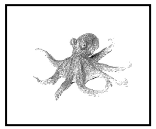

РОЗДІЛ 2.
Процеси

У цьому розділі розглянуто концепцію процесу, стани процесу й операції над процесами. Наведено приклади реалізацій процесів та операцій над ними, що застосовані в мовах програмування та бібліотеках.

2.1. Поняття процесу

Процес – абстрактне поняття, що включає опис визначених дій, пов'язаних з виконанням програми в комп'ютерній системі [9; 35; 45; 48]. При цьому процес оформлюється так, щоб система керування процесами в ОС могла ефективно перерозподіляти ресурси системи (процесори, пам'ять, прилади введення-виведення, файли і т.ін.). Процес характеризується власним набором ресурсів і виділеною для нього ділянкою оперативної пам'яті.

Поняття процесу вперше з'явилось в багатозадачних операційних системах і сьогодні є фундаментальним для кожної сучасної ОС. В ОС процес був пов'язаний з кожною множиною прикладних або системних програм, які виконуються в комп'ютерній системі. Це дозволяє системі керування процесами ОС, яка розміщується в ядрі ОС, ефективно маніпулювати процесами за допомогою спеціального Блоку Керування Процесом (БКП або PCB – Process Control

Block) [9]. Блок Керування Процесом – динамічна структура даних, яка містить основну інформацію про процес:

- ім'я процесу;
- поточний стан процесу;
- пріоритет процесу;
- місце розміщення процесу в пам'яті;
- ресурси, що пов'язані з процесом та ін.

Стосовно паралельного програмування процеси – це частини однієї програми користувача, які виконуються одночасно. Такі процеси отримали назву *легкі процеси* (*lightweight processes*). Для позначення легких процесів використовують також терміни *потік* (Java, C, C#) або *задача* (Ада, MPI). Тому традиційні процеси іноді називають *важкими процесами* (*heavyweight processes*).

Під час виконання кількох програм у комп'ютерній системі або звичайному комп'ютері здійснюється постійне переключення з одного процесу на інший. Теж саме відбувається і для одиночної паралельної програми, якщо вона включає кілька легких процесів (*потоків*). Але час, який витрачається на переключення для легких процесів, менший, ніж для тяжких. Ще одна відмінність важких та легких процесів полягає в тому, що легкі процеси постійно взаємодіють, оскільки є частинами однієї програми (*тісно зв'язані* процеси), в той час, як важки процеси взаємодіють рідко (*слабо зв'язані* процеси).

2.2. Стан процесу. Керування процесами

Керування процесом включає виконання деяких операцій над ним, дозволяючи процесу пройти визначені стани (рис. 2.1). Види станів процесу визначаються конкретною ОС і системою керування процесами, але, змістять здебільше такі стани:

Породження – створення процессу і підготовка до першого виконання в системі;
Готовності – процес готовий до виконання і чекає звільнення процесора;
Виконання – процес виконується на процесорі;
Блокування – процес призупинений через очікування

визначеної події: завершення введення даних, завершення заданого часу очікування, сигналу від іншого процесу, звільнення ресурсу та інше;

Завершення – нормальне або аварійне завершення виконання процесу.

Причинами переходів є операції, що виконуються ОС над процесом: породження і завершення процесу, блокування і розблокування, завершення кванта часу роботи процесора, операції уведення-виведення та ін.

Процес також може мати особливий стан, який отримав назву *тупик* (deadlock). Процес у тупиковому стані блокований і очікує на подію, яка ніколи не відбудеться. Це зумовлює неможливість його продовження і, як наслідок, – зависання програми в цілому. Тупики – одна з головних проблем, що виникають під час виконання паралельних програм. Боротьба з тупиками зводиться до таких дій:

- відвернення тупиків;
- запобігання тупиків;
- визначення тупика;
- ліквідація тупика.

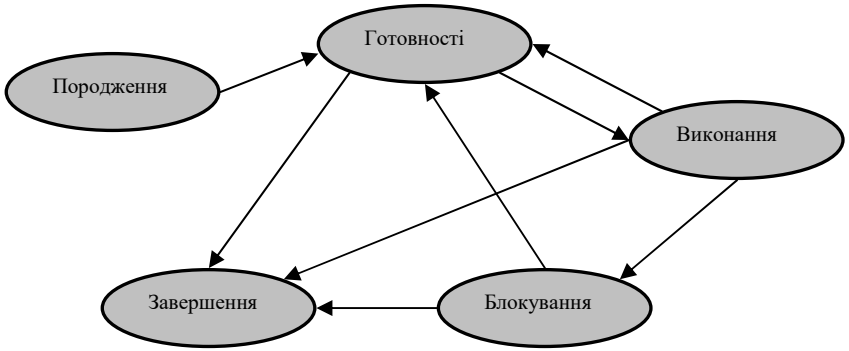

Рис. 2.1. Діаграма станів процесу

У цілому тупики є серйозною загрозою і боротьбі з ними слід приділяти особливу увагу під час розроблення та налагодження паралельної програми. Приклади і методи запобігання тупикам будуть розглянуті в розділах, у яких вивчається взаємодії процесів.

Засоби роботи з процесами розподіляються на бібліотечні та мовні. Прикладами перших є засоби бібліотек WinAPI, PVM, MPI, OpenMP, POSIX, Pthread, які реалізовані у вигляді набору функцій. Вони дозволяють роботу з процесами в будь-яких мовах програмування, зокрема і тих, які не мають вбудованих засобів роботи з процесами. Пізніші мови програмування, такі як C#, Java або Ада, безпосередньо мають засоби роботи з процесами, які вбудовані в мови, що забезпечує більш ефективне їх виконання та взаємодію.

Незалежно від реалізації (бібліотечної або мовної) засоби роботи з процесами мають забезпечити розробнику паралельного додатка такі можливості:

- об’явити процес (групу процесів);
- установити пріоритет процесу;
- запустити процес на виконання,
- призупинити процес на визначений час;
- блокувати та розблокувати процес;
- організувати взаємодію з іншими процесами (синхронізація або передування даних);
- завершити процес.

2.3. Процеси в мові Java

Процес в мові Java реалізований у вигляді *потоку* (`thread`) [26; 33; 45]. Java – об’єктно-орієнтована мова програмування, тому класи-потоки створюються використанням класу `Thread`. Можливі два підходи до створення потоку:

- за допомогою наслідування класу `Thread;`
- з використанням інтерфейсу `Runnable.`

Використання класу Thread. Клас `Thread` інкапсулює потік і містить в собі набір методів для керування потоком (табл. 2.1). Для створення потоку потрібно оголосити підклас через розширення

класу Thread і створити екземпляр цього підкласу. Підклас має перевизначити метод run(), що є точкою входу в потік. Необхідно також викликати метод start(), щоб почати виконання потоку.

Клас Thread має сім конструкторів. Простий конструктор не має параметрів, а інші конструктори дозволяють задавати як параметри інтерфейс Runnable, ім'я потоку або ім'я групи потоків.

Таблиця 2.1. Методи класу Thread

Метод	Дія
getName()	Отримати ім'я потоку
getPriority()	Отримати пріоритет потоку
isAlive()	Визначити, чи виконується потік
join()	Чекати завершення потоку
run()	Указати точку входу в потік
sleep()	Призупинити потік на заданий інтервал часу
start()	Запустити потік на виконання

❖ **Приклад 2.1**

```
/* ----------------------------------------------------
-- Java. Створення потоків Nord і West шляхом        --
--       розширення класу  Thread                    --
----------------------------------------------------*/
   class  Nord   extends   Thread {

      // перевизначення методу  run()
      public void run(){
         System.out.println("Process Nord ");
      }
   }// Nord

   class  West extends Thread {

      public void run(){
         System.out.println("Process West ");
      }
   }// West

   // головний потік
   class  MainThread {

      // точка входу в основний клас
      public static void main(String  args []){
```



```
      // оголошення екземплярів  потоків
      Nord N = new  Nord();
      West W = new  West();

      // запуск потоків
      N.start();
      W.start();

      // виконання головного потоку
      System.out.println("Process MainThread
                                        finished ");

   }// main
}// MainThread
```

Керувати порядком запуску потоків можна за допомогою пріоритету потоку. Для встановлення пріоритету потоку використовують метод setPriority() з класу Thread:

```
final void setPriority (int  Рівень);
```

де параметр Рівень визначає пріоритет потоку. Пріоритет вибирається з діапазону MIN_PRIORITY і MAX_PRIORITY, граничні значення якого відповідно дорівнюють 1 і 10. За умовчанням потік отримує пріоритет NORM_PRIORITY, що дорівнює 5.

Призупинити потік на зазначений відрізок часу можна за допомогою методу sleep():

```
static void sleep(long Час) throws
                              InterruptedException;
```

де параметр Час задає час затримки потоку в мілісекундах. Метод може збуджувати виключення InterruptedException, що потребує обов'язкового створення блоків try/catch під час його використання.

Метод yield() викликає призупинення поточного потоку, після чого з черги готових до виконання потоків вибирається і запускається потік з більшим або однаковим пріоритетом:

```
final void yield();
```

Якщо в черзі готових до виконання немає потоків з таким пріоритетом, то потік продовжує своє виконання. Метод `yield()` дозволяє організувати більш "справедливе" виконання потоків.

Метод `join()` дозволяє організовувати очікування завершення викликаного потоку. Можливе використання в головному потоці, якщо потрібно, щоб він був завершений останнім – після завершення усіх запущених ним потоків:

```
final void join()
              throws InterruptedException;
```

Використання інтерфейсу Runnable. Потік можна створити також, визначивши клас, який реалізує інтерфейс `Runnable`. В інтерфейсі `Runnable` описано абстрактний метод `run()`, який необхідно визначити в створюваному класі - потоці:

```
public void run();
```

У методі `run()` слід описати код, який визначає дії створюваного потоку. Далі під час створення класу оголошується об'єкт типу `Thread`. При цьому можна використати конструктори, визначені в класі `Thread`. Після оголошення потокового об'єкта необхідно викликати його метод `start()`, який, в свою чергу, знаходить метод `run()`, визначений в об'єкті, і передає йому керування.

❖ Приклад 2.2

```
/* ------------------------------------------------------
-- Java. Створення потоків використання                --
-- інтерфейсу    Runnable                               --
------------------------------------------------------*/
   class Denon implements Runnable{

      String  Name;
      Thread  t;

      // Конструктор класу Denon
      Denon(String ім'я){
         Name = ім'я;
         t  = new  Thread(this, N);
         System.out.println("   New thread" + Name);
         t.start();
      }

      // точка входу в потік
      public void run(){

         try{
            for (int i=1; i<5; i++){
               System.out.println("Start of process "
                                                + Name);
               Thread.sleep(500);
            }
         }
         catch(InterruptedException e){
            System.out.println("Error in process");
         }
         System.out.println(Finish of process "
                                             + Name);
     }
   }// Denon
   // Головний потік, що використає клас Denon
   public class  Titan {

      // точка входу в основний клас
      public static void main(String  args[]){
         System.out.println("Process Titan started");

         // оголошення екземплярів класу Denon
         Denon A = new Denon("A");
         Denon B = new Denon("B");

         // чекати завершення потоку A і B
         try{
            A.t.join();
            B.t.join();
         }
         catch(InterruptedException e){
            System.out.println("Error in Titan
                                          process");
         }
         System.out.println("Process Titan finished ");

      }// main
   }// Titan
```

2.4. Процеси в мові Ада

Ада – відома мова програмування, яку було розроблено в 1983 році на замовлення Міністерства оборони США [16; 58]. Призначення мови – розроблення великих програмних систем, робота яких характеризується високою надійністю. Ада – одна з перших мов програмування, які мають вбудовані засоби роботи з процесами.

Процеси в мові Ада реалізовані у вигляді спеціальних модулів – *задач* (`task`). Задачний модуль `task` один із п'яти видів модулів мови, має стандартну форму у вигляді специфікації і тіла. Специфікація визначає інтерфейс задачі, де задається ім'я задачі, а також у разі необхідності, – пріоритет задачі, засоби взаємодії з іншими задачами, місце розташування в пам'яті. Простіший вид специфікації задачі містить тільки ім'я задачі. Така специфікація має назву *виродженої*. Тіло задачі визначає дії задачі під час виконання.

Задачний модуль не є одиницею компіляції в мові і тому задача має бути описаною в підпрограмі або пакеті. Мова немає явних засобів запуску задачі. Задача *автоматично* стартує під час запуску підпрограми або входячи в блок, у яких її описано. Якщо задачі розміщені в головній програмі, то при запуску основної програми задачі починають виконуватися одночасно з головною програмою, яка, в свою чергу, завжди розглядається як задача і виконується паралельно з вкладеними в ній задачами. Явний запуск задач можливо реалізувати шляхом розміщення їх в окремій процедурі, виклик якої в потрібний момент спричинить запуск вкладених у ній задач. Крім того, запуск задач можливо реалізувати за допомогою посилального (`access`) типу та генератора `new`.

❖ Приклад 2.3

```
-------------------------------------------------------
--  Ада95.Оголошення та запуск задач                 --
-------------------------------------------------------
procedure Lab23 is

   -- специфікації задач A і B (вироджені)
   task A;
   task B;

   -- тіла задач A і B
   task body A is
   begin
      put_line("Process A started");
   end A;

   task body A is
   begin
      put_line("Process B started");
   end B;

-- основна програма
begin
      -- місце автоматичного запуску задач A і B
   put_line("Main procedure started ");
end Lab23;
```


Задачний тип (`task type`) дозволяє користувачу описати тип, об'єктами якого будуть задачі. Задачі, які створені з використанням задачного типу ідентичні. Однак реалізація задачного типу за допомогою *дискримінанта* дозволяє внести необхідні особливості в кожну задачу для її ініціалізації. Механізм дискримінантів схожий на конструктори класів у мові Java.

❖ Приклад 2.4

```
-------------------------------------------------------
--   Ада95.Задачний тип з дискримінантом             --
-------------------------------------------------------
procedure Lab24 is

   -- задачний тип з дискримінантом
   task type  ЗадачнТип(Номер : integer);

   -- тіло задачного типу
   task body ЗадачнТип   is
      Номер_Задачі :  integer:= Номер;
   begin
      put(" Process  started  ");
      put(Номер_Задачі);
   end ЗадачнТип;

   A: ЗадачнТип(1);  -- створення задач
   B: ЗадачнТип(2);
begin
   null;              -- порожній оператор
end Lab24;
```

Пріоритет процесу задається в специфікації задачі за допомогою прагми `Priority`. Пріоритет задачі – ціле число в діапазоні 1 – 7. Більше значення відповідає більшій пріоритетності задачі. Пріоритет визначає спроможність задачі виборювати ресурси – процесор, прилади введення–виведення, а також під час взаємодії з іншими задачами або захищеними модулями. Тобто пріоритети працюють там, де з'являються черги, у яких процеси можуть знаходитись при очікуванні. Формування таких черг та вибірка з них здійснюються з використанням пріоритетів задач.

Призупинення задачі на вказаний відрізок часу в мові виконується за допомогою оператора `delay`, у якому час очікування указується в секундах (тип `Duration` з системного пакета `Calendar`). Під час виконанні оператора `delay` задача блокується і керування передається іншій задачі, яка знаходиться у стані готовності. Оператор `delay until T` блокує задачу до вказаного часу `T`.

❖ Приклад 2.5

```
------------------------------------------------------------
--Ада95.Використання пріоритетів і оператора delay      --
------------------------------------------------------------
procedure Lab25 is

   -- специфікації задач A і B
   task A is
      pragma Priority(4);   -- пріоритет задачі A
   end  A;
   task B is
      pragma Priority(5);   -- пріоритет задачі B
   end B;

   -- тіло задач A і B
   task body A is
   begin
      put_line("Process A started");
      delay(4.5);    -- затримка задачі A на 4,5 с
      put_line("Process A finished");
   end A;
   -------------
   task body B is
   begin
      put_line("     Process B started");
      delay(7.4);    -- затримка задачі B на 7,4 с

      put_line("     Process B finished");
   end B;

-- основна програма
begin

   put_line("  Main procedure started ");

   delay(12.25);          -- затримка на 12,25 с

   put_line(" Main procedure finished  ");

end Lab25;
```


Порівняльний аналіз засобів мов Java та Ада95 для роботи з процесами наведено в праці [13].

2.5. Процеси в бібліотеках MPI та PVM

Бібліотеки MPI та PVM – найбільш розповсюдженими засобами програмування для комп'ютерних систем з розподіленою пам'яттю та розподілених комп'ютерних систем [2; 25; 42].

Бібліотека (інтерфейс) MPI (Message Passing Interface) містить набір функцій, що дозволять організувати роботу з процесами в мовах, які не мають вбудованих засобів програмування процесів. У разі застосування бібліотеки MPI використовується мова послідовного програмування типу C або Фортран, процеси в якій програмуються за допомогою засобів інтерфейсу MPI.

Усі ресурси MPI бібліотеки (функції, типи, константи) розпочинаються з префікса `MPI_`.

Загальна структура MPI програми має вигляд:

```
#include "mpi.h"

/* Описання */
int main(int args, char * argv []){

  /* Локальні описання */

  MPI_Init(&args, &argv);
```

```
  /* Тіло програми */

  MPI_Finalize();
return 0;
}
```

Основні функції MPI бібліотеки, потрібні для створення процесів, наведені в таб. 2.2:

Таблиця 2.2. Основні функції MPI

Функції	Дії
`MPI_Init()`	Підключити до MPI (параметри `args` і `argv` визначають аргументи програми). Завжди викликається першою
`MPI_Comm_size()`	Визначити кількість процесів, що необхідно запустити
`MPI_Comm_rank()`	Отримати ранг (номер) процесу в групі (в комунікаторі)
`MPI_Finalize()`	Завершити виконання програми

Процес у MPI має назву *задача*. Задачі в MPI можуть бути об'єднані в іменовану групу. Об'єкт - група дозволяє звертатися до групи як єдиному цілому (наприклад, відправляти повідомлення всім задачам групи), а також визначати дії, які виконуються тільки членами групи.

Функція

```
MPI_Group(MPI_Comm_World, &size);
```

створює групу.

У рамках групи всі задачі мають унікальні ідентифікатори (*ранки*) – впорядковані числа, які розпочинаються з нуля. Спочатку всі задачі належать до однієї базової групи, з якої потім формуються нові групи. MPI надає набір функцій для роботи з групами.

Створюючи в додатку задачі, їх прив'язують до спільної ділянки зв'язку. Для цього використовують поняття *комунікатор* – описник ділянки зв'язку процесів, які об'єднані в групу, а також – для різних груп. Програма може вміщувати кілька ділянок зв'язку.

Нумерація процесів всередині ділянки зв'язку незалежна. Комунікатор може бути використаний як параметр MPI-функції і для обмеження сфери її дії тільки заданою ділянкою зв'язку. Крім того, комунікатор забезпечує вимоги безпеки. MPI автоматично створює комунікатор `MPI_Comm_World`, який є базовим для кожного додатка і створюється автоматично під час виклику функції `MPI_Init()`.

Функції

```
MPI_Comm_size(MPI_Comm_World, &size);
MPI_Comm_rank(MPI_Comm_World, &rank);
```

для комунікатора `MPI_Comm_World` повертають значення: `size` – розміру групи (кількість задач, що приєднані до ділянки зв'язку) і `rank` – порядковий номер задачі, яка викликає цю функцію.

❖ Приклад 2.6

Створення процесів, кожний з яких виводить на дисплей повідомлення про старт процесу.

```
/*----------------------------------------------------------
-- MPI.Створення процесів                                 --
----------------------------------------------------------*/
   #include <stdio.h>
   #include <string.h>
   #include <mpi.h>

   /* Визначення розміру та буфера повідомлень  */
   #define розмір_буфера 256

   int main(int args, char *argv[]){

      int кількість_процесів;
      int мій_ранг;
      int ранг_відправника;
      int ранг_отримувача;
      int тег_повідомлення;

      MPI_Status статус;
      char буфер[розмір_буфера];

      /* Початок  роботи MPI */
      MPI_Init(&args, &argv);
```

```
/* Отримати номер поточного процесу в групі */
MPI_Comm_rank(MPI_COMM_WORLD, &мій_ранг);

/* Отримати загальну кількість процесів,
   що старували */
MPI_Comm_size(MPI_COMM_WORLD,
              &кількість_процесів);

/* Виведення повідомлення
   на дисплей */
if (мій_ранг != 0){
   sprintf("Start of process 0");
}
else{
   /* вивести на дисплей повідомлення з
      інших процесів */
   prinf("%s\n", мій_ранг);
    }
}
/* Завершення роботи  MPI */

MPI_Finalize();
return 0;

}/* main */
```

Бібліотека PVM (Parallel Virtual Machine) була розроблена в університеті штату Тенесі, США [25]. Дозволяє розробляти додатки для множини гетерогенних комп'ютерів, з'єднаних мережею, розглядаючи їх як одну велику паралельну комп'ютерну систему. Система містить великий набір засобів для керування процесами і ресурсами, пуску і завершення задач, засоби синхронізації задач, конфігурування PVM. Підтримується мовами Фортран, C, C++ .

Підтримка різних гетерогенних комп'ютерів – основна особливість PVM. Програма, що написана для однієї архітектури, може виконуватися на іншій без модифікації. PVM додаток – це набір взаємодіючих задач (процесів), які забезпечують розв'язання однієї задачі користувача. Модель обчислень в PVM – набір взаємодіючих послідовних задач, кожна задача має особисту нить керування, взаємодія задач виконується за допомогою посилці повідомлень.

У PVM вузол системи з комп'ютерам має назву *хост*. PVM має дві компоненти: PVM бібліотеку і демон, який має розміщуватися

в кожному хості PVM. Для запуску PVM додатку необхідно запустити PVM і сконфігурувати віртуальну машину.

PVM додаток – композиція послідовних програм, кожна з яких пов'язана з одним або кількома процесами в паралельній програмі. Ці програми компілюються індивідуально для кожного хосту PVM. Одна із задач (що ініціалізує) запускається вручну й активізує інші задачі додатка. PVM додаток може мати кілька структур: зірка, майстер-робітник та ін.

Задача в PVM запускається або вручну, або це виконується з іншої задачі. Динамічне створення задачі виконується за допомогою PVM функції `pvm_spawn()`. Задача, що викликає функцію `pvm_spawn()`, є батьковскою задачею, створювана задача – дочірньою задачею. Створюючі дочірню задачу потрібно вказати:

- машину, на якій буде виконуватися дочірня задача;
- шлях (`path`) до виконавчого файлу;
- кількість копій дочірньої задачі;
- масив аргументів.

Задачі в PVM мають унікальний ідентифікатор (`Tid`), який отримують під час створення задачі. Ідентифікатор використовують для взаємодії задач.

Формат функції `pvm_spawn()`:

```
num = pvm_spawn(Child, Args, Flag,
                       Where, HowMany,&Tids);
```

Параметри:

- `Child` - ім'я exec файлу, котрий визначає виконання створюваної задачі. Має резидентно знаходитись на хості, де буде виконуватися.
- `Args` - покажчик на масив аргументів задачі. Має значення `NULL`, якщо аргументів немає.
- `Flag` - застосовують для завдання режимів виконання процесів, що запускаються.

- `Where` – ім'я хосту або типу архітектури, де буде виконуватися задача, що створена залежно від значення змінної `Flag`; тобто визначається місце запуску процесу.
- `HowMany` - кількість ідентичних дочірніх задач для запуску.
- `Tids` - ідентифікатор (TID) дочірньої задачі.

Функція `pvm_spawn()` запускає копії виконуваного файлу `Child`.

Приклади створення процесів:

```
n1 = pvm_Spawn("user/rew/work", 0, 1, "Dina",
                                          2, &tid1);
n2 = pvm_Spawn("user/rew/work", 0, 1,
                                  "Zond", 4, &tid2);
```

Дві створені задачі відповідають двом і чотирьом копіями програми `"work"` для двох хостів – `Dina` і `Zond`. Обидві машини мають містити файл `work.exe` в директоріях `"user/rew/"`. Значення `n1` і `n2` – номери задач, TID задачі будуть розміщені в `tid1` і `tid2`.

2.6. Процеси в бібліотеці WinAPI

Бібліотека WinAPI (Windows API) входить до складу ОС Windows і містить набір функцій, які призначені для роботи з процесами.

Для створення потоку в Win32 використовують функції `CreateThread()` і `CreateRemoteThread()`. Дані функції повертають ідентифікатор процесу, який є унікальним і ідентифікує його в системі. Під час створення потоку визначається початковий адрес коду, з якого має виконуватися потік. Зазвичай, це назва функції, яка буде виконуватися як процес.

Функція

```
HANDLE Ім'я_Потоку = CreateThread(
   LPSECURITY_ATTRIBUTES  атр,  // атрибут безпеки
   SIZE_T    рс,               // розмір стеку
```



```
   LPTHREAD_STERT_ROUTUNE фп,   // функція потоку
   LPVOID  афт,         // аргумент функції потоку
   DWORD   пр,          // прапорець
   LPDWORD іп);         // ідентифікатор потоку
```

створює потік, для якого фактичні параметри визначають ім'я функції та її параметр, атрибути безпеки, початковий розмір стека потоку, прапорець створення.

❖ **Приклад 2.7**

Створення двох потоків, які будуть виконувати функцію `Task_Func`:

```
void Task(void);

int main(void){

   DWORD TidA, TidB;
   HANDLE hThreadA, hThreadB;

   // Створення потоків
   hThreadA = CreateThread(NULL, 0,
        (LPTHREAD_START_ROUTINE)Task_Func,
                                   NULL,0, &TidA);
   hThreadB = CreateThread(NULL, 0,
        (LPTHREAD_START_ROUTINE)Task_Func,
                                  NULL,0, &TidB);

   // Закриття потоків
   CloseHandle(hThreadA);
   CloseHandle(hThreadB);
}

// функція, яка виконується в потоці
void Task_Func(void){
    printf("Потік стартував ");
        .  .  .
    printf("Потік завершився ");
}
```

Потік виконується доти, доки не відбудеться одна з таких подій:

- функція повертає значення потоку;
- потік викликає функцію `ExitThread();`

- інший потік викликає функцію `ExitProcess();`
- інший потік викликає функцію `TerminateThread()` з дескриптором потоку;
- інший потік викликає функцію `TerminateProcess()` з дескриптором процесу.

За допомогою функції `GetExitCodeThread()` можна отримати значення стану завершення потоку. Під час виконання потік має стан `STILL_ACTIVE`.

Пріоритет потоку встановлюється за допомогою функції `SetThreadPriority()`. Отримати поточне значення пріоритету процесу можна за допомогою функції `GetThreadPriority()`.

Створюючись, потік отримує пріоритет, що дорівнює значенню `TRHREAD_PRIORITY_NORMAL.`

Потік залишається в системі, поки він не закінчить роботу і всі його дескриптори не будуть закритими за допомогою функції `CloseHandle().`

Призупинення процесу виконується функціями `SuspendThread()` і `ResumeThread()`. Функції `Sleep()` і `SleepEx()` блокують виконання процесу на заданий інтервал часу. Функція `SwitchToThread()` блокує процес і передає керування іншому потоку.

2.7. Паралельні алгоритми

2.7.1. Аналіз паралельного алгоритму

Ефективна реалізація будь-якої задачі в паралельній системі можлива тільки тоді, коли алгоритм її розв'язування поданий в паралельній формі, тобто буде розроблений паралельний алгоритм. З погляду паралельних властивостей (внутрішнього паралелізму) усі задачі можна поділити на три групи:

1) повністю паралельні задачі;
2) частково паралельні задачі;
3) задачі, які не допускають паралельної обробки.

Прикладом повністю паралельної задачі є операція додавання матриць, яку можна виконати за один крок, на якому здійснюється формування кожного елемента матриці.

Частково паралельною є задача, у якій чергуються паралельні та послідовні частини. Це найбільш поширений тип задач з погляду побудови паралельних алгоритмів. Приклад такої задачі - операція скалярного множення векторів.

Існують також задачі, для яких неможливо побудувати паралельний алгоритм. Наприклад, операція $a = b + c + d$ не може бути подано у вигляді паралельного алгоритму.

До появи паралельних комп'ютерних систем використовувались виключно послідовні алгоритми, орієнтовані спочатку на людину, а потім на однопроцесорні комп'ютери. Проблема розроблення паралельних алгоритмів з'явилась з появою реальних паралельних систем. "Паралельна математика", яка вивчала методи і засоби побудови та аналізу паралельних алгоритмів бурхливо розвивалась у 60-70-х роках минулого століття [4; 7; 36]. Учені отримали цікаві результати про паралельні властивості цілого ряду задач. При цьому для окремих задач довелося відмовитися від традиційних підходів і розробити цілком нові, які дозволили побудувати ефективні паралельні алгоритми. З'ясувалося, що оцінка ефективності різних паралельних алгоритмів - важке завдання, тому що для цього треба враховувати різні додаткові фактори, наприклад, час передачі даних між процесорами.

Для спрощення аналізу паралельних алгоритмів було запропоновано *концепцію необмеженого паралелізму*, яка ґрунтувалась на таких поняттях [4; 7]:

- кожна операція виконується за одиницю часу,
- час передавання даних між процесорами системи не враховується,
- застосовується будь-яка потрібна (необмежена) кількість процесорів.

Зрозуміло, що концепція необмеженого паралелізму не відповідає дійсності, що до часу виконання паралельної програми в реальних КС, оскільки не враховується більшість додаткових параметрів цієї системи, але вона дозволяє на етапі проектування парале-

льних алгоритмів оцінити для них час виконання і вибрати оптимальний алгоритм.

Концепція необмеженого паралелізму дозволяє спростити розрахунок *коефіцієнта прискорення* (*Кп*), який показує скорочення часу виконання паралельної програми в паралельній системі з P процесорами (Tp) в порівнянні з часом виконання послідовної програми в однопроцесорній системі (T_1):

$$Кп = T_1 / Tp$$

Поряд з коефіцієнтом прискорення використовується поняття коефіцієнта *ефективності* застосування комп'ютерної системи (*Ке*), який показує ступінь використання P процесорів системи:

$$Ке = T_1 / (Tp * P)$$

Для поєднання необмеженого паралелізму в концепції необмеженого паралелізму з обмеженим числом процесорів можна використати лему Брента [4]:

Лема Брента. Якщо час розв'язання задачі, що включає n операцій в паралельній системі з необмеженим числом процесорів дорівнює t, то час виконання цієї задачі в системі з p процесорів t_p буде не більше, ніж

$$t_p = t + (n - t)/p .$$

Теорема, що наведена нижче, дозволяє оцінити час виконання бінарних операцій, алгоритм виконання яких можна подати бінарним деревом.

Теорема Мунро – Петерсена [4]. Якщо в комп'ютерній системі з p процесорів виконується обчислення скалярної величини, яке потребує m бінарних операцій, то необхідний час t_p :

$$t_p > = \begin{cases}]\log p[+](m + 1 - 2\text{^}]\log p[/ p[& \text{якщо } m > = 2\text{^}[\log p] \\ ]\log (m+1)[& \text{в інших випадках,} \end{cases}$$

де]x[– найменше ціле число, більше, або що дорівнює x, а логарифм береться за основою 2; ^ – означає ступінь.

За допомогою теореми Мунро-Петерсона можна оцінити час виконання операції множення матриць MA = MB * MC розміром $N*N$. Ця операція потребує N множень і N–1 додатків. Тобто $m = 2N - 1$. Звідки

$$\log (m+1) = \log (2N) = 1 + \log N$$

і на підставі теореми Мунро-Петерсона отримуємо

$$t_p > = \begin{cases}]\log p[+]2N - 2\text{^}]\log p[/ p[& \text{якщо } 2N-1 > = 2\text{^}]\log p[\\ ]\log N[+ 1 & \text{в інших випадках.} \end{cases}$$

Якщо кількість процесорів необмежена, то оптимальний час для операції дорівнює]log N[+ 1. При цьому на першому кроці виконуються паралельно всі множення ($N * N * N$), а потім за]log N[кроків – паралельні додавання. Для $N = 100$ час дорівнює 8 одиницям, при цьому знадобиться 1000 000 процесорів.

Ярусно-паралельна форма алгоритму. Подання паралельного алгоритму може бути виконано різними способами [7]. *Ярусно-паралельна форма* (ЯПФ) для паралельного алгоритму являє собою набір ярусів, у яких операції виконуються паралельно (рис. 2.2). Висота ЯПФ (H) – кількість ярусів, ширина i-го ярусу (R_i) – кількість операцій в ярусі, ширина ЯПФ (R) – найбільша з ширини ярусів. Виконання ЯПФ здійснюється за ярусами. Тому час виконання задачі, алгоритм який представлений ЯПФ, дорівнює висоті ЯПФ, а максимальна потрібна кількість процесорів – ширині ЯПФ.

Одна й та сама задача може мати декілька паралельних алгоритмів, і відповідно, декілька ЯПФ, що відрізняються як висотою, так і шириною. Застосовуючи ЯПФ, можна вибрати алгоритм з необхідним часом виконання або із заданою кількістю процесорів.

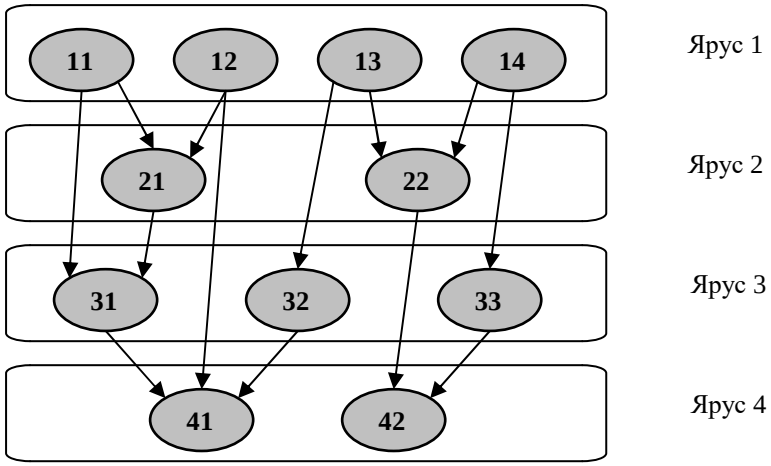

Рис. 2.2. Ярусно-паралельна форма алгоритму

Для ЯПФ, показаній на рис. 2.2, параметри дорівнюють $H = 4$, $R_1 = 4$, $R_2 = 2$, $R_3 = 3$, $R_4 = 2$, $R = \max(4, 2, 3, 2) = 4$. Знання цих параметрів дозволяє розрахувати:

- час розв'язування задачі в однопроцесорній системі:

$$T_1 = R_1 + R_2 + R_3 + R_4 = 11;$$

- час розв'язування задачі в багатопроцесорній системі:

$$T_p = H = 4;$$

- кількість процесорів, потрібних на кожному кроку обчислення ЯПФ:

$$P_1 = 4, \quad P_2 = 2, \quad P_3 = 3, \quad P_4 = 2;$$

- потрібну кількість процесорів, яка дозволяє розв'язати задачу за мінімальний час:

$$P = R = 4;$$

- коефіцієнт прискорення:

$$K_n = T_1/T_p = 11/4 = 2{,}75;$$

- коефіцієнт ефективності:

$$K_e = K_n/P = 2{,}75/4 = 0{,}69 \quad (69\ \%).$$

Паралельні алгоритми для задач лінійної алгебри. Розглянемо і оцінимо паралельні алгоритми для задач лінійної алгебри із застосуванням концепції необмеженого паралелізму.

- **Додавання векторів.** Операція додавання векторів розміру N

$$A = B + C$$

виконується за співвідношенням $a_i = b_i + c_i$, де $i = 1..N$.

Час виконання задачі в однопроцесорній КС дорівнює $T_1 = N$, час виконання в паралельній КС дорівнює $T_p = 1$ для $P = N$, коефіцієнт прискорення $K_n = P$, тобто це ідеальна для виконання в паралельній системі задача.

Якщо кількість процесорів обмежена $P < N$, то паралельний алгоритм можна показати у вигляді

$$A_н = B_н + C_н,$$

де $H = N/P$; $A_н$ – H елементів вектора A.

- **Скалярне множення векторів.** Операція скалярного множення векторів

$$a = (B * C)$$

виконується за співвідношенням

$$a = b_1 * c_1 + \ldots + b_i * c_i + \ldots + b_n * c_n \qquad i = 1..N.$$

Час виконання задачі в однопроцесорній КС дорівнює $T_1 = N + (N - 1)$.

Операцію скалярного множення векторів відносять до групи операцій, для яких побудова паралельних алгоритмів базується на *здвоюванні* [6]. Для подібних схем час виконання має логарифмічну залежність. Час виконання скалярного множення в паралельній КС складається із виконання множення $b_i * c_i$, яке можна здійснити за один крок при $P = N$, і з наступного додавання результатів множення. Тут використовується здвоювання, що дозволяє виконати множення за час, що дорівнює $\lceil \log N \rceil$.

Таким чином, час виконання операції скалярного множення $Tp = 1 + \lceil \log N \rceil$ для $P = N$, коефіцієнт прискорення $Kn = (2*N - 1)/(1 + \lceil \log N \rceil)$, тобто це частково паралельна задача, у якій чергуються паралельні і послідовні частини.

Якщо кількість процесорів P обмежена і значно менша за N, то паралельний алгоритм можна показати у вигляді

$$a_i = (B_н * C_н), \qquad i = 1 .. P.$$
$$a = a + a_i$$

- **Множення вектора на матрицю.** Операція множення вектора на матрицю

$$A = B * MC$$

виконується за співвідношенням

$$a_i = b1 * c_{1i} + \ldots + b_i * c_{ii} + \ldots + b_n * c_{ni}, \qquad i = 1 .. N.$$

Час виконання задачі в однопроцесорній КС дорівнює $T_1 = N*(N + (N - 1))$. В паралельному алгоритмі використано схему здвоювання, тому час виконання в паралельній КС дорівнює $Tp = 1 + \lceil \log_2 N \rceil$ якщо $P = N*N$, і коефіцієнт прискорення дорівнює $Kn = (2N*(N+(N - 1))/(1 + \log_2 N)$, тобто це частково паралельна задача, в якій чергуються паралельні і послідовні частини.

Якщо кількість процесорів обмежена і $P < N$, то паралельний алгоритм можна подати у вигляді

$$A_н = (B * MC_н),$$

де $MC_н$ - H рядків (стовпців) матриці MC.

- **Додавання матриць.** Операція додавання матриць

$$MA = MB + MC$$

виконується за співвідношенням

$$a_{ij} = b_{ij} + c_{ij}, \qquad \text{де } i,j = 1.. N.$$

Час виконання задачі в однопроцесорній КС дорівнює $T_1 = N*N$, час виконання в паралельній КС дорівнює $Tp = 1$ для $P = N*N$, коефіцієнт прискорення $Kn = P$, тобто це теж ідеальна для виконання в паралельній КС задача, але слід звернути увагу на те, як збільшилася кількість процесорів в КС, що потрібне розв'язання задачі за одиницю часу.

Якщо кількість процесорів обмежена $P << N$, то паралельний алгоритм можна подати у вигляді

$$MA_н = MB_н + MC_н .$$

- **Множення матриць.** Операція множення матриць

$$MA = MB * MC$$

виконується за співвідношенням

$$a_{ij} = a_{ij} + b_{ik} * c_{kj}, \qquad \text{де } i,j,k = 1 .. N.$$

Час виконання задачі в однопроцесорній КС $T_1 = N*N*(N + (N - 1))$, час виконання в паралельній КС $Tp = 1 + \lceil \log N \rceil$ якщо P

$= N*N*N$, коефіцієнт прискорення $Kп = N*N*(N + (N – 1)) / (1 +]\log N[ )$.

Якщо кількість процесорів обмежена і значно менше N, паралельний алгоритм множення матриць можна показати у вигляді

$$MA_н = MB * MC_н.$$

Кожен процес формує свою частину результату $MAн$ (H рядків), використовуючи повну матрицю MB і частину матриці $MCн$ (H стовпців).

2.7.2. Розроблення паралельного алгоритму

Розроблення алгоритму розв'язування завдання завжди була важливим і найбільш складним етапом розроблення програми [18; 21; 28; 53]. Для паралельного алгоритму цей етап має свої особливості і складається з окремих кроків: *декомпозиції, проектування взаємодії задач, об'єднання, планування обчислень.*

Етапи розроблення паралельного алгоритму показано на рис. 2.3.

1. **Декомпозиція.** Крок, на якому вхідне завдання аналізується і розкладається на частини – підзадачі, виконання яких може бути паралельним (на рис. 2.3 це підзадачі A .. P). Існують різні види декомпозиції : *декомпозиція за даним, функціональна декомпозиція, об'єктна декомпозиція* та ін. [7; 21] . Кількість підзадач має перевищувати кількість процесорів КС, що дозволить оптимізувати наступні етапи розроблення алгоритму. При цьому також слід мінімізувати кількість обчислень, а також обсяг інформації, що пересилається між підзадачами. Наступне планування обчислень (загрузки процесорів) буде спрощеним, якщо підзадачі мають однаковий розмір. Щодо розміру підзадач, то він пов'язаний з поняттям «зерна» алгоритму, яке, в свою чергу пов'язане з рівнем паралелізму і типом використовуваної паралельної системи. Дрібнозернистий паралелізм – це рівень команд, середньозернистий – рівень циклів, процедур і модулів (класів), великозернистий – рівень програм.

Декомпозиція – найважливіший крок розроблення алгоритму, який впливає на всі інші етапи. Помилки в декомпозиції мають найбільш тяжкі наслідки для всієї програми в цілому, як за її коректності, так і за часом її виконання. Тому слід приділяти особливу увагу виконанню декомпозиції, розглядати кількість можливих її варіантів, з яких після попереднього порівняльного аналізу (наприклад, за допомогою концепції необмеженого паралелізму), вибрати оптимальний.

2. **Проектування взаємодії підзадач.** Визначаються дії і методи, які потрібні для організації взаємодії підзадач (передавання даних, синхронізація, доступ до спільних даних та ін.) Види взаємодії можуть бути *локальними або глобальними* залежно від того, яку кількість зв'язків з іншими має кожна підзадача; *синхронними або асинхронними* залежно від виду виконання обміну даних; *статичними або динамічними*, якщо вони можуть змінюватися в процесі виконання програми.

   Організація взаємодії тісно пов'язана з типом архітектури використовуваної КС. Для систем, де використовується взаємодія, що ґрунтується на моделі посилання повідомлень, оптимальність цієї організації буде значною мірою визначати час виконання програми, оскільки в таких системах час передавання повідомлень можна порівнювати з часом безпосередніх обчислень і впливати на загальний час роботи програми. Скорочення часу на взаємодію процесів можна досягнути, якщо зменшити загальну кількість взаємодій, а також обсяг даних, що передаються, і, по можливості, виконувати їх одночасно.

3. **Об'єднання.** Виконується об'єднання окремих підзадач у більші задачі з метою зменшення обсягів даних, які потрібно передавати між задачами. Кінцева мета об'єднання – збільшення ефективності алгоритму за рахунок скорочення часу його виконання і складності реалізації.

   Результат кроку об'єднання для рис. 2.3 – формування трьох задач $T1$ (підзадачі A, B, G, H), $T2$ (підзадачі C, K) і $T3$ (підзадачі D, E, F, L, M, P).

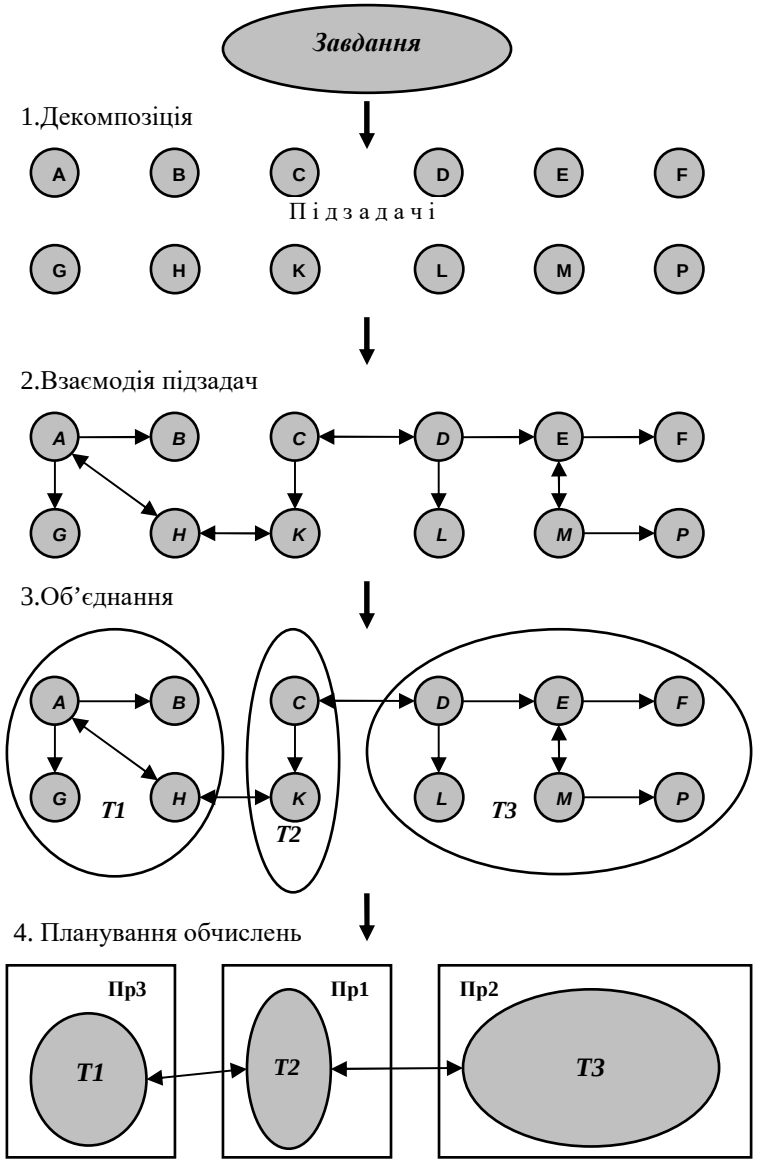

Рис. 2.3. Етапи розробки паралельного алгоритму

4. **Планування обчислень.** Виконується розподіл задач по процесорах КС. Оптимальне розміщення дозволить мінімізувати реальні затрати часу, які будуть пов'язані з виконанням обчислень і обміном даних між процесорами системи На рис. 2.3 задача Т1 буде виконуватися на процесорі 3, задача Т2 – на процесорі 1, задача Т3 – на процесорі 2. Планування обчислень (task scheduling) є важливою складовою процесу розроблення та виконання програми і на нього треба звертати окрему увагу. Вирішення проблеми планування залежить від виду планування (динамічне або статичне) і пов'язане з використанням спеціальних різноманітних алгоритмів планування [32; 34; 54].