

CS 7172

Parallel and Distributed Computing

POSIX Threads Programming

Kun Suo

Computer Science, Kennesaw State University

<https://kevinsuo.github.io/>

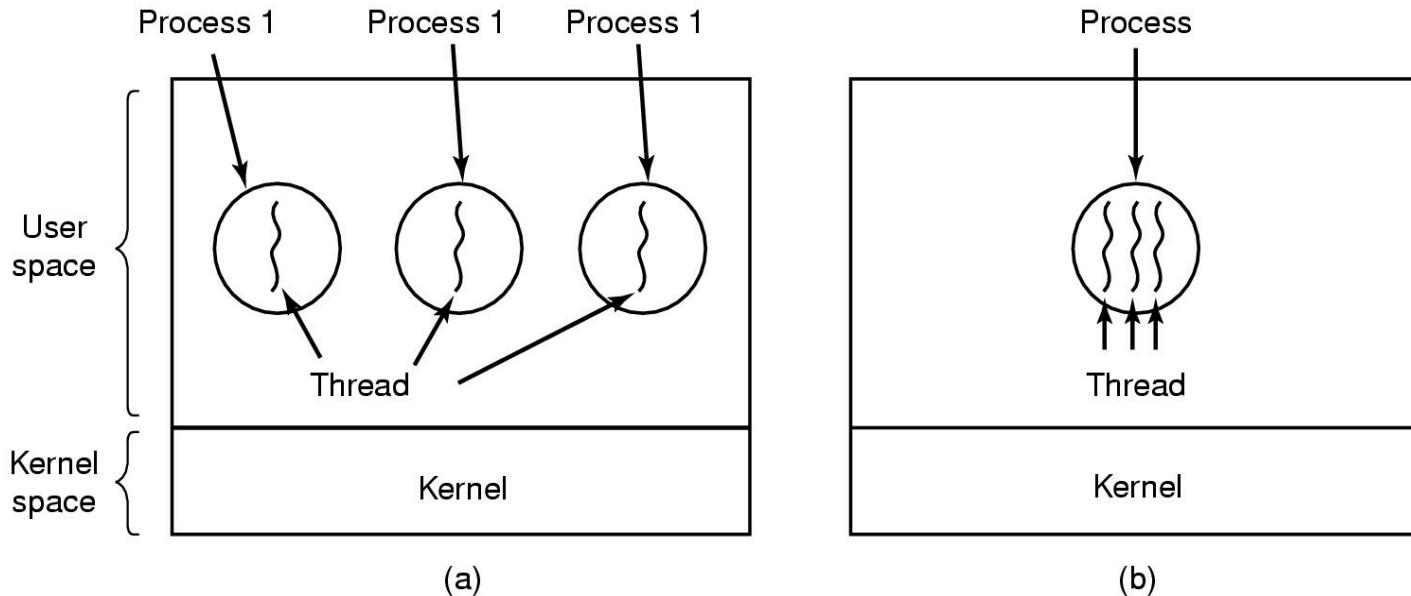
Outline

- What is pthread?
 - Thread model
 - Pthread overview
- Pthread management
 - Creation and termination
 - Identification
 - Join and detach
- Pthread data sharing
 - Mutex
 - Condition variable
 - Barrier
 - Semaphore



The Thread Model

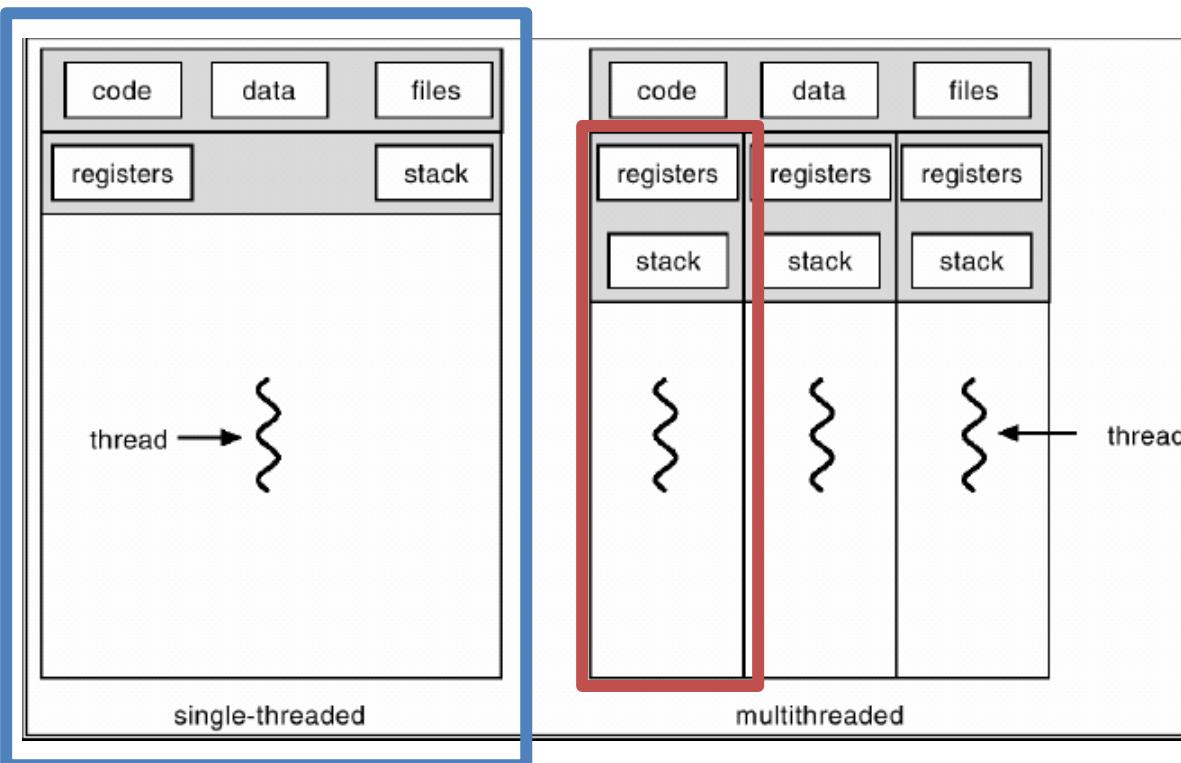
- Process: for resource grouping and execution
- Thread: a finer-granularity entity for execution and parallelism



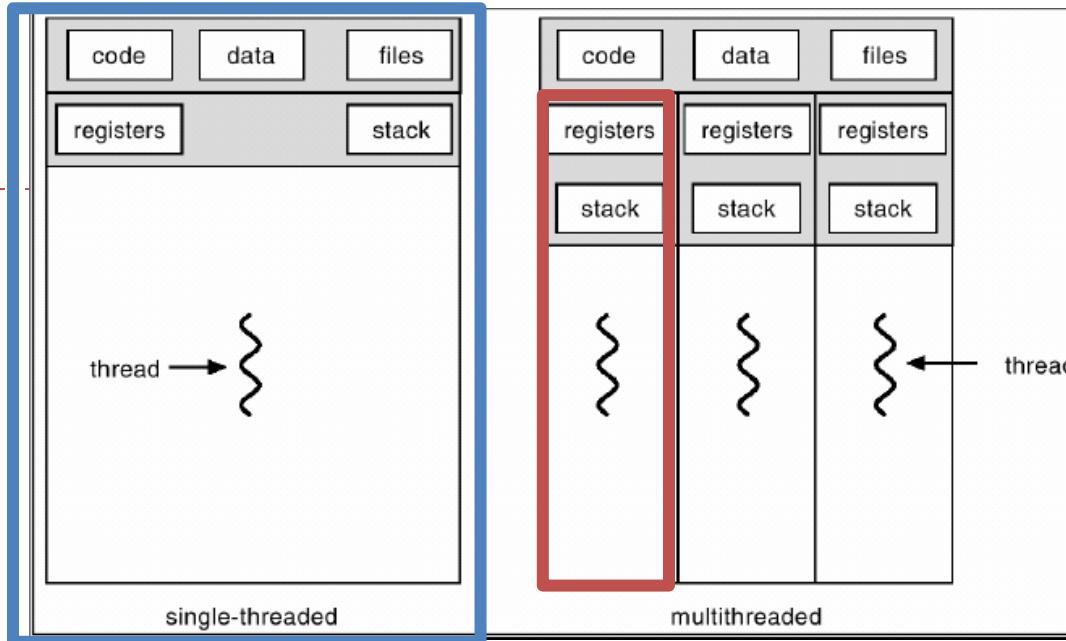
(a) Three processes each with one thread, but **different address spaces**

(b) One process with three threads, **sharing the address space**

The Thread Model



Process	Thread
Have data/code/heap	Share code, data, heap
Include at least one thread	Multiple can coexist in a process
Have own address space, isolated from other processes	Only have own stack and registers

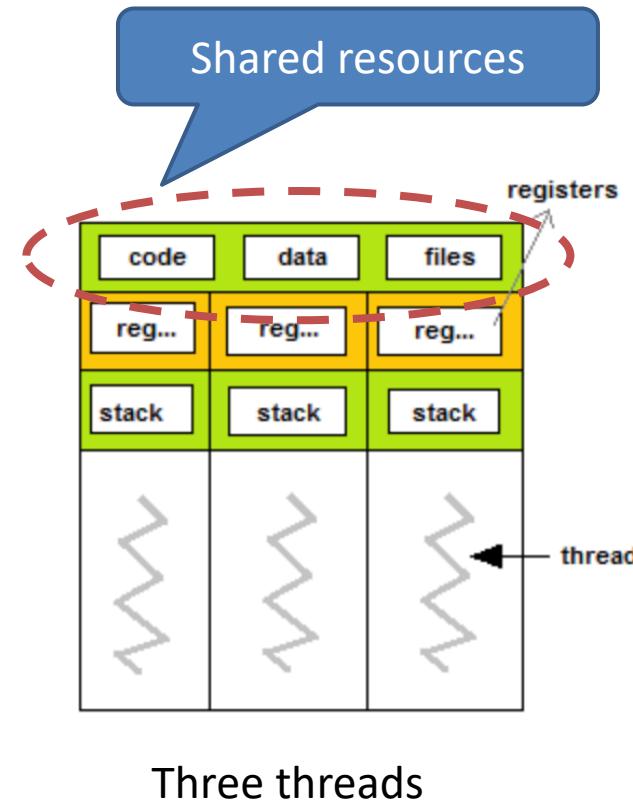


Per process items	Per thread items
<ul style="list-style-type: none"> Address space Global variables Open files Child processes Pending alarms Signals and signal handlers Accounting information 	<ul style="list-style-type: none"> Program counter Registers Stack State

Items shared by all threads in a process

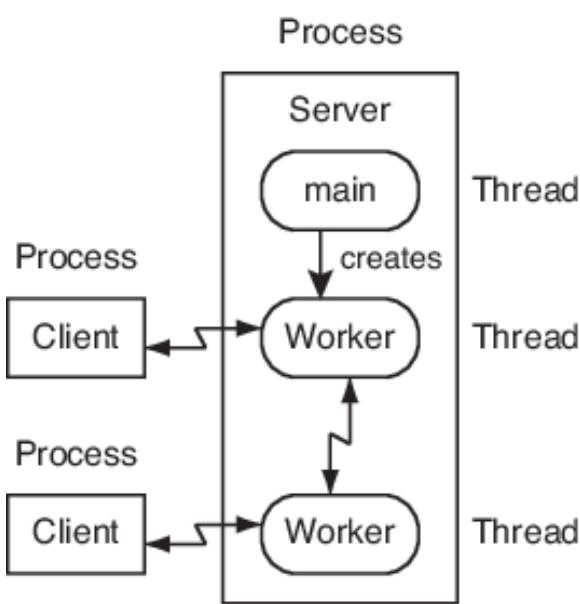
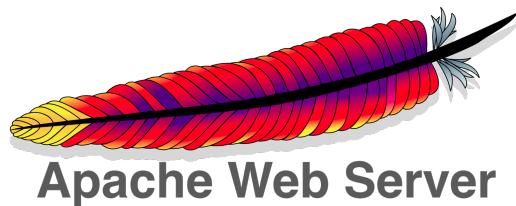
Items private to each thread

The Thread Model

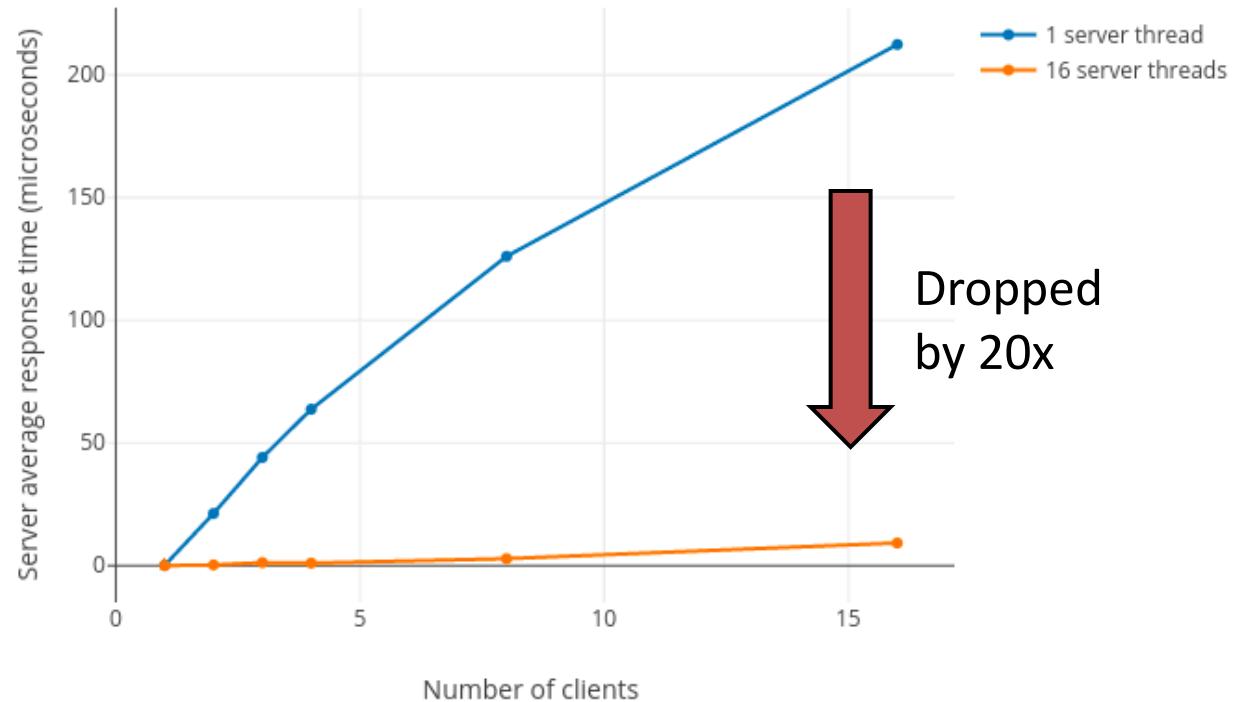


- Because threads within the same process share resources
 - Changes made by one thread to **shared system resources** (closing a file) will be seen by all other threads
 - Two pointers having the same value point to the **same data**
 - Reading and writing to the **same memory location** is possible, and therefore requires explicit synchronization by the programmer!

Multi-threads for application performance

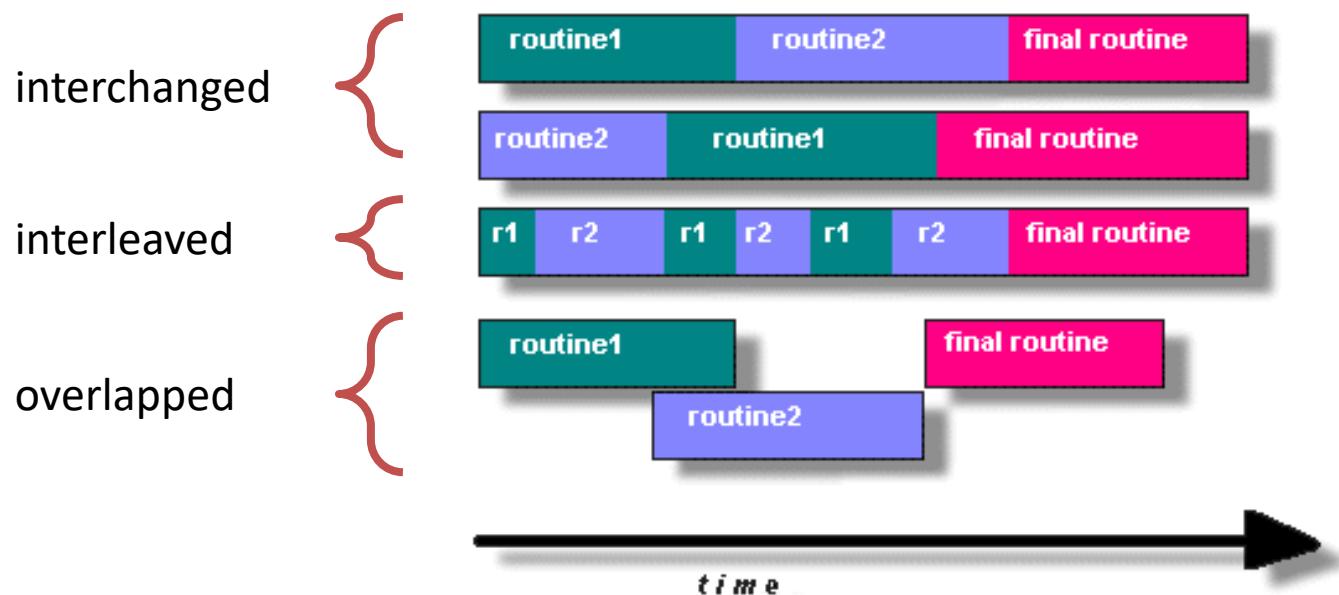


Multi-threaded performance vs. single-threaded (test1)



Design Programs into high performance multi-threads

- A program must be able to be organized into discrete, independent tasks which can execute concurrently
 - E.g.: routine1 and routine2 can be interchanged, interleaved, and/or overlapped in real time
 - Thread-safeness: race conditions

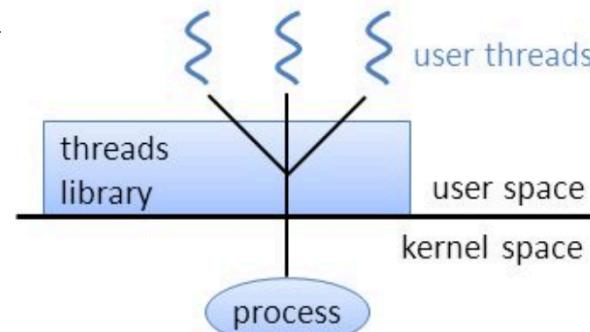


Threading Models

- N:1 (User-level threading)

- GNU Portable Threads

<https://www.gnu.org/software/pth/>

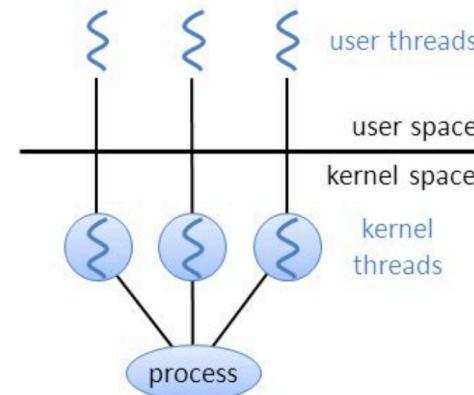


- 1:1 (Kernel-level threading)

- Native POSIX Thread Library (Pthread)

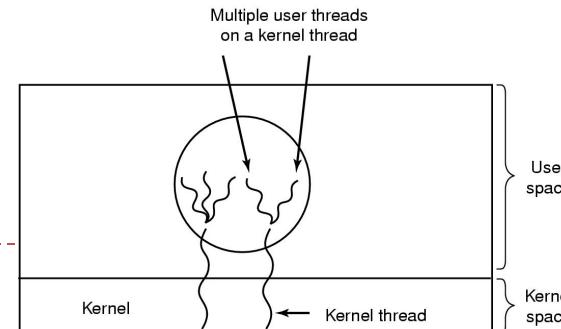
https://en.wikipedia.org/wiki/Native_POSIX_Thread_Library

Linux/MacOS/Windows



- M:N (Hybrid threading)

- Solaris <https://www.oracle.com/solaris/solaris11/>



Pthreads Overview

- What are Pthreads?
 - An IEEE standardized thread programming interface (IEEE POSIX 1003.1c)
 - <http://www.open-std.org/jtc1/sc22/WG15/>
 - POSIX (Portable Operating System Interface) threads
 - Defined as a set of C language programming types and procedure calls, implemented with a pthread.h header/include file and a thread library

To software developer:

a thread is a “procedure” that runs independently from its main program



Why Pthreads ?

- Performance!
 - Lightweight
 - Communication
 - Overlapping CPU work with I/O
 - Fine granularity of concurrency
 - Priority/real-time scheduling
 - Asynchronous event handling

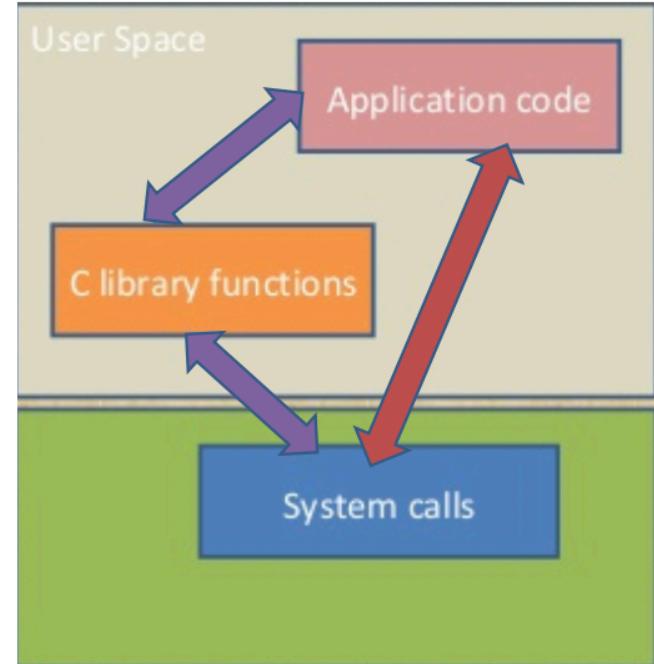


Why Pthreads ?

Time in second with running 50000
fork() or Pthread_create()

Pthread_create()
in the user space

Clone()
in the kernel



10~50 times faster

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
IBM 1.9 GHz POWER5 p5-575	50.66	3.32	42.75	1.13	0.54	0.75
INTEL 2.4 GHz Xeon	23.81	3.12	8.97	1.70	0.53	0.30
INTEL 1.4 GHz Itanium 2	23.61	0.12	3.42	2.10	0.04	0.01

Outline

- What is pthread?
 - Thread model
 - Pthread overview
- Pthread management
 - Creation and termination
 - Identification
 - Join and detach
- Pthread data sharing
 - Mutex
 - Condition variable
 - Barrier
 - Semaphore



The Pthreads API

- The API is defined in the ANSI/IEEE POSIX 1003.1 – 1995
 - Naming conventions: all identifiers in the library begins with `pthread_`
 - Three major classes of subroutines
 - ▶ Thread management, mutexes, condition variables

Routine Prefix	Functional Group
<code>pthread_</code>	Threads themselves and miscellaneous subroutines
<code>pthread_attr_</code>	Thread attributes objects
<code>pthread_mutex_</code>	Mutexes
<code>pthread_mutexattr_</code>	Mutex attributes objects.
<code>pthread_cond_</code>	Condition variables
<code>pthread_condattr_</code>	Condition attributes objects
<code>pthread_key_</code>	Thread-specific data keys



Compiling Pthreads Programs

Platform	Compiler Command	Description
IBM AIX	<code>xlc_r / cc_r</code>	C (ANSI / non-ANSI)
	<code>xlC_r</code>	C++
	<code>xlf_r -qnosave</code> <code>xlf90_r -qnosave</code>	Fortran - using IBM's Pthreads API (non-portable)
INTEL LINUX	<code>icc -pthread</code>	C
	<code>icpc -pthread</code>	C++
COMPAQ Tru64	<code>cc -pthread</code>	C
	<code>cxx -pthread</code>	C++
All Above Platforms	<code>gcc -lpthread/pthread</code>	GNU C
	<code>g++ -lpthread/pthread</code>	GNU C++
	<code>guidec -pthread</code>	KAI C (if installed)
	<code>KCC -pthread</code>	KAI C++ (if installed)



```

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

void *myThread1(void)
{
    int i;
    for (i=0; i<3; i++)
    {
        printf("This is the 1st pthread.\n");
        sleep(1);
    }
}

int main()
{
    int ret=0;
    pthread_t id1;

    printf("This is main thread!\n");

    ret = pthread_create(&id1, NULL, (void*)myThread1, NULL);
    if (ret)
    {
        printf("Create pthread error!/\n");
        return 1;
    }

    pthread_exit(NULL);

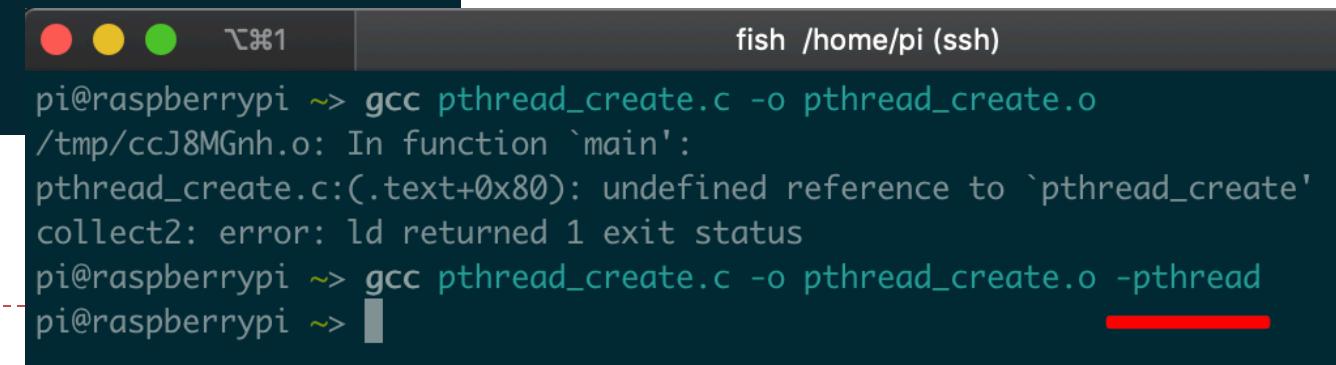
    return 0;
}

```

Compiling Pthreads Programs

[https://github.com/kevinsuo/CS7172
blob/master/pthread_create.c](https://github.com/kevinsuo/CS7172/blob/master/pthread_create.c)

\$ gcc file.c -o file.o -pthread



A terminal window titled "fish /home/pi (ssh)" is shown. The command "gcc pthread_create.c -o pthread_create.o" is entered, followed by the output of the compilation process. The output shows an undefined reference to 'pthread_create' and a collect2 error indicating ld returned 1 exit status.

```

pi@raspberrypi ~> gcc pthread_create.c -o pthread_create.o
/tmp/ccJ8MGnh.o: In function `main':
pthread_create.c:(.text+0x80): undefined reference to `pthread_create'
collect2: error: ld returned 1 exit status
pi@raspberrypi ~> gcc pthread_create.c -o pthread_create.o -pthread
pi@raspberrypi ~>

```



1, Creation & Termination

pthread_create (threadid, attr, start_routine, arg)

- creates a thread and makes it executable;
- the third parameter is just the name of the function for the thread to run.
- *arg* must be passed by reference as a pointer cast of type void

pthread_exit (status)

- If main() finishes before the threads it has created, and exists with the pthread_exit(), the other threads will continue to execute. Otherwise, they will be automatically terminated when main() finishes

Initially, your main() program comprises a single, default thread.



1, Creation & Termination example

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

void *myThread1(void)
{
    int i;
    for (i=0; i<3; i++)
    {
        printf("This is the 1st pthread.\n");
        sleep(1);
    }
}

int main()
{
    int ret=0;
    pthread_t id1;

    printf("This is main thread!\n");

    ret = pthread_create(&id1, NULL, (void*)myThread1, NULL);
    if (ret)
    {
        printf("Create pthread error!\n");
        return 1;
    }

    pthread_exit(NULL);

    return 0;
}
```

[https://github.com/kevinsuo/CS7172
/blob/master\(pthread_create.c](https://github.com/kevinsuo/CS7172/blob/master(pthread_create.c)

```
pi@raspberrypi ~/Downloads> ./test.o
This is main thread!
This is the 1st pthread.
This is the 1st pthread.
This is the 1st pthread.
```

Without it: Child threads will end if
the main thread ends.

1, Creation & Termination example

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

void *myThread1(void)
{
    int i;
    for (i=0; i<3; i++)
    {
        printf("This is the 1st pthread.\n");
        sleep(1);
    }
}

int main()
{
    int ret=0;
    pthread_t id1;

    printf("This is main thread!\n");

    ret = pthread_create(&id1, NULL, (void*)myThread1, NULL);
    if (ret)
    {
        printf("Create pthread error!\n");
        return 1;
    }

    // pthread_exit(NULL);

    return 0;
}
```

```
pi@raspberrypi ~/Downloads> ./test.o
This is main thread!
This is the 1st pthread. ↴
```

Without it: Child threads will end if the main thread ends.

2, identification

pthread_self (void)

- returns the ID of the thread in which it is invoked.

- A thread can obtain its own thread ID by calling the **pthread_self()** function



2, identification

[https://github.com/kevinsuo/CS7172/
blob/master/pthread_self.c](https://github.com/kevinsuo/CS7172/blob/master/pthread_self.c)

```
void *myThread(void)
{
    pid_t pid;
    pthread_t tid;

    printf("thread pid %u tid %u \n",
           (unsigned int)getpid(), (unsigned int)pthread_self());
}

int main()
{
    int ret=0;
    pthread_t id1, id2;

    printf("This is main thread pid %u!\n", (unsigned int)getpid());

    ret = pthread_create(&id1, NULL, (void*)myThread, NULL);
    ret = pthread_create(&id2, NULL, (void*)myThread, NULL);
    if (ret)
    {
        printf("Create pthread error!/\n");
        return 1;
    }

    pthread_join(id1, (void *)ret);
    pthread_join(id2, (void *)ret);

    pthread_exit(NULL);

    return 0;
}
```

pid cannot distinguish the threads.

```
pi@raspberrypi ~/Downloads> ./test.o
This is main thread pid 2288!
thread pid 2288 tid 1994093680
thread pid 2288 tid 1985700976
```

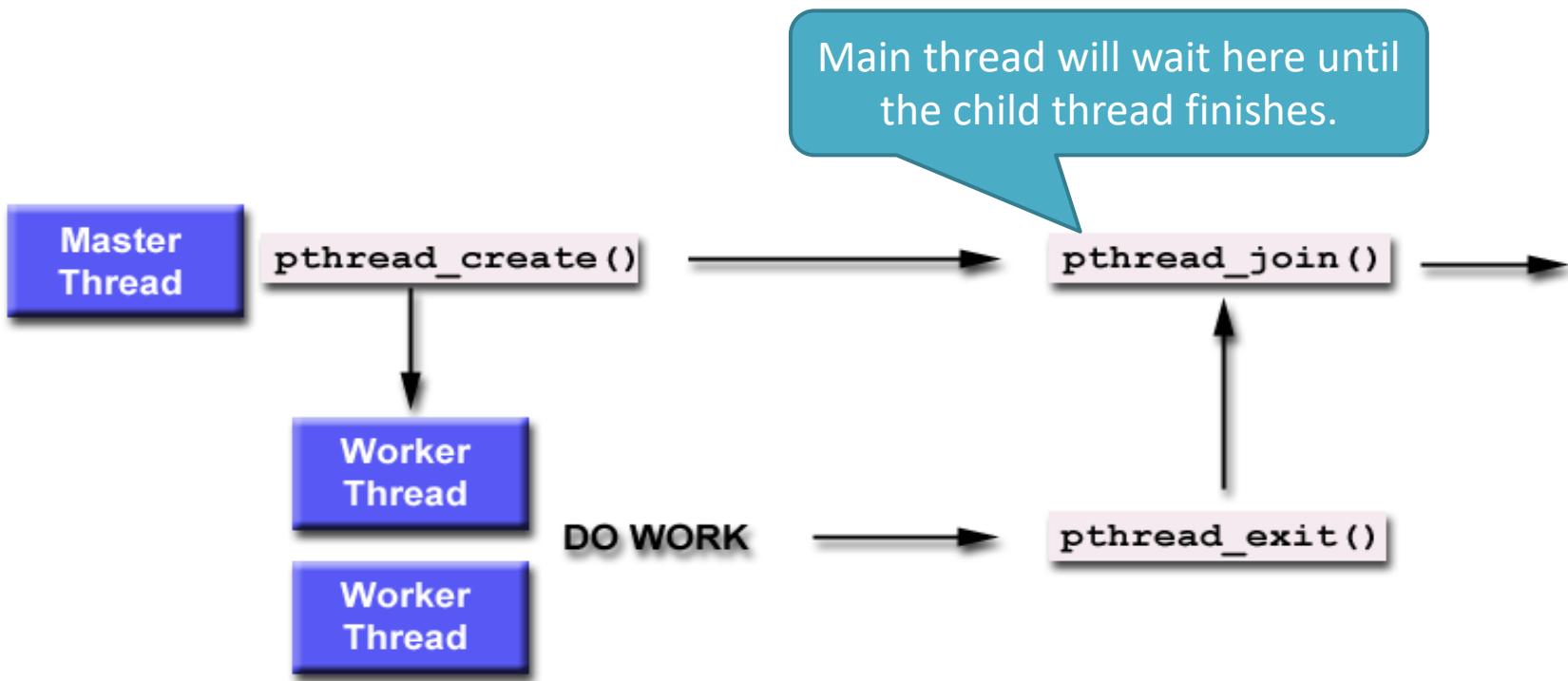


3, join

pthread_join (threadid, status)

* Joining is one way to accomplish synchronization between threads: The `pthread_join()` subroutine blocks the calling thread until the specified `threadid` thread terminates.

The main thread will be blocked here to wait the child thread to finish



3, join example

[https://github.com/kevinsuo/CS7172
/blob/master/pthread_join.c](https://github.com/kevinsuo/CS7172/blob/master/pthread_join.c)

```
int count = 0;

void *myThread1(void)
{
    int i;
    for (i=0; i<3; i++)
    {
        printf("This is the 1st pthread.\n");
        count++;
        sleep(1);
    }
}

int main()
{
    int ret=0;
    pthread_t id1;

    printf("This is main thread!\n");

    ret = pthread_create(&id1, NULL, (void*)myThread1, NULL);
    if (ret)
    {
        printf("Create pthread error!/\n");
        return 1;
    }

    // pthread_join(id1, (void *)ret);

    printf("count:%d\n", count);
    pthread_exit(NULL);

    return 0;
}
```

```
pi@raspberrypi ~/Downloads> ./test.o
This is main thread!
This is the 1st pthread.
count:0
This is the 1st pthread.
This is the 1st pthread.
```

Count is 0 as the main thread
keeps executing.

3, join example

```
int count = 0;

void *myThread1(void)
{
    int i;
    for (i=0; i<3; i++)
    {
        printf("This is the 1st pthread.\n");
        count++;
        sleep(1);
    }
}

int main()
{
    int ret=0;
    pthread_t id1;

    printf("This is main thread!\n");

    ret = pthread_create(&id1, NULL, (void*)myThread1, NULL);
    if (ret)
    {
        printf("Create pthread error!\n");
        return 1;
    }

    pthread_join(id1, (void *)ret);

    printf("count:%d\n", count);
    pthread_exit(NULL);

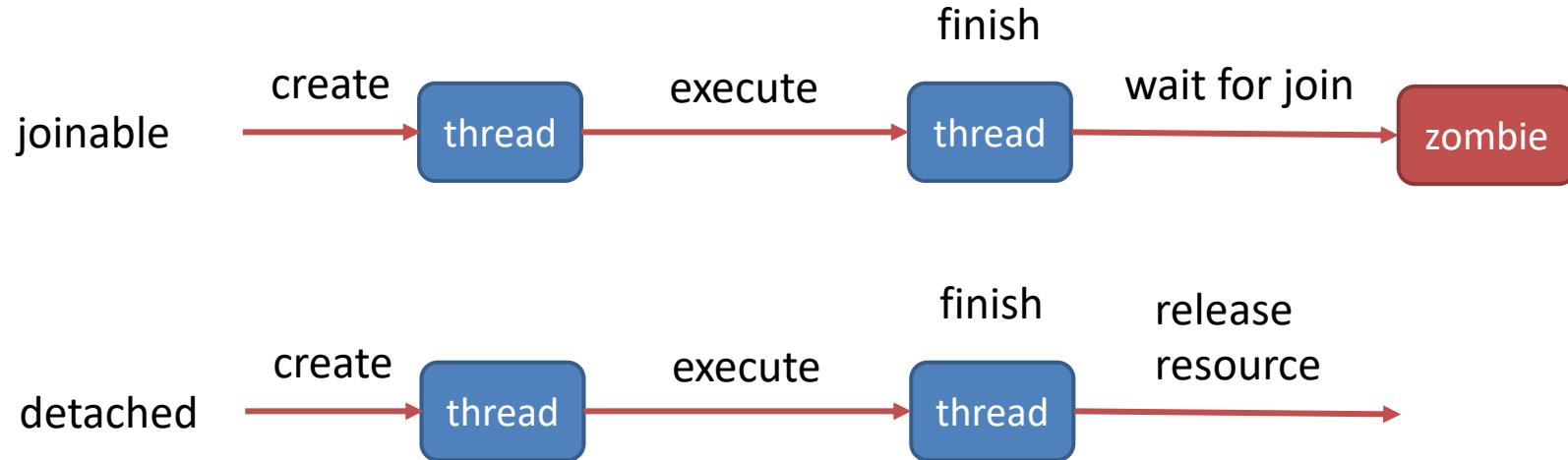
    return 0;
}
```

```
pi@raspberrypi ~/Downloads> ./test.o
This is main thread!
This is the 1st pthread.
This is the 1st pthread.
This is the 1st pthread.
count:3
```

Count is 3 as the main thread is blocked here until child thread finished.

4, detach

- The default state of a thread is **joinable**. If a thread finishes but is not joined, its state is similar to the zombie process, which is, some resources are not recycled.



4, detach example

[https://github.com/kevinsuo/CS7172/b
lob/master/pthread_detach.c](https://github.com/kevinsuo/CS7172/blob/master/pthread_detach.c)

Resources will not be released after execution.

```
void *myThread(void)
{
//    pthread_detach(pthread_self());
    printf("This is child thread tid %u!\n", (unsigned int)pthread_self());
}
```

```
int main()
{
    int ret=0;
    pthread_t id1, id2;

    printf("This is main thread pid %u!\n", (unsigned int)getpid());

    ret = pthread_create(&id1, NULL, (void*)myThread, NULL);
    ret = pthread_create(&id2, NULL, (void*)myThread, NULL);
    if (ret)
    {
        printf("Create pthread error!/\n");
        return 1;
    }

    pthread_join(id1, (void *)ret);
    pthread_join(id2, (void *)ret);

    pthread_exit(NULL);

    return 0;
}
```

```
pi@raspberrypi ~/Downloads> ./test.o
This is main thread pid 2356!
This is child thread tid 1994404976!
This is child thread tid 1986012272!
```



4, detach example

[https://github.com/kevinsuo/CS7172/b
lob/master/thread_detach.c](https://github.com/kevinsuo/CS7172/blob/master(pthread_detach.c)

```
void *myThread(void)
{
    pthread_detach(pthread_self());
    printf("This is child thread tid %u!\n", (unsigned int)pthread_self());
}
```

Set itself not joinable
Resources will be released.

```
int main()
{
    int ret=0;
    pthread_t id1, id2;

    printf("This is main thread pid %u!\n", (unsigned int)getpid());

    ret = pthread_create(&id1, NULL, (void*)myThread, NULL);
    ret = pthread_create(&id2, NULL, (void*)myThread, NULL);
    if (ret)
    {
        printf("Create pthread error!/\n");
        return 1;
    }

    pthread_join(id1, (void *)ret);
    pthread_join(id2, (void *)ret);

    pthread_exit(NULL);

    return 0;
}
```

```
pi@raspberrypi ~/Downloads> ./test.o
This is main thread pid 2356!
This is child thread tid 1994404976!
This is child thread tid 1986012272!
```



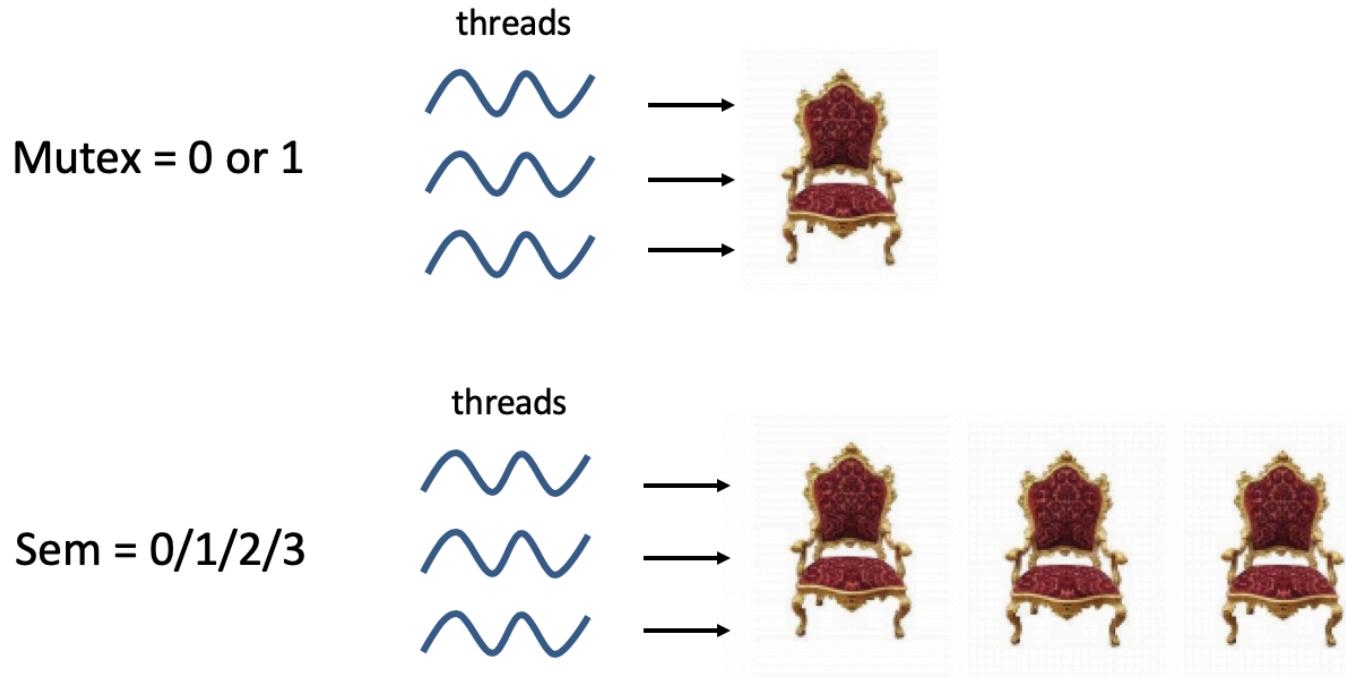
Outline

- What is pthread?
 - Thread model
 - Pthread overview
- Pthread management
 - Creation and termination
 - Identification
 - Join and detach
- Pthread data sharing
 - Mutex
 - Condition variable
 - Barrier
 - Semaphore



5, mutex

- Mutex: a simplified version of the semaphores
 - A variable that can be in one of two states: unlocked or locked
 - Supports synchronization by controlling access to shared data



5, mutex

- A typical sequence in the use of a mutex
 1. Create and initialize a mutex variable
 2. Several threads attempt to lock the mutex
 - ▶ Only one succeeds and that thread owns the mutex
 - ▶ The owner thread performs some set of actions
 3. The owner unlocks the mutex
 4. Another thread acquires the mutex and repeats the process
 5. Finally the mutex is destroyed

```
int counter = 0;
static pthread_mutex_t mlock; 1

void *compute()
{
    int i = 0;
    2 pthread_mutex_lock(&mlock);
    while (i < 10000) {
        counter = counter + 1;
        i++;
    }
    3 pthread_mutex_unlock(&mlock);
    printf("Counter value: %d\n", counter);
}

pthread_t thread1, thread2, thread3, thread4, thread5;

pthread_create(&thread1, NULL, compute, (void *)&thread1);
pthread_create(&thread2, NULL, compute, (void *)&thread2);
pthread_create(&thread3, NULL, compute, (void *)&thread3);
pthread_create(&thread4, NULL, compute, (void *)&thread4);
pthread_create(&thread5, NULL, compute, (void *)&thread5);
```

5, mutex – Creating and Destroying Mutexes

[pthread_mutex_init](#) (mutex, attr)

[pthread_mutex_destroy](#) (mutex)

[pthread_mutexattr_init](#) (attr)

[pthread_mutexattr_destroy](#) (attr)

1. Mutex variables must be declared with type `pthread_mutex_t`, and must be initialized before can be used. The mutex is initially not locked.

Statically: `pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER`

Dynamically, with [pthread_mutex_init](#) (mutex, attr)

2. The attr object must be declared with type `pthread_mutexattr_t`
3. Programmers should free a mutex object that is no longer used



```

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

int counter = 0;
static pthread_mutex_t mlock;

void *compute()
{
    int i = 0;
    pthread_mutex_lock(&mlock);
    while (i < 10000) {
        counter = counter + 1;
        i++;
    }
    pthread_mutex_unlock(&mlock);
    printf("Counter value: %d\n", counter);
}

int main()
{
    pthread_t thread1, thread2, thread3, thread4, thread5;
    if (pthread_mutex_init(&mlock, NULL) != 0)
    {
        printf("mutex init failed\n");
        return 1;
    }

    pthread_create(&thread1, NULL, compute, (void *)&thread1);
    pthread_create(&thread2, NULL, compute, (void *)&thread2);
    pthread_create(&thread3, NULL, compute, (void *)&thread3);
    pthread_create(&thread4, NULL, compute, (void *)&thread4);
    pthread_create(&thread5, NULL, compute, (void *)&thread5);

    pthread_exit(NULL);
    pthread_mutex_destroy(&mlock);
    exit(0);
}

```

<https://github.com/kevinsuo/CS7172/blob/master/mutexlock.c>

```

pi@raspberrypi ~$ ./mutexlock.o
Counter value: 10000
Counter value: 20000
Counter value: 30000
Counter value: 40000
Counter value: 50000

```

Init the mutex lock

destroy the mutex lock

5, mutex – Locking and Unlocking of Mutexes

`pthread_mutex_lock` (mutex) ; P(down)

`pthread_mutex_trylock` (mutex)

`pthread_mutexattr_unlock` (mutex) ; V(up)

1. `pthread_mutex_lock` (mutex) is a blocking call.
2. `pthread_mutex_trylock` (mutex) is a non-blocking call, useful in preventing the deadlock conditions (priority-inversion problem)
3. If you use multiple mutexes, the order is important;

```

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

int counter = 0;
static pthread_mutex_t mlock;

void *compute()
{
    int i = 0;
    pthread_mutex_lock(&mlock);
    while (i < 10000) {
        counter = counter + 1;
        i++;
    }
    pthread_mutex_unlock(&mlock);
    printf("Counter value: %d\n", counter);
}

int main()
{
    pthread_t thread1, thread2, thread3, thread4, thread5;
    if (pthread_mutex_init(&mlock, NULL) != 0)
    {
        printf("mutex init failed\n");
        return 1;
    }

    pthread_create(&thread1, NULL, compute, (void *)&thread1);
    pthread_create(&thread2, NULL, compute, (void *)&thread2);
    pthread_create(&thread3, NULL, compute, (void *)&thread3);
    pthread_create(&thread4, NULL, compute, (void *)&thread4);
    pthread_create(&thread5, NULL, compute, (void *)&thread5);

    pthread_exit(NULL);
    pthread_mutex_destroy(&mlock);
    exit(0);
}

```

Lock the “counter”

Unlock the “counter”

<https://github.com/kevinsuo/CS7172/blob/master/mutexlock.c>



Critical section

```

pi@raspberrypi ~/Downloads> ./mutexlock.o
Counter value: 10000
Counter value: 20000
Counter value: 30000
Counter value: 40000
Counter value: 50000

```

pthread_mutex_lock vs. pthread_mutex_trylock

- `pthread_mutex_lock` (mutex) is a blocking call.

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

int main(void)
{
    int i = 0;
    int res;

    pthread_mutex_t lock;
    pthread_mutex_init(&lock, NULL);

    pthread_mutex_lock(&lock);
    if((res = pthread_mutex_lock(&lock) != 0)) {
        printf("don't lock\n");
    } else {
        printf("locked\n");
    }

    return 0;
}
```

A terminal window titled "fish /home/pi/test (ssh)" shows the following command sequence:

```
pi@raspberrypi ~/test> gcc test.c -o test.o -lpthread
pi@raspberrypi ~/test> ./test.o
```

The window has three tabs at the top: red, yellow, and green. The green tab is active.

1st lock. Get the lock.

2nd lock. Blocked here.



pthread_mutex_lock vs. pthread_mutex_trylock

- `pthread_mutex_trylock` (mutex) is a non-blocking call.

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

int main(void)
{
    int i = 0;
    int res;

    pthread_mutex_t lock;
    pthread_mutex_init(&lock, NULL);

    pthread_mutex_trylock(&lock);
    if((res = pthread_mutex_trylock(&lock) != 0)) {
        printf("don't lock\n");
    } else {
        printf("locked\n");
    }

    return 0;
}
```

```
pi@raspberrypi ~/test> gcc test.c -o test.o -lpthread
pi@raspberrypi ~/test> ./test.o
don't lock
pi@raspberrypi ~/test>
```

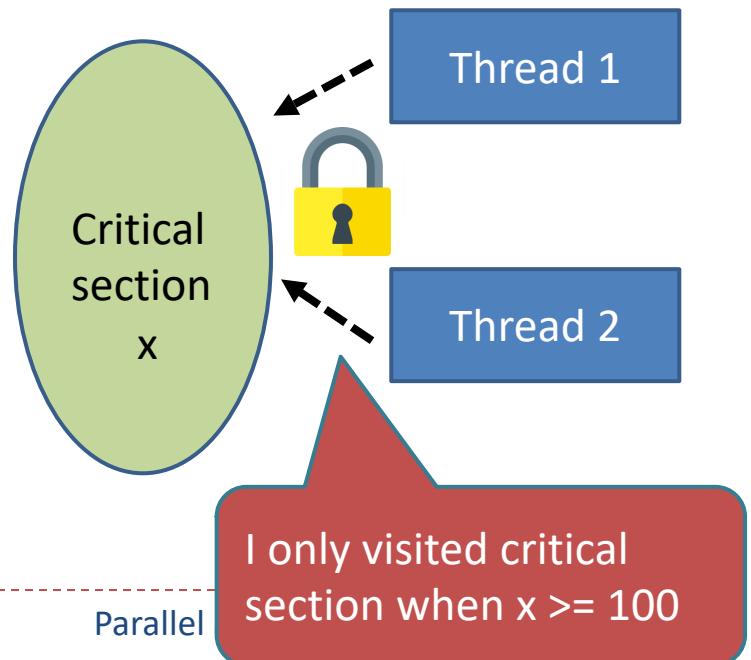
1st lock. Get the lock.

2nd lock. Not blocked here.

[https://github.com/kevinsuo/CS7172
blob/master/trylock.c](https://github.com/kevinsuo/CS7172/blob/master/trylock.c)

6, condition variable

- Mutexes: support synchronization by controlling thread access to data (**without conditions**)
- Condition variables: another way for threads to synchronize (**with conditions**)
- Example:
 - T1: increase x every time;
 - T2: when x is larger than 99, then set x to 0;



6, condition variable example

- Mutex without condition

- Example: T1: increase x every time;
- T2: when x is larger than 99, then set x to 0;

```
//thread 1:  
  
while(true)  
{  
    pthread_mutex_lock(&mutex);  
    iCount++;  
    pthread_mutex_unlock(&mutex);  
}
```

```
//thread 2:  
while(true)  
{  
    pthread_mutex_lock(&mutex);  
    if(iCount >= 100)  
    {  
        iCount = 0;  
    }  
    pthread_mutex_unlock(&mutex);  
}
```

T2 needs to:
lock;
determine;
unlock;
every time to check

6, condition variable – Waiting and Signaling

Con. Variables

[pthread_cond_wait](#) (condition, mutex)

[pthread_cond_signal](#) (condition)

[pthread_cond_broadcast](#) (condition)

1. wait() blocks the calling thread until the specified condition is signaled. It should be called while mutex is locked
2. signal() is used to signal (or wake up) another thread which is waiting on the condition variable. It should be called after mutex is locked, and programmers must unlock mutex in order for wait() routine to complete
3. broadcast() is useful when multiple threads are in blocked waiting
4. wait() should come before signal() – conditions are not counters, signal would be lost if not waiting



6, condition variable example

- Mutex with condition

- Example: T1: increase x every time;
- T2: when x is larger than 99, then set x to 0;

```
//thread1 :  
while(true)  
{  
    pthread_mutex_lock(&mutex);  
    iCount++;  
    pthread_mutex_unlock(&mutex);  
  
    pthread_mutex_lock(&mutex);  
    if(iCount >= 100)  
    {  
        pthread_cond_signal(&cond);  
    }  
    pthread_mutex_unlock(&mutex);  
}
```

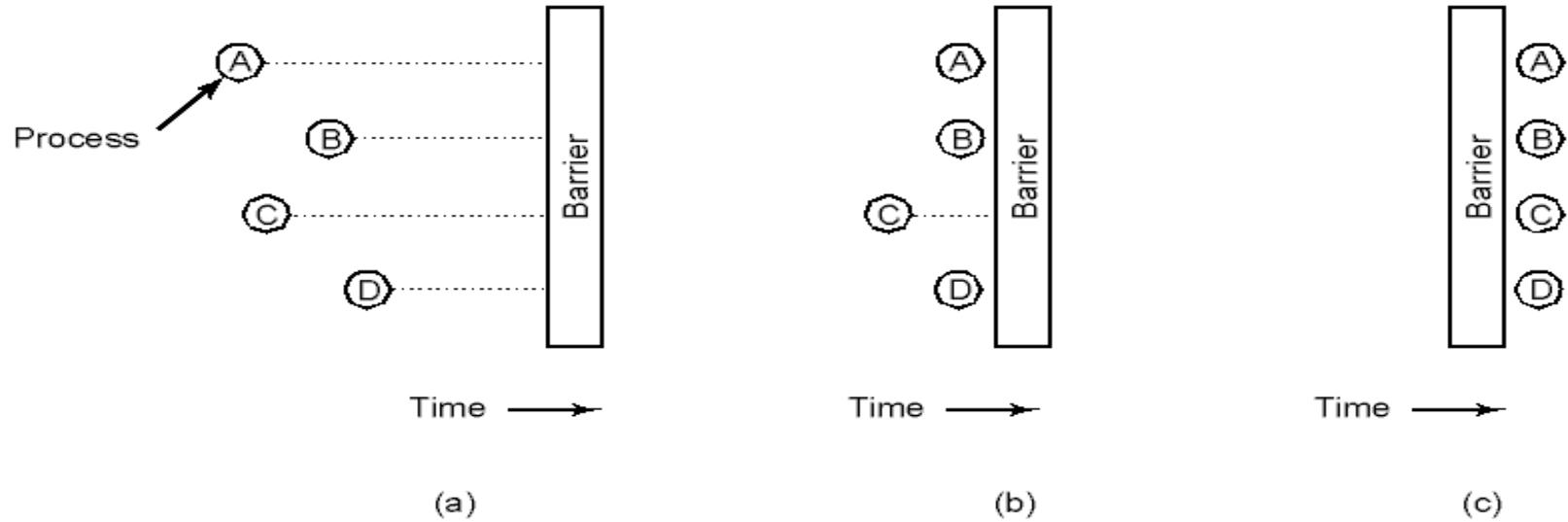


```
//thread2:  
while(1)  
{  
    pthread_mutex_lock(&mutex);  
    while(iCount < 100)  
    {  
        pthread_cond_wait(&cond, &mutex);  
          
    }  
    printf("iCount >= 100\r\n");  
    iCount = 0;  
    pthread_mutex_unlock(&mutex);  
}
```

When T2 executes here:

- 1 : release mutex
- 2 : blocked here
- 3 : when waked, get mutex and execute

7, barriers



- Use of a barrier (for programs operate in *phases*, neither enters the next phase until all are finished with the current phase) for groups of processes to do synchronization
 - (a) processes approaching a barrier
 - (b) all processes but one blocked at barrier
 - (c) last process arrives, all are let through

7, barriers – Initializing and waiting on barriers

[pthread_barrier_init](#) (barrier, attr, count)

[pthread_barrier_wait](#) (barrier)

[pthread_barrier_destroy](#)(barrier)

1. init() function shall allocate any resources required to use the barrier referenced by **barrier** and shall initialize the barrier with **attr**. If attr is NULL, default settings will be applied. The **count** argument specifies the number of threads that must call wait() before any of them successfully return from the call.
2. wait() function shall synchronize participating threads at the barrier.
3. destroy() function shall destroy the barrier and release any resources used by the barrier.



7, barriers example

https://github.com/kevinsuo/CS7172/blob/master/pthread_barrier.c

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

pthread_barrier_t b;

void* task(void* param) {
    int id = (int) param;
    printf("before the barrier %d\n", id);
    pthread_barrier_wait(&b);
    printf("after the barrier %d\n", id);
}

int main() {
    int nThread = 5;
    int i;

    pthread_t thread[nThread];
    pthread_barrier_init(&b, 0, nThread);
    for (i = 0; i < nThread; i++)
        pthread_create(&thread[i], 0, task, (void*)i);
    for (i = 0; i < nThread; i++)
        pthread_join(thread[i], 0);
    pthread_barrier_destroy(&b);
    return 0;
}
```

All threads are waiting here.

Create barrier

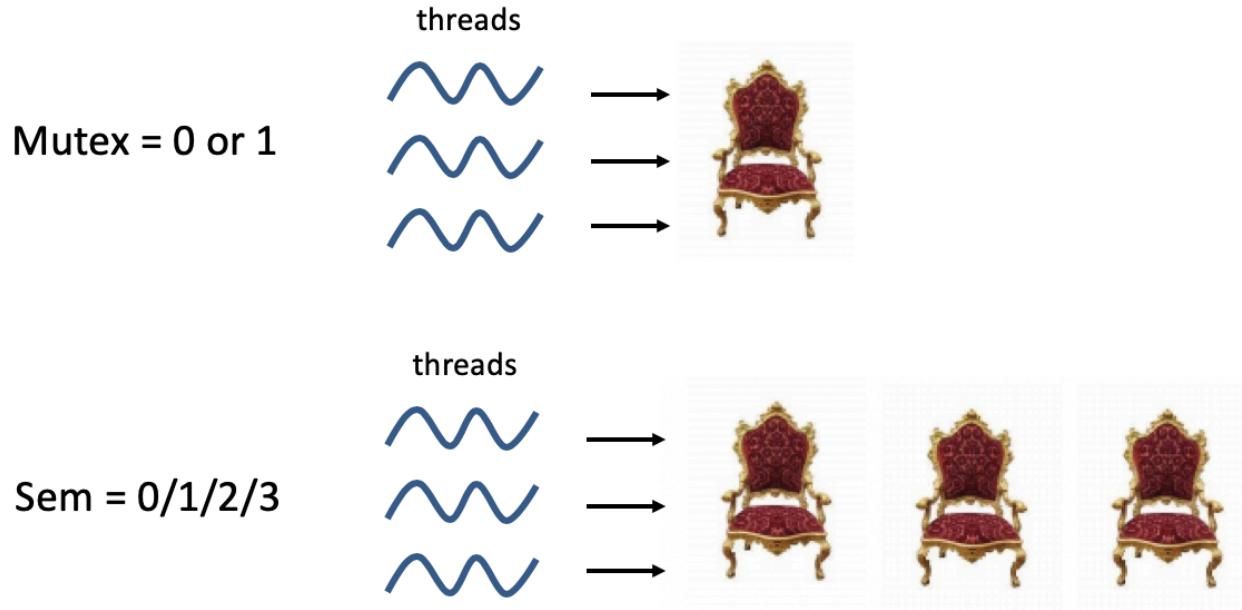
Destroy barrier

```
pi@raspberrypi ~/Downloads> ./test.o
before the barrier 1
before the barrier 0
before the barrier 2
before the barrier 3
before the barrier 4
after the barrier 0
after the barrier 2
after the barrier 3
after the barrier 1
after the barrier 4
```



8, semaphore

- Semaphores are **counters** for resources shared between threads. The basic operations on semaphores are: **increment** the counter atomically, and wait until the counter is non-null and **decrement** it atomically.



Semaphore

- Down operation (P; request):
 - Checks if a semaphore is > 0 , sem--
 - ▶ Request one unit resource and one process enters
- Up operation (V; release)
 - $\text{sem}++$
 - ▶ Release one unit resource and one process leaves



8, semaphore

int sem_init(sem_t *sem, int pshared, unsigned int value)

//Init a sem. The **pshared** argument indicates whether this semaphore is to be shared between the threads of a process, or between processes. The **value** argument specifies the initial value for the semaphore (how many resources).

int sem_destroy(sem_t * sem)

//Destroy a sem

int sem_wait(sem_t * sem)

//Request one unit resource, blocking call

int sem_trywait(sem_t * sem)

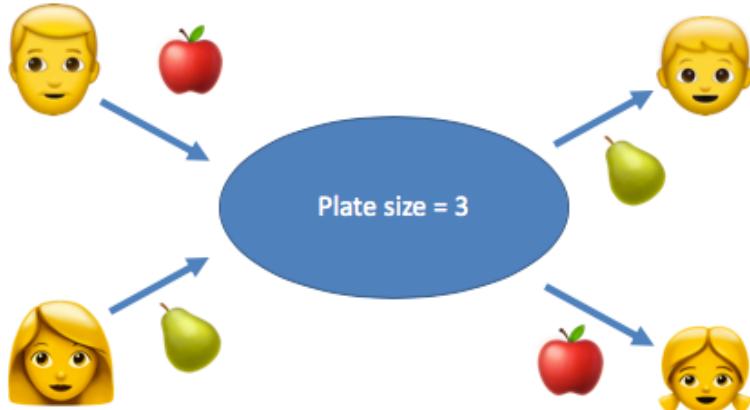
//Request one unit resource, non-blocking call

int sem_post(sem_t * sem)

//Release one unit resource



Semaphore example



- **Semaphore:**

- Son: whether there is pear, s_1
- Daughter: whether there is apple, s_2
- Father/Mother: whether there is space, s_3

Father thread:
peel apple
 $P(s_3)$
put apple
 $V(s_2)$

Mother thread:
peel pear
 $P(s_3)$
put apple
 $V(s_1)$

Son thread:
 $P(s_1)$
get pear
 $V(s_3)$
eat pear

Daughter thread:
 $P(s_2)$
get apple
 $V(s_3)$
eat apple

Semaphore example

- Semaphore:

- Son: whether there is pear, s1
- Daughter: whether there is apple, s2
- Father/Mother: whether there is space, s3

Father thread:

 peel apple
 P(s3)
 put apple
 V(s2)

Daughter thread:

 P(s2)
 get apple
 V(s3)
 eat apple

```
void *father(void *arg) {
    while(1) {
        sleep(5); //simulate peel apple
        P(s3) [sem_wait(&remain)];
        sem_wait(&mutex);
        nremain--;
        napple++;
        sem_post(&mutex);
        V(s2) [sem_post(&apple)];
    }
}
```

```
void *daughter(void *arg) {
    while(1) {
        P(s2) [sem_wait(&apple)];
        sem_wait(&mutex);
        nremain++;
        napple--;
        sem_post(&mutex);
        V(s3) [sem_post(&remain)];
        sleep(10); //simulate eat apple
    }
}
```

[https://github.com/kevinsuo/CS7172/
blob/master/semaphore.c](https://github.com/kevinsuo/CS7172/blob/master/semaphore.c)

Semaphore example

```
pi@raspberrypi ~/Downloads> ./semaphore.o
father 🧑 before put apple, remain=3, apple🍎=0, pear🍐=0
father 🧑 after  put apple, remain=2, apple🍎=1, pear🍐=0

daughter👩 before eat apple, remain=2, apple🍎=1, pear🍐=0
daughter👩 after  eat apple, remain=3, apple🍎=0, pear🍐=0

mother 🧑 before put pear , remain=3, apple🍎=0, pear🍐=0
mother 🧑 after  put pear , remain=2, apple🍎=0, pear🍐=1

son   🧑 before eat pear , remain=2, apple🍎=0, pear🍐=1
son   🧑 after  eat pear , remain=3, apple🍎=0, pear🍐=0

father 🧑 before put apple, remain=3, apple🍎=0, pear🍐=0
father 🧑 after  put apple, remain=2, apple🍎=1, pear🍐=0

mother 🧑 before put pear , remain=2, apple🍎=1, pear🍐=0
mother 🧑 after  put pear , remain=1, apple🍎=1, pear🍐=1

daughter👩 before eat apple, remain=1, apple🍎=1, pear🍐=1
daughter👩 after  eat apple, remain=2, apple🍎=0, pear🍐=1

father 🧑 before put apple, remain=2, apple🍎=0, pear🍐=1
father 🧑 after  put apple, remain=1, apple🍎=1, pear🍐=1

son   🧑 before eat pear , remain=1, apple🍎=1, pear🍐=1
son   🧑 after  eat pear , remain=2, apple🍎=1, pear🍐=0

mother 🧑 before put pear , remain=2, apple🍎=1, pear🍐=0
mother 🧑 after  put pear , remain=1, apple🍎=1, pear🍐=1

father 🧑 before put apple, remain=1, apple🍎=1, pear🍐=1
father 🧑 after  put apple, remain=0, apple🍎=2, pear🍐=1

daughter👩 before eat apple, remain=0, apple🍎=2, pear🍐=1
daughter👩 after  eat apple, remain=1, apple🍎=1, pear🍐=1
```

gcc -pthread semaphore.c
-o semaphore.o

[https://youtu.be/ZIW
wvcuROME](https://youtu.be/ZIWwvcuROME)

Conclusion

- What is pthread?
 - Thread model
 - Pthread overview
- Pthread management
 - Creation and termination
 - Identification
 - Join and detach
- Pthread data sharing
 - Mutex
 - Condition variable
 - Barrier
 - Semaphore

