CHAPTER 4 多執行緒 (Multithreaded Programming)

林政宏
國立臺灣師範大學電機工程學系

CHAPTER 4 多執行緒

- 4.1 概論
- 4.2 多核心程式(Multicore Programming)
- 4.3 多執行緒模式(Multithreading Models)
- 4.4 執行緒庫(Thread Libraries)
- 4.5 隱性執行緒(Implicit Threading)
- 4.6 執行緒的事項
- 4.7 作業系統範例

Multicore Programming

- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**

- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency
- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation

4.1 概論

- 4.1 概論
  - 執行緒(Thread)是 CPU使用時的一個基本單位，它是由一個**執行緒ID**、**程式計數器**、一組**暫存器**，以及一個**堆疊**空間所組成。

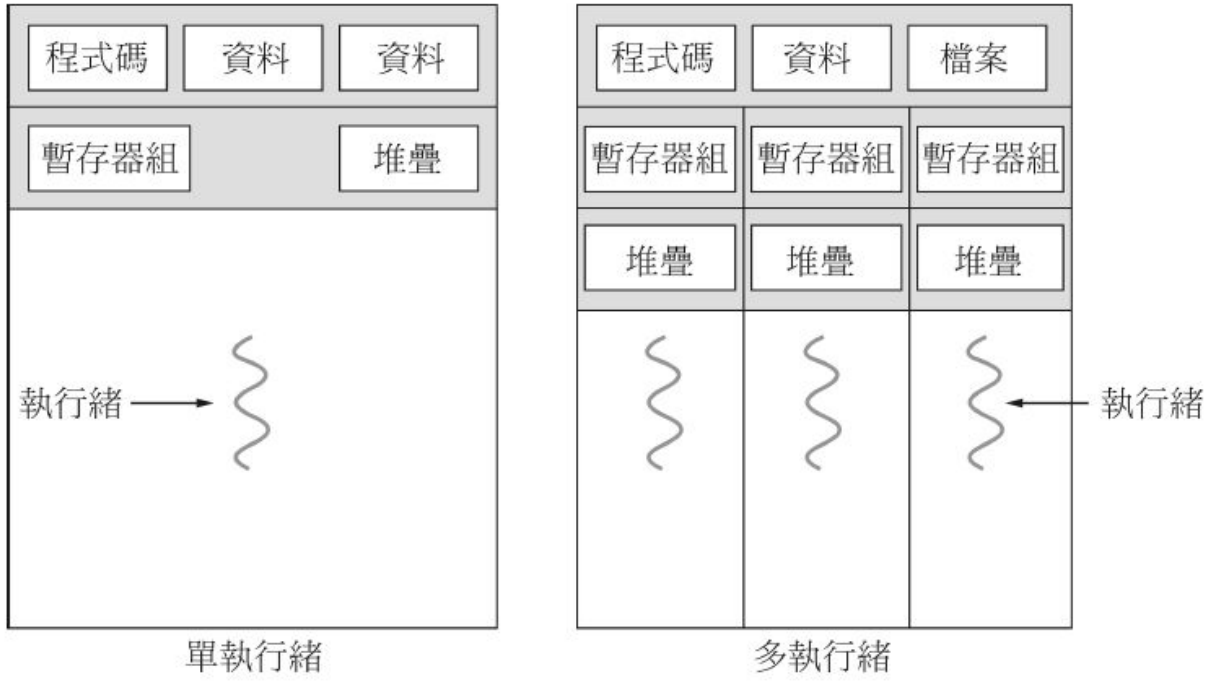
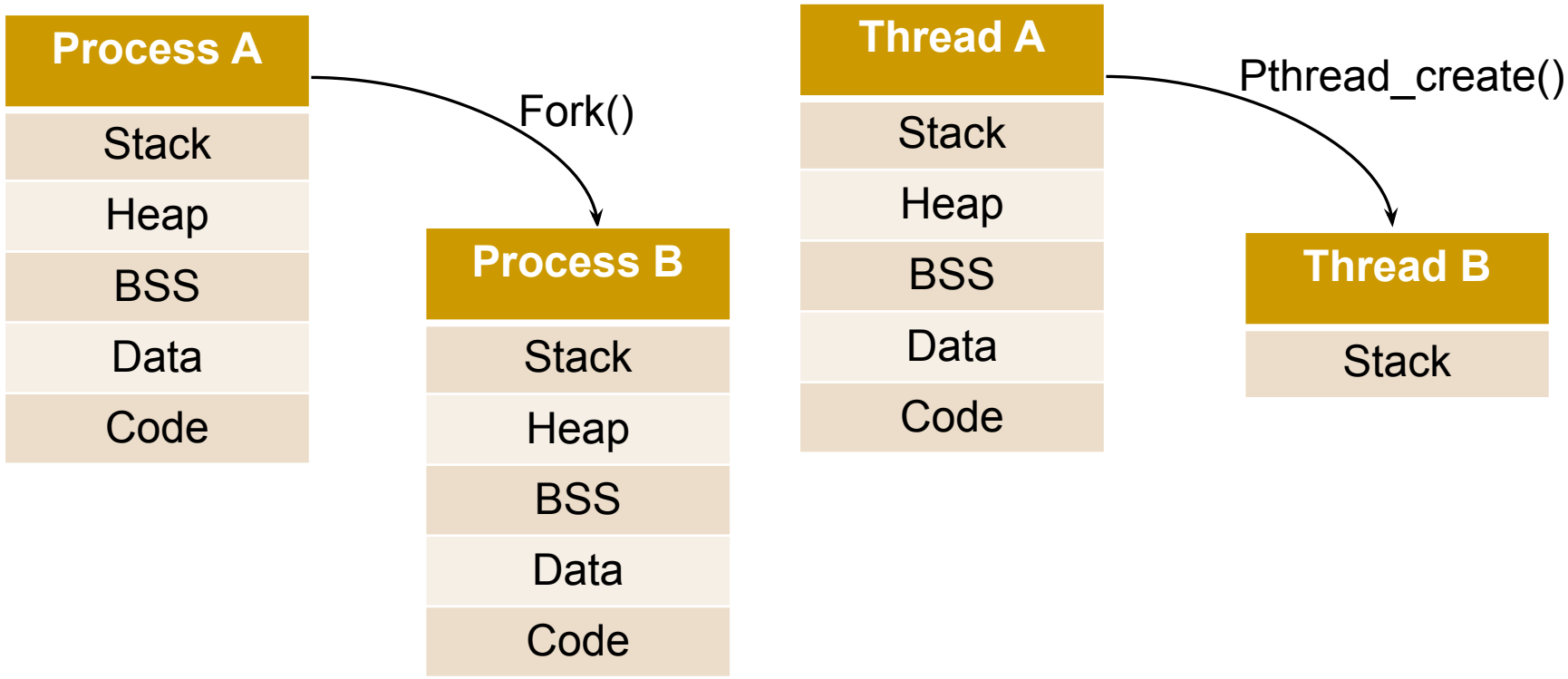


圖 4.1 單執行緒和多執行緒的行程

Difference between creating a process and a thread



- Process A and Process B do not share resources.
- Thread B only applies its own stack memory.
- Thread B shares code, data, BSS, heap, files with thread A.

4.1.1 動 機

- 許多在桌上型PC執行的套裝軟體都是多執行緒。
- 應用程式通常都製作成有許多執行緒控制的個別行程。
- 網頁瀏覽器可能有一個執行緒顯示影像或文字，而另一執行緒則從網路擷取資料。
- 文書處理器可能有一個執行緒在顯示圖形，另一個執行緒從使用者讀入按鍵，而第三個執行緒在背景下執行拼字和文法校正。

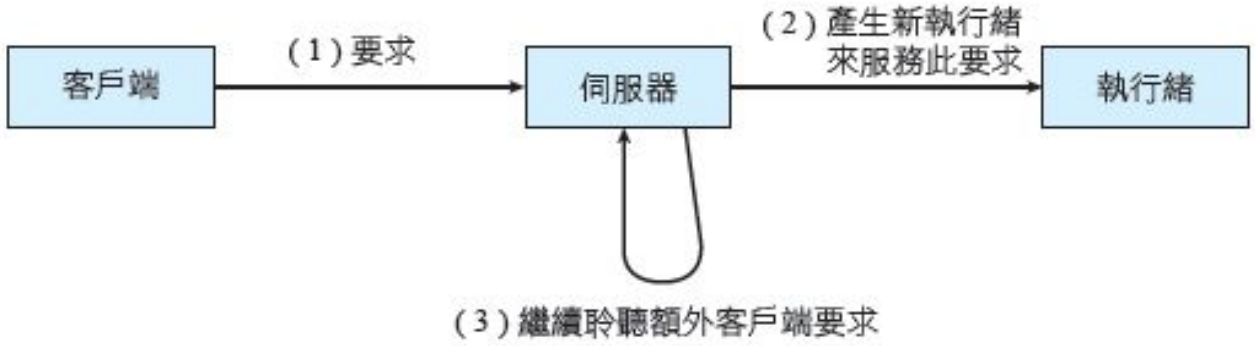


圖4,2 多執行緒伺服器架構

4.1.2 利 益

撰寫多執行緒程式的好處可以分成四個主要類別:

1. 應答(Responsiveness): 將交談式的應用程式多執行緒化, 可以在一個程式某一部份被暫停, 或程式在執行冗長操作時, 依然持續執行, 因此增加了對使用者的應答。

2. 資源分享(Resource Sharing): 執行緒間將共用它們所屬行程的記憶體和資源。程式碼和資料共用的好處是讓應用程式有數個不同的執行緒在同一位址空間活動。

3. 經濟(Economy) : 對於行程產生所配置的記憶體和資源耗費很大。反之, 因為執行緒共用它們所屬行程的資源, 所以執行緒的產生和內容交換就比較經濟。憑經驗去測量出產生和維護行程比執行緒多出多少時間可能很困難, 但通常產生和維護行程會比執行緒更費時。

4. 可擴展性(Scalability): 在多處理器的架構下, 多執行緒的利益可以大幅提升, 因為每一執行緒可以並行地在不同的處理器上執行。不論有多少CPU可以使用, 單一執行緒只能在一個CPU上執行。多處理器上並行增加多執行緒。

4.1.3 多核心程式的挑戰

在位多核心系統編寫程式中目前的挑戰有以下五個領域：

- 1.切割活動(Dividing activities)：檢查應用程式來找出可以被切割成個別的、同時發生的任務，因此可以在個別的核心上平行地執行。
- 2.平衡(Balance)：當識別任務可以平行地執行時，程式員也必須保證任務執行為相等的工作。
- 3.資料分裂(Data splitting)：正如同應用程式被分割成個別的任務，藉由任務來存取和運用的資料必須被分割到個別的核心上執行。
- 4.資料相依性(Data dependency)：藉由任務存取的資料必須在兩個或多個任務之間檢查其相依性。在一個任務依靠另一個任務的情況下，程式員必須確認任務的執行與資料的相依性是同步的。
- 5.測試與除錯(*Testing and debugging*)：當一個程式在多核心上平行地執行時，有許多不同的執行路徑。測試和除錯這類同步的程式原來就比測試和除錯單一執行緒的應用程式更加困難。

圖4.3 在單一核心系統同步執行

圖 4.4 在多重核心系統平行執行

Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- *S* is serial portion
- *N* processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- i.e. if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As *N* approaches infinity, speedup approaches 1 / *S*
  Serial portion of an application has disproportionate effect on performance gained by adding additional cores
- But does the law take into account contemporary multicore systems?

Gustafson's Law

- proposed by John L. Gustafson in 1988
- $Speedup\ (N) = s + p \cdot N$

$$= s + (1-s)N$$

$$= N - s \cdot (N-1)$$

  - *Speedup*(*N*) is the speedup achievable by scaling up the problem size when using *N* processors.
  - *N* is the number of processors.
  - *s* and *p* are the fractions of time spent executing the serial parts and the parallel parts. $s + p = 1$

Gustafson's Law

- **Focus on Scalability**: Gustafson's Law argues that as the number of processors increases, the problem size can be scaled up to accommodate the larger resources available. This approach aims to maintain a balanced workload across all processors.
- **Different Perspective**: Unlike Amdahl's Law, which focuses on the fixed problem size and the impact of sequential portions, Gustafson's Law focuses on exploiting parallelism by scaling the problem size to match the available resources.
- **Emphasis on Real-world Applications**: Gustafson's Law is often considered more relevant for real-world parallel computing scenarios where the problem size can be adjusted to fit the available parallelism. This is particularly applicable in areas such as scientific computing, where larger datasets or simulations can be used to take advantage of parallel resources.

User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Win32 threads
  - Java threads

- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

User Threads

- **User threads** are managed entirely by the **user-space** application or the **runtime library**, without direct intervention from the operating system kernel.
- These threads are **lightweight** and are generally faster to create and manage because they don't require kernel involvement.
- User threads are implemented using threading libraries or frameworks provided by the programming language or the operating system.
- Examples of user-level threading libraries include **POSIX Threads** (**pthread**) in Unix-like systems and **Win32 Threads** in Windows.

Kernel Threads

- **Kernel threads** are managed and scheduled by the **operating system** kernel.
- They are more **heavyweight** compared to user threads because they involve kernel resources and context switches.
- Kernel threads are generally slower to create and manage than user threads due to the involvement of the kernel.
- They provide benefits such as better support for multi-core processors, efficient blocking I/O operations, and preemptive multitasking.
- Kernel threads are typically used for tasks that require direct access to kernel resources or privileged operations, such as device drivers, file system operations, and system-level services.

多執行緒模式

- 多對一模式(**Many-to-one** model)
  - Many user-level threads are multiplexed onto a single kernel-level thread
  - One thread blocking causes all to block
  - Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time

- Few systems currently use this model because it cannot take advantage of multiple processing cores.

- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**

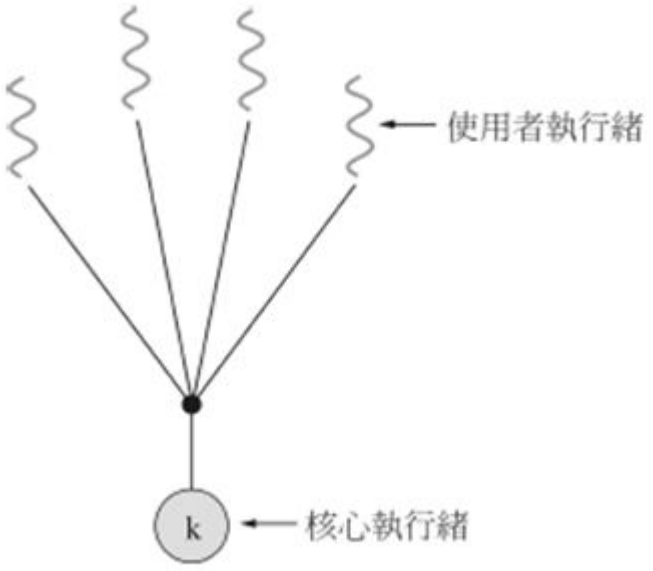


圖 4.5 多對一模式

一對一模式(One-to-One)

- Map each user thread to a kernel thread
- Creating a user-level thread requires creating a kernel thread
- Supporting multiprocessors
- The number of threads is restricted due to the overhead of creating kernel threads.

- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

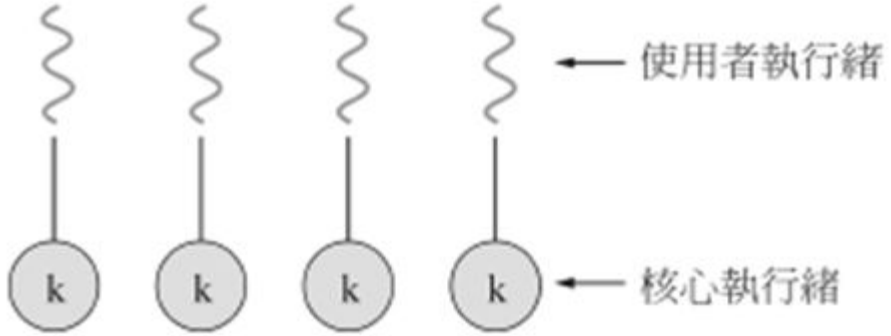


圖 4.6 一對一模式

多對多模式(Many-to-Many)

- Multiplexes many user-level threads to a smaller of equal number of kernel threads
- The number of kernel threads may be specific to a particular application or a particular machine.
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package

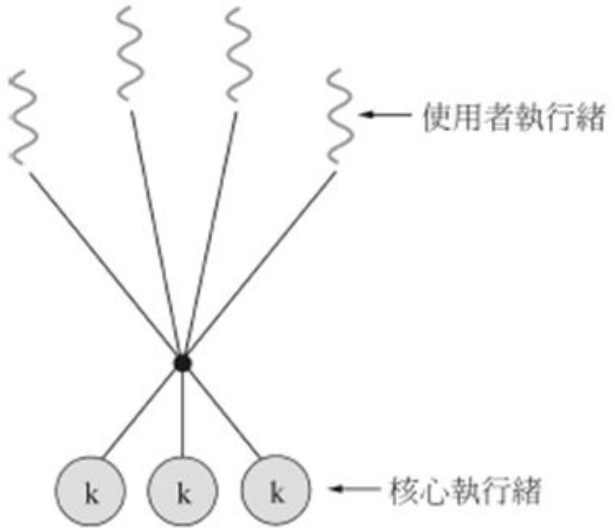


圖 4.7 多對多模式

Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to a kernel thread

- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier

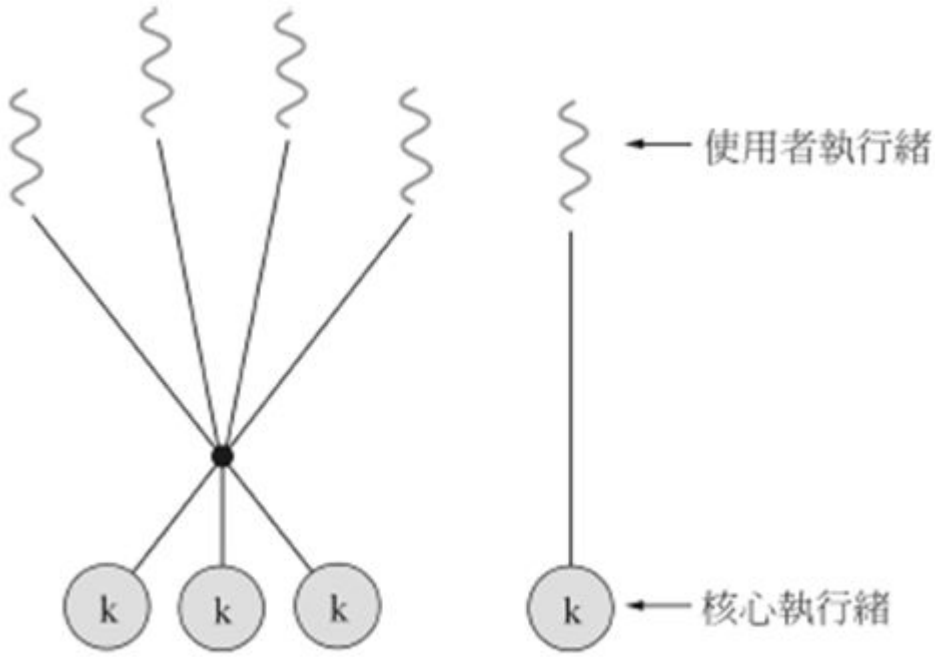


圖 4.8 二層模式

4.4 執行緒的函式庫

- **Thread library** provides programmer with API for creating and managing threads

- Two primary ways of implementing
  - Library entirely in user space with no kernel support
  - Kernel-level library supported by the OS

4.4 執行緒程式庫(Thread Libraries)

- 4.3.1 Pthreads
  - Pthreads依據POSIX以 (IEEE1003.1c)標準定義執行緒產生和同步的API。
  - Pthreads是執行緒行為的**規格**，而非製作。作業系統設計者可以用任何他們期望的方式製作此規格。

- 4.3.2 Win32執行緒
  - 使用Win32執行緒程式庫產生執行的技巧與Pthreads技巧，在許多方面很相似。當使用Win32API時，必須含有windows.h的標題檔。

- 4.3.3 Java執行緒
  - 執行緒是在Java程式、Java語言和JavaAPI中程式執行的基本模式，Java的API提供執行緒的產生與管理一組豐富的特性。所有Java程式至少包含一個單一執行緒控制，即使只包含一個main()方法的Java程式也是以一個單一執行緒在JVM下執行。

Pthreads

- May be provided either as user-level or kernel-level
- stands for POSIX Threads, which is a standardized threading API for Unix-like operating systems (Solaris, Linux, Mac OS X)
- Common in UNIX operating provide a set of functions and data structures for creating and managing threads in a multi-threaded program
- Each thread in a pthread program has its own stack, program counter, and set of registers, but shares memory and resources with other threads within the same process.

Difference of functions

Functions	Thread	Process
Create	pthread_create	fork, vfork
Exit	pthread_exit	exit
Wait	pthread_join	wait, waitpid
Cancel/Terminate	pthread_cancel	abort
Read ID	pthread_self	getpid
Scheduling	SCHED_OTHER, SCHED_FIFO, SCHED_RR	SCHED_OTHER, SCHED_FIFO, SCHED_RR
communication	Message, mutex	Shared memory, pipe, message passing

Preliminaries

- Include pthread.h in the main file
- Compile program with –lpthread
  - gcc –o test test.c –lpthread
  - may not report compilation errors otherwise but calls will fail
- Good idea to check return values on common functions

pthread_attr_init

- #include <pthread.h>

  int pthread_attr_init(pthread_attr_t *attr);
- 以內定(default)的屬性初始化執行緒的屬性
- 若成功返回0, 若失敗返回-1。
- 用pthread_attr_destroy對其去除初始化

  int pthread_attr_destroy(pthread_attr_t*attr);

執行緒屬性

- POSIX定義的執行緒屬性有 : 可分離狀態(detachstate)、執行緒堆疊大小(stacksize)、執行緒堆疊地址(stackaddr)、作用域（scope）、繼承排程(inheritsched)、排程策略(schedpolicy)和排程引數(schedparam)。有些系統並不支援所有這些屬性, 使用前注意檢視系統文件。
- 但是所有Pthread系統都支援detachstate屬性, 該屬性可以是PTHREAD_CREATE_JOINABLE或PTHREAD_CREATE_DETACHED, 預設的是joinable的。擁有joinable屬性的執行緒可以被另外一個執行緒等待, 同時還可以獲得執行緒的返回值, 然後被回收。而detached的執行緒結束時, 使用的資源立馬就會釋放, 不用其他執行緒等待。

執行緒的相關API

- pthread_create()：建立一個執行緒
- pthread_exit()：終止當前執行緒
- pthread_cancel()：中斷另外一個執行緒的執行
- pthread_join()：阻塞當前的執行緒，直到另外一個執行緒執行結束
- pthread_attr_init()：初始化執行緒的屬性
- pthread_attr_setdetachstate()：設定脫離狀態的屬性（決定這個執行緒在終止時是否可以被結合）
- pthread_attr_getdetachstate()：獲取脫離狀態的屬性
- pthread_attr_destroy()：刪除執行緒的屬性
- pthread_kill()：向執行緒傳送一個訊號

Creating a thread: pthread_create

- int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*thread_function)(void *), void *arg);
  - first argument – pointer to the identifier of the created thread
  - second argument – thread attributes
  - third argument – pointer to the function the thread will execute
  - fourth argument – the argument of the executed function (usually a struct)
  - returns 0 for success

pthread_join

- int pthread_join(pthread_t thread, void **thread_return)
  - main thread will wait for daughter thread *thread* to finish
  - first argument – the thread to wait for
  - second argument – pointer to a pointer to the return value from the thread
  - returns 0 for success
  - used to wait for a thread to terminate and retrieve its exit status. This allows for coordination between threads, ensuring that one thread does not proceed until another has completed its work.
  - Threads should always be joined; otherwise, a thread might keep on running even when the main thread has already terminated

Exiting a Thread

- pthreads exist in user space and are seen by the kernel as a single process
  - if one issues and *exit()* system call, all the threads are terminated by the OS
  - if the *main()* function exits, all of the other threads are terminated
- To have a thread exit, use *pthread_exit()*
- Prototype:
  - void pthread_exit(void *status);
    - *status:* the exit status of the thread – passed to the *status* variable in the *pthread_join()* function of a thread waiting for this one

Pthread Example: create one thread

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <string.h>

int sum;
void*runner(void*param);

int main(int argc, char*argv[]){
    pthread_attr_t attr;
    pthread_t tid;

    if(argc!=2){
        printf("using command %s <integer>\n", argv[1]);
        exit(1);
    }

    pthread_attr_init(&attr);
    pthread_create(&tid, &attr, runner, argv[1]);
    pthread_join(tid, NULL);

    printf("sum: %d\n", sum);

    return 0;
}
```

Thread shares global variables

```
void *runner(void*param){
    int i, upper = atoi((char*)param);
    sum = 0;
    for(i=1; i<=upper; i++){
        sum = sum + i;
    }
    pthread_exit(0);
}
```

Example for creating multiple threads

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

#define NUM_THREADS 5

int main() {
    pthread_t threads[NUM_THREADS];
    int thread_args[NUM_THREADS];
    int i, result;

    // Create threads
    for (i = 0; i < NUM_THREADS; i++) {
        thread_args[i] = i;
        result = pthread_create(&threads[i], NULL, thread_function, &thread_args[i]);
        if (result != 0) {
            perror("Thread creation failed");
            exit(EXIT_FAILURE);
        }
    }
```

```
void *thread_function(void *arg) {
    int thread_id = *((int *)arg);
    printf("Thread %d: Hello, World!\n", thread_id);
    pthread_exit(NULL);
}
```

Example for creating multiple threads

```
// Join threads
    for (i = 0; i < NUM_THREADS; i++) {
        result = pthread_join(threads[i], NULL);
        if (result != 0) {
            perror("Thread join failed");
            exit(EXIT_FAILURE);
        }
    }

    printf("Main: All threads have exited successfully.\n");

    return 0;
}
```

Quiz 1

```c
#include <pthread.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <wait.h>

int value = 0;
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
        pid_t pid;
        pthread_t tid; pthread_attr_t attr;

        pid = fork();
        if (pid == 0) { /* child process */
                pthread_attr_init(&attr);
                pthread_create(&tid,&attr,runner,NULL);
                pthread_join(tid,NULL);
                printf("CHILD: value = %d",value); /* LINE C */
        }
        else if (pid > 0) { /* parent process */
                wait(NULL);
                printf("PARENT: value = %d",value); /* LINE P */
        }
}

void *runner(void *param) {
        value = 5;
        pthread_exit(0);
}
```

Quiz 1

```c
#include <pthread.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <wait.h>

int value = 0;
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
        pid_t pid;
        pthread_t tid; pthread_attr_t attr;

        pid = fork();
        if (pid == 0) { /* child process */
                pthread_attr_init(&attr);
                pthread_create(&tid,&attr,runner,NULL);
                pthread_join(tid,NULL);
                printf("CHILD: value = %d",value); /* LINE C */
        }
        else if (pid > 0) { /* parent process */
                wait(NULL);
                printf("PARENT: value = %d",value); /* LINE P */
        }
}

void *runner(void *param) {
        value = 5;
        pthread_exit(0);
}
```

5

0

Problem: vector addition using pthread

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define N 10000
int A[N], B[N], C[N], goldenC[N];
struct v {
  int pos;
};

void *runner(void *param); /* the thread */

int main(int argc, char *argv[]) {
        int i, j, k;
        pthread_t tid[N];       //Thread ID
        pthread_attr_t attr[N]; //Set of thread attributes
        struct timespec t_start, t_end;
        double elapsedTime;

        for(i = 0; i < N; i++) {
            A[i] = rand()%100;
            B[i] = rand()%100;
        }
```

```
void *runner(void *param) {
        struct v *data = param;
        int n;
        C[data->pos] = A[data->pos] + B[data->pos];

        pthread_exit(0);
}
```

```
// start time
clock_gettime( CLOCK_REALTIME, &t_start);

for(i = 0; i < N; i++) {
        //Assign a row and column for each thread
        struct v *data = (struct v *) malloc(sizeof(struct v));
        data->pos = i;
        pthread_attr_init(&attr[i]);
        //Create the thread
        pthread_create(&tid[i],&attr[i],runner,data);
}

for(i = 0; i < N; i++) {
        pthread_join(tid[i], NULL);
}
// stop time
clock_gettime( CLOCK_REALTIME, &t_end);

// compute and print the elapsed time in millisec
elapsedTime = (t_end.tv_sec - t_start.tv_sec) * 1000.0;
elapsedTime += (t_end.tv_nsec - t_start.tv_nsec) / 1000000.0;
printf("Parallel elapsedTime: %lf ms\n", elapsedTime);
//Print out the resulting matrix
```

```
    // start time
    clock_gettime( CLOCK_REALTIME, &t_start);
    for(i = 0; i < N; i++) {
        goldenC[i]=A[i] + B[i];
    }
    // stop time
    clock_gettime( CLOCK_REALTIME, &t_end);
    // compute and print the elapsed time in millisec
    elapsedTime = (t_end.tv_sec - t_start.tv_sec) * 1000.0;
    elapsedTime += (t_end.tv_nsec - t_start.tv_nsec) / 1000000.0;
    printf("Sequential elapsedTime: %lf ms\n", elapsedTime);
        int pass = 1;
    for(i = 0; i < N; i++) {
        if(goldenC[i]!=C[i]){
                pass = 0;
        }
    }
    if(pass==1)
        printf("Test pass!\n");
    else
        printf("Test fail!\n");
    return 0;
}
```

```
Parallel elapsedTime: 244.534305 ms
Sequential elapsedTime: 0.031056 ms
Test pass!
```

Pthread limitation

- **System Resources**:
  - The primary limitation on the number of pthreads is the availability of system resources such as memory (both stack and heap space) and CPU resources.
  - Each pthread requires memory for its stack and data structures maintained by the operating system. Creating a large number of pthreads can quickly consume system memory.
- **Stack Size**:
  - Each pthread has its own stack for storing local variables and function call information. The default stack size for pthreads can vary depending on the operating system and compiler settings.
  - Creating a large number of pthreads with large stack sizes can lead to stack overflow errors and consume significant memory resources.

Pthread limitation

- **Operating System Limits**:
  - Operating systems impose limits on the maximum number of threads a process can create. These limits are often configurable but are typically set to prevent resource exhaustion and ensure system stability.
  - Exceeding these limits may result in errors such as "resource temporarily unavailable" or "unable to create thread".
- **Processor and Core Limits**:
  - The number of pthreads that can effectively run in parallel is limited by the number of available processors and CPU cores.
  - Creating more pthreads than available processors can lead to inefficient thread scheduling and context switching overhead.

Pthread limitation

- **Application Design**:
  - Even if the system allows creating a large number of pthreads, the application's design and architecture may not scale effectively with a large number of threads.
  - Coordinating and synchronizing a large number of threads can introduce overhead and complexity, potentially negating the benefits of parallelism.
- To address these limitations, it's important to carefully design multi-threaded applications, considering factors such as thread creation and destruction, resource utilization, and synchronization. Additionally, monitoring system resource usage and adjusting thread counts based on system capabilities can help optimize performance and avoid resource exhaustion.

Optimization of vector addition using 4 pthreads

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

#define N 10000000
#define NUM_THREADS 4

int A[N];
int B[N];
int C[N];
int goldenC[N];
struct v {
   int pos;
};

void *runner(void *param); /* the thread */
```

```
void *runner(void *param) {
   struct v *data = param;
   int i;
   for(i = 0; i< N/NUM_THREADS; i++){
      C[data->pos+i] = A[data->pos+i] * B[data->pos+i];
   }

   pthread_exit(0);
}

int main(int argc, char *argv[]) {
   int i, j, k;
   pthread_t tid[NUM_THREADS];
   pthread_attr_t attr[NUM_THREADS];
   struct timespec t_start, t_end;
   double elapsedTime;

   for(i = 0; i < N; i++) {
      A[i] = rand()%100;
      B[i] = rand()%100;
   }
```

```
// start time
   clock_gettime( CLOCK_REALTIME, &t_start);

   for(i = 0; i < NUM_THREADS; i++) {
         //Assign a row and column for each thread
         struct v *data = (struct v *) malloc(sizeof(struct v));
         data->pos = i*N/NUM_THREADS;
         pthread_attr_init(&attr[i]);
         pthread_create(&tid[i],&attr[i],runner,data);


    }
      for(i = 0; i < NUM_THREADS; i++) {
      pthread_join(tid[i], NULL);
   }
   // stop time
   clock_gettime( CLOCK_REALTIME, &t_end);

   // compute and print the elapsed time in millisec
   elapsedTime = (t_end.tv_sec - t_start.tv_sec) * 1000.0;
   elapsedTime += (t_end.tv_nsec - t_start.tv_nsec) / 1000000.0;
   printf("Parallel elapsedTime: %lf ms\n", elapsedTime);
```

```
   // start time
   clock_gettime( CLOCK_REALTIME, &t_start);
   for(i = 0; i < N; i++)
      goldenC[i] = A[i] * B[i];
   // stop time
   clock_gettime( CLOCK_REALTIME, &t_end);
   // compute and print the elapsed time in millisec
   elapsedTime = (t_end.tv_sec - t_start.tv_sec) * 1000.0;
   elapsedTime += (t_end.tv_nsec - t_start.tv_nsec) / 1000000.0;
   printf("Sequential elapsedTime: %lf ms\n", elapsedTime);
   int pass = 1;
   for(i = 0; i < N; i++) {
      if(goldenC[i] != C[i]){
            pass = 0;
      }
   }
   if(pass==1)
      printf("Test pass!\n");
   else
      printf("Test fail!\n");
   return 0;
}
```

Performance is improved

```
./vectoradd_pthreadopt
Parallel elapsedTime: 7.437189 ms
Sequential elapsedTime: 27.158524 ms
Test pass!
```

- Scale of input size
- Thread number

Another solution

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

#define NUM_THREADS 4
#define VECTOR_SIZE 1000000

int vector_a[VECTOR_SIZE];
int vector_b[VECTOR_SIZE];
int result[VECTOR_SIZE];

typedef struct {
    int start_index;
    int end_index;
} ThreadArgs;
```

```
void *vector_addition(void *arg) {
    ThreadArgs *args = (ThreadArgs *)arg;
    int start = args->start_index;
    int end = args->end_index;

    for (int i = start; i < end; i++) {
        result[i] = vector_a[i] + vector_b[i];
    }

    pthread_exit(NULL);
}
```

```
int main() {
    pthread_t threads[NUM_THREADS];
    ThreadArgs thread_args[NUM_THREADS];
    int chunk_size = VECTOR_SIZE / NUM_THREADS;
    int remainder = VECTOR_SIZE % NUM_THREADS;

    // Initialize vectors
    for (int i = 0; i < VECTOR_SIZE; i++) {
        vector_a[i] = i;
        vector_b[i] = i;
    }

    // Create threads
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_args[i].start_index = i * chunk_size;
        thread_args[i].end_index = (i == NUM_THREADS - 1) ? VECTOR_SIZE : (i + 1) * chunk_size;

        int result = pthread_create(&threads[i], NULL, vector_addition, &thread_args[i]);
        if (result != 0) {
            perror("Thread creation failed");
            exit(EXIT_FAILURE);
        }
    }
```

```
// Join threads
    for (int i = 0; i < NUM_THREADS; i++) {
        int result = pthread_join(threads[i], NULL);
        if (result != 0) {
            perror("Thread join failed");
            exit(EXIT_FAILURE);
        }
    }


    // Print result
    printf("Result vector:\n");
    for (int i = 0; i < VECTOR_SIZE; i++) {
        printf("%d ", result[i]);
    }
    printf("\n");

    return 0;
}
```

Matrix multiplication

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

#define N 100
#define NUM_THREADS 10000

int A [N][N];
int B [N][N];
int C [N][N];
int goldenC [N][N];
struct v {
   int i; /* row */
   int j; /* column */
};

void *runner(void *param); /* the thread */
```

```
for(i = 0; i < N; i++) {
   for(j = 0; j < N; j++) {
      C[i][j] = 0;
      for(k=0; k<N; k++){
C[i][j]+=A[i][k] * B[k][j];
      }
   }
}
```

```
void *runner(void *param) {
      struct v *data = param;
      int k, sum = 0;
      for(k = 0; k< N; k++){
            sum += A[data->i][k] * B[k][data->j];
      }
      C[data->i][data->j] = sum;
      pthread_exit(0);
}
```

```
int main(int argc, char *argv[]) {
        int i, j, k;
        pthread_t tid[N][N];       //Thread ID
        pthread_attr_t attr[N][N]; //Set of thread attributes
        struct timespec t_start, t_end;
        double elapsedTime;

        for(i = 0; i < N; i++) {
            for(j = 0; j < N; j++) {
                        A[i][j] = rand()%100;
                        B[i][j] = rand()%100;
                }
        }
```

```
      // start time
clock_gettime( CLOCK_REALTIME, &t_start);

for(i = 0; i < N; i++) {
    for(j = 0; j < N; j++) {
        //Assign a row and column for each thread
        struct v *data = (struct v *) malloc(sizeof(struct v));
        data->i = i;
        data->j = j;
        /* Now create the thread passing it data as a parameter */
        //Get the default attributes
        pthread_attr_init(&attr[i][j]);
        //Create the thread
        pthread_create(&tid[i][j],&attr[i][j],runner,data);

    }
}
```

```
for(i = 0; i < N; i++) {
    for(j = 0; j < N; j++) {
        pthread_join(tid[i][j], NULL);
    }
}
// stop time
clock_gettime( CLOCK_REALTIME, &t_end);

// compute and print the elapsed time in millisec
elapsedTime = (t_end.tv_sec - t_start.tv_sec) * 1000.0;
elapsedTime += (t_end.tv_nsec - t_start.tv_nsec) / 1000000.0;
printf("Parallel elapsedTime: %lf ms\n", elapsedTime);
//Print out the resulting matrix
// start time
clock_gettime( CLOCK_REALTIME, &t_start);
for(i = 0; i < N; i++) {
        for(j = 0; j < N; j++) {
                sum = 0;
                for(k=0; k<N; k++){
                        sum += A[i][k] * B[k][j];
                }
                goldenC[i][j] = sum;
        }
}
```

```
    // stop time
    clock_gettime( CLOCK_REALTIME, &t_end);

    // compute and print the elapsed time in millisec
    elapsedTime = (t_end.tv_sec - t_start.tv_sec) * 1000.0;
    elapsedTime += (t_end.tv_nsec - t_start.tv_nsec) / 1000000.0;
    printf("Sequential elapsedTime: %lf ms\n", elapsedTime);

    int pass = 1;
    for(i = 0; i < N; i++) {
        for(j = 0; j < N; j++) {
            if(goldenC[i][j]!=C[i][j]){
                pass = 0;
            }
        }
    }
    if(pass==1)
        printf("Test pass!\n");

    return 0;
}
```

./matrixmul_pthread
Parallel elapsedTime: 253.273290 ms
Sequential elapsedTime: 3.785991 ms

Problem

- Too many threads
- Reading B matrix is inefficient
- Transpose B would be better

Homework

- Optimization of matrix multiplication

Race condition

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

int counter = 0;
void *increment_counter(void *arg) {
    for (int i = 0; i < 1000000; i++) {
        counter++;
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t thread1, thread2;

    pthread_create(&thread1, NULL, increment_counter, NULL);
    pthread_create(&thread2, NULL, increment_counter, NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    printf("Counter value: %d\n", counter);

    return 0;
}
```

```
$ ./race
Counter value: 1019080
$ ./race
Counter value: 1267493
$ ./race
Counter value: 1101581
```

Race condition solution

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

int counter = 0;
pthread_mutex_t mutex;

void *increment_counter(void *arg) {
    for (int i = 0; i < 1000000; i++) {
        // Lock the mutex before accessing the shared data
        pthread_mutex_lock(&mutex);
        counter++;
        // Unlock the mutex after accessing the shared data
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(NULL);
}
```

```
int main() {
    pthread_t thread1, thread2;

    // Initialize the mutex
    pthread_mutex_init(&mutex, NULL);

    // Create threads
    pthread_create(&thread1, NULL, increment_counter, NULL);
    pthread_create(&thread2, NULL, increment_counter, NULL);

    // Wait for threads to complete
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    // Destroy the mutex
    pthread_mutex_destroy(&mutex);

    printf("Counter value: %d\n", counter);
    return 0;
}
```

```
$ ./race_solve
Counter value: 2000000
```

隱性線程(Implicit Threading)

- 隨著線程數量的增加, 顯式線程(explicit threads)的編程正確性更加困難
- 由編譯器和運行時庫(run-time libraries) 創建和管理線程, 而不是由程序員
- Three methods explored
  - Thread Pools
  - OpenMP
  - Grand Central Dispatch

- Other methods include Microsoft Threading Building Blocks (TBB), **java.util.concurrent** package

OpenMP

- OpenMP (Open Multi-Processing) simplifies parallel programming by allowing developers to add parallelism to their existing sequential code with minimal modifications.
- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in **shared-memory** environments
- Identifies **parallel regions** – blocks of code that can run in parallel
- OpenMP programs typically consist of a master thread (the main thread) and a team of parallel threads.

#pragma omp parallel

Create as many threads as there are cores

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

Limitation

- **Limited to Shared-Memory Systems**:
  - OpenMP is designed for shared-memory systems, where threads can access the same memory address space.
  - It cannot be directly used for distributed-memory parallelism or cluster computing.

- **Limited Task Granularity**:
  - OpenMP works best when parallelism can be expressed at the loop or function level.
  - Fine-grained parallelism with small tasks may lead to overhead from thread creation and synchronization, reducing performance gains.

Limitation

- **Limited Control Over Scheduling**:
  - OpenMP provides high-level directives for parallelism but offers limited control over scheduling policies and load balancing. Users have some control through environment variables and clauses, but fine-tuning scheduling behavior can be challenging.
- **Potential for Data Races and Deadlocks**:
  - As with any parallel programming model, developers must carefully manage shared data access and synchronization to avoid data races and deadlocks. Improper use of OpenMP directives can lead to subtle bugs that are difficult to debug.

Limitation

- **Limited Support for Nested Parallelism**:
  - While recent versions of OpenMP support nested parallelism, its use may introduce additional overhead, and not all compilers fully support this feature. Nested parallel regions can also complicate code and make it harder to reason about parallel execution.
- **Performance Overhead**:
  - OpenMP introduces overhead from thread creation, synchronization, and managing parallel regions. For some applications, this overhead may outweigh the benefits of parallelism, particularly for small-scale computations.

Limitation

- **Limited Portability and Compiler Support**:
  - OpenMP is a standardized API, but not all compilers fully support the latest features or provide consistent behavior across different platforms. Portability issues may arise when migrating code between compilers or platforms.
- Despite these limitations, OpenMP remains a powerful and widely used tool for parallel programming, offering a relatively simple and portable way to parallelize code on shared-memory systems. With careful consideration of its limitations and proper usage, OpenMP can significantly improve the performance of many types of applications.

Omp 用法

- #pragma omp directive [clause]
  - parallel
  - for

```
#pragma omp parallel for
for( int i = 0; i < 10; ++ i )
    Test( i );
```

```
#pragma omp parallel
{
    #pragma omp for
    for( int i = 0; i < 10; ++ i )
        Test( i );
}
```

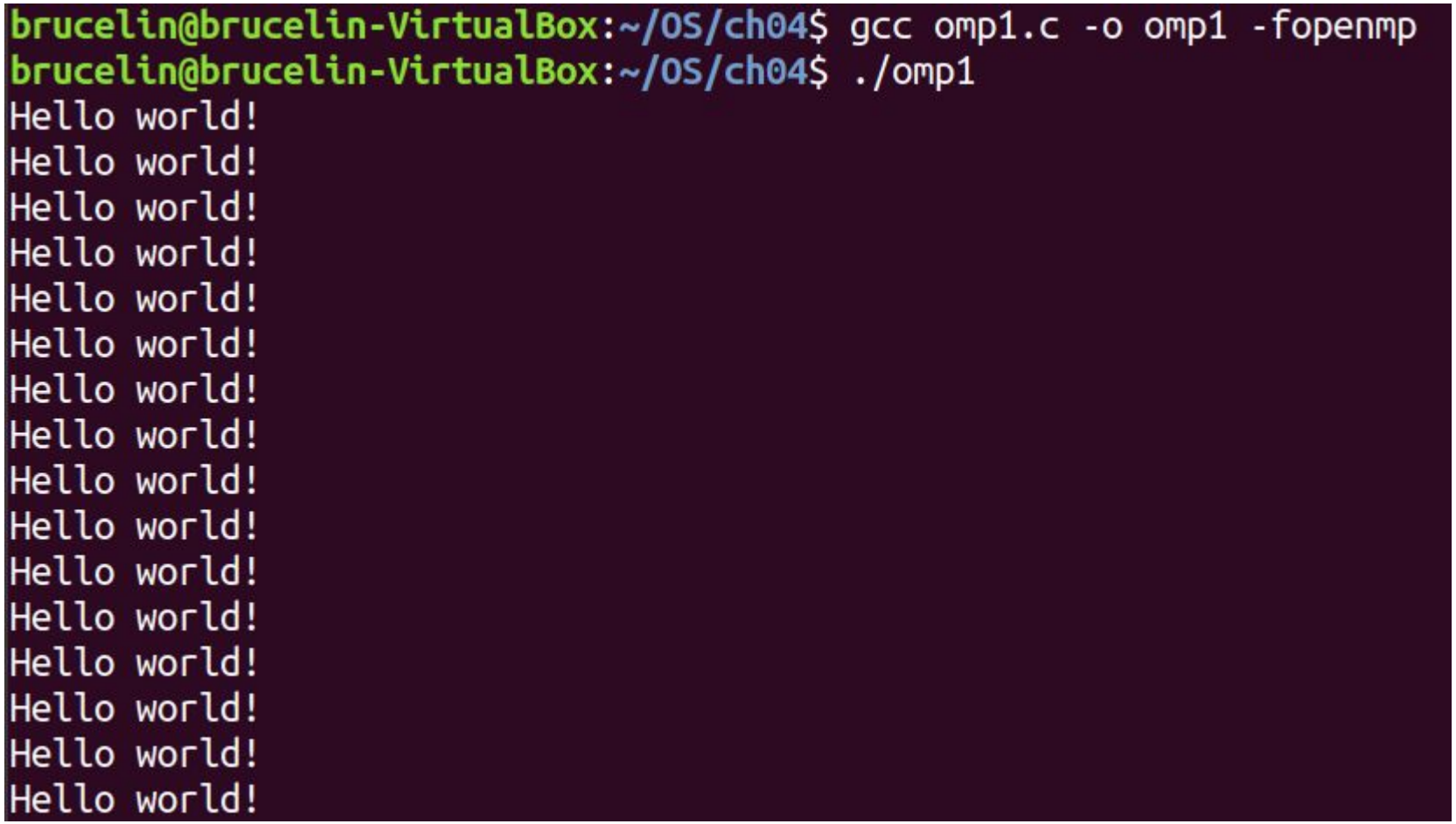
#pragma omp parallel

```
#include <stdio.h>
#include <omp.h>
int main(){

    omp_set_num_threads(16);

    // Do this part in parallel
    #pragma omp parallel
    {
      printf( "Hello  world!\n" );
    }

    return 0;
}
```


brucelin@brucelin-VirtualBox:~/OS/ch04$ gcc omp1.c -o omp1 -fopenmp
brucelin@brucelin-VirtualBox:~/OS/ch04$ ./omp1
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!

#pragma omp parallel for

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

int main(){
    int i;

    omp_set_num_threads(16);

    #pragma omp parallel for schedule(dynamic)
    for(i=0; i<16; i++){
        printf("%d ", i);
    }

    printf("\n");


    return 0;
}
```

```
brucelin@brucelin-VirtualBox:~/OS/ch04$ gcc omp2.c -o omp2 -fopenmp
brucelin@brucelin-VirtualBox:~/OS/ch04$ ./omp2
3 4 13 8 7 6 5 2 9 1 15 10 0 12 11 14
```

directive

directive	function
parallel	代表接下來的程式區塊將被平行化。
for	用在 for 迴圈之前，會將迴圈平行化處理。(註:迴圈的 index 只能是 int)
master	指定由主執行緒來執行接下來的程式。
ordered	指定接下來被程式，在被平行化的 for 迴圈將依序的執行。
atomic	這個指令的目的在於避免變數被同時修改而造成計算結果錯誤。
barrier	等待，直到所有的執行緒都執行到 barrier。用來同步化。

directive

directive	function
Sections	將接下來的 section 平行化處理。
Single	之後的程式將只會在一個執行緒執行，不會被平行化。
threadprivate	定義一個變數是一個線程私有
critical	強制接下來的程式區塊一次只會被一個執行緒執行。
flush	Specifies that all threads have the same view of memory for all shared objects.

clause

clause	functions
copyin	讓 threadprivate 的變數的值和主執行緒的值相同。
copyprivate	將不同執行緒中的變數共用。
default	設定平行化時對變數處理方式的預設值。
firstprivate	讓每個執行緒中，都有一份變數的複本，以免互相干擾；而起始值則會是開始平行化之前的變數值。

clause

clause	functions
if	判斷條件，可以用來決定是否要平行化。
lastprivate	讓每個執行緒中，都有一份變數的複本，以免互相干擾；而在所有平行化的執行緒都結束後，會把最後的值，寫回主執行緒。
nowait	忽略 barrier（等待）。
num_threads	設定平行化時執行緒的數量。

clause

clause	functions
ordered	使用於 for，可以在將迴圈平行化的時候，將程式中有標記 directive ordered 的部份依序執行。
private	定義變數為私有變數，讓每個執行緒中，都有一份變數的複本，以免互相干擾。
reduction	對各執行緒的變數，直行指定的運算元來合併寫回主執行緒。
schedule	設定 for 迴圈的平行化方法；有 dynamic、guided、runtime、static 四種方法。
shared	將變數設定為各執行緒共用。

Find the error

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>
#define N 16
int main(){
        int i;
        int temp;
        int A[N], B[N], AA[N], BB[N];

        for(i=0; i<N; i++){
                A[i] = rand() % 256;
                B[i] = rand() % 256;
                AA[i] = A[i];
                BB[i] = B[i];
        }

        for(i=0; i<N; i++){
                temp = A[i];
                A[i] = B[i];
                B[i] = temp;
        }

        #pragma omp parallel for
        for(i=0; i<N; i++){
                temp = AA[i];
                AA[i] = BB[i];
                BB[i] = temp;
        }

        for(i=0; i<N; i++){
                if(A[i] != AA[i] || B[i]!=BB[i])
                        break;
        }

        if(i==N)
                printf("Test pass!!!\n");
        else
                printf("Test failure\n");
        return 0;
}
```

Solution

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>
#define N 16
int main(){
        int i;
        int temp;
        int A[N], B[N], AA[N], BB[N];

        for(i=0; i<N; i++){
                A[i] = rand() % 256;
                B[i] = rand() % 256;
                AA[i] = A[i];
                BB[i] = B[i];
        }
        for(i=0; i<N; i++){
                temp = A[i];
                A[i] = B[i];
                B[i] = temp;
        }

#pragma omp parallel for private(temp)
        for(i=0; i<N; i++){
                temp = AA[i];
                AA[i] = BB[i];
                BB[i] = temp;
        }

        for(i=0; i<N; i++){
                if(A[i] != AA[i] || B[i]!=BB[i])
                        break;
        }

        if(i==N)
                printf("Test pass!!!\n");
        else
                printf("Test failure\n");
        return 0;
}
```

Find the error

```
#include <stdio.h>
#include <omp.h>
#define N 4

int main(){
	int i, j;
	#pragma omp parallel for
	for(i=0; i<N; i++){
		for(j=0; j<N; j++){
			printf("i:%d, j:%d\n", i, j);
		}
	}
	return 0;
}
```

```
brucelin@brucelin-VirtualBox:~/OS/ch04$ ./omp3
i:0, j:0
i:0, j:1
i:0, j:2
i:0, j:3
i:2, j:0
i:3, j:0
i:1, j:0
```

Solution

```
#include <stdio.h>
#include <omp.h>
#define N 4

int main(){
    int i, j;
    #pragma omp parallel for private( j)
    for(i=0; i<N; i++){
        for(j=0; j<N; j++){
            printf("i:%d, j:%d\n", i, j);
        }
    }
    return 0;
}
```

```
i:1, j:0
i:1, j:1
i:1, j:2
i:1, j:3
i:3, j:0
i:3, j:1
i:3, j:2
i:3, j:3
i:0, j:0
i:0, j:1
i:0, j:2
i:0, j:3
i:2, j:0
i:2, j:1
i:2, j:2
i:2, j:3
```

parallelize only the outer

平行內外迴圈

```
#include <stdio.h>
#include <omp.h>
#define N 4

int main(){
    int i, j;
    #pragma omp parallel for collapse(2)
    for(i=0; i<N; i++){
        for(j=0; j<N; j++){
            printf("i:%d, j:%d\n", i, j);
        }
    }
    return 0;
}
```

```
i:3, j:2
i:1, j:2
i:1, j:3
i:2, j:3
i:2, j:2
i:0, j:0
i:3, j:1
i:2, j:1
i:0, j:1
i:3, j:0
i:3, j:3
i:1, j:0
i:1, j:1
i:0, j:2
i:0, j:3
i:2, j:0
```

Find the bug

```
$ ./omp_err3
Parallel elapsedTime: 11.153300 ms
Sum of array elements: 82241449327.000000
$ ./omp_err3
Parallel elapsedTime: 12.583400 ms
Sum of array elements: 79367531455.000000
```

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

#define ARRAY_SIZE 1000000

int main() {
    int i;
    double sum = 0.0;
    double x[ARRAY_SIZE];

    // Initialize array
    for (i = 0; i < ARRAY_SIZE; i++) {
        x[i] = i + 1;
    }

    #pragma omp parallel for
    for (i = 0; i < ARRAY_SIZE; i++) {
        sum += x[i];
    }
    printf("Sum of array elements: %lf\n", sum);
    return 0;
}
```

Solution

```
#include <stdio.h>
#include <omp.h>
#define ARRAY_SIZE 1000000

int main() {
    int i;
    double sum = 0.0;
    double x[ARRAY_SIZE];

    // Initialize array
    for (i = 0; i < ARRAY_SIZE; i++) {
        x[i] = i + 1;
    }

    #pragma omp parallel for
    for (i = 0; i < ARRAY_SIZE; i++) {
        #pragma omp atomic
        sum += x[i];
    }
    printf("Sum of array elements: %lf\n", sum);
    return 0;
}
```

```
$ ./omp_err3
Parallel elapsedTime: 98.533809 ms
Sum of array elements: 500000500000.000000
$ ./omp_err3
Parallel elapsedTime: 89.085808 ms
Sum of array elements: 500000500000.000000
```

Solution II

- The reduction(+:sum) clause specifies that the variable sum should be treated as a **private variable** for each thread, and the partial sums computed by each thread should be combined using the addition operator (+) after the loop.

```
$ ./omp_err3_reduction
Parallel elapsedTime: 2.777399 ms
Sum of array elements: 500000500000.000000
$ ./omp_err3_reduction
Parallel elapsedTime: 2.906600 ms
Sum of array elements: 500000500000.000000
```

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

#define ARRAY_SIZE 1000000

int main() {
    int i;
    double sum = 0.0;
    double x[ARRAY_SIZE];

    // Initialize array
    for (i = 0; i < ARRAY_SIZE; i++) {
        x[i] = i + 1;
    }

    // Parallel loop with reduction
    #pragma omp parallel for reduction(+:sum)
    for (i = 0; i < ARRAY_SIZE; i++) {
        sum += x[i];
    }
    printf("Sum of array elements: %lf\n", sum);
    return 0;
}
```

Compare the performance of atomic and reduction

- Please compare the performance of atomic add and reduction Add.
- double a[1000000]

```
for(i=0; i<1000000; i++)
    a[i] = i;

for(i=0; i<1000000; i++)
    sum = sum + a[i];
```

Accelerate matrix multiplication using openmp

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

#define N 1000

int A [N][N];
int B [N][N];
int C [N][N];
int goldenC [N][N];
```

```
int main(int argc, char *argv[]) {
    int i, j, k;
    int sum;
    struct timespec t_start, t_end;
    double elapsedTime;

    for(i = 0; i < N; i++) {
        for(j = 0; j < N; j++) {
            A[i][j] = rand()%100;
            B[i][j] = rand()%100;
        }
    }
```

```
// start time
 clock_gettime( CLOCK_REALTIME, &t_start);

 #pragma omp parallel for private(j, k, sum)
 for(i = 0; i < N; i++) {
     for(j = 0; j < N; j++) {
         for(k=0; k<N; k++){
             C[i][j] += A[i][k] * B[k][j];
         }
     }
 }
 // stop time
 clock_gettime( CLOCK_REALTIME, &t_end);
```

```
// compute and print the elapsed time in millisec
elapsedTime = (t_end.tv_sec - t_start.tv_sec) * 1000.0;
elapsedTime += (t_end.tv_nsec - t_start.tv_nsec) / 1000000.0;
printf("Parallel elapsedTime: %lf ms\n", elapsedTime);
//Print out the resulting matrix
// start time
clock_gettime( CLOCK_REALTIME, &t_start);
for(i = 0; i < N; i++) {
    for(j = 0; j < N; j++) {
        sum = 0;
        for(k=0; k<N; k++){
            sum += A[i][k] * B[k][j];
        }
        goldenC[i][j] = sum;
    }
}
// stop time
clock_gettime( CLOCK_REALTIME, &t_end);
```

```
    // compute and print the elapsed time in millisec
    elapsedTime = (t_end.tv_sec - t_start.tv_sec) * 1000.0;
    elapsedTime += (t_end.tv_nsec - t_start.tv_nsec) / 1000000.0;
    printf("Sequential elapsedTime: %lf ms\n", elapsedTime);

    int pass = 1;
    for(i = 0; i < N; i++) {
        for(j = 0; j < N; j++) {
            if(goldenC[i][j]!=C[i][j]){
                pass = 0;
            }
        }
    }
    if(pass==1)
        printf("Test pass!\n");

    return 0;
}
```

4.6 執行緒的重要事項

- 如果有一個行程(process)的某一個執行緒(thread)呼叫fork(), 新的process是否會複製所有的執行緒, 或者只複製呼叫fork()的那一個執行緒?
  - 兩種方式皆有
- 如果有一個執行緒(thread)呼叫exec(), exec()的參數會取代整個new process, 包括其所有的執行緒。

4.6.2 信號處理(signal handling)

- 信號由於特定事件的發生而產生
- 產生的信號被送到一個行程。
- 一旦送達後，此信號必須處理。
- 一個信號應該被傳送到那裡呢?通常有以下的選擇存在:
  1. 傳送信號到此信號作用的執行緒。
  2. 傳送信號到行程中的每一個執行緒。
  3. 傳送信號到行程中特定的執行緒。
  4. 指定一個特定的執行緒來接收該行程的所有信號。

4.6.3 執行緒取消(thread cancellation)

- 執行緒取消(thread cancellation)是在一個執行緒完成之前結束它。
  - 非同步取消 (asynchronous cancellation):一個執行緒立即終止目標執行緒。
  - 延遲取消 (deferred cancellation):目標執行緒可以週期地檢查它是否該被取消，逼允許目標執行緒有機會以有條不紊的方式結束自己。

4.6.4 執行緒池(Thread pool)

- 執行緒池的優點有:

1. 通常對於服務一項要求時, 使用現存的執行緒比等待產生一個執行緒快。
2. 執行緒限制了任何時候執行緒的個數。這對於無法支援大量並行執行緒的系統特別重要。
3. 將running task與 creating thread分開, 這個特點允許我們使用不同的方式來執行這個task。例如:我們可以在一個time delay後執行或週期性的執行這個task。

4.6.5 排班程式活化作用

- 在使用者執行緒程式庫與核心之間通信其中一個技巧稱為排班程式活化作用(scheduler activation)。
- 以下列方式工作:核心以一組虛擬處理器 (LWPs)提供一個應用程式，並且應用程式在可用的虛擬處理器上排班執行。再者，核心必須通知應用程式一些事件，這種流程稱為向上呼叫(upcall)。執行緒程式庫用一個向上呼叫處理程式 (upcall handler)處理向上呼叫，而且向上呼叫必須在虛擬處理器上執行。

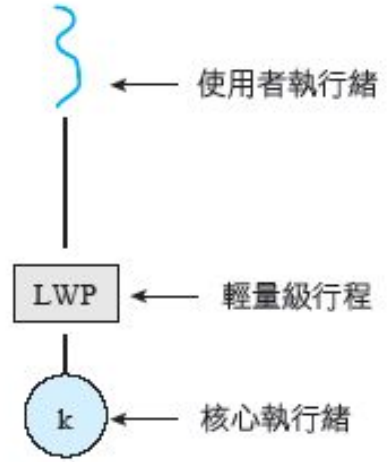


圖 4.12 輕量級行程（LWP）

4.5.2 Linux 執行緒

- Linux提供一個fork系統，它擁有傳統的複製行程功能。
- Linux也提供clone系統呼叫來產生執行緒。
- 然而Linux無法區分行程與執行緒。事實上，當在程式內一連串控制時，Linux通常使用任務(task)而不是行程或執行緒。
- 當啟動clone，它傳遞一組旗標，決定有多少共用發生在父任務與子任務之間。下列是這些旗標當中的一部份。

旗標	意義
CLONE_FS	共用檔案系統訊息
CONE_VM	共用相同記憶體空間
CLONE_SIGHAND	共用訊號處理程式
CLONE_FILES	共用一組的開啓檔案

Example of clone()

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sched.h>
#include <wait.h>
#include <string.h>
static int child_func(void* arg) {
  char* buf = (char*)arg;
  printf("Child sees buf = \"%s\"\n", buf);
  strcpy(buf, "hello from child");
  return 0;
}
int main(int argc, char** argv) {
  // Allocate stack for child task.
  const int STACK_SIZE = 65536;
  char* stack = malloc(STACK_SIZE);
  if (!stack) {
    perror("malloc");
    exit(1);
  }
```

```
  // When called with the command-line argument "vm", set the CLONE_VM flag on.
  unsigned long flags = 0;
  if (argc > 1 && !strcmp(argv[1], "vm")) {
    flags |= CLONE_VM;
  }

  char buf[100];
  strcpy(buf, "hello from parent");
  if (clone(child_func, stack + STACK_SIZE, flags | SIGCHLD, buf) == -1) {
    perror("clone");
    exit(1);
  }

  int status;
  if (wait(&status) == -1) {
    perror("wait");
    exit(1);
  }

  printf("Child exited with status %d. buf = \"%s\"\n", status, buf);
  return 0;
}
```

```
$ gcc clone_example3.c –o clone_example3 -lpthread
$./clone_example3
Child sees buf = "hello from parent"
Child exited with status 0. buf = "hello from parent"
$ ./clone_example3 vm
Child sees buf = "hello from parent"
Child exited with status 0. buf = "hello from child"
```

Difference between fork() and clone()

Fork()	Clone()
• used to create a new process. • After fork() is called, two identical processes are created: the parent process and the child process. These processes are identical in almost all respects, except for their process IDs (pid). • The child process gets a copy of the parent process's address space, including all memory regions and file descriptors. • both processes have separate copies of memory, file descriptors, etc. Any changes made in the parent or child process do not affect the other process.	• used to create a new process or a new thread within the same process. • clone() allows fine-grained control over what resources are shared between the parent and child threads/processes by passing various flags. • This allows for more fine-grained control over resource sharing. For example, you can choose to share the memory address space, file descriptors, signal handlers, and more. • useful for implementing thread pooling, where a pool of threads can be created and reused for executing tasks in a multithreaded application.