**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**

**«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»**

Кафедра обчислювальної техніки

**КУРСОВА РОБОТА**

з дисципліни «Паралельні та розподілені обчислення»

на тему: «Розробка програмного забезпечення для паралельних

комп’ютерних систем»

Студента 3 курсу групи ІО-01

напряму підготовки 050102

«Комп’ютерна інженерія»

Бовсунівський О. В.

Керівник доцент Корочкін О.В.

Національна оцінка\_\_\_\_\_\_\_\_\_

Кількість балів: \_\_\_\_\_\_\_\_\_\_\_

Оцінка: ECTS \_\_\_\_\_\_\_\_\_\_\_

Члени комісії \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)

\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)

\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)

Київ-2013 рік

Національний технічний університет України

“Київський політехнічний інститут”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Освітньо-кваліфікаційний рівень бакалавр

Напрям підготовки 6.050102 «Комп’ютерна інженерія»

***З А В Д А Н Н Я***

НА КУРСОВУ РОБОТУ СТУДЕНТУ

Бовсунівський Олег Вікторович

1. Тема роботи «Розробка програмного забезпечення для паралельних

комп’ютерних систем»

керівник роботи Корочкін Олександр Володимирович к.т.н., доцент

2. Строк подання студентом роботи 11 травня 2013 р.

3. Вихідні дані до роботи

- засоби роботи з процесами в мові C#, MPI

- математична задача A=B+C(MO(MX\*MZ))

- структури ПКС ОП та ПКС ЛП

- мови і бібліотеки програмування: C#, MPI, Win32

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

- Огляд Win32 і POSIX

- розробка і тестування програми ПРГ1 для ПКС ОП

- розробка і тестування програми ПРГ2 для ПКС ЛП

5. Перелік графічного матеріалу

- структурна схема ПКС ОП

- структурна схема ПКС ЛП

- схеми алгоритмів процесів і головної програми для ПРГ1

- схеми алгоритмів процесів і головної програми для ПРГ2.

6. Дата видачі завдання \_\_\_\_\_\_\_\_\_\_\_\_\_

КАЛЕНДАРНИЙ ПЛАН

|  |  |  |
| --- | --- | --- |
| №  з/п | Назва етапів виконання КР | Строк виконання етапів КР |
|  | Виконання огляду для розділу 1 | 20.03.2013 |
|  | Розробка паралельного алгоритму  рішення задачі | 1.04.2013 |
|  | Розробка алгоритмів процесів | 6.04.2014 |
|  | Розробка схем взаємодії процесів | 13.04.2013 |
|  | Розробка програм | 20.04.2013 |
|  | Тестування програм | 30.04.2013 |
|  | Оформлення КР | 10.05.2013 |
|  | Захист КР | 18.05.2013 |

**Студент** \_\_\_\_\_\_\_\_\_ Бовсунівський О. В.

**Керівник роботи** \_\_\_\_\_\_\_\_\_\_ Корочкін О.В.

**Пояснювальна записка**

**Зміст**

[Розділ 1. Огляд Win32 і POSIX. 7](#_Toc356293242)

[1.1 Огляд POSIX 7](#_Toc356293243)

[Бібліотека Pthreads 10](#_Toc356293244)

[1.1.1 Потоки 10](#_Toc356293245)

[1.1.2 Семафори 11](#_Toc356293246)

[1.1.3 Блокування (м'ютекси) 12](#_Toc356293247)

[1.1.4 Умовні змінні 13](#_Toc356293248)

[1.1.5 Різне 14](#_Toc356293249)

[1.2 Огляд і ключові поняття Win32 16](#_Toc356293250)

[1.2.1 Створення і завершення потоків в Win32 17](#_Toc356293251)

[1.2.2 Синхронізація потоків в Win32. 20](#_Toc356293252)

[1.2.3 Пріоритети в Win32 26](#_Toc356293253)

[1.2.4 Взаємодія між процесами в Win32 26](#_Toc356293254)

[Висновки до Розділу 1 27](#_Toc356293255)

[Розділ 2. Розробка програмного забезпечення для паралельних комп’ютерних систем 29](#_Toc356293256)

[2.1 Розробка ПЗ для системи з спільною пам’яттю 29](#_Toc356293258)

[2.1.1 Розробка паралельного математичного алгоритму 29](#_Toc356293259)

[2.1.2 Розробка алгоритмів процесів 29](#_Toc356293260)

[2.1.3 Розробка схеми взаємодії задач 30](#_Toc356293261)

[2.1.4 Розробка програми 30](#_Toc356293262)

[2.2 Розробка ПЗ для системи з локальною пам’яттю 30](#_Toc356293263)

[2.2.1 Розробка паралельного математичного алгоритму 30](#_Toc356293264)

[2.2.2 Розробка алгоритмів процесів 31](#_Toc356293265)

[2.2.3 Розробка схеми взаємодії задач 31](#_Toc356293266)

[2.2.4 Розробка програми 32](#_Toc356293267)

[Висновки до Розділу 2 32](#_Toc356293268)

[Розділ 3. Тестування програмного забезпечення 33](#_Toc356293269)

[3.1 Тестування ПЗ для системи з спільною пам’яттю 33](#_Toc356293271)

[3.2 Тестування ПЗ для системи з локальною пам’яттю 38](#_Toc356293272)

[Висновки до Розділу 3 42](#_Toc356293273)

[Висновки 43](#_Toc356293274)

[Література 45](#_Toc356293275)

[Додатки 46](#_Toc356293276)

[Додаток А. Схеми алгоритмів процесів 47](#_Toc356293277)

[Додаток Б. Схеми алгоритмів основних програм 48](#_Toc356293278)

[Додаток В. Лістинг програм 49](#_Toc356293279)

[Лістинг програми для системи зі спільною пам’яттю 50](#_Toc356293280)

[Лістинг програми для системи з локальною пам’яттю 56](#_Toc356293281)

# Огляд Win32 і POSIX.

## Огляд POSIX

Потоково-орієнтована бібліотека POSIX (Portable Operating System Interface - інтерфейс переносимої операційної системи) містить набір системних функцій, які в поєднанні з мовними засобами успішно використовуються для підтримки паралелізму.

Pthreads - це набір інтерфейсів потокової обробки, розроблений комітетами Інституту інженерів з електротехніки та електроніки (Institute of Electrical and Electronics Engineers - IEEE), що відповідають за специфікацію інтерфейсу POSIX (Portable Operating System Interface). Буква P в слові Pthreads означає POSIX і фактично інтерфейс Pthreads іноді називають також потоками POSIX. По суті справи, комітет POSIX визначив основний набір функцій і структур даних, який повинні були взяти на озброєння велику кількість постачальників, щоб багатопоточний код можна було легко передавати з однієї операційної системи в іншу. Мрії комітету були здійснені постачальниками UNIX, які майже всі впровадили інтерфейс Pthreads. (Винятком стала корпорація Sun, яка продовжує віддавати перевагу потокам Solaris \* в якості основного програмного інтерфейсу.) Можливість перенесення Pthreads була додатково розширена за рахунок утвердження в Linux і перенесення на платформу Windows.

Pthreads являє собою прикладний програмний інтерфейс для виконання більшості дій, необхідних для потоків. Сюди входить створення і переривання потоків, очікування завершення потоків і управління взаємодією між ними. В останній категорії є декілька способів блокування, які не дають двом потокам одночасно змінювати одні й ті ж значення даних: м'ютекси, змінні умови і семафори. (З технічної точки зору семафори не є частиною Pthreads, але вони концептуально тісно пов'язані з багатопоточність і доступні у всіх системах, в яких виконується Pthreads).

Щоб використовувати Pthreads, розробники повинні написати код спеціально для цього програмного інтерфейсу. Це означає, що вони повинні включити файли заголовка, вказати структури даних Pthreads і назвати функції, специфічні для Pthreads. По суті, цей процес нічим не відрізняється від використання інших бібліотек. Так само як інші бібліотеки в UNIX і Linux, бібліотека Pthreads просто пов'язана з кодом програми (через параметр-lpthread).

Незважаючи на те, що бібліотека Pthreads практично універсальна (хоч вона і не така розгорнута, як деякі вбудовані набори програмних інтерфейсів) і легко переноситься, вона страждає від цілого ряду серйозних обмежень, загальних для всіх вбудованих програмних інтерфейсів: для неї потрібно об'ємний код з підтримкою багатопоточності. Інакше кажучи, написання коду для Pthreads неминуче прив'язує базу коду до багатопотокової моделі. Більше того, ухвалення певних рішень, таких як кількість використовуваних потоків, може виявитися жорстко запрограмованим у програмі. Щоб компенсувати ці обмеження, Pthreads забезпечує розширене управління багатопотоковими операціями - цей програмний інтерфейс спочатку є низькорівневим, що призводить, головним чином, до виконання простих завдань розбиття на потоки в кілька етапів. Наприклад, використання багатопотокового циклу для послідовної обробки великого блоку даних вимагає оголошення багатопоточних структур, створення кожного потоку окремо, обчислення і призначення кордонів циклів для кожного потоку, і, нарешті, обробки переривання потоків - всі ці функції повинні бути закодовані розробником. Якщо цикл не просто повторюється, обсяг коди, що забезпечує потоки, може істотно збільшитися. Справедливості заради треба сказати, що необхідність коду великого обсягу притаманна для всіх вбудованих програмних інтерфейсів, а не тільки для Pthreads.

Через великий обсяг коду, необхідного для виконання звичайних операцій, розробники почали все частіше шукати більш зручну альтернативу Pthreads.

Бібліотека Pthread містить більше 60 функцій, які можна розділити на наступні категорії.

1. Функції управління потоками.

o Конфігурування потоків.

o Відміна потоків.

o Стратегії планування потоків.

o Доступ до даних потоків.

o Обробка сигналів.

o Функції доступу до атрибутів потоків.

o Конфігурування атрибутів потоків.

o Конфігурування атрибутів, що відносяться до стека потоків.

o Конфігурування атрибутів, що відносяться до стратегій планування потоків.

2. Функції управління м'ютексів.

o Конфігурування м'ютексів.

o Управління пріоритетами.

o Функції доступу до атрибутів м'ютексів.

o Конфігурування атрибутів м'ютексів.

o Конфігурування атрибутів, що відносяться до протоколів м'ютексів.

o Конфігурування атрибутів, що відносяться до управління пріоритетами м'ютексів.

3. Функції управління умовними змінними.

o Конфігурування умовних змінних.

o Функції доступу до атрибутів умовних змінних.

o Конфігурування атрибутів умовних змінних.

o Функції спільного використання умовних змінних.

Бібліотека Pthread може бути реалізована на будь-якій мові, але для відповідності стандарту POSIX вона повинна бути узгоджена зі стандартизованим інтерфейсом. Бібліотека Pthread - не єдина реалізація потокового API-інтерфейсу. Існують інші реалізації, створені сторонніми фірмами виробниками апаратних і програмних засобів. Наприклад, середа Sun підтримує бібліотеку Pthread і власний варіант бібліотеки потоків Solaris.

### Бібліотека Pthreads

### Потоки

Компіляція компілятором gcc з ключем:-pthread

Підключення бібліотеки:

# Include <pthread.h>

Спочатку створюється один потік, виконуючий функцію main. Цей потік може створювати дочірні потоки, які починають виконуватися паралельно. Ті, у свою чергу, також можуть створювати свої дочірні потоки. Всі потоки мають доступ до всіх глобальним змінним програми.

Кожен потік повинен бути описаний у вигляді окремої функції виду:

void \* func (void \*);

Для кожного потоку описується:

pthread\_t pid; / / дескриптор потоку

pthread\_attr\_t attr; / / атрибути потоку

Ініціалізація атрибутів потоку:

pthread\_attr\_init (& attr);

pthread\_attr\_setscope (& attr, PTHREAD\_SCOPE\_SYSTEM);

PTHREAD\_SCOPE\_SYSTEM - потік конкурує за процесор зі всіма потоками системи

PTHREAD\_SCOPE\_PROCESS - потік конкурує за процесор з потоками, створеними батьківським потоком

Створення потоку:

pthread\_create (& pid, & attr, func, arg);

arg - параметр, переданий функції func при запуску

Якщо потік створений успішно, функція повертає нуль.

Завершення потоку:

pthread\_exit (value);

value - значення, що повертається або NULL

Функція викликається неявно, якщо потік просто завершив виконання.

Очікування завершення дочірнього потоку батьківським:

pthread\_join (pid, value\_ptr);

value\_ptr - адреса змінної для повертається потоком значення (або NULL).

### Семафори

Семафор - особливий тип розділяється змінної, яка обробляється двома неподільними операціями: P і V. Значення семафора - невід'ємні цілі числа. Операція V сигналізує, що подія відбулася: вона збільшує значення семафора на 1. Операція P призупиняє процес до моменту, коли подія відбудеться: вона чекає, коли значення семафора стане позитивним, і зменшує його на 1.

Підключення бібліотеки:

# Include <semaphore.h>

Опис семафора:

sem\_t lock;

Ініціалізація семафора:

sem\_init (& lock, SHARED, init\_value);

SHARED = 1 - семафор може бути розділяються між процесами

0 - семафор використовують потоки тільки одного процесу

init\_value - початкове значення

Операції над семафором:

sem\_wait (& lock); / / Операція P: {while (lock == 0); lock -;}

sem\_post (& lock); / / Операція V: {lock + +;}

Приклад: виробник-споживач

sem\_t empty, full;

int data;

int main ()

{...

  sem\_init (& empty, SHARED, 1);

  sem\_init (& full, SHARED, 0);

  ...

}

void \* Producer (void \* arg)

{...

  for (...)

  {Sem\_wait (& empty);

    data = Expression ();

    sem\_post (& full);

  }

  ...

}

void \* Consumer (void \* arg)

{...

  for (...)

  {Sem\_wait (& full);

    Use (data);

    sem\_post (& empty);

  }

  ...

}

### Блокування (м'ютекси)

М'ютекси використовуються для виділення критичних інтервалів.

Опис мьютекса:

pthread\_mutex\_t mutex;

Ініціалізація мьютекса (з атрибутами за замовчуванням):

pthread\_mutex\_init (& mutex, NULL);

Операції над м'ютексів:

pthread\_mutex\_lock (& ​​mutex); / / блокування мьютекса

pthread\_mutex\_unlock (& ​​mutex); / / розблокування мьютекса

Приклад: опис критичної секції

pthread\_mutex\_lock (& ​​mutex);

Use (data);

pthread\_mutex\_unlock (& ​​mutex);

### Умовні змінні

Умовна мінлива використовуються для призупинення процесу, виконання якого може продовжуватися після виконання деякої умови. Про це йому сигналізує інший процес. З кожною умовної змінної пов'язана чергу очікують її процесів.

Опис умовної змінної:

pthread\_cond\_t cond;

Ініціалізація умовної змінної (з атрибутами за замовчуванням):

pthread\_cond\_init (& cond, NULL);

Операції над умовними змінними:

pthread\_cond\_wait (& cond, & mutex); / / постановка процесу в чергу очікування

/ / (Розблокує mutex)

pthread\_cond\_signal (& cond); / / запуск першого процесу в черзі

pthread\_cond\_broadcast (& cond); / / запуск всіх процесів у черзі

Приклад: реалізація бар'єру

pthread\_cond\_t go;

pthread\_mutex\_t barrier;

int n = 0, nproc;

void barrier ()

{Pthread\_mutex\_lock (& ​​barrier);

  n + +;

  if (n <nproc) pthread\_cond\_wait (& go, & barrier);

          else {n = 0; pthread\_cond\_broadcast (& go);}

  pthread\_mutex\_unlock (& ​​barrier);

}

Групи операцій

Управління потоками

М'ютекси

Умовні змінні

Управління потоками

Створення та видалення потоків

Створення потоку:

int pthread\_create (pthread\_t \* thread, const pthread\_attr\_t \* attr, void \* (\* start\_routine) (void \*), void \* arg);

Завершення потоку:

void pthread\_exit (void \* value\_ptr);

Ініціалізація атрибутів потоку:

int pthread\_attr\_init (pthread\_attr\_t \* attr);

Знищення атрибутів потоку:

int pthread\_attr\_destroy (pthread\_attr\_t \* attr);

Завершення потоку з іншого потоку: pthread\_cancel

Очікування завершення потоку:

int pthread\_join (pthread\_t thread, void \*\* value\_ptr);

Від'єднання потоку:

int pthread\_detach (pthread\_t thread, \*\* value\_ptr);

### Різне

Отримання ідентифікатора поточного потоку:

pthread\_t pthread\_self (void);

Визначення ідентичності ідентіфікоторов потоку:

int pthread\_equal (pthread\_t t1, pthread\_t t2);

Виконання функції одноразово в процесі:

int pthread\_once (pthread\_once\_t \* once\_control, void (\* init\_routine) (void));

pthread\_once\_t once\_control = PTHREAD\_ONCE\_INIT;

Переключитися на інший потік (відмовитися від поточного кванта часу):

void pthread\_yield ();

М'ютекси

Mutual exclusion - взаємне виключення. Використовуються для взаємовиключного доступу до ресурсу. Потік може заснути тільки при очікуванні мьютекса.

М'ютекс: pthread\_mutex\_t.

Атрибути мьютекса: pthread\_mutexattr\_t.

Створення та видалення м'ютексів

Ініціалізація мьютекса:

int pthread\_mutex\_init (pthread\_mutex\_t \* mutex, pthread\_mutexattr\_t \* attr);

Знищення мьютекса:

int pthread\_mutex\_destroy (pthread\_mutex\_t \* mutex);

Ініціалізація атрибутів мьютекса:

int pthread\_mutexattr\_init (pthread\_mutexattr\_t \* attr);

Видалення атрибутів мьютекса:

int pthread\_mutexattr\_destroy (pthread\_mutexattr\_t \* attr);

Захоплення і звільнення мьютекса

Захоплення мьютекса:

int pthread\_mutex\_lock (pthread\_mutex\_t \* mutex);

Спроба захоплення мьютекса:

int pthread\_mutex\_trylock (pthread\_mutex\_t \* mutex);

Звільнення мьютекса:

int pthread\_mutex\_unlock (pthread\_mutex\_t \* mutex);

Умовні змінні

Дозволяють синхронізувати потоки залежно від значень змінних. Використовуються тільки спільно з м'ютексів.

Умовна змінна: pthread\_cond\_t.

Атрибути умовної змінної: pthread\_condattr\_t.

Створення та видалення умовних змінних

Ініціалізація умовної змінної:

int pthread\_cond\_init (pthread\_cond\_t \* condition, pthread\_condattr\_t \* attr);

Видалення умовної змінної:

int pthread\_cond\_destroy (pthread\_cond\_t \* condition);

Ініціалізація атрибутів умовної змінної:

int pthread\_condattr\_init (pthread\_condattr\_t \* attr);

Видалення атрибутів умовної змінної:

int pthread\_condattr\_destroy (pthread\_condattr\_t \* attr);

Використання умовних змінних

Очікування події:

int pthread\_cond\_wait (pthread\_cond\_t \* cond);

int pthread\_cond\_timedwait (

pthread\_cond\_t \* cond, pthread\_mutex\_t \* mutex, const struct timespec \* abstime);

Сигналізація про подію:

int pthread\_cond\_signal (pthread\_cond\_t \* condition);

int pthread\_cond\_broadcast (pthread\_cond\_t \* condition);

## Огляд і ключові поняття Win32

Win32 містить ряд функцій, які реалізують кілька механізмів синхронізації потоків по доступу до ресурсу (в системі Windows потоки - threads - є завданнями, які можуть виконуватися паралельно або в режимі поділу часу). Бібліотека містить процедури для реалізації деяких механізмів[9].

Об'єкти синхронізації та функції очікування. (synchronization objects and wait functions) Об'єкт синхронізації може бути дозволений або заборонений. Функції очікування дозволяють потоку блокувати своє виконання до тих пір, поки об'єкт не буде дозволений. Крім цього API реалізує і інші механізми:

* Перехресне введення - виведення (overlapped input-output) (не розглядається)
* Асинхронний виклик процедур (asynchronus procedure calls)
* Об'єкти критичної ділянки (Critical section objects)
* Блокований доступ до змінних (Interlocked variable access)

### Створення і завершення потоків в Win32

**Створення потоку.**

Первинний потік, який присутній в програмі, починає своє виконання з головної функції потоку типу WinMain. Для створення вторинного потоку необхідно створити і для нього вхідну функцію, яка виглядає приблизно так:

*DWORD WINAPI ThreadFunc (PVOID pParam) {DWORD dwResult = 0; ......... return dwResult;}.*

Ім'я у функції вторинного потоку, на відміну від первинного, може бути будь-яким, проте за наявності декількох різних потоків, назвати функції необхідно по-різному, інакше система створить різні реалізації однієї і тієї ж функції. Коли потік закінчить своє виконання, він поверне управління системі, пам'ять, відведена під його стек, буде звільнена, а лічильник користувачів його об'єкта ядра "потік" зменшиться на 1. Коли лічильник обнулиться, цей об'єкт ядра буде зруйнований.

Для створення свого потоку необхідно використовувати функцію *CreateThread: HANDLE CreateThread (LPSECURITY\_ATTRIBUTES lpThreadAttributes, DWORD dwStackSize, LPTHREAD\_START\_ROUTINE lpStartAddress, LPVOID lpParameter, DWORD dwCreationFlags, LPDWORD lpThreadId);*

При кожному виклику цієї функції система створює об'єкт ядра (потік). Це не сам потік, а компактна структура даних, яка використовується операційною системою для управління потоком і зберігає статистичну інформацію про потоці. Система виділяє пам'ять під стек потоку з адресного простору процесу. Новий потік виконується в контексті того ж процесу, що і батьківський потік. Тому він отримує доступ до всіх описувач об'єктів ядра, всієї пам'яті і стека всіх потоків в процесі. За рахунок цього потоки в рамках одного процесу можуть легко взаємодіяти один з одним. CreateThread - це Windows-функція, яка створює потік. Якщо ви пишете код на С / С + + не викликайте її. Замість неї Ви повинні використовувати \_beginthreadex з бібліотеки Visual C + +.

Параметри функції CreateThread[10]:

* LpThreadAttributes - є покажчиком на структуру LPSECURITY\_ATTRIBUTES. Для присвоєння атрибутів захисту за замовчуванням, передавайте в цьому параметрі NULL.
* DwStackSize - параметр визначає розмір стека, що виділяється для потоку із загального адресного простору процесу. При передачі 0 - розмір встановлюється в значення за замовчуванням.
* LpStartAddress - покажчик на адресу вхідний функції потоку. LpParameter - параметр, який буде передано всередину функції потоку.
* DwCreationFlags - приймає одне з двох значень: 0 - виконання починається негайно, або CREATE\_SUSPENDED - виконання припиняється до подальших вказівок.
* LpThreadId - Адреса змінної типу DWORD в який функція повертає ідентифікатор, приписаний системою новому потоку.

**Завершення потоку.**

Потік можна завершить чотирма способами:

* функція потоку повертає керування (рекомендований);
* потік самознищується викликом функції ExitThread;
* інший потік процесу викликає функцію TerminateThread;
* завершується процес, що містить даний потік.

Всі способи, за винятком рекомендованого, є небажаними і повинні використовуватися тільки у форс-мажорних обставинах. Функція потоку, повертаючи управління, гарантує коректну очищення всіх ресурсів, що належать даному потоку. При цьому:

* будь-які С + + об'єкти, створені даними потоком, знищуються відповідними деструкторами;
* система коректно звільняє пам'ять, яку займав стек потоку; - система встановлює код завершення даного потоку. Його функція і повертає;
* Лічильник користувачів даного об'єкта ядра (потік) зменшується на 1.

При бажанні негайно завершити потік зсередини використовують функцію ExitThread (DWORD dwExitCode). При цьому звільняються всі ресурси ОС, виділені даному потоку, але С + + ресурси (наприклад, об'єкти класів С + +) не очищуються. Саме тому не рекомендовано завершувати потік, використовуючи цю функцію. Якщо ж її використовувати, то кодом повернення потоку буде той параметр, який передано у дану функцію. Як і для CreateThread для бібліотеки Visual C + + існує її аналог \_endthreadex, який і варто використовувати.

Якщо з'явилася необхідність знищити потік зовні, то це має зробити функція TeminateThread. Ця функція зменшить лічильник користувачів об'єкта ядра (потік) на 1, однак при цьому не зруйнує і не очистить стек потоку. Стек буде існувати, поки не завершиться процес, якому належить потік. При задачах, постійно створюють і знищують потоки, це призводить до втрати пам'яті усередині процесу. При завершенні процесу відбувається наступне:

* Завершення потоку відбувається примусово.
* Деструктори об'єктів не викликаються.
* При завершенні потоку з такої причини, пов'язаний з ним об'єкт ядра (потік) не звільняється до тих пір, поки не будуть закриті всі зовнішні посилання на цей об'єкт.

### Синхронізація потоків в Win32.

Працюючи паралельно, потоки спільно використовують адресний простір процесу. Також всі вони мають доступ до описувач (handles) відкритих в процесі об'єктів. Якщо декілька потоків одночасно звертаються до одного ресурсу або необхідно якось впорядкувати роботу потоків, то для цього використовують об'єкти синхронізації і відповідні механізми.

Кожен з об'єктів має функції створення і відкриття об'єктів, функції установки значень і скидання. Функції відкриття дозволяють безлічі потоків використовувати один і той самий об'єкт, але за умови, що він був до цього створений іншим потоком. Доступ до об'єкту при його відкритті визначається значенням відповідного параметра у функції відкриття. Кожен об'єкт при створенні має ідентифікатор (Handle), який в основному використовується у виклику функцій обслуговування цих об'єктів і при виклику функцій очікування. Ідентифікатор можна дублювати за допомогою функції DuplicateHadle, а так само закрити, використовуючи функцію CloseHandle. Об'єкт автоматично видаляється після того, як останній ідентифікатор закритий. При завершенні процесу створений ним ідентифікатор автоматично закривається.

При створенні об'єкта одним з параметрів є структура SECURITY\_ATTRIBUTES, яка містить дескриптор безпеки і інформацію про те, чи може ця структура бути успадкованими.

*typedef struct \_SECURITY\_ATTRIBUTES {/ / sa*

*DWORD nLength;*

*LPVOID lpSecurityDescriptor;*

*BOOL bInheritHandle;*

*} SECURITY\_ATTRIBUTES;*

Параметри:

* nLength - визначає довжину структури. Встановлюється в значення довжини для SECURITY\_ATTRIBUTES.
* lpSecurityDescriptor - вказує на об'єкт, який контролює поділ об'єкта. Якщо це поле встановлено в NULL, об'єкту може бути призначений дескриптор за замовчуванням викликає процесу.
* bInheritHandle - визначає, чи може об'єкт бути успадковувати процесом, який створюється.

**Об'єкти Event**

Об'єкти подій є об'єктами синхронізації, встановлювані в дозволений стан функціями SetEvent або PulseEvent. Існують два типи подій, використовуваних при синхронізації[2]:

* Ручного скидання (Manual-reset event). Стан об'єкта події залишається в дозволеному стані до тих пір, поки воно явно не буде скинуто функцією ResetEvent. Поки встановлений у дозволене стан цей об'єкт будь-яку кількість чекають потоків або будь-яких інших потоків, які визначають ідентифікатор цього об'єкта у виклику функції очікування можуть бути звільнені.
* Автоматичного скидання (Auto-reset event). Стан залишається дозволеним, до тих пір, поки одиночний чекаючий потік не звільниться, і стан об'єкта-події автоматично встановиться в заборонений стан.

Події часто використовуються для повідомлення деякого потоку або потокам про те, що деяка операція (наприклад введення-виведення) закінчена, і що повідомляє про це потік встановлює стан об'єкта в авторизований.

Функції, використовувані при управлінні подіями:

* CreateEvent

Створює іменовану або неіменовану подію:

*HANDLE CreateEvent (*

*LPSECURITY\_ATTRIBUTES lpEventAttributes, / / ​​покажчик на security attributes*

*BOOL bManualReset, / / ​​прапор ручного скидання*

*BOOL bInitialState, / / ​​прапор початкового стану*

*LPCTSTR lpName / / покажчик на ім'я об'єкта події*

*);*

*SetEvent*

*Функція встановлює значення події в дозволений стан.*

*BOOL SetEvent (*

*HANDLE hEvent / / ідентифікатор об'єкта події*

*);*

* PulseEvent

Функція встановлює стан події в дозволений стан і після звільнення потоків чи потоку скидає стан у заборонений. Навіть якщо жоден з потоків не вимагає розблокування стан все одно скидається. Параметри і вихід такий же як і у попередньої функції.

*BOOL PulseEvent (*

*HANDLE hEvent / / ідентифікатор об'єкта події*

*);*

* Reset Event

Використовується для скидання подій. Параметри такі ж як і в попередніх випадках.

*BOOL ResetEvent (*

*HANDLE hEvent / / ідентифікатор об'єкта події*

*);*

* OpenEvent

Функція повертає покажчик на іменовану вже існуючу подію.

*HANDLE OpenEvent (*

*DWORD dwDesiredAccess, / / ​​прапор доступу*

*BOOL bInheritHandle, / / ​​прапор спадкування*

*LPCTSTR lpName / / покажчик на ім'я об'єкта*

*);*

**Об'єкти Mutex**

Ці об'єкти знаходяться в дозволеному стані, якщо вони не належать жодному з потоків, і в забороненому стані, якщо який або потік володіє об'єктом. В поточний момент часу тільки одному з потоків може належати м’ютекс, за рахунок чого і реалізуються функції синхронізації паралельних потоків. Наприклад, якщо існують два потоки, які виконують запис в пам'ять, що розділяється, то один з них захоплює цей об'єкт, виконує операції доступу до пам'яті, після чого звільняє м’ютекс, тим самим дозволяючи іншому процесу виконати необхідні операції по доступу до поділюваного пам'яті. Потік може використовувати одну з функцій очікування для запиту на володіння об'єктом. У разі, якщо м’ютекс зайнятий іншим потоком, функція очікування блокує викликає потік до звільнення об'єкта. Якщо потік вже володіє м’ютексом, то виклик функцій очікування не призведе до глухого кута цього ж процесу.

Функції, використовувані при управлінні м’ютексом[3]:

* CreateMutex

Використовується для створення іменованого або неіменованого мьютекса.

*HANDLE CreateMutex (*

*LPSECURITY\_ATTRIBUTES lpMutexAttributes, / / ​​покажчик на security attributes*

*BOOL bInitialOwner, / / ​​прапор початкового володіння*

*LPCTSTR lpName / / покажчик імені об'єкта*

*);*

* OpenMutex

Використовується при відкритті існуючого об'єкта для отримання ідентифікатора.

*HANDLE OpenMutex (*

*DWORD dwDesiredAccess, / / ​​прапор доступу*

*BOOL bInheritHandle, / / ​​прапор спадкування*

*LPCTSTR lpName / / покажчик на ім'я об'єкта*

*);*

* ReleaseMutex

Функція звільняє мьютекс

*BOOL ReleaseMutex (*

*HANDLE hMutex / / ідентифікатор об'єкта*

*);*

**Об'єкти Semaphore**

Семафор у реалізації Win32 визначається як лічильник, приймає значення від нуля до певного значення. Лічильник декрементується кожен раз, коли потік закінчує очікування семафора і виконує функцію очікування його і інкрементіруется при звільненні семафора функцією ReleaseSemaphore. Якщо лічильник дорівнює нулю, то вважається, що він знаходиться в забороненому стані, і при виклику функцій очікування останні будуть блокувати процес до того моменту, поки один з потоків не звільнить семафор (інкрементіруется лічильник). Семафор застосовується для контролю над розділяються ресурсами, доступ до яких дозволено певному числу користувачів. Лічильник семафора не може бути менше нуля (стан блокування) і більше максимального значення.

Функції, використовувані при управлінні семафорами[4]:

* CreateSemaphore

Функція використовується для створення об'єкта семафора.

*HANDLE CreateSemaphore (*

*LPSECURITY\_ATTRIBUTES lpSemaphoreAttributes, / / ​​покажчик на security attributes*

*LONG lInitialCount, / / ​​початкове значення*

*LONG lMaximumCount, / / ​​максимальне значення*

*LPCTSTR lpName / / покажчик на ім'я об'єкта*

*);*

* OpenSemaphore

*Функція повертає ідентифікатор вже створеного іменованого семафора.*

*HANDLE OpenSemaphore (*

*DWORD dwDesiredAccess, / / ​​прапор доступу*

*BOOL bInheritHandle, / / ​​прапор спадкування*

*LPCTSTR lpName / / покажчик на ім'я об'єкта*

*);*

* ReleaseSemaphore

*Функція збільшує лічильник семафора на певне значення.*

*BOOL ReleaseSemaphore (*

*HANDLE hSemaphore, / / ​​ідентифікатор об'єкта семафора*

*LONG lReleaseCount, / / ​​число, додаємо до поточного стану семафора*

*LPLONG lpPreviousCount / / адресу попереднього лічильника*

*);*

**Критичні секції.**

Критична секція гарантує, що ділянка коду програми, обгороджена нею, не будуть виконуватися одночасно більш ніж одним потоком. Строго кажучи, критична секція не є об'єктом ядра. Вона являє собою структуру, що містить кілька прапорів і якийсь (не важливо який) об'єкт ядра. При вході в критичну секцію спочатку перевіряються прапори, і якщо з'ясовується, що вона вже зайнята іншим потоком, то виконується звичайна wait-функція. Критична секція примітна тим, що для перевірки, зайнята вона чи ні, програма не переходить в режим ядра (не виконується wait-функція) а лише перевіряються прапори. Через це синхронізація за допомогою критичних секцій найбільш швидка. Таку синхронізацію називають "синхронізація в режимі користувача". Приклад використання[1]:

*CRITICAL\_SECTION m\_crisec; // Критична секція*

*InitializeCriticalSection(&m\_crisec);//Ініціалізація критичної секції*

*EnterCriticalSection(&m\_crisec); // Вхід в КС*

*//Код буде виконуватися тільки одним потоком одночасно*

*LeaveCriticalSection(&m\_crisec); // Вихід з КС*

*DeleteCriticalSection(&m\_crisec);//Видалення КС*

### Пріоритети в Win32

У Windows існує 32 рівня пріоритету, від 0 до 31[6]. Вони групуються так: 31 - 16 рівні реального часу; 15 - 1 динамічні рівні; 0 - системний рівень, зарезервований для потоку обнулення сторінок (zero-page thread).

При створенні процесу, йому призначається один з шести класів пріоритетів:

Real time class (значення 24),

High class (значення 13),

Above normal class (значення 10),

Normal class (значення 8),

Below normal class (значення 6),

і Idle class (значення 4).

Пріоритет кожного потоку (базовий пріоритет потоку) складається з пріоритету його процесу і відносного пріоритету самого потоку. Є сім відносних пріоритетів потоків:

Normal: такий же як і у процесу;

Above normal: +1 до пріоритету процесу;

Below normal: -1;

Highest: +2;

Lowest: -2;

Time critical: встановлює базовий пріоритет потоку для Real time класу в 31, для решти класів в 15.

Idle: встановлює базовий пріоритет потоку для Real time класу в 16, для решти класів в 1.

### Взаємодія між процесами в Win32

Потоки одного процесу не мають доступу до адресного простору іншого процесу. Однак існують механізми для передачі даних між процесами.

**Колективна пам'ять**

Система віртуальної пам'яті в Win32 використовує файл підкачки - swap file (або файл розміщення - page file), маючи можливість перетворення сторінок оперативної пам'яті в сторінки файлу на диску і навпаки. Система може проектувати на оперативну пам'ять не тільки файл розміщення, але і будь-який інший файл. Додатки можуть використовувати цю можливість. Це може використовуватися для забезпечення більш швидкого доступу до файлів, а також для спільного використання пам'яті[5].

Такі об'єкти називаються проекціями файлів (на оперативну пам'ять) (file-mapping object). Для створення проекції файлу спочатку викликається функція CreateFileMapping (). Їй передається дескриптор (вже відкритого) файлу або вказується, що потрібно використовувати page file операційної системи. Крім цього, в параметрах їй передається прапор захисту, максимальний розмір проекції й ім'я об'єкта. Потім викликається функція MapViewOfFile (). Вона відображає уявлення файлу (view of a file) в адресний простір процесу.

По закінченні роботи викликається функція UnmapViewOfFile (). Вона звільняє пам'ять і записує дані у файл (якщо це не файл підкачки). Щоб записати дані на диск негайно, використовується функція FlushViewOfFile (). Проекція файлу, як і інші об'єкти ядра, може використовуватися іншими процесами через успадкування, дублювання дескриптора або по імені.

## Висновки до Розділу 1

• У даному розділі були представлені короткі теоретичні основи багатопоточності, а також був проведений огляд засобів і можливостей, якими володіє бібліотека Win32.

• Виконано аналіз засобів синхронізації потоків, який показав, що в Win32 для даних цілей використовуються критичні секції, семафори, події і м’ютекси. Даний набір засобів дозволяє ефективно вирішувати завдання взаємного виключення і синхронізації.

• На основі аналізу, виконаного в розділі 1 можна зробити висновок, що бібліотека Win32 є універсальним набором функцій, що дозволяють організувати роботу з процесами (потоками) в мовах, що не мають вбудованих засобів програмування процесів.

# Розробка програмного забезпечення для паралельних комп’ютерних систем



## Розробка ПЗ для системи з спільною пам’яттю

### Розробка паралельного математичного алгоритму

*Ah = Bh + C \* (MO \* ( MXh \* MZ))*

*СР: C, MO, MZ.*

### Розробка алгоритмів процесів

Блок схема алгоритму роботи процесів представлена у додатку А.

Алгоритм роботи процесів зі спільною пам’яттю:

|  |  |
| --- | --- |
| **T1** | **ТС/КУ** |
| 1. Очікування сигналу про закінчення вводу від Т2. 2. Копія С1=С, MO1= MO, MZ1= MZ. 3. Обчислення Ah = Bh + C1 \* (MO1 \* ( MXh \* MZ1)) 4. Сигнал Т2 про закінчення обчислення. | W2.1  КУ  S2.1 |
| **T2** | **ТС/КУ** |
| 1. Ввід MX, B,C. 2. Очікувати сигнали про закінчення вводу з Т3, Т4. 3. Сигнал всім процесам про закінчення вводу. 4. Копія С2=С, MO2= MO, MZ2= MZ. 5. Обчислення Ah = Bh + C2 \* (MO2 \* ( MXh \* MZ2)) 6. Очікування сигналу про завершення обчислення з Т1, Т3, Т4. 7. Вивід А. | W3-4.1  S1,3-4.1  КУ  W1,3-4.2 |
| **T3** | **ТС/КУ** |
| 1. Ввід MO. 2. Сигнал Т2 про закінчення введення. 3. Очікувати сигнали про закінчення вводу з Т2. 4. Копія С3=С, MO3= MO, MZ3= MZ. 5. Обчислення Ah = Bh + C3 \* (MO3 \* ( MXh \* MZ3)) 6. Сигнал Т2 про закінчення обчислення. | S2.1  W2.1  КУ  S2.2 |
| **T4** | **ТС/КУ** |
| 1. Ввід MZ. 2. Сигнал Т2 про закінчення введення. 3. Очікувати сигнали про закінчення вводу з Т2. 4. Копія С4=С, MO4= MO, MZ4= MZ. 5. Обчислення Ah = Bh + C4 \* (MO4 \* ( MXh \* MZ4)) 6. Сигнал Т2 про закінчення обчислення. | S2.1  W2.1  КУ  S2.2 |

### Розробка схеми взаємодії задач

**Мал. 2.1 Схема взаємодії задач для системи зі спільною пам’яттю**

### Розробка програми

Для реалізації алгоритму обчислення матричного рівняння на обчислювальній системі зі спільною пам’яттю було використано мову С# Ця мова має потужний апарат для забезпечення паралельної обробки.

Лістинг програми наведено у Додатку .

## Розробка ПЗ для системи з локальною пам’яттю

### Розробка паралельного математичного алгоритму

При розробці паралельного алгоритму передбачається, що розмірність задачі (N) більше чи рівне числу процесорів (P) (N> = P), при чому N кратне P, що дозволяє розбити задачу на окремі частини, кожна з яких обробляється своєю задачею. .

Розрахунок даного матричного рівняння можливо провести в два етапи:

*Ah = Bh + C \* (MO \* ( MXh \* MZ))*

### Розробка алгоритмів процесів

Алгоритми роботи процесів

**Т0**

1. Отримати BH, MXH, C, MZ, MO від Т1.
2. Обчислення Ah = Bh + C \* (MO \* ( MXh \* MZ))
3. Передати AH Т1.

**Т1**

1. Ввід B, C, MX.
2. Відправити C, B2H, MX2H в Т2.
3. Отримати MO, MZ від Т2.
4. Відправити MO, MZ, C, BH, MXH в Т0.
5. Обчислення Ah = Bh + C \* (MO \* ( MXh \* MZ))
6. Отримати A2H від Т2, AH від Т0.
7. Вивід А.

**Т2**

1. Ввід MO.
2. Отримати MZ від Т3; C, B2H, MX2H від Т1.
3. Відправити MO, C, BH, MXH в Т3; MZ, MO в Т1.
4. Обчислення Ah = Bh + C \* (MO \* ( MXh \* MZ))
5. Отримати AH від Т3.
6. Відправити A2H в Т1.

**Т3**

1. Ввід MZ.
2. Відправити MZ в Т2.
3. Отримати MO, C, BH, MXH від Т2.
4. Обчислення Ah = Bh + C \* (MO \* ( MXh \* MZ))
5. Відправити AH в Т2.

### Розробка схеми взаємодії задач

 Мал. 2.2 Схема взаємодії задач для системи з локальною пам’яттю

### Розробка програми

Для реалізації алгоритму обчислення матричного рівняння на обчислювальній системі з локальною пам’яттю було використано бібліотеку MPI. Ця бібліотека використовує підхід схожий на пересилку повідомлень.

Лістинг програми наведено у Додатку В.

## Висновки до Розділу 2

1. Порівнюючи процес розробки для системи зі спільною пам’яттю та системи з локальною пам’яттю можемо зробити висновок, що варіант зі спільною пам’яттю є більш простим у розробці та реалізації. Це зумовлено у першу чергу більш простим механізмом взаємодії потоків.
2. При написанні програми для системи з локальною пам’яттю велика вірогідність зробити помилку у механізмі передачі повідомлень. Такі помилки можливо виявити тільки на етапі часу виконування.

# Тестування програмного забезпечення

Тестування програмного забезпечення виконувалось методом заміру часу виконання програми при різних розмінностях матриць і кількості процесорів, після чого за отриманими результатами підраховувались коефіцієнти прискорення() та ефективності().

Тестування проводилося на системі з наступними характеристиками:

**Центральний процесор**: Intel Core 2 Quad Q6600, 2400 MHz

**Оперативна пам’ять:** 2х1024Мб

**Операційна система**: Microsoft Windows 7 Home x86

Для замірів часу використовувались функції зі стандартних бібліотек DateTime в C# та System в Java.



## Тестування ПЗ для системи з спільною пам’яттю

Таблиця 3.1Час виконання програми обчислення на системі зі СП

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N P | 1 | 2 | 3 | 4 |
| 900 | 27,2 | 14,3 | 10,4 | 7,3 |
| 1500 | 133,2 | 68,5 | 58,6 | 34,3 |
| 2400 | 561,2 | 284,2 | 214,1 | 152 |

Таблиця 3.2 Значення коефіцієнтів прискорення для системи с СП

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N P | 1 | 2 | 3 | 4 |
| 900 | 1 | 1,96479 | 2,51351 | 3,6856 |
| 1500 | 1 | 1,97301 | 2,6163 | 3,72805 |
| 2400 | 1 | 1,97464 | 2,62019 | 3,83803 |

Таблиця 3.3 Значення коефіцієнтів ефективності для системи с СП

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N P | 1 | 2 | 3 | 4 |
| 900 | 1 | 0,98239 | 0,83784 | 0,9214 |
| 1500 | 1 | 0,98651 | 0,8721 | 0,93201 |
| 2400 | 1 | 0,98732 | 0,8734 | 0,95951 |

Рис. 3.1 Залежність коефіцієнту прискорення від кількості ЦП для систем з СП

Рис. 3.2 Залежність коефіцієнту прискорення від кількості ЦП для систем з СП для N = 900

Рис. 3.3 Залежність коефіцієнту прискорення від кількості ЦП для систем з СП для N = 1500

Рис. 3.4 Залежність коефіцієнту прискорення від кількості ЦП для систем з СП для N = 2400

Рис. 3.5 Залежність коефіцієнту ефективності від кількості ЦП для систем з СП

Рис. 3.6 Залежність коефіцієнту ефективності від кількості ЦП для систем з СП для N = 900

Рис. 3.7 Залежність коефіцієнту ефективності від кількості ЦП для систем з СП для N = 1500

Рис. 3.8 Залежність коефіцієнту ефективності від кількості ЦП для систем з СП для N = 2400

## Тестування ПЗ для системи з локальною пам’яттю

Таблиця 3.4 Час виконання програми обчислення на системі зі ЛП

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N P | 1 | 2 | 3 | 4 |
| 900 | 32,6 | 26,8 | 14,2 | 9,34 |
| 1500 | 256,4 | 158,5 | 99,3 | 71,1 |
| 2400 | 683,21 | 380,7 | 250,03 | 180,17 |

Коефіцієнт прискорення обраховується за формулою [17]:

Таблиця 3.1 Значення коефіцієнтів прискорення для системи с ЛП

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | P | | | |
|  | 1 | 2 | 3 | 4 |
| 900 | 1 | 1,216418 | 2,295775 | 3,490364 |
| 1500 | 1 | 1,617666 | 2,582075 | 3,606188 |
| 2400 | 1 | 1,794615 | 2,732512 | 3,79203 |

Коефіцієнт ефективності обраховується за формулою [17]:

Таблиця 3.2 Значення коефіцієнтів ефективності для системи с ЛП

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | P | | | |
| 1 | 2 | 3 | 4 |
| 900 | 1 | 0,990558 | 0,985682 | 0,96285 |
| 1500 | 1 | 0,962391 | 0,94544 | 0,980886 |
| 2400 | 1 | 0,860929 | 0,997702 | 0,974579 |

Рис. 3.9 Залежність коефіцієнту прискорення від кількості ЦП для систем з ЛП

Рис. 3.2 Залежність коефіцієнту прискорення від кількості ЦП для систем з ЛП для N = 900

Рис. 3.3 Залежність коефіцієнту прискорення від кількості ЦП для систем з ЛП для N = 1500

Рис. 3.4 Залежність коефіцієнту прискорення від кількості ЦП для систем з ЛП для N = 2400

Рис. 3.6 Залежність коефіцієнту ефективності від кількості ЦП для систем з ЛП

Рис. 3.7 Залежність коефіцієнту ефективності від кількості ЦП для систем з ЛП

Рис. 3.8 Залежність коефіцієнту ефективності від кількості ЦП для систем з ЛП

Рис. 3.9 Залежність коефіцієнту ефективності від кількості ЦП для систем з ЛП

## Висновки до Розділу 3

1. Коефіцієнт прискорення у системі зі спільною пам’яттю прямую до максимального зі збільшенням N, це означає що майже усі дії виконуються паралельно, і це дійсно так, адже зі збільшенням N час на копіювання залишається незмінним, а усі обчислення виконуються паралельно. На мал. 3.1 зображено графік залежності коефіцієнта прискорення від кількості ядер, що показує лінійність коефіцієнта прискорення. Схожа ситуація і в системі з локальною пам’яттю, що видно на мал.3.9.
2. Коефіцієнт ефективності в обох системах спадає зі збільшенням кількості ядер. На мал. 3.5 і 3.13 видно, що з кожним наступним додаванням ядер, коефіцієнт ефективності спадає. Але коли кількість ядер збільшується до 4, коефіцієнт різко зростає. Це може бути пов’язано з тим, що процесорні ядра з’єднані попарно, і передача даних між попарними ядрами швидша ніж передача між парами ядер.
3. Коефіцієнт ефективності близький до одиниці, що свідчить про якісний паралельний алгоритм для задач, і вказує на гарний потенціал масштабування таких систем.
4. По часу виконання явну перевагу має система зі спільною пам’яттю. Основна причина цього саме матричне рівняння, при обчисленні якого потрібна синхронізація по проміжному результату. В системі з локальною пам’яттю вона складніше реалізована, тому триває довше.

# Висновки

1. Було розглянуто поняття процесу і потоку. Показано, що кожен потік має свою пам'ять і стек, де він зберігає свої дані і різні поведінки процесів. Був виконаний огляд засобів та можливостей паралельного програмування в бібліотеці Win32 та POSIX.
2. Система зі спільною пам’яттю була реалізована за допомогою C# засобів паралельного програмування. Система з локальною пам’яттю реалізована за допомогою MPI.
3. Варіант зі спільною пам’яттю є більш простим у розробці та реалізації. Це зумовлено у першу чергу більш простим механізмом взаємодії потоків. Показано, що мова C# має дуже зручні інструменти організації паралельних обчислень. Прозорий механізм синхронізованих методів, а також об’єктно-орієнтований підхід значно спрощують реалізацію кінцевої програми.
4. У системах з локальною пам’ятю немає критичних ділянок, але є велика вірогідність створити тупикову ситуацію, коли обидва процеси намагаються передати данні один одному. Такі помилки можливо виявити тільки на етапі часу виконання, і дуже не легко налагодити таку програму, і потрібно проектувати алгоритм програми так, щоб потім уникнути можливості появи тупикових ситуацію.
5. Було проведення тестування на швидкість виконання поставленого математичного завдання, обраховані коефіцієнти прискорення і ефективності, після чого стало видно що система зі спільною пам’яттю переважає над системою із локальною пам’яттю у швидкості. Було показано, що розроблене програмне забезпечення має добрий потенціал масштабованості і майже не залежить від обсягів даних які воно оброблює.

# Література

x

|  |  |
| --- | --- |
| 1 | Albahari J., Albahari B. C# 3.0 in a Nutshell, Third Edition: A Desktop Quick Reference. O'Reilly Media, 2007. 864 pp. |
| 2 | Albahari.com. Threading in C# [З мережі] URL: http:/​/​www.albahari.com/​threading/ |
| 3 | Intuit.ru. Средства синхронизации [З мережі] URL: http:/​/​www.intuit.ru/​department/​supercomputing/​parallcsharp/​5/ |
| 4 | MSDN. System.Threading - пространство имен // MSDN. URL: http:/​/​msdn.microsoft.com/​ru-ru/​library/​system.threading.aspx |
| 5 | MSDN. Threading [З мережі] // MSDN: [сайт]. URL: http:/​/​msdn.microsoft.com/​ru-ru/​library/​orm-9780596527570-03-19.aspx#Россия (Pусский) |
| 6 | MSDN. Threading (C# Programming Guide) [З мережі] // MSDN: [сайт]. URL: http:/​/​msdn.microsoft.com/​ru-ru/​library/​ms173178(v=VS.80).aspx |
| 7 | Oracle.com. Synchronized Methods [З мережі] URL: http:/​/​docs.oracle.com/​javase/​tutorial/​essential/​concurrency/​syncmeth.html |
| 8 | RSDN. Работа с потоками в C# [З мережі] // RSDN: [сайт]. URL: http:/​/​www.rsdn.ru/​article/​dotnet/​CSThreading1.xml |
| 9 | Wikimedia Foundation..NET\_Framework [З мережі] // Wikipedia: [сайт]. URL: http:/​/​ru.wikipedia.org/​wiki/​.NET\_Framework |
| 10 | Wikimedia Foundation. Ada Programming/Tasking [З мережі] // Wikibooks: [сайт]. URL: http:/​/​en.wikibooks.org/​wiki/​Ada\_Programming/​Tasking |
| 11 | Wikimedia Foundation. Common Language Runtime [З мережі] // Wikipedia: [сайт]. URL: http:/​/​uk.wikipedia.org/​wiki/​Common\_Language\_Runtime |
| 13 | Г. Ш. С# Учебный курс. С-П.: "Питер", 2003. 500 pp. |
| 14 | Жуков І.А. К.О.В. Паралельні та розподілені обчислення. Київ: Корнійчук, 2005. 226 pp. |

x

# Додатки

## Схеми алгоритмів процесів

## Схеми алгоритмів основних програм

## Лістинг програм

### Лістинг програми для системи зі спільною пам’яттю

/\*

\* Kursova P2

\* Bovsunivsky Oleg

\* Group IO-01

\* Task: Ah = Bh + C \* (MO \* ( MXh \* MZ))

\*/

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Threading;

namespace pro2\_lab3\_tar {

class Program {

private const int n = 600;

private const int p = 4;

private const int h = n / p;

private static int[] A = new int[n];//out

private static int[] B = new int[n];

private static int[] C = new int[n];

private static int[][] MO = new int[n][];

private static int[][] MZ = new int[n][];

private static int[][] MX = new int[n][];

private static Mutex mutex = new Mutex(false);

private static Semaphore semaphore = new Semaphore(0, 3);

private static EventWaitHandle event1 = new ManualResetEvent(false);

private static EventWaitHandle event2 = new ManualResetEvent(false);

private static EventWaitHandle event3 = new ManualResetEvent(false);

private static EventWaitHandle event4 = new ManualResetEvent(false);

private static EventWaitHandle event5 = new ManualResetEvent(false);

private static object lockObject = new object();

//time

static DateTime startTime;

private static void Task1() {

Console.WriteLine("Task1 started\n");

int[] C1 = new int[n];

int[][] MO1 = new int[n][];

int[][] MZ1 = new int[n][];

// Ожидание сигнал от T2 о завершении ввода.

semaphore.WaitOne();

// Копия MO1 = MO, MZ1 = MZ

mutex.WaitOne();

for (int i = 0; i < n; i++) {

MO1[i] = new int[n];

MZ1[i] = new int[n];

for (int j = 0; j < n; j++) {

MO1[i][j] = MO[i][j];

MZ1[i][j] = MZ[i][j];

}

}

mutex.ReleaseMutex();

// Копия С1 = С

lock (lockObject) {

for (int i = 0; i < n; i++) {

C1[i] = C[i];

}

}

// Счет AH = BH + C1 \* ( MO1 \* ( MZ1 \* MXH) )

int[][] temp1 = new int[n][];

for (int i = 0; i < h; i++) {

temp1[i] = new int[n];

for (int j = 0; j < n; j++) {

temp1[i][j] = 0;

for (int k = 0; k < n; k++) {

temp1[i][j] += MZ1[k][j] \* MX[i][k];

}

}

}

int[][] temp2 = new int[n][];

for (int i = 0; i < h; i++) {

temp2[i] = new int[n];

for (int j = 0; j < n; j++) {

temp2[i][j] = 0;

for (int k = 0; k < n; k++) {

temp2[i][j] += MO1[k][j] \* temp1[i][k];

}

}

}

for (int i = 0; i < h; i++) {

A[i] = 0;

for (int j = 0; j < n; j++) {

A[i] += C1[j] \* temp2[i][j];

}

A[i] += B[i];

}

// Сигнал Т2 о завершении счета

event1.Set();

Console.WriteLine("\nTask1 Finished");

}

private static void Task2() {

Console.WriteLine("Task2 started\n");

int[] C2 = new int[n];

int[][] MO2 = new int[n][];

int[][] MZ2 = new int[n][];

// Ввод MX, B, C

for (int i = 0; i < n; i++) {

MX[i] = new int[n];

B[i] = 1;

C[i] = 1;

for (int j = 0; j < n; j++) {

MX[i][j] = 1;

}

}

// Ожидание сигнал от Т3 о завершении ввода

event4.WaitOne();

// Ожидание сигнал от Т4 о завершении ввода

event5.WaitOne();

// Сигнал T1, T3, T4 о завершении ввода

semaphore.Release(3);

// Копия MO2 = MO, MZ2 = MZ

mutex.WaitOne();

for (int i = 0; i < n; i++) {

MO2[i] = new int[n];

MZ2[i] = new int[n];

for (int j = 0; j < n; j++) {

MO2[i][j] = MO[i][j];

MZ2[i][j] = MZ[i][j];

}

}

mutex.ReleaseMutex();

// Копия С2 = С

lock (lockObject) {

for (int i = 0; i < n; i++) {

C2[i] = C[i];

}

}

// Счет AH = BH + C1 \* ( MO1 \* ( MZ1 \* MXH) )

int[][] temp1 = new int[n][];

for (int i = h; i < 2 \* h; i++) {

temp1[i] = new int[n];

for (int j = 0; j < n; j++) {

temp1[i][j] = 0;

for (int k = 0; k < n; k++) {

temp1[i][j] += MZ2[k][j] \* MX[i][k];

}

}

}

int[][] temp2 = new int[n][];

for (int i = h; i < 2 \* h; i++) {

temp2[i] = new int[n];

for (int j = 0; j < n; j++) {

temp2[i][j] = 0;

for (int k = 0; k < n; k++) {

temp2[i][j] += MO2[k][j] \* temp1[i][k];

}

}

}

for (int i = h; i < 2 \* h; i++) {

A[i] = 0;

for (int j = 0; j < n; j++) {

A[i] += C2[j] \* temp2[i][j];

}

A[i] += B[i];

}

// Ожидание сигнал от Т1 о завершении счетa

event1.WaitOne();

// Ожидание сигнал от Т3 о завершении счетa

event2.WaitOne();

// Ожидание сигнал от Т4 о завершении счетa

event3.WaitOne();

// Вывод

Console.WriteLine("\n");

if (n < 10) {

for (int i = 0; i < n; i++) {

Console.WriteLine(A[i]);

}

}

Console.WriteLine("\nTask2 Finished");

Console.WriteLine("time=" + DateTime.Now.Minute + ":" + DateTime.Now.Second + ":" + DateTime.Now.Millisecond);

DateTime diff = new DateTime((DateTime.Now - startTime).Ticks);

Console.WriteLine("difference=" + diff.Minute + ":" + diff.Second + ":" + diff.Millisecond);

}

private static void Task3() {

Console.WriteLine("Task3 started\n");

int[] C3 = new int[n];

int[][] MO3 = new int[n][];

int[][] MZ3 = new int[n][];

// Ввод MO

for (int i = 0; i < n; i++) {

MO[i] = new int[n];

for (int j = 0; j < n; j++) {

MO[i][j] = 1;

}

}

// Сигнал Т2 о завершении ввода

event4.Set();

// Ожидание сигнал от Т2 о завершении ввода.

semaphore.WaitOne();

// Копия MO3 = MO, MZ3 = MZ

mutex.WaitOne();

for (int i = 0; i < n; i++) {

MO3[i] = new int[n];

MZ3[i] = new int[n];

for (int j = 0; j < n; j++) {

MO3[i][j] = MO[i][j];

MZ3[i][j] = MZ[i][j];

}

}

mutex.ReleaseMutex();

// Копия С3 = С

lock (lockObject) {

for (int i = 0; i < n; i++) {

C3[i] = C[i];

}

}

// Счет AH = BH + C3 \* ( MO3 \* ( MZ3 \* MXH) )

int[][] temp1 = new int[n][];

for (int i = 2 \* h; i < 3 \* h; i++) {

temp1[i] = new int[n];

for (int j = 0; j < n; j++) {

temp1[i][j] = 0;

for (int k = 0; k < n; k++) {

temp1[i][j] += MZ3[k][j] \* MX[i][k];

}

}

}

int[][] temp2 = new int[n][];

for (int i = 2 \* h; i < 3 \* h; i++) {

temp2[i] = new int[n];

for (int j = 0; j < n; j++) {

temp2[i][j] = 0;

for (int k = 0; k < n; k++) {

temp2[i][j] += MO3[k][j] \* temp1[i][k];

}

}

}

for (int i = 2 \* h; i < 3 \* h; i++) {

A[i] = 0;

for (int j = 0; j < n; j++) {

A[i] += C3[j] \* temp2[i][j];

}

A[i] += B[i];

}

// Сигнал Т2 о завершении счета

event2.Set();

Console.WriteLine("\nTask3 Finished");

}

private static void Task4() {

Console.WriteLine("Task4 started\n");

int[] C4 = new int[n];

int[][] MO4 = new int[n][];

int[][] MZ4 = new int[n][];

// Ввод MZ

for (int i = 0; i < n; i++) {

MZ[i] = new int[n];

for (int j = 0; j < n; j++) {

MZ[i][j] = 1;

}

}

// Сигнал Т2 о завершении ввода

event5.Set();

// Ожидание сигнал от Т2 о завершении ввода.

semaphore.WaitOne();

// Копия MO4 = MO, MZ4 = MZ

mutex.WaitOne();

for (int i = 0; i < n; i++) {

MO4[i] = new int[n];

MZ4[i] = new int[n];

for (int j = 0; j < n; j++) {

MO4[i][j] = MO[i][j];

MZ4[i][j] = MZ[i][j];

}

}

mutex.ReleaseMutex();

// Копия С4 = С

lock (lockObject) {

for (int i = 0; i < n; i++) {

C4[i] = C[i];

}

}

// Счет AH = BH + C4 \* ( MO4 \* ( MZ4 \* MXH) )

int[][] temp1 = new int[n][];

for (int i = 3 \* h; i < n; i++) {

temp1[i] = new int[n];

for (int j = 0; j < n; j++) {

temp1[i][j] = 0;

for (int k = 0; k < n; k++) {

temp1[i][j] += MZ4[k][j] \* MX[i][k];

}

}

}

int[][] temp2 = new int[n][];

for (int i = 3 \* h; i < n; i++) {

temp2[i] = new int[n];

for (int j = 0; j < n; j++) {

temp2[i][j] = 0;

for (int k = 0; k < n; k++) {

temp2[i][j] += MO4[k][j] \* temp1[i][k];

}

}

}

for (int i = 3 \* h; i < n; i++) {

A[i] = 0;

for (int j = 0; j < n; j++) {

A[i] += C4[j] \* temp2[i][j];

}

A[i] += B[i];

}

// Сигнал Т2 о завершении счета

event3.Set();

Console.WriteLine("\nTask4 Finished");

}

static void Main(string[] args) {

startTime = DateTime.Now;

Console.WriteLine("time=" + DateTime.Now.Minute + ":" + DateTime.Now.Second + ":" + DateTime.Now.Millisecond);

Thread thread1 = new Thread(Task1);

Thread thread2 = new Thread(Task2);

Thread thread3 = new Thread(Task3);

Thread thread4 = new Thread(Task4);

thread1.Start();

thread2.Start();

thread3.Start();

thread4.Start();

Console.ReadKey();

}

}

}

### Лістинг програми для системи з локальною пам’яттю

import mpi.\*;

import java.util.Date;

public class MPILab8 {

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

MPI.Init(args);

int rank = MPI.COMM\_WORLD.Rank();

System.out.println("Start thread " + rank);

int size = MPI.COMM\_WORLD.Size();

int P = 4; //(кол-во задач)вводится с консоли при запуске приложения.

int N = 4; //ранг матрицы

int H = N/P; //коеф. разбиения матрицы(целая часть для одной задачи)

Date date = new java.util.Date();

long startTime = date.getTime();

int currentH = 0;

if(rank == 1){

currentH=N;

}else if(rank == 2) {

currentH=2\*H;

}else{

currentH=H;

}

int[] Ah = new int[currentH];

int[] Bh = new int[currentH];

int[] C = new int[N];

int[] MXh = new int[currentH \* N];

int[] MO = new int[N \* N];

int[] MZ = new int[N \* N];

//ВВОД И РАССПРОСТРАНЕНИЕ ДАННЫХ

if (rank == 0) {

//receive Bh,MXh,C,MZ,MO from T1

MPI.COMM\_WORLD.Recv(MZ, 0, N\*N, MPI.INT, 1, 1);

MPI.COMM\_WORLD.Recv(MO, 0, N\*N, MPI.INT, 1, 1);

MPI.COMM\_WORLD.Recv(C, 0, N, MPI.INT, 1, 1);

MPI.COMM\_WORLD.Recv(MXh, 0, H \* N, MPI.INT, 1, 1);

MPI.COMM\_WORLD.Recv(Bh, 0, H, MPI.INT, 1, 1);

}else if (rank == 1) {

//ввод данных

for (int i = 0; i < N; i++) {

Bh[i] = 1;

C[i] = 1;

for (int j = 0; j < N; j++) {

MXh[i\*N + j] = 1;

}

}

//передача-прием данных

MPI.COMM\_WORLD.Send(C, 0, N, MPI.INT, 2, 1);

MPI.COMM\_WORLD.Send(MXh, 2\*H\*N, 2\*H \* N, MPI.INT, 2, 1);

MPI.COMM\_WORLD.Send(Bh, 2\*H, 2\*H, MPI.INT, 2, 1);

MPI.COMM\_WORLD.Recv(MZ, 0, N\*N, MPI.INT, 2, 1);

MPI.COMM\_WORLD.Recv(MO, 0, N\*N, MPI.INT, 2, 1);

MPI.COMM\_WORLD.Send(MZ, 0, N\*N, MPI.INT, 0, 1);

MPI.COMM\_WORLD.Send(MO, 0, N\*N, MPI.INT, 0, 1);

MPI.COMM\_WORLD.Send(C, 0, N, MPI.INT, 0, 1);

MPI.COMM\_WORLD.Send(MXh, H\*N, H \* N, MPI.INT, 0, 1);

MPI.COMM\_WORLD.Send(Bh, H, H, MPI.INT, 0, 1);

}else if (rank == 2) {

//ввод данных

for (int i = 0; i < N\*N; i++) {

MO[i] = 1;

}

//передача-прием данных

MPI.COMM\_WORLD.Recv(MZ, 0, N\*N, MPI.INT, 3, 1);

MPI.COMM\_WORLD.Recv(C, 0, N, MPI.INT, 1, 1);

MPI.COMM\_WORLD.Recv(MXh, 0, 2\*H \* N, MPI.INT, 1, 1);

MPI.COMM\_WORLD.Recv(Bh, 0, 2\*H, MPI.INT, 1, 1);

MPI.COMM\_WORLD.Send(MO, 0, N\*N, MPI.INT, 3, 1);

MPI.COMM\_WORLD.Send(C, 0, N, MPI.INT, 3, 1);

MPI.COMM\_WORLD.Send(MXh, H\*N, H \* N, MPI.INT, 3, 1);

MPI.COMM\_WORLD.Send(Bh, H, H, MPI.INT, 3, 1);

MPI.COMM\_WORLD.Send(MZ, 0, N\*N, MPI.INT, 1, 1);

MPI.COMM\_WORLD.Send(MO, 0, N\*N, MPI.INT, 1, 1);

}else if (rank == 3) {

//ввод данных

for (int i = 0; i < N\*N; i++) {

MZ[i] = 1;

}

//передача-прием данных

MPI.COMM\_WORLD.Send(MZ, 0, N\*N, MPI.INT, 2, 1);

MPI.COMM\_WORLD.Recv(MO, 0, N\*N, MPI.INT, 2, 1);

MPI.COMM\_WORLD.Recv(C, 0, N, MPI.INT, 2, 1);

MPI.COMM\_WORLD.Recv(MXh, 0, H \* N, MPI.INT, 2, 1);

MPI.COMM\_WORLD.Recv(Bh, 0, H, MPI.INT, 2, 1);

}

//счет

//buf=MXh\*MZ

int[] tmpMh = new int[currentH\*N];

for (int z = 0; z < H; z++) {

for (int i = 0; i < N; i++) {

tmpMh[z\*N + i] = 0;

for (int j = 0; j < N; j++) {

tmpMh[z\*N + i] += MXh[z\*N + j] \* MZ[j\*N + i];

}

}

}

//MXh=buf\*MO

for (int z = 0; z < H; z++) {

for (int i = 0; i < N; i++) {

MXh[z\*N + i] = 0;

for (int j = 0; j < N; j++) {

MXh[z\*N + i] += tmpMh[z\*N + j] \* MO[j\*N + i];

}

}

}

//tmpVh=MXh\*C

int[] tmpVh = new int[currentH];

for (int z = 0; z < H; z++) {

tmpVh[z] = 0;

for (int j = 0; j < N; j++) {

tmpVh[z] += MXh[z\*N + j] \* C[j];

}

}

//Ah=Bh+tmpVh

for (int j = 0; j < H; j++) {

Ah[j] += Bh[j] + tmpVh[j];

}

//ОТПРАВЛЕНИЕ РЕЗУЛЬТАТА

if (rank == 0) {

MPI.COMM\_WORLD.Send(Ah, 0, H, MPI.INT, 1, 1);

}else if (rank == 1) {

MPI.COMM\_WORLD.Recv(Ah, H, H, MPI.INT, 0, 1);

MPI.COMM\_WORLD.Recv(Ah, 2\*H, 2\*H, MPI.INT, 2, 1);

//output

if(N<9){

for (int z = 0; z < N; z++) {

System.out.println(String.format("A[%d]=", z) + Ah[z]);

}

}

}else if (rank == 2) {

MPI.COMM\_WORLD.Recv(Ah, H, H, MPI.INT, 3, 1);

MPI.COMM\_WORLD.Send(Ah, 0, 2\*H, MPI.INT, 1, 1);

}else if (rank == 3) {

MPI.COMM\_WORLD.Send(Ah, 0, H, MPI.INT, 2, 1);

}

System.out.println("Finish thread " + rank);

if (rank == 1) {

date = new java.util.Date();

long finishTime = date.getTime();

long time = finishTime - startTime;

System.out.println("N = " + N + " P=" + P + " Work time: " + time + " ms");

}

MPI.Finalize();

}

}