Chapter 4: Threads

肖 卿 俊

办公室：江宁区无线谷6号楼226办公室

电邮：csqjxiao@seu.edu.cn

主页： https://csqjxiao.github.io/PersonalPage

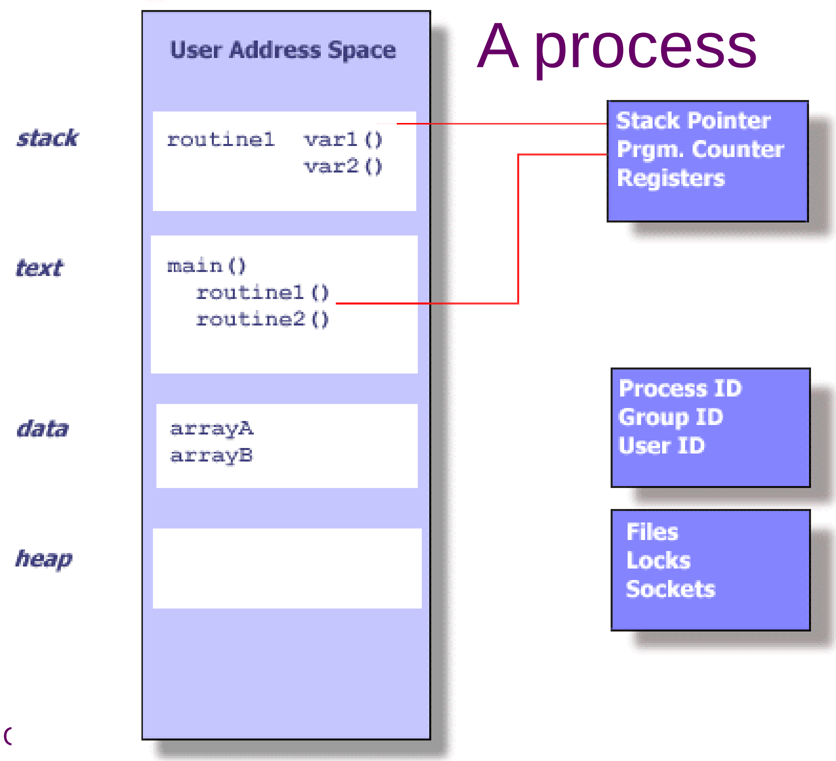
电话：025-52091022

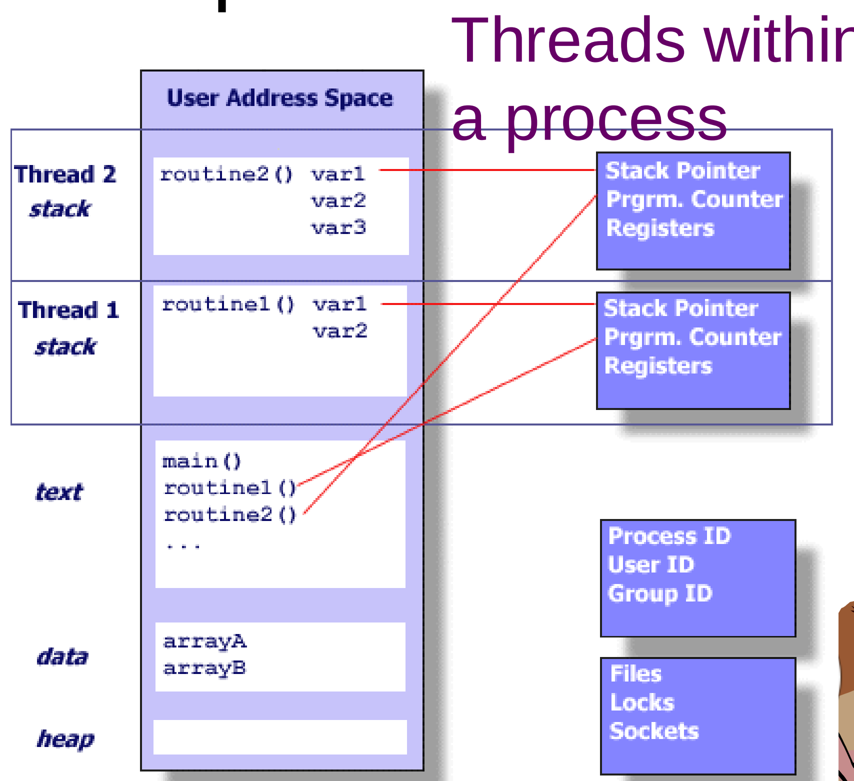
Chapter 4: Threads

- Overview
- Multithreading Models
- Thread Libraries
- Threading Issues
- Operating System Examples

What is a thread?

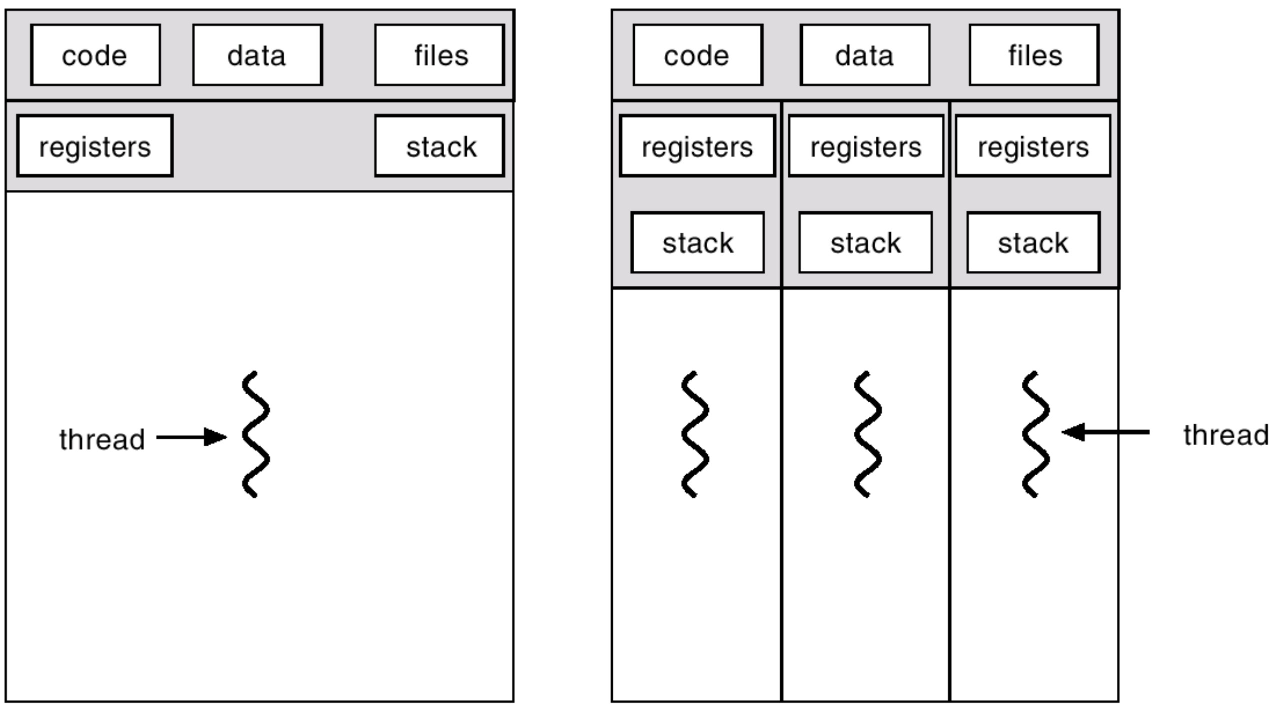
- A *thread*, also known as *lightweight process* (LWP), is a basic unit of CPU execution.
- A thread has a thread ID, a program counter (instruction pointer), a register set, and a stack. Thus, it is similar to a process has.





Single and Multi-threaded Processes

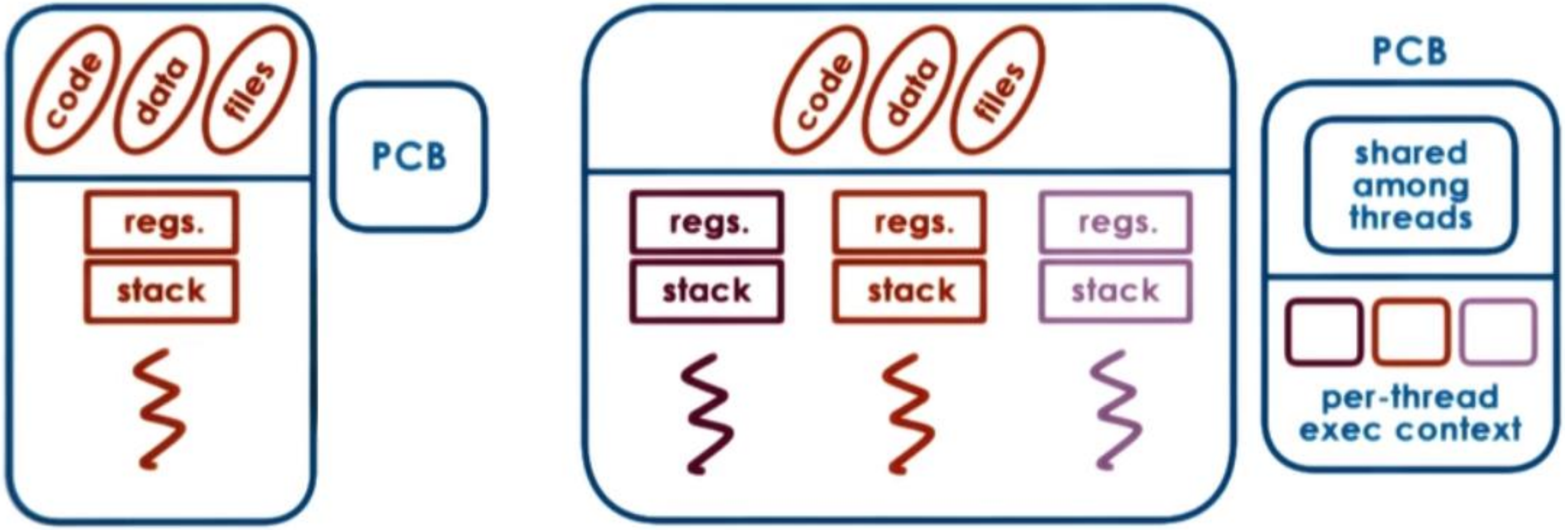
- A *process*, or *heavyweight process*, has a *single* thread of control after its creation.
- As more threads are created, a thread *shares* with other threads in the *same* process its code section, data section, and other OS resources (*e.g.*, files and signals).



single-threaded multithreaded

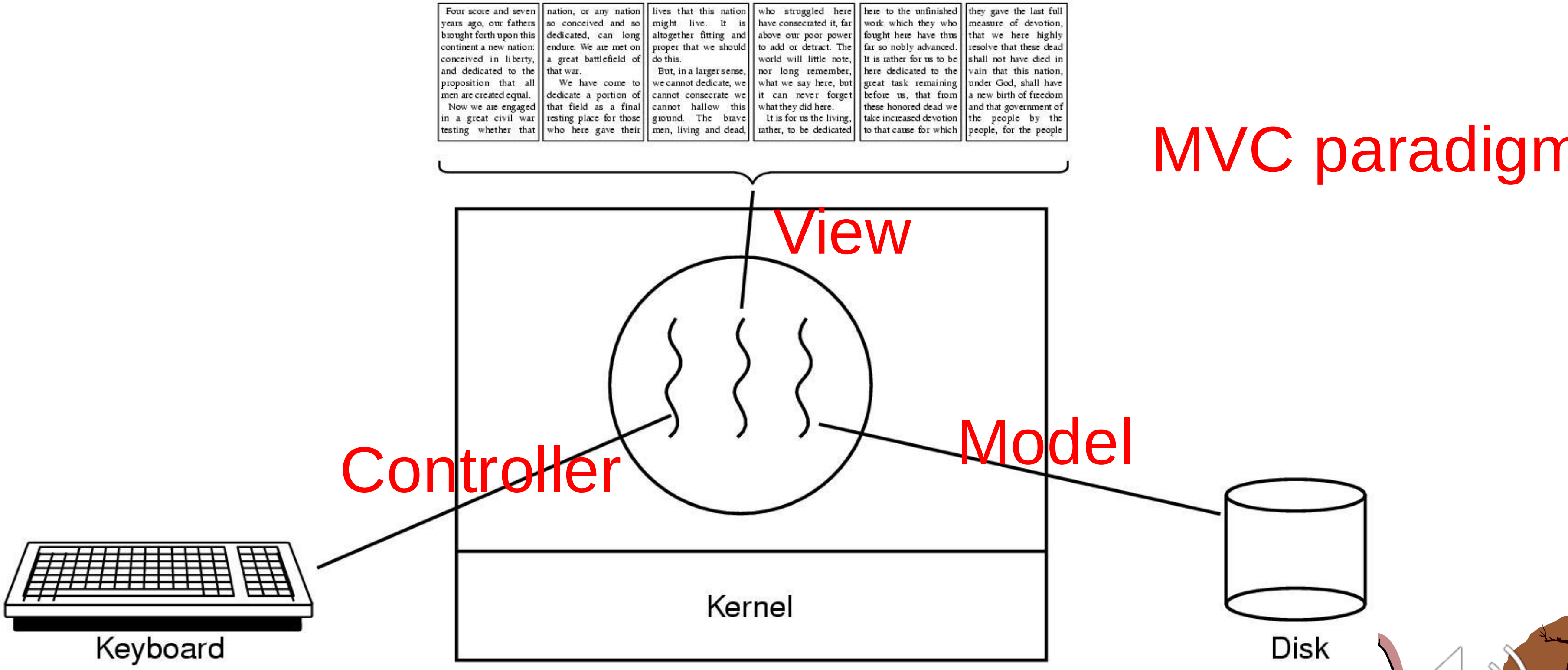
- Items shared by all threads in a process
- Items private to each thread

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	



Why Do We Use Threads? Thread Usage (1)

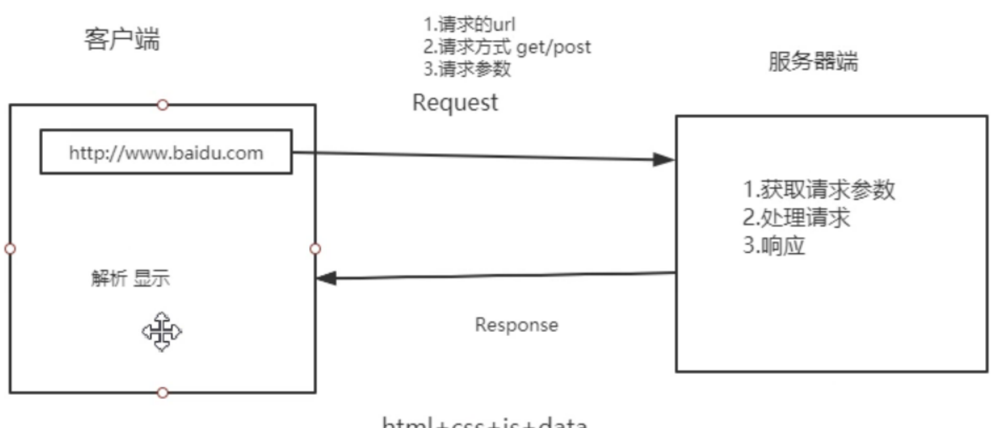
- To simplify programs in which multiple activities go on at once.
- Performance gain, when there is substantial amounts of both computing and I/O.

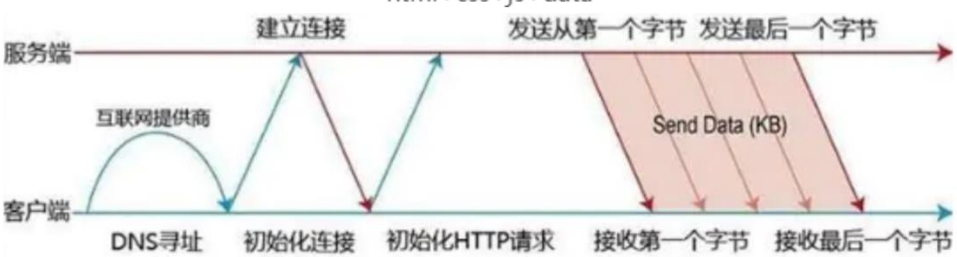


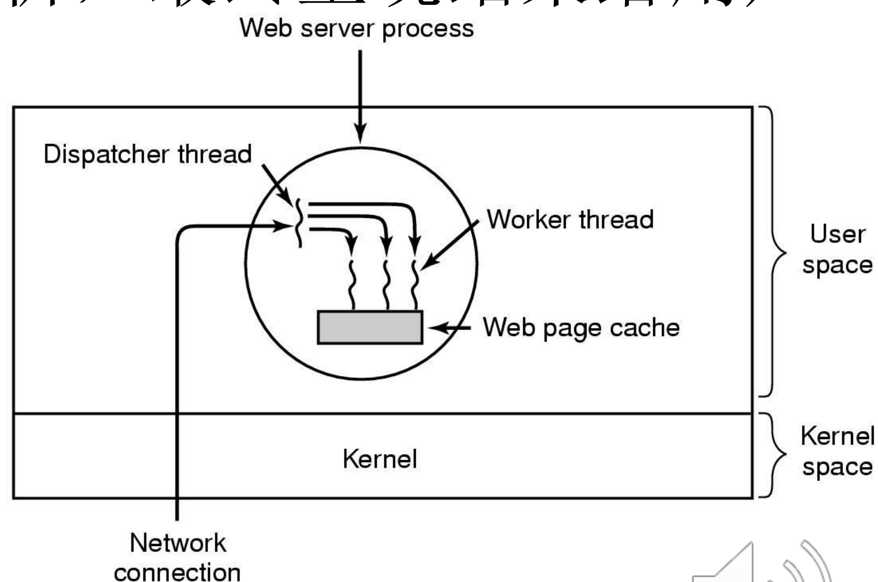
A word processor with three threads

Why Do We Use Threads? Thread Usage (2)

- 基于HTTP协议的浏览器和Web服务器交互过程
  1. 客户端浏览器向网站所在的服务器发送一个请求
  2. 网站服务器接收到这个请求后进行解析
  3. 浏览器中包含网页的源代码等内容（浏览器缓存中），浏览器再对其进行解析，最终呈现结果给用户



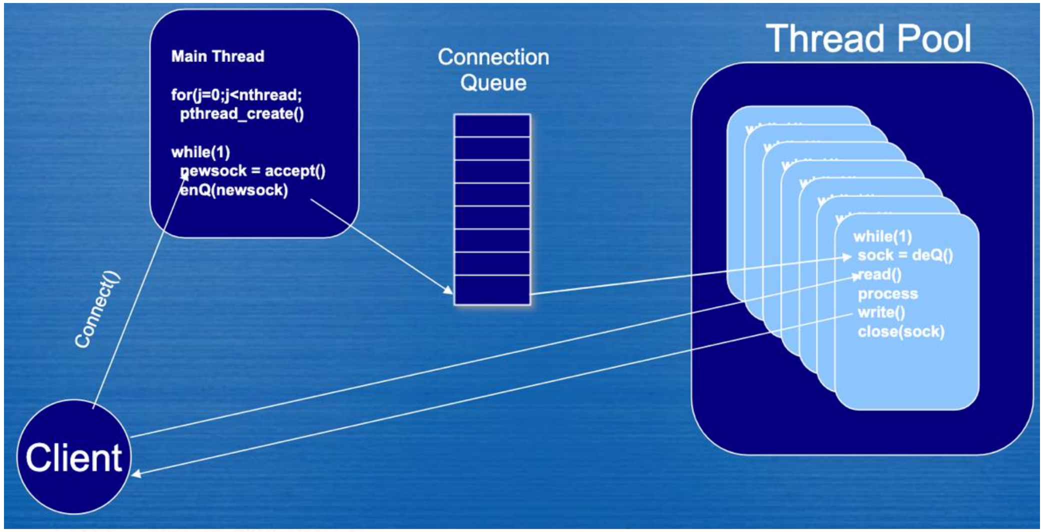




multithreaded web server

Why Do We Use Threads? Thread Usage (3)

- Rough outline of code for previous slide
  - (a) Dispatcher thread
  - (b) Worker thread
  - Note: An Event-Driven Framework




```
while (TRUE) {
  get_next_request(&buf);
  handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
  wait_for_work(&buf)
  look_for_page_in_cache(&buf, &page);
  if (page_not_in_cache(&page)
       read_page_from_disk(&buf, &page);
  return_page(&page);
}
```

(b)

Benefits

- Responsiveness
- Resource Sharing
- Economy
- Utilization of MP Architectures

Economy for Creation

- Compare timing of fork() and pthread_create()
  - Timings reflect 50,000 process/thread creations, were performed with the time utility, and units are in seconds, no optimization flags.

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
AMD 2.4 GHz Opteron (8cpus/node)	41.07	60.08	9.01	0.66	0.19	0.43
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	64.24	30.78	27.68	1.75	0.69	1.10
IBM 1.5 GHz POWER4 (8cpus/node)	104.05	48.64	47.21	2.01	1.00	1.52
INTEL 2.4 GHz Xeon (2 cpus/node)	54.95	1.54	20.78	1.64	0.67	0.90
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.54	1.07	22.22	2.03	1.26	0.67

http://www.cnblogs.com/mywolrd/archive/2009/02/04/1930708.html

Economy for Context Switching

- Process (notes: Process Control Block in OS Kernel)
- Light-weight Process and Kernel Threads
- User Threads

↓ Lower Cost in Creation and Context Switching

User Threads

- Thread management done by user-level threads library
  - Context switching of threads in the same process is done in user mode

- Examples

  - POSIX *Pthreads* (see scope parameter of pthread_create: PTHREAD_SCOPE_PROCESS or PTHREAD_SCOPE_SYSTEM)

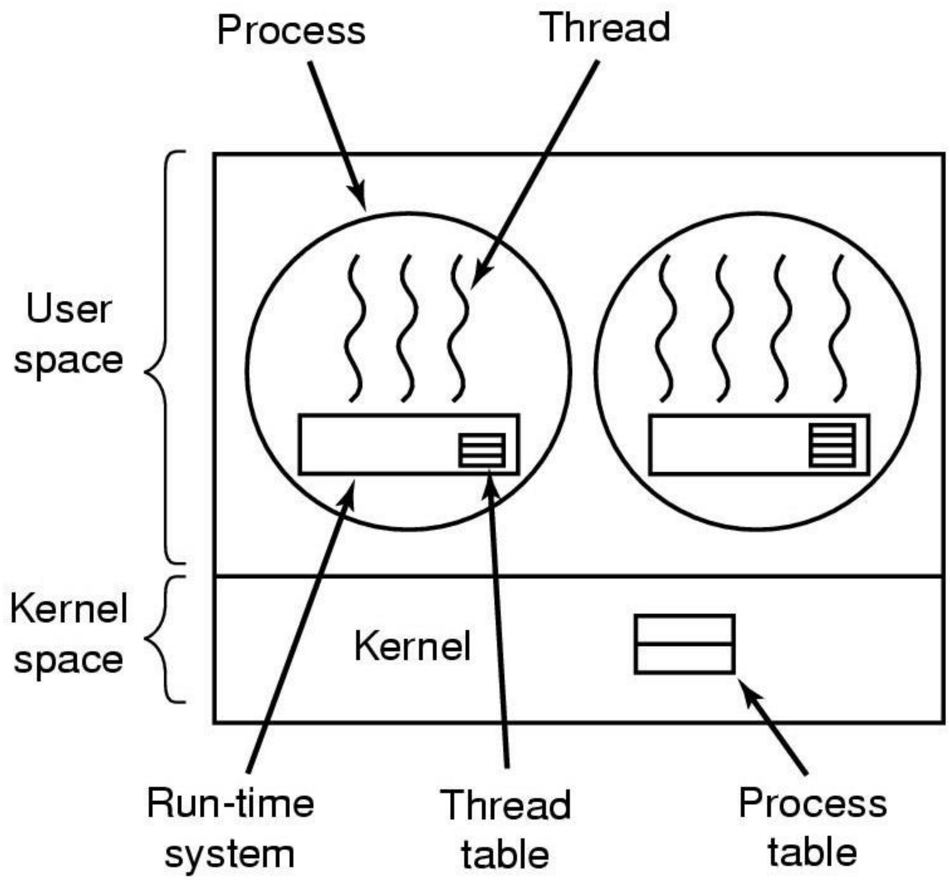
  - Mach *C-threads*

  - Solaris UI-*threads*

http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html#CREATIONTERMINATION

User Threads (Cont.)

- A user-level thread library provides all support for thread creation, termination, joining, and scheduling



A user-level threads package

Pros and Cons of User Threads

- User threads are supported at the user level. The kernel is not aware of user threads.
- Because there is no kernel intervention, user threads are usually more efficient.
- Unfortunately, since the kernel only recognizes the containing process (of the threads), *if one thread is blocked, each other threads of the same process are also blocked* since the containing process is blocked.
- Question: Can two user threads in a same process run simultaneously on two different CPU cores?

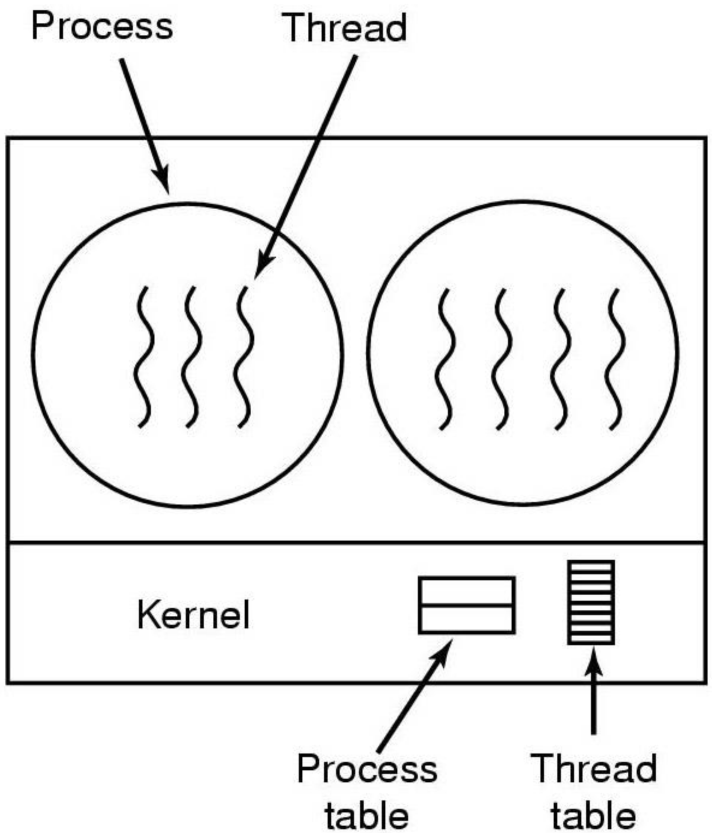
Kernel Threads

- Supported by the Kernel
- Examples
  - Windows 95/98/NT/2000
  - Solaris
  - Tru64 UNIX
  - BeOS
  - Linux and POSIX Thread

Kernel Threads (Cont.)

- Kernel threads are directly supported by the kernel. The kernel does thread creation, termination, joining, and scheduling in kernel space.
- Kernel threads are usually slower than the user threads.
- However, *blocking one thread will not cause other threads of the same process to block*. The kernel simply runs other threads.
- In a multiprocessor environment, the kernel can schedule threads on different processors

Implementing Threads in the Kernel



A threads package managed by the kernel (Note: POSIX *Pthreads* library supports the creation of kernel threads)

http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html

Chapter 4: Threads

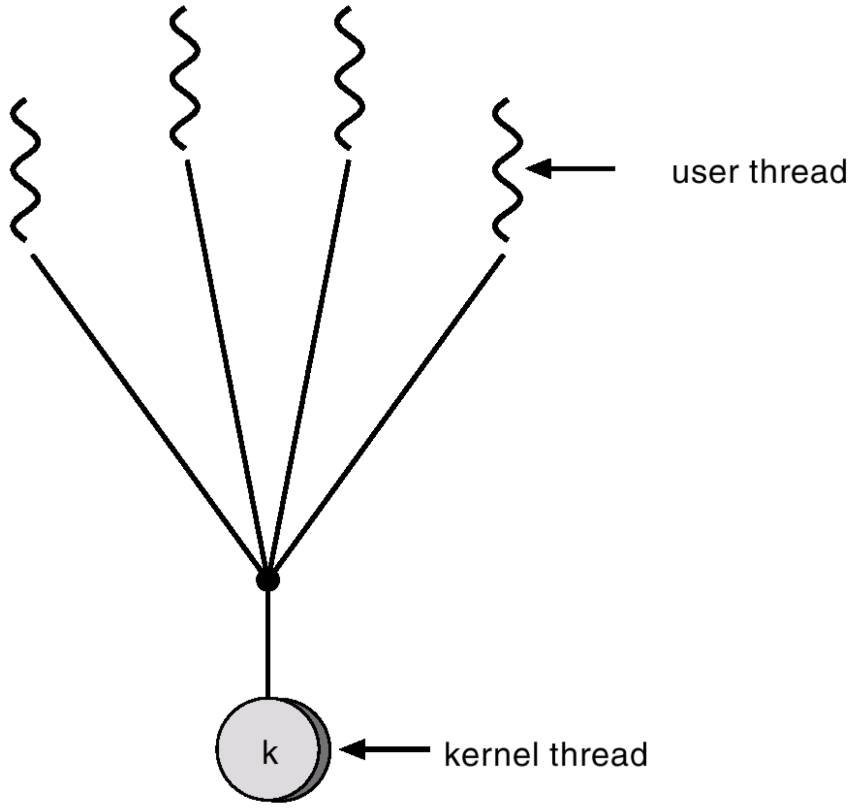
- Overview
- Multithreading Models
- Threading Issues
- Windows XP Threads
- Linux Threads
- Java Threads
- Pthreads
- Windows Threads API

Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

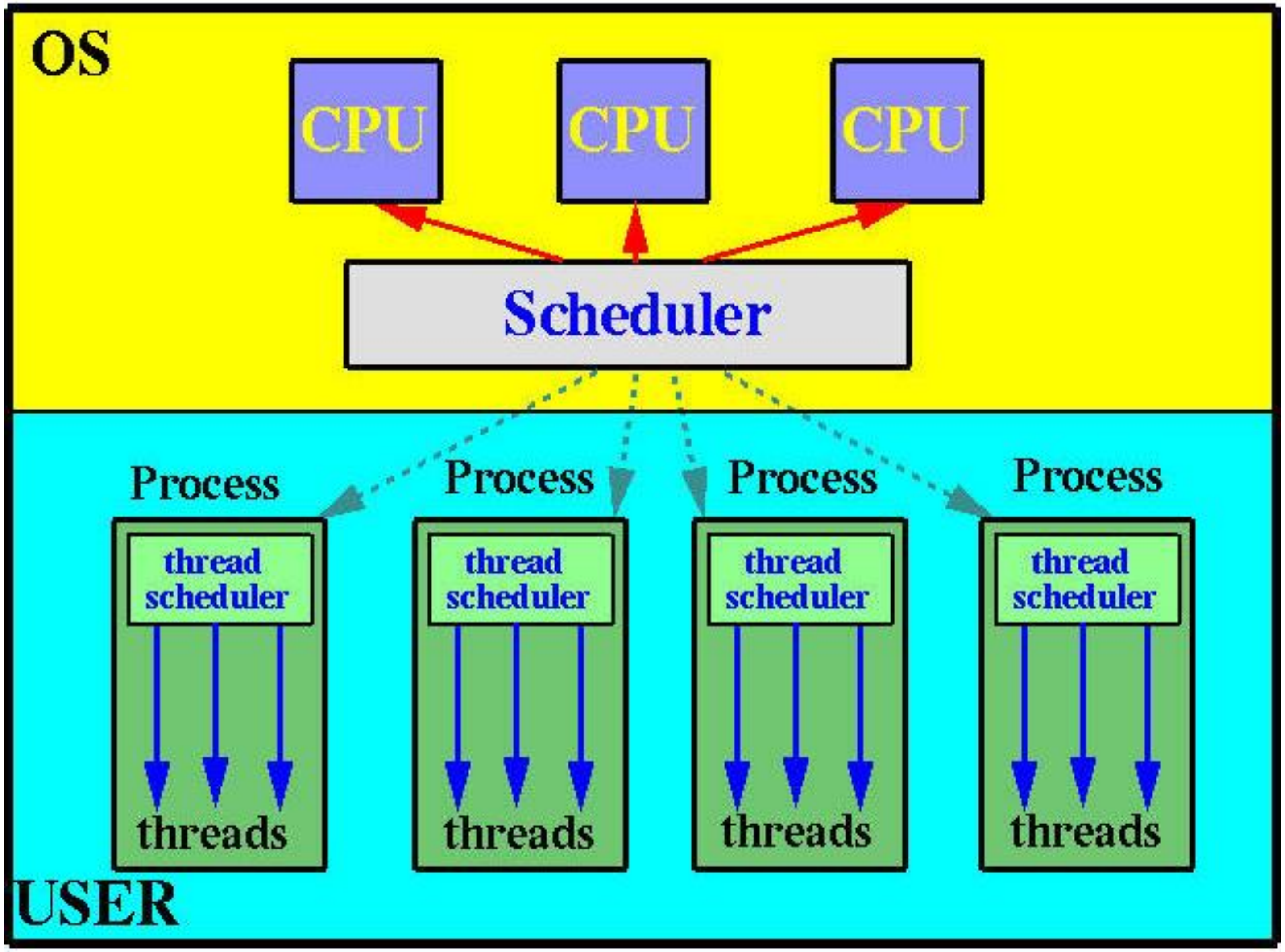
Many-to-One

- Many user-level threads mapped to a single kernel thread.



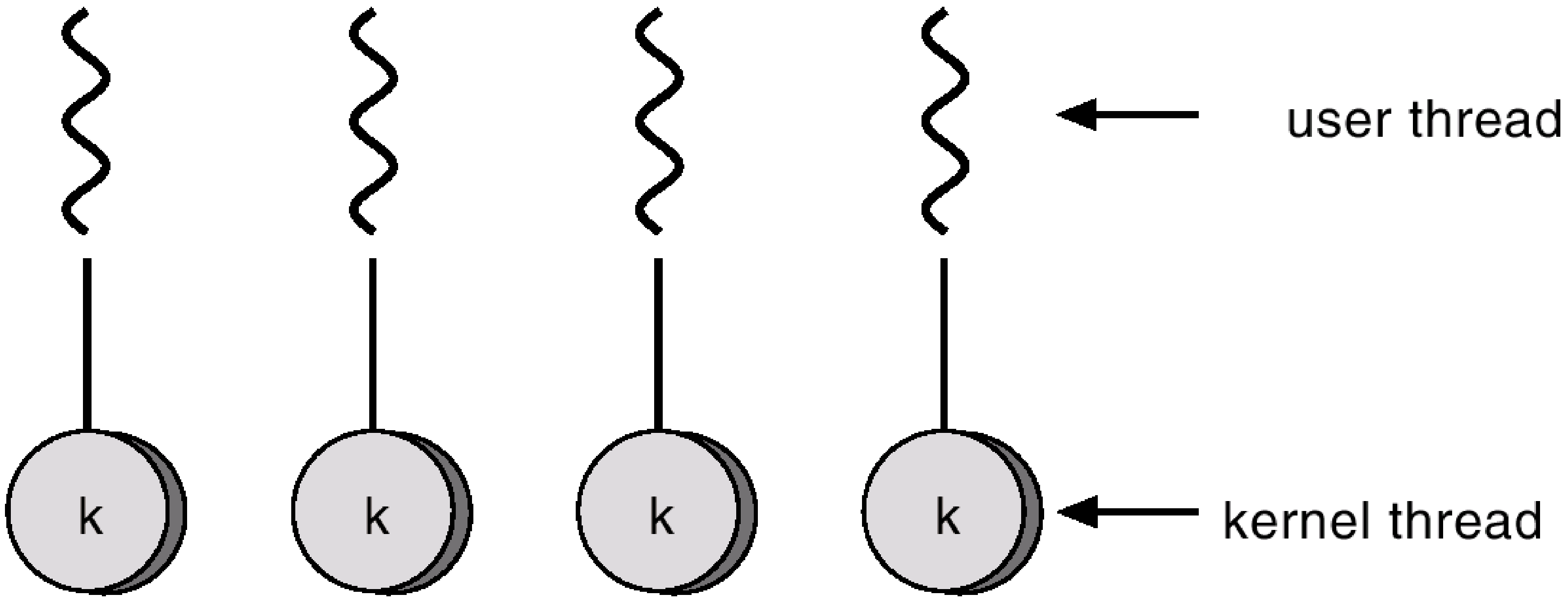
- Used on systems that do not support kernel threads.

Many-to-One Model (Cont.)



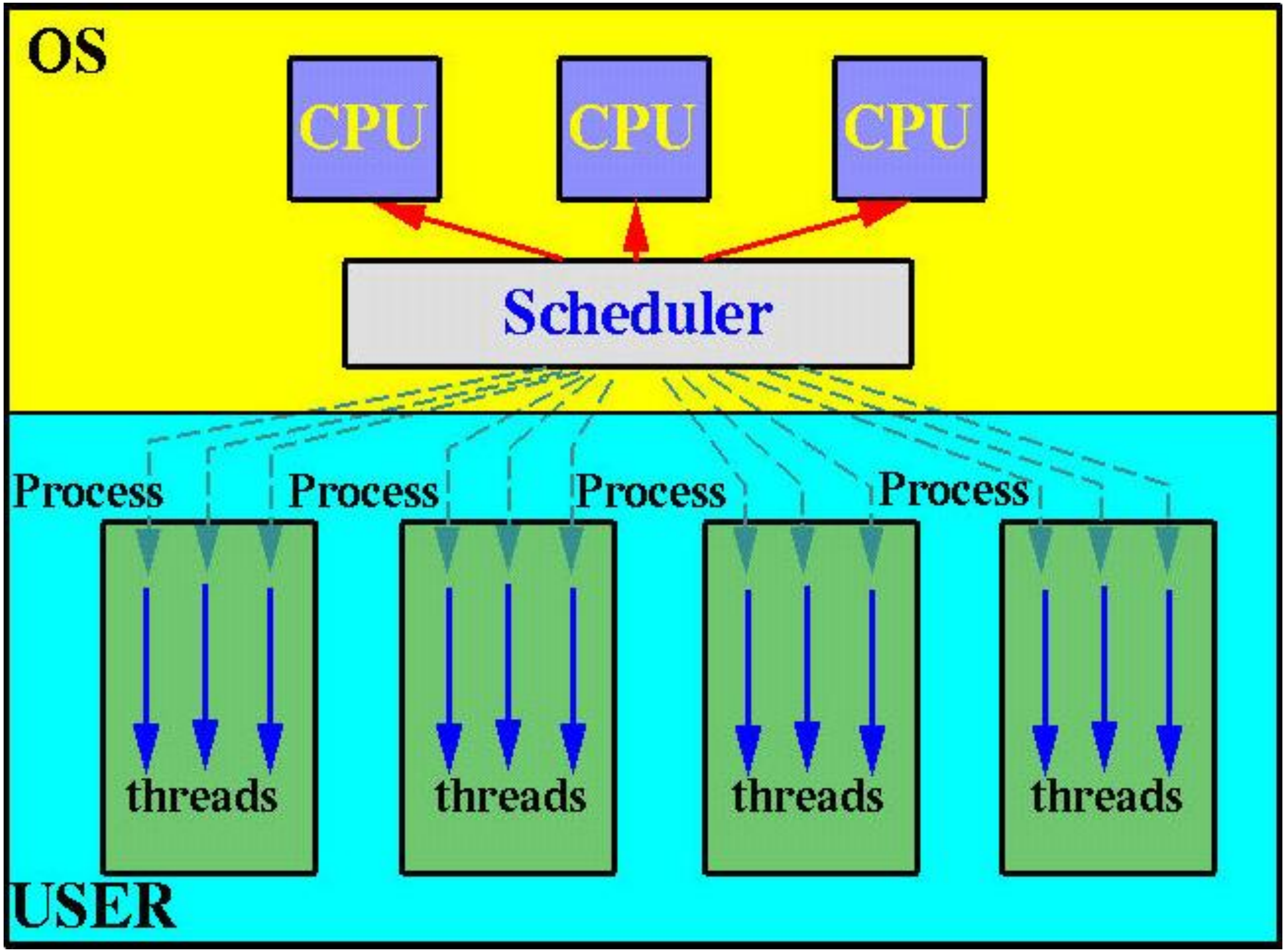
One-to-One

- Each user-level thread maps to kernel thread



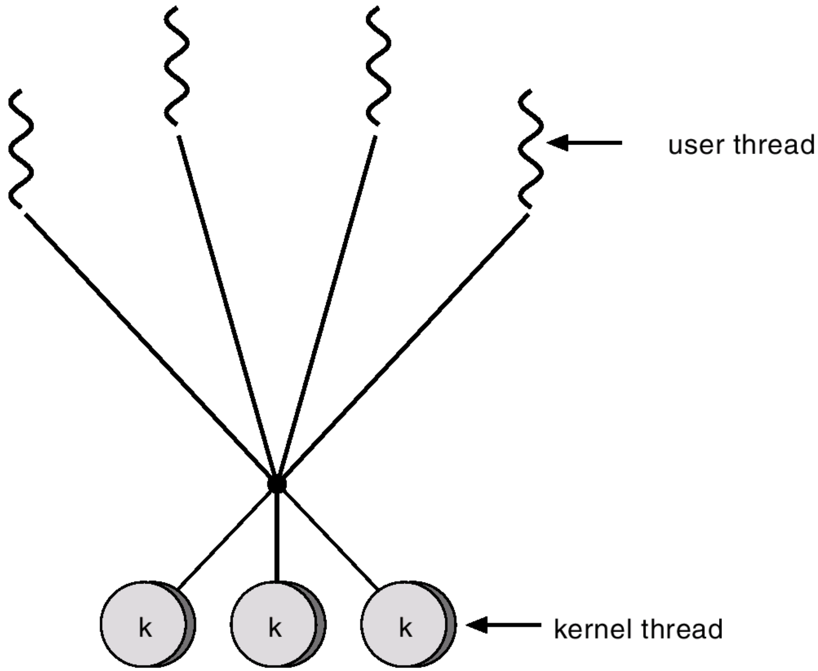
- Examples
  - Windows 95/98/NT/2000
  - OS/2

One-to-one Model (Cont.)



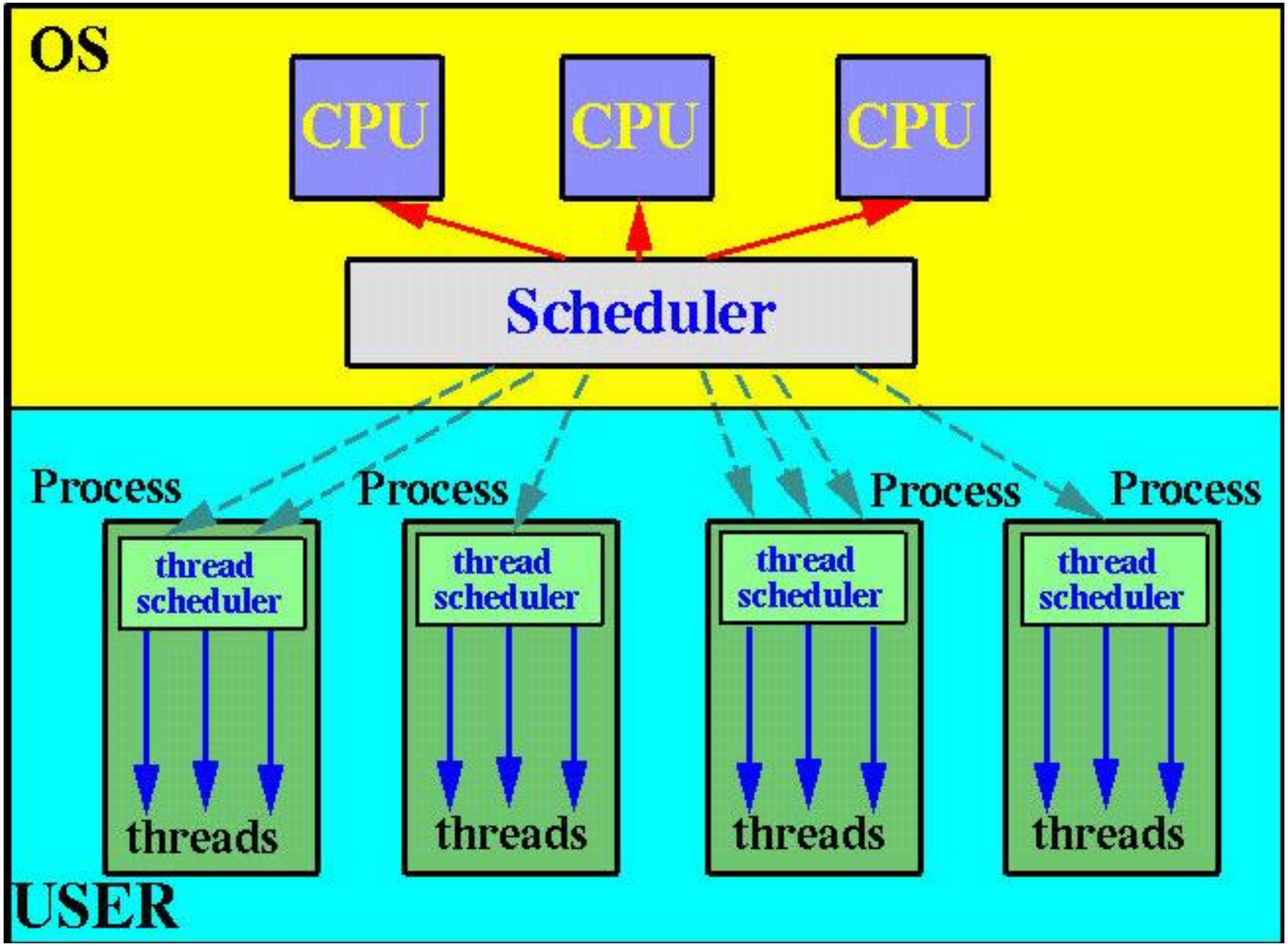
Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads.
- Allows the operating system to create a sufficient number of kernel threads.



- Windows NT/2000 with *ThreadFiber* package
- Solaris 2

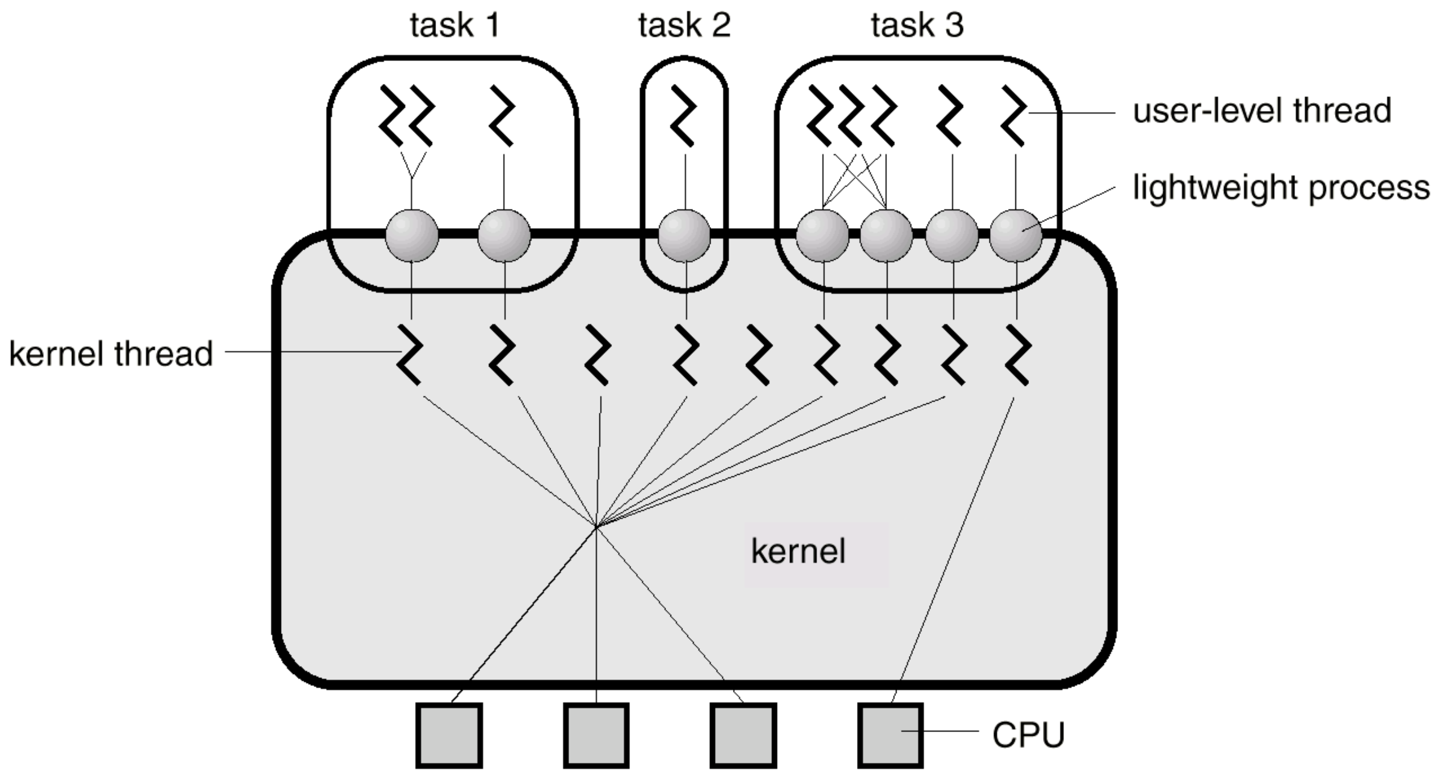
Many-to-Many Model (Cont.)



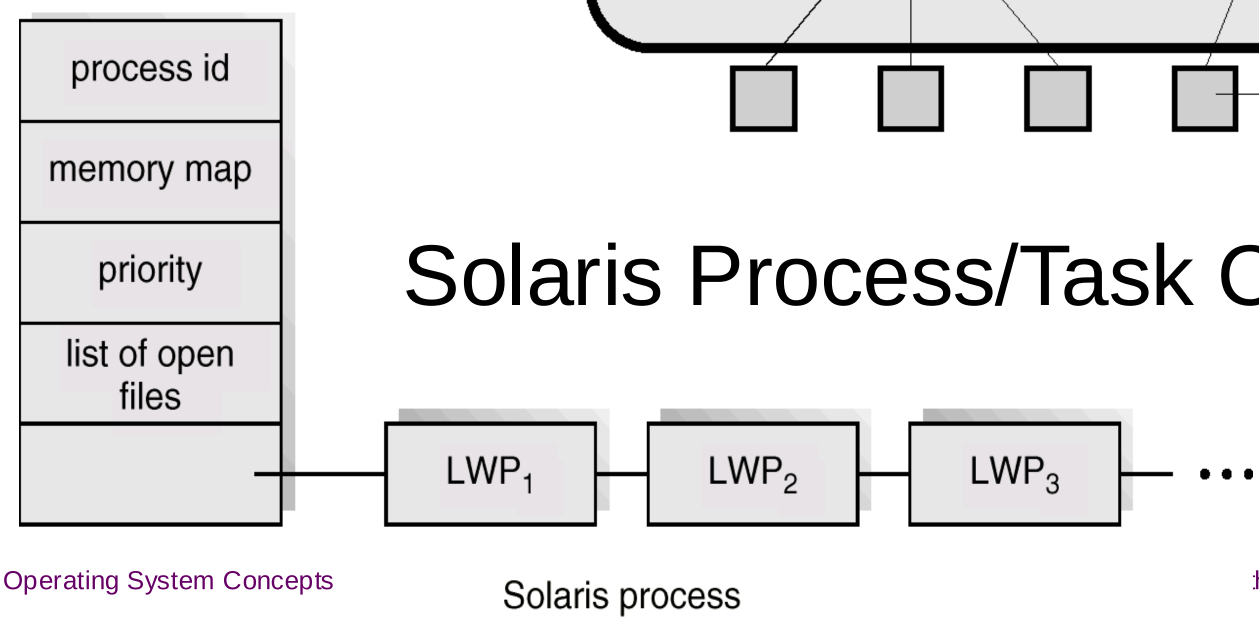
Chapter 4: Threads

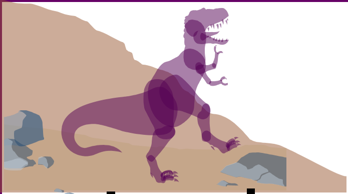
- Overview
- Multithreading Models
- Threading Issues
- Windows XP Threads
- Linux Threads
- Java Threads
- Pthreads
- Windows Threads API

Tasks, LWPs and Threads on Solaris 2



Solaris Process/Task Control Block



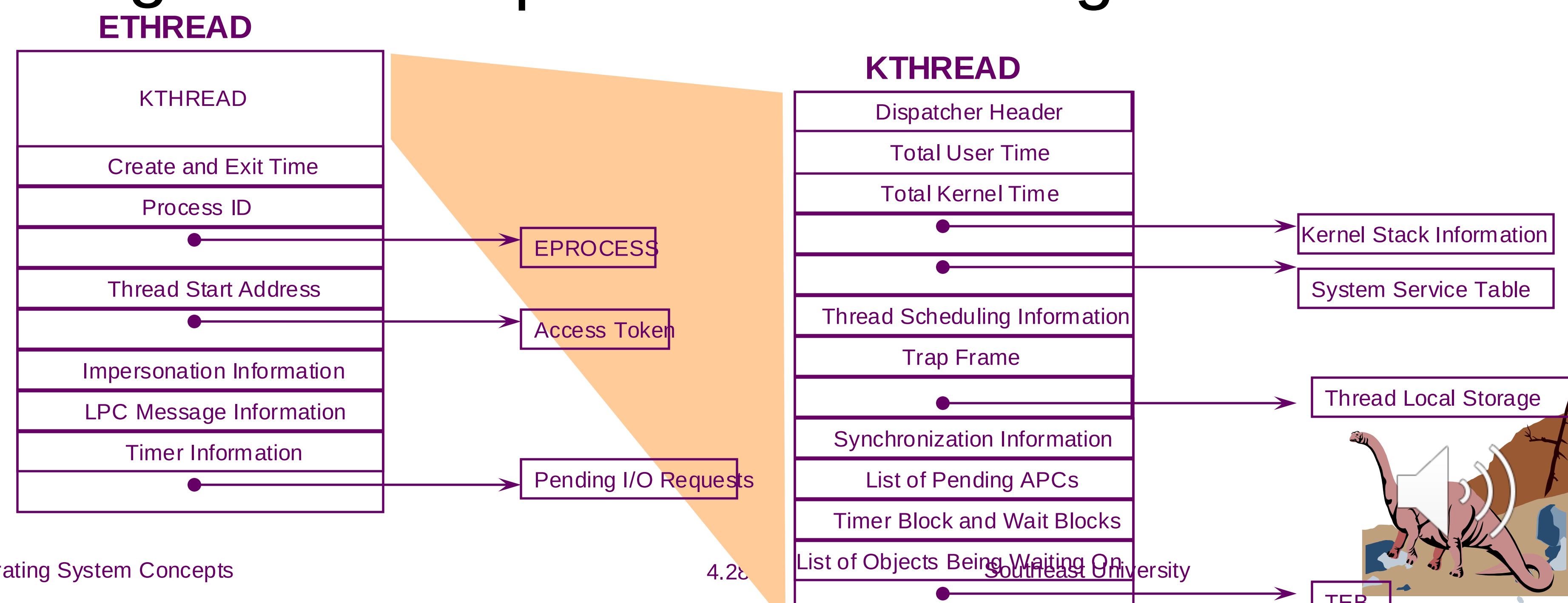


Windows XP Threads

- Implements the one-to-one mapping.
- Each thread has a corresponding **thread control block** in kernel, which contains

  - a thread id - separate user and kernel stacks

  - register set - private data storage area



Linux Threads
(not POSIX pthreads Library)

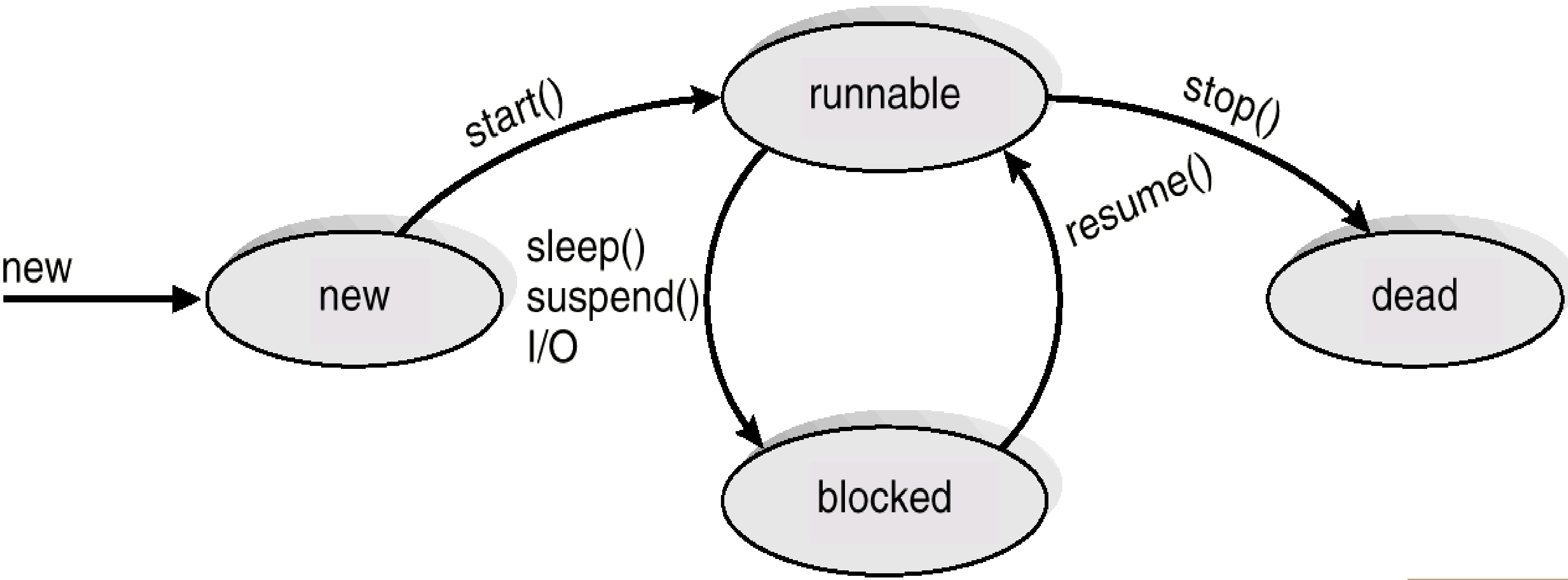
- Linux refers to them as *tasks* rather than *threads*.
- Thread creation is done through clone() system call.
- Clone() allows a child task to share the address space of the parent task (process)
- What is the difference between fork() and clone()?

  http://linux.die.net/man/2/clone

http://www.ibm.com/developerworks/cn/linux/kernel/l-thread/

Java Threads

- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface
- Java threads are managed by the JVM.
- Java Thread States



Chapter 4: Threads

- Overview
- Multithreading Models
- Threading Issues
- Windows XP Threads
- Linux Threads
- Java Threads
- Pthreads
- Windows Threads API

Pthreads

- a POSIX standard (IEEE 1003.1c) API for thread creation and synchronization.
  - API specifies behavior of the thread library,
  - Implementation is up to development of the library

POSIX 1003.1 Commands： http://www.unix.com/man-page-posix-repository.php

- Common in UNIX operating systems.
- Implemented over Linux operating system by Native POSIX Thread Library (NPTL)
  - NPTL is a 1×1 threads library, in that threads created by the user are in 1-1 correspondence with schedulable entities (i.e., task) in the kernel

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

用这个library可以在linux部署M:N的线程模型。
https://github.com/samanbarghi/uThreads

pthread_create

int pthread_create(tid, attr, function, arg);

- pthread_t * tid
  - handle or ID of created thread
- const pthread_attr_t *attr
  - attributes of thread to be created
- void *(*function) (void*)
  - function to be mapped to thread
- void *arg
  - single argument to function
- Integer return value for error code

pthread_create explained

spawn a thread running the function；

thread handle returned via pthread_t structure

- specify *NULL* to use default attributes

a single argument sent to function

- If no argument to function, specify *NULL*

check error codes!

EAGAIN – insufficient resources to create thread

EINVAL – invalid attribute

Threads states

- pthread threads have two states
  - joinable and detached
- threads are joinable by default
  - Resources are kept until *pthread_join.*
  - When a joinable thread terminates, some of the thread resources are kept allocated, and released only when another thread performs *pthread_join* on that thread.
  - can be reset with attribute or API call
- detached thread can not be joined
  - resources can be reclaimed at termination
  - cannot reset to be *joinable*

Waiting for a thread

int pthread_join(tid, val_ptr);

- pthread_t *tid
  - handle of joinable thread
- void **val_ptr
  - exit value returned by joined thread

pthread_join explained

calling thread waits for the thread with handle tid to terminate

- only one thread can be joined
- thread must be joinable

exit value is returned from joined thread

- Type returned is (void *)
- use NULL if no return value expected

ESRCH –thread not found

EINVAL – thread not joinable

Example 1

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

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: end\n");
    return 0;
}
```

Q1: Guess what are the possible outputs?

Q2: What if we remove the two Pthread_join() function calls?
Note: the termination of main thread will cause the automatic termination of children threads

A Quiz about fork() and pthread_create()

- What are the outputs of the program?

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

int value = 0;

void *runner(void *param){
  value = value + 10;
  pthread_exit(0);
}

int main(int argc, char *argv[])
{
    int  pid;
    pthread_t  tid;
    pid = fork();
    if (pid == 0) {
        pthread_create (&tid, NULL, runner, NULL);
        pthread_join (tid, NULL);
        printf ("CHILD: value = %d\n", value);  /*Print 1*/
    } else if (pid > 0) {
        value = value - 10;
        wait(NULL);
        printf("PARENT: value = %d\n", value);  /*Print 2*/
    }
}
```

- Answer:

CHILD: value = 10
PARENT: value = -10

Example 2

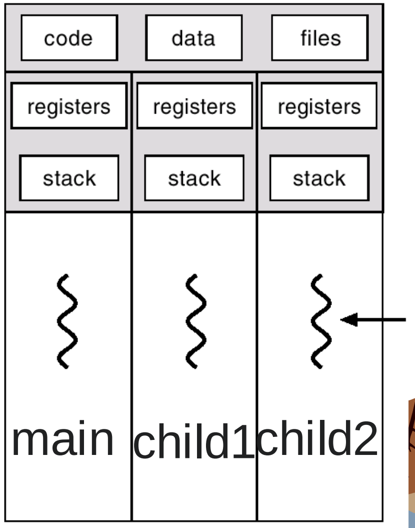
volatile int counter = 0; // shared global variable

```
void *
mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}
```

// The volatile keyword forces the compiler to always reads the current value of a volatile object from the memory location rather than keeping its value in temporary register at the point it is requested

```
int
main(int argc, char *argv[])
{
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

Q1: Guess what is the possible output



multithreaded

Discussion

- Why not deterministic?
- The Heart Of The Problem: Uncontrolled Scheduling
- What happens when executing “counter = counter + 1;” ?
- Understand the code sequence that the compiler generates for the update to counter.

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

- Now, you may tell the reason

用gcc -S命令简单验证一下

- GCC的选项-S使GCC在执行完汇编后停止

```
$ gcc -S t1.c -o t1.s // 汇编代码
$ gcc -c t1.s -o t1.o // 二进制代码
$ ld t1.o -o t1 // 链接后可执行代码
```

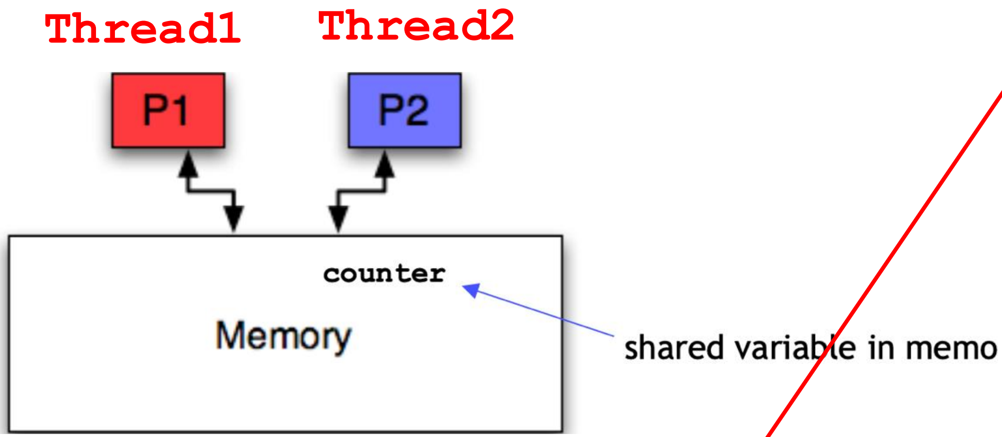
- 看t1.s汇编代码

```
void *
mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}
```

```
int
main(int argc, char *argv[])
{
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

```
_mythread:                          ## @mythread
        .cfi_startproc
        …
        movl    _counter(%rip), %eax
        addl    $1, %eax
        movl    %eax, _counter(%rip)
        …
        .cfi_endproc
                                    ## -- End function
_main:                              ## @main
        .cfi_startproc
        …
        .cfi_endproc
        .section
__TEXT,__cstring,cstring_literals
        …
        .globl  _counter    ## @counter
```

Uncontrolled Scheduling




```
movl    _counter(%rip), %eax
```

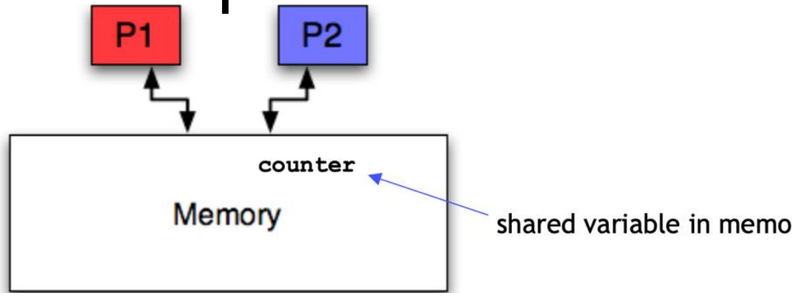
%rip-relative addressing for global variables

x86-64 code often refers to globals using %rip-relative addressing: a global variable named a is referenced as a(%rip). This style of reference supports *position-independent code (PIC)*, a security feature. It specifically supports *position-independent executables (PIEs)*, which are programs that work independently of where their code is loaded into memory.

			(after instruction)		
OS	Thread 1	Thread 2	PC	%eax	counter
	before critical section		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	51

Uncontrolled Scheduling

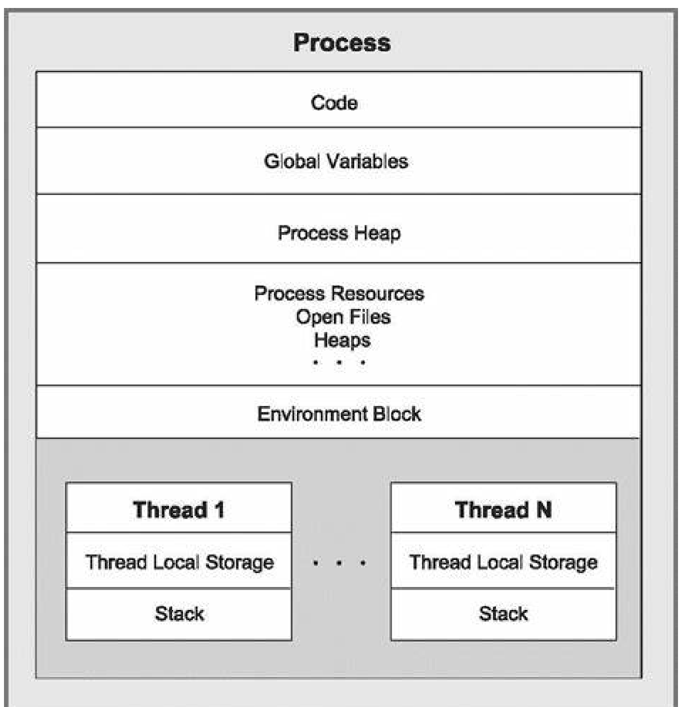
- Race condition
  - Several processes (threads) access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place.
  - Result indeterminate.

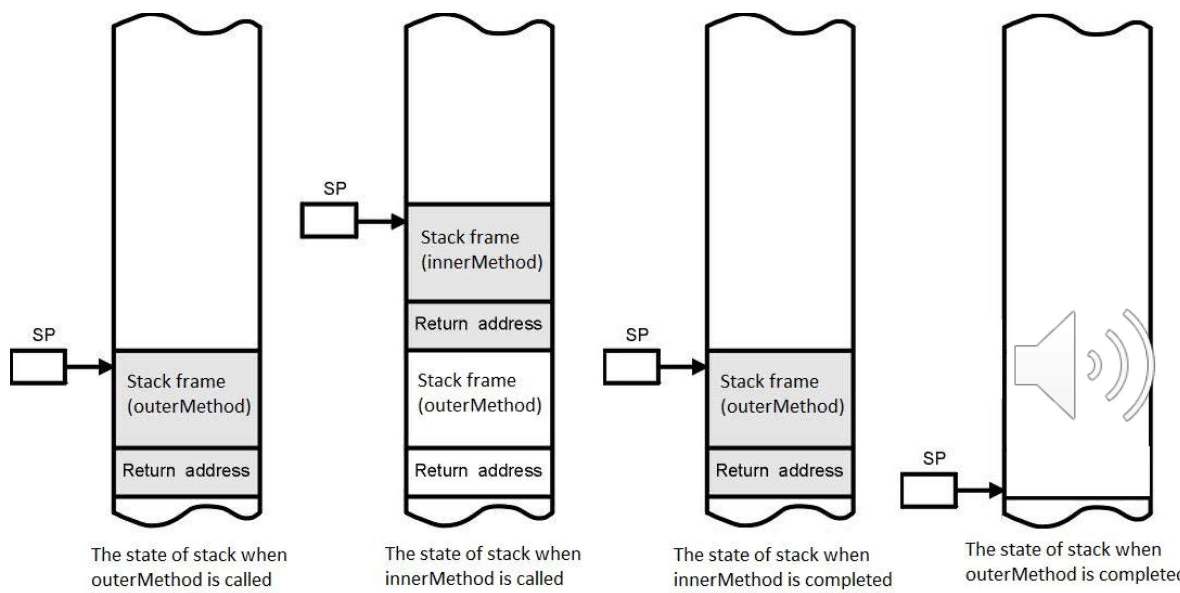


- Critical section
  - Multiple threads executing a segment of code, which can result in a race condition.
- What we want: Mutual exclusion
  - The property guarantees that if one thread is executing within the critical section, the others will be prevented from doing so.

Revisit the Threading Model

- "Data" is a public memory segment shared by all threads, which may incur race condition
- Stack is a private memory segment of a thread
- Question: What if a thread accesses the data variables on the stack of another thread?





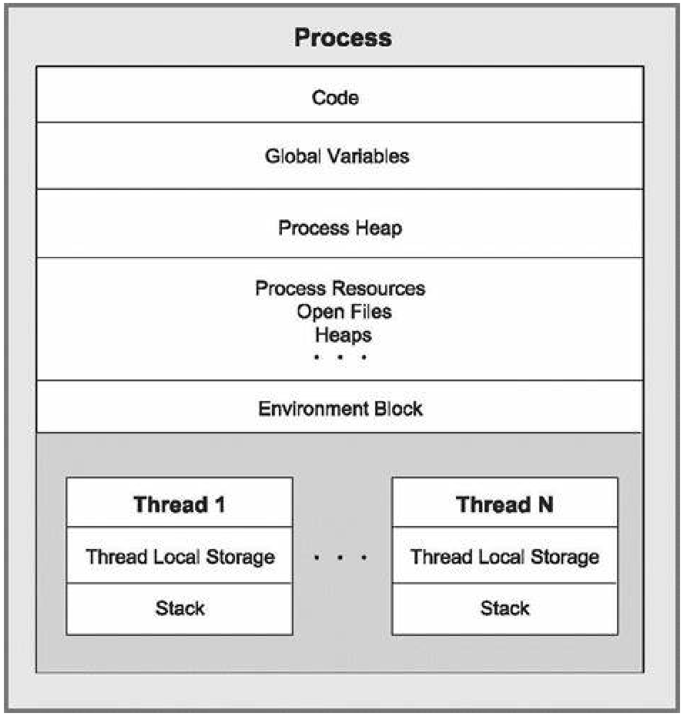
What are possible outputs of the program

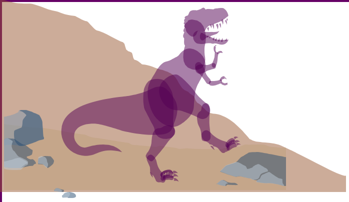
```
void * helloFunc ( void * ptr ) {
    int *data;
    data = (int *) ptr;
    printf("I'm Thread %d \n", *data);
}

int main() {
  pthread_t hThread[4];
  for (int i = 0; i < 4; i++)
      pthread_create(&hThread[i], NULL, helloFunc, (void *)&i);
  for (int i = 0; i < 4; i++)
      pthread_join(hThread[i], NULL);
  return 0;
}
```

注意： race condition， 多个子线程T0、T1、T2、T3同时访问主线程栈上的局部变量 i， 导致读写冲突。

Fix the problem by threat-local states

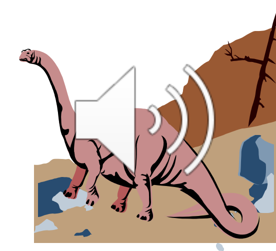
```
void * helloFunc ( void * ptr ) {
    int *data;
    data = (int *) ptr;
    printf("I'm Thread %d \n", *data);
}

int main() {
    pthread_t hThread[4];
    int thread_name[4];
    for (int i = 0; i < 4; i++) {
        thread_name[i] = i;
        pthread_create(&hThread[i], NULL,
helloFunc, (void *)&thread_name[i]);
    }
    for (int i = 0; i < 4; i++)
         pthread_join(hThread[i], NULL);
    return 0;
}
```





Chapter 4: Threads

- Overview
- Multithreading Models
- Threading Issues
- Windows XP Threads
- Linux Threads
- Java Threads
- Pthreads
- Windows Thread APIs



Windows Thread APIs

- CreateThread
- ExitThread
- TerminateThread
- GetExitCodeThread

- GetCurrentThreadId - returns global ID
- GetCurrentThread - returns handle
- SuspendThread/ResumeThread
- GetThreadTimes

Windows API Thread Creation

```
HANDLE CreateThread (
    LPSECURITY_ATTRIBUTES lpsa,
    DWORD cbStack,
    LPTHREAD_START_ROUTINE lpStartAddr,
    LPVOID lpvThreadParm,
    DWORD fdwCreate,
    LPDWORD lpIDThread)
```

cbStack == 0: thread‘s stack size defaults to primary thread‘s size

- lpstartAddr points to function declared as

  DWORD WINAPI ThreadFunc(LPVOID)

- lpvThreadParm is 32-bit argument
- LPIDThread points to DWORD that receives thread ID non-NULL pointer !

Windows API Thread Termination

```
VOID ExitThread( DWORD devExitCode )
```

- When the last thread in a process terminates, the process itself terminates

```
BOOL GetExitCodeThread (
HANDLE hThread, LPDWORD lpdwExitCode)
```

- Returns exit code or STILL_ACTIVE

Suspending and Resuming Threads

- Each thread has suspend count
- Can only execute if suspend count == 0
- Thread can be created in suspended state

```
DWORD ResumeThread (HANDLE hThread)
DWORD SuspendThread(HANDLE hThread)
```

- Both functions return suspend count or 0xFFFFFFFF on failure

Example: Thread Creation

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

DWORD WINAPI helloFunc(LPVOID arg ) {
        printf("Hello Thread\n");
        return 0;
}

main() {
        HANDLE hThread =
                CreateThread(NULL, 0, helloFunc,
                        NULL, 0, NULL );
        }
```

What's Wrong?

Example Explained

- Main thread is process
- When process goes, all threads go
- Need some methods of waiting for a thread to finish

Waiting for Windows* Thread

```
#include <stdio.h>
#include <windows.h>
BOOL thrdDone = FALSE;
DWORD WINAPI helloFunc(LPVOID arg ) {
        printf("Hello Thread\n");
        return 0;
}

main() {
        HANDLE hThrea
                Create
                        NULL, 0, NULL );
        }
```

thrdDone = TRUE;

Not a good idea!

while (!thrdDone);

Waiting for a Thread

Wait for one object (thread)

```
DWORD WaitForSingleObject(
      HANDLE hHandle,
      DWORD dwMilliseconds );
```

Calling thread waits (blocks) until

- Time expires
  - Return code used to indicate this
- Thread exits (handle is signaled)
  - Use **INFINITE** to wait until thread termination

Does not use CPU cycles

Waiting for Many Threads

Wait for up to 64 objects (threads)

```
DWORD WaitForMultipleObjects(
      DWORD nCount,
      CONST HANDLE *lpHandles, // array
      BOOL fWaitAll, // wait for one or all
      DWORD dwMilliseconds)
```

Wait for all: `fWaitAll==TRUE`

Wait for any: `fWaitAll==FALSE`

- Return value is first array index found

Notes on WaitFor* Functions

- Handle as parameter
- Used for different types of objects
- Kernel objects have two states
  - Signaled
  - Non-signaled
- Behavior is defined by object referred to by handle
  - Thread: signaled means terminated

Example: Waiting for multiple threads

```
#include <stdio.h>
#include <windows.h>
const int numThreads = 4;

DWORD WINAPI helloFunc(LPVOID arg ) {
  printf("Hello Thread\n");
  return 0; }

main() {
  HANDLE hThread[numThreads];
  for (int i = 0; i < numThreads; i++)
    hThread[i] =
      CreateThread(NULL, 0, helloFunc, NULL, 0, NULL );
  WaitForMultipleObjects(numThreads, hThread,
                                            TRUE, INFINITE);

}
```

Example: HelloThreads

- Modify the previous example code to print out
  - appropriate “Hello Thread”message
  - Unique thread number
    - ✓use for-loop variable of CreateThread loop
- Sample output:

```
Hello from Thread #0
Hello from Thread #1
Hello from Thread #2
Hello from Thread #3
```

What’s Wrong?

```
DWORD WINAPI threadFunc(LPVOID pArg) {
  int* p = (int*)pArg;
  int myNum = *p;
  printf( “Thread number %d\n”, myNum);
}
. . .
// from main():
for (int i = 0; i < numThreads; i++) {
  hThread[i] =
     CreateThread(NULL, 0, threadFunc, &i, 0, NULL);
}
```

What is printed for myNum?

Hello Threads Timeline

Time	*main*	*Thread 0*	*Thread 1*
T_0	i = 0	---	----
T_1	create(&i)	---	---
T_2	i++ (i == 1)	launch	---
T_3	create(&i)	p = pArg	---
T_4	i++ (i == 2)	myNum = *p myNum = 2	launch
T_5	wait	print(2)	p = pArg
T_6	wait	exit	myNum = *p myNum = 2

Race Conditions

- Concurrent access of same variable by multiple threads
  - Read/Write conflict
  - Write/Write conflict
- Most common error in concurrent programs
- May not be apparent at all times
- How to avoid data races?
  - Local storage
  - Control shared access with critical regions

Hello Thread: Local Storage solution

```
DWORD WINAPI threadFunc(LPVOID pArg)
{
  int myNum = *((int*)pArg);
  printf( "Thread number %d\n", myNum);
}
. . .

// from main():
for (int i = 0; i < numThreads; i++) {
  tNum[i] = i;
  hThread[i] =
      CreateThread(NULL, 0, threadFunc, &tNum[i],
                          0, NULL);
}
```

Chapter 4: Threads

- Overview
- Multithreading Models
- Threading Issues
- Windows XP Threads
- Linux Threads
- Java Threads
- Pthreads
- Windows Threads API

Threading Issues

1. Semantics of fork() and exec() system calls.
2. Thread cancellation.
3. Signal handling
4. Thread pools
5. Thread specific data
6. Scheduler Activations

Semantics of fork() and exec()

- Does **fork()** duplicate only the calling thread or all threads?
- In a Pthreads-compliant implementation, the fork() call always creates a new child process with a single thread, regardless of how many threads its parent may have had at the time of the call.
- Furthermore, the child's thread is a replica of the thread in the parent that called fork

Thread Cancellation

- Terminating a thread before it has finished

- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
    - ✓The point a thread can terminate itself is a ***cancellation point***.

Thread Cancellation (Cont.)

- With asynchronous cancellation, if the target thread owns some system-wide resources, the system may not be able to reclaim all recourses

- With deferred cancellation, the target thread determines the time to terminate itself. Reclaiming resources is not a problem.

- Most systems implement asynchronous cancellation for processes (e.g., use the kill system call) and threads.

- Pthread supports deferred cancellation.

An example of deferred cancellation

```
#include <stdio.h>
#include <pthread.h>
#include <sys/time.h>
#include <unistd.h>
void* thread_func(void* arg)
{
        //pthread_setcancelstate(PTHREAD_CANCEL_DISABLE,
NULL);
        int count = 0;
        while(1) {
                //pthread_testcancel();
                printf("count = %d\n", ++count);
                fflush(stdout);
                sleep(1);
                pthread_testcancel();
        }

        printf("this is a thread cancel test 111\n");
        pthread_testcancel();
```

An example of deferred cancellation

```
        printf("test 222\n");

        printf("Thread cancellation requested.
Exiting...\n");
        pthread_exit(NULL);
        return NULL;
}

int main()
{
        pthread_t thread;
        pthread_create(&thread, NULL, thread_func, NULL);
        sleep(5);
        pthread_cancel(thread);
        printf("call pthread_cancel.\n");
        pthread_join(thread, NULL);
        printf("Thread canceled successfully.\n");
        printf("end\n");
        return 0;
}
```

Output of deferred cancellation example

```
$ gcc -o deferred_cancellation deferred_cancellation.c -
lpthread
$ ./deferred_cancellation
count = 1
count = 2
count = 3
count = 4
count = 5
call pthread_cancel.
Thread canceled successfully.
end
```

Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- All signals follow the same pattern:
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled
- A **signal handler** is used to process signals

C program to illustrate User-defined Signal Handler

```
#define SIGHUP  1   /* Hang up the process */
#define SIGINT  2   /* Interrupt the process */
#define SIGQUIT 3   /* Quit the process */
#define SIGILL  4   /* Illegal instruction. */
#define SIGTRAP 5   /* Trace trap. */
#define SIGABRT 6   /* Abort. */

#include <stdio.h>
#include <signal.h>

// Handler for SIGINT, triggered by
// Ctrl-C at the keyboard
void handle_sigint(int sig)  {
    printf("Caught signal %d\n", sig);
}

int main()  {
    signal(SIGINT, handle_sigint);
    for (int i=0; 1; i++) {
            printf("hello world
%d\n", i);
            sleep(1);
    }
    return 0;
}
```

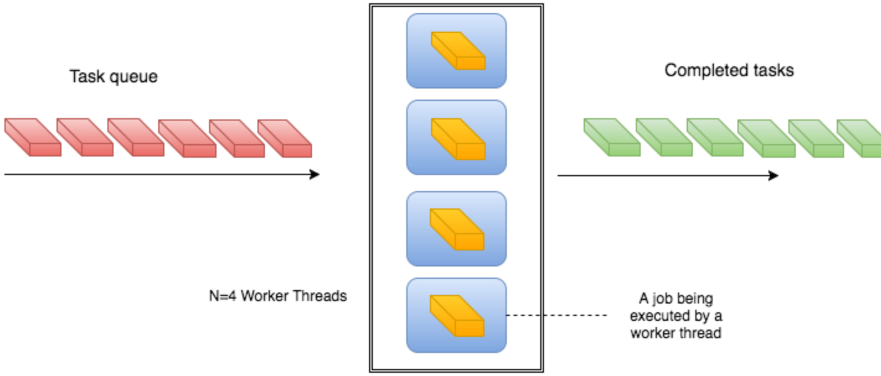
Try Press Ctrl+C

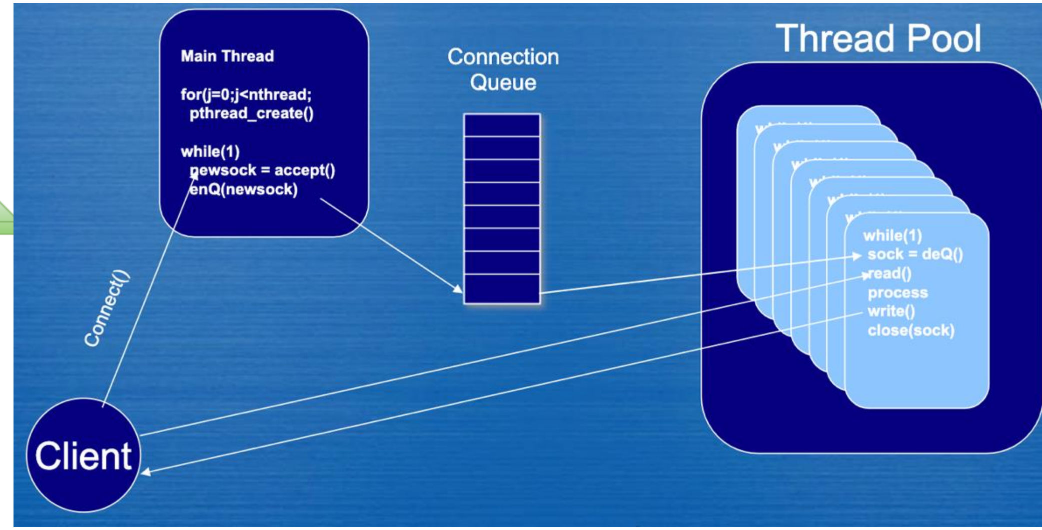
Signal Handling (Cont.)

- How to handle a signal when its target process has multiple threads?
- Options:
  1. Deliver the signal to the thread to which the signal applies
  2. Deliver the signal to every thread in the process
  3. Deliver the signal to certain threads in the process
  4. Assign a specific thread to receive all signals for the process

Thread Pools

- Create a number of threads in a pool where they await work





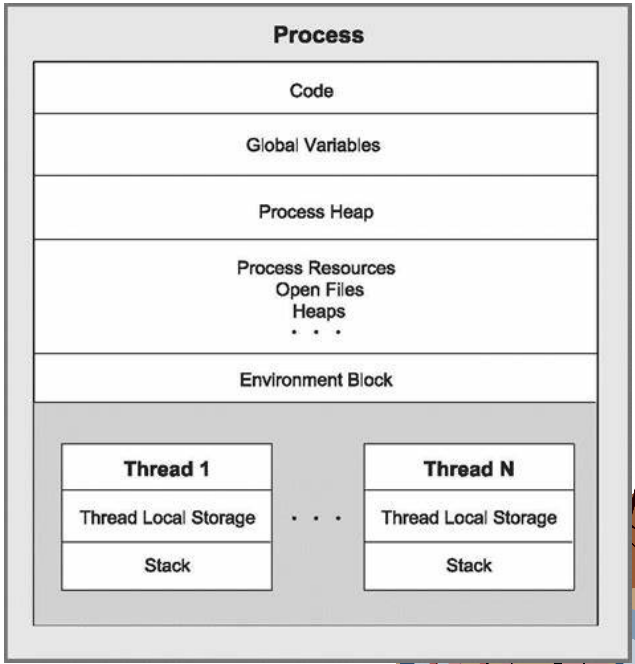
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool

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

Thread Specific Data

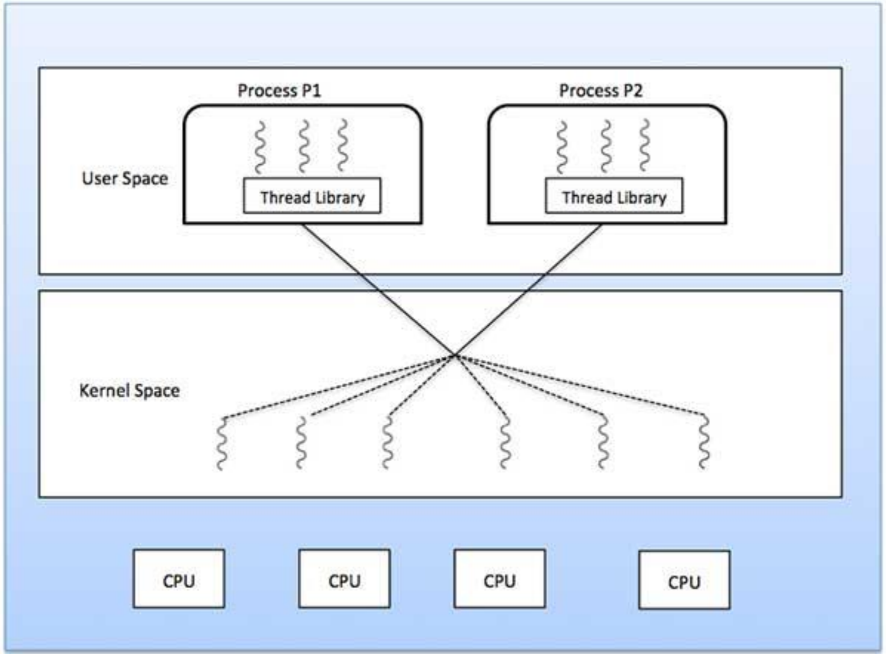
- Allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool) https://en.wikipedia.org/wiki/Thread-local_storages
- Pthreads library supports thread specific data
- pthread_key_create and pthread_key_delete are used respectively to create and delete a key for thread-specific data.

https://en.wikipedia.org/wiki/Thread-local_storage#Pthreads_implementation



Thread Scheduler Activations

- **Background:** Server-version operating systems often use many-to-many and two-level thread models
  - The thread library needs to maintain the appropriate number of kernel threads allocated to the process
  - Requires kernel-user space communication to do it



Thread Scheduler Activations

- Scheduler activations provide **upcalls:** a communication mechanism from the kernel to the user-mode thread lib
- When the kernel knows a thread has blocked/resumed, it notifies the process' run-time system about this event
- This communication allows an application to maintain the correct number of available kernel threads