

Parallel Computing (III)

多執行緒 (Multithreaded Programming)

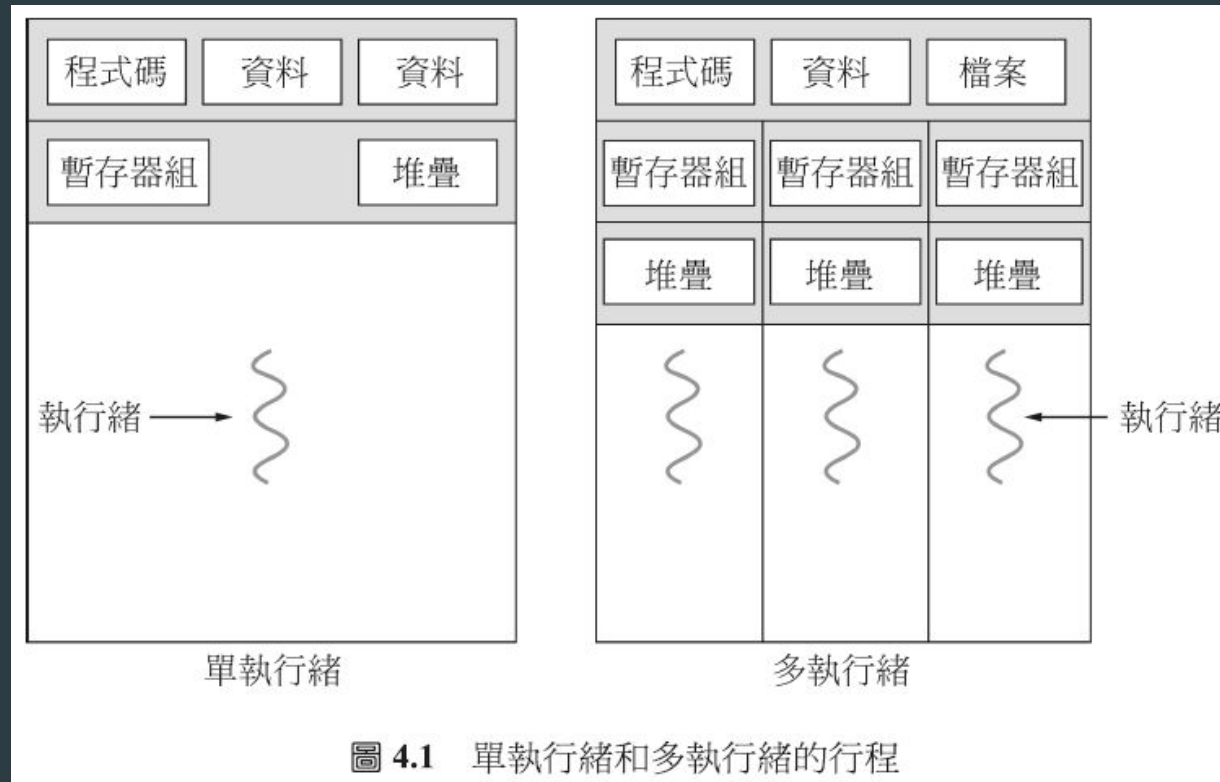
Cheng-Hung Lin

Multicore Programming

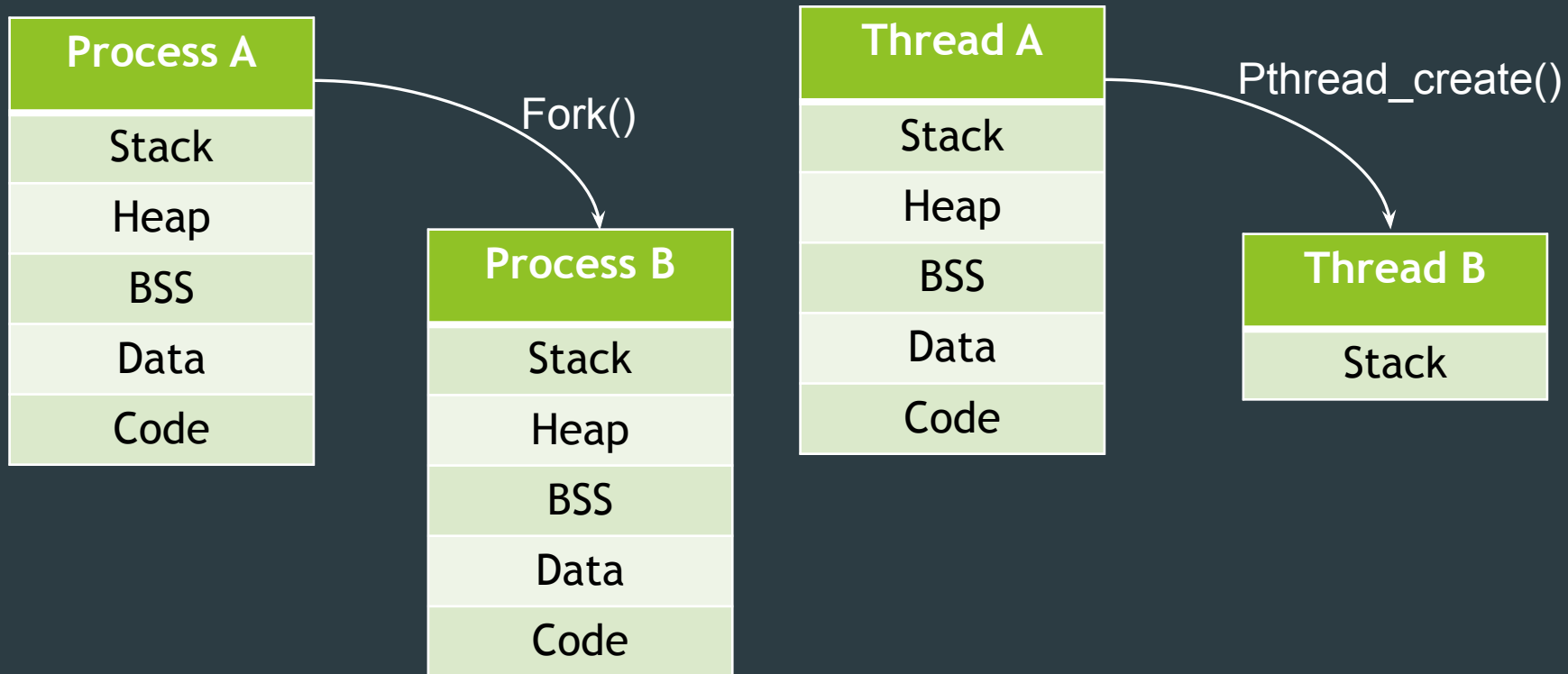
- ▶ **Multicore or multiprocessor** systems putting pressure on programmers, challenges include:
 - ▶ **Dividing activities**
 - ▶ **Balance**
 - ▶ **Data splitting**
 - ▶ **Data dependency**
 - ▶ **Testing and debugging**
- ▶ *Parallelism* implies a system can perform more than one task simultaneously
- ▶ *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

執行緒(Thread)

- ▶ 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。



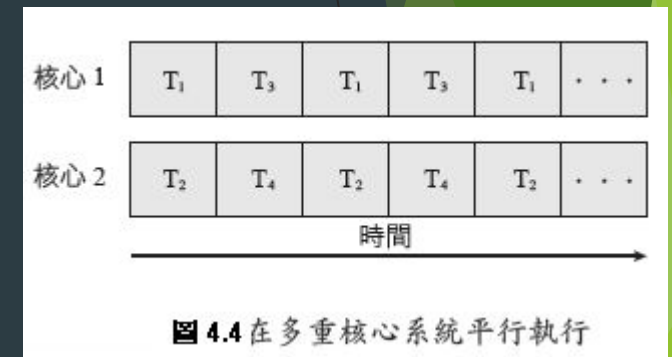
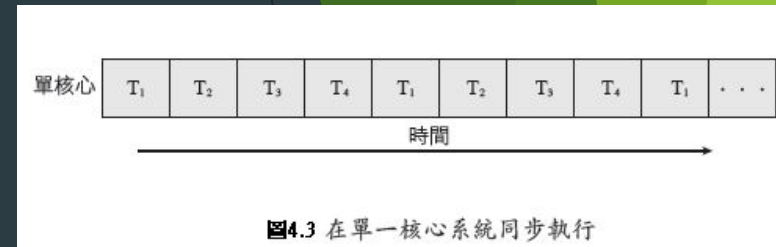
Difference between creating a process and a thread



- 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.

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：
- ▶ 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。
- ▶ 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。
- ▶ 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。
- ▶ 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。
- ▶ 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。



Difference of functions

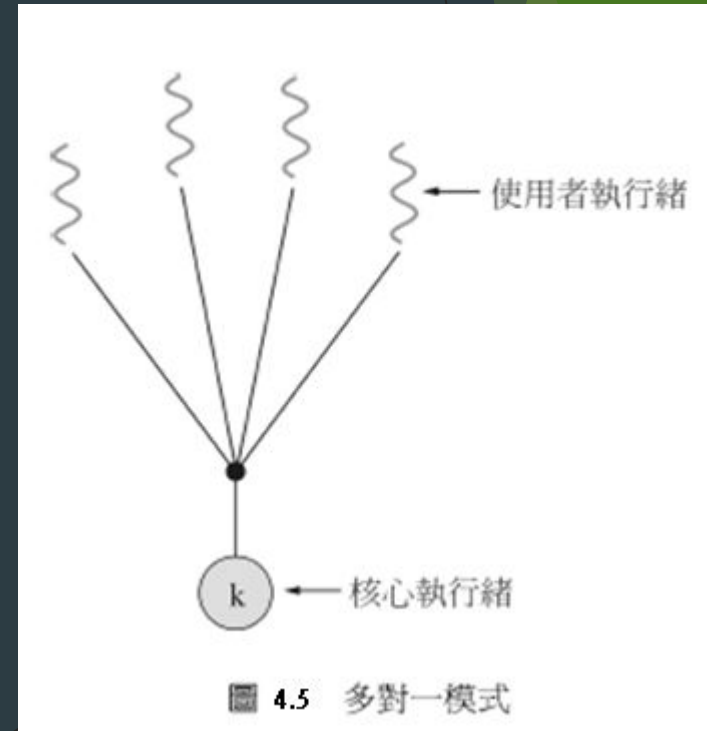
Functions	Thread	Process
Create	pthread_create	fork, vfork
Exit	pthread_exit	exit
Wait	pthread_join	wait, waitpid
Cancel/Terminate	pthread_cancel	abort
Read ID	pthread_self	getpid
Scheduling	SCHED_OTHER, SCHED_FIFO, SCHED_RR	SCHED_OTHER, SCHED_FIFO, SCHED_RR
communication	Message, mutex	Shared memory,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

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

多執行緒模式

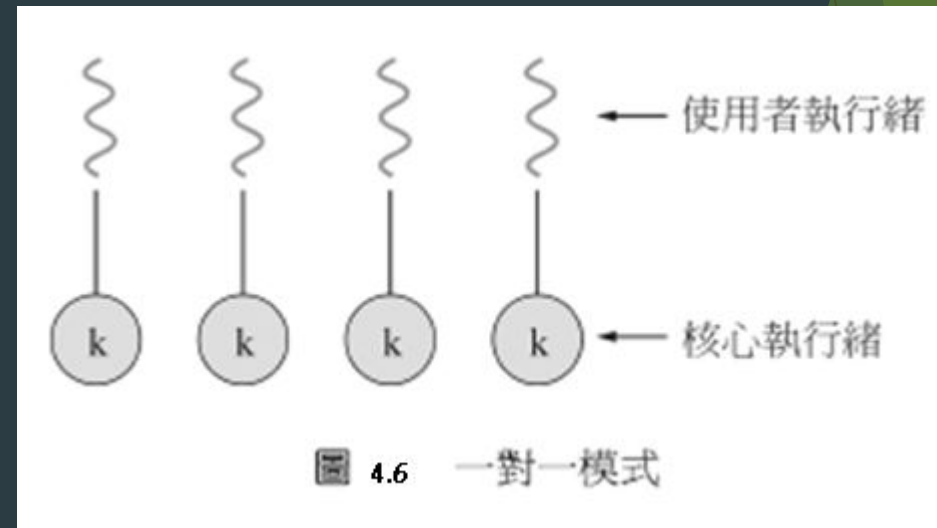
- ▶ 多對一模式(Many-to-one model)
 - ▶ Many user-level threads mapped to a single kernel thread
 - ▶ One thread blocking causes all to block
 - ▶ Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- ▶ Few systems currently use this model because it cannot take advantage of multiple processing cores.
- ▶ Examples:
 - ▶ Solaris Green Threads
 - ▶ GNU Portable Threads



一對一模式(One-to-One)

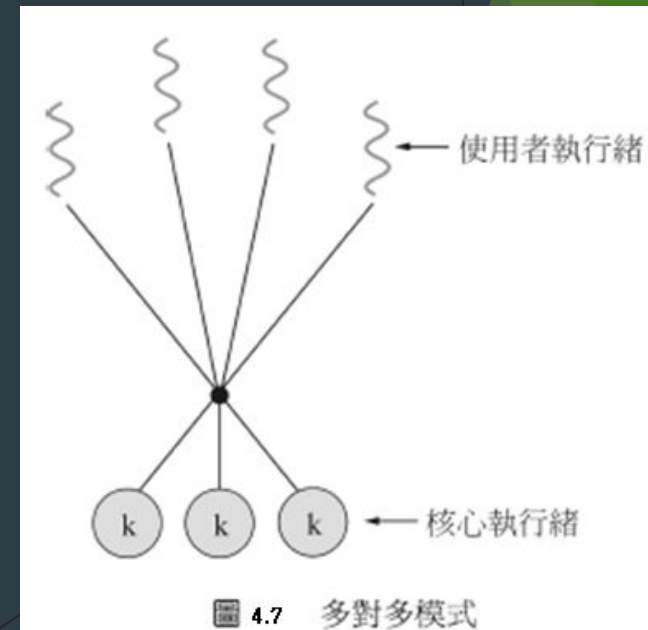
- ▶ Map each user thread to a kernel thread
- ▶ Creating a user-level thread requires creating a kernel thread
- ▶ Supporting multiprocessors
- ▶ The number of threads is restricted due to the overhead of creating kernel threads.

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



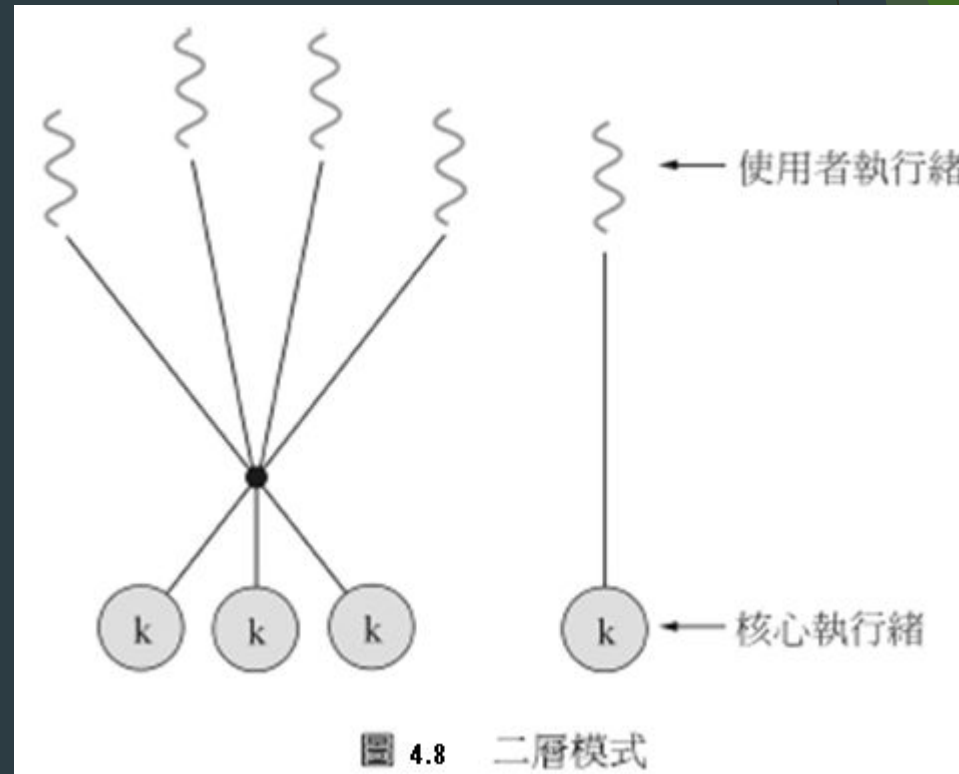
多對多模式(Many-to-Many)

- ▶ **Multiplexes** many user-level threads to a smaller or 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



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



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

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**
- ◆ A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ◆ *Specification*, not *implementation*
- ◆ API specifies behavior of the thread library, implementation is up to development of the library
- ◆ Common in UNIX operating systems (Solaris, Linux, Mac OS X)

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

Waiting for the threads to finish: `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
 - 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 an *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

Pthreads Example

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

int sum; /* this data is shared by the thread(s) */

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

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

    if (argc != 2) {
        fprintf(stderr,"usage: a.out <integer value>\n");
        /*exit(1);*/
        return -1;
    }

    if (atoi(argv[1]) < 0) {
        fprintf(stderr,"Argument %d must be non-negative\n",atoi(argv[1]));
        /*exit(1);*/
        return -1;
    }
}
```

```
pthread_attr_init(&attr); /* get the default attributes */  
  
pthread_create(&tid,&attr,runner,argv[1]); /* create the thread */  
  
pthread_join(tid,NULL); /* now wait for the thread to exit */  
  
printf("sum = %d\n",sum);  
}
```

// The thread will begin control in this function

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

Thread shares global variables

Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

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

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

Figure 4.10 Pthread code for joining ten threads.

Quiz 1

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

int value = 0; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */
int main(int argc, char *argv[])
{
    pid_t pid;
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of attributes for the thread */

    pid = fork();

    if (pid == 0) { /* child process */
        pthread_attr_init(&attr); /* get the default attributes */
        pthread_create(&tid,&attr,runner,argv[1]); /* create the thread */
        pthread_join(tid,NULL); /* now wait for the thread to exit */
        printf("CHILD: value = %d\n",value);
    }
```

```
    else if (pid > 0) { /* parent process */
        wait(NULL);
        printf("PARENT: value = %d\n",value);
    }
}

/**
 * The thread will begin control in this function
 */
void *runner(void *param)
{
    value = 5;

    pthread_exit(0);
}
```

Quiz 1 Answer

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

int value = 0; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */
int main(int argc, char *argv[])
{
    pid_t pid;
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of attributes for the thread */

    pid = fork();

    if (pid == 0) { /* child process */
        pthread_attr_init(&attr); /* get the default attributes */
        pthread_create(&tid,&attr,runner,argv[1]); /* create the thread */
        pthread_join(tid,NULL); /* now wait for the thread to exit */
        printf("CHILD: value = %d\n",value);
    }
```

```
    else if (pid > 0) { /* parent process */
        wait(NULL);
        printf("PARENT: value = %d\n",value);
    }
}

/**
 * The thread will begin control in this function
 */
void *runner(void *param)
{
    value = 5;

    pthread_exit(0);
}
```

```
$ ./quiz1
CHILD: value = 5
PARENT: value = 0
```

Problem: vector addition using pthread

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

```
#define N 100
```

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

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

```
int main(int argc, char *argv[]) {
    int i, j, k, *pos;
    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;
```

```
    }
```

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

```
    for(i = 0; i < N; i++) {
        //Assign a row and column for each thread
        pos = (int*)malloc(sizeof(int));
        *pos = i;
        pthread_attr_init(&attr[i]);
        pthread_create(&tid[i],&attr[i],runner,pos);
    }
```

```

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;
}

//The thread will begin control in this function
void *runner(void *param) {
    int *pos = param;
    int i;
    C[*pos] = A[*pos] + B[*pos];
    pthread_exit(0);
}

```

Homework

- ▶ Change N to 1000000
- ▶ Create 4 threads
- ▶ Each thread performs addition on $1000000/4$ elements.



Matrix multiplication

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

```
#define N 100
```

```
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 */
```

```
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 sequentialelapsedTime;
    double parallelelapsedTime;

    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;
        //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);

```

```

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);
}

```

```

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

```



```
// start time
clock_gettime( CLOCK_REALTIME, &t_start);
for(i = 0; i < N; i++) {
    for(j = 0; j < N; j++) {
        for(k=0; k< N; k++){
            goldenC[i][j]+=A[i][k] * B[k][j];
        }
    }
}
// stop time
clock_gettime( CLOCK_REALTIME, &t_end);

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

printf("Speedup %.2lf\n", sequentialelapsedTime/parallelelapsedTime);
```

```
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;
}
```

Problem

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

Synchronization Problem

- ▶ Assume variable “counter” is shared by processes.
- ▶ The statement “counter++” & “counter--” may be implemented in machine language as:

```
move ax, counter
add  ax, 1
move counter, ax
```

```
move bx, counter
sub   bx, 1
move counter, bx
```

```
Process0
main() {
    .
    counter++;
    .
}
```

```
Process1
main() {
    .
    counter--;
    .
}
```

Instruction Interleaving

► Instruction Interleaving

producer: move ax, counter ☐ ax = 5

producer: add ax, 1 ☐ ax = 6

context switch

consumer: move bx, counter ☐ bx = 5

consumer: sub bx, 1 ☐ bx = 4

context switch

producer: move counter, ax ☐ counter = 6

context switch

consumer: move counter, bx ☐ counter = 4

☐ The value of counter may be either 4, 5, or 6

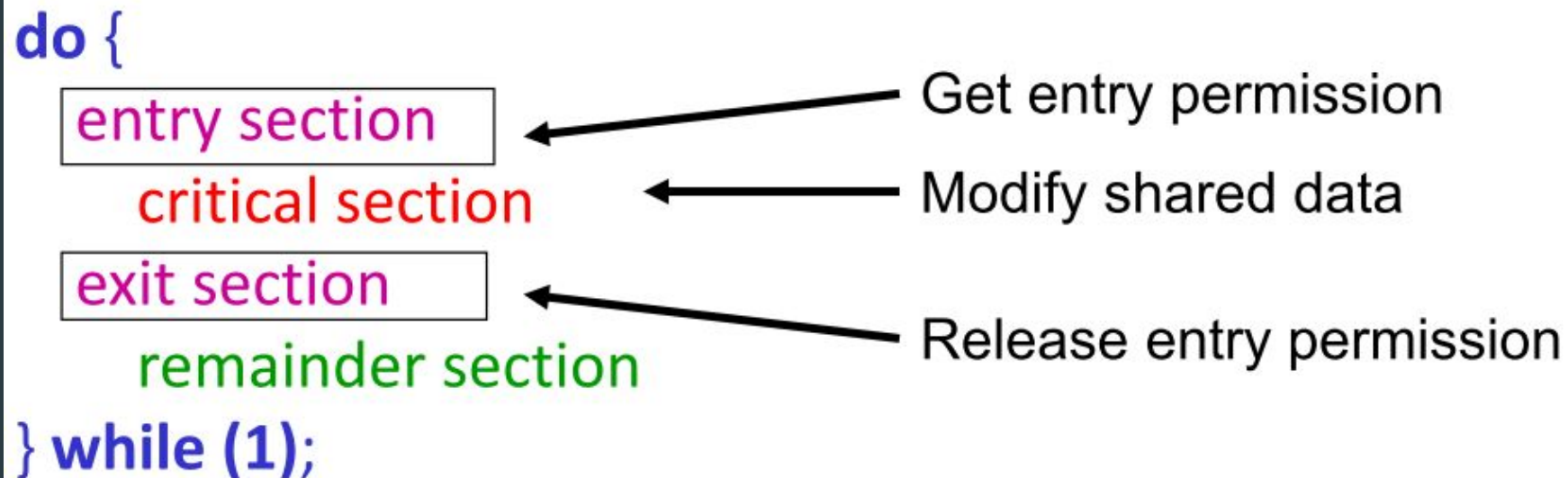
☐ The ONLY correct result is 5!

Thread-Safe Routines

- ▶ System calls or library routines are called “**Thread-Safe**” if they can be called from multiple threads simultaneously and always produce correct results

Critical Section & Mutual Exclusion

- ▶ **Critical Section** is a piece of code that can only be accessed by one process/thread at a time
- ▶ **Mutual exclusion** is the problem to insure only one process/thread can be in a critical section
- ▶ E.g.: The design of entry section & exit section provides mutual exclusion for the critical section



Critical section problem

- ▶ 多個程序同時對共享資料(shared data)進行存取可能會造成資料不一致(data inconsistency)
- ▶ 為保持資料的一致性，必須確保合作程序的執行順序
- ▶ 假設生產者/消費者問題中，共用一個緩衝(buffer)。透過一個變數count來記錄這個緩衝的使用數量。其中count初始化為0。當生產者生產一個資料時，count加一，相反的當消費者消費一個資料時，count減一。

Producer

```
while (true) {  
    /* produce an item and put in nextProduced */  
    while (count == BUFFER_SIZE); // do nothing  
    buffer [in] = nextProduced;  
    in = (in + 1) % BUFFER_SIZE;  
    count++;  
}
```

Consumer

```
while (true) {  
    while (count == 0); // do nothing  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    count--;  
  
    /* consume the item in nextConsumed  
}
```

- 這個不正確的狀態是因為允許兩個行程並行處理這個 counter 變數。像這種數個行程同時存取和處理相同資料的情況，而且執行的結果取決於存取時的特殊順序，就叫**競爭情況 (race condition)**。

```
register1 = counter  
register1 = register1 + 1  
counter = register1
```

```
register2 = counter  
register2 = register2 - 1  
counter = register2
```

T_0 ：	生產者	執行	$register_1 = counter$	$\{register_1 = 5\}$
T_1 ：	生產者	執行	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2 ：	消費者	執行	$register_2 = counter$	$\{register_2 = 5\}$
T_3 ：	消費者	執行	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4 ：	生產者	執行	$counter = register_1$	$\{counter = 6\}$
T_5 ：	消費者	執行	$counter = register_2$	$\{counter = 4\}$

Solution to critical section problem

- ▶ 解決critical section problem的方案必須滿足下面三個要求
 - ▶ **互斥(mutual exclusion)**:如果行程 P_i 正在臨界區(critical section)間內執行, 則其它的行程不能在其臨界區間內執行。
 - ▶ **進行(Progress)**:如果沒有行程在臨界區間內執行, 同時某一行程想要進入臨界區間, 那麼只有那些不在剩餘區間執行的行程才能加入決定誰將在下一次進入臨界區間, 並且這個選擇不得無限期地延遲下去。
 - ▶ **限制住的等待(bound waiting)**:在一個行程已經要求進入臨界區間, 而此要求尚未被答應之前, 允許其它的行程進入臨界區間的次數有一個限制。

```
do {  
    entry section  
    臨界區間  
    exit section  
    剩餘區間  
} while (TRUE);
```

圖 6.1 典型行程 P_i 的一般結構圖

Locks

- ▶ Lock: the simplest mechanism for ensuring mutual exclusion of critical section
 - ▶ Spinlock is one of the implementation:

```
while (lock == 1);      /* no operation in while loop */  
lock = 1;               /* enter critical section */  
.  
critical section  
.  
lock = 0;               /* leave critical section */
```

- ▶ Locks are implemented in Pthreads by a special type of variables “mutex”
- ▶ Mutex is abbreviation of “**m**utual **e**xclusion”

Mutex Locks

do {

acquire lock

critical section

release lock

remainder section

} while (true);

```
acquire(){  
    while(!available); //busy wait  
    available = false;  
}
```

```
release(){  
    available = true;  
}
```

Mutex Locks

- ▶ OS designers build software tools to solve critical section problem
- ▶ Simplest is mutex lock
- ▶ 要進入critical section先透過acquire() 取得 lock 之後再透過release() 釋放lock
 - ▶ 利用布林變數指示lock 可用還是不可用
- ▶ **acquire()** and **release()** 必須是原子化(atomic)
 - ▶ 通常用hardware atomic instructions實現
- ▶ But this solution requires **busy waiting**
 - ▶ This lock therefore called a **spinlock**

Pthread Lock/Mutex Routines

- ▶ To use mutex, it must be a **global variable** declared as of type **pthread_mutex_t** and initialized with **pthread_mutex_init()** or **PTHREAD_MUTEX_INITIALIZER**;
- ▶ A mutex is destroyed with **pthread_mutex_destory()**
- ▶ A critical section can then be protected using **pthread_mutex_lock()** and **pthread_mutex_unlock()**

Mutex Locks

- ▶ A Mutex lock is created like a normal variable
 - ▶ *pthread_mutex_t mutex;*
- ▶ Mutexes must be **initialized** before being used
 - ▶ a mutex can only be initialized once
 - ▶ prototype:
 - ▶ *int pthread_mutex_init(pthread_mutex_t *mp, const pthread_mutexattr_t *mattr);*
 - ▶ *mp*: a pointer to the mutex lock to be initialized
 - ▶ *mattr*: attributes of the mutex – usually NULL
 - ▶ Example

```
/* pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; */
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
```

Linux的同步

- Linux在2.6版之前是一個不可搶先的核心，也就是行程在核心模式執行是不可搶先，甚至如果一個較高優先的權行程可以執行時。然而，現在Linux完全可以搶先，所以當任務正在核心執行時是可以搶先的。

單一處理器	多處理器
不具核心搶先式	獲得盤旋鎖
具有核心搶先式	釋放盤旋鎖

Pthreads的同步

- Threads API在執行緒同步提供**mutex locks**、**condition variables**和**read-write locks**。這一個API對電腦程式設計師是可用的以及不是任何特別核心的一部份。mutex locks呈現使用Pthreads的基本同步技術。mutex locks用來保護臨界區間的程式碼，也就是執行緒在進入臨界區間前獲得鎖，然後在離開臨界區間將鎖釋放。

Locking a Mutex

- ▶ To insure **mutual exclusion** to a **critical section**, a thread should lock a **mutex**
 - ▶ when locking function is called, it does not return until the current thread owns the lock
 - ▶ if the mutex is already locked, calling thread blocks
 - ▶ if multiple threads try to gain lock at the same time, the return order is based on priority of the threads
 - ▶ higher priorities return first
 - ▶ no guarantees about ordering between same priority threads
- ▶ prototype:
 - ▶ `int pthread_mutex_lock(pthread_mutex_t *mp);`
 - ▶ *mp*: mutex to lock

Unlocking a Mutex

- ▶ When a thread is finished within the critical section, it needs to **release** the mutex
 - ▶ calling the **unlock** function releases the lock
 - ▶ then, any threads waiting for the lock compete to get it
 - ▶ very important to remember to release mutex
 - ▶ prototype:
 - ▶ `int pthread_mutex_unlock(pthread_mutex_t *mp);`
 - ▶ *mp*: mutex to unlock

Pthreads Synchronization

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

```
pthread_mutex_t mutex;
```

```
/*create the mutex lock*/
```

```
pthread_mutex_init(&mutex, NULL);
```

```
/*acquire the mutex lock*/
```

```
pthread_mutex_lock(&mutex);
```

```
/* critical section*/
```

```
/*release the mutex lock*/
```

```
pthread_mutex_unlock(&mutex);
```

Example

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

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int count;
```

```
void * thread_run(void *arg){
    int i;
    pthread_mutex_lock(&mutex);

    for (i = 0; i < 3; i++) {
        printf("[%ld]value of count: %d\n", pthread_self(), ++count);
    }

    pthread_mutex_unlock(&mutex);
    return 0;
}
```

Critical Section

```
int main(int argc, char *argv[]){
    pthread_t thread1, thread2, thread3, thread4;
    pthread_create(&thread1, NULL, thread_run, 0);
    pthread_create(&thread2, NULL, thread_run, 0);
    pthread_create(&thread3, NULL, thread_run, 0);
    pthread_create(&thread4, NULL, thread_run, 0);
    printf("thread1 id: %ld\n", thread1);
    printf("thread2 id: %ld\n", thread2);
    printf("thread3 id: %ld\n", thread3);
    printf("thread4 id: %ld\n", thread4);
    pthread_join(thread1, 0);
    pthread_join(thread2, 0);
    pthread_join(thread3, 0);
    pthread_join(thread4, 0);
    pthread_mutex_destroy(&mutex);
    return 0;
}
```

```
$ ./mutex1
[139816833672960]value of count: 1
[139816833672960]value of count: 2
[139816833672960]value of count: 3
thread1 id: 139816833672960
thread2 id: 139816825280256
thread3 id: 139816816887552
thread4 id: 139816687630080
[139816687630080]value of count: 4
[139816687630080]value of count: 5
[139816687630080]value of count: 6
[139816816887552]value of count: 7
[139816816887552]value of count: 8
[139816816887552]value of count: 9
[139816825280256]value of count: 10
[139816825280256]value of count: 11
[139816825280256]value of count: 12
```

如果拿掉mutex lock會如何?

Example of Race condition

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *thread(void *param);
int count = 0;
#define N 10000
int main(int argc, char *argv[])
{
    int i;
    pthread_t tid[N];
    pthread_attr_t attr[N];

    //Create N threads
    for(i = 0; i < N; i++){
        pthread_attr_init(&attr[i]);
        pthread_create(&tid[i], &attr[i], thread, NULL);
    }
    for(i = 0; i < N; i++){
        pthread_join(tid[i], NULL);
    }
    printf("count is %d\n", count);
    return 0;
}

void *thread(void *param)
{
    count++;
}
```

Solution using Mutex lock

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

void *thread(void *param);
int count = 0;
pthread_mutex_t mutex;
#define N 10000
int main(int argc, char *argv[]){
    int i;
    pthread_t tid[N];
    pthread_attr_t attr[N];
    pthread_mutex_init(&mutex, NULL);

    //Create N threads
    for(i = 0; i < N; i++){
        pthread_attr_init(&attr[i]);
        pthread_create(&tid[i], &attr[i], thread, NULL);
    }
```

```
        for(i = 0; i < N; i++){
            pthread_join(tid[i], NULL);
        }
        printf("count is %d\n", count);

        return 0;
    }

void *thread(void *param)
{
    //acquire the mutex lock
    pthread_mutex_lock(&mutex);
    count++;
    //release the mutex lock
    pthread_mutex_unlock(&mutex);
}
```

Example

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

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

```
#define MAX_SIZE 5
```

```
pthread_mutex_t bufLock;
```

```
int count;
```

```
void producer(char* buf) {
```

```
    for(;;) {
```

```
        while(count == MAX_SIZE);
```

```
        pthread_mutex_lock(&bufLock);
```

```
        printf("enter a char:");
```

```
        buf[count] = getchar();
```

```
        getchar();
```

```
        count++;
```

```
        pthread_mutex_unlock(&bufLock);
```

```
    }
```

```
}
```

```
void consumer(char* buf) {
```

```
    for(;;) {
```

```
        while(count == 0);
```

```
        pthread_mutex_lock(&bufLock);
```

```
        printf("output buffer:");
```

```
        printf("%c\n", buf[count-1]);
```

```
        count--;
```

```
        pthread_mutex_unlock(&bufLock);
```

```
    }
```

```
}
```

```
int main() {
```

```
    char buffer[MAX_SIZE];
```

```
    pthread_t p;
```

```
    count = 0;
```

```
    pthread_mutex_init(&bufLock, NULL);
```

```
    pthread_create(&p, NULL, (void*)producer, buffer);
```

```
    consume(buffer);
```

```
    return 0;
```

```
}
```

死結(Deadlock)

Deadlock example

```
#include<pthread.h>
#include<stdio.h>
#include<stdlib.h>
#include <unistd.h>
pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t lock2 = PTHREAD_MUTEX_INITIALIZER;
int test=0;
void *proc1(){
    printf("\nThis is proc1 using rs1\n");
    pthread_mutex_lock(&lock1);
    usleep(200); //200 micro seconds
    printf("\np1 trying to get rs2...\n");
    pthread_mutex_lock(&lock2);
    test++;
    printf("Test: %d\n", test);
    printf("\nproc1 got rs2!!\n");
    pthread_mutex_unlock(&lock2);
    pthread_mutex_unlock(&lock1);
    return 0;
}
```

```
void *proc2(){
    printf("\nThis is proc2 using rs2\n");
    pthread_mutex_lock(&lock2);
    usleep(200);
    printf( "\np2 trying to get rs1...\n");
    pthread_mutex_lock(&lock1);
    test--;
    printf("Test: %d\n", test);
    printf("\nproc2 got rs1!!\n");
    pthread_mutex_unlock(&lock1);
    pthread_mutex_unlock(&lock2);
    return 0;
}
```

```
int main(){
    pthread_t t1,t2;
    pthread_create(&t1,NULL, proc1 , NULL);
    pthread_create(&t2,NULL, proc2 , NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    // will never arrive here
    pthread_mutex_destroy(&lock1);
    pthread_mutex_destroy(&lock2);
}
```

```
brucelin@HC-FERMI:~/pthread_code$ ./deadlock
```

```
This is proc1 using rs1
```

```
This is proc2 using rs2
```

```
p2 trying to get rs1...
```

```
p1 trying to get rs2...
```



Semaphore

- ▶ A tool to generalize the **synchronization** problem
 - ▶ Deadlock may occur if not use appropriately !
- ▶ More specifically...
 - ▶ a record of **how many units of a particular resource** are available
 - ▶ If #record = 1 \square **binary semaphore, mutex lock**
 - ▶ If #record > 1 \square **counting semaphore**
 - ▶ accessed only through 2 atomic operations: **wait** & **signal**
- ▶ Spinlock implementation:
 - ▶ Semaphore is an integer variable

```
wait (S) {                               signal (S) {
    while (S <= 0) ;                       S++;
    S--;                                    }
}
```

Semaphore Example

► shared data:

► **semaphore** S ; // initially S = 1

► Process P_i :

do {

wait (S) ;

critical section

signal (S);

 remainder section

} while (1) ;

POSIX Semaphore

- ▶ pthreads allows the specific creation of **semaphores**
 - ▶ can do **increments** and **decrements** of semaphore value
 - ▶ semaphore can be **initialized to any value**
 - ▶ thread blocks if semaphore value **is less than or equal to zero** when a decrement is attempted
 - ▶ as soon as semaphore value is **greater than zero**, one of the blocked threads wakes up and continues
 - ▶ no guarantees as to which thread this might be

POSIX Semaphore

- ▶ POSIX Semaphore routines:
 - ▶ `sem_init(sem_t *sem, int pshared, unsigned int value)`
 - ▶ `sem_wait(sem_t *sem)`
 - ▶ `sem_post(sem_t *sem)`
 - ▶ `sem_getvalue(sem_t *sem, int *valptr)`
 - ▶ `sem_destroy(sem_t *sem)`

```
#include <semaphore.h>
```

```
sem_t sem;
```

```
/*create a semaphore and initialized to 1 */
```

```
sem_init(&sem, 0, 1);
```

```
/*acquire the semaphore*/
```

```
sem_wait(&sem);
```

```
/*critical section*/
```

```
/*release the semaphore*/
```

```
sem_post(&sem);
```

Creating Semaphores

- ▶ Semaphores are created like other variables
 - ▶ `sem_t semaphore;`
- ▶ Semaphores must be initialized
 - ▶ Prototype:
 - ▶ `int sem_init(sem_t *sem, int pshared, unsigned int value);`
 - ▶ *sem*: the semaphore value to initialize
 - ▶ *pshared*: share semaphore across processes – usually 0
 - ▶ *value*: the initial value of the semaphore

Decrementing a Semaphore

- ▶ Prototype:

- ▶ `int sem_wait(sem_t *sem);`
 - ▶ *sem*: semaphore to try and decrement

- ▶ If the semaphore value is **greater than 0**, the *sem_wait* call **return immediately**

- ▶ otherwise it blocks the calling thread until the value becomes greater than 0

Incrementing a Semaphore

- ▶ Prototype:
 - ▶ `int sem_post(sem_t *sem);`
 - ▶ *sem*: the semaphore to increment
- ▶ Increments the value of the semaphore by 1
 - ▶ if any threads are blocked on the semaphore, they will be unblocked

Example 1

```
#include <stdio.h>
#include <semaphore.h>
#include <pthread.h>
#define MAX_SIZE 5
sem_t empty, full;
void producer(char* buf) {
    int in = 0;
    for(;;) {
        sem_wait(&empty);
        buf[in] = getchar();
        getchar();
        in = (in + 1) % MAX_SIZE;
        sem_post(&full);
    }
}
```

```
void consumer(char* buf) {
    int out = 0;
    for(;;) {
        sem_wait(&full);
        printf("%c\n", buf[out]);
        out = (out + 1) % MAX_SIZE;
        sem_post(&empty);
    }
}

int main() {
    char buffer[MAX_SIZE];
    pthread_t p;
    sem_init(&empty, 0, MAX_SIZE);
    sem_init(&full, 0, 0);
    pthread_create(&p, NULL, (void*)producer, buffer);
    consumer(buffer);
    return 0;
}
```

Example II

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

sem_t semaphore;
int counter = 0;

void* child() {
    for(int i = 0; i < 5; ++i) {
        sem_wait(&semaphore);
        printf("Counter = %d\n", ++counter);
        sleep(1);
    }
    pthread_exit(NULL);
}
```

```
int main(void) {
    sem_init(&semaphore, 0, 0);
    pthread_t t;
    pthread_create(&t, NULL, child, NULL);
    printf("Post 2 jobs.\n");
    sem_post(&semaphore);
    sem_post(&semaphore);
    sleep(4);
    printf("Post 3 jobs.\n");
    sem_post(&semaphore);
    sem_post(&semaphore);
    sem_post(&semaphore);
    pthread_join(t, NULL);
    return 0;
}
```

Parting Notes

- ▶ Very important to get all the ordering right
 - ▶ one simple mistake can lead to problems
 - ▶ no progress
 - ▶ mutual exclusion violation
- ▶ Comparing primitives
 - ▶ Using **mutual exclusion with CV's** is faster than using semaphores
 - ▶ Sometimes semaphores are intuitively simpler

Semaphore Drawback

- ▶ Although semaphores provide a convenient and effective synchronization mechanism, **its correctness is depending on the programmer**
- ▶ All processes access a shared data object must **execute wait() and signal() in the right order and right place**

Condition Variables (CV)

- ▶ CV represent some **condition** that a thread can:
 - ▶ Wait on, until the condition occurs; or
 - ▶ Notify other waiting threads that the condition has occurred
- ▶ Three operations on condition variables:
 - ▶ **wait()** --- **Block** until another thread calls **signal()** or **broadcast()** on the CV
 - ▶ **signal()** --- Wake up **one thread** waiting on the CV
 - ▶ **broadcast()** --- Wake up **all threads** waiting on the CV
- ▶ In Pthread, CV type is a `pthread_cond_t`
 - ▶ Use `pthread_cond_init(&theCV, NULL)` to initialize
 - ▶ `pthread_cond_wait (&theCV, &somelock)`
 - ▶ `pthread_cond_signal (&theCV)`
 - ▶ `pthread_cond_broadcast (&theCV)`

Condition Variables (CV)

- ▶ Notice in the previous example a *spin-lock* was used
 - ▶ wait for a condition to be true
 - ▶ the buffer to be full or empty
 - ▶ spin-locks require CPU time to run
 - ▶ waste of cycles
- ▶ **Condition variables** allow a thread to block until a specific condition becomes true
 - ▶ recall that a blocked process cannot be run
 - ▶ doesn't waste CPU cycles
 - ▶ blocked thread goes to wait queue for condition
- ▶ When the condition becomes true, some other thread signals the blocked thread(s)

Condition Variables (CV)

- ▶ A CV is created like a normal variable
 - ▶ *pthread_cond_t cond;*
- ▶ CVs must be initialized before being used
 - ▶ a CV can only be initialized once
 - ▶ prototype:
 - ▶ *int pthread_cond_init(pthread_cond_t *cv, const pthread_condattr_t *cattr);*
 - ▶ *cv*: a pointer to the condition variable to be initialized
 - ▶ *cattr*: attributes of the condition variable – usually NULL
 - ▶ *pthread_cond_init(&cond, NULL);*
 - ▶ *pthread_cond_t cond = PTHREAD_COND_INITIALIZER;*

Blocking on CV

- ▶ A **wait** call is used to block a thread on a CV
 - ▶ puts the thread on a **wait queue** until it gets a signal that the condition is true
 - ▶ even after signal, condition may still not be true!
 - ▶ blocked thread does not compete for CPU
 - ▶ the **wait** call should occur under the protection of a mutex
 - ▶ this mutex is automatically released by the wait call
 - ▶ the mutex is automatically reclaimed on return from wait call
- ▶ **prototype:**
 - ▶ **`int pthread_cond_wait(pthread_cond_t *cv, pthread_mutex_t *mutex);`**
 - ▶ *cv*: condition variable to block on
 - ▶ *mutex*: the mutex to release while waiting

Signaling a Condition

- ▶ A **signal** call is used to “**wake up**” a single thread waiting on a **condition**
 - ▶ multiple threads may be waiting and there is no guarantee as to which one wakes up first
 - ▶ thread to wake up does not actually wake until the **lock** indicated by the wait call becomes available
 - ▶ condition thread was waiting for may not be true when the thread actually gets to run again
 - ▶ should always do a wait call inside of a while loop
 - ▶ if no waiters on a condition, signaling has no effect
 - ▶ prototype:
 - ▶ **int pthread_cond_signal(pthread_cond_t *cv);**
 - ▶ *cv*: condition variable to signal on

Using Condition Variable

- ▶ Example:
- ▶ The left thread is designed to take action when $x=0$
- ▶ Another thread is responsible for decrementing the counter

```
pthread_cond_t  cond;  
pthread_cond_init (cond, NULL);
```

```
pthread_mutex_t  mutex;  
pthread_mutex_init (mutex, NULL);
```

```
action() {  
    pthread_mutex_lock (&mutex)  
    if (x != 0)  
        pthread_cond_wait (cond, mutex);  
    pthread_mutex_unlock (&mutex);  
    take_action();  
}
```

```
counter() {  
    pthread_mutex_lock (&mutex)  
    x--;  
    if (x==0)  
        pthread_cond_signal (cond);  
    pthread_mutex_unlock (&mutex);  
}
```

- ▶ All condition variable operation MUST be performed while a **mutex** is locked!!!

Why is the lock necessary?

Using Condition Variable

```
pthread_cond_t cond;  
pthread_cond_init (&cond, NULL);
```

```
pthread_mutex_t mutex;  
pthread_mutex_init (&mutex, NULL);
```

```
action() {  
    pthread_mutex_lock (&mutex)  
    while (x != 0)  
        pthread_cond_wait (&cond, &mutex);  
    pthread_mutex_unlock (&mutex);  
    take_action();  
}
```

```
counter() {  
    pthread_mutex_lock (&mutex)  
    x--;  
    if (x==0)  
        pthread_cond_signal (&cond);  
    pthread_mutex_unlock (&mutex);  
}
```

- ▶ Because event counter “x” is a **SHARED** variable
- ▶ If no lock on thread action()...
 - ▶ Wait after any thread (i.e. not counter) sets “x” to 0
- ▶ If no lock on thread counter()...
 - ▶ No guarantee that decrement and test of “x” is **atomic**
- ▶ Requiring CV operations to be done while holding a lock **prevents a lot of common programming mistakes**

Using Condition Variable


```
action() {  
→ pthread_mutex_lock (&mutex)  
  while (x != 0)  
    pthread_cond_wait (cond, mutex);  
  pthread_mutex_unlock (&mutex);  
  take_action();  
}
```

```
→ counter() {  
  thread_mutex_lock (&mutex)  
  x--;  
  if (x==0)  
    pthread_cond_signal (cond);  
  pthread_mutex_unlock (&mutex);  
}
```

- ▶ What really happens...
 - ▶ 1. Lock mutex

Using Condition Variable

```
action() {  
    pthread_mutex_lock (&mutex)  
    while (x != 0)  
        → pthread_cond_wait (cond, mutex);  
    pthread_mutex_unlock (&mutex);  
    take_action();  
}
```

```
thread_mutex_lock (&mutex)  
x--;  
if (x==0)  
    pthread_cond_signal (cond);  
pthread_mutex_unlock (&mutex);  
}
```

- ▶ What really happens...
 - ▶ 1. Lock mutex
 - ▶ 2. Wait()
 - ▶ 1. Put the thread into sleep(into waiting queue) & releases the lock(mutex)

- ▶ 1. Lock mutex

Using Condition Variable

```
action() {  
    pthread_mutex_lock (&mutex)  
    while (x != 0)  
        → pthread_cond_wait (cond, mutex);  
    pthread_mutex_unlock (&mutex);  
    take_action();  
}
```

```
counter() {  
    thread_mutex_lock (&mutex)  
    x--;  
    if (x==0)  
        → pthread_cond_signal (cond);  
    pthread_mutex_unlock (&mutex);  
}
```

► What really happens...

- 1. Lock mutex
- 2. Wait()
 - 1. Put the thread into sleep & releases the lock
 - 2. Waked up, but the thread is locked

1. Lock mutex
2. Signal()

Using Condition Variable

```
action() {  
    pthread_mutex_lock (&mutex)  
    while (x != 0)  
        pthread_cond_wait (cond, mutex);  
    pthread_mutex_unlock (&mutex);  
    take_action();  
}
```

```
counter() {  
    thread_mutex_lock (&mutex)  
    x--;  
    if (x==0)  
        pthread_cond_signal (cond);  
    pthread_mutex_unlock (&mutex);  
}
```

► What really happens...

- 1. Lock mutex
 - 2. Wait()
 - 1. Put the thread into sleep & releases the lock
 - 2. Waked up, but the thread is locked
 - 3. Re-acquire lock and resume execution
1. Lock mutex
 2. Signal()
 3. Releases the lock

Using Condition Variable

```
action() {  
    pthread_mutex_lock (&mutex)  
    while (x != 0)  
        pthread_cond_wait (cond, mutex);  
→ pthread_mutex_unlock (&mutex);  
    take_action();  
}
```

```
counter() {  
    thread_mutex_lock (&mutex)  
    x--;  
    if (x==0)  
        pthread_cond_signal (cond);  
    pthread_mutex_unlock (&mutex);  
→ }
```

► What really happens...

- 1. Lock mutex
- 2. Wait()
 - 1. Put the thread into **sleep & releases the lock**
 - 2. **Waked up**, but the thread is locked
 - 3. Re-acquire lock and resume execution
- 3. Release the lock

1. Lock mutex
2. Signal()
3. Releases the lock

Using Condition Variable

```
action() {  
    pthread_mutex_lock (&mutex)  
    while (x != 0)  
        pthread_cond_wait (cond, mutex);  
    pthread_mutex_unlock (&mutex);  
    take_action();  
}
```

```
counter() {  
    thread_mutex_lock (&mutex)  
    x--;  
    if (x==0)  
        pthread_cond_signal (cond);  
    pthread_mutex_unlock (&mutex);  
}
```

► What really happens...

- 1. Lock mutex
- 2. Wait()

- 1. Put the thread into sleep & releases the lock
 - 2. Waked up, but the thread is locked
 - 3. Re-acquire lock and resume execution

- 3. Release the lock

1. Lock mutex
2. Signal()
3. Releases the lock

Another reason why condition variable op. MUST within mutex lock

Example 1

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

pthread_mutex_t mutex;
pthread_cond_t cond;
int x = 5;

void action(void) {
    pthread_mutex_lock(&mutex);
    if(x!=0){
        pthread_cond_wait(&cond, &mutex);
    }
    pthread_mutex_unlock(&mutex);
    printf("Take action!\n");
}
```

```
void counter(void) {
    pthread_mutex_lock(&mutex);
    while(x!=0){
        x--;
        printf("x:%d\n",x);
    }
    if(x==0){
        pthread_cond_signal(&cond);
    }
    pthread_mutex_unlock(&mutex);
    printf("counter release!\n");
}
```

```
int main() {
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond, NULL);
    pthread_t p;

    pthread_create(&p, NULL, (void*)counter, NULL);
    action();
    return 0;
}
```

\$./cv

x:4

x:3

x:2

x:1

x:0

counter release!
Take action!

Example 2

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

#define MAX_SIZE 5
pthread_mutex_t lock;
pthread_cond_t notFull, notEmpty;
int count;

void producer(char* buf) {
    for(;;) {
        sleep(1); // let consumer to get mutex
        pthread_mutex_lock(&lock);
        while(count == MAX_SIZE)
            pthread_cond_wait(&notFull, &lock);
        printf("enter a char:");
        buf[count] = getchar();
        getchar();
        count++;
        pthread_cond_signal(&notEmpty);
        pthread_mutex_unlock(&lock);
    }
}
```

```
void consumer(char* buf) {
    for(;;) {
        pthread_mutex_lock(&lock);
        while(count == 0)
            pthread_cond_wait(&notEmpty, &lock);
        printf("output buffer:");
        printf("%c\n", buf[count-1]);
        count--;
        pthread_cond_signal(&notFull);
        pthread_mutex_unlock(&lock);
    }
}

int main() {
    char buffer[MAX_SIZE];
    pthread_t p;
    count = 0;
    pthread_mutex_init(&lock, NULL);
    pthread_cond_init(&notFull, NULL);
    pthread_cond_init(&notEmpty, NULL);

    pthread_create(&p, NULL, (void*)producer, buffer);
    consumer(buffer);
    return 0;
}
```


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
- 隱性線程(**Implicit Threading**)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