керування апаратною складовою електронно-обчислювальної машини (ЕОМ). Перша складова забезпечує роботу всіх основних життєвоважливих функцій ОС: керування пам'яттю (внутрішня/зовнішня), засоби введення/виведення, багато(задачність/поточність), мережеві можливості тощо. Друга ж складова відповідає за надання доступу користувача до периферійних пристроїв ЕОМ: принтери, веб-камери, звукові карти тощо.

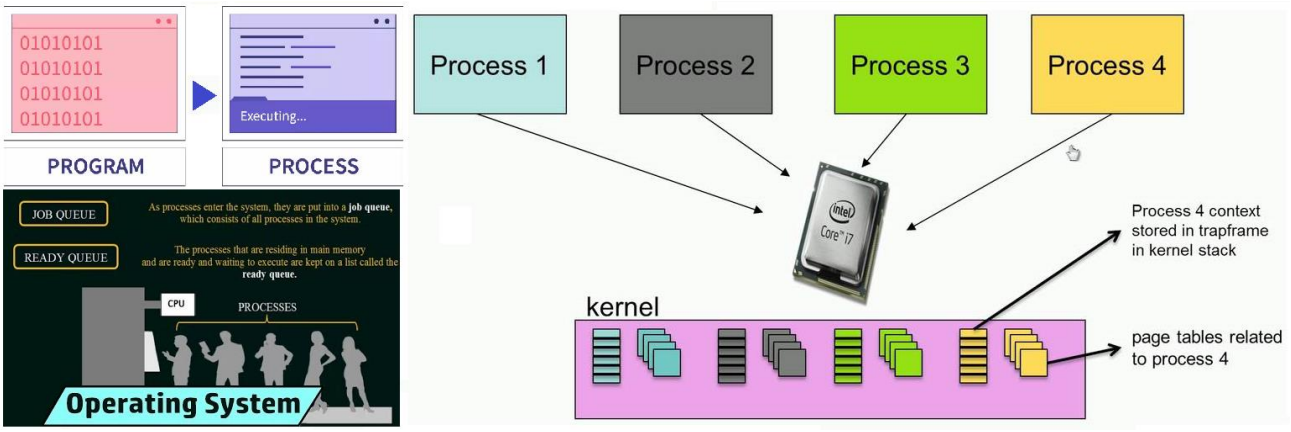
Прикладами СПЗ є такі програми як: ОС, драйвери, файлові менеджери, архіватори, різноманітні утиліти тощо. Проте, варто зауважити те, що чіткої межі між ППЗ і СПЗ не існує, оскільки будь-яке ППЗ використовує так чи інакше системний підхід створення ПЗ. Так, браузери використовують доступ до мережевих властивостей ОС, медіа-плеєри забезпечують роботу зі звуком, більшість сучасних програм є багатопоточними і т.п.

Зважаючи на вище сказане, потреба в спеціалістах з написання СПЗ є значною, в контексті того, що так чи інакше кожному прикладному програмісту

доводиться працювати з елементами системного управління. Тому, кожен ІТ-спеціаліст повинен принаймні розумітися в базових принципах побудови СПЗ.

При цьому, як дещо вище було сказано, одним із напрямків системного програмування являється робота з процесами, які являються чи не найголовнішими структурними елементами, що піддаються управлінню з боку ОС. Так, всім відома системна утиліта Task Manager, яка являє собою програму по забезпеченню моніторингу роботи Windows NT. Важливою функціональною складовою такої утиліти являється доступ до запуску нових та керування вже запущеними в системі процесами. Дана програма є незамінним інструментарієм не лише для системного адміністратора, але й для рядового користувача. Проте, доволі таки часто, існує необхідність щодо власної реалізації системного програмного продукту, який забезпечував би керування процесами на боці оператора ЕОМ. Зважаючи на вище сказане і враховуючи те, що тематика системного програмування є досить широкою та обширною, ціллю даного посібника слугує ознайомлення читачів з методами та прийомами програмування процесів, в контексті їх багатозадачної інфраструктури, та їх розширеної моделі – завдань. В навчальному виданні розглядаються такі теми як створення, маніпуляція, знищення процесів, міжпроцесна взаємодія та інші елементи управління процесами. Також висвітлена тема щодо програмної реалізації завдань (в тому числі і фонових), як у синхронному так і у асинхронному режимах.

РОЗДІЛ 1. ПРОЦЕСИ



1.1. Теоретична частина

1.1.1. Загальні положення

Одним із фундаментальних понять, яким оперує теорія ОС являється процес. Всім відомо, що на ЕОМ може бути інстальована велика кількість як прикладних так і системних програм, одні з яких «чекають» доки користувач їх запустить, інші – запускаються системою автоматично. Як тільки відбудеться їх запуск, весь програмник код таких програм, буде завантажуватися з файлової системи в основну пам’ять комп’ютера. При цьому, центральний процесор (ЦП) буде крок за кроком, динамічно виконувати їх інструкції, маніпулюючи даними, які прописані в програмному коді, задіюючи як пам’ять даних так і пам’ять команд.

Зазвичай, спектр задіяних програм є доволі таки різноманітним та широким: одні програми керують роботою зовнішніх пристроїв, інші – очікують на запит від користувача або інших додатків, ще інші забезпечують ті чи інші фонові завдання тощо. Враховуючи такий великий потенційний перелік запущених програм, доцільно поділити такі застосунки на певні об’єкти, якими власне і керуватиме ОС. Саме такими дрібними елементами являються **процеси**. Спрощено можна сказати, що процеси є екземплярами виконуваних програм. По-суті процес, це деяка абстракція, до якої належать набір даних, інструкцій коду, можливих відкритих файлів і пристроїв, виділеної для програми пам'яті в момент її виконання, та інші елементи. Іншими словами

можна сказати, що процес представляє собою об'єкт ОС, який є контейнером системних ресурсів, призначених для підтримки виконання тієї чи іншої програми. З такого ж погляду, можна дати визначення і для **програми**, як деякого набору файлів, які містять інформацію про процес та способи його побудови під час виконання.

Програми і процеси являються доволі таки суміжними поняттями. Проте, між ними є велика відмінність. Так, програми являються статичними елементами, які інстальовані в зовнішню пам’ять. Велика кількість таких програм не виконується доти, доки користувач не запустить їх власноруч, окрім тих, які запускаються автоматично або ж за допомогою **планувальника завдань** – програмного модуля ОС, який керує згідно певного алгоритму виконанням завдань. Зазвичай, програми містять деякий набір файлів, пов’язаних між собою, ієрархія яких, формується розробником в інтегрованих середовищах розробки. При цьому, запуск програм здійснюється одним із таких файлів – виконуваним (*.exe, *.com, *.class тощо), який власне і містить набір інструкцій, що повинні бути виконаними ЦП. Як тільки даний файл буде запущений користувачем, відбудеться запуск екземпляру програми – процесу. На відміну від програми, процес являється динамічною структурою, який постійно знаходиться в основній пам’яті ЕОМ і змінює свої виконувані інструкції, в процесі свого виконання. Екземплярів запущеної програми може бути велика кількість, в залежності від її налаштувань. Так, можна запустити декілька екземплярів MS Word і працювати одночасно з декількома текстовими документами. Це ж саме можна сказати і про інші програмні застосунки. Зрозуміло, що все це повинно підтримуватися багатозадачністю, яка на даний час, реалізована переважною більшістю сучасних ОС.

З іншої сторони, не завжди процес і програма напряму пов'язані між собою, що можна спостерігати на прикладі Android. В такій ОС: програми можуть виконуватися при відсутності процесу; кілька програм можуть використовувати один процес; один додаток може використовувати кілька процесів; процес може бути присутнім в системі навіть якщо його додаток не

діє. В цьому немає ніяких протиріч з теорією ОС, оскільки об'єктні бібліотеки і модулі теж є процесами. Зважаючи на це, для процесу можна надати і інше визначення, як об'єктові, якому виділений процесорний час і який може працювати в асинхронному режимі.

З процесом також тісно пов’язане таке поняття як **системний виклик** (СВ), який являє собою звернення користувацького застосунку до функцій ядра ОС. Кожна ОС містить перелік системних викликів, до яких може звернутися екземпляр запущеної програми. Власне код ядра ОС і містить її найбазовіший функціонал (доступ до пристроїв введення/вивдення, пам’яті, процесорного часу, багатозадачність тощо), доступ до якого отримують прикладні програми користувача. При цьому, системні виклики вимагають виконання привілейованих операцій, оскільки ядро ОС власне і працює в привілейованому режимі. В залежності від повноважень процесу, з якого викликається той чи інший СВ, ядро може або виконати, або заборонити такий виклик. Процесом, який виконує СВ може бути: активний процес користувача; системний процес, викликаний «зовнішніми» подіями (напр. клавіатури або миші); процес управління пакетними завданнями; тощо.

З точки зору розробника, СВ являє собою виклик деякої підпрограми із системної бібліотеки (напр. POSIX, WinAPI). Це може здійснюватися як напряму, так і через можливості мови програмування, адже зсередини тієї чи іншої функції/методу буде здійснюватися відповідний СВ. Такі системні виклики можуть викликатися як явно, так неявно (напр. переривання, виключні ситуації).

Що ж стосується процесів, то до системних викликів, які забезпечують їх функціонування, можна віднести такі як створення, очікування, завершення їх роботи та ряд інших. Оскільки СВ по створенню нового процесу здійснюється з того чи іншого процесу, кожен новостворений процес повинен мати «батька». В той же час, такий створений процес також може мати «нащадків» (дочірні процеси), хоча і не обов'язково. Таким чином, формується ієрархія у вигляді дерева процесів, яка реалізується в кожній ОС по-різному. Так наприклад, в

UNIX-подібних системах, самим головним предком, що стоїть на вершині такого дерева, є процес init з ідентифікатором 1.

Оскільки в багатозадачних ОС формується ієрархія взаємопов'язаних процесів, в залежності від архітектури ОС, взаємодія між ними може відбуватися по-різному. Так, в одних системах, дочірні процеси можуть напряму підпорядковуватися батьківським і при закритті останнього, самі будуть знищені. В цьому власне проявляється суть синхронного завершення батьківського потоку, при якому, відбувається автоматичне завершення всіх його нащадків. З іншої сторони, асинхронне завершення процесу, передбачає продовження виконання всіх його нащадків. Так наприклад, в багатьох ОС, зокрема Windows, процеси можуть бути рівноправними.

В залежності від реалізації ОС, виконуваний на них код (процеси і потоки виконання) може працювати в різних **режимах привілеїв**. Так наприклад, *у Windows NT передбачені наступні режими:*

- **режим користувача** – найменш привілейований режим роботи процесора, в якому не надається прямого доступу до апаратного обладнання та обмежений доступ до пам'яті. В такому режимі, власне і працюють запущені користувачем процеси;
- **режим ядра** – привілейований режим роботи процесора, в якому надається повний доступ до всієї системної пам'яті та всіх інструкцій ЦП. В такому режимі виконується код ОС (напр. драйвери та системні служби).

При цьому, ЕОМ досить часто перемикається в різні режими роботи. Так наприклад, користувацький додаток може здійснити запит на використання тих чи інших апаратних ресурсів. В такому разі, буде здійснений відповідний СВ, внаслідок чого, запуститься код ОС.

Такий поділ режимів з врахуванням привілеїв, дозволяє відділити критично важливі дані ОС від даних користувача, тим самим гарантуючи безпечну роботу ЕОМ.

При цьому, обов'язковою умовою реалізації тих чи інших режимів, є апаратна підтримка процесором відповідних режимів роботи. Так, процесори з

архітектурою x86 і x64 забезпечують підтримку 4-ох рівнів привілеїв (4 кільця). Тому, для їх використання необхідна відповідна реалізація з боку ОС, хоча і не обов'язково підтримувати всі режими. З вище описаних режимів Windows NT видно, що ОС підтримує лише два режими: рівень привілеїв 0 (0-ве кільце) – режим ядра; рівень привілеїв 3 (3-тє кільце) – режим користувача. Причина, через яку Windows використовує лише два рівні, полягає в тому, що в деяких апаратних архітектурах, що підтримувалися в минулому (напр., Compaq Alpha і Silicon Graphics MIPS), було реалізовано лише два рівні привілеїв.

Щодо створення процесу, то такій дії передують наступні ситуації:

- ініціалізація системи;
- запит користувача;
- активний процес здійснює СВ по створенню нового процесу;
- запуск пакетного завдання (структура, яка забезпечує послідовний запуск взаємопов'язаних процесів).

В свою чергу, *завершення роботи процесу відбувається з наступних причин*:

- вихід (добровільно) – напр. команда користувача на закінчення процесу;
- вихід при виникненні помилки (добровільно) – напр. введення команди на запуск неіснуючої програми;
- виникнення фатальної помилки (примусово) – напр. помилка запущеної програми, яка містить недопустимі інструкції (напр. ділення на нуль, посилання на неіснуючий елемент);
- знищення іншим процесом (примусово) – напр. завершення роботи всіх запущених процесів після команди по завершенню роботи ЕОМ.

При цьому, *створення та запуск процесу передбачає наступні основні стадії*:

- вказується програма, яку необхідно виконати;
- ОС створює адресний простір для процесу і структури, які описують новий процес;
- заповнюються структури, які описують новий процес;

- з виконуваного файлу, яким запускається новий процес, в адресний простір процесу копіюються код та дані. Якщо така інформація одразу ж не поміщається в пам’ять, вона надходить в область підкачки;
- стан процесу встановлюється як «готовий до виконання»;
- до черги процесів, які очікують на виконання ЦП, додається новий процес.

1.1.2. Класифікація процесів

Стосовно класифікації процесів, то існує велика кількість їх поділу за певними характеристикам. Розглянемо деякі із них:

- *за часовими характеристиками* розрізняють:
  - **інтерактивні** процеси. Час існування таких процесів визначається реакцією ЕОМ на запит обслуговування (час реакції становить секунди);
  - **процеси реального часу**, які мають гарантований час закінчення роботи (мс);
  - **пакетні** процеси, які запускаються один за одним (час реакції становить порядку годин);
- *за генеалогічною ознакою* процеси поділяються на:
  - **батьківські**;
  - **дочірні**;
- *за результативністю* розрізняють:
  - **еквівалентні** процеси, які можуть реалізовуватися як на одному, так і на багатьох процесорах поодинці або за різними алгоритмами, тобто мають різні траси, які визначають порядок та час перебування процесу в різних станах;
  - **тотожні** процеси реалізовуються за однією і тією ж програмою, але мають різні траси;
  - **однакові** процеси, які реалізуються за однією програмою і мають однакові траси;
- *за способом опрацювання ЦП*, процеси можна класифікувати на:

- **послідовні**;
- **паралельні**;
- **комбіновані**;

- *за місцем реалізації:*
  - **внутрішні** – реалізуються на ЦП;
  - **зовнішні** – реалізуються на зовнішніх процесорах;
- *за належністю до ОС*:
  - **системні** – нативні, вбудовані в ОС;
  - **користувацькі** – пов'язані з прикладними програмами користувача;
- *за зв'язністю* розрізняють процеси:
  - **взаємопов'язані** – мають деякий зв'язок;
  - **ізольовані** – слабко зв'язані;
  - **незалежні** – використовують спільні ресурси, проте мають власні інформаційні бази;
  - **взаємодіючі** – мають інформаційні зв'язки і розділяють загальні структури даних;
  - **взаємозв'язані за ресурсами**;
  - **конкуруючі**.

1.1.3. Моделі процесів

Розглянемо модельні уявлення стосовно процесів. *З точки зору кількості станів, в яких може перебувати виконуваний процес, виділяють:*

- **модель трьох станів процесу** – модель, згідно якої, процес може перебувати в наступних трьох станах (рис. 1.1):
  - *виконання* – активний стан процесу, який безпосередньо поступив на виконання до ЦП. В такому стані, процес володіє всіма необхідними йому ресурсами;
  - *очікування* – пасивний стан, в якому процес є заблокований, очікуючи тих чи інших подій (напр., чекає введення користувачем даних або звільнення потрібного йому пристрою);

- *готовність* – пасивний стан, в якому він стоїть в черзі процесів і готовий до виконання, проте в даний момент не виконується.

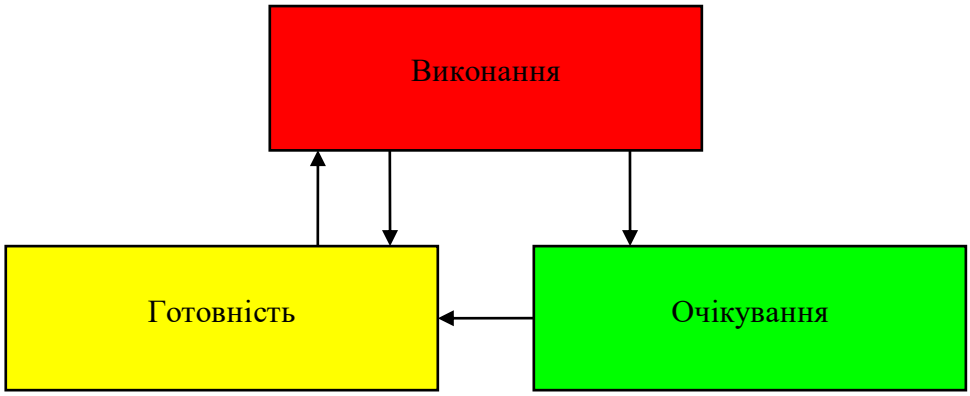


Рис. 1.1. Модель трьох станів процесу.

Перехід типу *виконання → очікування* може наприклад відбутися у випадку, якщо процес був у фазі свого виконання, проте деякою інструкцією здійснив запит на введення з клавіатури користувачем даних.

В свою чергу, перехід типу *очікування → готовність*, згідно попереднього прикладу, може відбутися наприклад тоді, коли процес здійснив запит на введення користувачем даних і перебуває в стадії їх очікування. В той же час, планувальник завдань може потіснити даний процес новим процесом. В такому разі, очікуючий процес поступиться місцем новому, а сам перейде в стан готовності.

Перехід *готовність → виконання* відбувається тоді, коли планувальник завдань здійснює вибірку з черги готових до виконання процесів і передає його на виконання до ЦП. При цьому, одним процесором може виконуватися лише один процес.

Процес переходу типу *виконання → готовність* може наприклад відбутися тоді, коли квант часу для виконання ЦП даного процесу закінчився, проте всі інструкції процесу ще не виконалися. В такому разі, процес переводиться в стан готовності;

- **модель п'яти станів процесу** – модель, в якій окрім вище наведених трьох станів також передбачається додаткові два (рис. 1.2):

- *народження* – пасивний стан, при якому самого процесу ще не існує, проте вже готова структура для його появи;
- *смерть* – ситуація, при якій процес вже завершив свою роботу, проте структура даних, яка його описує, залишилася в списку процесів. В такому разі, говорять про **процеси-зомбі**.

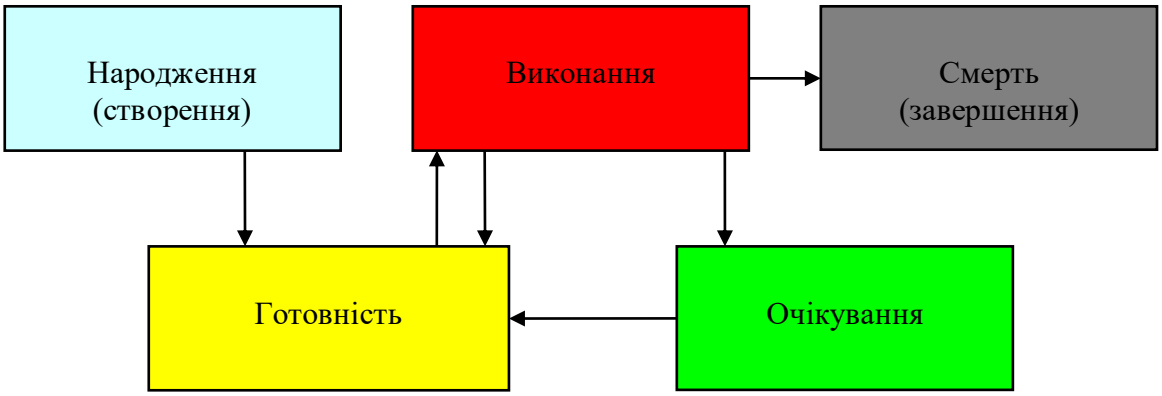


Рис. 1.2. Модель п’яти станів процесу.

Така модель передбачає наступну поведінку процесів:

- *народження → готовність:* створення процесу;
- *готовність → виконання:* запуск/відновлення процесу;
- *виконання → очікування:* блокування процесу;
- *виконання → готовність:* зміна пріоритету процесу;
- *очікування → готовність:* пробудження процесу;
- *виконання → смерть:* знищення процесу.

При цьому, в ОС може бути запущена досить таки велика кількість різноманітних процесів, роботою яких, як вже зазначалося, керує планувальник завдань. *Серед головних цілей, які повинен виконувати планувальник слугують наприклад наступні*:

- максимальна пропускна здатність (загальна к-сть виконаних завдань за одиницю часу);
- мінімізація часу очікування (час від готовності роботи до початку виконання першої точки);
- забезпечення справедливості (однаковий час ЦП для кожного процесу або належний час відповідно до пріоритету завдання).

В найпростішому випадку, робота планувальника виглядає наступним чином (рис. 1.3):

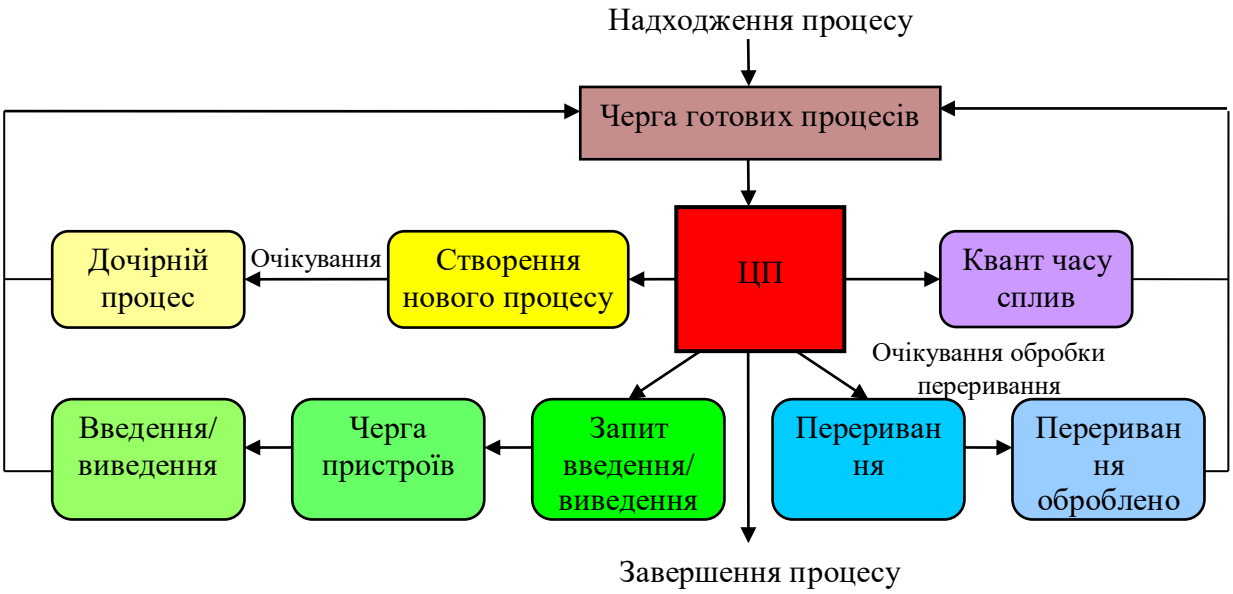


Рис. 1.3. Черга процесів, які поступають на виконання до ЦП.

1.1.4. Багатозадачність та багатопоточність

В залежності від реалізації ОС, процеси можуть складатися з дрібніших елементів, які називаються **потоками виконання**. Такі об’єкти являють собою незалежні складові елементи процесу, які працюють паралельно однин одному. Такий підхід в багатьох випадках забезпечує більш ефективну роботу ЕОМ.

Слід зауважити, що *кожен процес має свій власний адресний простір, кожен з яких захищений від втручання від іншого процесу*, забезпечуючи тим самим інформаційну безпеку. В той же час, *потоки знаходяться в межах одного процесу, спільно використовуючи його ресурси*. При цьому, *кожний процес повинен складатися як мінімум з одного потоку*.

В такому разі, *модельне уявлення процесів можна класифікувати по наступному критерію (рис. 1.4):*

- **однозадачні ОС** – містять лише один адресний простір, в межах якого, в кожний момент часу, може виконуватися лише один однопоточний процес. До таких ОС належить напр. MS-DOS;

- **багатозадачні ОС на рівні потоків** – однозадачні ОС, які підтримують виконання в певний момент часу лише одного процесу, проте такий процес може складатися вже з декількох потоків виконання. Такі ОС в основному використовуються у вбудованих системах. При цьому, їх реалізація повинна мати апаратну підтримку багатопоточності з боку ЦП;
- **багатозадачні ОС на рівні процесів**. Такі системи забезпечують одночасне виконання декількох однопоточних процесів. ОС такого типу не оперують поняттям потоку виконання. Представниками даних ОС являються традиційні версії UNIX;
- **багатозадачні ОС на рівні процесів та потоків**. Реалізація таких ОС передбачає підтримку одночасного виконання декількох процесів, кожен з яких, може складатися також з певної кількості потоків виконання. Дана концепція також передбачає апаратну підтримку процесором мультипоточності. Такий підхід реалізується в більшості сучасних ОС, серед яких напр. Windows NT, сучасні версії UNIX.

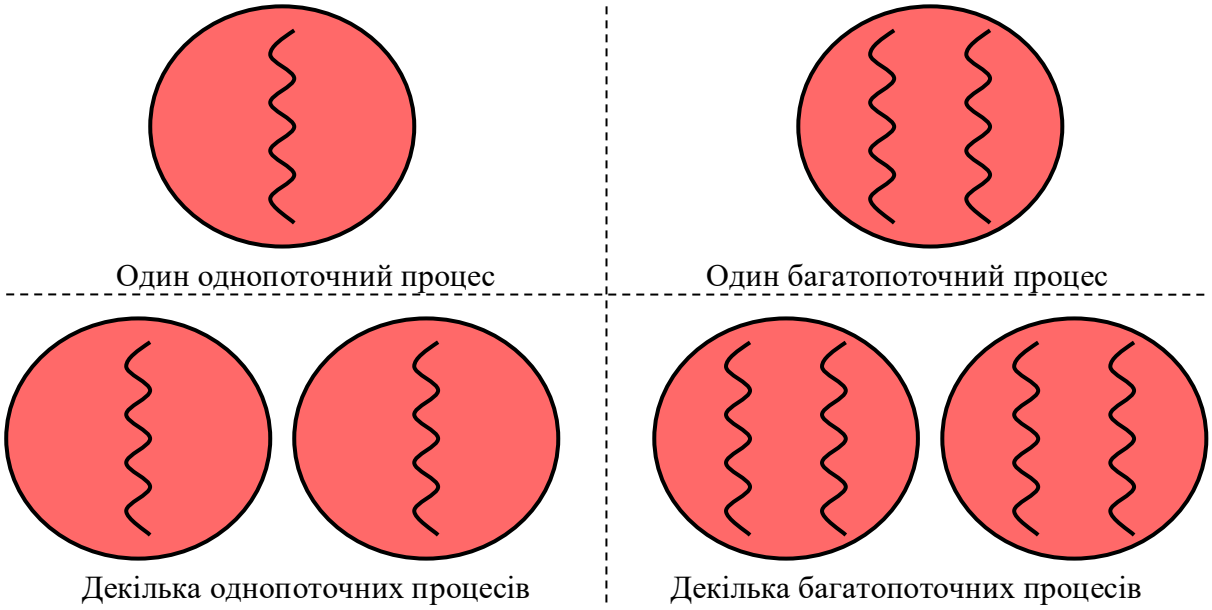


Рис. 1.4. Класифікація ОС по критерію багатозадачності.

Функція багатозадачності на рівні процесів і/або потоків одним процесором зазвичай реалізується механізмом тимчасового мультиплексування. В такому випадку, відбувається перемикання контексту між різними

процесами/потоками з виділенням для них, з боку ЦП, певного часового проміжку – **кванту часу** (рис. 1.5). Висока частота такого перемикання, забезпечує ілюзію паралельності виконання програм, хоча в дійсності це не так. В такому випадку, слід говорити про **псевдопаралельність**.

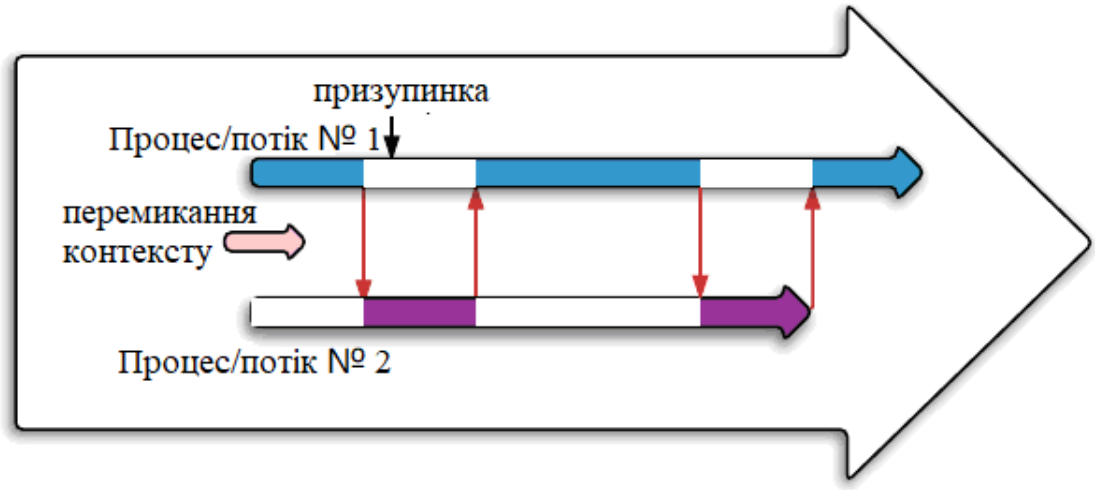


Рис. 1.5. Приклад багатозадачності в однопроцесорній системі

З іншої сторони, при наявності декількох процесорів можна казати про істинну **паралельність**, за умови, що кількість запущених процесів/потоків не перевищує число процесорів ЕОМ. Це ж саме стосується і **розподілених систем**, як мережі ЕОМ, що здійснюють виконання спільних завдань.

1.2. Практична частина

1.2.1. Запуск процесів

Клас Runtime

Технологія Java дозволяє користувачеві запустити на виконання хоча б один – головний процес, який представлений своїм виконавчим середовищем. По-суті, це є звичайний додаток, запущений користувачем. Такий процес містить як мінімум один потік виконання, програмний код якого, описаний в головному методі main, як точка входу в програму. При цьому, головний процес може містити інструкції, що дозволяють запускати дочірні процеси, які часто називають **підпроцесами** (така назва є досить таки умовною, оскільки породжені підпроцеси є повноцінними процесами). В той же час, процес з яким

користувач працює в певний конкретний момент часу називається **поточним процесом**. Підпроцесом також називатимемо і такий процес, керування яким здійснюється з поточного процесу, навіть якщо такий підпроцес не породжений поточним процесом.

Відомо, що середовищем виконання, в якому запускаються Java-процеси, «опікується» клас Runtime (пакет java.lang.Runtime):

public class Runtime extends Object,

а тому логічно, що він надає ряд методів для керування процесами.

Кожна Java-програма має єдиний екземпляр такого класу, який дозволяє для неї взаємодіяти із середовищем, в якому вона працює. Додаток не може створити власний екземпляр цього класу. Runtime немає конструкторів і створюється зазвичай власним статичним методом

public static Runtime getRuntime().

Отримавши екземпляр такого класу (в поточному процесові), користувачеві надається можливість доступу до низки методів (табл. 1.1), які власне і запускають ті чи інші підпроцеси.

Таблиця 1.1. Методи класу Runtime, що повертають значення Process.

Метод	Опис дій метода
public Process exec(String command) throws IOException (IllegalArgumentException, NullPointerException, SecurityException)*	Виконує вказану команду command в окремому процесі. Повертає екземпляр Process для управління підпроцесом. Це ж стосується і всіх нижче наведених методів. Метод еквівалентний з exec(command, null, null).
public Process exec(String[] cmdarray) throws IOException (IndexOutOfBoundsException, NullPointerException, SecurityException)	Виконує вказану команду та її аргументи в окремому процесі. cmdarray - масив, що містить команду виклику та її аргументи. Еквівалентно з exec(cmdarray, null, null).
public Process exec(String[] cmdarray, String[] envp) throws	Аналогічно попередньому, проте додатковим параметром слугує envp - масив рядків, кожен елемент

IOException (IndexOutOfBoundsException, NullPointerException, SecurityException)	якого має параметри змінної середовища у форматі name = value, або null, якщо підпроцес має успадкувати середовище поточного процесу. Еквівалентно з exec(cmdarray, envp, null).
public Process exec(String[] cmdarray, String[] envp, File dir) throws IOException (IndexOutOfBoundsException, NullPointerException, SecurityException)	Аналогічно попередньому, проте додатково вказується робочий каталог підпроцесу dir (якщо null, підпроцес успадковує робочий каталог поточного процесу).
public Process exec(String command, String[] envp) throws IOException (IllegalArgumentException, NullPointerException, SecurityException)	Виконує вказану команду command в окремому процесі із зазначеним середовищем envp. Еквівалентно exec(command, envp, null).
public Process exec(String command, String[] envp, File dir) throws IOException (IllegalArgumentException, NullPointerException, SecurityException)	Аналогічно попередньому, із зазначенням робочого каталогу підпроцесу dir (якщо null, підпроцес успадковує робочий каталог поточного процесу). Еквівалентно з exec(cmdarray, envp, dir), де cmdarray містить масив усіх лексем command.

** - всі виключення, які зазначені в дужках, являються неконтрольованими, а тому обробляються на власний розсуд розробника. Дане формулювання дійсне на протязі всього посібника.*

Як видно із попередньої таблиці, щоб створити деякий підпроцес, використовується перевантажений метод exec(), який в залежності від його реалізації, приймає той чи інший набір аргументів. При цьому, «природа» генерованих підпроцесів може бути найрізноманітнішою – починаючи від системних програм і закінчуючи користувацькими процесами, написаними на різних мовах програмування. Слід зауважити, що запуск з допомогою таких методів деяких нативних процесів тих чи інших ОС, є досить таки платформозалежним, що неодмінно слід враховувати.

Лістинг 1.1. Запуск підпроцесів в середовищі ОС Windows NT.

```
// import java.io.File;

public class ProcessDemo1 {
	public static void main(String[] args) {
		Runtime r = Runtime.getRuntime();
		System.out.println("Поточний Java-процес розпочав свою
			роботу");
		try {
			System.out.println("Запуск підпроцесу № 1");
			/* запуск підпроцесу; P.S.:
			- оскільки до багатьох влаштованих у Windows програм
			створені відповідні змінні оточення, тому немає
			необхідності зайвий раз прописувати повні шляхи до їх
			виконуваних файлів;
			- при необхідності, виконувані файли програм можна
			прописувати із розширенням (notepad.exe) */
			r.exec("notepad");

			System.out.println("Запуск підпроцесу № 2");
			/* запуск підпроцесу з повним шляхом до виконуваного
			файлу */
			r.exec("C:\\Users\\User\\AppData\\Roaming\\Spotify\\
				Spotify.exe");

			System.out.println("Запуск підпроцесу № 3");
			/* якщо шлях до підпроцесу містить пробіли і не
			запускається звичайним чином, його слід помістити в
			лапки (керуючий символ \" тотожний для "), згідно
			наступного запису */
			r.exec("\"C:\\Program Files\\CCleaner\\CCleaner.exe\"");

			System.out.println("Запуск підпроцесу № 4");
```

```
/* запуск підпроцесу з аргументами: розпаковка архіву;
такий запис є більш бажаним в порівнянні з однорядковим
із пробілами між командою і аргументами, на кшталт
r.exec("C:\\Program Files\\WinRAR\\WinRAR.exe x
     D:\\Java-Test\\Process\\ProcessArchive.rar *.*
     D:\\Java-Test\\Process\\unArch\\") */
String[] command1 = {"C:\\Program Files\\WinRAR\\
     WinRAR.exe", "x", "D:\\Java-Test\\Process\\
     ProcessArchive.rar", "*.*", "D:\\Java-Test\\
     Process\\unArch\\"};
r.exec(command1);

/* альтернативою попередньому запуску підпроцесу
слугують наступні інструкції, проте слід зауважити, що
використання нижче описаного методу залежить від ОС і в
деяких випадках може викликати неочікувані помилки;
для усунення таких помилок, як варіант, необхідно
створити додаткову змінну середовища для відповідного
підпроцесу */
// String[] command2 = {"WinRAR.exe", "x",
//    "D:\\basketball.rar", "*.*", "D:\\basketball\\"};
// File dirPro = new File("C:\\Program Files\\
//    WinRAR\\");
// r.exec(command2, null, dirPro);

System.out.println("Запуск підпроцесу № 5");
/* запуск невиконуваних файлів, асоційованих з деякою
програмою напряму неможливий - інструкція
r.exec("D:\\Java-Test\\Process\\forest.jpg")
не працюватиме, а тому, попередньо необхідно викликати
програму запуску (напр. mspaint) */
r.exec("mspaint D:\\Java-Test\\Process\\forest.jpg");

System.out.println("Запуск підпроцесу № 6");
```

```
                /* запуск командного рядка; P.S.: звичайна команда
                r.exec("cmd.exe") відкривала б cmd і одразу його
                закривала б, тому необхідно використовувати пусту
                команду start з ключем /c (або /k), яка відкриває cmd в
                новому вікні */
                r.exec("cmd.exe /c start");

                System.out.println("Запуск підпроцесу № 7");
                /* виведення тексту в cmd-підпроцесі; P.S.:
                - команда echo виводить текст на екран;
                - команда start виконує команду/програму в новому вікні
                */
                r.exec(new String[] {"cmd.exe", "/c", "start", "echo",
                     "Hello World from cmd-subProcess"});
           }
           catch (Exception ex) {
                ex.printStackTrace();
           }
           System.out.println("Поточний Java-процес завершив свою
                роботу");
           /* слід звернути увагу на те, що нормальне (не аварійне)
           завершення поточної Java-програми не передбачає завершення її
           запущених підпроцесів, а тому вони працюватимуть далі */
     }
}
```

Консоль поточного Java-додатку:

```
Поточний Java-процес розпочав свою роботу
Запуск підпроцесу № 1
Запуск підпроцесу № 2
Запуск підпроцесу № 3
Запуск підпроцесу № 4
Запуск підпроцесу № 5
```

```
Запуск підпроцесу № 6
Запуск підпроцесу № 7
Поточний Java-процес завершив свою роботу
```

Консоль підпроцесу № 7 (команда echo):

```
"Hello World from cmd-subProcess"
```

Лістинг 1.2. Запуск підпроцесів з командного рядка Windows.

```
import java.io.FileWriter;
import java.util.Scanner;
// import java.io.File;
// import java.awt.Desktop;

public class ProcessDemo2 {
	public static void main(String[] args) {
		// директорія файлів програм
		String WIN_PROGRAMFILES = System.getenv("programfiles");
		// символ дільника в шляху
		String FILE_SEPARATOR = System.getProperty("file.separator");
		Runtime r = Runtime.getRuntime();
		System.out.println("Поточний Java-процес розпочав свою
			роботу");
		try {
			/* запуск калькулятора з командного рядка; по суті,
			відбувається ланцюг виконуваних програм: cmd --> calc;
			P.S.:
			- багато програм і команд необхідно відкривати із
			певного середовища (напр. ком.рядок, провідник), а тому,
			в першу чергу, потрібно викликати дане середовище, хоча
			і у випадку calc це необов'язково;
			- команда start виконує команду/програму в новому вікні,
			а оскільки калькулятор являється віконною програмою,
```

```
то так чи інакше відкриється в новому вікні, а тому,
можна не використовувати команду start
(r.exec(new String[] {"cmd", "/c", "calc.exe"})) */
r.exec(new String[] {"cmd", "/c", "start", "calc.exe"});

/* запуск редактора реєстру; деякі процеси вимагають
підвищених привілеїв, і в деяких випадках інструкція
r.exec("regedit") може не виконуватися;
як варіант, можна здійснити запуск через cmd */
r.exec(new String[] {"cmd", "/c", "start", "regedit"});

// відкриття сторінки пошуку в браузері по замовчуванню
r.exec(new String[] {"cmd.exe", "/c", "start",
     "http://google.com"});

// запуск програми, незалежно від версії Windows
String[] command =
     {"cmd.exe", "/c", WIN_PROGRAMFILES
     + FILE_SEPARATOR + "Total Commander"
     + FILE_SEPARATOR + "Totalcmd64.exe"};
r.exec(command);

// отримання списку інстальованих на ПК програм
r.exec(new String[] {"cmd", "/c", "start", "cmd", "/k",
     "wmic product get name,version"});

/* запуск файлів (лінійка програм: cmd --> програма,
асоційована з даним типом файлів (залежить від
налаштувань Windows)) */
r.exec("cmd.exe /c D:\\Java-Test\\Process\\forest.jpg");
// запуск через провідник
// File file = new File("D:\\Java-Test\\Process\\
//    forest.jpg"); */
// r.exec("c:\\windows\\explorer.exe /n, /root, " +
```

```
//    file.getPath());
/* альтернативний запуск, з використанням класу
java.awt.Desktop */
// Desktop.getDesktop().open(file);
// запуск через rundll32
// r.exec ("rundll32.exe SHELL32.DLL, ShellExec_RunDLL "
//    + file.getAbsolutePath());

/* якщо шлях до файлу містить пробіли (або існує
ймовірність їх присутності), необхідно вказати заголовок
команди запуску */
String[] command2 =
      {"cmd", "/c", "start", "\"DummyTitle\"",
      "D:\\Java-Test\\Process\\Path with Space\\
      Documentation_Link.doc"};
r.exec(command2);

/* запуск підпроцесу cmd, який запускає дві програми
(диспетчер завдань і монітор ресурсів) */
r.exec(new String[] {"cmd", "/c", "taskmgr", "&",
      "resmon"});
/* запуск підпроцесу powershell, який запускає дві
програми */
// r.exec(new String[] {"powershell", "-command",
//    "&\"{taskmgr; regedit}\""});

// формування і запуск cmd-файлу
String[] cmdCommands = {"echo Hello World from
      CMD!!!\n", "java -version\n", "pause"};
String cmdFilePath = "D:\\Java-Test\\Process\\
      Hello.cmd";
FileWriter fr = new FileWriter(cmdFilePath);
for (String cmdCommand : cmdCommands)
      fr.write(cmdCommand);
```

```
                fr.close();
                r.exec(new String[] {"cmd.exe", "/K", "start",
                        cmdFilePath});
            }
            catch (Exception ex) {
                ex.printStackTrace();
            }
            /* слід звернути увагу на те, що зупинивши аварійно
            хід виконання поточного Java-додатку, можна спостерігати,
            що всі його запущені підроцеси теж припинять свою роботу */
            System.out.println("Поточний Java-процес завершив свою
                роботу");
            // очікування натиснення клавіші вводу
            new Scanner(System.in).nextLine();
        }
}
```

Консоль поточного Java-додатку:

```
Поточний Java-процес розпочав свою роботу
Поточний Java-процес завершив свою роботу
```

Консоль команди wmic:

```
Name                                                        Version
Viber                                                       17.4.0.10
Пакет сум?сност? для випуску 2007 системи Office  12.0.6514.5001
NI Logos 19.0                                               19.00.49152
NI LabWindows/CVI Shared Runtime 2019                       19.00.49152
NI Circuit Design Suite Databases                           14.20.49251
NI LabVIEW Runtime 2018 SP1 f3                              18.60.49152
NI Trace Engine                                             19.00.49152
Intel(R) LMS                                                1.0.0.0
...
```

Консоль командного файлу Hello.cmd:

```
C:\Users\User\eclipse-workspace\ProcessDemo2>echo Hello World from CMD!!!
Hello World from CMD!!!

C:\Users\User\eclipse-workspace\ProcessDemo2>java -version
java version "17.0.2" 2022-01-18 LTS
Java(TM) SE Runtime Environment (build 17.0.2+8-LTS-86)
Java HotSpot(TM) 64-Bit Server VM (build 17.0.2+8-LTS-86, mixed mode, sharing)

C:\Users\User\eclipse-workspace\ProcessDemo2>pause
Press any key to continue . . .
```

Лістинг 1.3. Запуск Java-підпроцесів користувача:

- 1-ий (class-файл SubProcess3_1) – отримуючи дані (в якості параметрів) від батьківського процесу (ProcessDemo3) (однонаправлена міжпроцесна взаємодія), здійснює вивід тексту в файл;
- 2-ий (class-файл SubProcess3_2) і 3-ій (jar-файл SubProcess3_3) – виводять текстові повідомлення у власні консолі.

Головний (поточний) процес ProcessDemo3:

```
import java.util.Scanner;

public class ProcessDemo3 {

	static Runtime r = Runtime.getRuntime();

	/* java-підпроцес SubProcess3_1 у вигляді class-файлу з вхідними
	параметрами */
	static void runSubProcess3_1(String a, String b) {
		/* шлях до директорії, що містить нескомпільований файл
```

```
        SubProcess4_1.java */
        String javaFilePath = "C:\\Users\\User\\eclipse-workspace\\
              SubProcess3_1\\src\\";
        // ім'я файлу без розширення
        String javaFileName = "SubProcess3_1";
        /* шлях до директорії, де буде міститися скомпільований файл
        SubProcess4_1.class */
        String classFilePath = "C:\\Users\\User\\eclipse-workspace\\
              SubProcess3_1\\bin\\";
        try {
              /* компіляція додатку (SubProcess3_1.java -->
              SubProcess3_1.class) (якщо існує необхідність);
              P.S.: якщо запустити підпроцес без ключа "-d" і
              параметра classFilePath, компіляція відбудеться у ту ж
              директорію, де знаходиться java-файл
              (див. докладніше команду "javac --help" в cmd) */
              r.exec(new String[] {"javac", "-d", classFilePath,
                    javaFilePath + javaFileName + ".java"});
              Thread.sleep(1000); // очікування завершення компіляції
              /* запуск Java-підпроцесу (SubProcess3_1.class) з
              параметрами a, b (див. докладніше команду "java --help"
              в cmd) */
              r.exec(new String[] {"java", "-cp", classFilePath,
                    javaFileName, a, b});
              // альтернатива попередній інструкції
              // r.exec(new String[] {"cmd", "/K", "cd",
              //    classFilePath, "&", "java", javaFileName, a, b});

        }
        catch (Exception e) {
              System.out.println("Error from class-SubProcess3_1!!!");
        }
}
```

```
/* java-підпроцес SubProcess3_2 у вигляді class-файлу без
вхідних параметрів */
static void runSubProcess3_2() {
	// шлях до нескомпільованого файлу SubProcess3_2.java
	String javaFilePath = "C:\\Users\\User\\eclipse-workspace\\
		SubProcess3_2\\src\\";
	// ім'я файлу без розширення
	String javaFileName = "SubProcess3_2";
	/* шлях до директорії, де буде міститися скомпільований файл
	SubProcess3_2.class */
	String classFilePath = "C:\\Users\\User\\eclipse-workspace\\
		SubProcess3_2\\bin\\";
	try {
		/* компіляція додатку (SubProcess3_2.java -->
		SubProcess3_2.class) (якщо існує необхідність) */
		r.exec(new String[] {"javac", "-d", classFilePath,
			javaFilePath + javaFileName + ".java"});
		Thread.sleep(1000); // очікування завершення компіляції
		/* запуск Java-підпроцесу (SubProcess3_2.class) без
		параметрів;
		P.S.: такий запис супроводжується відкриттям додаткового
		вікна cmd, в якому власне і буде відбуватися робота
		даного підпроцесу; це необхідно для того, оскільки перша
		консоль відразу ж буде закрита, в той час як виведення
		тексту на екран власне і відбуватиметься в другу
		консоль; слід звернути увагу на те, що консолі поточного
		Java-додатку і даного Java-підпроцесу SubProcess3_2 є
		різними */
		r.exec(new String[] {"cmd", "/c", "start", "cmd", "/k",
			"java", "-classpath", classFilePath, javaFileName});
	}
	catch (Exception e) {
		System.out.println("Error from class-SubProcess3_2!!!");
	}
```

```
}

/* java-підпроцес SubProcess3_3 у вигляді jar-файлу без вхідних
параметрів */
static void runSubProcess3_3() {
	// ім'я jar-файлу
	String jarFileName = "SubProcess3_3.jar";
	/* ім'я class-файлу без розширення, який містить точку входу
	(метод main) */
	String mainClassFileName = "SubProcess3_3_1";
	// адреса зберігання jar-файлу
	String jarDirPath = "D:\\Java-Test\\Process\\";
	/* адреса з class-файлами, з яких буде створений jar */
	String classDirPath = "C:\\Users\\User\\eclipse-workspace\\
		SubProcess3_3\\bin\\";
	try {
		/* створення jar-файлу ((SubProcess3_3_1 /
		SubProcess3_3_2).class --> SubProcess3_3.jar) (при
		необхідності); така операція передбачає вже створені
		наперед скомпільовані class-файли, а тому при
		необхідності їх потрібно створити (див. вище);
		команди для запуску jar (див. докладніше команду
		"jar --help" в cmd):
		--create, --file створюють jar-файл;
		--main-class задає клас з точкою входу в програму;
		-C вказує директорію з class-файлами, які необхідно
		включити в jar;
		P.S.:
		- jar-архіви необхідні, якщо проект складається з
		декількох класів/бібліотек/файлів тощо, а тому, щоб
		структурувати такий проект, його поміщають в архівний
		jar-файл;
		- jar-архіви можна також створити і засобами IDE
		(напр. в Eclapse IDE: Export --> Runnable JAR file) */
```

```
                r.exec(new String[] {"jar", "--create", "--file",
                        jarDirPath + jarFileName, "--main-class",
                        mainClassFileName, "-C", classDirPath, "."});
                /* альтернативні варіанти попередньої інструкції: */
                // r.exec(new String[] {"jar", "cfe", jarDirPath +
                //    jarFileName, mainClassFileName, "-C", classDirPath,
                //    "."});
                // r.exec(new String[] {"cmd", "/K", "cd", classDirPath,
                //    "&", "jar", "cfe", jarDirPath + jarFileName,
                //    mainClassFileName, "*.class"});
                Thread.sleep(1000); /* очікування створення jar */
                // запуск jar
                r.exec(new String[] {"cmd", "/c", "start", "cmd", "/k",
                        "java", "-jar", jarDirPath + jarFileName});
            }
            catch (Exception e) {
                System.out.println("Error from jar-SubProcess3_3!!!");
            }
        }

        public static void main(String[] args) {
            runSubProcess3_1("2", "3");
            runSubProcess3_2();
            runSubProcess3_3();
            System.out.println("Hello from class-ProcessDemo3!!!");
            new Scanner(System.in).nextLine();
        }
}
```

Консоль поточного Java-додатку ProcessDemo3:

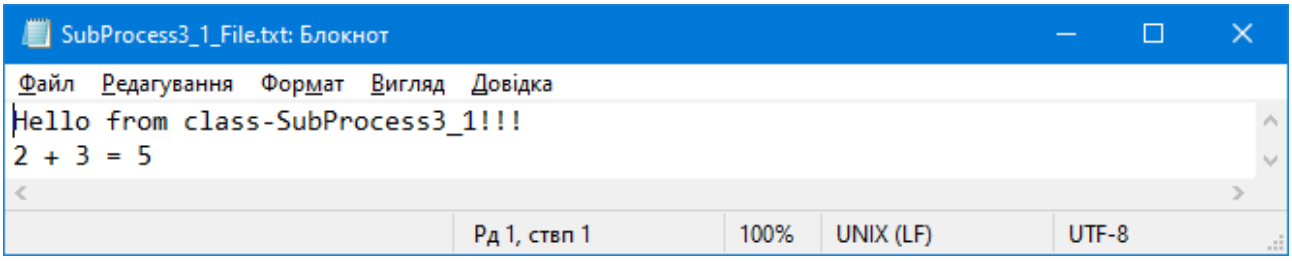
```
Hello from class-ProcessDemo3!!!
```

Java-підпроцес SubProcess3_1:

```
import java.io.IOException;
import java.io.FileWriter;

public class SubProcess3_1 {
	public static void main(String[] args) throws IOException {
		int sum = Integer.parseInt(args[0]) +
			Integer.parseInt(args[1]);
		FileWriter fr = new FileWriter("D:\\Java-Test\\Process\\
			SubProcess3_1_File.txt");
		fr.write("Hello from class-SubProcess3_1!!!" + '\n');
		fr.write(args[0] + " + " + args[1] + " = " + sum);
		fr.close();
	}
}
```

Згенерований файл Java-підпроцесу SubProcess3_1:



Java-підпроцес SubProcess3_2:

```
import java.util.Scanner;

public class SubProcess3_2 {
	public static void main(String[] args) {
		System.out.println("Hello from class-SubProcess3_2!!!");
		new Scanner(System.in).nextLine();
	}
```

```
}
```

Консоль Java-підпроцесу SubProcess3_2:

```
Hello from class-SubProcess3_2!!!
```

Java-підпроцес SubProcess3_3:

```
import java.util.Scanner;

public class SubProcess3_3_1 {
	public static void main(String[] args) {
		SubProcess3_3_2 sp = new SubProcess3_3_2("Hello from
			jar-SubProcess3_3!!!");
		new Scanner(System.in).nextLine();
	}
}

// нижче описаний клас знаходиться в іншому файлі
public class SubProcess3_3_2 {
	SubProcess3_3_2(String line) {
		System.out.println(line);
	};
}
```

Консоль Java-підпроцесу SubProcess3_3:

```
Hello from jar-SubProcess3_3!!!
```

1.2.2. Маніпулювання роботою процесів

Клас Process

Для маніпуляції процесами, технологією Java передбачений Process API, який дозволяє запускати, отримувати інформацію та керувати процесами

операційної системи. Як видно із попередньої таблиці, кожен із методів exec() повертає об’єкт класу Process. Всі процеси в Java представлені абстрактним класом Process (Java 8):

public abstract class Process extends Object,

який знаходиться в пакеті java.lang.Process і представлений єдиним конструктором (табл. 1.2):

Таблиця 1.2. Конструктори класу Process.

Конструктор	Опис дій конструктора
public Process()	Створення нового об'єкта Process.

Окрім ряду методів унаслідуваних від Object (clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait), клас Process реалізує власні методи, представлені в наступній таблиці:

Таблиця 1.3. Методи класу Process.

Метод	Опис дій метода
public abstract void destroy()	Перериває підпроцес, примусове завершення якого залежить від платформи.
public Process destroyForcibly()	Перериває підпроцес, примусово його завершуючи та повертаючи на нього посилання. *Примітка: такий підпроцес деякий короткий час ще може працювати (isAlive() може повертати true) після того, як було викликано даний метод.* При необхідності, метод може бути пов'язаний із waitFor().
public abstract int exitValue() (IllegalThreadStateException)	Повертає код завершення підпроцесу. Нормальне завершення повертає значення 0.
public abstract InputStream getErrorStream()	Повертає потік введення для читання даних з потоку виведення помилок err викликаючого підпроцесу.
public abstract InputStream getInputStream()	Повертає потік введення для читання даних з потоку виведення out викликаючого підпроцесу.

public abstract OutputStream getOutputStream()	Повертає потік виведення для запису даних в потік введення in викликаючого підпроцесу.
public boolean isAlive()	Повертає true, якщо підпроцес все ще працює.
public abstract int waitFor() throws InterruptedException	Очікує завершення викликаючого підпроцесу. Повертає код завершення даного підпроцесу (0 - якщо процес завершився нормально).
public boolean waitFor (long timeout, TimeUnit unit) throws InterruptedException (NullPointerException)	Очікує завершення викликаючого підпроцесу протягом періоду часу timeout (unit - одиниці часового проміжку). Повертає true, якщо підпроцес завершив свою роботу до зазначеного timeout.

З таблиці видно, що клас Process надає методи керування процесами, такі як отримання інформації про процес, введення/виведення в/з процес(у), завершення його виконання тощо.

Слід звернути увагу на тому, що підпроцеси не припиняють свого виконання, навіть якщо більше немає посилань на їх Process-об’єкти, і навпаки, продовжуватимуть виконуватися асинхронно по відношенню до батьківського процесу, до того моменту, доки вони повністю не завершать свою роботу.

Алгоритм управління роботою процесів є досить таки тривіальним:

№	Етапи алгоритму	Приклад програмного коду
1.	Створення об’єкта виконавчого середовища.	Runtime r = Runtime.getRuntime();
2.	Запуск на виконання підпроцесу.	Process pro = r.exec("notepad");
3.	Маніпулювання роботою процесу.	pro.destroy();

Лістинг 1.4. Очікування і завершення роботи підпроцесів в середовищі Windows.

```
import java.util.concurrent.TimeUnit;

public class ProcessDemo4 {
    public static void main(String[] args) {
```

```
Runtime r = Runtime.getRuntime();
Process pro;
      try {
            System.out.println("Запуск підпроцесу № 1");
            pro = r.exec("msinfo32.exe"); // відомості про систему
            System.out.println("Зачекайте 5 с");
            Thread.sleep(5000); // очікування 5 с
            pro.destroy(); // завершення підпроцесу
            /* у випадку, якщо destroy() не працює (оскільки
            залежить від реалізації ОС), можна викликати метод
            destroyForcibly(), який примусово завершить підпроцес */
            if (pro.isAlive()) {
                  pro.destroyForcibly();
                  /* за необхідності, можна почекати доки підпроцес
                  остаточно завершиться, оскільки згідно
                  специфікації, після виклику destroyForcibly(),
                  підпроцес досить таки короткий час може ще
                  працювати */
                  while (pro.isAlive());
            }
            /* повернення коду завершення підпроцесу:
             0 - підпроцес був закритий користувачем (тобто
            завершився зсередини підпроцесу);
             1 - підпроцес був закритий поточним Java-процесом
            (тобто завершився ззовні, відносно підроцесу) */
            System.out.println("Підроцес № 1 завершений. Код
                  завершення: " + pro.exitValue());

            System.out.println("Запуск підпроцесу № 2");
            pro = r.exec("C:\\Program Files\\WinRAR\\WinRAR.exe");
            /* очікування завершення підпроцесу;
            P.S.: як було сказано дещо раніше, якщо поточний Java-
            процес завершиться аварійно, то і всі запущені в ньому
            підпроцеси теж завершать свою роботу достроково */
```

```
            pro.waitFor();
            System.out.println("Підпроцес № 2 завершений. Код
                  завершення: " + pro.exitValue());

            System.out.println("Запуск підпроцесу № 3");
            pro = r.exec("C:\\Users\\User\\AppData\\Roaming\\
                  Spotify\\Spotify.exe");
            /* очікування завершення підпроцесу, або спливання
            тайм-ауту, після чого, поточний Java-процес продовжить
            своє виконання */
            pro.waitFor(5, TimeUnit.SECONDS);
            if (pro.isAlive())
                  System.out.println("Підпроцес № 3 не завершив своєї
                        роботи");
            else
                  System.out.println("Підпроцес № 3 завершив свою
                        роботу");

            System.out.println("Створення і запуск підпроцесу № 4");
            pro = r.exec("notepad");
            Thread.sleep(5000);
            /* альтернативний метод завершення підпроцесу через його
            ім'я (ключі: /IM - ім'я процесу, /F - примусове
            завершення); такий спосіб завершення підпроцесу
            передбачає виклик ще одного підпроцесу */
            r.exec(new String [] {"cmd", "/c", "start", "taskkill",
                  "/F", "/IM", "\"notepad.exe\""});
            /* альтернативний метод завершення підпроцесу через його
            ID (ключ /PID) (напр. 11308) */
            // r.exec(new String [] {"cmd", "/c", "start",
            //    "taskkill", "/F", "/PID", "11308"});
            System.out.println("Підроцес № 4 завершив свою роботу");
      } catch (Exception ex) {
            ex.printStackTrace();
```

```
            }
            System.out.println("Поточний Java-процес завершив свою
                  роботу");
      }
}
```

```
Підпроцес № 1 завершений. Код завершення: 1
Запуск підпроцесу № 2
Підпроцес № 2 завершений. Код завершення: 0
Запуск підпроцесу № 3
Підпроцес № 3 не завершив своєї роботи
Запуск підпроцесу № 4
Підпроцес № 4 завершив свою роботи
Поточний Java-процес завершив свою роботу
```

Запитання:

1. Що таке процес? Чим відрізняється процес від програми?
2. В яких режимах привілеїв може працювати процес? Яким чином відбувається перемикання між відповідними режимами?
3. За якими критеріями можна класифікувати процеси?
4. Назвіть основні моделі процесу.
5. Дайте визначення багатозадачності та багато поточності. Яким чином можна класифікувати ОС виходячи з даних понять?
6. Що таке перемикання контексту процесу? Квант часу? Паралелізм та псевдопаралелізм?
7. Що мають спільного та відмінного декілька запущених процесів?
8. Що мають спільного та відмінного декілька потоків виконання?
9. Яким чином можна здійснити програмний запуск процесу (як виконуваного так і невиконуваного файлу)? Які методи/функції призначені для керування роботою процесу?
10. Яким чином здійснюється програмний запуск процесу з командного рядка та у віконному режимі?

Вправи*:

1. В залежності від ОС запустити відповідну командну оболонку, в якій вивести повідомлення «Hello World», а також інформацію про всі інстальовані програми в системі.

2. Підпроцес відкриває деякий файл (напр. текстовий або графічний). Перевірити через хвилину, чи в даний файл впродовж даного проміжку часу вносилися зміни. Якщо так, то зачекати доки даний файл не буде закритий користувачем. Якщо ж ні – закрити файл. Результат перевірки вивести в консоль.

3. Користувач з консолі головного процесу вводить назву програми. Перевірити в підпроцесі чи дана програма інстальована в системі. Якщо вона інстальована, то здійснити її запуск.

4. Запустити підпроцес, який кожні 5 хв. очищає корзину і рівно о 12:00 запускає очистку диску, ім’я якого задається користувачем з клавіатури.

5. Головний процес сканує структуру деякого каталогу на наявність файлів. Якщо файли присутні, то кожному із них присвоюється певний символ. Далі користувачу пропонується ввести з клавіатури символ, в залежності від якого, повинен відкритися той чи інший файл в новому підпроцесі.

6. Здійснити на деякому дискові (вводиться користувачем з клавіатури) пошук деякого файлу. Отримані виведені консольні дані помістити в текстовий файл. Запустити перший із знайдених файлів.

7. Користувач вводить із консолі дату та час. Якщо дата співпадає із поточною (тобто сьогоднішньою), то здійснити запуск підпроцесу, який перезавантажує ПК в заданий користувачем час.

8. Батьківський процес породжує два підпроцеси: 1-ий – створює по деякій адресі папку; 2-ий – в новоствореній директорії створює деякий файл, копіюючи його в іншу папку.

9. В головному процесі генеруються 4 підпроцеси. Перший із них архівує деякий файл, другий – переміщує даний архів в деяку папку, третій – розархівовує даний архів, а четвертий – видаляє архів.

10. Намагатися запустити 100 однотипних системних підпроцесів (на власний розсуд). Одночасно із запуском кожного наступного підпроцесу, здійснити моніторинг стану вільного місця в оперативній пам'яті. Як тільки вільної пам'яті стане недостатньо, запуск наступних підпроцесів припинити, а половина із попередньо запущених підпроцесів повинні примусово припинити свою роботу. Після даних маніпуляцій, запустити додатковий підпроцес по діагностиці оперативної пам'яті.

**Примітки:*

- *в завданнях використовувати лише системні нативні процеси ОС (напр. tasklist, cmd, Notepad тощо);*
- *використання можливостей мови програмування (напр. Java) повинно бути зведено до мінімуму (напр., для архівування, зчитування/запису з/в файл тощо слід користуватися не можливостями мови програмування, а відповідними системними процесами). Лише в тих випадках, де неможливо без цього обійтися, власне і застосовувати можливості мови програмування. При цьому, можливостями мови програмування щодо маніпуляції процесами (напр. створення, знищення, очікування тощо) можна користуватися без обмежень;*
- *запуск таких системних процесів здебільшого здійснюється з командного рядка, деякі основні команди якого можна знайти в мережі Internet (див. нижче наведений перелік команд);*
- *детальну інформацію про кожен із процесів можна отримати, набравши в командному рядку, наприклад одну із нижче перелічених команд:*
  - *ім'я програми (напр. defrag);*
  - *help ім'я програми (напр. help format);*
  - *ім'я програми /? (напр. compact /?).*

Перелік деяких з основних команд командного рядка Windows:

- arp – управління кешем ARP;
- at – запуск програм в зазначений час;
- attrib – перегляд та зміна атрибутів файлів/папок;
- auditpol – налаштування політик аудиту;
- backup – створення резервної копії даних;
- cd – зміна поточного каталогу;
- chkdsk – перевірка диска на наявність помилок з подальшим виведенням звіту;
- cipher – шифрування файлів;
- cls – очистка екрану;
- comp – порівняння вмісту декількох файлів;
- compact – перегляд та зміна параметрів стиснення файлів в розділах NTFS;
- convert – перетворення файлової системи (FAT/FAT32 → NTFS);
- copy, xcopy – копіювання файлів та розширене копіювання файлів/каталогів, відповідно;
- curl – взаємодія з URL;
- defrag – дефрагментатор;
- del, erase – видалення файлів;
- dir – перегляд вмісту каталогу;
- diskpart – керування розділами та дисками;
- diskperf – моніторинг продуктивності диска;
- echo – виведення тексту в консоль;
- eventvwr – відкриття переглядача подій;
- exit – закриття командного рядка;
- expand – розпакування стислих файлів;
- fc – порівняння файлів з подальшим виведенням даної інформації;
- find – пошук тексту в одному/декількох файлах;
- findstr – пошук рядків у файлах;
- format – форматування диска;

- ftype – виведення або зміна типів файлів;
- gpupdate – оновлення групових політик;
- hostname – відображення імені комп'ютера;
- ipconfig – інформація про IP-адресу;
- label – зміна мітки диска;
- lpr – створення завдання друку на мереженому принтері;
- lusrmgr.msc – відкриття менеджера користувачів;
- md, mkdir – створення папки;
- move – переміщення одного/декількох файлів з однієї папки в іншу;
- msconfig – налаштування системи;
- net localgroup – керування локальними групами;
- net use – підключення до спільного ресурсу;
- net user – керування обліковими записами користувачів;
- net view – перегляд доступних мережевих ресурсів;
- netstat – перегляд активних мережевих з'єднань;
- nslookup – запит DNS для знаходження IP-адреси;
- perfmon – перегляд моніторингу продуктивності системи;
- ping – перевірка з'єднання з іншим хостом;
- powercfg – керування живленням системи;
- powershell – запуск PowerShell з командного рядка;
- print – друк текстового файлу;
- qprocess – відображення інформації про процеси;
- rd, rmdir – видалення папки;
- recover – відновлення даних на пошкодженому диску;
- regedit – редактор реєстру;
- ren, rename – перейменування файлів/папок і файлу/папки, відповідно;
- replace – заміна файлів;
- robocopy – надійне копіювання файлів з багатьма параметрами;
- route – перегляд або налаштування таблиці маршрутизації;
- runas – запуск команди від імені іншого користувача;

- schtasks – керування запланованими завданнями;
- sfc /scannow – перевірка цілісності системних файлів;
- shutdown – перезавантаження, вимикання комп’ютера і т.п.;
- sort – сортування даних у файлах, з можливим подальшим виводом в консоль;
- start – запуск програми або команди в окремому вікні;
- systeminfo – інформація про конфігурацію системи;
- taskkill – примусове завершення процесів;
- tasklist – перегляд списку запущених процесів;
- taskmgr – відкриття диспетчера завдань;
- time – перегляд/зміна поточного часу;
- title – зміна заголовка поточного вікна командного рядка;
- tracert – визначення маршруту до хоста;
- tree – графічне відображення структури заданого диска/папки;
- tskill – завершення процесу;
- type – виведення вмісту текстових файлів;
- ver, winver – інформація щодо версії Windows;
- vol – виведення інформації про мітку і серійний номер тому для диска;
- wbadmin – команда для резервного копіювання;
- whoami – виведення інформації про поточного користувача.

РОЗДІЛ 2. МІЖПРОЦЕСНА ВЗАЄМОДІЯ

2.1. Теоретична частина

2.1.1. Загальні положення

З точки зору взаємодії, одночасно запущені процеси в ЕОМ бувають двох типів:

- **незалежні** (independed processes) – процеси, які не можуть впливати на роботу один одного і між ними не відбувається обмін інформацією;
- **взаємодіючі** (cooperating processes) – процеси, між якими відбувається взаємодія та обмін даними.

Під обміном даними між паралельно запущеними процесами розуміється власне обмін даними між окремими потоками таких взаємодіючих процесів. Говорять, що процес який надсилає дані являється **постачальником** (відправником), в той час, процес, що такі дані отримує – **споживач** (адресат, одержувач).

У випадку, якщо потоки виконуються в одному процесі, то для обміну даними між ними можна використовувати глобальні змінні і засоби синхронізації потоків. Якщо ж взаємодіючі потоки представляють різні процеси справа йде складніше – потоки не можуть звертатися до загальних змінних і для обміну даними між ними, існують спеціальні засоби ОС.

Обмін даними між взаємодіючими процесами потребує певної проміжної ланки – так званий **канал передачі даних**. Такий канал складається з вхідного/вихідного буферу пам'яті, двох потоків ядра ОС та загальної пам'яті,

доступ до якої мають такі потоки ядра. В загальному випадку, *алгоритм обміну інформацією через канал виглядає наступним чином (рис. 2.1)*:

- потік користувача Т1 записує дані в буфер В1, використовуючи спеціальну функцію ядра ОС;
- перший потік ядра ОС К1 зчитує дані з вхідного буферу В1, записуючи їх в загальну пам'ять М;
- другий потік ядра К2 читає дані з пам'яті М і записує їх в буфер В2;
- потік користувача Т2 зчитує з буфера В2 дані.

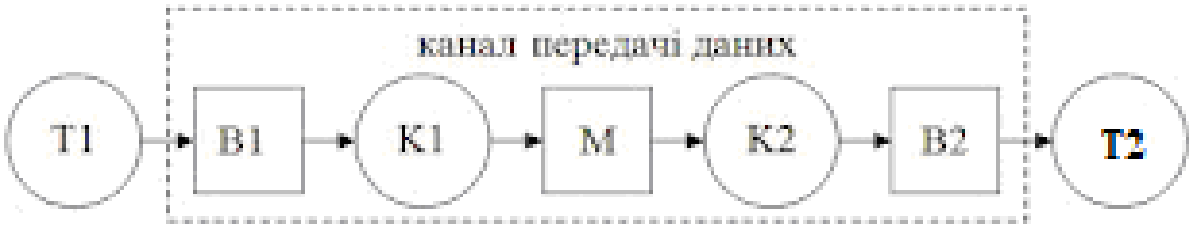


Рис. 2.1. Схема каналу передачі даних.

Слід зауважити, що використання такого каналу дозволяє також обмінюватися інформацією і по мережі. В такому випадку, загальна пам'ять М може розглядатися як передавальне середовище.

Програмно реалізувати такий алгоритм можна наступним чином. Для цього, необхідно потоки ядра ОС та загальну пам'ять замінити файлом. Таким чином реалізується найпростіший канал передачі даних між процесами – використання спільного файлового ресурсу, який буде розглянуто дещо нижче.

Розрізняють два способи передачі даних між взаємодіючими процесами:

- **потоком** – дані передаються неперервною послідовністю байтів. В такому випадку, загальна пам'ять М може бути відсутньою, а надсилання інформації виконується одним потоком ядра безпосередньо з буфера В1 в буфер В2.
- **повідомленням** – дані передаються групою байтів.

З точки зору напряму передачі інформації між процесами розрізняють:

- **напівдуплексний зв'язок** – дані передаються лише в одному напрямку;
- **дуплексний зв'язок** – інформація передається в обох напрямках.

При цьому, розглядаючи лише однонапрямлену передачу даних, *з точки зору топології, розрізняють наступні види зв'язків (рис. 2.2):*

- **1 → 1** (один до одного) – між собою пов'язані лише два процеси (рис. 2.2.а);
- **1 → N** (один до декількох) – один процес пов'язаний з N процесами (рис. 2.2.б);
- **N → 1** (декілька до одного) – кожен з N процесів пов'язаний з одним процесом (рис. 2.2.в);
- **N → M** (декілька до декількох) – кожен з N процесів пов'язаний з кожним з M процесів (рис. 2.2.г).

Таким чином, канали зв’язку ототожнюватимемо із **міжпроцесною взаємодією** (МПВ) (inter-process communication (IPC)), під якою розумітимемо набір методів для обміну даними між потоками взаємодіючих процесів.

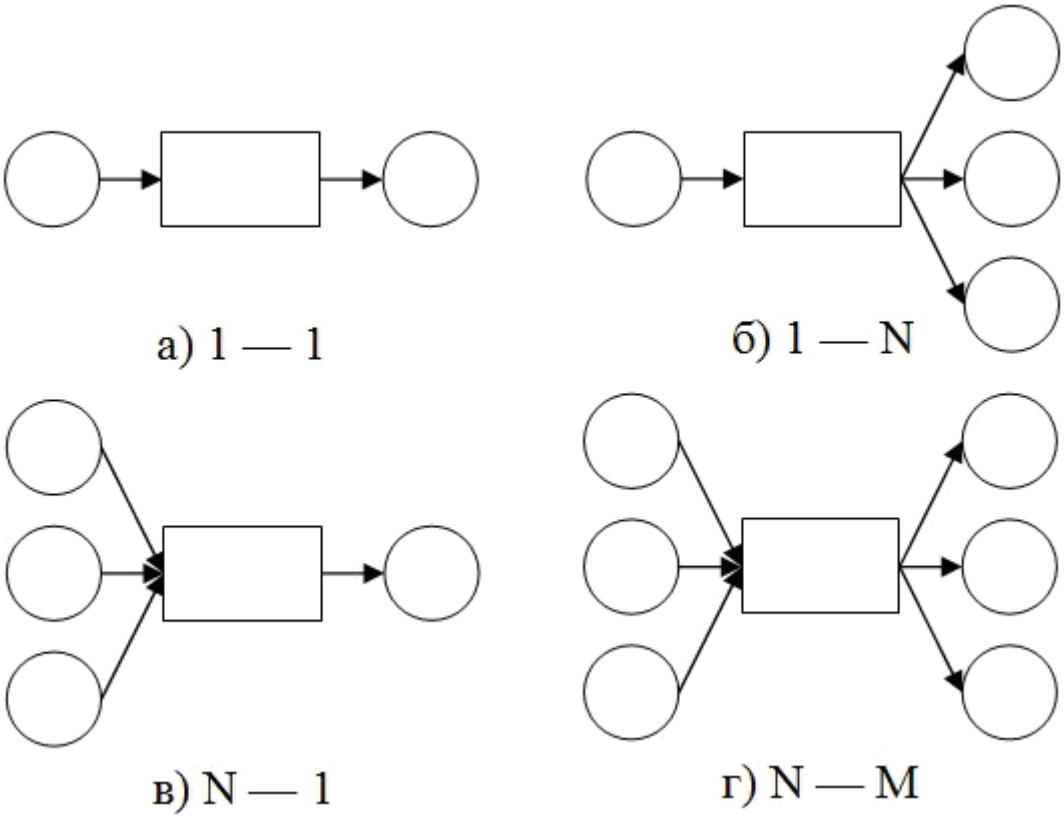


Рис. 2.2. Топології міжпроцесних зв'язків: а) один до одного; б) один до декількох; в) декілька до одного; г) декілька до декількох.

Примітка: кола – процеси, прямокутники – канали зв’язку.

2.1.2. Синхронізація процесів

Як вже зазначалося раніше, операційні системи підтримують таку відособленість процесів: кожний процес має власний адресний простір, кожному процесу призначаються власні ресурси – файли, вікна, семафори та інші. Така відособленість потрібна для того, щоб захистити один процес від іншого, оскільки вони, спільно використовуючи ті чи інші ресурси ЕОМ, конкурують один з одним. В загальному випадку, процеси належать різним користувачам, що розділяють один комп'ютер, і ОС бере на себе роль «арбітра» «в суперечках» процесів за ресурси.

Так, доволі таки часто, процеси одночасно претендують на доступ до одних і ти же ресурсів, серед яких процесорний час, доступ до тих чи інших комірок оперативної пам'яті (ОП), файлів, пристроїв введення/виведення тощо. Такі спільні ресурси називають **критичними секціями**. Зважаючи на це, в ОС повинен бути передбачений механізм, яких дозволяє забезпечити заборону одночасного доступу до спільних ресурсів. В такому випадку говорять про **синхронізацію процесів**, як засіб, що забезпечує почерговий доступ процесів до спільного ресурсу. Синхронізація дозволяє уникнути (або мінімізувати) наприклад таких небажаних ситуацій як:

- **конкуренція процесів (стан гонок)** – одночасний доступ до спільного ресурсу декількома процесами може призвести до порушення логіки їх виконання;
- **взаємне блокування процесів** – ситуація в якій кілька процесів перебувають в стані очікування ресурсів, зайнятих один одним, і жоден з них не може продовжувати своє виконання;
- **голодування** – стан деякого процесу, який не може отримати доступ до критичної секції, оскільки вона зайнята іншим процесом (або декількома). Така ситуація може призвести до того, що процес в стані очікування може чекати ресурс нескінченно довго;

- **зайняте очікування** – часте опитування процесом доступу до критичного розділу. Внаслідок цього, даний процес позбавляє часу на обробку ЦП інших процесів;
- **інверсія пріоритету** – виникає, коли процес з високим пріоритетом, працюючи із критичною секцією, переривається процесом з нижчим пріоритетом. Така ситуація може скластися внаслідок певних обставин і може призвести до серйозних наслідків в системах реального часу;

Синхронізація може підтримуватися як на програмному так і на апаратному рівнях. *До найбільш поширених засобів синхронізації, які підтримуються багатьма ОС, можна віднести: семафори, м'ютекси, бар'єри, блокування та інші*.

2.1.3. Класифікація міжпроцесної взаємодії

Класифікувати міжпроцесну взаємодію можна в залежності від локалізації взаємодіючих процесів:

- *взаємодія процесів в межах однієї ЕОМ*, під управлінням однієї ОС. Механізми такої взаємодії забезпечуються ядром ОС, в якій виконуються процеси. Взаємодія процесів забезпечується всіма підсистемами ядра ОС: підсистема управління введенням/виведенням забезпечує передачу даних між процесами; підсистема управління оперативною пам'яттю розподіляє під процеси спільну ОП, підсистема управління процесами забезпечує синхронізацію виконання процесів та впроваджує механізм обміну сигналів, за допомогою якого процеси повідомляються про виникнення в системі надзвичайних подій. До таких типів МПВ відносять сигнали, канали, обмін повідомленнями, спільна пам'ять тощо.

  Можна виділити наступні *способи одномашинної комунікації*:

  - *взаємодія між двома різними процесами;*
  - *взаємодії процесу з самим собою*. Наприклад, для синхронізації або обміну даними між різними потоками одного процесу.

- *взаємодія процесів, які виконуються на різних ЕОМ*, незалежно від того чи однакова чи ні ОС на запущених машинах. До таких засобів міжмашинної комунікації відносять наприклад сокети, механізм віддаленого виклику процедури та інші.

Існують різні технології міжпроцесної взаємодії. Різноманітність таких підходів зумовлена такими критеріями як продуктивність, системні характеристики ЕОМ, модульність, пропускна здатність мережі і т.д. При цьому, різні механізми МПВ не виключають один одного і можуть бути одночасно реалізовані ОС. Разом з тим, деякі механізми МПВ підтримують обмін даними як в межах однієї машини, так і по мережі (напр. обмін повідомленнями, віддалений виклик процедур). Розглянемо більш детально найбільш поширені типи МПВ.

2.1.3.1. Спільний файловий ресурс

Спільний файловий ресурс передбачає обмін даними між процесами через посередника у вигляді файлу, який розміщується в зовнішній пам'яті (рис. 2.3). Процеси записують/зчитують дані звичайним способом, як це передбачено при роботі з файлами. При цьому, для пришвидшення МПВ, виділяється буфер обміну. Таким чином, робота даного методу складається з наступних етапів: читання з буферу → модифікація даних в буфері → запис у файл.

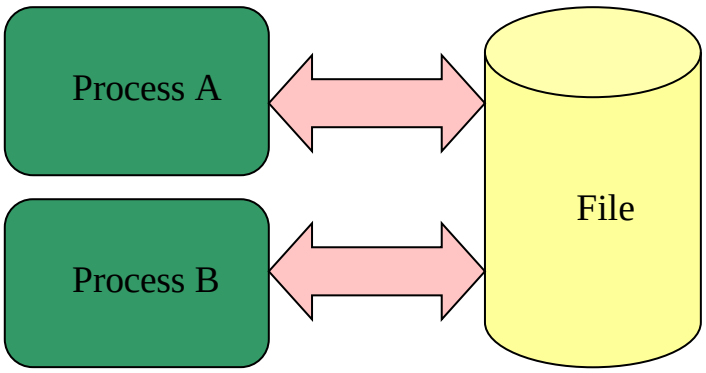


Рис. 2.3. Механізм спільного файлу.

Недоліком такого способу обміну інформацією є його низька продуктивність, за рахунок того, що кожна операція читання/запису є системним викликом. При цьому, необхідне постійне запам'ятовування поточної позиції файлу, вчасно пересуваючи її в позицію, звідки відбуватиметься читання/запис, що також накладає додаткові затрати. Такий метод передбачає використання об’єкта синхронізації (напр. семафора), задля запобігання неконтрольованого доступу до даних файлу з боку процесів, між якими власне і відбувається МПВ.

2.1.3.2. Відображення файлу в пам'ять

Відображення файлу в пам'ять (memory-mapped file, file mapping) – спосіб роботи з файлами, при якому, всьому файлу або деякій його безперервній частині, співставляється у відповідність певна ділянка ОП (рис. 2.4):

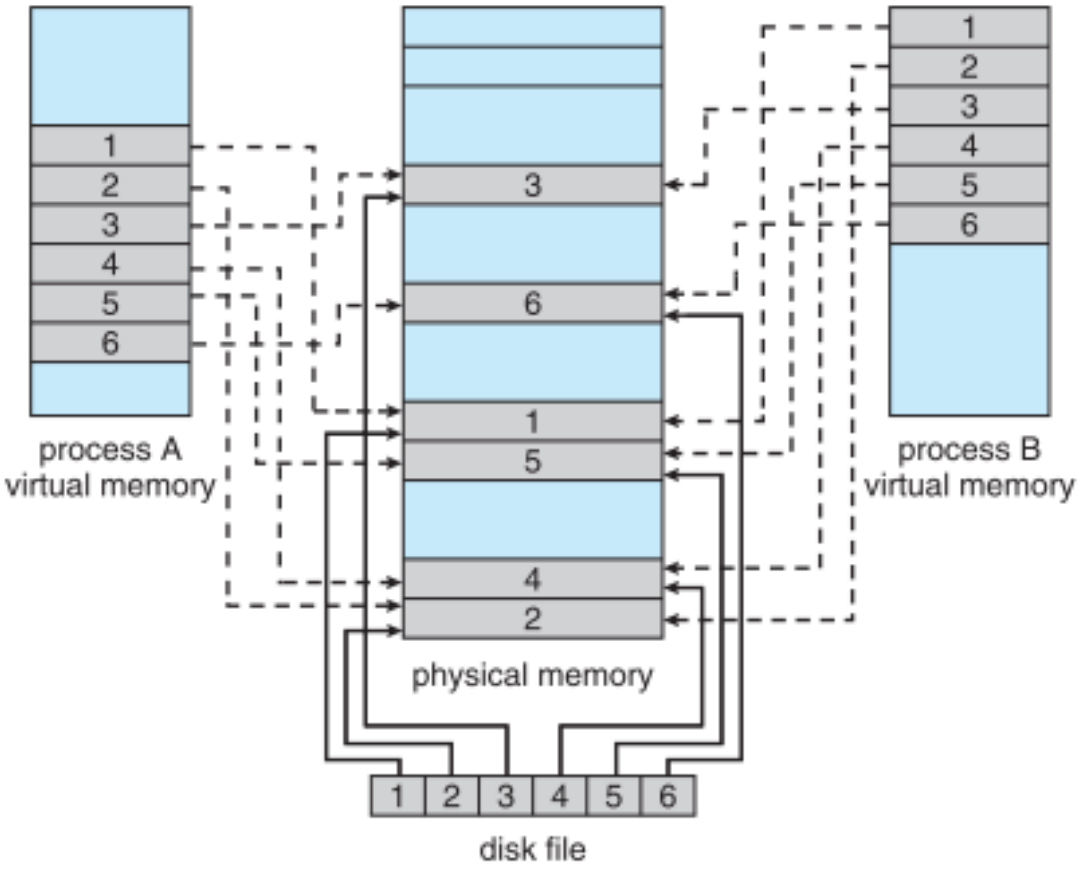


Рис. 2.4. Механізм відображення файлу в пам’ять.

При цьому, читання даних з таких адрес пам’яті, фактично призводить до читання даних з відображеного файлу, в той час як запис даних за цими адресами, здійснюється у файл. Відображати можна як звичайні файли, так і файли пристроїв. При такій МПВ, декілька процесів одночасно відображають один і той же фізичний файл на свою пам’ять, власне після чого, звертаються до неї. Як і у випадку обміну даними через файл, відображення файлу в пам’ять також повинно передбачати синхронізацію. Даний спосіб МПВ є одним із найбезпечніших, за рахунок того, що унеможливлюється виникнення виключних ситуацій.

В порівнянні зі звичайним читанням/записом, метод відображення є більш продуктивним і має наступні переваги:

- на відміну від МПВ через файл, яка передбачає 3-етапний процес, метод відображення складається лише з 1-го етапу – модифікації даних в певній області пам'яті;
- менше навантаження ОС – відображається одразу не весь файл, а при необхідності, його певна частина (блоками розміром в сторінку пам’яті). Таким чином, навіть маючи невелику кількість фізичної пам'яті (наприклад, 32 МБ), можна легко відобразити файл розміром 100 МБ або більше;
- виграш при записі з пам'яті на диск – якщо користувач оновив велику кількість даних в пам'яті, вони можуть одночасно (за один прохід головки над диском) бути записані на диск.

В свою чергу, недоліками даного методу являються те, що:

- розмір файлу відображення залежить від архітектури системи;
- такий спосіб дозволяє проводити МПВ лише в межах однієї локальної машини, і не може бути застосований для передачі інформації в мережі.

Окрім МПВ, відображення файлів використовується також для завантаження процесу в пам’ять, коли власне файл процесу завантажується в пам’ять.

2.1.3.3. Канали

Канали (pipe) – це абстракції у вигляді проміжних ланок, які дозволяють процесам обмінюватися інформацією, забезпечуючи механізми для синхронізації та організації такого обміну. *Канали бувають двох типів:*

- **неіменовані** (анонімні) – односторонній канал зв'язку між двома взаємопов'язаними процесами (батьківським і дочірнім). Здебільшого представляє собою структуру, що знаходиться в зовнішній пам'яті (у файловій системі). Зазвичай батьківський процес відкриває такий канал, підключаючись до одного з його кінців, куди власне і надсилає дані. В той же час, дочірній процес підключається до іншого кінця каналу, з якого зчитує дані, що були надіслані від батьківського процесу. Інформація, що зчитується одразу ж видаляється і не може бути повторно доступною для зчитування. При цьому, взаємодіючі процеси можуть бути переведені в стан очікування при спробі запису в переповнений, або зчитування даних з пустого каналу. Такий канал діє по принципу FIFO – дані, що були надіслані першими, першими і зчитуватимуться. Для двохстороннього обміну даними необхідно створювати два неіменовані канали (рис. 2.5):

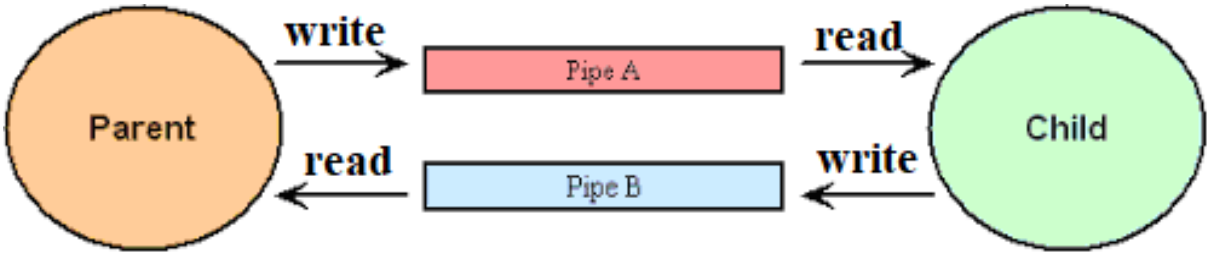


Рис. 2.5. Механізм неіменованих каналів.

При цьому, батьківський процес може створювати декілька дочірніх процесів, об'єднуючи їх в **конвеєр** (рис. 2.6) – структуру, в якій вихід кожного процесу послідовно пов'язаний із входом кожного наступного процесу, тим самим забезпечуючи МПВ. Дані, що передаються по каналах буферизуються ОС, допоки їх не прочитає наступний процес. При цьому, як тільки один із процесів завершує своє виконання, пов'язані з ним канали зникають.

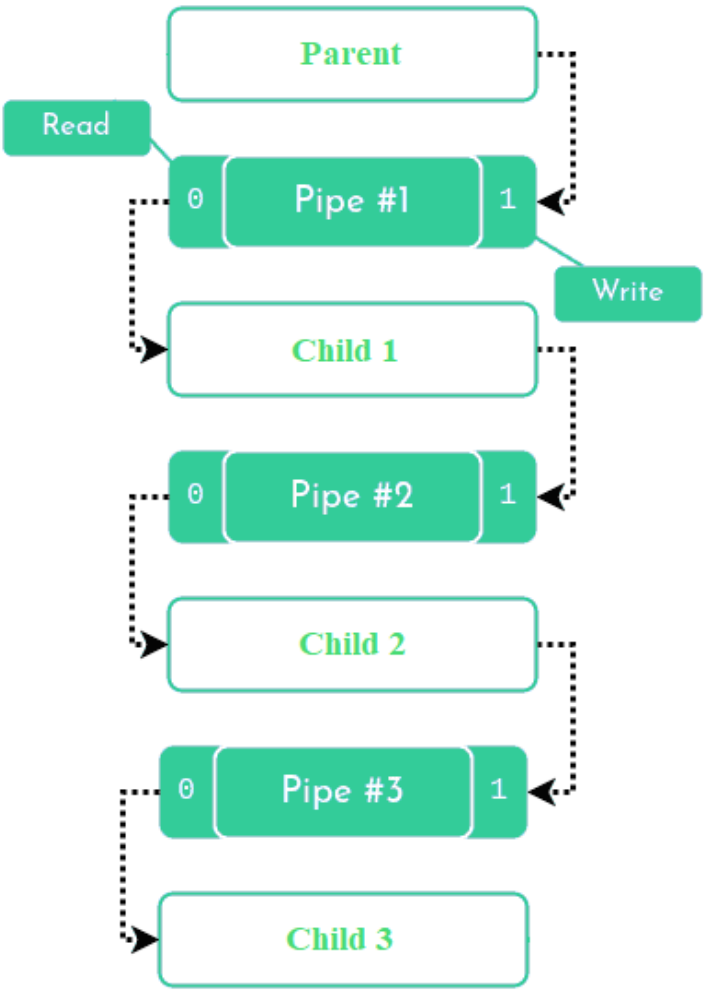


Рис. 2.6. Механізм конвеєра процесів.

- **іменовані** (FIFO-файли) – концепція дещо схожа з неіменованими каналами. Вони також є об'єктами файлової системи зі стратегією FIFO, які можуть об’єднувати взаємодіючі процеси в конвеєр. З точки зору ядра ОС, іменований канал являє собою спеціалізований файл з власним ім'ям, якому відповідає той чи інший запис в каталозі. Виклик такого каналу здійснюється по його імені, що на відміну від неіменованих каналів, дозволяє здійснювати обмін даними між непов’язаними між собою «чужими» процесами. Для такого каналу характерно те, що для процесів необхідно надавати права доступу до такого об’єкта, щоб визначити, хто матиме можливість записувати/зчитувати дані. При цьому, декілька процесів можуть одночасно записувати або зчитувати дані в/з канал(у) (рис. 2.7):

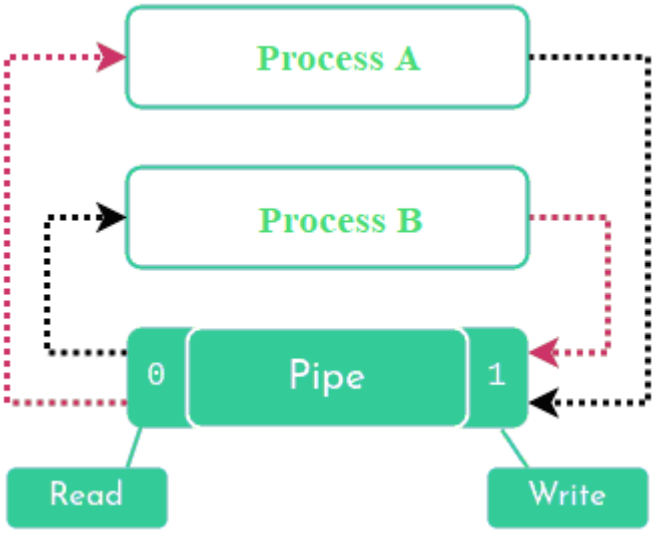


Рис. 2.7. Механізм іменованих каналів.

Режим роботи іменованого каналу може бути як напівдуплексним (процеси можуть здійснювати комунікацію лише в одному із напрямків) так і повнодуплексним. Об’єкт іменованого каналу буде існувати навіть після завершення роботи процесів, які до нього підключені, а тому, після обміну інформацією, його необхідно закривати. Іменований канал дозволяє різним процесам обмінюватися даними, навіть якщо програми, що виконуються в цих процесах, спочатку не були написані для взаємодії з іншими програмами. Типове застосування іменованих каналів – розробка «**клієнт-серверних**» додатків. Сервером виступає процес, який створює канал, в той час, як клієнт підключається до такого каналу. Іменовані канали можуть використовуватися як для обміну даними між процесами, що запущені на одній машині, так і для МПВ різних ЕОМ.

2.1.3.4. Спільна пам’ять

Спільна пам’ять (shared memory) – механізм, який забезпечує взаємозв’язок між декількома (як пов’язаними так і непов’язаними) процесами, яким надається спільний доступ до деякого сегмента ОП (рис. 2.8). Сегмент пам'яті, що розділяється, створюється деяким процесом, розміщуючись у його вільній частині віртуального адресного простору. Після створення спільного

сегмента пам'яті, будь-який з користувальницьких процесів має змогу приєднати його до свого власного віртуального простору і працювати з ним як зі звичайним сегментом пам'яті. Таким чином, різні процеси можуть мати різні адреси однієї і тієї ж комірки, яка належить пам'яті, що розділяється. При цьому, відправлення даних відбувається шляхом запису в спільну пам'ять, а їх отримання – читанням. Такий тип МПВ може бути досить таки корисним наприклад для обміну великими обсягами даних (напр. відео- чи аудіо-потоки).

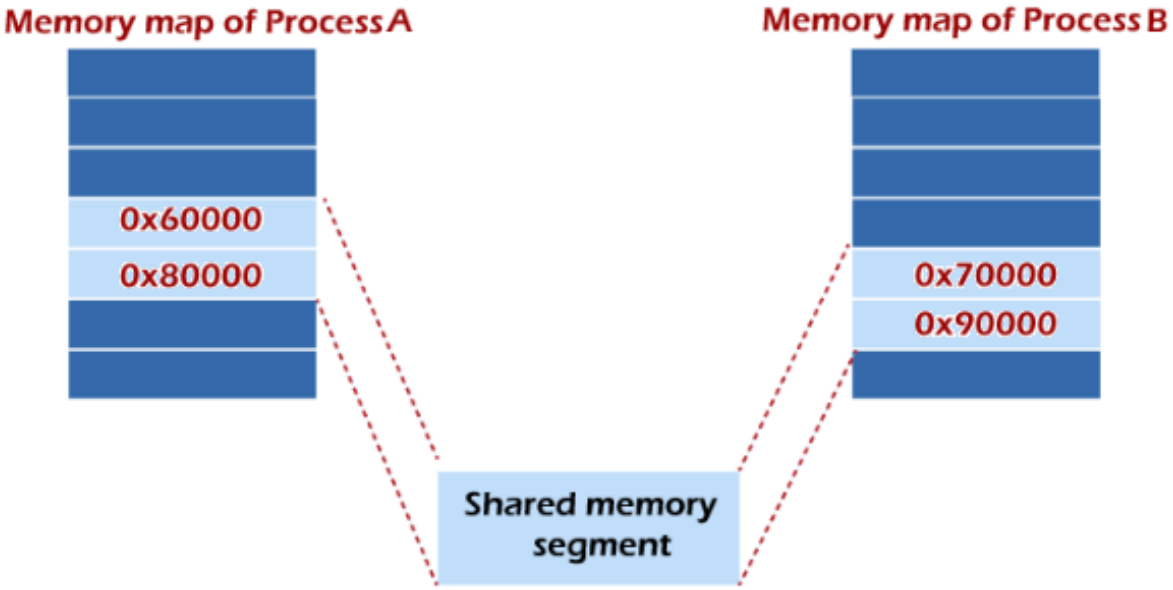


Рис. 2.8. Механізм спільної пам’яті.

Варто зауважити, що всі інші засоби МПВ здійснюють обмін інформацією через ядро ОС, тим самим знижуючи продуктивність, за рахунок перемикання контексту між процесом і ядром. В той же час, механізм спільної пам’яті не використовує системні виклики ядра, роблячи його найшвидшим засобом МПВ.

Недоліком даного методу є те, що процеси повинні бути запущені на одній машині, проте якщо вони виконуються на різних процесорах, їх архітектура повинна бути кеш-когерентною (властивість кешу, яка забезпечує цілісність даних, що зберігаються в локальних кешах для роздільного ресурсу). Іншим недоліком слугує відсутність будь-яких засобів синхронізації, проте для подолання такого недоліку, можна використовувати техніку семафорів.

2.1.3.5. Сигнали

Сигнали (signals) – одна із форм МПВ, яка використовується в ОС, сумісними зі стандартом POSIX. Концепція проявляється в надсиланні одним процесом сигналу (системного повідомлення) для іншого процесу або певному потоку в межах того самого процесу (рис. 2.9):

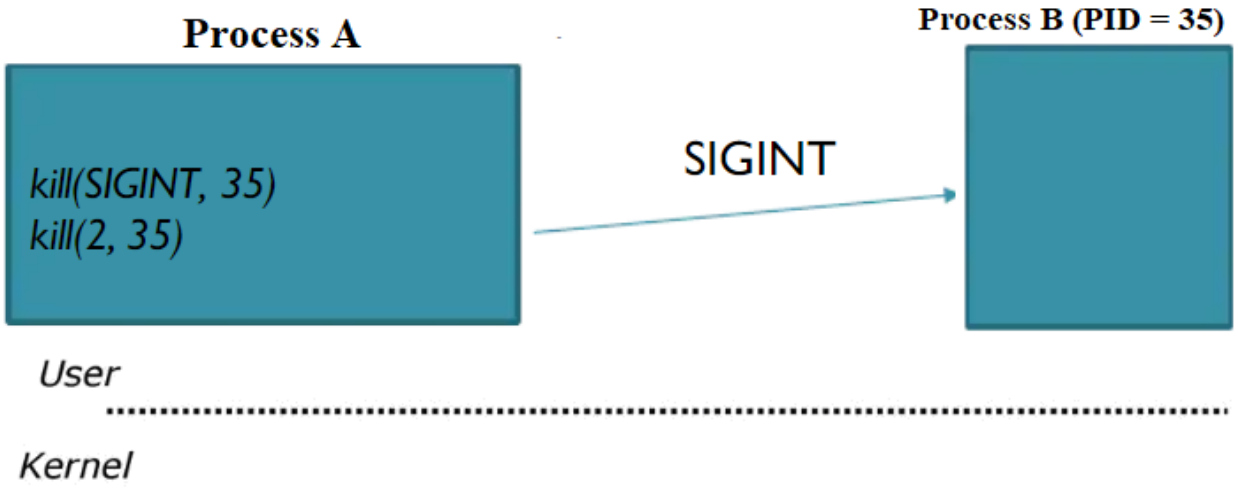


Рис. 2.9. Механізм сигналів: процес А посилає сигнал SIGINT (числова константа 2) для процесу В з ідентифікатором 35.

Примітка: kill – UNIX-утиліта (системний виклик), що посилає процесу сигнал.

Такі сигнали не дозволяють передати дані і надсилаються задля сповіщення про настання певної події. Зазвичай сигнали являють собою асинхронні (неочікувані) повідомлення, хоча і можуть бути синхронними (напр. помилка обчислень з плаваючою комою, помилка адресації). Здебільшого сигнали використовуються для переривання, призупинення, припинення або завершення процесу. Як тільки сигнал надсилається, ОС здійснює переривання виконання цільового (якому здійснюється надсилання) процесу, запускаючи деяку підпрограму – **обробника сигналу**. Такий обробник можна задати в цільовому процесі, або ж якщо він не описаний, буде здійсненний запуск стандартного обробника по-замовчуванню. Процес може вказати дві поведінки за замовчуванням: ігнорування сигналу і власне обробник сигналу за замовчуванням (для більшості сигналів завершує виконання процесу). Сигнали

досить таки схожі на переривання, проте різниця полягає в тому, що переривання опосередковуються процесором і обробляються ядром, в той час як сигнали опосередковуються ядром і обробляються окремими процесами. Відправником сигналів може слугувати: один процес іншому (або самому собі), натискання спеціальних клавіш в терміналі, ядро ОС (при виникненні помилкових системних викликів або апаратних переривань, інформування про подію введення/виведення, збій апаратних засобів).

Класифікувати сигнали можна по трьом групам:

- **системні сигнали** (помилки апаратури, системні помилки);
- **сигнали від пристроїв**;
- **користувацькі сигнали**.

Кожному сигналу відповідає певне ціле число, яке в залежності від реалізації системи може відрізнятися. Список сигналів визначений в єдиній специфікації UNIX. Серед прикладів можна навести наступні: сигнал переривання процесу, сингал спливання часу очікування, невірне звернення до фізичної пам'яті, недопустима інструкція процесора, зависання, користувацький сигнал та багато інших (рис. 2.10):

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
$
```

Рис. 2.10. Список сигналів викликаний командою kill –l.

Головною перевагою сигналів являється їх простота, оскільки їх реалізація не потребує використання будь-яких структур даних (напр. черги, буфери) для передачі або зберігання інформації. Серед основних напрямків використання

сигналів є реалізація базових механізмів синхронізації та міжпроцесної взаємодії, а також обробка різноманітних виняткових ситуацій (переривання, помилки).

В свою чергу, недоліком сигналів являється їхня ненадійність і непередбачуваність. Сигнали можуть бути втрачені, дубльовані або змінені операційною системою, залежно від типу сигналу, політики доставки сигналу та завантаження системи. Враховуючи те, що сигнали можуть в будь-який момент перервати виконання процесу, вони можуть перешкоджати нормальному виконанню процесу, спричинивши неузгодженість, пошкодження або взаємоблокування, якщо процес не готовий належним чином обробляти надісланий для нього сигнал. Що стосується налагодження і тестування, то тут сигнали також можуть бути складними у використанні, оскільки їх важко відтворювати та контролювати.

2.1.3.6. Обмін повідомленнями

Обмін повідомленнями (massage passing) – найпопулярніша концепція, яка передбачає відсутність спільних ресурсів між процесами, які обмінюються даними у вигляді повідомлень. Процес який надсилає повідомлення являється **відправником**, в той час як процес, що його отримує – **отримувачем**. При цьому, відправник має змогу надсилати повідомлення одному або декільком одержувачам. Відправлення та прийом повідомлень зазвичай реалізується відповідними системними викликами тієї чи іншої ОС, що робить даний метод дещо повільнішим (напр. в порівнянні з методом спільної пам’яті), проте легшим в реалізації. Всі без виключення повідомлення надсилаються відправником в ядро ОС, з якого вони в подальшому отримуються отримувачем (рис. 2.11).

Примітка: форма повідомлень залежить від підтримки ОС та мови програмування і може виступати у вигляді викликів функцій (віддалений виклик процедур), сигналів, пакетів даних тощо. Проте, кожна із таких форм зазвичай розглядається як окремий вид МПВ.

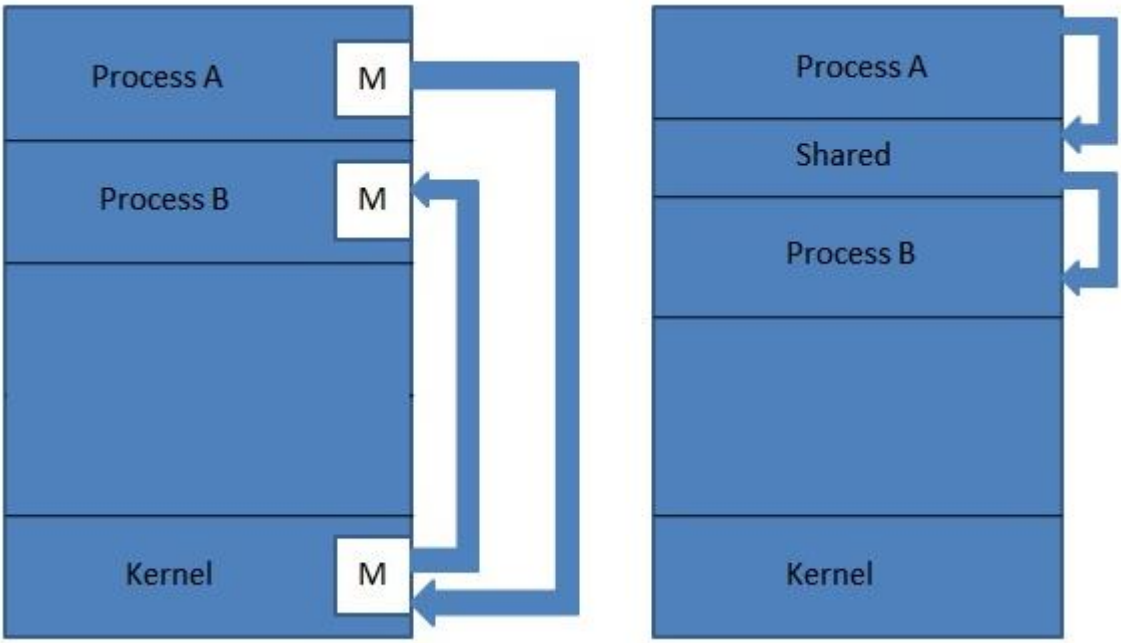


Рис. 2.11. Механізми обміну повідомленнями та спільної пам’яті.

Передача повідомлень використовується в різноманітних цілях, серед яких: обміну даними між процесами, координація дій кількох процесів і реалізація складних протоколів зв’язку між процесами. Однією з причин популярності передачі повідомлень є її здатність підтримувати взаємодію клієнт-сервер. Найпоширенішим стандартизованим інструментом, що дозволяє проводити таку МПВ являється інтерфейс передачі повідомлень (Message Passing Interface, MPI), який реалізований на багатьох мовах програмування, серед яких C/C++, Java, Python тощо. Механізм повідомлень відноситься до числа основних концепцій системи Windows (заповнення вікон програми будь-якими зображеннями, відгук на вибір пунктів меню, управління діалоговими вікнами).

Перевагою даного методу є те, що обмін даними може відбуватися як в межах однієї ЕОМ так і в розподілених мережевих комп’ютерних системах. Серед інших переваг, які забезпечує даний механізм, – високий рівень інкапсуляції та розподілу, що робить код чистішим, коротшим і легшим для розуміння та обслуговування.

В свою чергу, недоліком такого типу МПВ є те, що при передачі даних по мережі, повідомлення може бути втрачено. Виходом із такої ситуації може

слугувати алгоритм, при якому отримувач при отриманні повідомлення, повинен у відповідь надіслати спеціальне підтвердження. Якщо ж таке підтвердження не буде отримане отримувачем впродовж певного інтервалу часу, відправник повинен повторно надіслати аналогічне повідомлення, яке було втрачено.

Стандартне повідомлення зазвичай складається з двох частин:

- **заголовка** – містить тип повідомлення, ідентифікатори відправника та отримувача, довжину повідомлення, керуючу інформацію (пріоритет, порядковий номер і дії з малим дисковим простором), контрольну суму (інформацію про цілісність даних) та інше;
- **тіло** – містить оригінальне повідомлення, яким потрібно обмінятися.

Якщо ж повідомлення не містить заголовка, тоді воно містить тип та довжину даних і власне сам дані.

Повідомлення можуть бути:

- **синхронними** – відправник відправляє повідомлення і чекає доки отримувач отримає повідомлення (до отримання підтвердження), після чого продовжує свою роботу. В той же час, якщо отримувач ще не отримав повідомлення, функція отримання повідомлення блокує роботу коду отримувача до тих пір, доки повідомлення не буде отримане. Таким чином, така відправка повідомлень є блокуючою, забезпечуючи одночасну відправку/отримання повідомлення для взаємодіючих процесів. Розробники також впроваджують політику **тайм-аутів**, за рахунок того, що отримувач з тих чи інших причин може взагалі не отримати повідомлення. В такому випадку, відправник чекатиме доки отримувач отримає повідомлення певний проміжок часу, після чого, якщо все-таки не було отримане повідомлення, відправник розблокується, продовживши свою роботу;
- **асинхронними** – відправник відправляє повідомлення і продовжує далі виконання своїх інструкцій, не очікуючи доки отримувач отримає дане повідомлення. При цьому, отримувач буде заблокованим в інструкції для

отримання повідомлення доти, доки таке повідомлення не буде отриманим. Таким чином відправка не являється блокуючою операцією, в той час, як отримання є блокуючим. Такий обмін інформацією дозволяє відправити/отримати повідомлення в різні проміжки часу;

- **гібридними** – в такому випадку, відправник чекатиме доти, доки отримувач буде готовий отримати повідомлення і лише тоді надішле повідомлення.

При цьому, *відправка повідомлення може відбуватися наступними способами:*

- **пряма передача повідомлень** – відправник надсилає повідомлення заздалегідь вказаному адресату, по певному його ідентифікатору. Сама процедура ідентифікації відправника/отримувача називається **адресацією** *і може бути двох типів*:
  - **симетрична** – окрім того, що відправник вказує одержувача, одержувач теж повинен вказати ідентифікатор процесу, який надсилає йому інформацію. В такому разі, в найпростішому випадку, системні виклики (базові примітиви) відправки/отримання виглядатимуть наступним чином:
    - send(P, message) // відправка повідомлення для процесу P
    - receive(Q, message) // отримання повідомлення від процесу Q
  - **несиметрична** – отримувач може отримувати повідомлення від будь-кого, не обов’язково називаючи ідентифікатор відправника. В такому разі, примітиви надсилання/отримання записуються як:
    - send(P, message) // відправка повідомлення для процесу P
    - receive(id, message) /* отримання повідомлення від будь-якого процесу; id встановлюється на ім'я процесу, з яким відбулося спілкування */

Базові примітиви слугують основою для більш потужних засобів мережної комунікації (напр. розподілена файлова система, служба виклику віддалених процедур) (рис. 2.12):

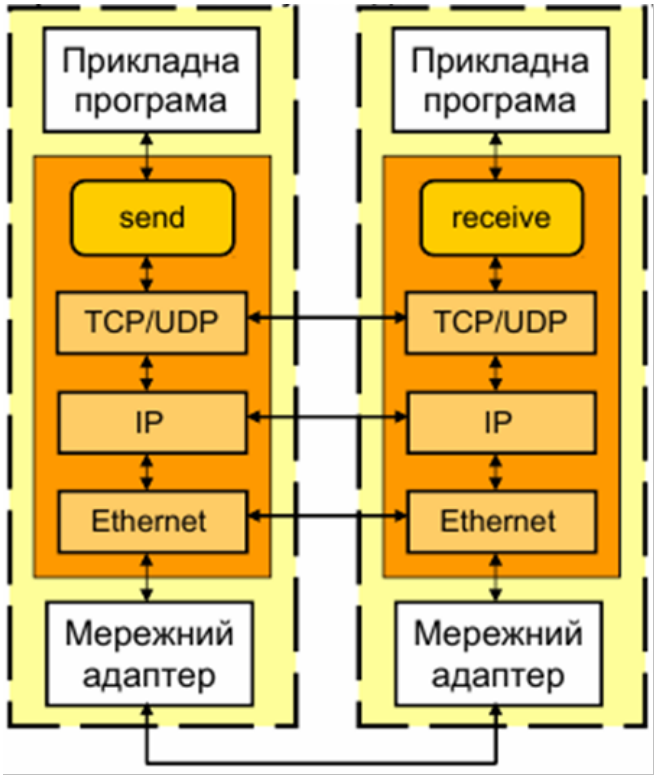


Рис. 2.12. Примітиви обміну повідомленнями і мережеві протоколи.

Прикладом прямого обміну є традиційні сигнали. Серйозним недоліком обробки прямих повідомлень є обмежена модульність – зміна ідентифікатора процесу-отримувача вимагає внесення змін до кожного відправника, які мають зв’язок з таким процесом;

- **непряма передача повідомлень** – надсилання повідомлень між процесами через посередника у вигляді електронної (поштової) скриньки (mailbox, структура даних, яка представляє собою буфер для конкретної кількості повідомлень, яка зазвичай вказується при його створенні) або порту (port, (апаратний порт)/(порт логічного зв’язку)). Така концепція МПВ зумовлена тим, що в однопроцесорних ЕОМ відправник і отримувач не можуть працювати одночасно. В той же час, нема також гарантії, що вони одночасно працюватимуть і в багатопроцесорних системах, а тому, виникає необхідність структури даних, що зберігатиме надіслані, проте ще не отримані повідомлення.

Спочатку відправник надсилає повідомлення на електронну скриньку, яка в свою чергу надсилає дане повідомлення для одержувача. При цьому,

задіяних процесів (відправників/отримувачів) може бути декілька, при чому, вони можуть бути задіяні також через декілька поштових скриньок.

Системні виклики при непрямій передачі повідомлень записуються як:

- send(A, message) // відправка повідомлення на електронну адресу A
- receive(A, message) // отримання повідомлення з електронної адреси A

Саме непрямий обмін даними використовується в більшості сучасних технологій обміну повідомленнями.

В свою чергу, *повідомлення можуть бути наступного характеру*:

- **фіксованого розміру** – проста фізична реалізація, проте ускладнюється програмний код;
- **змінного розміру** – проста програмна реалізація з більш складною фізичною складовою;
- **типові повідомлення** – повідомлення, що застосовуються при непрямому зв'язку через електронну пошту.

В свою чергу, в багатьох випадках міжпроцесної передачі повідомлень використовується підхід буферизації даних. Тоді говорять про **чергу повідомлень** (message queue), як тимчасовий буфер, в якому зберігаються повідомлення надіслані відправником, допоки вони не будуть отримані отримувачем (рис. 2.13):

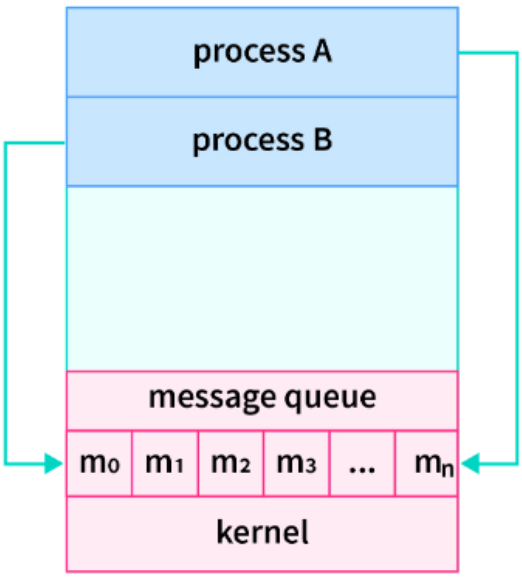


Рис. 2.13. Механізм черги повідомлень.

Така черга генерується відправником, до якої отримують доступ повідомлення, які приймають дані. Черга повідомлень представляє собою зв'язаний список, який розміщений в ядрі ОС, з яким одночасно може працювати декілька процесів. ОС підтримує спеціальну область ОП, в якій зберігаються черги повідомлень (напр. UNIX-подібні ОС підтримують до 1024 таких черг), кожна з яких має власний ідентифікатор.

Такі черги можуть бути реалізованими одними із наступних дисциплін:

- **FIFO** – перше записане повідомлення буде прочитане першим;
- **LIFO** – останнє записане повідомлення буде прочитане першим;
- **пріоритетний доступ** – повідомлення мають пріоритети і в залежності від їх пріоритетності, «витягуються» ті чи інші повідомлення;
- **довільний доступ** – повідомлення зчитуються в довільному порядку.

Існують наступні три типи реалізації таких черг:

- **нульова ємність** – буферизація в системі відсутня (довжина черги рівна нулю). Недоліком такого підходу є те, що якщо приймач працює повільно, дані будуть втрачені, оскільки відправник повинен чекати, доки одержувач отримає повідомлення. В противному випадку, дані будуть втрачені. Звідси випливає, що взаємодіючі процеси повинні бути синхронізовані;
- **обмежена ємність** – черга такого типу має заздалегідь визначену довжину n. Відправник може надсилати повідомлення, поки чергу не буде заповнено;
- **необмежена ємність** – обмеження на довжину буфера відсутнє. В такій черзі може потенційно знаходитися нескінченна кількість повідомлень і відправник ніколи не чекатиме.

2.1.3.7. Віддалений виклик процедур

Віддалений виклик процедур (ВВП) (Remote procedure call, RPC) – механізм МПВ, який передбачає комунікацію між процесами, які знаходяться в різних адресних просторах (зазвичай на різних ЕОМ (різні фізичні адреси) або ж менш поширено – реалізованими в межах однієї ЕОМ (взаємодіючі процеси

мають різні віртуальні адресні простори, навіть якщо фізичний адресний простір однаковий). В такому випадку, деякий процес №1 здійснює виклик підпрограми (функції/процедури/методу), яка «прописана» в іншому процесі №2. Така форма взаємодії є типом **«клієнт-сервер»** (виклик – клієнт, виконавець – сервер), що зазвичай реалізується у вигляді передачі повідомлень «запит-відповідь». У такій моделі МПВ лише один із двох процесів активний у будь-який момент часу. Клієнт здійснює запит (синхронний/асинхронний), для виконання певної процедури з наданими параметрами, надсилаючи повідомлення для віддаленого сервера. Після виконання такої процедури на запит, сервер надає повідомлення-відповідь назад для клієнта. Зазвичай повідомлення містить транспортну адресу, яка включає в себе номери мережі, хосту та порту. ВВП передбачає існування транспортного протоколу низького рівня (напр. TCP/IP або UDP) і для підвищення продуктивності, представлений на сеансовому та представницьких рівнях.

Дана технологія є досить таки поширеною і реалізується такими технологіями як CORBA, Java RMI, DCOM, ONC RPC, .NET Remoting, SOAP, MRPC, gRPC тощо.

Головною перевагою даного методу є те, що він забезпечує абстракцію – деякий процес може викликати процедуру у віддаленій системі так само, як би він викликав локальну процедуру, проте мережеві деталі приховані від користувача. Така абстракція зумовлена використанням так званих заглушок, внаслідок чого, процес розробки є набагато простішим. Проте, якщо в разі **локального виклику процедури** (LPC), процес передає параметри в викликану процедуру та отримує результат роботи через стек або загальні області пам’яті, то в разі видаленого виклику, передача параметрів відбувається у вигляді запиту по мережі, що вимагає обов’язкового використання транспортного рівня мережевої архітектури. При цьому, на відміну від локального виклику процедури, віддалені виклики можуть зазнати краху через ті чи інші проблеми із мережею. Реалізація віддалених викликів є набагато складнішим процесом в порівнянні з локальними процедурами. Як було зазначено, на відміну від

локального виклику процедури, віддалений виклик використовує транспортний рівень мереженої архітектури, проте це залишається прихованим від розробника.

Недоліком даної технології є те, що обмін даними відбувається здебільшого на різних машинах і те, що кожна із них може мати різну програмну (ОС) і/або апаратну архітектуру. При цьому, ускладнювати ситуацію може також і те, що взаємодіючі процеси можуть бути написані на різних мовах програмування, які до того ж всього можуть використовувати різні технології ВВП. Саме тому, був прийнятий один із стандартів – мова опису інтерфейсу (IDL), яка є «містком» між взаємодіючими ЕОМ, дозволяючи проводити обмін даними незалежно від їх архітектури (рис. 2.14). Використання IDL хоч і далеко не завжди забезпечує сумісність клієнта із сервером, проте слугує кроком вперед до вирішення даної проблеми.

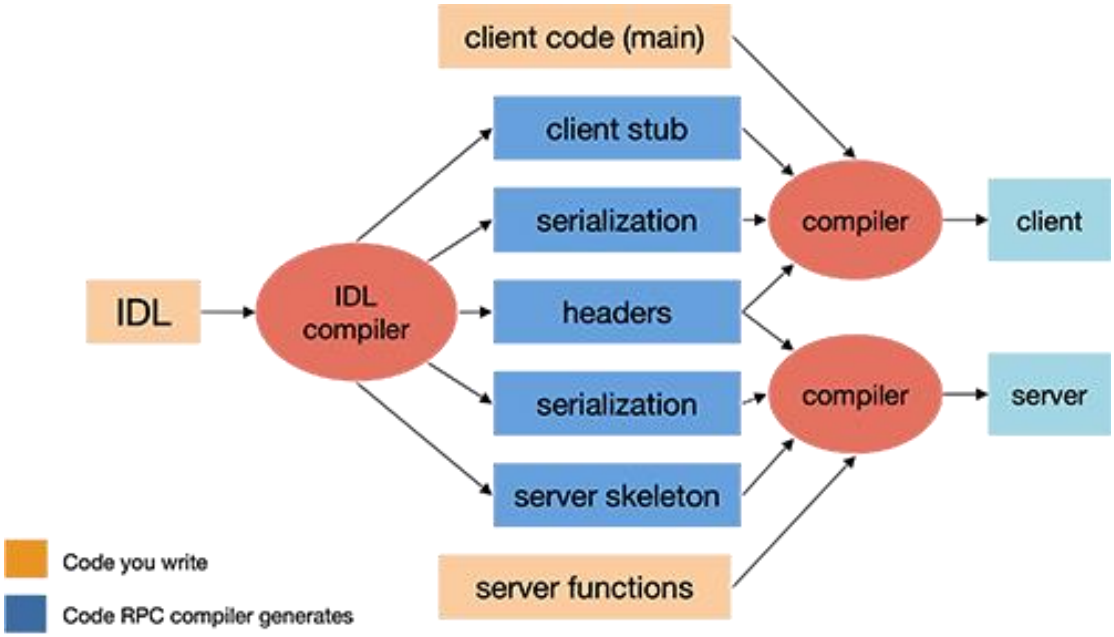


Рис. 2.14. Етапи компіляції для віддалених викликів процедур.

Іншим недоліком може слугувати можливий збій в мережі, що може призвести до повторного виклику віддаленої процедури і як наслідок – можливе її виконання декілька разів, що в деяких випадках може бути неприпустимо. Процедури, які можна виконати декілька разів, в результаті яких

не відбудеться небажаних ефектів, називаються **ідемпотентними**. В противному випадку, кажуть про **неідемпотентні процедури**.

Враховуючи те, що ВВП відбувається через мережу, важливим елементом такої взаємодії являється безпека. Тому, багато технологій ВВП передбачає шифрування даних, накладення на них цифрового підпису, авторизації, аутентифікації та інших складових інформаційної безпеки.

Наступний алгоритм характеризує послідовність дій ВВП (рис. 2.15):

- клієнтський процес викликає **заглушку** (стаб (stub)), яка знаходиться в його адресному просторі, ніби він викликає звичайну процедуру. Заглушка являє собою локальну процедуру, яка упаковує аргументи, які використовуватимуться в параметрах віддаленої процедури. При цьому, така упаковка повинна мати деякий стандартизований вигляд задля того, щоб в подальшому сервер «зрозумів» передані для нього дані. Запаковані дані формують одне або декілька повідомлень. Такий процес упаковки називається **маршалізацією** (marshaling) і вимагає **серіалізації** всіх елементів даних (зазвичай у формат плоского масиву байтів).

  Примітки:

  - *параметри що передаються, можуть бути представлені лише за значенням, проте не в якості посилання, оскільки посилання це адреса пам'яті, яка на стороні клієнта і сервера може не співпадати. Внаслідок цього, хорошого рішення по даному питанню ще допоки не існує;*
  - *упарядковані дані, що передаються через мережу можуть містити дані, які роблять їх самоописними: ідентифікація окремих параметрів за назвою та типом. Такий формат даних відповідає **явній типізації**, прикладом якого є JSON і XML. Навпаки, **неявна типізація** не надсилає таку інформацію і вимагає від віддаленої сторони знати точну очікувану послідовність параметрів;*

- заглушка звертається до ядра ОС, в якій запущений клієнт. Ядро в свою чергу надсилає повідомлення на віддалену машину (ядра ОС сервера). Всі

звернення до ядра, в тому числі і на стороні сервера, здійснюються при допомозі переривань та системних викликів. Передача даних по мережі відбувається по певному протоколу (зазвичай TCP або UDP) з використанням сокетів.

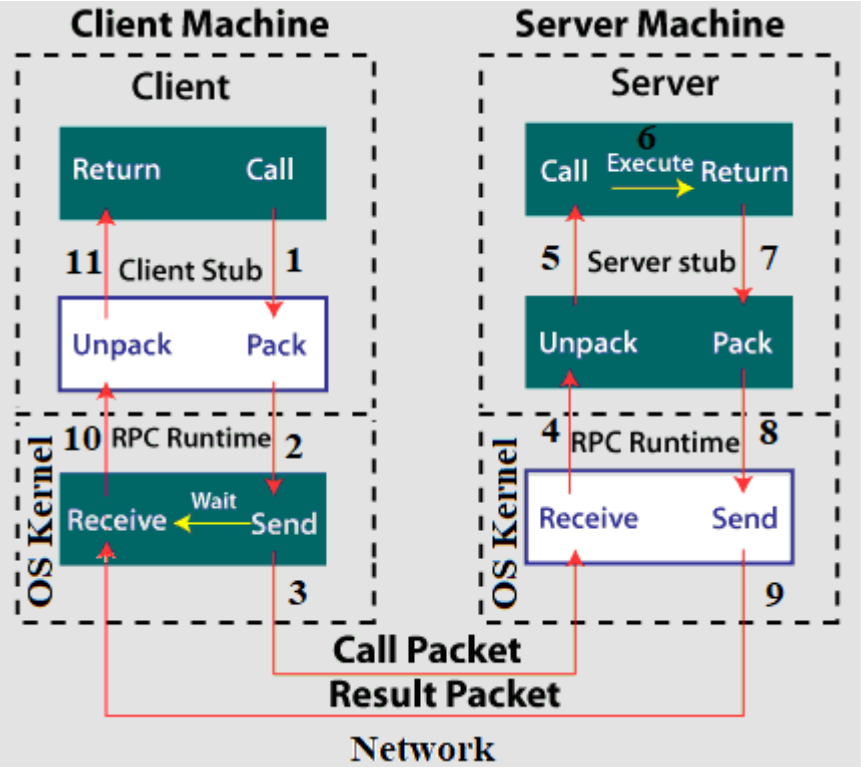
Примітки:

- *за керування передачею повідомлень через мережу (як в одну так і в другу сторону) відповідає **середовище виконання ВВП** (RPC Runtime), яке присутнє як на стороні клієнта так і на стороні сервера. Такий елемент являє собою бібліотеку підпрограм і служб, які обробляють мережеві комунікації, що лежать в основі механізму ВВП (контроль зв'язку між клієнтами та серверами, пошук серверів для клієнтів за запитом, маршрутизація, обробка помилок зв'язку, шифрування тощо);*
- *щоб визначити транспортну адресу сервера, клієнт звертається до **сервера імен** – мережевої служби, яка надає хост і порт потрібного віддаленого інтерфейсу. В такому випадку, клієнт надсилає серверу ім'я, яке ідентифікує інтерфейс. Таке ім'я досить часто є унікальним числом, ідентифікуючи службу, яка містить набір функцій на сервері. В свою чергу, сервер повертає хост і номер порту для служби, яка реалізує функції. В багатьох випадках, сервер імен знаходиться на машині, де виконуються віддалені процедури, і сервер повертає лише номер порту. Коли служба запускається, вона реєструє свої інтерфейси на сервері імен ВВП.*

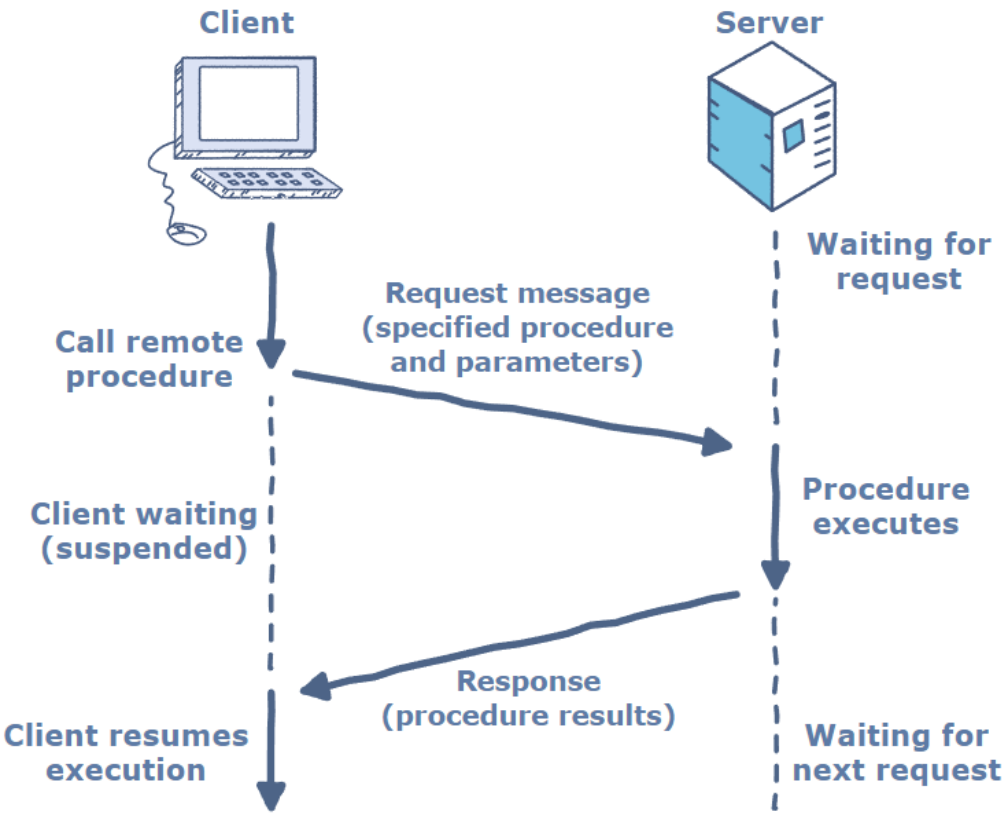
  *Вище описана процедура визначення адреси називається **динамічним зв'язуванням** і передбачає етапи іменування (сервер експортує інтерфейс, задля того, щоб клієнти могли ним користуватися) та розташування (клієнт повинен імпортувати наданий сервером інтерфейс перед початком зв'язку).*

- ядро ОС сервера передає отримане повідомлення до заглушки на стороні сервера (скелета);

- сервер-заглушка розпаковує отримане повідомлення (**демаршалізація/десеріалізація**);
- на основі розпакованих параметрів, серверна заглушка здійснює виклик необхідної процедури (на стороні клієнта, вона являється процедурою віддаленого виклику);
- результат виконання такої процедури повертається назад на заглушку сервера;
- сервер-заглушка запаковує отриманий результат;
- заглушка сервера передає дані для ядра ОС сервера;
- ядро ОС сервера передає повідомлення для ядра ОС клієнта;
- ядро ОС клієнта звертається до його заглушки, яка розпаковує отриманий результат;
- отриманий результат виконання віддаленої процедури передається процесу клієнта;
- клієнт продовжує свою роботу.



а)



б)

Рис. 2.15. Етапи виконання (а) та часова шкала (б) віддаленого виклику процедури.

Існує декілька типів ВВП:

- **синхронний ВВП** – процес на стороні клієнта надсилає запит, після чого, відбувається його блокування до моменту, доки сервер не надішле відповідь. Як тільки відповідь надійшла, клієнт продовжує свою роботу;
- **асинхронний ВВП** – на відміну від синхронного виклику віддалених процедур, процес-клієнт надіславши дані для сервера, не очікує у відповідь серверні дані і продовжує свою роботу, не блокуючись. Асинхронний ВВП часто використовується у випадку, коли клієнту потрібно продовжити виконання без блокування або ж коли серверу може знадобитися багато часу для підтвердження;
- **ВВП зворотного виклику** (Callback RPC) – забезпечує парадигму P2P (мережа рівноправних вузлів) між взаємодіючими процесами. Така концепція дозволяє процесу бути як клієнтом так і сервером;

- **широкомовний ВВП** (Broadcast RPC) – клієнт здійснює запит в мережі, який обробляють всі сервери, здатні обробити даний запит. Такий тип МПВ дозволяє вказати, що запит клієнта має бути широкомовним, оголошуючи при цьому відповідні широкомовні порти. Широкомовний ВВП часто використовується, коли декілька серверів можуть обробити запит, і клієнт хоче отримати відповіді від усіх них;
- **ВВП пакетного режиму** (Batch-mode RPC) – дозволяє на стороні клієнта поставити в чергу декілька запитів, відправивши їх в подальшому на сервер одним пакетом. Використання такого типу ВВП дозволяє мінімізувати навантаження на систему, за рахунок зменшення кількості запитів. Проте, для такої МПВ потрібен надійний протокол передачі даних. При цьому, ВВП пакетного режиму ефективний лише для процесів з низькою швидкістю запитів. ВВП пакетного режиму часто використовується, коли клієнту потрібно зробити кілька запитів до сервера, проте немає необхідності отримувати негайні відповіді.

При цьому, *ВВП може бути представлене трьома рівнями:*

- **верхній** – абсолютно прозорий для ОС та мережі. Виклик віддаленої процедури відбувається так само як і звичайної функції;
- **середній** – розробник сам визначає віддалену процедуру та організовує кодування даних, що передаватимуться через мережу. Проте, безпосередньо з сокетами, програміст справи не має;
- **нижній** – розробник детально програмує ВВП з усіма її тонкощами. Даний підхід використовується досить таки рідко.

Найбільша ефективність використання ВВП досягається в тих застосунках, в яких існує інтерактивний зв'язок між віддаленими компонентами з невеликим часом відповідей і відносно малою кількістю даних, що передаються.

Що ж стосується сфери використання, то дана технологія досить таки широко використовується в розподілених обчисленнях та хмарних сервісах.

Варто зауважити, що окрім вище перелічених різновидів МПВ існують і інші, які характерні для тих чи інших ОС.

2.2. Практична частина

2.2.1. Міжпроцесна взаємодія

Методи get(Input/Output/Error)Stream

За замовчуванням, створені підпроцеси не мають власного терміналу або консолі. Всі їх стандартні операції введення/виведення (stdin, stdout, stderr) перенаправлені до каналу, пов'язаного з батьківським процесом. Доступ до даних такого каналу можна здійснити через потоки, отримані за допомогою методів, getOutputStream(), getInputStream() і getErrorStream(). Батьківський процес використовує такі потоки для передачі вхідних і отримання вихідних даних від підпроцесів. Оскільки деякі ОС надають обмежений розмір буфера для стандартних потоків введення/виведення, повільний запис вхідного або зчитування вихідного потоків деякого підпроцесу може призвести до блокування такого підпроцесу, або ж навіть до тупикової ситуації. Тому такі дані необхідно якнайшвидше передавати.

При необхідності, джерело/адресат введення/виведення підпроцесів також можна перенаправляти з допомогою методів класу ProcessBuilder, який буде розглянуто дещо згодом.

Нижче представлений можливий алгоритм розробки програмного коду процесів, які передбачають між собою обмін даними. Така послідовність дій є досить таки умовною, оскільки не враховує можливе перенаправлення стандартних входів/виходів взаємодіючих процесів, тип даних, що передаються та інше.

№	Етапи алгоритму	Приклад програмного коду
Батьківський процес		
1.	Створення об'єкта виконавчого середовища.	`Runtime r = Runtime.getRuntime();`
2.	Запуск на виконання підпроцесу.	`Process pro = r.exec(new String[] {"java", "-classpath", "C:\\Users\\User\\eclipse-workspace\\SubProcess\\bin\\", "SubProcess"});`
3.1	Створення потоку	`OutputStream os = pro.getOutputStream();`

	виведення, з метою запису даних в підпроцес.	OutputStreamWriter osw = new OutputStreamWriter(os); /* додатково можна скористатися буферизацією: BufferedWriter bw = new BufferedWriter(osw); */
3.2	Надсилання даних в підпроцес.	osw.write(5); // bw.write(5);

3.3	Закриття новостворених потоків виведення.	os.close();
		osw.close();
		// bw.close();

4.1	Створення потоку введення, з метою зчитування даних з підпроцесу.	InputStream is = pro.getInputStream(); InputStreamReader isr = new InputStreamReader(is); /* додатково можна скористатися буферизацією: BufferedReader br = new BufferedReader(isr); */
4.2	Отримання даних від підпроцесу.	isr.read(); // br.read();

4.3	Закриття новостворених потоків введення.	is.close();
		isr.close();
		// br.close();

Дочірній підпроцес		
1.1	Створення об'єкта Scanner, пов'язаного зі стандартним входом.	Scanner s = new Scanner(System.in);
1.2	Отримання даних від батьківського процесу.	s.nextInt();
1.3	Закриття новоствореного потоку введення.	s.close();
2.	Надсилання даних в батьківський процес.	System.out.println(10);

Зрозуміло, що даний алгоритм може видозмінюватися, в залежності від поставленої задачі. Зокрема, може відбуватися як одно- так і двосторонній обмін даними. При цьому, будь-хто із учасників МПВ може першим ініціювати надсилання інформації. В такому разі, інший учасник буде першим зчитувати дані.

Лістинг 2.1. Одностороння передача даних від системного Windows-підпроцесу tasklist до поточного процесу.

```
import java.io.*;

public class ProcessInputStreamDemo {

	static Runtime r = Runtime.getRuntime();
	static Process pro;
	static String line;

	public static void main(String args[]) {
		try {
			/* Windows-утиліта tasklist для отримання інформації про
			запущені в ОС процеси */
			pro = r.exec("tasklist.exe /fo csv /nh");
			/* байтовий потік введення, який отримується з потоку
			виведення даних підпроцесу pro */
			InputStream is = pro.getInputStream();
			// символьний потік введення даних
			InputStreamReader isr = new InputStreamReader(is);
			/* буферизований символьний потік введення даних;
			P.S.: варто зауважити, що проводити обмін даними між
			процесами можна не лише посимвольно, а й побайтово */
			BufferedReader bri = new BufferedReader(isr);
			// зчитування даних, що надходять від підпроцесу
			while ((line = bri.readLine()) != null)
				if (!line.trim().equals("")) {
					// отримання лише імені процесу
					line = line.substring(1);
					System.out.println(line.substring(0,
						line.indexOf("\"")));
				}
			is.close();
```

```
                isr.close();
                bri.close();
            }
            catch (Exception e) {
                System.out.println("Помилка поточного процесу при роботі
                    з tasklist");
            }
        }
}
```

```
System Idle Process
System
Secure System
Registry
smss.exe
csrss.exe
wininit.exe
…
```

Лістинг 2.2. Двосторонній обмін даними між поточним процесом і двома Java-підпроцесами.

Головний (поточний) процес ProcessInputOutputErrorStreamDemo:

```
import java.io.*;

public class ProcessInputOutputErrorStreamDemo {

    static Runtime r = Runtime.getRuntime();
    static Process pro;
    static String line;
    static BufferedReader bri, bre;
    static BufferedWriter bw;
```

```
/* метод для обміну даними з Java-підпроцесом SubProcess6 */
static void runUserSubProcess4() {
      String javaFileName = "SubProcess4";
      String classFilePath = "C:\\Users\\User\\eclipse-workspace\\
            SubProcess4\\bin\\";
      try {
            pro = r.exec(new String[] {"java", "-classpath",
                  classFilePath, javaFileName});
            // буферизований символьний потік введення даних
            bri = new BufferedReader(new InputStreamReader
                  (pro.getInputStream()));
            /* буферизований символьний потік введення, який
            утворюється з потоку виведення помилок підпроцесу pro */
            bre = new BufferedReader(new InputStreamReader
                  (pro.getErrorStream()));
            // буферизований символьний потік виведення даних
            bw = new BufferedWriter(new OutputStreamWriter
                  (pro.getOutputStream()));
            // надсилання даних в підпроцес
            bw.write("hello!!!\n");
            /* очищення буферного потоку, на випадок якщо він
            переповнився (що описано в документації);
            в даній ситуації може він і не знадобиться, проте при
            запису великих даних, це може стати в нагоді */
            bw.flush();
            bw.close();
            System.out.println("Дані надіслані");
            /* зчитування даних (як звичайних, так і помилок), що
            надходять від підпроцесу */
            while ((line = bri.readLine()) != null ||
                  (line = bre.readLine()) != null)
                  System.out.println("Підпроцес надіслав: " + line);
            /* P.S.: отримати дані буферу і вивести їх в консоль
            можна також і нижче поданими 4-ма інструкціями */
```

```
            // bri.lines().forEach(System.out::println);
            // bre.lines().forEach(System.out::println);
            // pro.getInputStream().transferTo(System.out);
            // pro.getErrorStream().transferTo(System.out);
            System.out.println("Дані отримані");
            bri.close();
            bre.close();
      }
      catch (Exception e) {
            System.out.println("Помилка поточного процесу при роботі
                  з SubProcess4");
      }
}

/* метод для обміну даними з системною командою зміни дати */
static void runSystemSubProcess() {
      try {
            /* зміна системної дати;
            P.S.: така команда може не діяти, якщо вона викликається
            не під правами адміністратора; в такому разі, даний
            процес необхідно запустити не з IDE, а з cmd з правами
            адміністратора */
            pro = Runtime.getRuntime().exec("cmd /c date");
            bw = new BufferedWriter(new OutputStreamWriter
                  (pro.getOutputStream()));
            bri = new BufferedReader(new InputStreamReader
                  (pro.getInputStream()));
            bw.write("25-05-2023");
            bw.close();
            while ((line = bri.readLine()) != null)
                  System.out.println(line);
            bri.close();
      } catch (Exception e) {
            System.out.println("Помилка поточного процесу при зміні
```

```
                    дати");
            }
      }

      public static void main(String[] args) {
            System.out.println("Обмін даними між користувацьким
                  підпроцесом");
            runUserSubProcess4();
            System.out.println("\nОбмін даними між системною командою");
            runSystemSubProcess();
      }
}
```

Java-підпроцес SubProcess4:

```
import java.util.Scanner;
// import java.io.*;

public class SubProcess4 {
      public static void main(String[] args) {
            try {
                  Scanner readConsole = new Scanner(System.in);
                  /* обмін даними:
                  1) зчитування з консолі даних (nextLine()), надісланих
                  від батьківського процесу; хоча і батьківський процес
                  надсилає дані одноразово, можна перестрахуватися в
                  циклі, на випадок, якщо таких даних буде надіслано
                  декілька разів (hasNextLine());
                  2) надсилання даних для батьківського процесу
                  (System.out.println) */
                  while (readConsole.hasNextLine())
                        System.out.println(readConsole.nextLine().
                              toUpperCase());
                  /* P.S.: нижче описані інструкції - альтернатива для
```

```
                попередніх дій */
                // BufferedReader br = new BufferedReader
                //    (new InputStreamReader(System.in));
                // String line;
                // while ((line = br.readLine()) != null)
                //    System.out.println(line.toUpperCase());
                // br.close();
                readConsole.close();
            }
            catch(Exception e) {
                System.out.println("Помилка згенерована в SubProcess4");
            }
            /* навмисно створена помилка (доступ до неіснуючого
            аргументу), щоб показати, як батьківський процес отримує
            інформацію про помилки */
            System.out.println(args[0]);
        }
}
```

Консоль поточного Java-процесу ProcessInputOutputErrorStreamDemo:

```
Обмін даними між користувацьким підпроцесом
Дані надіслані
Підпроцес надіслав: HELLO!!!
Підпроцес надіслав: Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: Index 0 out of bounds for
length 0
Підпроцес надіслав:   at SubProcess4.main(SubProcess4.java:29)
Дані отримані

Обмін даними між системною командою
The current date is: 29.03.2023
Enter the new date: (dd-mm-yy) 25-05-2023
```

2.2.2. Конструктор процесів. Атрибути та перенаправлення вводу/виводу процесів. Конвеєр процесів

Клас ProcessBuilder

Для створення і керування роботою процесів ОС також передбачається використання класу ProcessBuilder (Java 8) (пакет java.lang.ProcessBuilder):

public final class ProcessBuilder extends Object,

який надає додаткові можливості в порівнянні з Process, керуючи рядом атрибутів процесу.

Даний клас передбачає два конструктори (табл. 2.1):

Таблиця 2.1. Конструктори класу ProcessBuilder.

Конструктор	Опис дій конструктора
public ProcessBuilder(List<String> command) (NullPointerException)	Створює об'єкт ProcessBuilder. В якості параметру використовується список рядків command, який містить дані щодо підпроцесу: ім'я програми (зовнішній програмний файл) та всі необхідні аргументи командного рядка, якщо такі є.
public ProcessBuilder(String... command)	Аналогічний попередньому, проте дані підпроцесу задаються в якості параметра змінної довжини.

Опис методів, якими оперує клас ProcessBuilder, представлений в нижче наведеній таблиці. При цьому, окрім таких методів, клас наслідує від Object наступні методи: clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait.

Таблиця 2.2. Методи класу ProcessBuilder.

Метод	Опис дій метода
public List<String> command()	Повертає посилання на список List з ім'ям програми та її аргументами.
public ProcessBuilder command(List<String> command) (NullPointerException)	Встановлює ім'я програми та її аргументи, у вигляді списку command. Зміна в даному списку відноситься до викликаючого об'єкта.

	Повертає викликаючий об'єкт.
public ProcessBuilder command(String... command)	Аналогічно попередньому, проте ім'я програми та її аргументи задаються в якості рядкового масиву.
public File directory()	Повертає робочий каталог для викликаючого об'єкта. Якщо каталог співпадає з поточним каталогом процесу Java, повертається null. По замовчуванню, робочим каталогом являється поточний робочий каталог поточного процесу.
public ProcessBuilder directory(File directory)	Встановлює робочий каталог directory для викликаючого об'єкта. Якщо каталог співпадає з поточним каталогом процесу Java, встановлюється null. Метод повертає посилання на викликаючий об'єкт.
public Map<String, String> environment() (SecurityException)	Повертає змінні середовища, що пов'язані з викликаючим об'єктом у вигляді колекції пар «ключ-значення».
public ProcessBuilder inheritIO()	Встановлює джерело та адресат для стандартного вводу і виводу підпроцесу однаковими з поточним процесом Java. Метод повертає посилання на викликаючий об'єкт.
public ProcessBuilder.Redirect redirectError()	Повертає адресат для стандартного потоку виведення помилок у вигляді об'єкта ProcessBuilder.Redirect.
public ProcessBuilder redirectError(File file)	Встановлює файл file, як адресат для стандартного потоку виведення помилок. Метод повертає посилання на викликаючий об'єкт.
public ProcessBuilder redirectError(ProcessBuilder.Redirect destination) (IllegalArgumentException)	Аналогічно попередньому, проте адресатом являється об'єкт destination типу ProcessBuilder.Redirect.
public boolean redirectErrorStream()	Перевіряє на співпадіння стандартних потоків виведення помилок та даних для даного об’єкта. Якщо їхні адресати співпадають, повертає true, в іншому разі - false.

public ProcessBuilder redirectErrorStream(boolean redirectErrorStream)	Перенаправляє стандартний потік виводу помилок в стандартний потік виведення даних, у разі якщо redirectErrorStream = true. Якщо ж redirectErrorStream = false, дані потоки мають різних адресатів (за замовчуванням). Метод повертає посилання на викликаючий об'єкт.
public ProcessBuilder.Redirect redirectInput()	Повертає джерело для стандартного потоку введення у вигляді об'єкта ProcessBuilder.Redirect.
public ProcessBuilder redirectInput(File file)	Встановлює файл file, як джерело для стандартного потоку введення. Метод повертає посилання на викликаючий об'єкт.
public ProcessBuilder redirectInput(ProcessBuilder.Redirect source) (IllegalArgumentException)	Аналогічно попередньому, проте джерелом являється об'єкт source типу ProcessBuilder.Redirect.
public ProcessBuilder.Redirect redirectOutput()	Повертає адресат для стандартного потоку виведення у вигляді об'єкта ProcessBuilder.Redirect.
public ProcessBuilder redirectOutput(File file)	Встановлює адресат для стандартного потоку виведення у вигляді файлу file. Метод повертає посилання на викликаючий об'єкт.
public ProcessBuilder redirectOutput(ProcessBuilder.Redirect destination) (IllegalArgumentException)	Аналогічно попередньому, проте адресатом являється об'єкт destination типу ProcessBuilder.Redirect.
public Process start() throws IOException (IndexOutOfBoundsException, NullPointerException, SecurityException)	Запускає новий процес, використовуючи атрибути конструктора. Метод повертає об’єкт Process для управління таким підпроцесом.

Кожен екземпляр класу ProcessBuilder керує набором атрибутів відповідного процесу (змінні середовища, робочий каталог, джерело/адресат вводу/виводу тощо). Як видно з таблиці, методом для створення процесів являється start(). Даний метод дозволяється багаторазово викликати з одного екземпляру задля створення нових підпроцесів з ідентичними або пов’язаними

атрибутами. На відміну від інструкції Runtime.getRuntime().exec(), починаючи з Java 1.5, команда ProcessBuilder.start() є бажанішим способом запуску підпроцесів.

Варто зауважити, що зміна атрибутів конструктора процесів вплине лише на процеси, які в подальшому будуть запущені методом start() даного об'єкта, в той час, як ніколи не вплине на попередньо запущені процеси або ж поточний Java-процес.

Багато з методів даного класу дозволяють перенаправляти дані підпроцесу. Так, як зазначалося дещо раніше, по-замовчуванню підпроцес записує/читає вихідні/вхідні дані з/в канал(у). Доступ до даних такого каналу відбувається з допомогою потоків (методи getOutputStream(), getInputStream() і getErrorStream() класу Process). Однак, такий стандартний ввід/вивід може бути перенаправлений в інші місця призначення (напр. методи redirectOutput, redirectInput, redirectError), що часто буває досить таки корисним.

Також слід звернути увагу на те, що даний клас не є синхронізований. У випадку, якщо декілька потоків ProcessBuilder одночасно отримують доступ до екземпляру, і принаймі один із таких потоків структурно змінює деякі із атрибутів, його необхідно синхронізувати зовні.

Варто зауважити, що в Java 9 було доповнено клас ProcessBuilder методом:
static List<Process> startPipeline(List<ProcessBuilder> builders),
який запускає процес для кожного екземпляру ProcessBuilder, створюючи конвеєр процесів, пов'язаних їхніми стандартними потоками вводу/виводу. В такому випадку, вихід 1-го процесу пов'язується із входом 2-го, вихід 2-го процесу пов'язується з входом 3-го і т.д. При цьому, всі пов'язані між собою потоки введення/виведення процесів мають бути Redirect.PIPE (обмінюватися даними через канали) і є недоступними. Доступними для переспрямування являються лише вхід 1-го і вихід останнього процесу, які ініціалізуються за допомогою параметрів перенаправлення відповідного ProcessBuilder.

Якщо під час запуску будь-якого з процесів конвеєра виникає виняткова ситуація, всі процеси примусово знищуються.

Клас ProcessBuilder.Redirect

Характерною рисою методів класу ProcessBuilder являється те, що багато з них повертають значення типу ProcessBuilder.Redirect (Java 8). Даний клас (пакет java.lang.ProcessBuilder.Redirect) являється вкладеним в ProcessBuilder і представляє джерело/адресат введення/виведення підпроцесу, відповідно:

public abstract static class ProcessBuilder.Redirect extends Object,

Клас ProcessBuilder.Redirect не передбачає конструкторів. Містить наступні поля (табл. 2.3.а) та методи (не враховуючи унаслідувані від Object методи clone, finalize, getClass, notify, notifyAll, toString, wait) (табл. 2.3.б):

Таблиця 2.3.а. Поля класу ProcessBuilder.Redirect.

Поле	Опис поля
static ProcessBuilder.Redirect INHERIT	Вказує на те, що джерело/адресат вводу/виводу підпроцесу будуть такими самими, що й в поточного процесу. Це нормальна поведінка більшості інтерпретаторів команд ОС (оболонок).
static ProcessBuilder.Redirect PIPE	Вказує на те, що ввід/вивід підпроцесу буде підключено до поточного процесу через канал. Це типова обробка стандартного введення/виведення підпроцесу.

Таблиця 2.3.б. Методи класу ProcessBuilder.Redirect.

Метод	Опис дій метода
public static ProcessBuilder.Redirect appendTo(File file) (NullPointerException)	Переспрямовує запис у вказаний файл file, повертаючи дане перенаправлення ProcessBuilder.Redirect. Кожна операція запису спочатку просуває позицію до кінця файлу, а потім записує дані. Чи виконується просування позиції та запис даних за одну атомарну операцію, залежить від системи.
public boolean equals(Object obj)	Порівнює вказаний об’єкт obj з даним перенаправленням на рівність. Повертає true лише

	тоді, коли обидва об'єкти ідентичні або вони є Redirect-екземплярами одного типу, пов'язаними з ненульовими рівними екземплярами File.
public File file()	Повертає об'єкт File, який пов'язаний з даним переспрямуванням.
public static ProcessBuilder.Redirect from(File file) (NullPointerException)	Переспрямовує зчитування з вказаного файлу file, повертаючи дане перенаправлення.
public int hashCode()	Повертає значення хеш-коду даного перенаправлення.
public static ProcessBuilder.Redirect to(File file) (NullPointerException)	Переспрямовує запис у вказаний файл file, повертаючи дане перенаправлення. Якщо вказаний файл існував на момент запуску підпроцесу, його попередній вміст буде перезаписано.
public abstract ProcessBuilder.Redirect.Type type()	Повертає тип перенаправлення.

Із вище поданих таблиць видно, що кожен екземпляр Redirect є одним з:

- Redirect.PIPE (див. табл. 2.3.а);
- Redirect.INHERIT (див. табл. 2.3.а);
- Redirect.from(File) (див. табл. 2.3.б);
- Redirect.to(File) (див. табл. 2.3.б);
- Redirect.appendTo(File) (див. табл. 2.3.б).

Клас ProcessBuilder.Redirect.Type

Слід зауважити, що метод type() класу ProcessBuilder.Redirect повертає значення типу ProcessBuilder.Redirect.Type (Java 8), який є вкладеним класом в ProcessBuilder.Redirect:

public static enum ProcessBuilder.Redirect.Type extends Enum<ProcessBuilder.Redirect.Type>.

Даний клас ProcessBuilder.Redirect.Type відповідає за тип перенаправлення ProcessBuilder.Redirect і містить наступне перелічення (табл. 2.4):

Таблиця 2.4. Константи переліченння класу ProcessBuilder.Redirect.Type.

Константа	Опис константи
APPEND	Тип переспрямування, отриманого з Redirect.appendTo(File).
INHERIT	Тип Redirect.INHERIT.
PIPE	Тип Redirect.PIPE.
READ	Тип переспрямування, отриманого з Redirect.from(File).
WRITE	Тип переспрямування, отриманого з Redirect.to(File).

Лістинг 2.3. Запуск двох підпроцесів.

Головний (поточний) процес ProcessBuilderDemo:

```
import java.io.File;
import java.util.*;

public class ProcessBuilderDemo {
     public static void main(String[] args) {
          try {
               System.out.println("Запуск підпроцесу № 1");
               String javaFileName = "SubProcess5";
               String classFilePath = "C:\\Users\\User\\
                    eclipse-workspace\\SubProcess5\\bin\\";
               // список з командами підпроцесу
               ArrayList<String> commands1 = new ArrayList<String>();
               commands1.add("java");
               commands1.add("-classpath");
               commands1.add(classFilePath);
               commands1.add(javaFileName);
               /* створення об'єкта ProcessBuilder з командами
               commands1 */
               ProcessBuilder pb1 = new ProcessBuilder(commands1);
               // отримання команд підпроцесу
               List<String> com = pb1.command();
               System.out.println("Команди підпроцесу № 1: " + com);
```

```
/* перенаправлення стандартного потоку вводу-виводу
(в тому числі і виводу помилок) підпроцесу у стандартний
ввід-вивід поточного процесу; нижче наведена інструкція
рівносильна наступним трьом: */
// pb1.redirectInput(ProcessBuilder.Redirect.INHERIT);
// pb1.redirectOutput(ProcessBuilder.Redirect.INHERIT);
// pb1.redirectError(ProcessBuilder.Redirect.INHERIT);
pb1.inheritIO();
System.out.print("Підпроцес № 1 каже: ");
Process pro1 = pb1.start(); // запуск підпроцесу
pro1.waitFor();

System.out.println("\nЗапуск підпроцесу № 2");
String jpgFileName = "forest.jpg";
String jpgFilePath = "D:\\Java-Test\\Process\\";
String[] list = {"mspaint", jpgFileName};
// створення об'єкта ProcessBuilder без команд
ProcessBuilder pb2 = new ProcessBuilder();
/* встановлення робочого каталогу підпроцесу (в даному
випадку, це шлях до файлу); у вище описаному підпроцесі
№ 1 це була б директорія classFilePath) */
pb2.directory(new File(jpgFilePath));
/* встановлення команд для підпроцесу;
альтернативним варіантом може слугувати перевантажена
версія даного методу, яка в якості параметру приймає
список (нижче подані 2 інструкції): */
// List<String> commands2 = List.of("mspaint",
//    jpgFileName);
// pb2.command(commands2);
pb2.command(list);
// отримання робочого каталогу підпроцесу
System.out.println("Робочий католог підпроцесу № 2: " +
      pb2.directory().getPath());
// отримання змінних середовища підпроцесу
```

```
                Map<String, String> env = pb2.environment();
                /* отримання змінної середовища, що відповідає за ОС, в
                якій запускатиметься підпроцес */
                System.out.println("Підпроцес № 2 здійснив запуск в ОС:
                      " + env.get("OS"));
                /* нижче подана команда дозволяє вивести порядково в
                консоль всі змінні середовища підпроцесу */
                // env.forEach((key, value) -> System.out.println(key +
                //    value));
                pb2.start();
          }
          catch (Exception e) {
                System.out.println("Помилка поточного процесу");
          }
     }
}

public class SubProcess5 {
     public static void main(String[] args) throws Exception {
          System.out.println("Hello from SubProcess5");
          /* навмисна генерація виключення, задля того, щоб побачити як
          виводиться інформація про дану помилку в батьківському процесі
          */
          throw new Exception();
     }
}
```

Консоль поточного Java-процесу ProcessBuilderDemo:

```
Запуск підпроцесу № 1
Команди підпроцесу № 1: [java, -classpath, C:\Users\User\eclipse-
workspace\SubProcess5\bin\, SubProcess5]
Підпроцес № 1 каже: Hello from SubProcess5
Exception in thread "main" java.lang.Exception
     at SubProcess5.main(SubProcess5.java:6)
```

```
Запуск підпроцесу № 2
Робочий католог підпроцесу № 2: D:\Java-Test\Process
Підпроцес № 2 здійснив запуск в ОС: Windows_NT
```

Лістинг 2.4. Запуск трьох підпроцесів з перенаправленням їх вводу/виводу з/в файл(у).

Головний (поточний) процес ProcessBuilderRedirectDemo:

```
import java.io.File;
// import java.lang.ProcessBuilder.Redirect;

public class ProcessBuilderRedirectDemo {

	static String workDir = "D:\\Java-Test\\Process\\";

	/* робота із системною командою ping - утилітою cmd,
	яка виводить відповідну службову інформацію при спробі підключення
	до web-адресата */
	static void runSystemSubProcess() throws Exception {
		System.out.println("Робота з підпроцесом № 1");
		ProcessBuilder pb = new ProcessBuilder(new String[] {"ping",
			"google.com"});
		/* файл, в якому зберігатиметься результат роботи (виведення
		даних) підпроцесу */
		String pingOutputFileName = "pingOutput.txt";
		File pingOutput = new File(workDir + pingOutputFileName);
		/* перенаправлення адресата стандартного потоку виведення
		даних підпроцесу у файл */
		pb.redirectOutput(pingOutput);
		pb.start();
	}
```

```
/* робота з користувацьким Java-підпроцесом для виведення даних у
файли */
static void runUserSubProcess6_1() throws Exception {
	System.out.println("\nРобота з підпроцесом № 2");
	String javaFileName = "SubProcess6_1";
	String classFilePath = "C:\\Users\\User\\eclipse-workspace\\
		SubProcess6_1\\bin\\";
	ProcessBuilder pb = new ProcessBuilder("java", "-classpath",
		classFilePath, javaFileName);
	/* файл, в якому зберігатиметься результат роботи (виведення
	даних) підпроцесу pb */
	String javaSubprocess6_1_OutputFileName =
		"javaSubProcess6_1_Output.log";
	File javaSubprocess6_1_Output = new File(workDir +
		javaSubprocess6_1_OutputFileName);
	/* перенаправлення адресата стандартного потоку виведення
	даних підпроцесу у файл;
	P.S.: дана інструкція дає аналогічний результат роботи, що і
	наступна: */
	// pb.redirectOutput(ProcessBuilder.Redirect.to
	//    (javaSubprocess6_1_Output));
	pb.redirectOutput(javaSubprocess6_1_Output);
	/* отримання і виведення в консоль поточного процесу адресата
	виведення даних підпроцесу */
	System.out.println(pb.redirectOutput());
	/* файл, в якому зберігатиметься результат роботи (виведення
	помилок) підпроцесу pb */
	String javaSubprocess6_1_ErrorFileName =
		"javaSubProcess6_1_Error.log";
	File javaSubprocess6_1_Error = new File(workDir +
		javaSubprocess6_1_ErrorFileName);
	/* перенаправлення адресата стандартного потоку виведення
	помилок підпроцесу у файл;
	P.S.: дана інструкція дає аналогічний результат роботи, що і
```

```
        наступна: */
        // pb.redirectError(ProcessBuilder.Redirect.appendTo
        //    (javaSubprocess6_1_Error));
        pb.redirectError(javaSubprocess6_1_Error);
        /* отримання і виведення в консоль поточного процесу адресата
        потоку виведення помилок підпроцесу */
        System.out.println(pb.redirectError());
        /* перевірка на співпадіння адресатів потоків виведення даних
        і помилок підпроцесу */
        if (!pb.redirectErrorStream())
              System.out.println("Адресати для виведення даних і
                    помилок підпроцесу SubProcess6_1 не співпадають");
        else
              System.out.println("Адресати для виведення даних і
                    помилок підпроцесу SubProcess6_1 співпадають");
        pb.start();
  }

  /* робота з користувацьким Java-підпроцесом для введення даних з
  файлу */
  static void runUserSubProcess6_2() throws Exception {
        System.out.println("\nРобота з підпроцесом № 3");
        String javaFileName2 = "SubProcess6_2";
        String classFilePath2 = "C:\\Users\\User\\eclipse-workspace\\
              SubProcess6_2\\bin\\";
        ProcessBuilder pb = new ProcessBuilder("java", "-classpath",
              classFilePath2, javaFileName2);
        /* файл, з якого відбуватиметься зчитування даних підпроцесом
        pb */
        String javaSubprocess6_2_InputFileName =
              "javaSubProcess6_2_Input.txt";
        File javaSubprocess6_2_Input = new File(workDir +
              javaSubprocess6_2_InputFileName);
        /* перенаправлення джерела стандартного потоку введення
```

```
            підпроцесу у файл;
            P.S.: дана інструкція дає аналогічний результат роботи, що і
            наступна: */
            // pb.redirectInput(ProcessBuilder.Redirect.from
            //    (javaSubprocess6_2_Input));
            pb.redirectInput(javaSubprocess6_2_Input);
            /* отримання і виведення в консоль поточного процесу адресата
            введення даних підпроцесу */
            System.out.println(pb.redirectInput());
            /* перенаправлення адресата стандартного потоку виведення
            даних підпроцесу в стандартний вивід поточного процесу */
            pb.redirectOutput(ProcessBuilder.Redirect.INHERIT);
            pb.start();
      }

      public static void main(String[] args) {
            try {
                  runSystemSubProcess();
                  runUserSubProcess6_1();
                  runUserSubProcess6_2();
            }
            catch (Exception e) {
                  System.out.println(e);
            }
      }
}
```

Консоль поточного Java-процесу ProcessBuilderRedirectDemo:

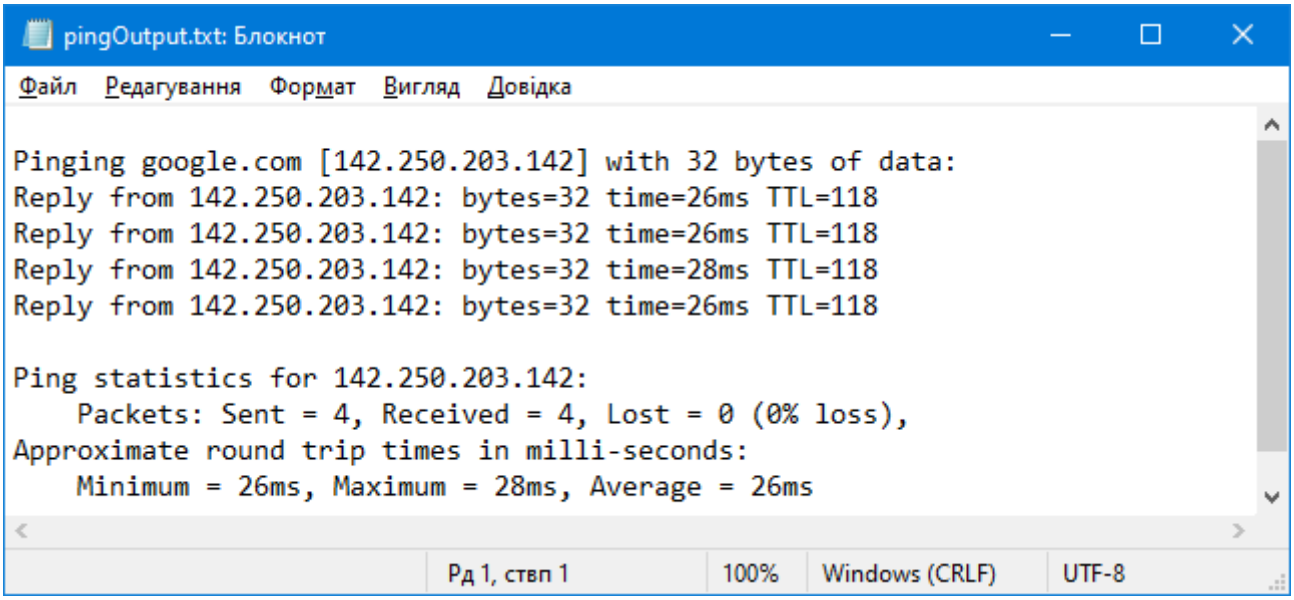
```
Робота з підпроцесом № 1

Робота з підпроцесом № 2
redirect to write to file "D:\Java-
Test\Process\javaSubProcess6_1_Output.log"
```

```
redirect to write to file "D:\Java-Test\Process\javaSubProcess6_1_Error.log"
Адресати для виведення даних і помилок підпроцесу SubProcess6_1 не співпадають

Робота з підпроцесом № 3
redirect to read from file "D:\Java-Test\Process\javaSubProcess6_2_Input.txt"
Line1
Line2
Line3
```

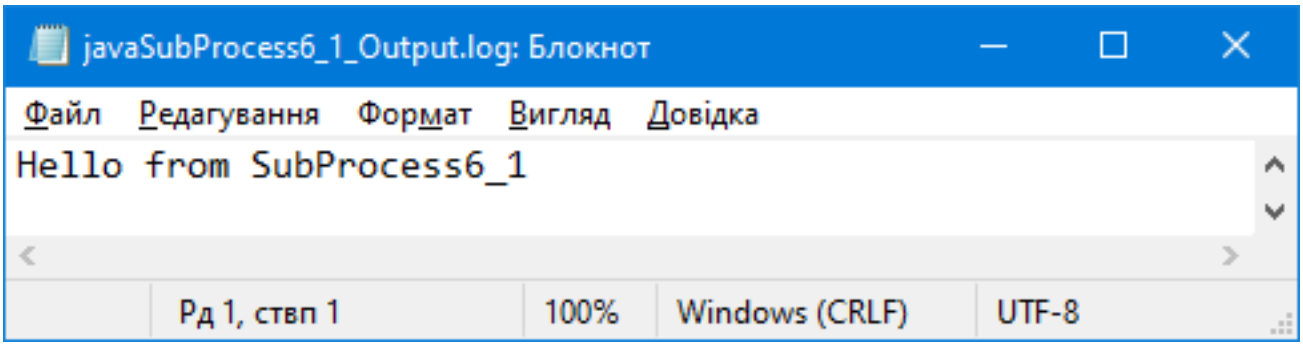
Згенерований файл системного підпроцесу ping:

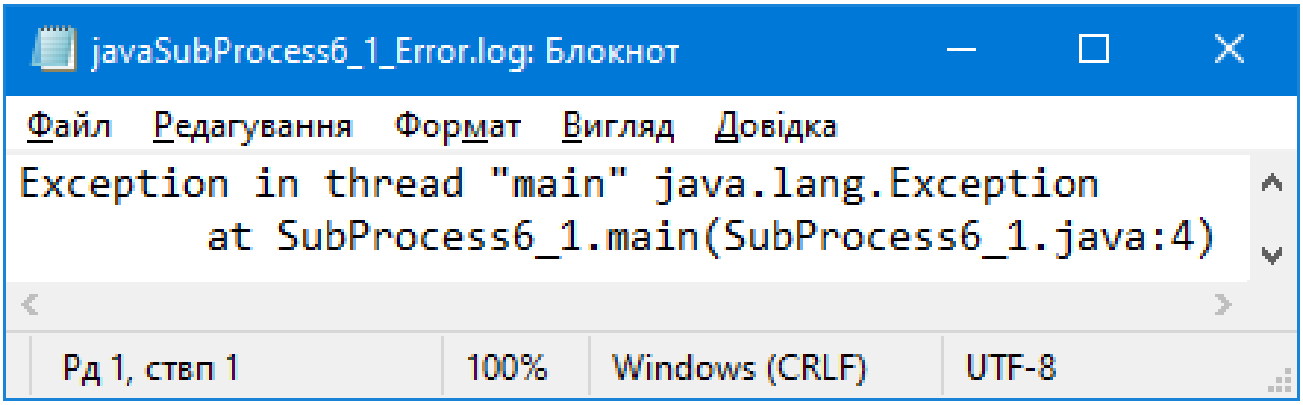


Java-підпроцес SubProcess6_1:

```
public class SubProcess6_1 {
	public static void main(String[] args) throws Exception {
		System.out.println("Hello from SubProcess6_1");
		throw new Exception();
	}
}
```

Згенеровані файли Java-підпроцесу SubProcess6_1:



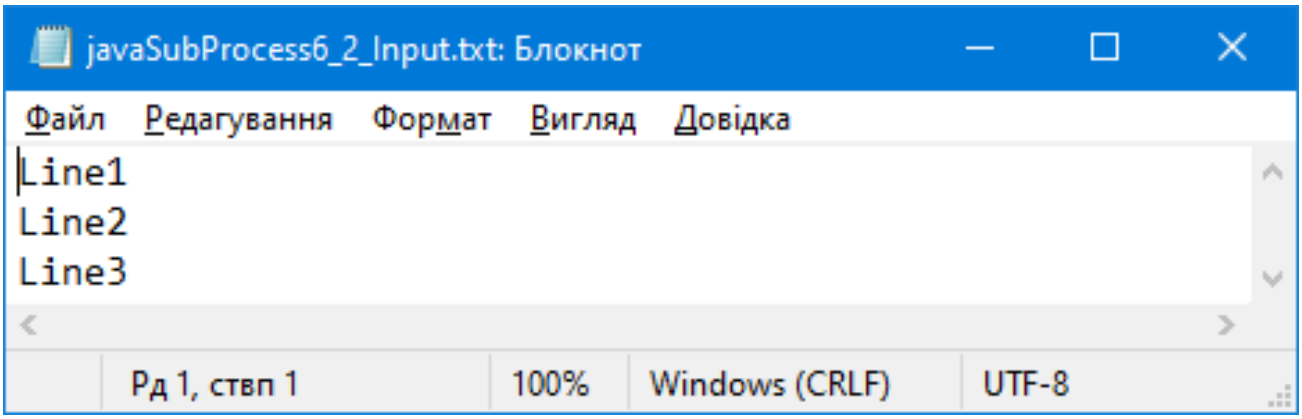


Java-підпроцес SubProcess6_2:

```
import java.util.Scanner;

public class SubProcess6_2 {
	public static void main(String[] args) {
		Scanner readConsole = new Scanner(System.in);
		while (readConsole.hasNextLine())
			System.out.println(readConsole.nextLine());
		readConsole.close();
	}
}
```

Файл, що зчитується Java-підпроцесом SubProcess6_2:



Лістинг 2.5. Запуск конвеєра процесів.

Головний (поточний) процес ProcessBuilderStartPipelineDemo:

```
// import java.util.Arrays;
import java.util.List;

public class ProcessBuilderStartPipelineDemo {

	// шлях до нескомпільованого файлу SubProcess9_1.java
	static String javaFilePath = "C:\\Users\\User\\eclipse-
		workspace\\SubProcess9_1\\src\\";
	// ім'я файлу без розширення
	static String javaFileName = "SubProcess9_1";
	/* шлях до директорії, де буде міститися скомпільований файл
	SubProcess9_1.class */
	static String classFilePath = "C:\\Users\\User\\eclipse-
		workspace\\SubProcess9_1\\bin\\";

	// метод для компіляції додатку (*.java --> *.class)
	static void compileSubProcess(String javaFilePath, String
		javaFileName, String classFilePath) {
		Runtime r = Runtime.getRuntime();
		try {
			r.exec(new String[] {"javac", "-d", classFilePath,
				javaFilePath + javaFileName + ".java"});
```

```
                Thread.sleep(1000);
            }
            catch (Exception e) {
                System.err.println("Compile Subprocess Error");
            }
        }

        public static void main(String[] args) throws Exception {
            /* компіляція додатку, що представляє підпроцес (в разі
            необхідності) */
            compileSubProcess(javaFilePath, javaFileName, classFilePath);
            /* ProcessBuilder для зміни директорії і виведення інформації
            про всі її об'єкти */
            ProcessBuilder pb1 = new ProcessBuilder("cmd", "/c", "cd",
                "/d", "D:\\Java-Test\\Process", "&", "dir");
            /* ProcessBuilder для пошуку об'єктів, імена яких містять
            підрядок "java" */
            ProcessBuilder pb2 = new ProcessBuilder("find", "\"java\"");
            /* ProcessBuilder користувацького підпроцесу із
            перенаправленням його виводу в поточний процес */
            ProcessBuilder pb3 = new ProcessBuilder(new String[] {"java",
                "-classpath", classFilePath, javaFileName})
                .redirectOutput(ProcessBuilder.Redirect.INHERIT);
            /* список об'єктів ProcessBuilder;
            P.S.: альтернативою для даної інструкції слугують наступні:
            1) ProcessBuilder[] pb = {pb1, pb2, pb3};
               List<ProcessBuilder> builders = List.of(pb);
            2) List<ProcessBuilder> builders = Arrays.asList(pb1, pb2,
                pb3); */
            List<ProcessBuilder> builders = List.of(pb1, pb2, pb3);
            // конвеєр підпроцесів
            List<Process> processes = ProcessBuilder.startPipeline
                (builders);
            for (Process process : processes)
```

```
                process.waitFor();
        }
}
```

Java-підпроцес SubProcess9_1:

```
import java.util.Scanner;

public class SubProcess9_1 {
        public static void main(String[] args) {
                final var sc = new Scanner(System.in);
                String line;
                while (sc.hasNext()) {
                        line = sc.nextLine();
                        /* виокремлення імен об'єктів (файлів і папок)
                        директорії, з подальшим їх виведенням;
                        така процедура парсингу зчитуваних даних являється
                        необхідною, оскільки команда dir в cmd передбачає
                        виведення структури папки з деякими атрибутами її
                        об'єктів (напр. час і дата створення), які відділяються
                        як мінімум 1-им пробілом */
                        System.out.println(line.substring(line.lastIndexOf(" ")
                                + 1));
                }
                sc.close();
        }
}
```

Консоль поточного Java-процесу ProcessBuilderStartPipelineDemo:

```
javaSubProcess6_1_Error.log
javaSubProcess6_1_Output.log
javaSubProcess6_2_Input.txt
```

Запитання:

1. Дайте означення міжпроцесній взаємодії. Які Ви знаєте види зв'язків взаємодіючих процесів з точки зору топології?
2. Назвіть основні проблемні ситуації, які можуть виникнути між взаємодіючими процесами? Що таке синхронізація процесів?
3. Дайте характеристику МПВ через спільний файловий ресурс.
4. Охарактеризуйте МПВ через відображення файлу в пам'ять та спільну пам’ять.
5. Дайте характеристику МПВ через канали.
6. Охарактеризуйте МПВ через сигнали.
7. Дайте характеристику МПВ через обмін повідомленнями.
8. Охарактеризуйте МПВ через віддалений виклик процедур.
9. Використання яких програмних структур дозволяє проводити МПВ? Які методи/функції призначені для МПВ?
10. Вкажіть умовний програмний алгоритм реалізації МПВ.

Вправи:

1. Поточний процес надсилає випадкове ціле число в підпроцес, який приймає дане число і у відповідь виводить таблицю множення на дане число.
2. В поточному процесі, користувач з консолі вводить символи, які надсилаються в підпроцес, який в свою чергу, перевіряє отримані символи на належність до того чи іншого алфавіту. Результат перевірки підпроцесу повинен бути надісланий назад в батьківський процес.
3. Реалізувати гру «Хрестики-нулики» між батьківським і дочірнім процесами.
4. Батьківський процес надсилає поточну дату для 3-ох підпроцесів. Перший підпроцес визначає пору року по отриманій даті; другий – кількість секунд, яка пройшла з початку року; третій – кількість днів, яка залишилася до Різдва Христового. Розрахована інформація кожним із підпроцесів надсилається назад в батьківський процес.

5. Реалізувати чат між поточним і дочірнім процесом. Спілкування повинно відбуватися доти, доки будь-який із процесів не напише в спільній консолі слово «exit».

6. Два підпроцеси виконують певне завдання (напр. розв’язують квадратне рівняння), яке надається їм від батьківського процесу (напр. надсилаються коефіцієнти квадратного рівняння). Як тільки один із них виконає першим таке завдання, робота іншого (ще не завершеного) підпроцесу, примусово завершується батьківським підпроцесом.

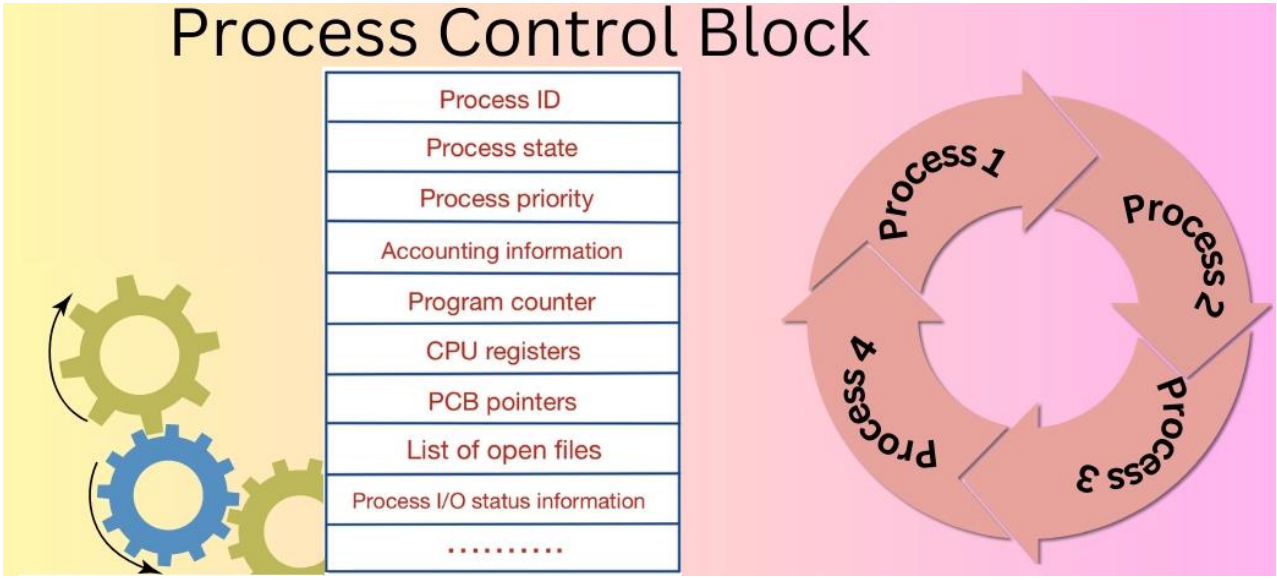
7. Поточний процес надсилає рядок до підпроцесу, який обраховує кількість входжень кожного символу, що знаходяться в отриманому рядку. Підпроцес повинен передати назад в батьківський процес той символ, який найчастіше зустрічається в рядку.

8. Батьківський процес надсилає цілочисельний масив для дочірнього підпроцесу, який в свою чергу визначає найбільший та найменший елементи, а також середнє арифметичне елементів отриманого масиву, повертаючи назад з батьківський процес, такі оброблені дані.

9. Два дочірніх підпроцеси періодично надсилають для батьківського процесу випадкові числа. Поточний процес сумує отримані числа для кожного окремого підпроцесу. Як тільки сума чисел для будь-якого підпроцесу буде більшою від 100, батьківський підпроцес надсилає для такого підпроцесу рядок з подякою. Після цього, два підпроцеси повинні завершити свою роботу.

10. Батьківський процес передає в дочірній підпроцес масив цілих чисел. Підпроцес перемножує отриманий масив на «свій, власний» та передає результуючий масив назад в батьківський процес.

РОЗДІЛ 3. ДЕСКРИПТОРИ ПРОЦЕСІВ



3.1. Теоретична частина

3.1.1. Загальні положення

Кожному процесові, з боку ОС, повинні бути виділені певні системні ресурси, серед який: ЦП, пам’ять та доступ до пристроїв введення/виведення. При цьому, будь-який процес працює у своєму віртуальному адресному просторі. Сукупність ділянок фізичної пам'яті, що відображаються на віртуальні адреси процесу, називається **образом процесу**. До нього входять безпосередньо як адресний простір процесу (коди команд, вхідні/проміжні та вихідні дані), так і інші елементи пам’яті, серед яких блок управління процесом (дані, необхідні ОС для управління процесом) та стеки (стек користувача; системний стек ядра: для зберігання параметрів і адрес виклику процедур і системних служб). В більшості сучасних ОС, віртуальний адресний простір процесу поділяється на дві неперервні частини: користувацьку (віртуальний простір активного процесу) і системну (віртуальний простір коду ОС, що викликається системними викликами), задля спрощення переходу між прикладним кодом та системним кодом ядра. Перерозподіл між такими частинами залежить від реалізації ОС. Також для більшості сучасних ОС з підтримкою віртуальної пам’яті характерно і те, що образ процесу складається з набору блоків (сегменти, сторінки або їх комбінація), не обов'язково

розташованих послідовно. Це забезпечує розміщення в основній пам'яті лише частини образу процесу (активна частина), в той час як у вторинній пам'яті, перебуватиме повний образ. Коли в основну пам'ять завантажується частина образу, вона туди не переноситься, а копіюється. Проте, якщо частина образу в основній пам'яті модифікується, вона повинна бути скопійована на диск.

Слід зауважити, що в залежності від модельного уявлення процесів, їх образ має різну архітектуру, що показано на наступному рисункові:

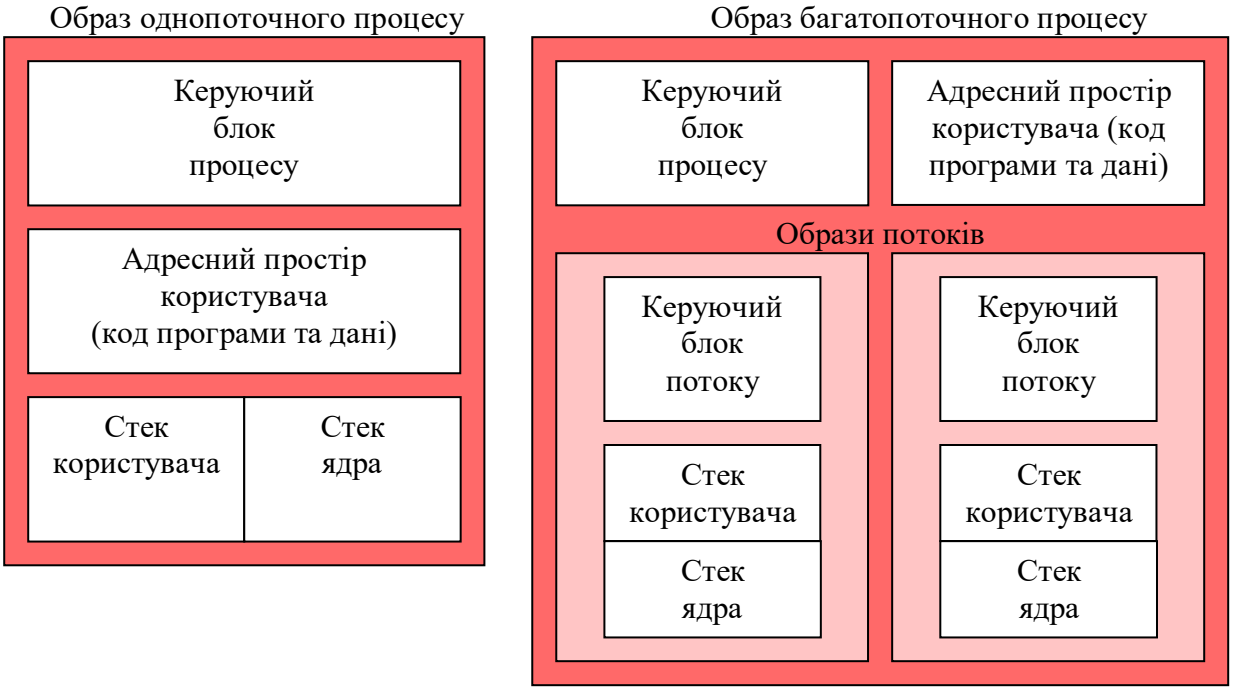


Рис. 3.1. Структура образу процесів.

До **адресного простору користувача** належать коди програми та дані, що вона використовує.

В свою чергу, **стеки користувача/ядра** представляють собою місце в пам’яті, де зберігаються дані про виконання коду програми в режимі користувача та ядра (напр. системні функції, що запускаються внаслідок системних викликів), відповідно. Коли процесор перемикається між такими режимами, він також перемикає стеки, задля того, щоб зберегти та відновити відповідні контексти виконання.

Зародження нового процесу передбачає створення **дескриптора процесу** (описувача процесу, блок управління процесом, керуючий блок процесу

(Process control block, PCB)) – набір інформаційних структур, які містять усі відомості (атрибути) про процес, необхідних для керування процесом з боку ОС. В залежності від реалізації ОС, до таких відомостей можуть наприклад входити:

- **ідентифікатор процесу** (Process IDentifier, PID) – унікальне число, що надається для ідентифікації активного процесу. Таке число може бути використане як параметр різних функцій, з метою маніпулюванням процесами (напр. надання пріоритету процесу, його знищення тощо);
- дані про розташування пам'яті виконуваного модуля;
- інформація про використання ресурсів (відкриті файли, мережеві з'єднання тощо);
- ступінь привілейованості процесу (пріоритет і права доступу);
- перелік взаємопов’язаних процесів;
- події, здійснення яких, чекає процес;
- перелік потоків, з яких складається процес (у випадку підтримки ОС багатопоточності);
- тощо.

В багатьох ОС, дескриптори окремих процесів об’єднуються в список, утворюючи **таблицю процесів**. При цьому, пам’ять для такої таблиці відводиться динамічно, в області ядра. Використовуючи таку таблицю, ОС здійснює планування і синхронізацію процесів. Загальний вигляд такої таблиці виглядає наступним чином (табл. 3.1):

Таблиця 3.1. Загальний вигляд таблиці процесів.

Управління процесом	Управління пам’яттю	Управління файлами
Регістри	Вказівник на інформацію про текстовий сегмент	Кореневий каталог
Лічильники команд	Вказівник на інформацію про сегмент даних	Робочий каталог
Слово стану програми	Вказівник на інформацію про сегмент стеку	Дескриптори файлів

Вказівник стеку		Ідентифікатор користувача
Стан процесу		Ідентифікатор групи
Пріоритет		
Параметри планування		
Ідентифікатор процесу		
Батьківський процес		
Група процесу		
Сигнали		
Час запуску процесу		
Використаний час ЦП		
Час ЦП, використовуваний дочірніми процесами		
Час наступного аварійного сигналу		

Прикладами описувачів процесів для різних ОС є наступні:

- об’єкт-процес (object-process) – Windows NT/2000/XP;
- дескриптор процесу – UNIX;
- керуючий блок процесу (PCB – Process Control Block) – OS/2.

По-суті, виконання черги процесів, які поступають на виконання до ЦП являють собою дескриптори процесів, об’єднаними в списки. Кожен із таких дескрипторів містять принаймні один вказівник на сусідній в черзі дескриптор (рис. 3.2). Така організація черги дозволяє легко переходити від виконання одного процесу до іншого, переводити їх з одного стану в інший.

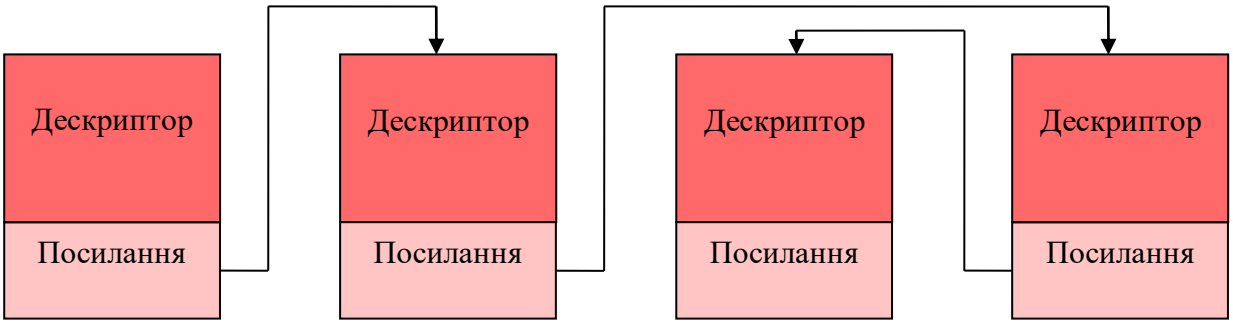


Рис. 3.2. Черга процесів.

Окрім дескриптора, деякі ОС оперують таким поняттям як **контекст процесу**. Так, процес в ході свого виконання, може неодноразово призупиняти своє виконання, з тих чи інших причин (напр. призупинка користувачем, перемикання з боку планувальника тощо). Власне і контекст процесу містить інформацію про процес, необхідну для відновлення ходу його виконання з перерваного місця. Зокрема:

- вміст регістрів ЦП;
- коди помилок системних викликів;
- інформація про відкриті файли, з боку даного процесу;
- інформація про незавершені операції вводу/виводу;
- та інші дані, що характеризують стан системи в момент переривання.

Для прикладу, наступний алгоритм показує роботу деякого процесу при виникненні того чи іншого переривання. В такому разі, процес призупиняє свою роботу, надаючи змогу виконатися обробнику переривання, після чого, процес відновить своє виконання з того контексту, в якому він перебував на момент призупинки:

- апаратною частиною поміщається лічильник команд активного процесу в стек;
- апаратною частиною завантажується новий лічильник команд з вектора переривання (область пам'яті, що містить посилання на процедуру обробника переривання);
- процедура на асемблері зберігає регістри (враховуючи те, що такі дії не можуть бути написані на мовах програмування високого рівня);
- процедура на асемблері встановлює покажчик на новий стек;
- запускається процедура мовою С (оскільки більшість ОС написана саме такою мовою), яка обслуговує переривання;
- планувальник здійснює прийняття рішення щодо процесу, який буде запущений наступним;
- процедура на мові С повертає управління асемблерного коду;
- асемблерна процедура запускає новий процес.

3.2. Практична частина

3.2.1. Атрибути процесів

Інтерфейс ProcessHandle

В Java 9 було представлено інтерфейс ProcessHandle (пакет java.lang) для додаткових можливостей по керуванню процесами:

public interface ProcessHandle extends Comparable<ProcessHandle>.

По-суті, ProcessHandle відповідає за роботу із дескрипторами процесів. Даний інтерфейс надає додаткові можливості для роботи з ідентифікаторами процесів, створенням списків дочірніх процесів, час виконання процесів, моніторингу по працездатності процесів, їх знищенню та інші елементи управління.

В нижче поданій таблиці представлені методи даного інтерфейсу:

Таблиця 3.2. Методи інтерфейсу ProcessHandle.

Метод	Опис дій метода
static Stream<ProcessHandle> allProcesses() (SecurityException, UnsupportedOperationException)	Повертає потік Stream<ProcessHandle> усіх процесів (як підпроцесів так і інших, запущених в ОС, процесів), видимих для поточного процесу. Враховуючи те, що процеси створюються та завершуються асинхронно, немає ніякої гарантії, що деякий процес у потоці є активним або що інші процеси не були створені з моменту створення потоку. *Примітка: Stream являється інструментом для спрощення і полегшення обробки набору даних.*
Stream<ProcessHandle> children() (SecurityException)	Повертає потік Stream<ProcessHandle> для підпроцесів, які є прямими нащадками поточного процесу. Для таких дочірніх процесів, метод parent() повертає поточний процес. Як правило, not alive процес, немає нащадків. Слід зауважити, що оскільки процеси

	створюються та завершуються асинхронно, немає гарантії, що деякий із процесів (або взагалі всі) є alive.
int compareTo(ProcessHandle other) (NullPointerException, ClassCastException)	Порівнює даний ProcessHandle із other. Повертає від'ємне, нуль або додатне число, якщо даний об'єкт менший, рівний або більший за вказаний об'єкт other, відповідно.
static ProcessHandle current() (SecurityException, UnsupportedOperationException)	Повертає ProcessHandle для поточного процесу. Для знищення поточного процесу не можна використовувати ProcessHandle, замість цього слід скористатися System.exit.
Stream<ProcessHandle> descendants() (SecurityException)	Повертає потік Stream<ProcessHandle> для всіх нащадків поточного процесу (включаючи прямих нащадків, нащадків нащадків і т.д.). Як правило, not alive процес, немає нащадків. Слід зауважити, що оскільки процеси створюються та завершуються асинхронно, немає гарантії, що деякий із процесів (або взагалі всі) є alive.
boolean destroy() (IllegalStateException)	Завершує процес, дозволяючи процесу повністю завершити його роботу. У разі, якщо процес не активний, жодних дій не виконується. Повертає true у разі успіху, інакше false. Після виклику destroy(), процес може не завершитися негайно і метод isAlive() може повернути істину протягом короткого проміжку часу. *Примітка: контроль з боку ОС може запобігти завершенню процесу.*
boolean destroyForcibly() (IllegalStateException)	Примусово завершує процес, негайно припиняючи хід його виконання. Всі вище описані позиції для destroy() також характерні і для даного методу.
boolean equals(Object other)	Порівнює даний ProcessHandle із other. Повертає true, якщо об’єкт other не є нульовим, має таку ж

	реалізацію та представляє той самий процес, інакше, повертає false.
int hashCode()	Повертає значення хеш-коду для даного об'єкта. Хеш-код є функцією значення pid(). Якщо два ProcessHandle рівні згідно equals(), виклик методу hashCode для кожного з двох об'єктів повинен давати однаковий результат.
ProcessHandle.Info info()	Повертає екземпляр ProcessHandle.Info, що надає доступ до методів, які повертають інформацію про процес, якщо вона доступна.
boolean isAlive()	Перевіряє, чи процес, представлений даним об'єктом ProcessHandle ще виконує свою роботу. Процес вважається активним, доки дійсний PID. Завершення процесу залежить від його реалізації та ОС.
static Optional<ProcessHandle> of (long pid) (SecurityException, UnsupportedOperationException)	Повертає Optional<ProcessHandle> для існуючого процесу з ідентифікатором pid. Якщо такий процес не існує, повертає пусте значення.
CompletableFuture<ProcessHandle> onExit() (IllegalStateException)	Повертає CompletableFuture<ProcessHandle> для завершення процесу. Надає CompletableFuture можливість запускати залежні функції або дії, які можуть виконуватися синхронно або асинхронно після завершення процесу. Коли процес завершено, CompletableFuture completed не залежить від статусу завершення процесу. Метод onExit можна викликати декілька разів, щоб викликати незалежні дії, коли процес завершується. Виклик onExit().get() очікує завершення процесу та повертає ProcessHandle.
Optional<ProcessHandle> parent() (SecurityException)	Повертає Optional<ProcessHandle> для батьківського процесу. Якщо дочірній процес не має батьківського або якщо батьківський процес недоступний (напр. через обмеження ОС), метод повертає порожнє значення.

long pid() (UnsupportedOperationException)	Повертає ідентифікатор процесу, який йому призначає ОС. ОС може повторно використовувати ідентифікатор процесу після його завершення, а тому, для порівняння ProcessHandles слід використовувати методи equals/compareTo.
boolean supportsNormalTermination()	Повертає true, якщо destroy() нормально завершує процес. Якщо ж destroy() примусово завершує процес, повертає false.

З таблиці видно, що екземпляри ProcessHandle повертаються фабричними методами current(), of(long), children() і descendants(), parent(), allProcesses().

Слід також зауважити, що в Java 9 було також доповнено методи класу Process (children, descendants, info, onExit, pid, supportsNormalTermination, toHandle), які виконують аналогічні дії, що і відповідні методи ProcessHandle.

Інтерфейс ProcessHandle.Info

Слід зауважити, що ProcessHandle містить вкладений інтерфейс ProcessHandle.Info:

static interface ProcessHandle.Info,

який надає додаткову інформацію стосовно процесу:

Методи такого інтерфейсу представлені в нижче поданій таблиці:

Таблиця 3.3. Методи інтерфейсу ProcessHandle.Info.

Метод	Опис дій метода
Optional<String[]> arguments()	Повертає масив рядків аргументів процесу. *Примітка: на деяких платформах (особливо Windows), процеси можуть вільно змінювати масив аргументів після запуску.*
Optional<String> command()	Повертає шлях до виконуваного файлу процесу.
Optional<String> commandLine()	Повертає командний рядок процесу. Якщо command() і arguments() повертають непорожні значення, це просто зручний метод,

	який об'єднує попередні два, і розділяє їх пробілами. В інакшому випадку, метод поверне найкраще, платформозалежне представлення командного рядка. *Примітки:* • *на деяких платформах (особливо Windows), через системні обмеження, повернутий шлях до виконуваного файлу та аргументи можуть бути скорочені;* • *шлях до виконуваного файлу може містити лише ім'я виконуваного файлу без інформації про повний шлях. При цьому, неможливо визначити, чи пробіл розділяє різні аргументи чи є частиною одного аргументу.*
Optional<Instant> startInstant()	Повертає час початку процесу.
Optional<Duration> totalCpuDuration()	Повертає загальний час, накопичений процесом.
Optional<String> user()	Повертає користувача процесу.

Як видно із попередніх двох таблиць, ряд методів повертає значення типу Optional<T>, який являється узагальненим класом-обгорткою над вказаним параметричним типом T. Такий клас є допоміжним і дозволяє працювати з об'єктами, навіть якщо вони повертають нульове значення (null), що значно полегшує роботу. Клас містить ряд корисних методів, серед яких можна виділити наступні (табл. 3.4):

Таблиця 3.4. Методи класу Optional<T>.

Метод	Опис дій метода
Optional<T> filter (Predicate<? super T> predicate) (NullPointerException)	Якщо значення присутнє, і воно відповідає даному предикату (напр. певній умові), повертає екземпляр Optional з об'єктом того ж типу T. В противному випадку, повертає порожній Optional.
T get()	Повертає значення типу T, якщо воно присутнє

(NoSuchElementException)	(обгорнуте) в даному екземплярі Optional. В противному разі, генерує NoSuchElementException.
void ifPresent (Consumer<? super T> consumer) (NullPointerException)	Якщо значення присутнє, викликає consumer (виконує певний код над об'єктом T, що обгорнутий даним екземпляром Optional).
boolean ifPresent()	Якщо значення присутнє, повертає true, інакше false.
<U> Optional<U> map(Function<? super T,? extends U> mapper) (NullPointerException)	Отримує існуюче значення T, застосовує для нього функцію відображення mapper (виконує певний код над об'єктом T, який обгорнутий даним екземпляром Optional) та повертає її результат типу U, обгорнутий в об'єкт Optional. Варто зауважити, що вихідний тип U не обов'язково повинен бути відмінним з вхідним T.
static <T> Optional<T> of(T value)	Створює екземпляр Optional, як обгортку навколо об'єкта other, у разі, якщо other не null. В противному випадку, генерує виключення NullPointerException.
T orElse(T other)	Повертає значення типу T, якщо воно присутнє, інакше повертає other.
T orElseGet (Supplier<? extends T> other) (NullPointerException)	Повертає значення типу T, у випадку, якщо воно присутнє, інакше викликає other та повертає результат даного виклику.

Лістинг 3.1. Отримання інформації щодо поточного процесу та його підпроцесів.

Головний (поточний) процес ProcessHandleDemo1:

```
import java.io.*;
import java.time.*;
import java.util.*;
// import java.util.stream.*;
```

```
public class ProcessHandleDemo1 {

	// метод для отримання інформації про процес
	static void processInfo(ProcessHandle ph) {
		String unk = "unknown";
		// отримання імені класу процесу
		System.out.println("ClassName: " + ph.getClass());
		// отримання ідентифікатора процесу
		System.out.println("PID: " + ph.pid());
		// отримання хеш-коду процесу
		System.out.println("HashCode: " + ph.hashCode());
		// отримання батьківського процесу
		Optional<ProcessHandle> parentOPH = ph.parent();
		/* отримання ідентифікатора батьківського процесу;
		P.S.: альтернативою до нижче описаного способу може
		слугувати використання перевантаженого методу ifPresent():
		*/
		// if (parentOPH.isPresent())
		//    System.out.println("Parent PID: " +
		//          parentOPH.get().pid());
		/* якщо ж просто скористатися інструкцією
		System.out.println("Parent: " + parentOPH),
		то в консоль буде виведено об'єкт Optional<ProcessHandle>,
		що в ряді випадків є незручно */
		parentOPH.ifPresent(parentPH ->
			System.out.println("Parent PID: " + parentPH.pid()));
		// об'єкт ProcessHandle.Info
		ProcessHandle.Info phi = ph.info();
		// отримання користувача процесу
		System.out.println("User: " + phi.user().orElse(unk));
		// шлях до процесу
		phi.command().ifPresent(pth ->
			System.out.println("Command: " + pth));
		/* отримання аргументів процесу;
```

```
P.S.:
- варто зауважити, що на деяких платформах (особливо
Windows) даний метод arguments() (так як і commandLine())
може не призводити до бажаних результатів;
- альтернативою нижче описаному алгоритму може слугувати
наступний код: */
// System.out.printf("Arguments: %s%n",
//    phi.arguments().map(a -> Stream.of(a).collect
//    (Collectors.joining(" "))).orElse(unk));
// або ж:
// System.out.println("Arguments: " +
//    phi.arguments().map(Arrays::toString).orElse(unk));
String[] argz = phi.arguments().orElse(new String[] {unk});
System.out.print("Arguments: ");
for (String arg : argz)
      System.out.printf("%s%n", arg);
/* отримання часу запуску процесу;
клас Instant відповідає за миттєвий поточний час;
метод даного класу atZone поєднує миттєвий час із часовим
поясом, який задається параметром ZoneId.systemDefault() і
повертає екземпляр ZonedDateTime, метод toLocalDateTime()
якого, повертає екземпляр LocalDateTime, який відповідає за
дату/час без часового поясу;
P.S.: якщо ж не проводити такі маніпуляції з часовою зоною,
тоді отриманий об'єкт Optional<Instant> може бути не
синхронізований із поточною часовою зоною, а тому, дата і
час можуть відрізнятися від системних */
System.out.printf("Start time: %s%n", phi.startInstant()
      .map(i -> i.atZone(ZoneId.systemDefault())
      .toLocalDateTime().toString()).orElse(unk));
/* отримання тривалості роботи процесу;
клас Duration відповідає за проміжок часу;
P.S.: альтернативою для нижче описаного способу може служити
наступний код: */
```

```
        // System.out.printf("Total CPU duration: %s ms%n",
        //    phi.totalCpuDuration()
        //    .orElse(Duration.ofMillis(0)).toMillis());
        phi.totalCpuDuration().ifPresent(duration ->
              System.out.printf("Total CPU duration: %s ms%n",
                    duration.toMillis()));
        System.out.println();
}

public static void main(String[] args) throws IOException {
        // екземпляр ProcessHandle поточного процесу
        ProcessHandle cph = ProcessHandle.current();
        // виведення даних поточного процесу
        processInfo(cph);

        // робота з підпроцесом SubProcess7
        String classFilePath = "C:\\Users\\User\\eclipse-workspace\\
              SubProcess7\\bin\\";
        String javaFileName = "SubProcess7";
        Process pro1 = new ProcessBuilder("java", "-cp",
              classFilePath, javaFileName).start();
        BufferedReader bri = new BufferedReader
              (new InputStreamReader(pro1.getInputStream()));
        // отримання ідентифікатору підпроцесу
        String childPID = bri.readLine();
        bri.close();
        /* екземпляр Optional<ProcessHandle> по ідентифікатору
        підпроцесу;
        P.S.: альтернативою до нижче описаної інструкції являється:
        */
        // Optional<ProcessHandle> oph1 = ProcessHandle.of
        //    (pro1.pid());
        Optional<ProcessHandle> oph1 = ProcessHandle.of
              (Long.parseLong(childPID));
```

```
            // екземпляр ProcessHandle підпроцесу
            ProcessHandle ph1 = oph1.get();
            processInfo(ph1);

            Process pro2 = new ProcessBuilder("notepad").start();
            // отримання екземпляру ProcessHandle з Process
            ProcessHandle ph2 = pro2.toHandle();
            processInfo(ph2);

            // порівняння екземплярів ProcessHandle
            System.out.println("Equals № 1 (CurrentJavaPro VS
                  JavaSubPro): " + cph.equals(ph1));
            System.out.println("Equals № 2 (CurrentJavaPro VS
                  NotepadeSubPro): " + cph.equals(ph2));
            System.out.println("Equals № 3 (CurrentJavaPro VS
                  CurrentJavaPro): " + cph.equals(cph));
      }
}
```

Java-підпроцес SubProcess7:

```
public class SubProcess7 {
      public static void main(String[] args) throws Exception {
            ProcessHandle cph = ProcessHandle.current();
            System.out.println(cph.pid());
      }
}
```

Консоль поточного Java-додатку:

```
ClassName: class java.lang.ProcessHandleImpl
PID: 8156
HashCode: 8156
Parent PID: 12680
User: DESKTOP-HJJ0F60\User
```

```
Command: C:\Program Files\Java\jdk-17.0.2\bin\javaw.exe
Arguments: unknown
Start time: 2023-04-21T13:19:46.867
Total CPU duration: 78 ms

ClassName: class java.lang.ProcessHandleImpl
PID: 11900
HashCode: 11900
Parent PID: 8156
User: DESKTOP-HJJ0F60\User
Command: C:\Program Files\Java\jdk-17.0.2\bin\java.exe
Arguments: unknown
Start time: 2023-04-21T13:19:46.946
Total CPU duration: 62 ms

ClassName: class java.lang.ProcessHandleImpl
PID: 11656
HashCode: 11656
Parent PID: 8156
User: DESKTOP-HJJ0F60\User
Command: C:\Windows\System32\notepad.exe
Arguments: unknown
Start time: 2023-04-21T13:19:47.018
Total CPU duration: 0 ms

Equals № 1 (CurrentJavaPro VS JavaSubPro): false
Equals № 2 (CurrentJavaPro VS NotepadeSubPro): false
Equals № 3 (CurrentJavaPro VS CurrentJavaPro): true
```

Лістинг 3.2. Отримання даних щодо всіх запущених в ОС процесів.

```
import java.util.stream.Stream;

public class ProcessHandleAllProcessesDemo1 {
```

```
// кількість процесів
static int countProcesses;

/* інформація про всі активні процеси, видимі для поточного
процесу */
static void allLiveProcessesInfo() {
	countProcesses = 0;
	/* Stream-потік з видимих активних процесів в системі */
	Stream<ProcessHandle> liveProcesses =
		ProcessHandle.allProcesses();
	System.out.println("Всі запущені процеси ОС:");
	/* фільтрування Stream-потоку по заданому критерію */
	liveProcesses.filter(p -> p.isAlive() &&
		p.info().command().isPresent())
		// сортування потоку
		.sorted()
		/* виконання певної дії над кожним елементом Stream-
		потоку */
		.forEach(ph ->
		System.out.println((++countProcesses) + ")"+ " PID: "
			+ ph.pid() + '\t' + ph.info().command().get()));
}

/* інформація про всі запущені програми, видимі для поточного
процесу;
P.S.: у вище описаному методі, відслідковувалися всі запущені
процеси в системі; проте, багато із запущених процесів можуть
належати для однієї програми (напр. кожна вкладка Chrome є
окремим процесом); тому, в деяких випадках, для зручності, можна
отримати інформацію про всі неповторювані запущені процеси ОС,
що власне і в нижче описаному методі представлено */
static void allLiveProgramsInfo() {
	countProcesses = 0;
```

```
            Stream<ProcessHandle> liveProcesses =
                  ProcessHandle.allProcesses();
            System.out.println("Всі запущені програми ОС:");
            liveProcesses.filter(ph -> ph.isAlive() &&
                  ph.info().command().isPresent())
                  /* конвертація потоку Stream<ProcessHandle> в
                  Stream<String>, кожен елемент якого, містить повний
                  шлях до процесу */
                  .map(ph -> ph.info().command().get())
                  /* отримання потоку Stream<String>, кожен елемент
                  якого, містить ім'я процесу */
                  .map(s -> s.substring(s.lastIndexOf(System.getProperty
                        ("file.separator")) + 1))
                  // видалення дублікатів з потоку
                  .distinct()
                  .sorted()
                  .forEach(s ->
                        System.out.println((++countProcesses) + ") "+s));
      }

      public static void main(String[] args) {
            allLiveProcessesInfo();
            allLiveProgramsInfo();
      }
}
```

```
Всі запущені процеси ОС:
1) PID: 296     C:\Windows\System32\CompPkgSrv.exe
2) PID: 592     C:\Program Files (x86)\Google\Chrome\Application\
chrome.exe
3) PID: 1396    C:\Program Files (x86)\Google\Chrome\Application\
chrome.exe
4) PID: 1988    C:\Program Files\Java\jdk-17.0.2\bin\javaw.exe
5) PID: 2824    C:\Program Files (x86)\Google\Chrome\Application\
```

```
chrome.exe
6) PID: 2928    C:\Program Files (x86)\Google\Chrome\Application\
chrome.exe
7) PID: 3452    C:\Program Files (x86)\Google\Chrome\Application\
chrome.exe
8) PID: 3484    C:\Program Files (x86)\Google\Chrome\Application\
chrome.exe
9) PID: 3492    C:\Program Files\AVG\Antivirus\AVGUI.exe
10) PID: 3584   C:\Windows\System32\oobe\UserOOBEBroker.exe
...
Всі запущені програми ОС:
1) AIMP.exe
2) AVGUI.exe
3) CompPkgSrv.exe
4) Domino.exe
5) Lightshot.exe
6) PsiphonPortable.exe
7) RtkAudUService64.exe
8) RuntimeBroker.exe
9) SearchApp.exe
10) SecurityHealthSystray.exe
...
```

Лістинг 3.3. Визначення найраніше запущеного процесу та процесу, який використовує найбільше процесорного часу, серед всіх запущених в ОС процесів.

```
import java.lang.ProcessHandle;
import java.time.*;
import java.time.format.DateTimeFormatter;
import java.util.*;
import java.util.function.Consumer;

public class ProcessHandleAllProcessesDemo2 {
```

```
public static void main(String[] args) {
	// об'єкт для роботи з датою та часом
	DateTimeFormatter formatter = DateTimeFormatter
		.ofPattern("dd-MM-yyyy HH:mm:ss")
		.withZone(ZoneId.systemDefault());
	/* функціональний інтерфейс "Споживач", що представляє
	операцію, яка приймає один вхідний аргумент і не повертає
	результату; даний інтерфейс можна використовувати як ціль
	для лямбда-виразу або посилання на метод */
	Consumer<String> printUser = user ->
		System.out.println("User: " + user);
	Consumer<String> printCmd = cmd ->
		System.out.println("Command: " + cmd);
	Consumer<String> printCmdLine = cmdln ->
		System.out.println("Command line: " + cmdln);
	Consumer<String[]> printArgs = arguments ->
		System.out.println("Arguments: " +
			Arrays.toString(arguments));
	Consumer<Instant> printInstant = inst ->
		System.out.println("Start time: " +
			formatter.format(inst));
	Consumer<Duration> printCPU = duration ->
		System.out.println("CPU time: " + duration.toMillis()
			+ " ms");
	Consumer<ProcessHandle.Info> printInfo = info -> {
		info.user().ifPresent(printUser);
		info.command().ifPresent(printCmd);
		info.commandLine().ifPresent(printCmdLine);
		info.arguments().ifPresent(printArgs);
		info.startInstant().ifPresent(printInstant);
		info.totalCpuDuration().ifPresent(printCPU);
		System.out.println();
	};
	// поточний момент системного годинника
```

```
            Instant now = Instant.now();
            System.out.println("Earliest Process:");
            // визначення найраніше запущеного процесу
            Optional<ProcessHandle.Info> ophi1 =
                  ProcessHandle.allProcesses()
                  .map(p -> p.info())
                  .filter(info -> info.startInstant().isPresent())
                  /* отримання максимального елементу з Stream-потоку
                  відповідно до вказаного "Споживача", який порівнює
                  елементи по їх часу запуску */
                  .max((p, q) -> q.startInstant().orElse(now)
                        .compareTo(p.startInstant().orElse(now)));
            ophi1.ifPresent(printInfo);
            System.out.println("Longest Process:");
            // визначення найдовше запущеного процесу
            Optional<ProcessHandle.Info> ophi2 =
                  ProcessHandle.allProcesses()
                  .map(p -> p.info())
                  .filter(info -> info.totalCpuDuration().isPresent())
                  .max((p, q) -> p.totalCpuDuration()
                        .orElse(Duration.ZERO)
                        .compareTo(q.totalCpuDuration()
                        .orElse(Duration.ZERO)));
            ophi2.ifPresent(printInfo);
      }
}
```

```
Earliest Process:
User: DESKTOP-HJJ0F60\User
Command: C:\Windows\System32\sihost.exe
Start time: 21-04-2023 09:04:27
CPU time: 781 ms

Longest Process:
```

```
User: DESKTOP-HJJ0F60\User
Command: C:\Program Files\qBittorrent\qbittorrent.exe
Start time: 21-04-2023 09:04:45
CPU time: 860359 ms
```

Лістинг 3.4. Завершення деякого процесу, при умові, що він запущений в ОС і не являється при цьому, нащадком поточного процесу (оскільки в такому разі, знищення процесу являтиметься доволі таки тривіальною задачею).

```
public class ProcessHandleAllProcessesDemo3 {
	public static void main(String[] args) {
		/* закриття всіх процесів деякої програми (напр. notepad);
		фільтрація дескрипторів процесів, так, щоб залишилися лише
		ті, які містять в своїй рядковій структурі набір символів
		"notepad" (метод contains) з подальшим закриттям даних
		процесів;
		P.S.: як видно із нижче поданого коду, не обов'язково
		користуватися методом command(), щоб отримати шлях до
		процесу і вже потім перевіряти, чи не закінчується даний
		шлях певним набором символів (напр. метод
		endsWith("notepad.exe")) оскільки об'єкт дескриптора процесу
		ProcessHandle вже містить у своїй рядковій структурі
		всю необхідну інформацію, включаючи шлях, PID та інші
		параметри;
		якщо все ж таки використовувати даний альтернативний підхід,
		тоді код для нього набуває наступного вигляду: */
		// ProcessHandle.allProcesses().filter(ph ->
		//    ph.info().command().orElse("")
		//      .endsWith("notepad.exe")).
		//      forEach(ph -> ph.destroyForcibly());
		ProcessHandle.allProcesses().filter(ph ->
			ph.info().toString()
			.contains("notepad"))
			.forEach(ph -> ph.destroyForcibly());
```

```
//          /* закриття лише одного процесу деякої програми (напр.
//          notepad);
//          вище описані методи дозволяють закрити всі без виключення
//          процеси однієї програми;
//          якщо ж необхідно закрити лише один запущений процес певної
//          програми, можна скористатися нижче поданим програмним кодом
//          */
//          ProcessHandle.allProcesses().filter(ph ->
//                    ph.info().toString()
//                    .contains("notepad"))
//                    /* повертає будь-який елемент потоку;
//                    щоб отримати перший елемент потоку, можна скористатися
//                    методом findFirst() */
//                    .findAny()
//                    /* Java 9: якщо значення присутнє, виконує певну дію з
//                    даним значення, інакше, виконує вказану дію на основі
//                    порожніх даних */
//                    .ifPresentOrElse(ph -> {
//                              System.out.println("Процес закритий");
//                              ph.destroyForcibly();
//                    },
//                              () -> System.out.println("Процес не знайдений"));
     }
}
```

Лістинг 3.5. Наступний додаток містить два корисних методи:

- 1-ий: перевіряє чи запущений деякий процес;
- 2-ий: знищує деякий процес, який працює більше заданого проміжку часу. Така необхідність може наприклад виникнути, коли процес завис.

```
import java.util.stream.Stream;
import java.util.*;
import java.time.*;
```

```
public class ProcessHandleAllProcessesDemo4 {

	// перевірка чи запущений процес
	static boolean isProRunning(String proName) {
		boolean run = ProcessHandle.allProcesses()
			/* формування потоку може також відбуватися і з
			допомогою посилання на методи */
			.map(ProcessHandle::info)
			.map(ProcessHandle.Info::command)
			/* метод map формує новий Stream-потік по заданому
			предикату, застосовуючи певну операцію до кожного
			елементу (в даному випадку <Optional<String>>) даного
			потоку; в той же час, метод flatMap дозволяє
			вивільнити вкладені елементи в Stream-потоці,
			сформувавши із них новий потік (в даному випадку,
			отримується Stream-потік із String-елементів,
			напротивагу методу map, який формував Stream-потік із
			<Optional<String>>-елементів) */
			.flatMap(Optional::stream)
			/* перевірка на існування того чи іншого елемента в
			потоці */
			.anyMatch(s -> s.contains(proName));
			/* якщо не використовувати метод flatMap, тоді
			останній рядок коду виглядав би як
			.anyMatch(s -> s.orElse("").contains(proName)); */
		return run;
	}

	/* знищення процесів, які працюють більше ніж задану кількість
	годин */
	static void destroySuspendedProcess(String proName, int hours) {
		Stream<ProcessHandle> allProcess, delProcesses;
		allProcess = ProcessHandle.allProcesses();
		delProcesses = allProcess.filter(ph -> {
```

```
                if (ph.isAlive()) {
                    ProcessHandle.Info phi = ph.info();
                    String cmd = phi.command().orElse(null);
                    if (cmd != null && cmd.endsWith(proName))
                        /* перевірка на те, чи працюють процеси
                        більше 2-ох годин */
                        if (Duration.between(phi.startInstant()
                            .get(), Instant.now()).toHours() >
                            hours)
                            return true;
                }
                return false;
            });
            delProcesses.forEach(ph -> ph.destroy());
        }

        public static void main(String[] args) {
            System.out.println("Eclipse запущений?: " +
                isProRunning("eclipse"));
            destroySuspendedProcess("notepad.exe", 2);
        }
}
```

```
Eclipse запущений?: true
```

Лістинг 3.6. Отримання інформації щодо нащадків поточного процесу.

Головний (поточний) процес ProcessHandleAllChildrenDescendantsDemo:

```
import java.util.stream.*;
import java.util.*;

public class ProcessHandleAllChildrenDescendantsDemo {
```

```
// кількість процесів
static int countProcesses;

/* інформація (PID + шлях до виконуваного файлу) про всіх прямих
нащадків поточного процесу */
static void allLiveChildrenProcesses() throws Exception {
	countProcesses = 0;
	System.out.println("Прямі нащадки поточного процесу:");
	/* Stream-потік з прямих нащадків поточного процесу */
	Stream<ProcessHandle> childrenProcessStream =
		ProcessHandle.current().children();
	childrenProcessStream.filter(ph -> ph.isAlive())
		.forEach(ph ->
			System.out.println((++countProcesses) +
				") PID: " + ph.pid() + '\t' +
				ph.info().command().get()));
	/* альтернативний метод, щодо ідентифікації прямих нащадків
	поточного процесу */
	// List<ProcessHandle> childrenProcessList =
	//	ProcessHandle.current().children()
	//	.collect(Collectors.toList());
	// countProcesses = 0;
	// for(ProcessHandle ph : childrenProcessList)
	//	System.out.println((++countProcesses) + ") PID: " +
	//		ph.pid() + '\t' + ph.info().command().get());
}

/* інформація (шлях до виконуваного файлу) про всіх нащадків
поточного процесу */
static void allLiveDescendantsProcesses() throws Exception {
	System.out.println("Всі нащадки поточного процесу:");
	/* Stream-потік з усіх нащадків поточного процесу */
	Stream<ProcessHandle> descendantsProcessStream =
		ProcessHandle.current().descendants();
```

```
            // використання посилань на методи
            descendantsProcessStream.filter(ProcessHandle::isAlive)
                  .map(ProcessHandle::info)
                  .map(ProcessHandle.Info::command)
                  .map(Optional::get)
                  .forEach(System.out::println);
            /* такий спосіб (верхні 4 інструкції) дозволяє отримати лише
            шлях до виконуваного процесу; якщо ж необхідно отримати ще й
            PID, то можна скористатися схожим способом, як і у вище
            описаному методі allLiveChildrenProcesses) */
      }

      public static void main(String[] args) throws Exception {
            String javaFileName = "SubProcess8";
            String classFilePath = "C:\\Users\\User\\eclipse-workspace\\
                  SubProcess8\\bin\\";
            new ProcessBuilder("java", "-classpath", classFilePath,
                  javaFileName).start();
            new ProcessBuilder("notepad").start();
            new ProcessBuilder("msinfo32.exe").start();
            /* очікування задля того, щоб мали змогу завантажитися
            підпроцеси */
            Thread.sleep(2000);
            allLiveChildrenProcesses();
            allLiveDescendantsProcesses();
      }
}
```

Java-підпроцес SubProcess8:

```
public class SubProcess8 {
      public static void main(String[] args) throws Exception {
            String javaFileName1 = "SubProcess8_1";
            String classFilePath1 = "C:\\Users\\User\\eclipse-workspace
```

```
                \\SubProcess8_1\\bin\\";
            new ProcessBuilder("java", "-classpath", classFilePath1,
                javaFileName1).start();
            new ProcessBuilder("mspaint").start();
            Thread.sleep(5000);
        }
}
```

Java-підпроцес SubProcess8_1:

```
public class SubProcess8_1 {
    public static void main(String[] args) throws Exception {
            new ProcessBuilder("C:\\Program Files\\WinRAR\\WinRAR.exe")
                .start();
            Thread.sleep(5000);
        }
}
```

Консоль поточного Java-процесу:

```
Прямі нащадки поточного процесу:
1) PID: 11396   C:\Program Files\Java\jdk-17.0.2\bin\java.exe
2) PID: 12936   C:\Windows\System32\notepad.exe
3) PID: 5512    C:\Windows\System32\msinfo32.exe
Всі нащадки поточного процесу:
C:\Program Files\Java\jdk-17.0.2\bin\java.exe
C:\Windows\System32\notepad.exe
C:\Windows\System32\msinfo32.exe
C:\Windows\System32\conhost.exe
C:\Program Files\Java\jdk-17.0.2\bin\java.exe
C:\Windows\System32\mspaint.exe
C:\Windows\System32\conhost.exe
C:\Program Files\WinRAR\WinRAR.exe
```

3.2.2. Відкладені завдання. Асинхронне програмування

Клас CompletableFuture<T>

Як вже зазначалося, клас ProcessHandle містить метод onExit(), який повертає значення CompletableFuture<ProcessHandle>. Об'єкт такого типу дозволяє програмувати відкладені завдання (задачі) по завершенню деякої події з допомогою callback-методів, запускати синхронні/асинхронні задачі та інші можливості, в багатьох випадках без яких, не можна обійтися. При цьому, такі задачі можуть виконуватися як в головному потоці, так і в фоновому.

Взагалому, синтаксис даного класу (пакет java.util.concurrent.CompletableFuture<T>) є наступним:

public class CompletableFuture<T> extends Object implements Future<T>, CompletionStage<T>.

Перелік методів даного класу є досить таки великим. Тому, доцільно привести лише найбільш вживані із них (Java 8) (табл. 3.5):

Таблиця 3.5. Методи клсасу CompletableFuture<T>.

Метод	Опис дій метода
static CompletableFuture<Void> allOf(CompletableFuture<?>... cfs) (NullPointerException)	Об'єднує декілька вказаних задач cfs. Повертає новий CompletableFuture, який завершується, коли завершуються всі вказані задачі cfs.
static CompletableFuture<Object> anyOf(CompletableFuture<?>... cfs) (NullPointerException)	Об'єднує декілька вказаних задач cfs. Повертає новий CompletableFuture, який завершується тоді, коли будь-яка із вказаних задач cfs завершується.
boolean cancel(boolean mayInterruptIfRunning)	Намагається завершити задачу, якщо вона ще не завершена, генеруючи CancellationException. Залежні від даного завдання, які ще не завершені, також завершаться винятково. Повертає true, якщо завдання скасовано. Параметр mayInterruptIfRunning – прапор, який вказує, чи потрібно переривати потік, який виконує

	завдання.
boolean complete(T value)	Завершує задачу, якщо вона ще не завершена, встановлюючи для неї значення value, яке можна повернути з допомогою методу get(). Повертає true, якщо завдання завершено.
public T get() throws InterruptedException, ExecutionException (CancellationException)	Очікує виконання задачі, повертаючи результат її виконання.
public T get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionException, TimeoutException (CancellationException)	За потреби, очікує заданий час timeout (unit – одиниці часу) для завершення завдання, після чого, повертає результат, якщо він доступний.
public T getNow(T valueIfAbsent) (CancellationException, CompletionException)	Якщо завдання завершено, повертає результат його виконання (або генерує будь-який виявлений виняток), в іншому випадку, повертає задане значення valueIfAbsent.
public boolean isCancelled()	Перевіряє, чи CompletableFuture було скасовано до його нормального завершення. Повертає true, якщо таке завдання було скасовано.
public boolean isDone()	Перевіряє чи завдання завершено. Повертає true, якщо завдання завершено будь-яким способом (звичайним, винятковим або шляхом скасування).
public T join() (CancellationException, CompletionException)	Аналогічно з get(). Проте, якщо завдання завершилося винятково, генерує (неперевірений) виняток.
public static CompletableFuture<Void> runAsync(Runnable runnable)	Запускає асинхронну задачу runnable, повертаючи нове значення CompletableFuture.
static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)	Запускає асинхронну задачу supplier, повертаючи нове значення CompletableFuture, яке додатково дозволяє повернути деяке значення типу

	U з допомогою методу get().
public CompletableFuture<Void> thenAccept(Consumer<? super T> action)	Callback-метод, який запускає асинхронну задачу action одразу після нормального завершення задачі, яка представлена викликаючим об’єктом (з якого власне і викликається thenAccept). Метод повертає нове значення CompletableFuture. При цьому, action надає доступ до результату CompletableFuture, до об'єкта якого прикріплений метод thenAccept.
public <U> CompletableFuture<U> thenApply(Function<? super T,? extends U> fn)	Callback-метод, який запускає асинхронну задачу fn одразу після нормального завершення задачі, яка представлена викликаючим об’єктом (з якого власне і викликається thenApply). Метод повертає нове значення CompletableFuture. При цьому, fn надає доступ до результату CompletableFuture, до об'єкта якого прикріплений метод thenApply, а також повертає певний результат типу U, який можна отримати методом get() із об’єкта, який повертає даний метод thenApply.
public <U,V> CompletableFuture<V> thenCombine(CompletionStage<? extends U> other, BiFunction<? super T,? super U,? extends V> fn)	Callback-метод, який поєднує дві асинхронні незалежні одна від одної задачі: 1-ша є об’єктом, з якого викликається даний метод thenCombine; 2-га є параметром other. По-суті, метод запускає нову задачу fn, як тільки виконаються попередньо описані дві задачі. При цьому, завдання fn отримує доступ до результатів 2-ох поєднаних задач. Метод повертає нове значення CompletableFuture.
public <U> CompletableFuture <U> thenCompose(Function <? super T ,? extends CompletionStage <U>> fn)	Callback-метод для поєднання двох асинхронних залежних одна від одної задач: 1-ша є об’єктом, з якого викликається даний метод thenCombine; 2-га

	є параметром fn. По-суті, метод запускає fn, як тільки завершить роботу задача, за яку відповідає об'єкт, який прикріплений до методу thenCompose. При цьому, fn отримує доступ до результату 1-ої задачі. Метод повертає нове значення CompletableFuture.
public CompletableFuture<Void> thenRun(Runnable action)	Callback-метод, який запускає асинхронне завдання action одразу після нормального завершення задачі, яка представлена викликаючим об'єктом (з якого власне і викликається thenRun). Метод повертає нове значення CompletableFuture. При цьому, action (на відміну від thenAccept і thenApply) не надає доступу до попередніх обчислень і не повертає жодного результату.

Слід зауважити, що всі задачі, що запускаються рядом даних методів можуть бути запущені як в головному потоці так і в фоновому. Якщо такі задачі виконуються доволі таки швидко, тоді вони можуть бути запущеними в головному потоці. Якщо ж їм потрібно відносно багато процесорного часу, тоді з великою долею ймовірності, вони запускатимуться у фоновому, по відношенню до головного потоку, режимі. Проте дане правило не є остаточним і може змінюватися виконавчим середовищем. Зважаючи на це, якщо потрібно, щоб завдання були дійсно запущеними в фоновому потоці, необхідно скористатися рядом додаткових методів. Так наприклад, методи thenAccept/thenApply/thenCombine/thenCompose/thenRun не дають 100 % гарантії де запускатимуться задачі, в той час, як їх суміжні методи thenAcceptAsync/thenApplyAsync/thenCombineAsync/thenComposeAsync/thenRun Async дозволяють запускати такі завдання в фоновому режимі (див. докладніше Javadoc).

Лістинг 3.7. Виконання певних завдань одразу після завершення виконання того чи іншого підпроцесу. Виконання таких дій можливо з допомогою блокуючого методу get(), а також callback-функцій thenRun, thenAccept і thenApply.

Примітка: callback-функції (ф-ції оберненого виклику) в програмуванні – функції, які викликаються всякий раз, коли відбудеться та чи інша подія. По-суті, це виклик певної події при настанні іншої події (напр. подія натискання на кнопку генерує іншу подію, завершення процесу передбачає виконання тих чи інших дій). Досить часто, такі функції працюють асинхронно-одночасно із основною програмою.

```
import java.io.IOException;
// робота з класом CompletableFuture
import java.util.concurrent.CompletableFuture;
// робота з деякими винятками методів класу CompletableFuture
import java.util.concurrent.ExecutionException;

public class ProcessHandleOnExitDemo1 {
     public static void main(String[] args) throws IOException,
          InterruptedException, ExecutionException {
          /* виконання певних дій, одразу ж після завершення підпроцесу,
          з очікування його завершення роботи */
          Process pro1 = new ProcessBuilder("notepad.exe").start();
          ProcessHandle ph1 = pro1.toHandle();
          /* отримання екземпляру CompletableFuture<ProcessHandle> для
          можливості подальших асинхронних дій */
          CompletableFuture<ProcessHandle> cf1 = ph1.onExit();
          System.out.println("Waiting for completion Notepad...");
          /* очікування завершення підпроцесу, який пов'язаний з
          об'єктом cf1; метод get() є блокуючим, а тому подальші дії
          головного процесу є неможливими; в даному випадку, таке
          блокування є синхронною дією, проте, об'єкт cf1 можна
```

```
застосовувати і для асинхронних операцій, що буде показано
дещо нижче;
P.S.: посилання PH1 не обов'язково отримувати */
ProcessHandle PH1 = cf1.get();
System.out.println("Notepad is Done?: " + cf1.isDone()
      + " ph1 = PH1?: " + ph1.equals(PH1));

/* виконання певних дій, одразу ж після завершення підпроцесу,
без очікування його завершення роботи */
Process pro2 = new ProcessBuilder("ping", "google.com.ua")
      .start();
ProcessHandle ph2 = pro2.toHandle();
CompletableFuture<ProcessHandle> cf2 = ph2.onExit();
/* виконання вказаних інструкцій одразу ж після завершення
виконання підпроцесу, пов'язаного з об'єктом cf2; на відміну
від методу get(), даний метод thenRun не є блокуючим, а тому,
весь нижче описаний код головного процесу буде виконуватися,
проте як тільки завершиться підпроцес пов'язаний з cf2, в
новому потоці почне виконуватися нижче подана інструкція,
паралельно (разом з тим і асинхронно) з виконанням коду
головного потоку - в такому випадку говорять про функцію
callback;  така асинхронна задача завершує своє виконання, при
одній із наступних умов:
- завершаться всі її інструкції, при умові, що головний потік
ще працюватиме;
- головний потік завершить свою роботу. В такому випадку,
немає гарантії, що всі інструкції CF2 повністю будуть
виконані, оскільки головний потік може першим завершити свою
роботу. Для того, щоб така задача CF2 повністю виконалася,
необхідно скористатися вище описаним методом get(), який
нічого не поверне (void), проте буде блокувати роботу
головного потоку, допоки весь код cf2.thenRun не виконається;
```

```
P.S.: посилання CF2 не обов'язково отримувати, проте його
отримання може дозволити здійснити інші асинхронні операції
(напр. створити ланцюг асинхронних подій (CF2.thenRun)) */
CompletableFuture<Void> CF2 = cf2.thenRun(() -> {
      System.out.println("\t" + "First Thread: After
            thenRun... Ping is Done");
      for (int i = 1; i <= 10; i++)
            try {
                  System.out.println("\t" + "1-" + i + ")
                        First Thread: After thenRun...");
                  Thread.sleep(2000);
            }
            catch (InterruptedException e) {
                  e.printStackTrace();
            }
});
/* нижче описаний цикл є допоміжним і показує асинхронну
роботу методу thenRun паралельно з головним потоком */
for (int i = 1; i <= 5; i++)
      try {
            System.out.println("1-" + i + ") Main
                  Thread...");
            Thread.sleep(1000);
      }
      catch (InterruptedException e) {
            e.printStackTrace();
      }

/* виконання певних дій, одразу ж після завершення
підпроцесу, без очікування його завершення роботи (ф-ція
callback); на відміну від thenRun, метод thenAccept надає
змогу доступу до результату CompletableFuture, до об'єкта
якого він прикріплений */
Process pro3 = new ProcessBuilder("ping", "ibm.com")
```

```
        .start();
ProcessHandle ph3 = pro3.toHandle();
CompletableFuture<ProcessHandle> cf3 = ph3.onExit();
CompletableFuture<Void> CF3 = cf3.thenAccept(PH3 -> {
        System.out.println("\t" + "Second Thread: After
                thenAccept... Ping is Done. ph3 = PH3?: " +
                ph3.equals(PH3));
        for (int i = 1; i <= 10; i++)
                try {
                        System.out.println("\t" + "2-" + i + ")
                                Second Thread: After thenAccept...");
                        Thread.sleep(2000);
                }
                catch (InterruptedException e) {
                        e.printStackTrace();
                }
});
// теж допоміжний цикл
for (int i = 1; i <= 5; i++)
try {
        System.out.println("2-" + i + ") Main Thread...");
        Thread.sleep(1000);
}
catch (InterruptedException e) {
        e.printStackTrace();
}

/* виконання певних дій, одразу ж після завершення
підпроцесу, без очікування його завершення роботи (ф-ція
callback); на відміну від thenAccept, метод thenApply надає
змогу доступу до результату CompletableFuture, до об'єкта
якого він прикріплений, а також повертає певний результат */
Process pro4 = new ProcessBuilder("ping", "microsoft.com")
        .start();
```

```
            ProcessHandle ph4 = pro4.toHandle();
            CompletableFuture<ProcessHandle> cf4 = ph4.onExit();
            CompletableFuture<String> CF4 = cf4.thenApply(PH4 -> {
                  System.out.println("\t" + "Third Thread: After
                        thenApply... Ping is Done. ph4 = PH4?: " +
                        ph4.equals(PH4));
                  return "Hello PID № " + PH4.pid();
            });
            // теж допоміжний цикл
            for (int i = 1; i <= 5; i++)
                  try {
                        System.out.println("3-" + i + ") Main
                              Thread...");
                        Thread.sleep(1000);
                  }
                  catch (InterruptedException e) {
                        e.printStackTrace();
                  }
            // очікування результату, який повертає метод thenApply
            System.out.println("Main Thread: After get... " +
                  CF4.get());
      }
}
```

```
Waiting for completion Notepad...
Notepad is Done?: true ph1 = PH1?: true
1-1) Main Thread...
1-2) Main Thread...
1-3) Main Thread...
1-4) Main Thread...
     First Thread: After thenRun... Ping is Done
     1-1) First Thread: After thenRun...
1-5) Main Thread...
2-1) Main Thread...
```

```
     1-2) First Thread: After thenRun...
2-2) Main Thread...
2-3) Main Thread...
     1-3) First Thread: After thenRun...
2-4) Main Thread...
     Second Thread: After thenAccept... Ping is Done. ph3 = PH3?: true
     2-1) Second Thread: After thenAccept...
2-5) Main Thread...
     1-4) First Thread: After thenRun...
3-1) Main Thread...
     2-2) Second Thread: After thenAccept...
3-2) Main Thread...
     1-5) First Thread: After thenRun...
3-3) Main Thread...
     2-3) Second Thread: After thenAccept...
3-4) Main Thread...
     1-6) First Thread: After thenRun...
     Third Thread: After thenApply... Ping is Done. ph4 = PH4?: true
3-5) Main Thread...
     2-4) Second Thread: After thenAccept...
     1-7) First Thread: After thenRun...
Main Thread: After get... Hello PID № 1420
```

Лістинг 3.8. Виконання асинхронних завдань, без очікування, доки підпроцес буде виконаний.

```
import java.io.*;
import java.util.concurrent.CompletableFuture;

public class ProcessHandleOnExitDemo2 {
     public static void main(String[] args) throws Exception {
          Process pro1 = new ProcessBuilder("ping", "google.com.ua")
               .start();
          /* асинхронна фонова задача, яка працює в окремому від
```

```
головного потоці і не повертає результату; така задача
завершує своє виконання, при одній із наступних умов:
- завершаться всі її інструкції, до того, як закінчить
роботу головний потік;
- головний потік завершить свою роботу, до моменту
завершення останньої інструкції фонової задачі;
P.S.: посилання CF1 не обов'язково отримувати, проте, при
необхідності його в подальшому можна використати */
CompletableFuture<Void> CF1 = CompletableFuture
      .runAsync(() -> {
            System.out.println("\t" + "First Thread: After
                  runAsync... Ping is Run");
            BufferedReader bri = new BufferedReader(new
                  InputStreamReader(pro1.getInputStream()));
            String line;
            try {
                  while ((line = bri.readLine()) != null) {
                        System.out.println("\t" + line);
                        Thread.sleep(500);
                  }
            }
            catch (Exception e) {
                  e.printStackTrace();
            }
            System.out.println("\t" + "First Thread: After
                  runAsync... Ping is Done");
      });
for (int i = 1; i <= 5; i++)
      try {
            System.out.println("1-" + i + ") Main
                  Thread...");
            Thread.sleep(1000);
      }
      catch (Exception e) {
```

```
                e.printStackTrace();
            }

        Process pro2 = new ProcessBuilder("ping", "ibm.com")
            .start();
        /* дія методу supplyAsync аналогічна з runAsync, проте
        задача повертає деякий результат */
        CompletableFuture<Integer> CF2 = CompletableFuture
            .supplyAsync(() -> {
                System.out.println("\t" + "Second Thread: After
                    supplyAsync... Ping is Run");
                try {
                    int exit = pro2.waitFor();
                    System.out.println("\t" + "Second Thread:
                        After supplyAsync... Ping is Done");
                    return exit;
                }
                catch (Exception e) {
                    throw new RuntimeException(e);
                }
            });
        for (int i = 1; i <= 5; i++)
            try {
                System.out.println("2-" + i + ") Main
                    Thread...");
                Thread.sleep(1000);
            }
            catch (Exception e) {
                e.printStackTrace();
            }
        // очікування результату, який повертає метод supplyAsync
        System.out.println("Main Thread: After get... " +
            CF2.get());
    }
```

```
}
```

```
1-1) Main Thread...
     First Thread: After runAsync... Ping is Run

     Pinging google.com.ua [192.178.25.163] with 32 bytes of data:
1-2) Main Thread...
     Reply from 192.178.25.163: bytes=32 time=25ms TTL=118
     Reply from 192.178.25.163: bytes=32 time=24ms TTL=118
1-3) Main Thread...
     Reply from 192.178.25.163: bytes=32 time=23ms TTL=118
1-4) Main Thread...
     Reply from 192.178.25.163: bytes=32 time=24ms TTL=118

1-5) Main Thread...
     Ping statistics for 192.178.25.163:
         Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
2-1) Main Thread...
     Second Thread: After supplyAsync... Ping is Run
     Approximate round trip times in milli-seconds:
         Minimum = 23ms, Maximum = 25ms, Average = 24ms
2-2) Main Thread...
     First Thread: After runAsync... Ping is Done
2-3) Main Thread...
2-4) Main Thread...
     Second Thread: After supplyAsync... Ping is Done
2-5) Main Thread...
Main Thread: After get... 0
```

Лістинг 3.9. Виконання ланцюгових асинхронних завдань.

```
import java.util.concurrent.CompletableFuture;

public class ProcessHandleOnExitDemo3 {
```

```
public static void main(String[] args) throws Exception {
	Process pro = new ProcessBuilder("ping", "google.com.ua")
		.start();
	ProcessHandle ph = pro.toHandle();
	CompletableFuture<ProcessHandle> cf = ph.onExit();

	// поєднання thenRun/thenRun
	CompletableFuture<Void> CF1 = cf.thenRun(() -> {
		System.out.println("\tCF1 (First thenRun): Ping is
			Done");
	}).thenRun(() -> {
		System.out.println("\tCF1 (Second thenRun): First
			thenRun is Done");
	});

	// поєднання supplyAsync/thenRun
	CompletableFuture<Void> CF2 = CompletableFuture
		.supplyAsync(() -> {
			try {
				Thread.sleep(1000);
			}
			catch (InterruptedException e) {
				e.printStackTrace();
			}
			if (pro.isAlive())
				System.out.println("\tCF2 (First
					supplyAsync): Ping is Run");
			else
				System.out.println("\tCF2 (First
					supplyAsync): Ping is Done");
			/* щоб викликати послідуючий thenRun, необхідно
			повернути CompletableFuture<T>, неважливо якого
			типу; в даному випадку повертається вкладений
```

```
            <CompletableFuture<ProcessHandle>>
            CompletableFuture */
            return ph.onExit();
      }).thenRun(() -> {
            System.out.println("\tCF2 (Second thenRun): First
                  supplyAsync is Done");
      });

// поєднання supplyAsync/thenAccept
CompletableFuture<Void> CF3 = CompletableFuture
      .supplyAsync(() -> {
            try {
                  Thread.sleep(1000);
                  if (pro.isAlive())
                        System.out.println("\tCF3 (First
                              supplyAsync): Ping is Run");
                  else
                        System.out.println("\tCF3 (First
                              supplyAsync): Ping is Done");
                  return pro.waitFor();
            }
            catch (Exception e) {
                  throw new RuntimeException(e);
            }
      }).thenAccept(cf3 -> {
            // значенням cf3 не обов'язково користуватися
            System.out.println("\tCF3 (Second thenRun): First
                  supplyAsync is Done");
            System.out.println("\tCF3 (Second thenRun): Ping
                  is Done");
      });

// допоміжний цикл
for (int i = 1; i <= 5; i++)
```

```
        try {
            System.out.println("1-" + i + ") Main
                Thread...");
            Thread.sleep(1000);
        }
        catch (Exception e) {
            e.printStackTrace();
        }

// поєднання supplyAsync/thenApply
CompletableFuture<String> CF4 = CompletableFuture
        .supplyAsync(() -> {
            if (pro.isAlive())
                return "\tCF4 (First supplyAsync): Ping is
                    Run";
            else
                return "\tCF4 (First supplyAsync): Ping is
                    Done";
        }).thenApply(message1 -> {
            if (pro.isAlive())
                return message1 + "\n\tCF4 (First
                    thenApply): Ping is Run";
            else
                return message1 + "\n\tCF4 (First
                    thenApply): Ping is Done";
        }).thenApply(message2 -> {
            if (pro.isAlive())
                return message2 + "\n\tCF4 (Second
                    thenApply): Ping is Run";
            else
                return message2 + "\n\tCF4 (Second
                    thenApply): Ping is Done";
        });
// очікування результату
```

```
            System.out.println(CF4.get());
            System.out.println("Main Thread: After get...");
      }
}
```

```
1-1) Main Thread...
1-2) Main Thread...
     CF3 (First supplyAsync): Ping is Run
     CF2 (First supplyAsync): Ping is Run
     CF2 (Second thenRun): First supplyAsync is Done
1-3) Main Thread...
1-4) Main Thread...
     CF3 (Second thenRun): First supplyAsync is Done
     CF3 (Second thenRun): Ping is Done
     CF1 (First thenRun): Ping is Done
     CF1 (Second thenRun): First thenRun is Done
1-5) Main Thread...
     CF4 (First supplyAsync): Ping is Done
     CF4 (First thenApply): Ping is Done
     CF4 (Second thenApply): Ping is Done
Main Thread: After get...
```

Лістинг 3.10. Поєднання двох асинхронних залежних одна від одної задач.

```
import java.util.concurrent.CompletableFuture;

public class ProcessHandleOnExitDemo4 {
      public static void main(String[] args) throws Exception {
            CompletableFuture<Long> cf1 = CompletableFuture
                  .supplyAsync(() -> {
                        long startTime = System.currentTimeMillis();
                        System.out.println("\tcf1: Ping is Run");
                        try {
                              Process pro = new ProcessBuilder("ping",
```

```
                    "microsoft.com").start();
                pro.waitFor();
            }
            catch (Exception e) {
                e.printStackTrace();
            }
            System.out.println("\tcf1: Ping is Done");
            return System.currentTimeMillis() - startTime;
        });
/* поєднання двох залежних одна від одної двох задач cf1 і
лямбда-функції, яка є параметром в методі thenCompose (по-суті
поєднання supplyAsync/supplyAsync); аргумент time1 методу
thenCompose є результатом попереднього кроку обчислення cf1;
таким чином, друга асинхронна задача (аргумент лямбда-функція)
чекатиме доки не отримає результат від першої задачі cf1;
метод thenCompose досить схожий на вкладені thenApply, проте
якщо thenApply повертає звичайне значення, то thenCompose
повертає CompletableFuture; таким чином, thenCompose слід
використовувати тоді, коли слід повернути CompletableFuture та
позбавитися від вкладеності */
CompletableFuture<Long> cf2 = cf1.thenCompose(time1 ->
        CompletableFuture.supplyAsync(() -> {
            long startTime = System.currentTimeMillis();
            System.out.println("\tcf2: Ping is Run");
            try {
                Process pro = new ProcessBuilder("ping",
                    "oracle.com").start();
                pro.waitFor();
            }
            catch (Exception e) {
                e.printStackTrace();
            }
            System.out.println("\tcf2: Ping is Done");
            long time2 = System.currentTimeMillis() -
```

```
                        startTime;
                  return time1 + time2;
            }));
      for (int i = 1; i <= 5; i++)
            try {
                  System.out.println("1-" + i + ") Main
                        Thread...");
                  Thread.sleep(1000);
            }
            catch (Exception e) {
                  e.printStackTrace();
            }
      System.out.println("Total Ping Time: " + cf2.get() + " ms");
  }
}
```

```
     cf1: Ping is Run
1-1) Main Thread...
1-2) Main Thread...
1-3) Main Thread...
1-4) Main Thread...
     cf1: Ping is Done
     cf2: Ping is Run
1-5) Main Thread...
     cf2: Ping is Done
Total Ping Time: 22068 ms
```

Лістинг 3.11. Поєднання двох асинхронних незалежних одна від одної задач.

```
import java.util.concurrent.CompletableFuture;

public class ProcessHandleOnExitDemo5 {
     public static void main(String[] args) throws Exception {
```

```
CompletableFuture<Long> cf1 = CompletableFuture
      .supplyAsync(() -> {
            long startTime = System.currentTimeMillis();
            System.out.println("\tcf1: Ping is Run");
            try {
                  Process pro = new ProcessBuilder("ping",
                        "google.com.ua").start();
                  pro.waitFor();
            }
            catch (Exception e) {
                  e.printStackTrace();
            }
            System.out.println("\tcf1: Ping is Done");
            return System.currentTimeMillis() - startTime;
      });
CompletableFuture<Long> cf2 = CompletableFuture
      .supplyAsync(() -> {
            long startTime = System.currentTimeMillis();
            System.out.println("\tcf2: Ping is Run");
            try {
                  Process pro = new ProcessBuilder("ping",
                        "ibm.com").start();
                  pro.waitFor();
            }
            catch (Exception e) {
                  e.printStackTrace();
            }
            System.out.println("\tcf2: Ping is Done");
            return System.currentTimeMillis() - startTime;
      });
/* виконання задачі cf3 одразу ж по завершенню незалежних
одна від одної двох задач cf1/cf2 */
CompletableFuture<Long> cf3 = cf1.thenCombine(cf2,
      (time1, time2) -> {
```

```
                    System.out.println("\tcf1/cf2 is Done");
                    return time1 + time2;
               });
          // допоміжний цикл
          for (int i = 1; i <= 5; i++)
               try {
                    System.out.println("1-" + i + ") Main
                         Thread...");
                    Thread.sleep(1000);
               }
               catch (Exception e) {
                    e.printStackTrace();
               }
          // отримання результату задачі cf3
          System.out.println("Total Ping Time1: " + cf3.get()
               + " ms");
     }
}
```

```
     cf1: Ping is Run
     cf2: Ping is Run
1-1) Main Thread...
1-2) Main Thread...
1-3) Main Thread...
1-4) Main Thread...
     cf1: Ping is Done
     cf2: Ping is Done
     cf1/cf2 is Done
1-5) Main Thread...
Total Ping Time1: 6202 ms
```

Лістинг 3.12. Поєднання декількох асинхронних задач.

```
import java.util.concurrent.CompletableFuture;
```

```
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class ProcessHandleOnExitDemo6 {

	// метод, по запуску асинхронної задачі
	static CompletableFuture<Long> runPing(String webSite) {
		return CompletableFuture.supplyAsync(() -> {
			try {
				Process pro = new ProcessBuilder("ping", webSite)
					.start();
				System.out.println("\t" + webSite + ": Ping is
					Run");
				pro.waitFor();
				System.out.println("\t" + webSite + ": Ping is
					Done");
				return pro.pid();
			} catch (Exception e) {
				e.printStackTrace();
				throw new RuntimeException(e);
			}
		});
	}

	public static void main(String[] args) throws Exception {
		System.out.println("Main: Start 3 Task");
		// запуск 1-ої асинхронної задачі
		CompletableFuture<Long> cf1 = runPing("google.com.ua");
		// запуск 2-ої асинхронної задачі
		CompletableFuture<Long> cf2 = runPing("ibm.com");
		// запуск 3-ої асинхронної задачі
		CompletableFuture<Long> cf3 = runPing("oracle.com");
		/* поєднання декількох асинхронних задач;
		як тільки дані задачі виконають свою роботу, зі змінною
```

```
allFuture можна буде проводити ті чи інші маніпуляції */
CompletableFuture<Void> allFuture = CompletableFuture
      .allOf(cf1, cf2, cf3);
// допоміжний цикл
for (int i = 1; i <= 5; i++)
      try {
            System.out.println("1-" + i + ") Main
                  Thread...");
            Thread.sleep(1000);
      }
      catch (Exception e) {
            e.printStackTrace();
      }
/* очікування доки не завершаться всі задачі;
при необхідності можна запустити ще одну задачу після
завершення даних трьох (напр. allFuture.thenRun) */
allFuture.get();
/* оскільки allFuture має тип CompletableFuture<Void>, тому
він не повертає жодного значення; враховуючи це, для доступу
до результату кожної із задач, необхідно скористатися
записом виду:
System.out.println(cf1.get());
System.out.println(cf2.get());
System.out.println(cf3.get());
якщо ж необхідно отримати результат всіх задач, тоді можна
скористатися нижче описаним кодом, що використовує
Stream-потік */
String result = Stream.of(cf1, cf2, cf3)
      /* конвертація елементів Stream-потоку:
      CompletableFuture<Long> -> long -> String;
      робота методу join() аналогічна get(), проте join()
      додатково може генерувати RuntimeException, у випадку
      помилки виконання */
      .map(cf -> cf.join().toString())
```

```
        /* перетворення Stream-потоку рядків в єдиний рядок */
        .collect(Collectors.joining(" "));
System.out.println("Processes PID: " + result);

System.out.println("\nMain: Start 3 Task");
// запуск ще раз тих же самих 3-ох задач
cf1 = runPing("google.com.ua");
cf2 = runPing("ibm.com");
cf3 = runPing("oracle.com");
/* поєднання декількох асинхронних задач;
на відміну від allOf, метод anyOf дозволяє отримати
CompletableFuture, який дає змогу проводити ті чи інші
маніпуляції, як тільки закінчить свою роботу будь-яка перша
із зазначених задач; недоліком даного методу є те, що якщо
задачі повертають результат різного типу, то наперед не
можна буде знати тип Вашого CompletableFuture */
CompletableFuture<Object> anyFuture = CompletableFuture
        .anyOf(cf1, cf2, cf3);
/* запуск нової задачі, як тільки завершиться anyFuture
(перша із трьох) */
anyFuture.thenRun(() -> System.out.println("\t" + "Some
        Fastest Task is Done"));
// допоміжний цикл
for (int i = 1; i <= 5; i++)
        try {
                System.out.println("2-" + i + ") Main
                        Thread...");
                Thread.sleep(1000);
        }
        catch (Exception e) {
                e.printStackTrace();
        }
/* отримання результату, як тільки завершиться будь-яка із
задач; даний метод повертає результат першої відпрацьованої
```

```
            задачі */
            System.out.println("Fastest Process PID: "
                  + anyFuture.get());
      }
}
```

```
Main: Start 3 Task
1-1) Main Thread...
      oracle.com: Ping is Run
      google.com.ua: Ping is Run
      ibm.com: Ping is Run
1-2) Main Thread...
1-3) Main Thread...
1-4) Main Thread...
      google.com.ua: Ping is Done
      ibm.com: Ping is Done
1-5) Main Thread...
      oracle.com: Ping is Done
Processes PID: 13288 10728 4364

Main: Start 3 Task
      google.com.ua: Ping is Run
2-1) Main Thread...
      oracle.com: Ping is Run
      ibm.com: Ping is Run
2-2) Main Thread...
2-3) Main Thread...
2-4) Main Thread...
      ibm.com: Ping is Done
      Some Fastest Task is Done
      google.com.ua: Ping is Done
2-5) Main Thread...
Fastest Process PID: 1448
```

Запитання:

1. Що таке образ процесу? Яка його структура?
2. Що таке дескриптор процесу? Яку інформацію він вміщає?
3. Дайте характеристику таблиці процесів?
4. Наведіть приклади описувачів процесів для різних ОС.
5. Що собою являє черга процесів?
6. Розкажіть детально про контекст процесу.
7. Яким чином можна програмно отримати дескриптор процесу? Які програмні структури використовуються для роботи з дескрипторами процесів?
8. Які програмні методи/функції дескриптора дозволяють отримати доступ до основних атрибутів процесу?
9. Чим відрізняються завдання від процесів? Що таке відкладені завдання?
10. Які програмні структури та їх методи/функції використовуються для синхронного та асинхронного програмування відкладених завдань?

Вправи:

1. Спробувати знищити деяку кількість запущених в системі процесів з випадковими ідентифікаторами, які є в системі. Закриття процесу не повинно стосуватися поточного процесу. Кількість процесів програмно визначається календарним числом.
2. Створити масив із 5-ти процесів (на власний розсуд). В поточному процесі визначити, який із даних процесів має найменший час виконання.
3. Вивести інформацію про всі процеси в системі, окрім інтернет-браузерів.
4. Завершити в головному процесі виконання його підпроцесів в другому поколінні («онуків»).
5. Визначити всі ідентифікатори процесів, що відповідають за одну програму (напр. Chrome).

6. Отримати всі запущені процеси, інстальовані поверх ОС (тобто не системні).

7. У випадку, якщо в системі запущений процес Eclipse (або будь-який інший на власний розсуд), запустити новий процес Eclipse, натомість старий екземпляр даного процесу знищити.

8. Визначити які з процесів ОС були запущені пізніше ніж о 15:00. Якщо такі є, перші три із них знищити.

9. Написати програму, яка постійно моніторить запуск побічних процесів, намагаючись зупинити їх виконання, якщо вони були запущені у вечірній час (22:00 ÷ 00:00).

10. Завершити роботу всіх дочірніх процесів по відношенню до головного процесу. Процеси, які йдуть в наступних поколіннях повинні продовжувати виконувати свою роботу.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ВВП – віддалений виклик процедур;

ЕОМ – електронно-обчислювальна машина;

МПВ – міжпроцесна взаємодія;

ОП – оперативна пам’ять;

ОС – операційна система;

ПЗ – програмне забезпечення;

ППЗ – прикладне програмне забезпечення;

СВ – системний виклик;

СПЗ – системне програмне забезпечення;

ЦП – центральний процесор.

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. A. Guerrieri Hands-On System Programming with Go: Build Modern and Concurrent Applications for Unix and Linux Systems Using Golang, Kindle Edition / Guerrieri A. – Packt Publishing, 2019. – 817 p.
2. A. Hoover System Programming with C and Unix 1st Edition / Hoover A. – Pearson, 2009. – 379 p.
3. A. McHoes Understanding Operating Systems, 8th Edition / McHoes A., Flynn I.M. – Cengage Learning, 2017. – 592 p.
4. A. Silberschatz Operating System Concepts, 10th Edition / Silberschatz A., Gagne G., Galvin P.B. – Wiley, 2021. – 1040 p.
5. A.S. Tanenbaum Modern Operating Systems, Rental Edition, 5th Edition / Tanenbaum A.S., Bos H. – Pearson, 2022. – 1184 p.
6. Albert Y. Zomaya Parallel and Distributed Computing Handbook / Albert Y. Zomaya. – McGraw-Hill, 2018. – 1179 p.
7. B. Goetz Java Concurrency in Practice / Goetz B. – Addison-Wesley Professional, 2006. – 432 p.
8. D. Dhamdhere Operating Systems, 1st Edition / Dhamdhere D. – McGraw-Hill Education, 2008. – 864 p.
9. D. Lea Concurrent Programming in Java: Design Principles and Pattern. 2nd Edition / Lea D. – Addison-Wesley Professional, 1999. – 422 p.
10. Dr Edward Lavieri Mastering Java 11: Develop Modular and Secure Java Applications Using Concurrency and Advanced JDK Libraries, 2nd Edition / Dr Edward Lavieri. – Packt Publishing, 2018. – 462 p.
11. Dr. Rian Quinn Hands-On System Programming with C++: Build Performant and Concurrent Unix and Linux Systems with C++17 / Dr. Rian Quinn. – Packt Publishing, 2018. – 522 p.
12. G. Loutsky Parallel Computing / Loutsky G., Zhukov I., Korochkin A. – Kyiv, Kornechuk, 2007. – 216 p.
13. G. Tomsho Guide to Operating Systems (MindTap Course List), 6th Edition / Tomsho G. – Cengage Learning, 2020. – 608 p.

14. H. Schildt Java: A Beginner's Guide, 8th Edition / Schildt H. – McGraw-Hill, 2018. – 720 p.

15. H. Schildt Java: The Complete Reference 8th Edition, 8th Edition / Schildt H. – McGraw-Hill, 2011. – 1000 p.

16. J. Bloch Effective Java 3rd Edition / Bloch J. – Addison-Wesley Professional, 2017. – 416 p.

17. J. Gosling The Java Language Specification. Java SE 11 / Gosling J., Joy B., Steele G., Bracha G., Buckley A., Smith D. – Boston: Addison-Wesley; Oracle America, Inc. Edition, 2018. – 755 p.

18. J. Holcombe Survey of Operating Systems, 6th Edition / Holcombe J., Holcombe C. – McGraw Hill, 2019. – 496 p.

19. J.-B. Persson Linux System Programming Techniques: Become a Proficient Linux System Programmer Using Expert Recipes and Techniques / Persson J.-B. – Packt Publishing, 2021. – 432 p.

20. J.M. Hart Windows System Programming, 4th Edition / Hart J.M. – Addison-Wesley Professional, 2010. – 609 p.

21. K. Sierra Head First Java: A Brain-Friendly Guide 3rd Edition / Sierra K., Bates B., Gee T. – O'Reilly Media, 2022. – 752 p.

22. K.U. Subhash Unix System Programming, Kindle Edition / Subhash K.U. – Morgan Kaufmann, 2022. – 503 p.

23. L.S. Darnell Create Your Own Operating System, 1st Edition / Darnell L.S. – CreateSpace Independent Publishing Platform, 2016. – 149 p.

24. P. Eshwarla Practical System Programming for Rust Developers: Build Fast and Secure Software for Linux/Unix Systems with the Help of Practical Examples / Eshwarla P. – Packt Publishing, 2020. – 388 p.

25. P. Yosifovich Windows 10 System Programming, Part 1 / Yosifovich P. – Independently published, 2020. – 528 p.

26. P. Yosifovich Windows 10 System Programming, Part 2. / Yosifovich P. – Independently published, 2021. – 555 p.

27. R. Anthony Systems Programming: Designing and Developing Distributed Applications, 1st Edition / Anthony R. – Morgan Kaufmann, 2015. – 548 p.

28. R. Love Linux System Programming: Talking Directly to the Kernel and C Library, 2nd Edition / Love R. – O'Reilly Media, 2013. – 456 p.

29. R.C. Martin Clean Code: A Handbook of Agile Software Craftsmanship 1st Edition / Martin R.C. – Pearson, 2008. – 464 p.

30. R.H. Arpaci-Dusseau Operating Systems: Three Easy Pieces, 1.00 Edition / Arpaci-Dusseau R.H., Arpaci-Dusseau A.C. – CreateSpace Independent Publishing Platform, 2018. – 714 p.

31. S. Rajasekaran Handbook of Parallel Computing. Models, Algorithms and Applications, 1st Edition / Rajasekaran S., Reif J. – New York: Chapman and Hall/CRC, 2007. –1224 p.

32. S.M. Palakollu Practical System Programming with C: Pragmatic Example Applications in Linux and Unix-Based Operating Systems, 1st ed. / Palakollu S.M. – Apress, 2020. – 292 p.

33. T. Anderson Operating Systems: Principles & Practice. Volume I: Kernels and Processes, 2nd edition / Anderson T., Dahlin M. – Recursive Books, 2015. – 296 p.

34. T. Anderson Operating Systems: Principles & Practice. Volume II: Concurency, 2nd Edition / Anderson T., Dahlin M. – Recursive Books, 2015. – 477 p.

35. T. McNamara Rust in Action, 1st Edition / McNamara T. – Manning, 2021. – 456 p.

36. T.W. Doeppner Operating Systems In Depth: Design and Programming, 1st Edition / Doeppner T.W. – Wiley, 2010. – 444 p.

37. U. Ramachandran Computer Systems: An Integrated Approach to Architecture and Operating Systems, 1st Edition / Ramachandran U., W. Leahy Jr. – Pearson, 2010. – 784 p.

38. W. Gropp Using MPI: Portable Parallel Programming with the Message-Passing Interface. 3rd Edition. / W. Gropp, E. Lusk, A. Skjellum. – Massachusetts Institute of Technology, 2014. – 330 с.

39. W. Stallings Operating Systems: Internals and Design Principles, 9th Edition / Stallings W. – Pearson, 2017. – 800 p.

40. Жуков І.А. Паралельні та розподілені обчислення: навч. посібник / І.А. Жуков, О.В. Корочків. – К.: «Корнійчук», 2005. – 226 с.

41. Качко О.Г. Паралельне програмування / О.Г. Качко. – Харків: ХНУРЕ, 2016. – 403 с.

42. Корочків О.В. Паралельні та розподілені обчислення. Вибрані розділи: навч. посібник [Електронний ресурс] / О.В. Корочкін, О.В. Русанова. – К.: КПІ ім. І. Сікорського, 2020. – 123 с.

43. Косовський В.М. Теорія паралельних обчислень / В.М. Коцовський. – Ужгород: ПП «АУТДОР–Шарк», 2021. – 188 с.

44. Кузьма К.Т. Паралельні та розподілені обчислення: навч. посібник для вищих закладів освіти / К.Т. Кузьма, О.В. Мельник. – Миколаїв: ФОП Швець В.М., 2020. – 172 с.

45. Кузьменко Б.В. Технологія розподілених систем та паралельних обчислень: конспект лекцій. Ч. 1. Розподілені об’єктні системи, паралельні обчислювальні системи та паралельні обчислення, паралельне програмування на основі MPI: навч. посібник / Б.В. Кузьменко, О.А. Чайковська – К.: Видавничий центр КНУКІМ, 2011 – 126 с.

46. Малашонок Г.І. Паралельні обчислення на розподіленій пам'яті: OpenMPI, Java, Math Partner: підручник / Г.І. Малашонок, А.А. Сідько. – Київ : НаУКМА, 2020. – 266 с.

47. Минайленко Р.М. Паралельні та розподілені обчислення: навч. посібник / Р.М. Минайленко. – Кропивницький: Видавець Лисенко В.Ф., 2021. – 153 с.

48. Рольщиков В.Б. Технології розподілених систем та паралельних обчислень: конспект лекцій / В.Б. Рольщиков. – Одеса: ОДЕКУ, 2016. – 155 с.

49. Семеренко В.П. Технології паралельних обчислень: навч. посібник / В.П. Семеренко. – Вінниця: ВНТУ, 2018. – 104 с.

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

ДЛЯ НОТАТОК

ДЛЯ НОТАТОК

Навчально-методичне видання

ЛЯШУК Тарас

ОПЕРАЦІЙНІ СИСТЕМИ ТА СИСТЕМНЕ ПРОГРАМУВАННЯ. Ч.1. БАГАТОЗАДАЧНІСТЬ: УПРАВЛІННЯ ПРОЦЕСАМИ

Навчальний посібник

Підп. до др. 30.09.2024 р.
Формат 60x84 $^{1}/_{16}$.
Папір офсет.
Друк цифр.
Гарнітура Times.
Ум. друк. арк. 11,6.
Обл. вид. арк. 11,6.
Тираж 300 прим.

Видавець: Олег Зень
Свідоцтво суб'єкта видавничої справи
серія РВ № 26 від 6 квітня 2004 р.
вул. Кн. Романа, 9/24, м. Рівне, 33022,
тел. 068 025 067 4, olegzen@ukr.net

Друк: VPM Поліграф
вул. Київська, 36,
м. Рівне, 33000;
642134@ukr.net