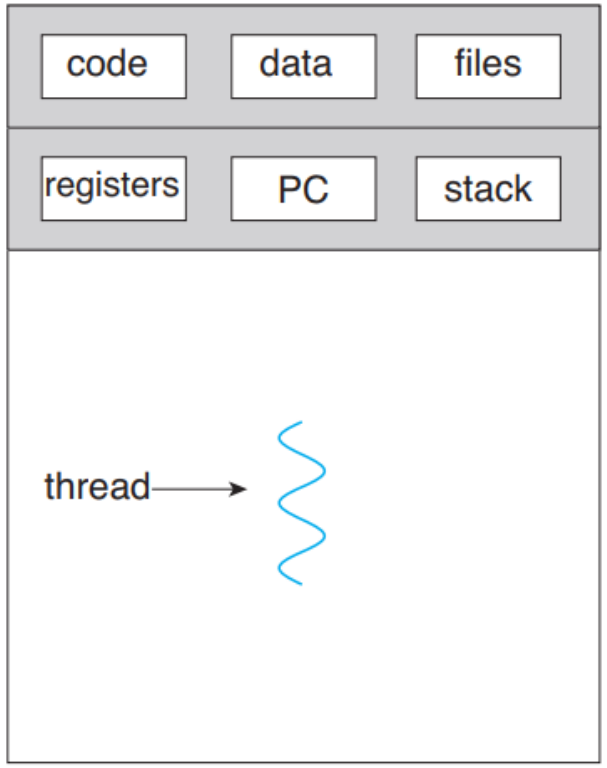
ПОТОКИ ТА МОДЕЛІ БАГАТОПОТОЧНОГО ВИКОНАННЯ КОДУ

Питання 2.2

Потоки



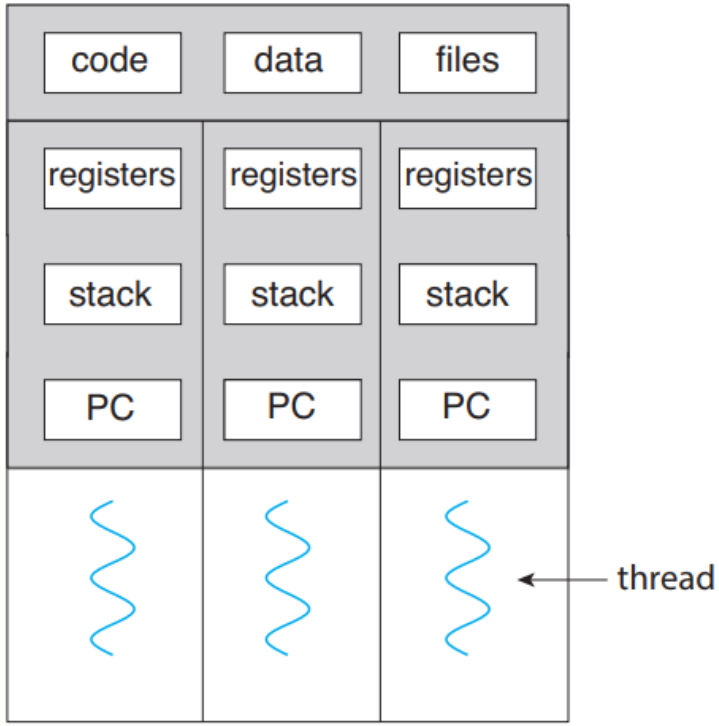
single-threaded process

Базова одиниця використання ЦП; включає ID потоку, program counter (PC), набір регістрів та стек.

- Ділить ресурси з іншими потоками в межах процесу: code section, сегмент даних та інші ресурси ОС, зокрема відкриті файли та сигнали.
- Традиційний процес має один потік управління.
- Якщо процес має багато потоків управління, він може виконувати більше однієї задачі (task) у момент часу.

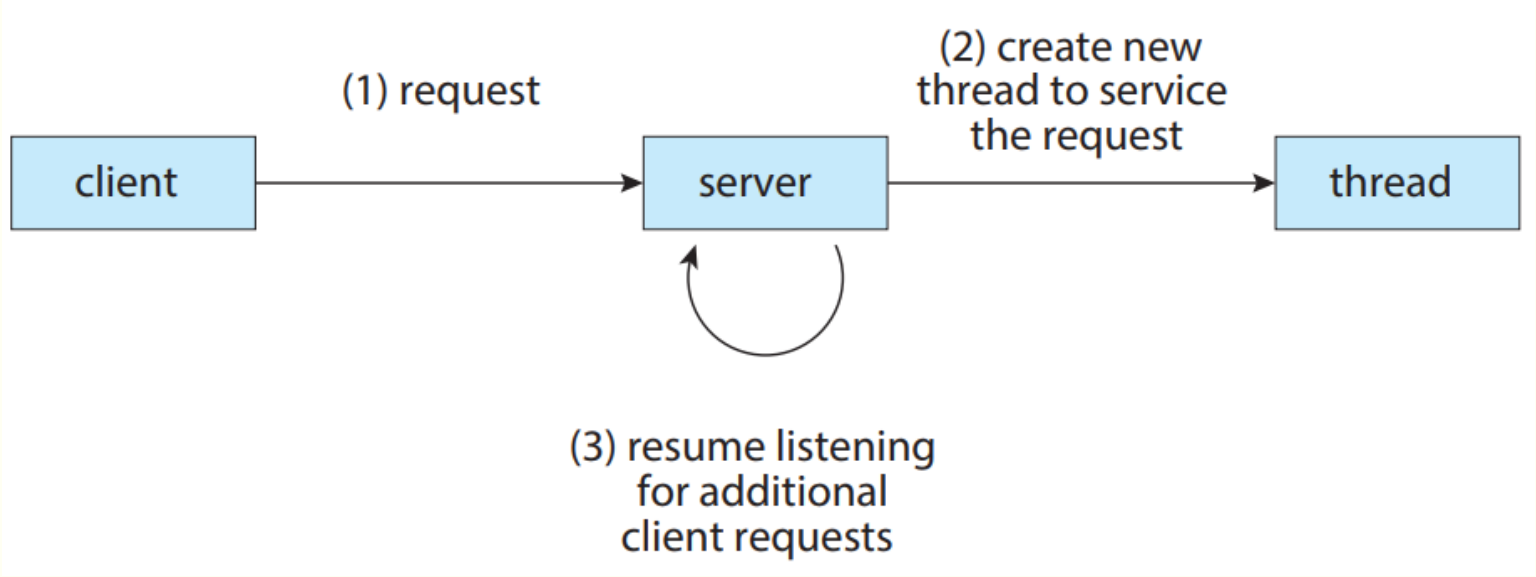
У деяких випадках один додаток може знадобитись для виконання кількох подібних задач.

- Наприклад, веб-сервер приймає запити від клієнтів на веб-сторінки, зображення, аудіо тощо.
- Завантажений веб-сервер може мати кілька (тисячі) клієнтів, які конкурентно отримують доступ.
- Якщо веб-сервер запустив однопоточний додаток, він зможе обслуговувати лише одного клієнта за раз, тобто клієнт може дуже довго чекати на обробку запиту.



multithreaded process

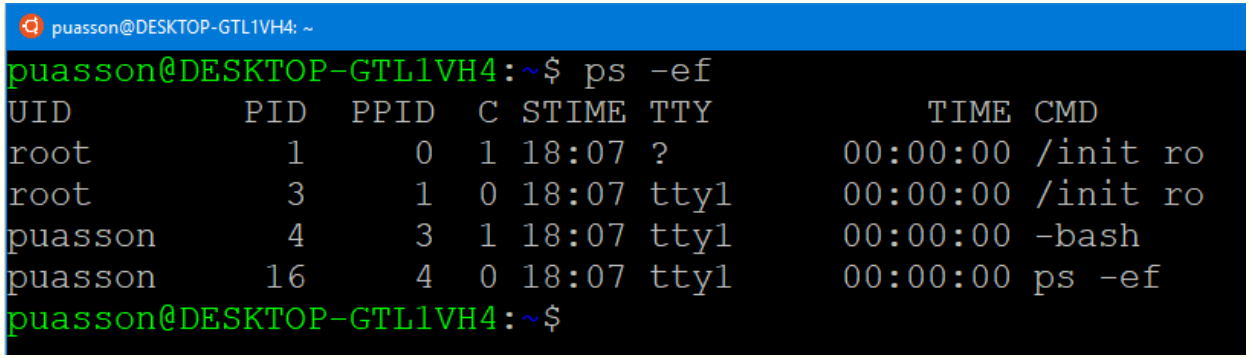
Багатопоточна серверна архітектура



- Одне з вирішень – мати сервер, який працює як один процес, який приймає запити.
  - При надходженні запиту він створює окремий процес для обробки цього запиту. Підхід був популярним до появи концепції потоків.
  - Створення процесу – часово затратна та ресурсоємна задача. Якщо новий процес виконуватиме ту ж задачу, навіщо такі додаткові витрати?
  - Загалом ефективніше використовувати 1 процес, який містить багато потоків.
  - Якщо веб-серверний процес багатопоточний, для кожного клієнта виділятиметься новий потік після запиту.

Більшість ядер ОС багатопоточні

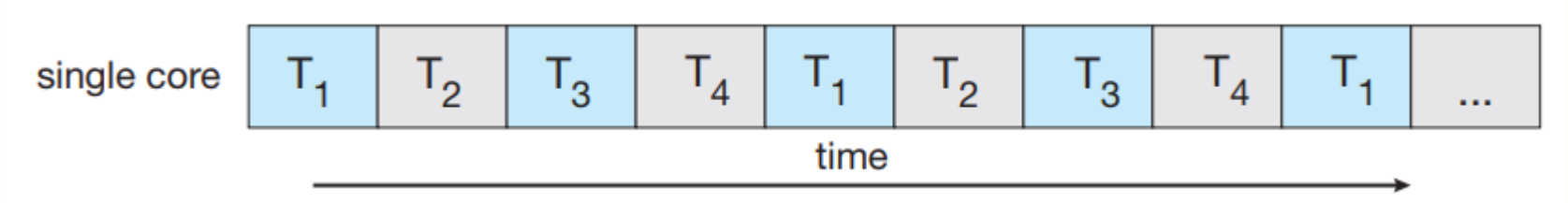
- Як приклад, при завантаженні Linux-систем створюється кілька kernel-потоків.
  - Кожен потік виконує конкретне завдання: управління пристроями, пам'яттю, обробка переривань тощо.
- Команда **`ps -ef`** може застосовуватись для показу kernel-потоків запущеної Linux-системи.
  - У переліку може бути kernel-потік kthreadd (pid = 2), який буде батьківським для інших kernel-потоків.
  - Багато додатків також можуть отримати переваги від багатьох потоків, зокрема, алгоритми сортування, алгоритми на базі дерев та графів.
  - Також програмісти ЦП-інтенсивних задач з видобування даних (data mining), графіки та штучного інтелекту можуть задіяти потужності сучасних багатоядерних систем, проектуючи паралелізоване ПЗ.



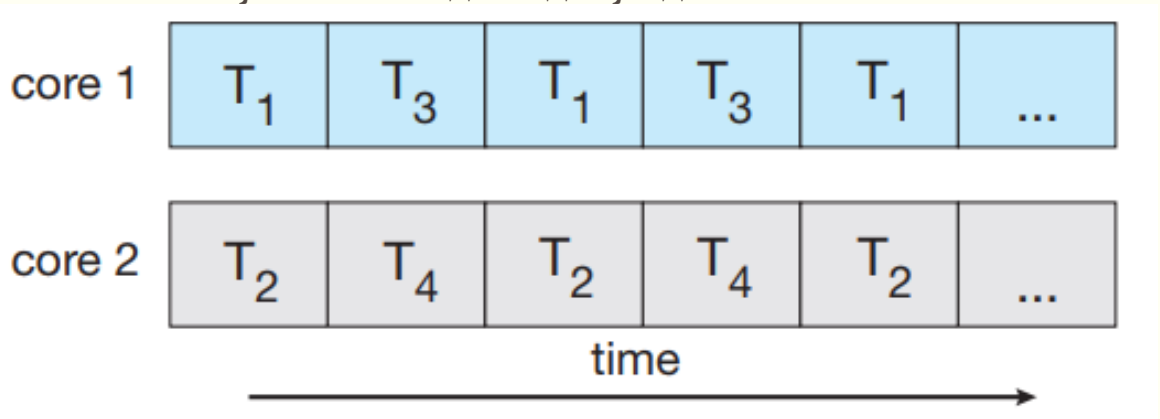
Переваги багатопоточності

- ***Респонсивність.*** Багатопоточність в інтерактивному додатку може дозволити програмі продовжувати роботу навіть якщо її частина заблокована або виконує тривалі операції.
  - Особливо корисно для користувацьких інтерфейсів.
  - Наприклад, клік по кнопці запускатиме тривалу операцію.
  - Однопоточний додаток не відповідатиме, поки операція не завершиться.
  - Запуск такої задачі в окремому асинхронному потоці дозволяє додатку залишатись респонсивним.
- ***Шеринг ресурсів.*** Ділитись ресурсами процеси можуть або через спільну пам'ять, або через передачу повідомлень.
  - Проте потоки ділять пам'ять та ресурси процесу, якому вони належать.
  - Перевага спільних сегментів коду та даних – можливість мати кілька різних потоків виконання діяльності в одному адресному просторі.
- ***Економія.*** Виділення пам'яті та ресурсів для створення процесу досить коштовне.
  - Маючи спільні ресурси процесу, створювати потоки та перемикатись між ними економніше.
  - Емпірично виміряти різницю в накладних витратах складно, проте загалом створення потоку споживає менше часу та пам'яті, а перемикання контексту зазвичай швидше між потоками, ніж процесами.
- ***Масштабованість.*** Переваги багатопоточності навіть зростають у мультипроцесних архітектурах, де потоки можуть працювати паралельно на різних ядрах.
  - Однопоточний процес може працювати лише на одному процесорі, незалежно від того, скільки їх буде.

Програмування багатоядерних систем



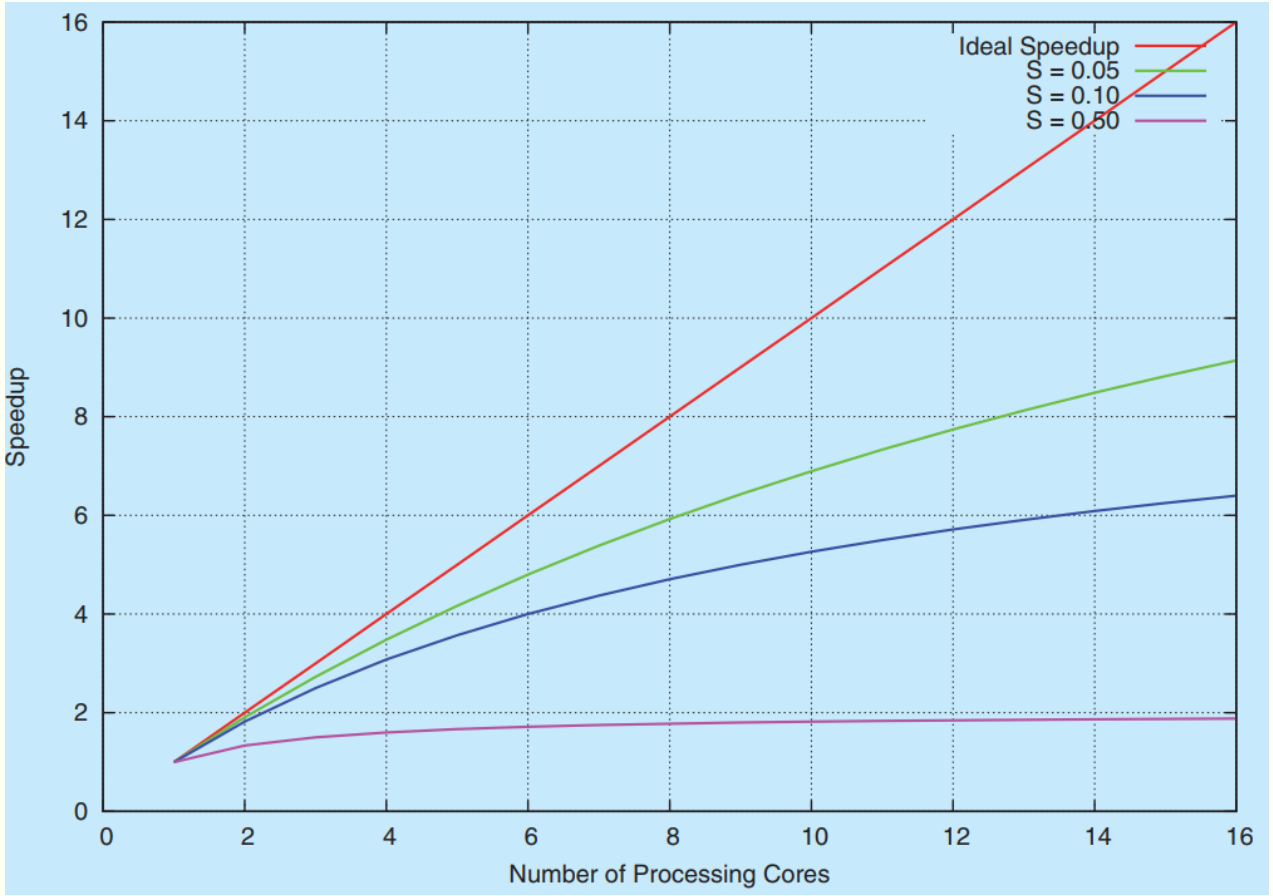
- На одноядерних системах конкурентність (concurrency) просто означає, що виконання потоків буде чергуватись з часом.
- Проте на багатоядерних системах конкурентність означає, що деякі потоки можуть працювати паралельно, оскільки система може присвоїти окремий потік кожному ядру.
  - Конкурентна система підтримує більше 1 задачі, дозволяючи всім задачам прогресувати.
  - Паралельна система може виконувати понад 1 задачу одночасно.



Виклики при програмуванні

- **1. Ідентифікація задач (tasks)**. Включає пошук у додатку частин, які можна розбити на окремі, конкурентні задачі. В ідеалі, задачі незалежні одна від одної.
- **2. Баланс**. Програмісти повинні також забезпечити виконання задачами рівноцінної роботи. Інколи деякі задачі можуть не робити великий внесок в загальний процес, в порівнянні з іншими задачами. Використання окремих ядер може бути не вартим того.
- **3. Розбиття даних (Data splitting)**. Як додатки розбиваються на окремі задачі, так і дані, які ці задачі отримують та якими оперують, повинні розподілятись на окремі ядра.
- **4. Залежність даних (Data dependency)**. Дані, доступні задачам, повинні перевірятись на залежності між 2 або більше задачами.
  - Коли одна задача залежить від даних з іншої, програміст повинен синхронізувати обчислення.
- **5. Тестування та налагодження**. Коли програма працює на багатьох ядрах, можливі багато різних шляхів виконання (execution paths).
  - Тестування та налагодження конкурентних програм набагато складніше, ніж однопоточних додатків.

Закон Амдаля (Amdahl's Law)

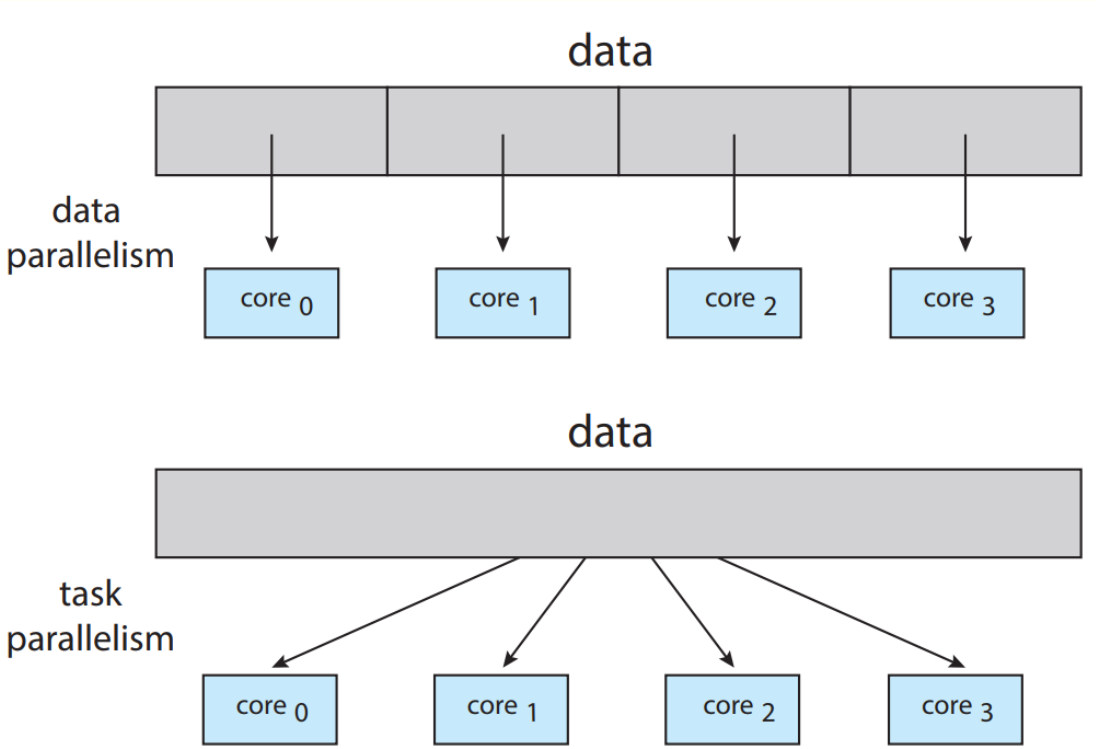


- Формула, яка визначає потенційний приріст продуктивності від додавання нових обчислювальних ядер для додатку, який має як послідовні, так і паралельні частини коду.
- Нехай S – відсоток послідовно виконуваного коду в системі з N ядрами:

$$\text{Прискорення} \leq \frac{1}{S + \frac{1-S}{N}}$$

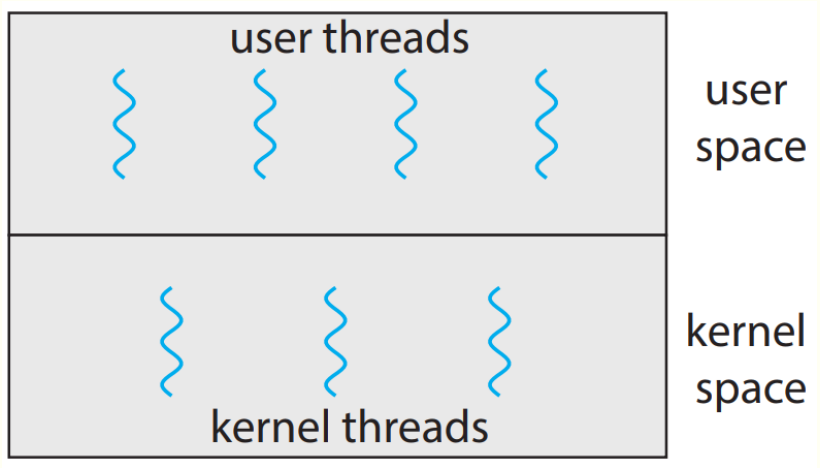
- Для співвідношення 75% паралелізованого коду на 25% послідовного прискорення на 2 ядрах буде максимум в 1.6 раз.
  - Для 4-ядерної системи прискорення складе 2.28 раза.

Типи паралелізму

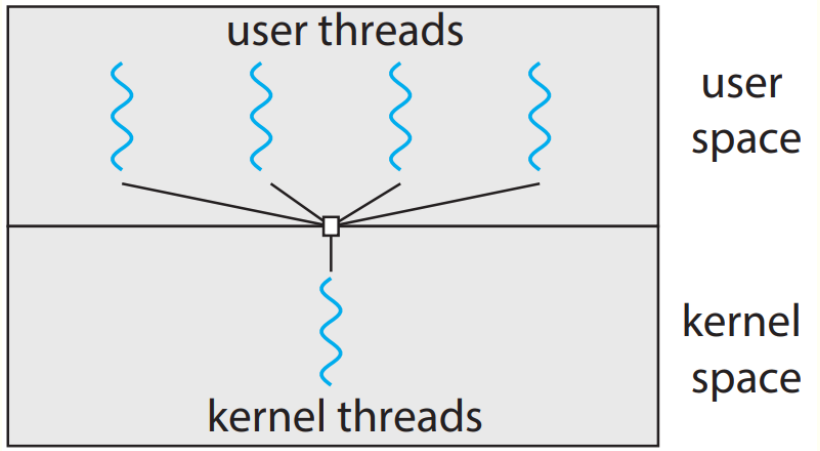


- **Паралелізм за даними** концентрується на розподілі однакових підмножин даних на багатьох обчислювальних ядрах та виконанні однієї операції на кожному ядрі.
  - Нехай треба додати вміст масиву розміру *N*. На одноядерній системі 1 потік просто просумує елементи [0] . . . [*N* - 1].
  - На 2-ядерній - потік *A на ядрі 0* може просумувати елементи [0] . . . [*N*/2 - 1], а потік *B на ядрі 1* – елементи [*N*/2] . . . [*N* - 1].
- **Паралелізм за задачами** задіює розподіл за задачами на потоки між багатьма обчислювальними ядрами.
  - Кожен потік виконуватиме унікальну операцію.
  - Різні потоки можуть оперувати одними або різними даними.
  - На попередньому прикладі паралелізм за задачами на 2 потоках виконує унікальні статистичні операції над масивом елементів.

Багатопоточні (Multithreading) моделі

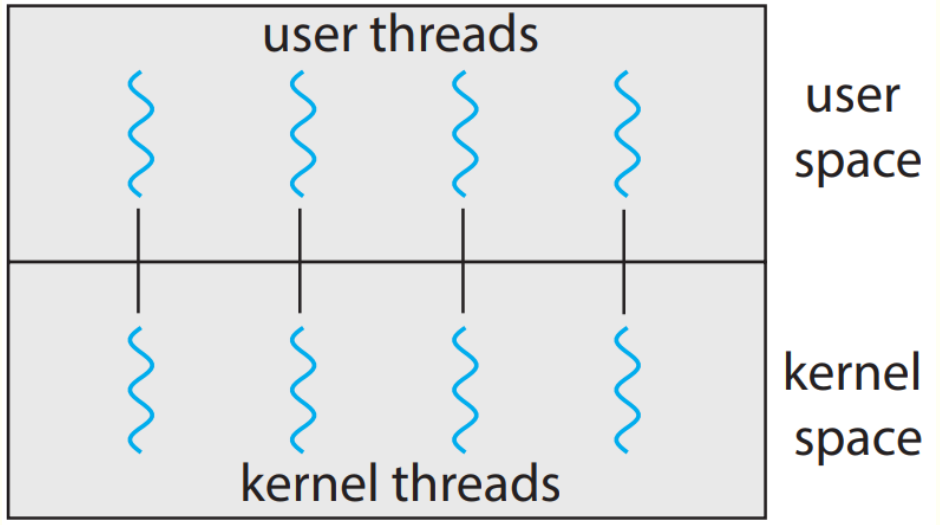


- Підтримка потоків може постачатись як на рівні користувача, так і на рівні ядра.
  - Користувацькі потоки управляються без підтримки ядра, а kernel-потоки підтримуються та управляються напряму ОС.
  - Майже всі сучасні ОС підтримують kernel-потоки.

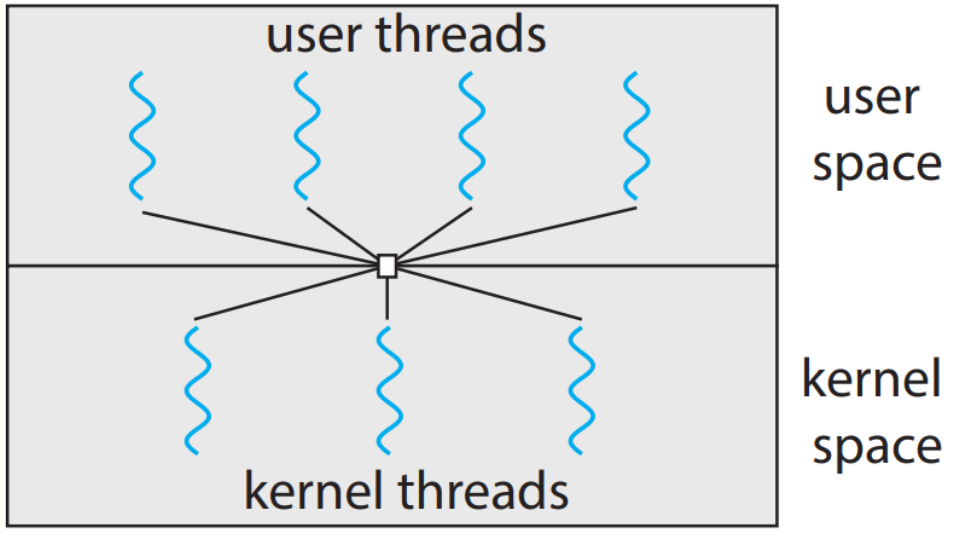


- Модель «багато-до-одного» відображає багато користувацьких потоків на один kernel-потік.
  - Управління потоком здійснюється ***трединговою бібліотекою (thread library)*** в просторі користувача, що ефективно.
  - Проте весь процес блокується, якщо потік виконує блокуючий системний виклик.
  - Також оскільки тільки 1 потік може отримати доступ до ядра за раз, багато потоків не можуть працювати паралельно на багатоядерних системах.

One-to-One Model. Many-to-Many Model

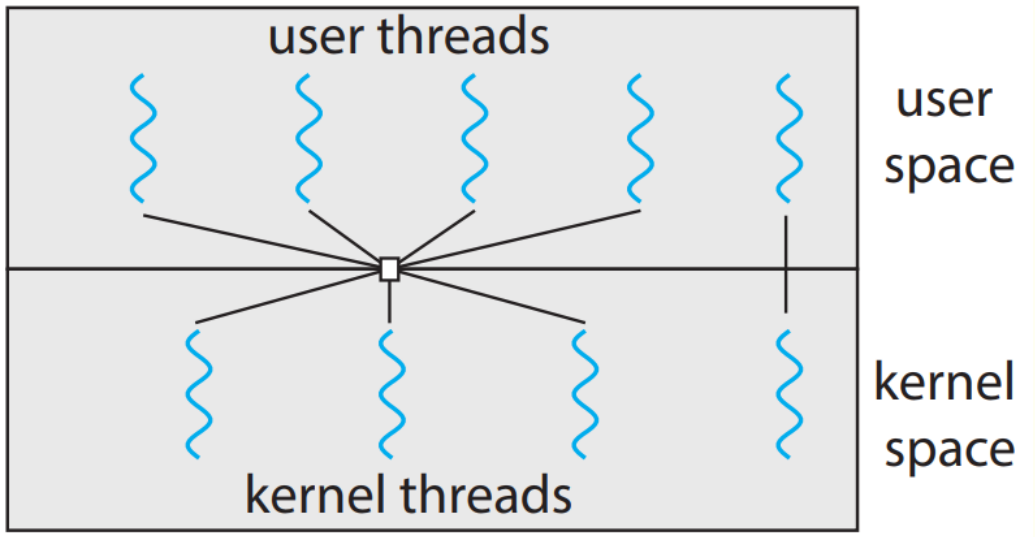


- Модель «один-до-одного» відображає користувацький потік на kernel-потік.
  - Забезпечує кращу конкурентність, ніж модель «багато-до-одного», дозволяючи іншому потоку працювати, коли потік здійснює блокуючий системний виклик.
  - Також дозволяє багатьом потокам працювати паралельно на мультипроцесорних системах.
  - Недолік: створення користувацького потоку вимагає створення відповідного kernel-потоку, проте велика кількість kernel-потоків може суттєво знизити швидкодію ОС.
  - Linux та Windows реалізують дану модель.



- Модель «багато-до-багатьох» мультиплексує багато користувацьких потоків на меншу або рівну кількість kernel-потоків.
  - Кількість kernel-потоків може бути специфічною або для конкретного додатку, або конкретної машини (залежно від кількості доступних ядер).

Дворівнева модель (two-level model)



- Модель «багато-до-одного» не підтримує паралелізм, оскільки ядро може спланувати лише 1 kernel-потік за раз.
  - Модель 1:1 допускає кращу конкурентність, проте розробник має бути обережним при створенні занадто великої кількості потоків у додатку.
  - Модель M:M не страждає цими недоліками. Також при виконанні потоком блокуючого системного виклику ядро може спланувати інший потік для виконання.
- Одна з варіацій моделі M:M також дозволяє користувацькому потоку прив'язатись до kernel-потоку.
  - Інколи таку варіацію називають ***дворівневою моделлю***.

Тредингові бібліотеки

- Тредингова бібліотека надає програмісту API для створення та управління потоками.
  - Існують 2 основні підходи до реалізації тредингової бібліотеки.
- 1) надати бібліотеку повністю в просторі користувача без підтримки ядра.
  - Весь код та структури даних для бібліотеки існують на рівні користувача. Виклик функції в бібліотеці призводить до локального виклику функції з простору користувача, а не системного виклику.
- 2) реалізувати бібліотеку на рівні ядра, яка підтримується напряму ОС.
  - Код і структури даних містяться в просторі ядра. Виклик функції з API для бібліотеки зазвичай призводить до системного виклику до ядра.
- Три основні тредингові бібліотеки сьогодні: POSIX Pthreads, Windows та Java.
  - Pthreads – тредингове розширення стандарту POSIX, може постачатись як бібліотека рівня ядра або користувача.
  - Тредингова бібліотека Windows – бібліотека рівня ядра, доступна на Windows-системах.
  - Java thread API дозволяє створювати та управляти потоками напряму з Java-програм.
  - Оскільки в більшості випадків Java-додаток працює на базі JVM поверх хостової ОС, Java thread API загалом реалізується за допомогою тредингової бібліотеки, доступної для хостової системи.

Тредингові бібліотеки

- Для POSIX- та Windows-тредингу будь-які глобально оголошені дані є спільними для всіх потоків, які належать одному процесу.
  - Оскільки Java не має еквівалентної нотації для глобальних даних, доступ до спільних даних повинен явно організовуватись між потоками.
  - Ілюстративний приклад: додавання до суми невід'ємного цілого числа в окремому потоці: Сума $= \sum_{i=1}^{N} i$
  - Кожна з трьох програм працюватиме з верхніми межами N, введеними в командному рядку.
- Представимо 2 загальні стратегії створення багатьох потоків: ***асинхронний тредінг*** та ***синхронний тредінг***.
  - ***Перша:*** як тільки parent створює дочірній потік, parent відновлює своє виконання та працює незалежно від дочірнього потоку та конкурентно.
    - Оскільки потоки незалежні, зазвичай шеринг даних між ними невеликий.
  - ***Друга:*** спрацьовує, коли батьківський потік створює один або кілька дочірніх процесів, а потім повинен чекати завершення їх роботи. Після цього всі дочірні потоки поєднуються з батьківським.
    - Зазвичай синхронний тредінг включає значний шеринг даних між потоками.
    - Наприклад, батьківський потік може комбінувати результати, обчислені його різними дочірніми потоками.

```c
#include <pthread.h>
#include <stdio.h>

#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   /* set the default attributes of the thread */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid, &attr, runner, argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will execute in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Pthreads

```c
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

- Стандарт POSIX (IEEE 1003.1c) визначає API для створення та синхронізації потоків (поведінку, а не реалізацію).
  - Проектувальники ОС можуть реалізувати специфікацію так, як вважають за потрібне.
  - Багато систем реалізують специфікацію Pthreads: більшість UNIX-подібних систем, включаючи Linux та macOS.
  - Хоч Windows не підтримує Pthreads нативно, деякі сторонні реалізації для Windows доступні.
- Програма демонструє основу Pthreads API для конструювання багатопоточної програми, яка знаходить суму невід'ємних цілих чисел в окремому потоці.
  - Окремий потік запускається спеціальною функцією runner().
  - На початку програми відбувається вхід в однопоточну функцію main().
  - Після ініціалізації main() створює другий потік, який запускає функцію runner().
  - Обидва потоки мають глобальні дані – змінна sum.

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* The thread will execute in this function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 1; i <= Upper; i++)
      Sum += i;
   return 0;
}

int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;

   Param = atoi(argv[1]);
   /* create the thread */
   ThreadHandle = CreateThread(
      NULL, /* default security attributes */
      0, /* default stack size */
      Summation, /* thread function */
      &Param, /* parameter to thread function */
      0, /* default creation flags */
      &ThreadId); /* returns the thread identifier */

    /* now wait for the thread to finish */
   WaitForSingleObject(ThreadHandle,INFINITE);

   /* close the thread handle */
   CloseHandle(ThreadHandle);

   printf("sum = %d\n",Sum);
}
```

Потоки Windows

- Аналогічно до Pthreads-версії, спільна змінна Sum для потоків оголошена глобально (DWORD = unsigned int32).
  - Визначається функція Summation(), яка виконуватиметься в окремому потоці та прийматиме вказівник на void (LPVOID).
- Потоки створюються у Windows API за допомогою функції CreateThread().
  - Передані атрибути включають безпекову інформацію, розмір стеку та прапорець, який вказує на запуск процесу в suspended стані.
- У ситуації, коли вимагається очікування завершення багатьох потоків застосовується функція WaitForMultipleObjects() з 4ма параметрами:
  - 1. Кількість об'єктів, яких слід очікувати
  - 2. Вказівник на масив об'єктів
  - 3. Прапорець, який вказує, чи всі об'єкти були сповіщені (signaled)
  - 4. Таймаут (або INFINITE)

Java Threads

- Усі Java-програми покладаються принаймні на 1 потік управління.
  - Потоки Java доступні на будь-якій системі, яка постачає JVM, включаючи Windows, Linux та macOS.
  - Java thread API доступний і для Android-додатків.
- Існують 2 підходи для створення потоків у Java-додатках:
  - 1) створити новий клас, породжений від класу Thread, та переозначити його метод run().
  - 2) більш поширений – визначити клас, який реалізує інтерфейс Runnable з одним абстрактним методом з сигнатурою «public void run()».

```
class Task implements Runnable
{
   public void run() {
      System.out.println("I am a thread.");
   }
}
```

```
Thread worker = new Thread(new Task());
worker.start();
```

- Виклик методу start() виконує 2 дії:
  - 1. Виділяє пам'ять та ініціалізує новий потік у JVM.
  - 2. Викликає метод run(), роблячи потік здатним до виконання на базі JVM. Напряму метод run() ніколи не викликається.

- Аналогічну функціональність до функцій pthread_join() та WaitForSingleObject() забезпечує метод join().
  - Зауважте, що він може викидати виняток InterruptedException, тому використовуємо оператор try-catch

```
try {
   worker.join();
}
catch (InterruptedException ie) { }
```

Java Executor Framework

- Починаючи з Java 5, введено кілька нових механізмів підтримки конкурентності з набагато кращим ступенем контролю створення та взаємодії потоків.
  - Інструменти доступні в пакеті java.util.concurrent.
  - Замість явного створення Thread-об'єктів створення потоку організується навколо інтерфейсу Executor:

```
public interface Executor
{
   void execute(Runnable command);
}
```

  - Екзекутор використовується так:

```
Executor service = new Executor();
service.execute(new Task());
```

- Executor framework базується на моделі виробник-споживач; задачі виробляються на основі інтерфейсу Runnable, а виконуючі потоки споживають їх.
  - Переваги: відокремлення створення потоку від його виконання, а також забезпечення механізму комунікації між конкурентними задачами.

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

class Summation implements Callable<Integer>
{
   private int upper;
   public Summation(int upper) {
      this.upper = upper;
   }

   /* The thread will execute in this method */
   public Integer call() {
      int sum = 0;
      for (int i = 1; i <= upper; i++)
         sum += i;

      return new Integer(sum);
   }
}

public class Driver
{
  public static void main(String[] args) {
     int upper = Integer.parseInt(args[0]);

     ExecutorService pool = Executors.newSingleThreadExecutor();
     Future<Integer> result = pool.submit(new Summation(upper));

     try {
         System.out.println("sum = " + result.get());
     } catch (InterruptedException | ExecutionException ie) { }
   }
}
```

Інтерфейс Callable та футури

Шеринг даних між потоками одного процесу ускладений в ОО мовах, як Java, оскільки немає глобальних змінних.

- Можна передати параметри в клас, який реалізує Runnable, проте потоки Java не можуть повертати результати.
- Для вирішення потреби пакет java.util.concurrent додатково визначає інтерфейс Callable.
- Результати від Callable-задач називають футурами (Future-об'єктами, фьючерсами).
- Отримати результат можна за допомогою методу get() з інтерфейсу Future.
- Тут клас Summation реалізує інтерфейс Callable, який задає метод «V call()».
- Для виконання коду створимо об'єкт за допомогою статичного методу newSingleThreadExecutor() та передамо цей об'єкт у Callable-задачу, використовуючи метод submit().

Неявний трединг

- Багатопоточний сервер матиме потенційні проблеми, незважаючи на швидше створення потоків, ніж процесів.
  - 1) Потік створюється та знищується відразу після виконання роботи.
  - 2) Якщо дозволяти кожному багатопоточному запиту оброблятись в окремому потоці, ми не накладаємо обмеження на кількість активних потоків у системі. Це призведе до вичерпання системних ресурсів.
  - Вирішенням останньої проблеми є використання ***пулу потоків (thread pool)***.
- Загальна ідея пулів потоків – створити багато потоків при запуску, а потім розміщати їх у пул, в якому вони працюватимуть.
  - Коли сервер отримує запит, він перенаправляє його в пул потоків та знову очікує на додаткові запити.
  - Якщо в пулі немає доступного потоку, він «просинається», і запит обробляється негайно.
  - Якщо всі потоки пулу зайняті, задача ставиться в чергу та очікує звільнення потоку для себе.
  - Як тільки потік завершує роботу, він повертається в пул, де чекатиме на нову.
  - Пули потоків добре працюють, коли направлені в пул задачі можна виконувати асинхронно.

Переваги пулу потоків

- 1. Обслуговування запиту існуючим потоком часто швидше за очікування при створенні нового потоку.
- 2. Пул потоків обмежує їх кількість, що важливо для систем, що не здатні підтримувати дуже багато потоків.
- 3. Відокремлення задачі на виконання від механіки створення потоку дозволяє використовувати різні стратегії запуску задачі.
  - Наприклад, задачу можна запланувати на виконання після затримки в часі або періодично виконувати.
- Кількість потоків у пулі визначається евристично на базі факторів, таких як кількість ЦП в системі, об’єм фізичної пам’яті, очікувана кількість клієнтських запитів тощо.
  - Розвинені thread-pool-архітектури дозволяють підлаштовувати динамічно кількість потоків у пулах залежно від usage patterns.

- Windows API пропонує кілька функцій, пов’язаних з пулами потоків.
  - Визначимо функцію, яка запускатиме окремий потік:

```
DWORD WINAPI PoolFunction(PVOID Param) {
    /* this function runs as a separate thread. */
}
```

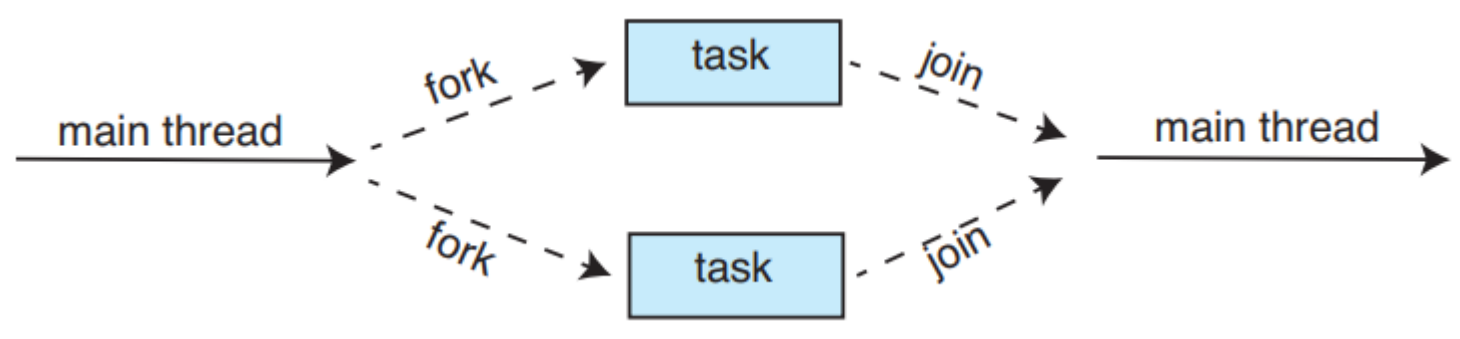
  - Вказівник на PoolFunction() передається в одну з функцій thread pool API, а потік з пулу виконує цю функцію.
- Такий член thread pool API – функція QueueUserWorkItem() з трьома параметрами:
  - LPTHREAD_START_ROUTINE Function—вказівник на функцію, яку потрібно запустити в окремому потоці.
  - PVOID Param—параметр, переданий у Function
  - ULONG Flags—прапорці, які вказують на те, як пул потоків створює та керує виконанням потоку
  - Приклад виклику: QueueUserWorkItem(&PoolFunction, NULL, 0);

Java Thread Pools (пакет java.util.concurrent)

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

public class ThreadPoolExample
{
public static void main(String[] args) {
   int numTasks = Integer.parseInt(args[0].trim());

   /* Create the thread pool */
   ExecutorService pool = Executors.newCachedThreadPool();

   /* Run each task using a thread in the pool */
   for (int i = 0; i < numTasks; i++)
      pool.execute(new Task());

   /* Shut down the pool once all threads have completed */
   pool.shutdown();
}
```

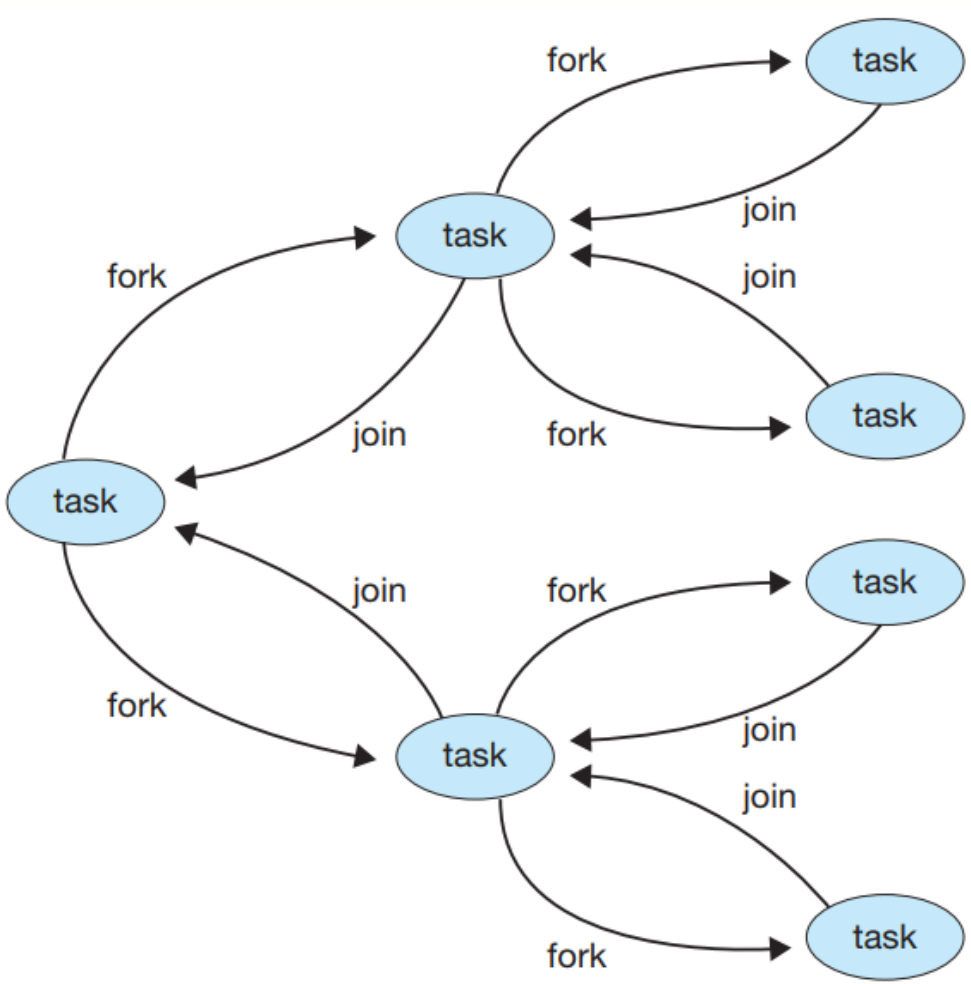
- Розглянемо 3 моделі:
  - 1. Однопоточний екзекутор—newSingleThreadExecutor()—створює пул на 1 потік.
  - 2. Фіксований екзекутор потоків—newFixedThreadPool(int size)—створює пул потоків із заданою їх кількістю.
  - 3. Кешований екзекутор потоків—newCachedThreadPool()—створює необмежений пул потоків, повторно використовуючи threads in many instances.
- Пул потоків створюється за допомогою одного з фабричних методів класу Executors:
  - static ExecutorService newSingleThreadExecutor()
  - static ExecutorService newFixedThreadPool(int size)
  - static ExecutorService newCachedThreadPool()
- Кожен з цих методів створює та повертає об’єкт, який реалізує інтерфейс ExecutorService.
  - ExecutorService розширяє інтерфейс Executor, дозволяючи викликати метод execute() для об’єкта та управляти завершенням роботи пулу потоків.

Fork Join



- Батьківський потік створює (fork) один або багато дочірніх потоків, а потім чекає на їх завершення і поєднується з ними, отримуючи та комбінуючи проміжні результати.
  - Синхронна модель часто характеризується як явне створення потоків, проте може використовуватись за основу неявного тредингу.
  - Тоді потоки будуть конструюватись не протягом fork-фази, а будуть роздаватись паралельні задачі.
  - Бібліотека керуватиме кількістю потоків для створення та призначатиме їм завдання.

Fork Join у Java



- У Java 7 представлено fork-join-бібліотеку, спроектовану для використання з рекурсивними divide-and-conquer алгоритмами на зразок швидкого сортування чи сортування злиттям.
  - Окремі задачі форкаються на етапі розподілу та призначаються для менших підзадач.
  - Алгоритм потрібно проектувати так, що окремі завдання можна було виконувати конкурентно.
- Загальний рекурсивний алгоритм поза fork-join-моделлю з Java наступний:

```
Task(problem)
   if problem is small enough
      solve the problem directly
   else
      subtask1 = fork(new Task(subset of problem)
      subtask2 = fork(new Task(subset of problem)

      result1 = join(subtask1)
      result2 = join(subtask2)

      return combined results
```

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

public class SumTask extends RecursiveTask<Integer>
{
   static final int THRESHOLD = 1000;

   private int begin;
   private int end;
   private int[] array;

   public SumTask(int begin, int end, int[] array) {
      this.begin = begin;
      this.end = end;
      this.array = array;
   }

   protected Integer compute() {
      if (end - begin < THRESHOLD) {
         int sum = 0;
         for (int i = begin; i <= end; i++)
            sum += array[i];

         return sum;
      }
      else {
         int mid = (begin + end) / 2;

         SumTask leftTask = new SumTask(begin, mid, array);
         SumTask rightTask = new SumTask(mid + 1, end, array);

         leftTask.fork();
         rightTask.fork();

         return rightTask.join() + leftTask.join();
      }
   }
}
```

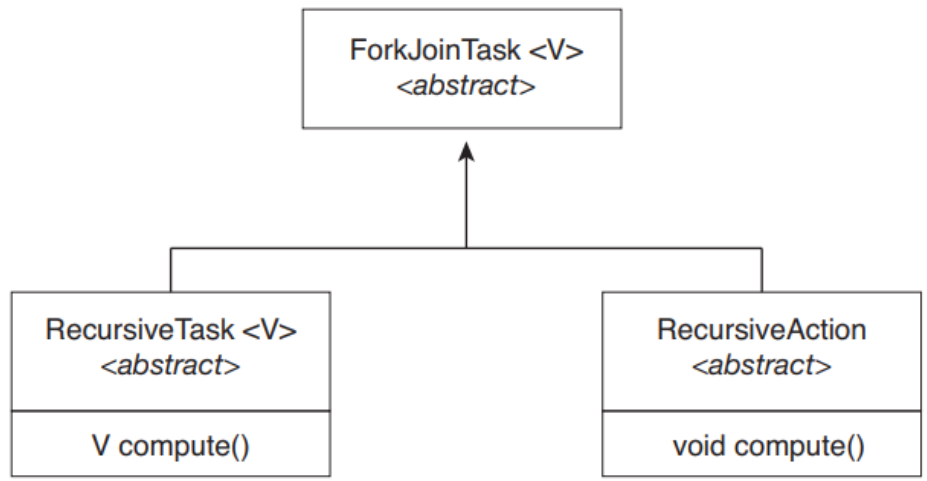
Ілюстрація стратегії Fork Join у Java

- Спроектуємо divide-and-conquer алгоритм, який додає елементи з масиву цілих чисел.
  - Починаючи з Java 7, введено новий пул потоків— ForkJoinPool, якому можна призначати задачі (успадковані від абстрактного базового класу ForkJoinTask, тут - SumTask).

```
ForkJoinPool pool = new ForkJoinPool();
// array contains the integers to be summed
int[] array = new int[SIZE];

SumTask task = new SumTask(0, SIZE - 1, array);
int sum = pool.invoke(task);
```

  - Після завершення початковий виклик invoke() повертає суму масиву.

Реалізація SumTask успадкована від RecursiveTask



- Фундаментальна відмінність: клас RecursiveTask повертає результат (через значення з compute()), а RecursiveAction – не повертає.
- Важливо визначати, коли підзадача стає “small enough”, щоб розв’язувати її напряму, без створення додаткових потоків.
  - У SumTask така умова – кількість елементів для додавання < THRESHOLD (1000).
  - На практиці треба виконувати часові заміри (trials) залежно від реалізації.
- Бібліотека конструює пул робочих потоків (worker threads) та збалансовує задачі серед доступних потоків-робітників.
  - Інколи існують тисячі задач, проте лише жменя потоків виконують роботу (наприклад, окремий потік для кожного ЦП).
  - Додатково кожен потік у ForkJoinPool підтримує чергу задач, які були ним форкнуті.
  - Якщо черга потоку порожня, він може «вкрасти» завдання з черги іншого потоку за допомогою work stealing algorithm, тим самим збалансовуючи навантаження між потоками.

OpenMP

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
   /* sequential code */

   #pragma omp parallel
   {
      printf("I am a parallel region.");
   }

   /* sequential code */

   return 0;
}
```

- Набір директив компілятора та API для програм мовами C/C++/ FORTRAN, який підтримує паралельне програмування у системах зі спільною пам’яттю.
  - OpenMP визначає паралельні регіони – блоки коду, які можна виконувати паралельно.
  - Розробники дописують директиви компілятора в паралельні регіони свого коду, що інструктує OpenMP runtime library виконувати ці регіони паралельно.
- Коли OpenMP зустрічає директиву #pragma omp parallel, вона створює стільки потоків, скільки ядер у системі.
  - Усі потоки паралельно виконують код регіону.
  - Після виходу з регіону потоки знищуються.

OpenMP постачає кілька додаткових директив

- Зокрема для паралельного виконання циклів.
  - Нехай існує 2 масиви, a та b, розміру N. Потрібно знайти суму їх відповідних елементів та зберегти її у масив c:

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
   c[i] = a[i] + b[i];
}
```

- OpenMP розбиває роботу всередині циклу між створеними потоками.
  - Додатково існують директиви розпаралелення, які дозволяють обирати різні рівні паралелізму.
  - Наприклад, задати кількість потоків вручну, визначати приватні для потоків дані тощо.

Grand Central Dispatch (GCD)

- Технологія Apple для ОС macOS та iOS.
  - Комбінує run-time library, API та language extensions, які дозволяють розробникам визначати секції коду (***tasks***) для паралельного запуску.
  - Як і OpenMP, GCD керує більшістю деталей реалізації тредингу.
  - GCD планує задачі для run-time виконання, розміщаючи їх у **dispatch queue**.
  - Коли задача видаляється з черги, вона приписується доступному потоку з пулу.
  - GCD ідентифікує 2 типи dispatch-черг: ***послідовні (serial)*** та ***багатопоточні (concurrent)***.
- Задачі з послідовної черги видаляються за FIFO-порядком.
  - Як тільки задача видаляється з черги, вона повинна завершити виконання, після чого інша задача буде взята з черги.
  - Кожен процес має власну послідовну чергу – ***головну (main queue)***, а розробники можуть створювати додаткові послідовні черги, локальні для конкретного процесу - ***private dispatch queues***.
  - Послідовні черги корисні для забезпечення послідовного виконання кількох задач (tasks).
- Задачі, розміщені в конкурентній черзі, також видаляються в порядку FIFO, проте кілька задач можуть видалятись за раз, а також паралельно виконуватись.

Існують кілька черг з конкурентим доступом на системному рівні (global dispatch queues)

- Вони діляться на 4 основних quality-of-service класи:
  - ***QOS_CLASS_USER_INTERACTIVE*— user-interactive** клас, який представляє задачі, що взаємодіють з користувачем (як UI та обробка подій). Завершення задачі, яка належить цьому класу, should require only a small amount of work.
  - ***QOS_CLASS_USER_INITIATED*—user-initiated** клас, подібний до попереднього в тому, що задачі пов'язані з респонсивним UI; проте user-initiated задачі можуть вимагати тривалішої обробки. Opening a file or a URL is a user-initiated task, for example. Tasks belonging to this class must be completed for the user to continue interacting with the system, but they do not need to be serviced as quickly as tasks in the user-interactive queue.
  - ***QOS_CLASS_UTILITY*** —The **utility** class represents tasks that require a longer time to complete but do not demand immediate results. This class includes work such as importing data.
  - ***QOS_CLASS_BACKGROUND*** —задачі належать **background-класу**, не видимі користувачеві та не time sensitive. Examples include indexing a mailbox system and performing backups.

- Спрямовані (submit) в dispatch queues задачі можуть виражатись 2 способами:
  - **1.** Для мов C/C++/Objective-C GCD ідентифікує розширення мови, яке називають ***блоком***. Це просто цілісна одиниця роботи, яка задається символом каретки (^) перед парою фігурних дужок:

```
^{ printf("I am a block"); }
```

  - **2.** Для мови Swift задача визначається за допомогою ***замикання (closure)***, подібного до блоку в тому, що воно виражає цілісну одиницю функціональності. Синтаксично замикання Swift записується так же, як і блок, проте без символу каретки.

```
let queue = dispatch_get_global_queue
   (QOS_CLASS_USER_INITIATED, 0)

dispatch_async(queue,{ print("I am a closure.") })
```

Проблеми тредингу. Системні виклики fork() та exec()

- Семантика системних викликів fork() та exec() змінюється в багатопоточних додатках.
- Якщо один потік викликає fork(),
  - чи новий процес дублюватиме всі потоки?
  - чи буде новий процес однопоточним?
- Деякі UNIX-системи мають 2 версії fork(): одна дублює всі потоки, інша – тільки потік, який форкнув.
  - Якщо exec() викликається негайно після форку, дублювання всіх потоків непотрібне, а задана програма замінить процес. Тоді доречне дублювання лише викликаючого потоку.
  - Якщо окремий процес не викликає exec() після форку, він має дублювати всі потоки.
  - Якщо потік викликає exec(), програма, визначена в параметрі exec(), замінить весь процес разом з усіма потоками.

Проблеми тредингу. Обробка сигналу

- Сигнал використовується в UNIX-системах для сповіщення процесу про трапляння деякої події.
  - Отримання сигналу може відбуватись синхронно або асинхронно, залежно від джерела та причини сигналювання від події.
- Усі сигнали (синхронні чи ні) слідують одному шаблону:
  - 1. Сигнал генерується настанням конкретної події.
  - 2. Сигнал доставляється процесу.
  - 3. Після доставки сигнал повинен оброблятись.
- Приклади ***синхронних сигналів***: illegal memory access та ділення на 0.
  - Синхронні сигнали доставляються тому ж процесу, що виконував операцію, від якої згенеровано сигнал.
- Коли сигнал згенеровано зовнішньою по відношенню до запущеного процесу подією, процес отримує ***асинхронний сигнал***.
  - Приклади: переривання процесу за допомогою Ctrl+C, завершення роботи таймеру тощо.
  - Зазвичай асинхронний сигнал надсилається іншому процесу.
- Сигнал може оброблятись (***handle)*** двома можливими обробниками:
  - **1. Обробником сигналу за умовчанням** (A default signal handler)
  - **2. Користувацьким обробником сигналу** (user-defined signal handler)

Проблеми тредингу. Обробка сигналу

- Кожен сигнал має ***обробник сигналу за умовчанням***, який ядро запускає при обробці сигналу.
  - Дію за умовчанням можна переозначити за допомогою ***користувацького (user-define) обробника сигналу***.
- Сигнали обробляються різними способами.
  - Деякі сигнали можуть ігноруватись, а інші (наприклад, illegal memory access) обробляються шляхом переривання програми.
  - Обробка сигналів в однопоточних програмах прямолінійна: сигнали завжди доставляються процесу.
  - Проте доставка сигналів набагато складніша в багатопоточних програмах: процес може мати кілька потоків.
  - Куди доставляти сигнал? Доступні наступні опції доставки:
    - **1.** Потоку, до якого сигнал застосовується.
    - **2.** Кожному потоку з процесу.
    - **3.** Деяким потокам з процесу.
    - **4.** Призначити окремий потік для отримання всіх сигналів для процесу.
- Метод доставки залежить від типу сигналу, який було згенеровано.
  - Наприклад, синхронні сигнали потребують своєї доставки тільки потоку, який спричинив сигнал.
  - Проте для асинхронних сигналів неясно: деякі з них (як Ctrl+C – переривання процесу) краще надсилати всім потокам.

- Стандартна UNIX-функція для доставки сигналу: **`kill(pid_t pid, int signal)`**
  - Вона задає процес (pid), до якого конкретний сигнал (signal) надходить.
  - Most multithreaded versions of UNIX allow a thread to specify which signals it will accept and which it will block.
  - Therefore, in some cases, an asynchronous signal may be delivered only to those threads that are not blocking it.
  - However, because signals need to be handled only once, a signal is typically delivered only to the first thread found that is not blocking it.
- Pthreads постачає функцію, яка дозволяє доставляти сигнал до заданого потоку (tid): **`pthread_kill(pthread_t tid, int signal)`**
- Хоч Windows явно не підтримує сигнали, вона дозволяє емулювати їх за допомогою ***асинхронних викликів процедур (asynchronous procedure calls, APCs)***.
  - Інфраструктура APC дозволяє користувацькому потоку задавати функцію, яка викликатиметься, коли the user thread отримує сповіщення про деяку подію.
  - whereas UNIX must contend with how to deal with signals in a multithreaded environment, the APC facility is more straightforward, since an APC is delivered to a particular thread rather than a process.

Threading Issues. Thread Cancellation

- **Thread cancellation** involves terminating a thread before it has completed.
  - For example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled.
  - Another situation might occur when a user presses a button on a web browser that stops a web page from loading any further.
  - Often, a web page loads using several threads—each image is loaded in a separate thread.
  - When a user presses the stop button on the browser, all threads loading the page are canceled.
- A thread that is to be canceled is often referred to as the **target thread**.
  - Cancellation of a target thread may occur in two different scenarios:
    - **1. Asynchronous cancellation**. One thread immediately terminates the target thread.
    - **2. Deferred cancellation**. The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.
- The difficulty with cancellation occurs in situations where resources have been allocated to a canceled thread or where a thread is canceled while in the midst of updating data it is sharing with other threads.
  - This becomes especially troublesome with asynchronous cancellation.
  - Often, the operating system will reclaim system resources from a canceled thread but will not reclaim all resources.
  - Therefore, canceling a thread asynchronously may not free a necessary system-wide resource.

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

  . . .

/* cancel the thread */
pthread_cancel(tid);

/* wait for the thread to terminate */
pthread_join(tid,NULL);
```

- With deferred cancellation, in contrast, one thread indicates that a target thread is to be canceled, but cancellation occurs only after the target thread has checked a flag to determine whether or not it should be canceled.
  - The thread can perform this check at a point at which it can be canceled safely.
  - In Pthreads, thread cancellation is initiated using the pthread_cancel() function.
  - The identifier of the target thread is passed as a parameter to the function.
  - The following code illustrates creating—and then canceling—a thread:
- Invoking pthread_cancel()indicates only a request to cancel the target thread, however; actual cancellation depends on how the target thread is set up to handle the request.
  - When the target thread is finally canceled, the call to pthread_join() in the canceling thread returns.
  - Pthreads supports three cancellation modes. Each mode is defined as a state and a type, as illustrated in the table below.
  - A thread may set its cancellation state and type using an API.

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- Pthreads дозволяє потокам відключити чи задіяти скасування.
  - Запити на скасування залишаються в очікуванні (pending), тому потік може пізніше задіяти скасування та respond to the request.
- За умовчанням тип відміни – deferred cancellation.
  - Проте відміна трапляється тільки тоді, коли потік досягає ***точки скасування (cancellation point)***.
  - Most of the blocking system calls in the POSIX and standard C library are defined as cancellation points, and these are listed when invoking the command man pthreads on a Linux system.
  - For example, the read() system call is a cancellation point that allows cancelling a thread that is blocked while awaiting input from read().
- One technique for establishing a cancellation point is to invoke the pthread_testcancel() function.
  - If a cancellation request is found to be pending, the call to pthread testcancel() will not return, and the thread will terminate; otherwise, the call to the function will return, and the thread will continue to run.
  - Additionally, Pthreads allows a function known as a **cleanup handler** to be invoked if a thread is canceled.
  - This function allows any resources a thread may have acquired to be released before the thread is terminated.

Код ілюструє, як потік може відповідати на запит відміни, використовуючи відкладене (deferred) скасування

```
while (1) {
   /* do some work for awhile */

    . . .

   /* check if there is a cancellation request */
   pthread_testcancel();
}
```

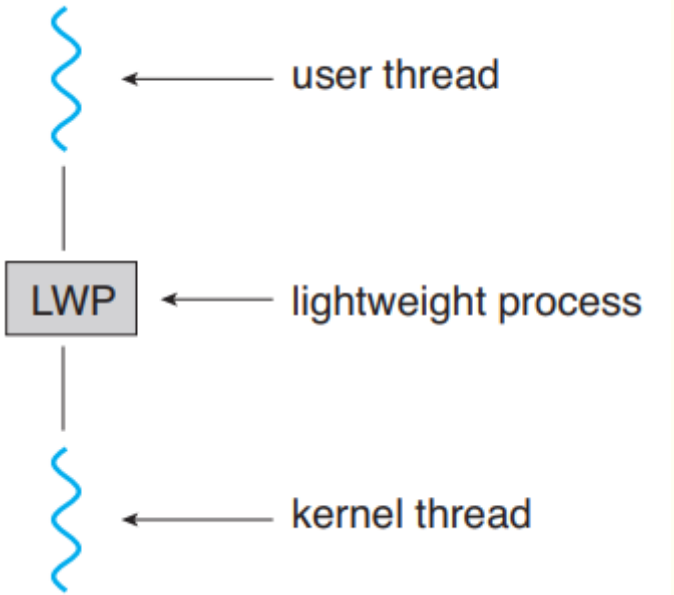
```
Thread worker;

 . . .

/* set the interruption status of the thread */
worker.interrupt()
```

- Через розглянуті проблеми асинхронне скасування не рекомендується в документації Pthreads.
- На Linux-системах скасування потоку за допомогою Pthreads API виконується з використанням сигналів.
  - У Java для цього застосовується аналогічна до відкладеного скасування політика: викликається метод interrupt(), який задає статус переривання цільового потоку як true.
  - Потік може перевірити цей статус, викликавши метод isInterrupted():
  - while (!Thread.currentThread().**isInterrupted**()) *{*
    . . .
    }

Локальне для потоку (Thread-Local) дані

- У деяких випадках кожному потоку може знадобитись власна копія певних даних.
  - Такі дані називають ***локальними для потоку (thread-local storage, TLS)***.
  - Наприклад, транзакційна система обробки може обслуговувати кожну транзакцію з унікальним ідентифікатором в окремому потоці. Для асоціювання потоку та ІД можна застосувати thread-local storage.
- TLS легко сплутати з локальними змінними.
  - Локальні змінні видимі лише протягом виклику однієї функції, а TLS-дані – для всіх функцій.
  - Також альтернативний підхід потрібний, коли розробник не контролює процес створення потоку. Наприклад, застосовуючи неявний підхід на зразок пулів потоків.
  - У дечому TLS подібні до статичних даних, проте TLS унікальні для кожного потоку.
- Більшість тредингових бібліотек підтримують TLS.
  - Java постачає клас ThreadLocal<T> з методами set() і get() для об’єктів ThreadLocal<T>.
  - Pthreads включає тип pthread_key_t, який забезпечує окремий для кожного потоку ключ, за яким можна отримати доступ до TLS.
  - У мові C# вимагається додавання storage-атрибуту [ThreadStatic], щоб оголосити локальні для потоку дані.
  - Компілятор gcc забезпечує storage class ключове слово для оголошення TLS-даних: static thread int threadID;

Активації планувальника (Scheduler Activations)



- Проблема взаємодії між ядром та тредінговою бібліотекою, пов'язана з моделлю M:M та дворівневою моделлю.
  - Така координація дозволяє багатьом kernel-потокам динамічно підлаштовуватись для кращої продуктивності.
- Багато систем реалізують ці моделі, розміщуючи проміжну структуру даних між user-потоками та kernel-потоками.
  - Таку структуру даних часто називають ***легковаговим процесом (lightweight process, LWP)***.
  - Для user-thread-бібліотеки LWP буде віртуальним процесором, на якому додаток може спланувати запуск user-потоку.
  - Кожний LWP прикріпляється до kernel-потоку, а ОС планує виконання саме kernel-потоків на фізичних процесорах.
  - Якщо kernel-потік блокує (зокрема, очікуючи на завершення вводу-виводу), LWP також блокується, а звідси, й відповідний потік рівня користувача.

Додаток може потребувати довільної кількості LWP для ефективної роботи

- Розглянемо CPU-bound додаток, запущений на одному процесорі.
  - Тоді лише 1 потік може працювати за раз, і одного LWP достатньо.
  - Додаток з інтенсивним вводом-виводом може потребувати кількох LWP для виконання.
  - Typically, an LWP is required for each concurrent blocking system call.
- Нехай 5 різних запитів на зчитування файлу відбувається одночасно.
  - Потрібні 5 LWP, оскільки all could be waiting for I/O completion in the kernel.
  - Якщо процес має лише 4 LWP, п’ятий запит повинен wait for one of the LWPs to return from the kernel.
- Одна з схем взаємодії між user-thread library та ядром називають **scheduler activation**.
  - It works as follows: The kernel provides an application with a set of virtual processors (LWPs), and the application can schedule user threads onto an available virtual processor.
  - Furthermore, the kernel must inform an application about certain events.
  - This procedure is known as an **upcall**. Upcalls are handled by the thread library with an **upcall handler**, and upcall handlers must run on a virtual processor.

- Подія, що запускає upcall, трапляється перед блокуванням потоку.
  - In this scenario, the kernel makes an upcall to the application informing it that a thread is about to block and identifying the specific thread.
  - The kernel then allocates a new virtual processor to the application.
  - The application runs an upcall handler on this new virtual processor, which saves the state of the blocking thread and relinquishes the virtual processor on which the blocking thread is running.
  - The upcall handler then schedules another thread that is eligible to run on the new virtual processor.
  - When the event that the blocking thread was waiting for occurs, the kernel makes another upcall to the thread library informing it that the previously blocked thread is now eligible to run.
  - The upcall handler for this event also requires a virtual processor, and the kernel may allocate a new virtual processor or preempt one of the user threads and run the upcall handler on its virtual processor.
  - After marking the unblocked thread as eligible to run, the application schedules an eligible thread to run on an available virtual processor.

ДЯКУЮ ЗА УВАГУ!

Наступне питання: принципи роботи планувальника ЦП

Intel Thread Building Blocks

- Intel threading building blocks (TBB) is a template library that supports designing parallel applications in C++.
  - As this is a library, it requires no special compiler or language support. Developers specify tasks that can run in parallel, and the TBB task scheduler maps these tasks onto underlying threads.
  - Furthermore, the task scheduler provides load balancing and is cache aware, meaning that it will give precedence to tasks that likely have their data stored in cache memory and thus will execute more quickly.
  - TBB provides a rich set of features, including templates for parallel loop structures, atomic operations, and mutual exclusion locking. In addition, it provides concurrent data structures, including a hash map, queue, and vector, which can serve as equivalent thread-safe versions of the C++ standard template library data structures.
- Let's use parallel for loops as an example. Initially, assume there is a function named apply(float value) that performs an operation on the parameter value.
  - If we had an array v of size n containing float values, we could use the following serial for loop to pass each value in v to the apply() function:

```
for (int i = 0; i < n; i++) {
   apply(v[i]);
}
```

- Alternatively, a developer could use TBB, which provides a parallel for template that expects two values:
  parallel for (***range body***)
  where ***range*** refers to the range of elements that will be iterated (known as the **iteration space**) and ***body*** specifies an operation that will be performed on a subrange of elements.
  We can now rewrite the above serial for loop using the TBB parallel for template as follows:
  parallel for (size t(0), n, [=](size t i) {apply(v[i]);});
  The first two parameters specify that the iteration space is from 0 to *n*-1 (which corresponds to the number of elements in the array v). The second parameter is a C++ lambda function that requires a bit of explanation. The expression [=](size t i) is the parameter i, which assumes each of the values over the iteration space (in this case from 0 to `n` - 1). Each value of i is used to identify which array element in v is to be passed as a parameter to the apply(v[i]) function.

- The TBB library will divide the loop iterations into separate “chunks” and create a number of tasks that operate on those chunks.
  - (The parallel for function allows developers to manually specify the size of the chunks if they wish to.) TBB will also create a number of threads and assign tasks to available threads. This is quite similar to the fork-join library in Java. The advantage of this approach is that it requires only that developers identify what operations can run in parallel (by specifying a parallel for loop), and the library man ages the details involved in dividing the work into separate tasks that run in parallel. Intel TBB has both commercial and open-source versions that run on Windows, Linux, and macOS. Refer to the bibliography for further details on how to develop parallel applications using TBB.