

# Threads

# 线程

教材：操作系统概念第四章

扩展阅读：深入理解计算机系统（CSAPP） 第十二章

# 内容

基本概念：

- 线程的定义

- 线程的应用场景

- 线程与进程的联系和区别

多线程模型

- 用户级线程 vs 系统级线程

- 多对一 vs 一对一模型

线程库

- POSIX Pthread standard

操作系统对线程的支持：

- Linux

多线程的问题

# 基本概念

# Silly Example

```
main() {  
    ComputePI("pi.txt");  
    PrintClassList("classlist.txt");  
}
```

What is the behavior here?

Program would never print out class list

Why?

ComputePI 一直不结束, 我们就一直看不到 classlist.txt 的打印结果

# Adding Threads

线程是程序中一个单一的顺序控制流程

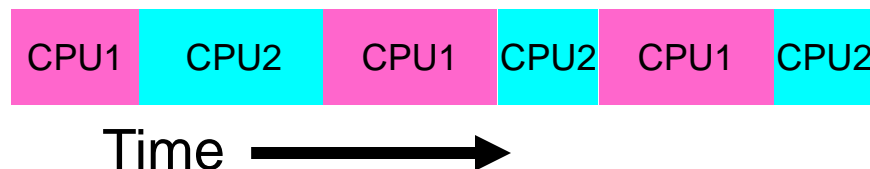
程序中创建了两个线程：两个独立的执行控制流程

```
main() {  
    create_thread(ComputePI, "pi.txt");  
    create_thread(PrintClassList, "classlist.txt");  
}
```

create\_thread:

创建了一个线程，仿佛在另外一个处理器上执行ComputePI 函数中的控制流程，  
又创建了一个线程，仿佛在另外一个处理器上执行PrintClassList 函数中的控制流程

这样就可以看到 classlist 了



# 线程可以屏蔽I/O 延迟

A thread is in one of the following three states:

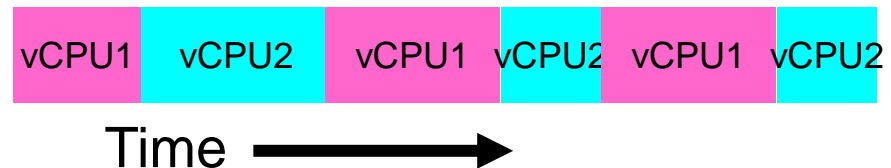
**RUNNING** – running

**READY** – eligible to run, but not currently running

**BLOCKED** – ineligible to run

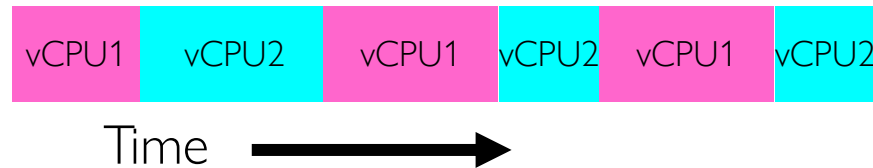
If a thread is waiting for an I/O to finish, the OS marks it as **BLOCKED**

Once the I/O finally finishes, the OS marks it as **READY**

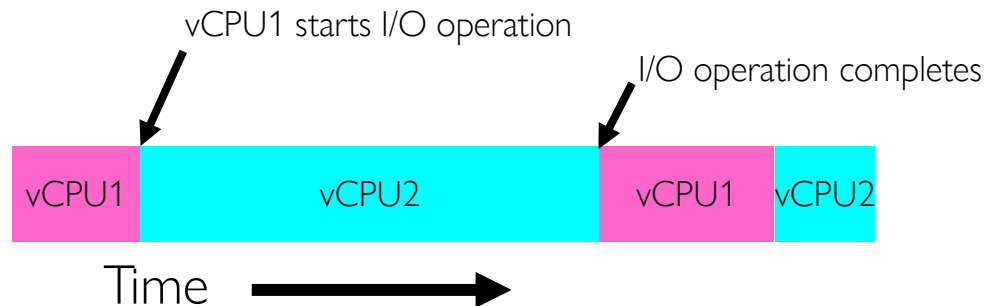


# 线程可以屏蔽I/O 延迟

If no thread performs I/O:



If thread 1 performs a blocking I/O operation:



# A Better Example for Threads

```
main() {  
    create_thread(ReadLargeFile, "pi.txt"); //读大文件  
    create_thread(RenderUserInterface); //用户界面  
}
```

What is the behavior here?

Still respond to user input

While reading file in the background



# Threads run within one application

一个应用中的多个任务可以分离成为多个线程，增加应用的资源占有率、提升响应速度和交互性

例如：文字处理word：

- ▶ 显示线程
- ▶ 读数据线程
- ▶ 拼写检查线程

例如：在线播放器：

- ▶ 一个线程音频解码
- ▶ 一个线程视频解码
- ▶ 一个线程网络接收

例如浏览器：

- ▶ 一个线程显示图片和文字；
- ▶ 另一个线程从网络接收数据

# 引入 Thread概念的动机

**Operating systems** must handle multiple things at once (MTAO)

Processes, interrupts, background system maintenance

**Networked servers** must handle MTAO

Multiple connections handled simultaneously

**Parallel programs** must handle MTAO

To achieve better performance

**Programs with user interface** often must handle MTAO

To achieve user responsiveness while doing computation

**Network and disk bound programs** must handle MTAO

To hide network/disk latency

Sequence steps in access or communication

# Threads vs. Processes

If we have two tasks to run concurrently, do we run them in separate threads, or do we run them in separate processes?

Depends on how much isolation we want

Threads are lighter weight [why?]

Processes are more strongly isolated

# Process vs. Thread

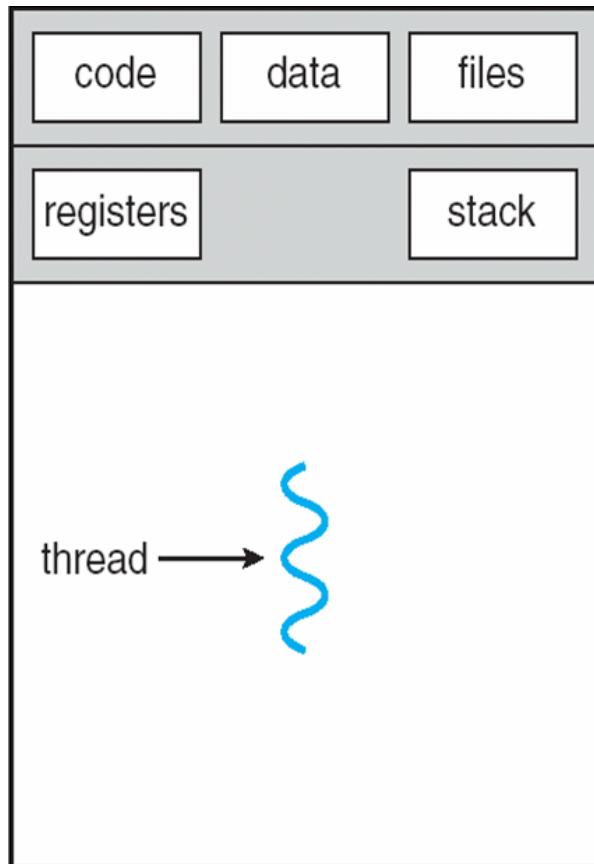
## Process

1. 独立
2. 携带相对更多的状态信息
3. 有独立的地址空间
4. 通过IPC(进程间通信机制)通信
5. 上下文切换速度较慢

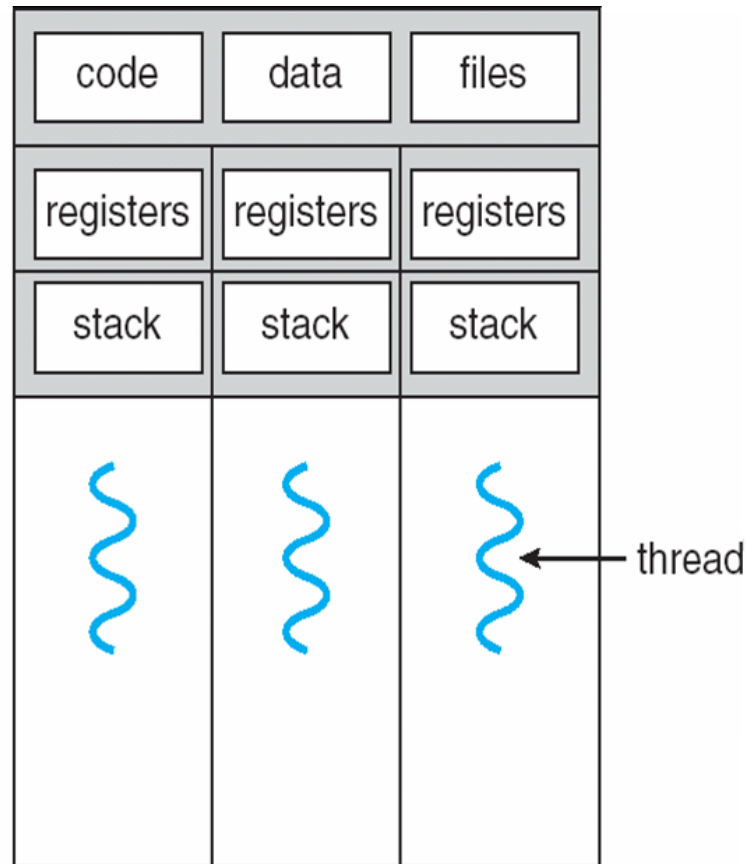
## Thread

1. 作为进程的子集存在
2. 共享进程的状态、内存、资源
3. 共享进程的地址空间
4. 线程之间通信更方便
5. 在同一个进程空间内的上下文切换, 速度更快

# Single and Multithreaded Processes



single-threaded process



multithreaded process

# 一个线程的信息包含哪些？

- 线程是程序中一个单一的顺序控制流程，是系统独立调度和分派CPU的基本单位。

一个线程独有：

线程ID, 程序计数器, 寄存器集, 栈 stack

- 单个进程中同时运行多个线程完成不同的工作：多线程

同一个进程的多个线程可共享：

- ▶ 共享代码段
- ▶ 共享全局数据段
- ▶ 共享资源，例如打开的文件

# Traditional View of a Process

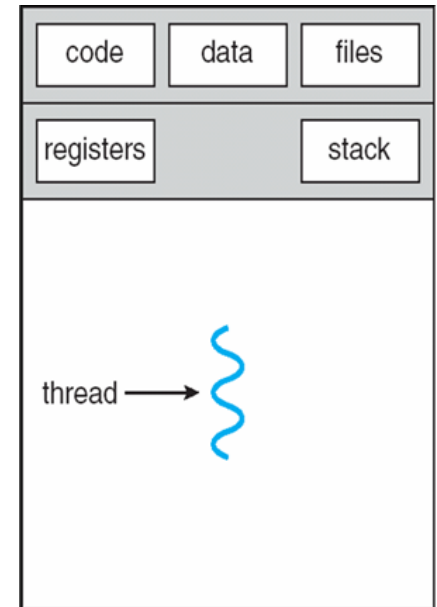
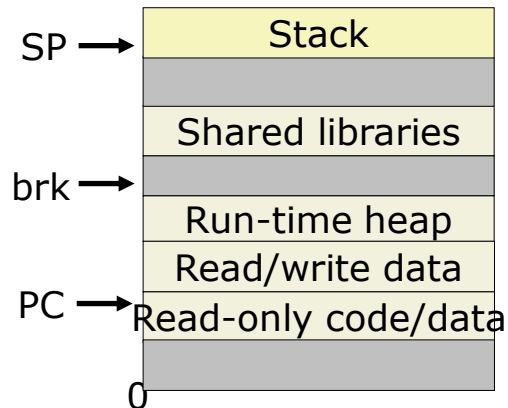
Process = process context + code, data, and stack

## Process context

Program context:  
Data registers  
Condition codes  
Stack pointer (SP)  
Program counter (PC)

Kernel context:  
VM structures  
Descriptor table  
(中断描述符表)  
brk pointer

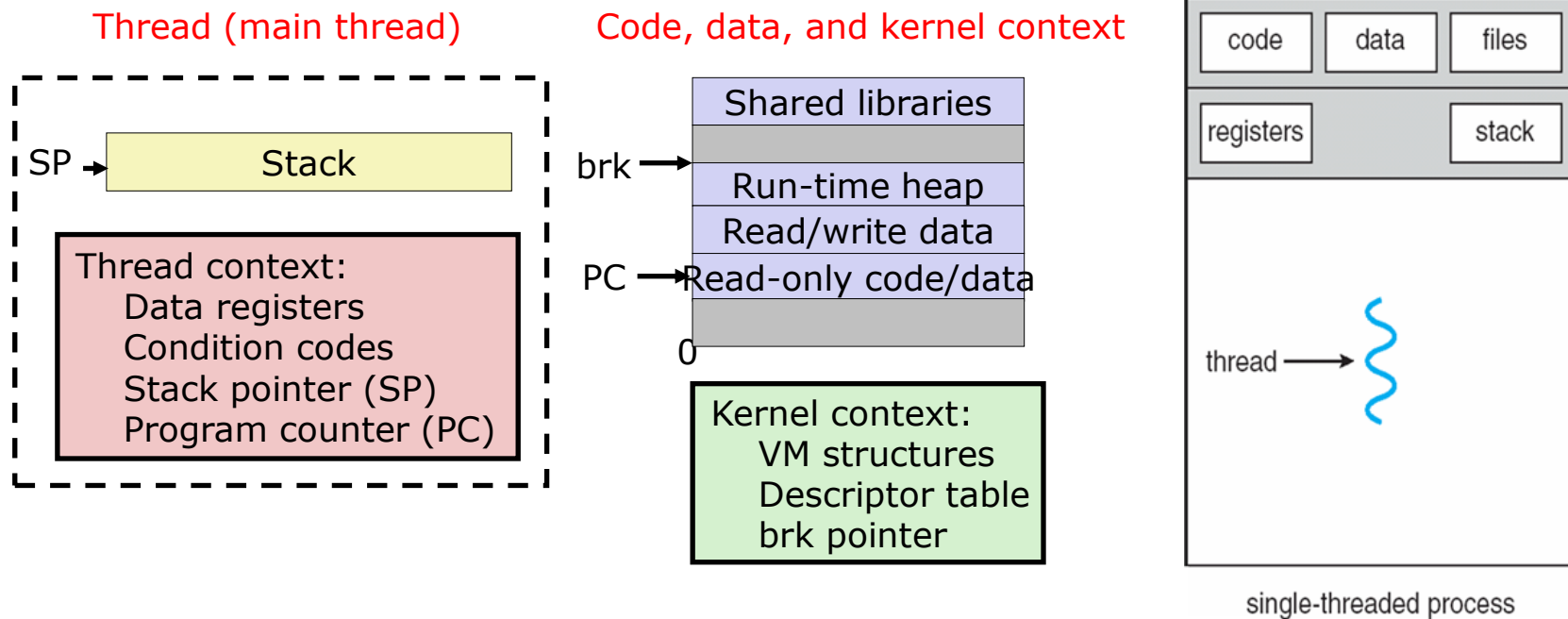
## Code, data, and stack



single-threaded process

# Alternate View of a Process

Process = thread + code, data, and kernel context





# A Process With Multiple Threads

Multiple threads can be associated with a process

Each thread shares the same code, data, and kernel context

Each thread has its own logical control flow

Each thread has its own stack for local variables

- ▶ but not protected from other threads

Each thread has its own thread id (TID)

Shared code and data

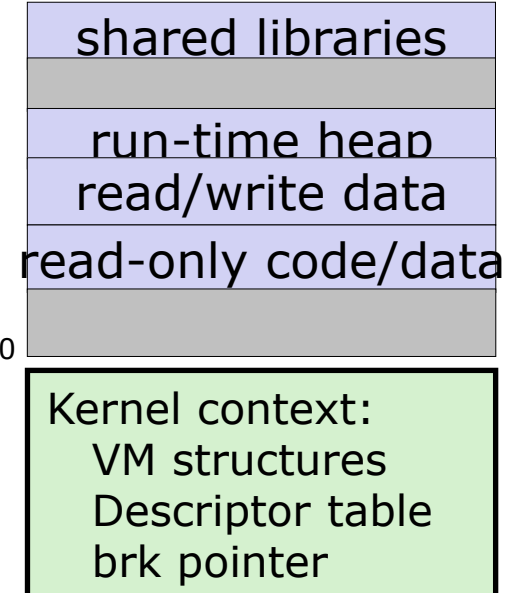
Thread 1 (main thread)    Thread 2 (peer thread)

stack 1

stack 2

Thread 1 context:  
Data registers  
Condition codes  
SP1  
PC1

Thread 2 context:  
Data registers  
Condition codes  
SP2  
PC2



# 练习

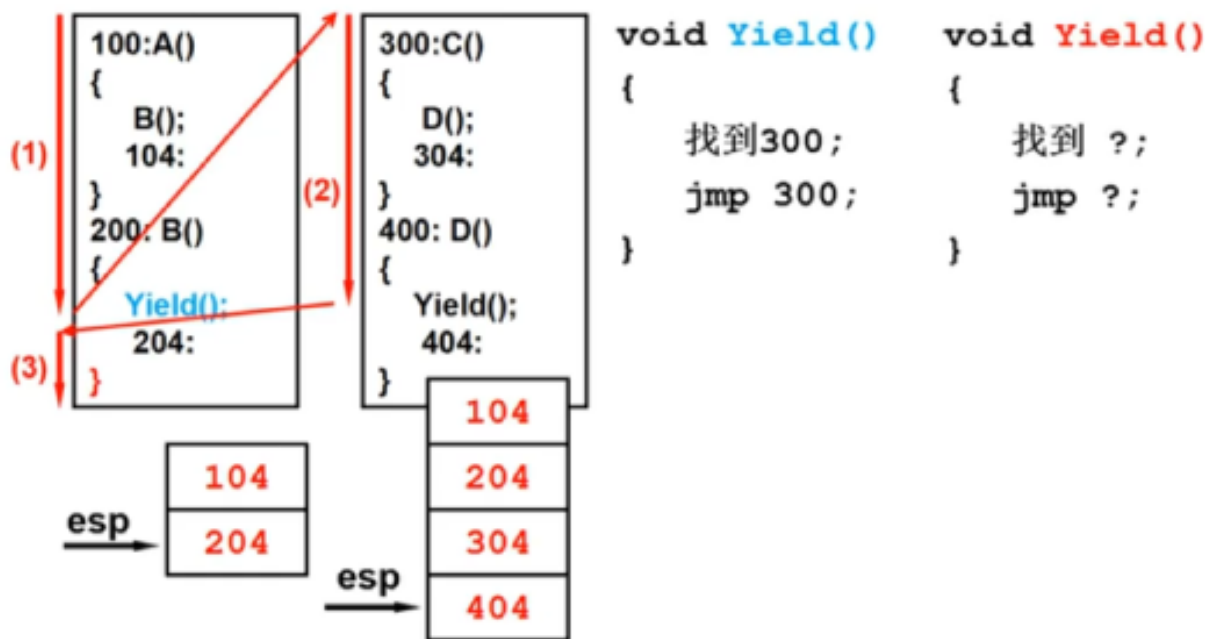
同一个进程中的线程，不可以共享（）。

- A. 堆栈
- B. 代码
- C. 公有数据
- D. 打开文件列表

# Why each thread has its own stack ?

yield()的作用:暂停当前正在执行的线程(放弃拥有的cup资源),并执行其他线程。

两个执行序列与一个栈... 会出什么问题?



# 线程的实现模型:用户级、内核级

# Supports for Threads

## Kernel level Threads

Supported and managed by the operating system kernel

Examples

- ▶ Windows XP/2000, Solaris, Linux, Tru64 UNIX, Mac OS X

## User level Threads

Thread management done by user-level threads library, supported above the kernel and are managed without kernel support

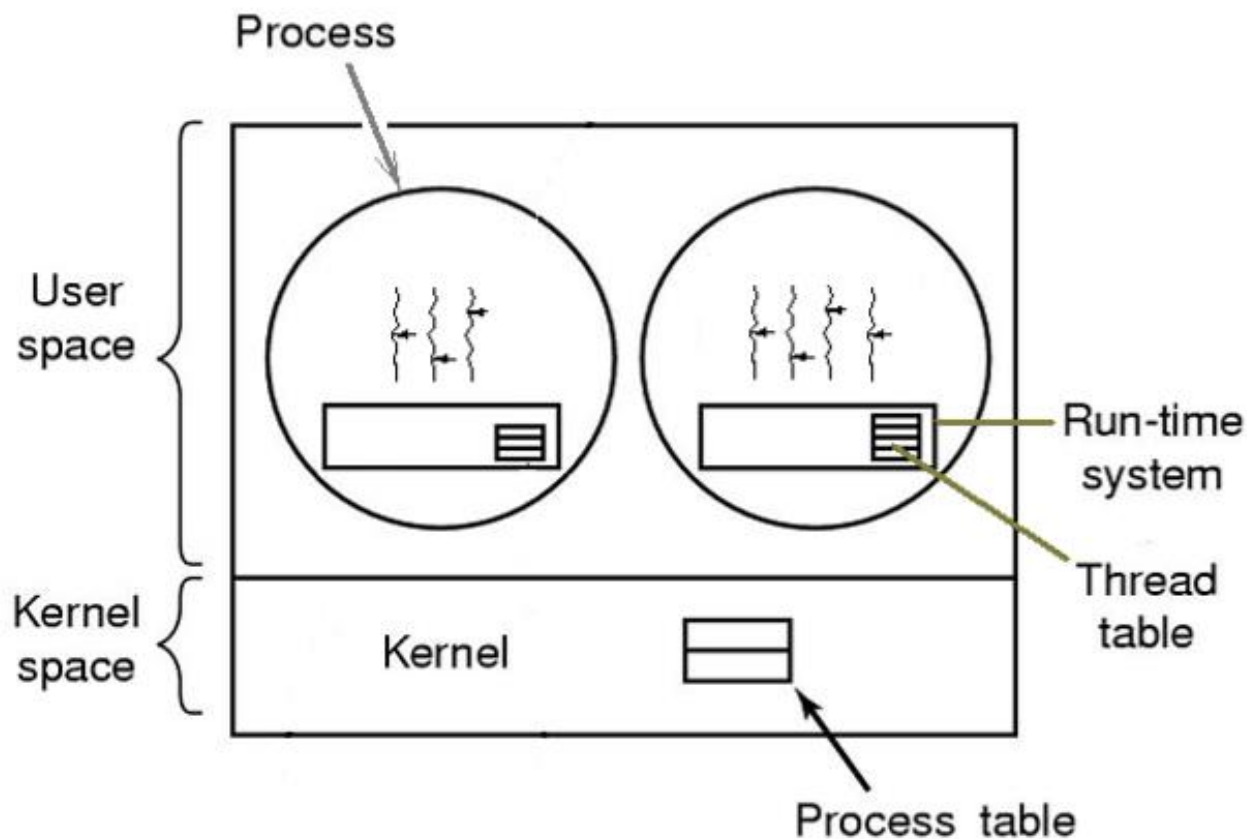
Three primary thread libraries:

- ▶ Win32 threads
- ▶ Java threads

# 在用户进程中实现线程

用户进程利用线程库(run-time system)提供的  
创建、同步、调度等管理线程的函数来控制线程，  
线程的控制信息也存放在进程用户数据区中

库函数



# 用户级线程的优缺点

## 优点:

线程的调度不需要内核直接参与，控制简单。

可以在不支持线程的操作系统中实现。

创建和销毁线程、线程切换代价等线程管理的代价比内核线程少得多。

允许每个进程定制自己的调度算法，线程管理比较灵活。这就是必须自己写管理程序，与内核线程的区别

线程能够利用的表空间和堆栈空间比内核级线程多。

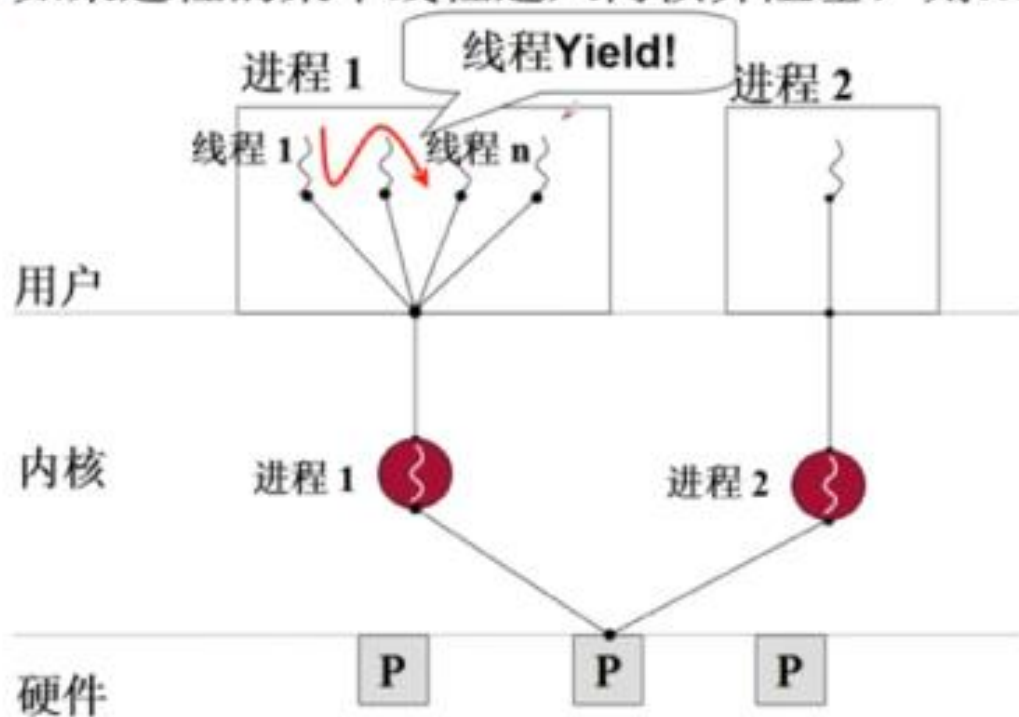
## 缺点:

同一进程中只能同时有一个线程在运行，如果有一个线程使用了系统调用而阻塞，那么整个进程都会被挂起。另外，页面失效也会产生同样的问题。

资源调度按照进程进行，多个处理机下，同一个进程中的线程只能在同一个处理机下分时复用

## 为什么说用户级线程——Yield是用户程序

- 如果进程的某个线程进入内核并阻塞，则...



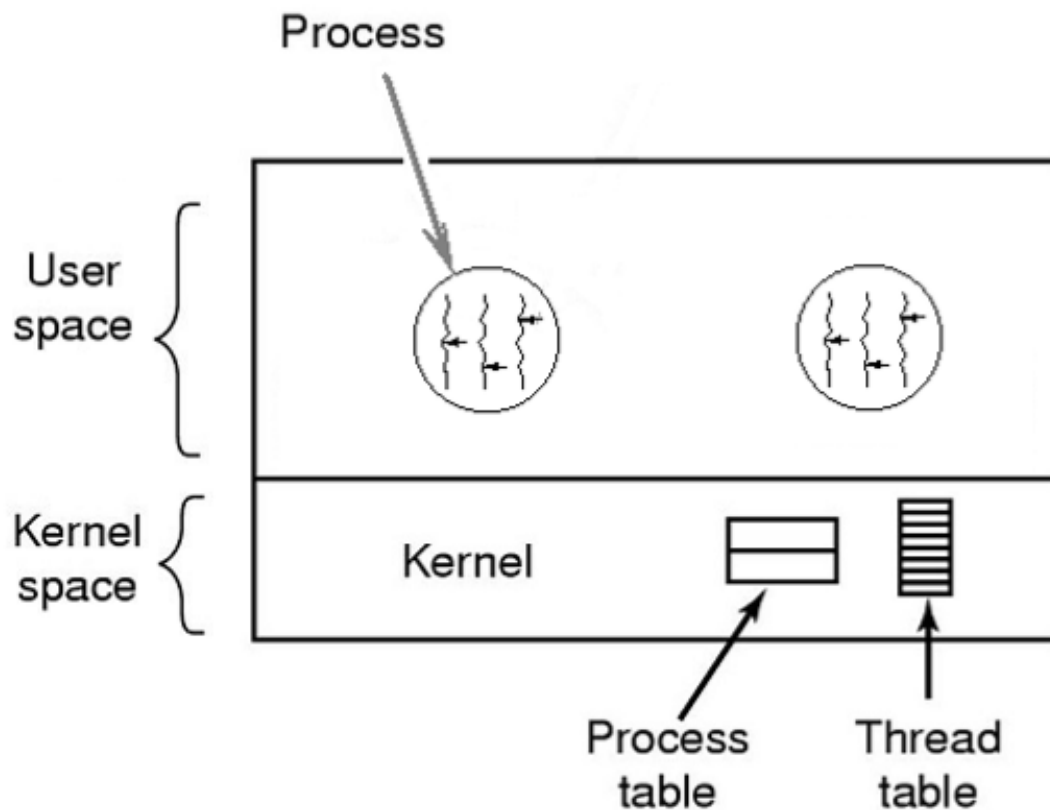
同一进程中只能同时有一个线程在运行，**如果有一个线程使用了系统调用而阻塞，那么整个进程都会被挂起。**另外，页面失效也会产生同样的问题。



# 在内核中实现线程

由内核实现线程的创建和撤销，  
线程的控制信息存放在内核

系统调用



# 内核级线程与用户级线程的区别

内核支持线程是OS内核可感知的， 用户级线程是OS内核不可感知的

内核支持线程的创建、撤消和调度都需OS内核提供支持， 而且与进程的创建、撤消和调度大体是相同的。

用户级线程执行系统调用指令时将导致其所属进程被中断；  
内核支持线程执行系统调用指令时， 只导致该线程被中断。

在只有用户级线程的系统内， CPU调度还是以进程为单位， 处于运行状态的进程中的多个线程， 由用户程序控制线程的轮换运行； 在有内核支持线程的系统内， CPU调度则以线程为单位， 由OS的线程调度程序负责线程的调度。

用户级线程的程序实体是运行在用户态下的程序， 而内核支持线程的程序实体则是可以运行在任何状态下的程序。

# 线程模型

# Multithreading Models

## 用户级线程与内核级线程之间的关系

Many-to-One

One-to-One

Many-to-Many

Two-Level Model

# Many-to-One Model

**Many user-level threads are mapped to a single kernel thread**

## 优点

Multiple threads are hidden by user-level thread library

## 缺点

单个线程阻塞会造成整个进程

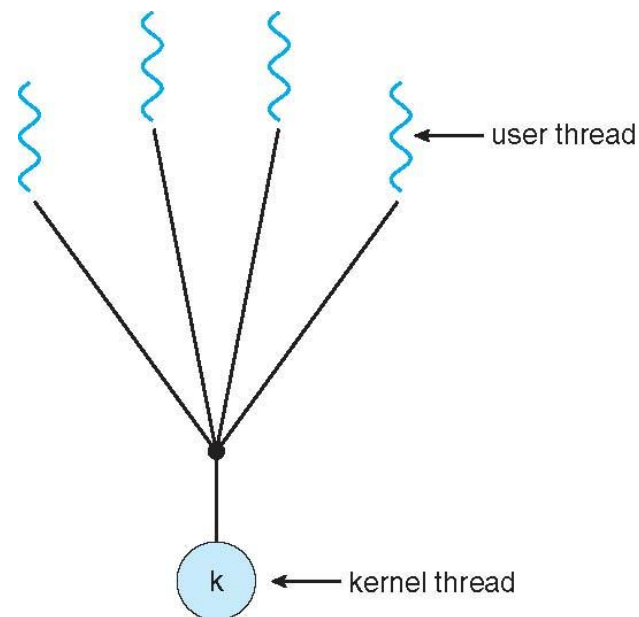
多个线程无法运行在多处理器上

多数系统已经放弃这个模式

Solaris Green Threads

GNU Portable Threads

早期Java 版本



# 练习

某个分时系统采用多对一线程模型。内存中有10个进程并发运行，其中9个进程中只有一个线程，另外一个进程A拥有11个线程。则A获得的CPU时间占总时间的()。

A.  $1/20$

B. 1

C. 0

D.  $1/10$

# One-to-One

**Each user-level thread is mapped to a kernel thread**

优点

并发性，可充分利用多处理器的优势

一个线程阻塞可以切换到同一个进程的另外一个线程

缺点

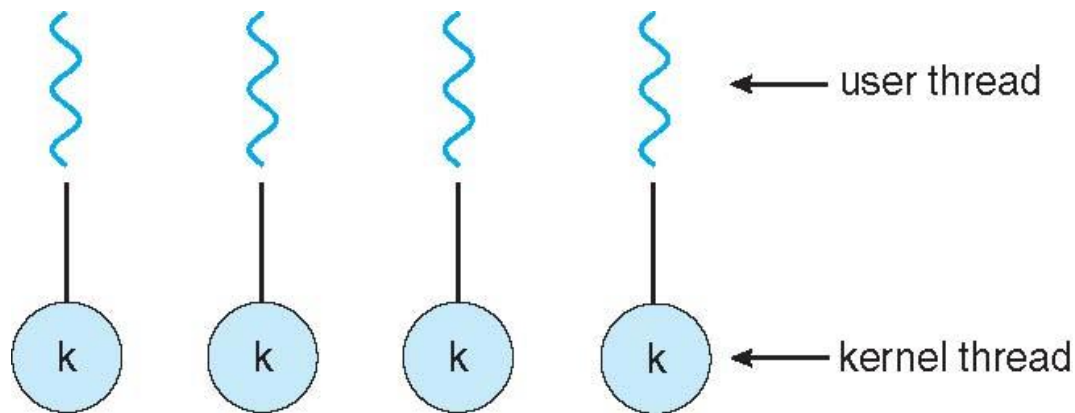
系统开销大

Examples

Windows NT/XP/2000

Solaris 9 and later

Linux



# 练习

某个分时系统采用一对一线程模型。内存中有10个进程并发运行，其中9个进程中只有一个线程，另外一个进程A拥有11个线程。则A获得的CPU时间占总的时间的()。

- A.  $1/20$
- B. 1
- C.  $11/20$
- D.  $1/10$



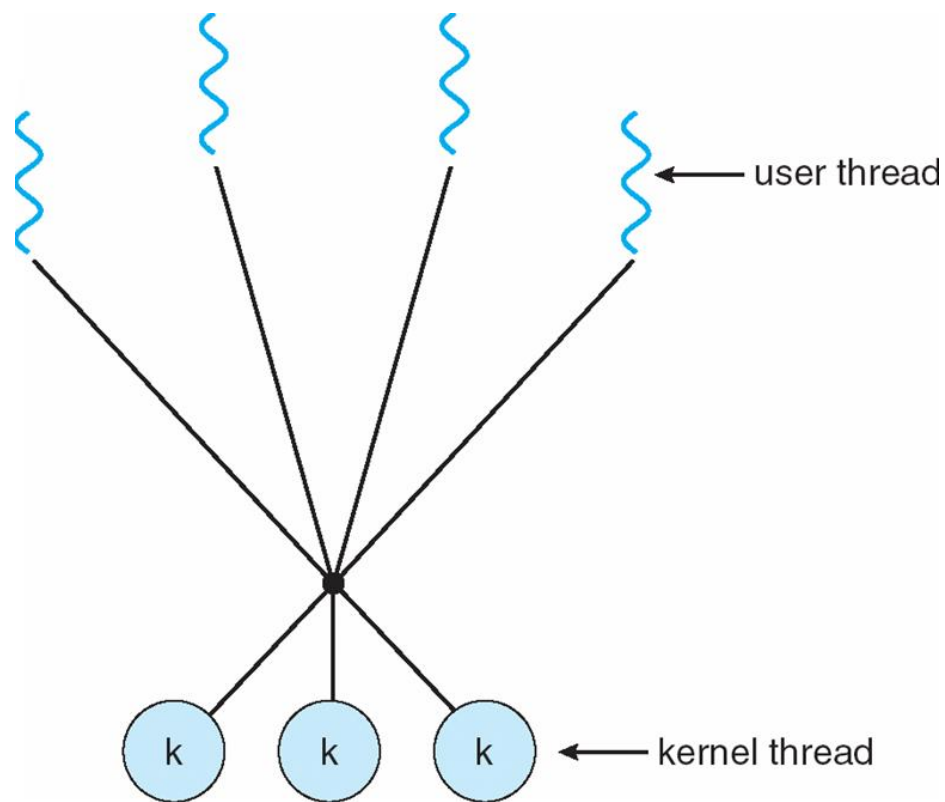
# Many-to-Many Model

**Allows many user level threads to be mapped to many kernel threads**

The operating system creates a sufficient number of kernel threads

Examples

Windows NT/2000 with the *ThreadFiber* package



# Two-Level Model

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

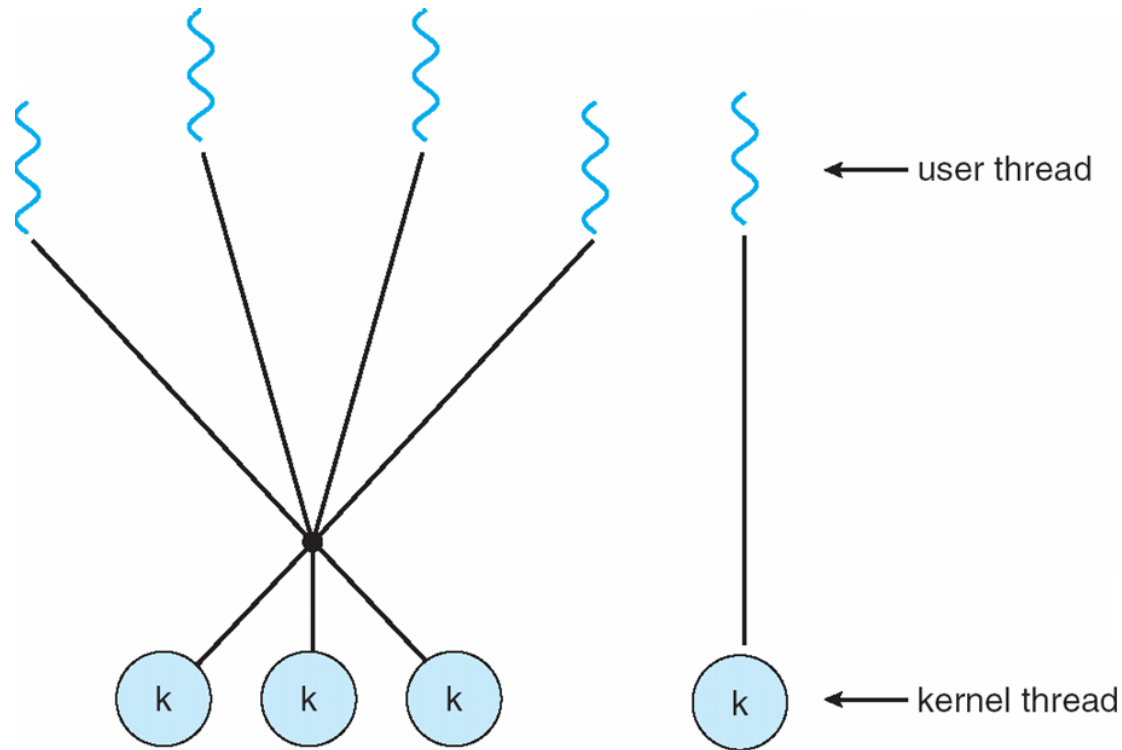
## Examples

IRIX

HP-UX

Tru64 UNIX

Solaris 8 and earlier



# 线程库

# Pthreads

Is provided either in user-level or kernel-level

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

Common in UNIX operating systems (Solaris, Linux, Mac OS X)

# Posix Threads (Pthreads) Interface

## Creating and reaping threads

- ▶ `pthread_create()`
- ▶ `pthread_join()`

## Determining your thread ID

- ▶ `pthread_self()`

## Terminating threads

- ▶ `pthread_cancel()`
- ▶ `pthread_exit()`
- ▶ `exit()` [terminates all threads] , `RET` [terminates current thread]

## Block threads

- ▶ `pthread_yield()`

## Synchronizing access to shared variables

- ▶ `pthread_mutex_init`
- ▶ `pthread_mutex_[un]lock`

# The Pthreads "hello, world" Program

```
void *thread(void *vargp);
```

```
int main()
```

```
{  
    pthread_t tid;  
    Pthread_create(&tid, NULL, thread, NULL);  
    Pthread_join(tid, NULL);  
    exit(0);  
}
```

Thread ID

Thread attributes  
线程属性  
(usually NULL)

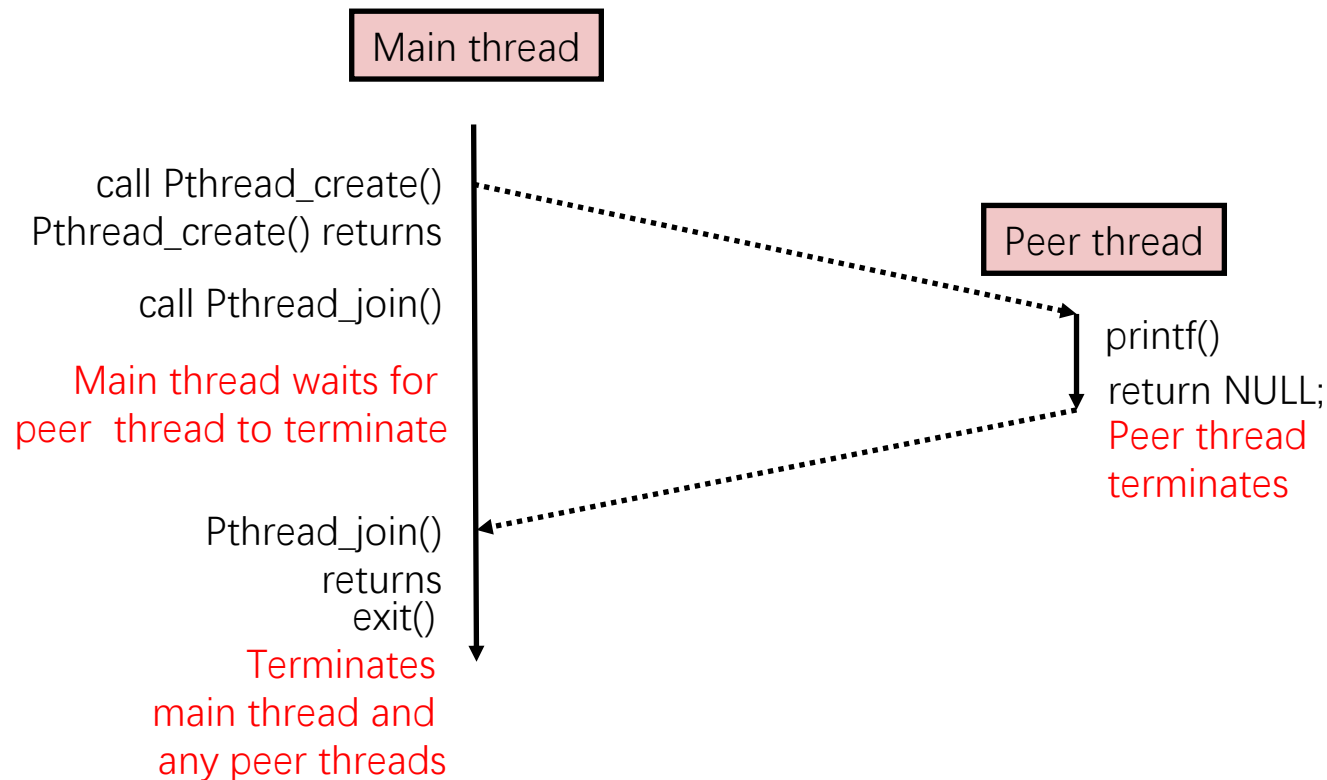
Thread routine  
线程运行函数的起始地址

Thread arguments  
传递给线程的参数(void \*p)

Return value  
(void \*\*p)  
用户定义的指针,  
存储被等待线程的返回值

```
void *thread(void *vargp) /* thread routine */  
{  
    printf("Hello, world!\n");  
    return NULL;  
}
```

# Execution of Thread “hello, world”



# pthread\_create 与 System Call

What happens when pthread\_create(...) is called in a process?

Library:

```
int pthread_create(...) {  
    Do some work like a normal fn..  
  
    asm code ... syscall # into %eax  
    put args into registers %ebx, ...  
    special trap instruction
```

Kernel:    get args from regs  
            dispatch to system func  
            Do the work to spawn the new thread  
            Store return value in %eax

```
    get return values from regs  
    Do some more work like a normal fn..  
};
```



# 多线程编程实例1

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

static long x = 0;

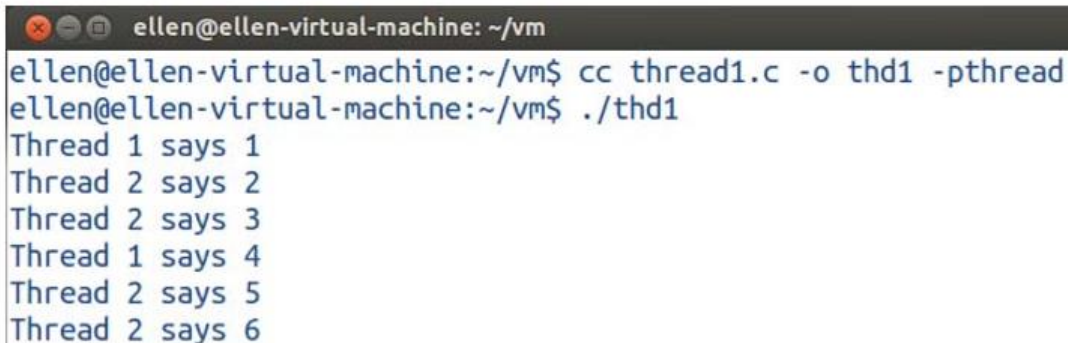
static void *thr_fn(void *arg) {
    while(x < 40) {
        printf("Thread 2 says %ld \n", ++x);
        sleep(1);
    }
}

int main() {
    pthread_t tid;

    int err = pthread_create(&tid, NULL, thr_fn, NULL);
    if (err != 0) {
        printf("can't create thread: %d\n", strerror(err));
        exit(1);
    }

    while(x < 20) {
        printf("Thread 1 says %ld \n", ++x);
        sleep(2);
    }

    return 0;
}
```



```
ellen@ellen-virtual-machine: ~/vm
ellen@ellen-virtual-machine:~/vm$ cc thread1.c -o thd1 -pthread
ellen@ellen-virtual-machine:~/vm$ ./thd1
Thread 1 says 1
Thread 2 says 2
Thread 2 says 3
Thread 1 says 4
Thread 2 says 5
Thread 2 says 6
```

# 练习题

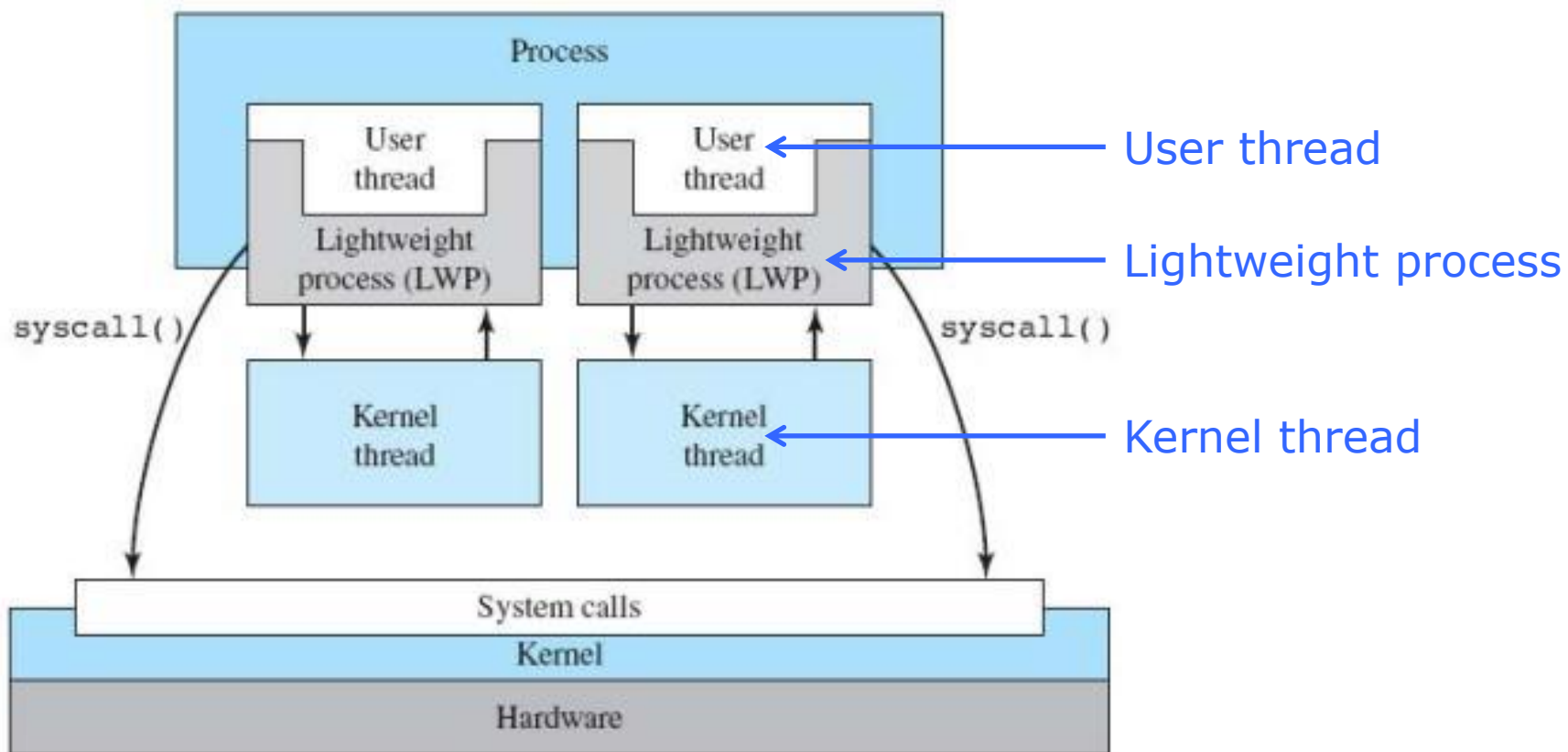
```
int value = 0;
void *runner(void *param) {
    value = 5;
    pthread_exit ( 0 );
}

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

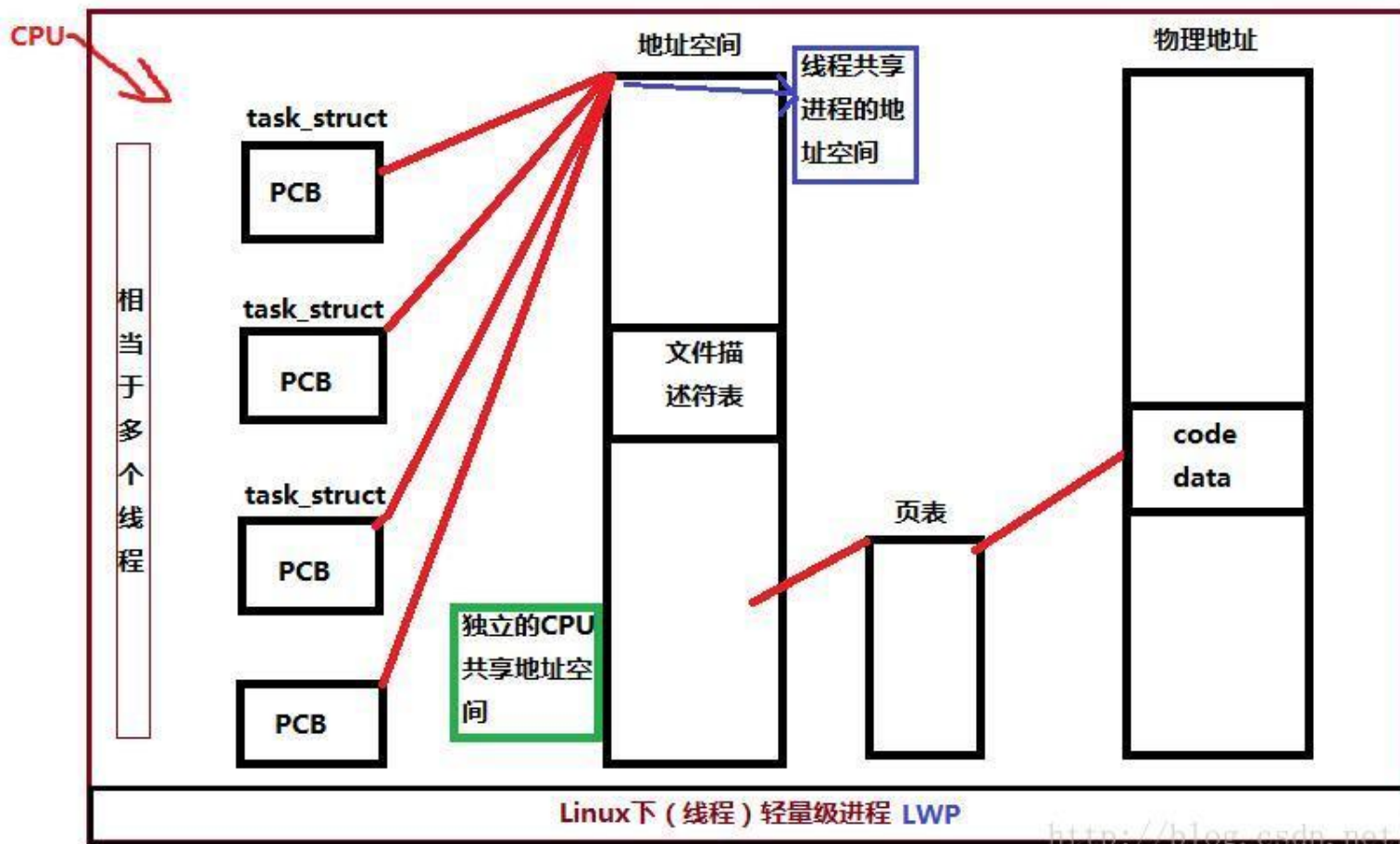
```
    if (pid == 0) {
        pthread_attr_init (&attr) ;
        pthread_create(&tid, &attr, runner, NULL);
        pthread_join(tid,NULL) ;
        printf("Child: value = %d", value);
    }
    else if (pid > 0) {
        wait (NULL) ;
        printf("Parent: value = %d", value);
    }
}
```

What are the outputs from the above program?

# Linux Thread Model



# Linux LWP



# Linux LWP

`fork()` and `clone()` system calls

`clone()` takes options to determine sharing on process create

`struct task_struct` points to process data structures  
(shared or unique)

flag	meaning
<code>CLONE_FS</code>	File-system information is shared.
<code>CLONE_VM</code>	The same memory space is shared.
<code>CLONE_SIGHAND</code>	Signal handlers are shared.
<code>CLONE_FILES</code>	The set of open files is shared.

# A Process With Multiple Threads

Multiple threads can be associated with a process

Each thread shares the same code, data, and kernel context

Each thread has its own logical control flow

Each thread has its own stack for local variables

▶ but not protected from other threads

Each thread has its own thread id (TID)

Shared code and data

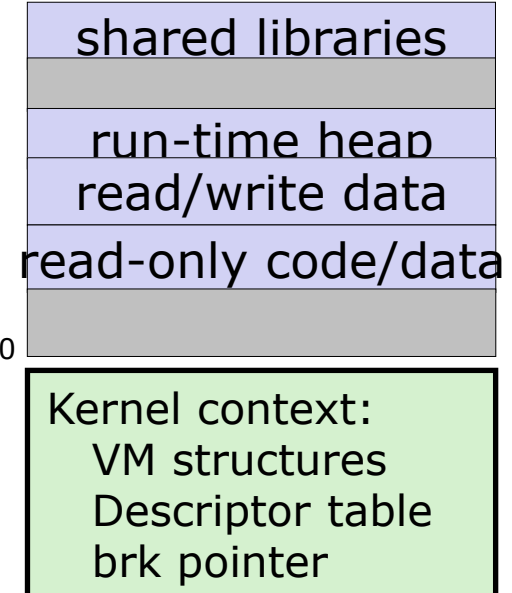
Thread 1 (main thread)    Thread 2 (peer thread)

stack 1

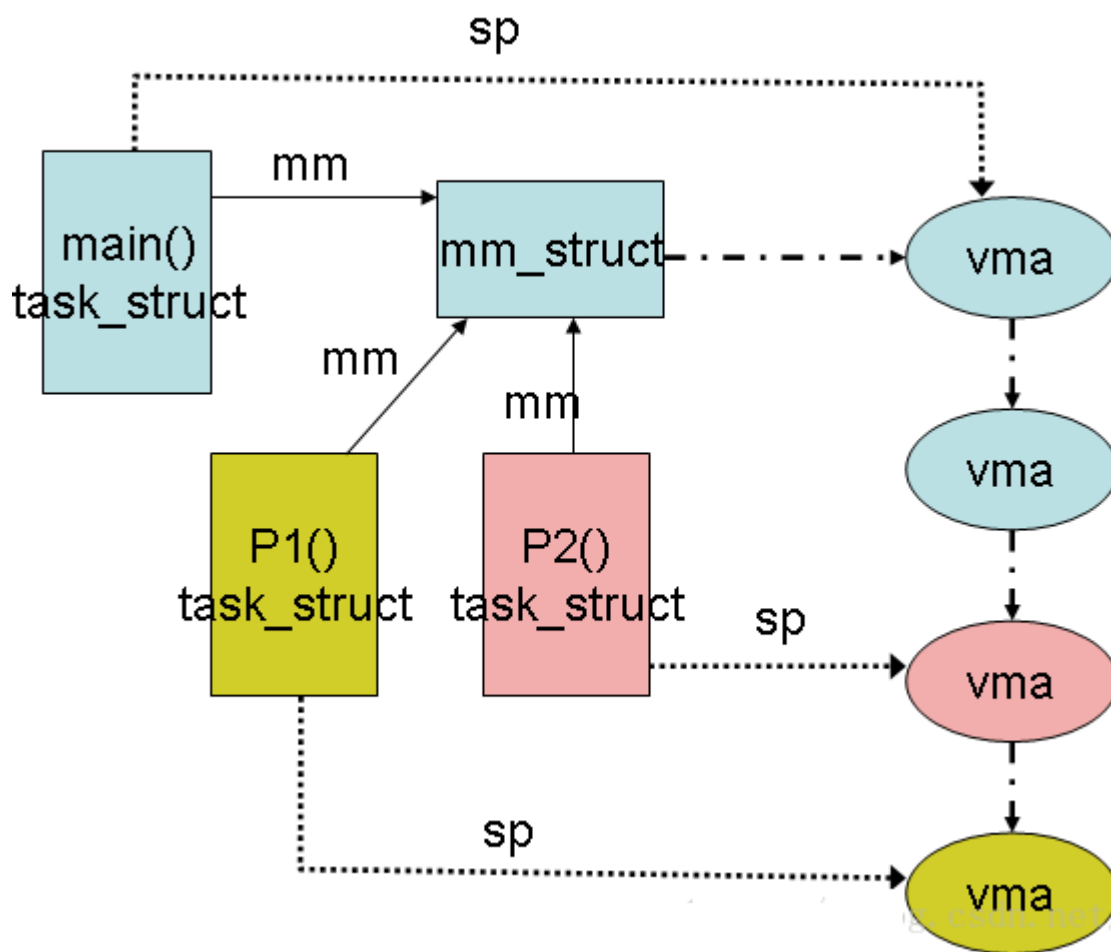
stack 2

Thread 1 context:  
Data registers  
Condition codes  
SP1  
PC1

Thread 2 context:  
Data registers  
Condition codes  
SP2  
PC2



# 进程创建和线程创建的不同



pthread库是通过调用mmap()来为新的线程创建栈空间

mmap函数：将一个文件或者其它对象映射进内存

[进程栈与线程栈的关系](#) [human ! 的专栏-CSDN博客](#) [进程栈和线程栈](#)

# 参考

操作系统 与 linux 内核

<https://www.icourse163.org/course/XIYOU-1461809182>



# 判断对错

Unix的Fork创建的是轻量级进程，它可以和创建它的父进程共享各类资源，从而使得它的创建、切换成本较低。

Unix的exec创建的进程可以和创建它的父进程共享各类资源，从而使得它的创建、切换成本较低。

# 线程的问题

# Threading Issues

**Semantics** of **fork()** and **exec()** system calls

Does **fork()** duplicate only the calling thread or all threads?

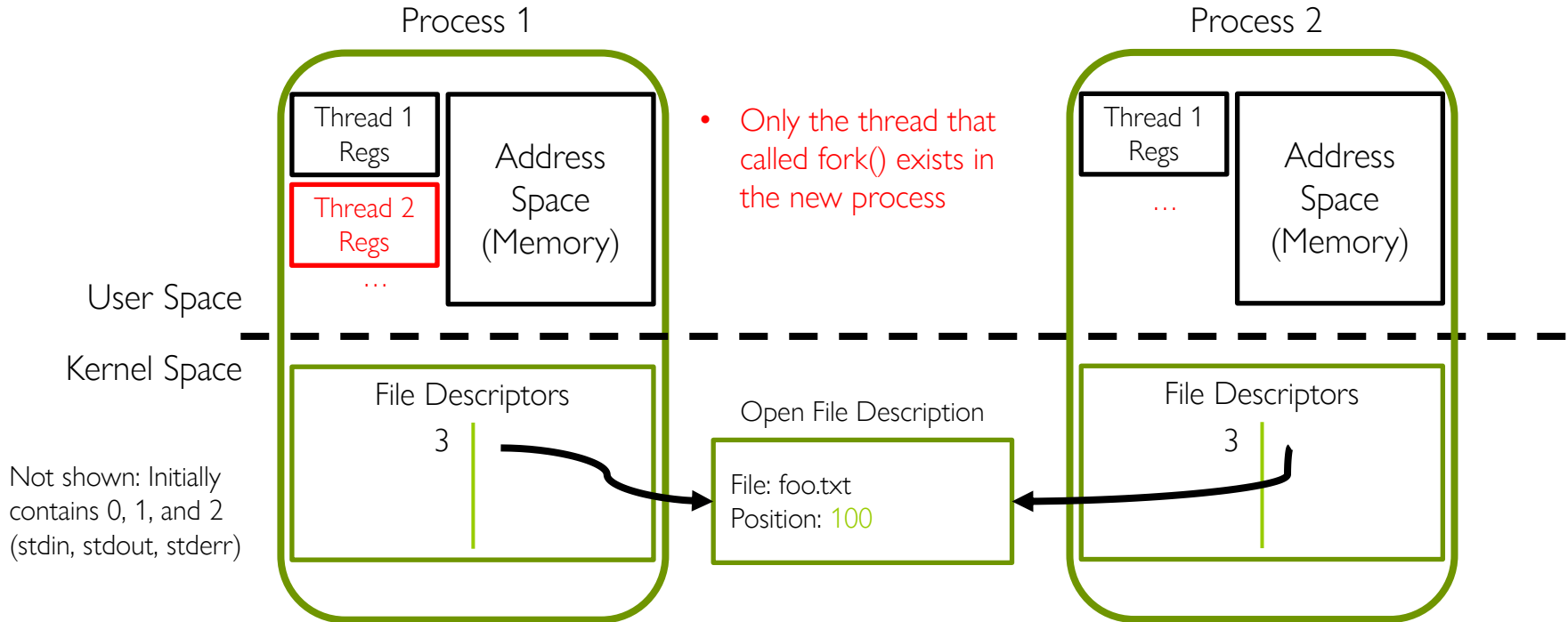
**exec()** will replace the entire process with the program specified in the parameter

---

Unless you plan to call `exec()` in the child process

**DON'T FORK() IN A PROCESS THAT  
ALREADY HAS MULTIPLE THREADS**

# fork() in a Multithreaded Processes



The child process always has just a single thread

- The thread in which fork() was called
- The other threads just vanish

# Possible Problems with Multithreaded `fork()`

When you call `fork()` in a multithreaded process, the other threads (the ones that didn't call `fork()`) just vanish

What if one of these threads was holding a lock?

What if one of these threads was in the middle of modifying a data structure?

No cleanup happens!

It's safe if you call `exec()` in the child

Replacing the entire address space

# Threading Issues (Cont.)

## Thread cancellation of target thread

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.

# Threading Issues (Cont.)

## Signal handling

Signals are used in UNIX systems to notify a process that a particular event has occurred.

## Synchronous and asynchronous

A **signal handler** is used to process signals

1. Signal is generated by particular event
2. Signal is delivered to a process
3. Signal is handled

## Delivery options:

- ▶ Deliver the signal to the thread to which the signal applies
- ▶ Deliver the signal to every thread in the process
- ▶ Deliver the signal to certain threads in the process
- ▶ Assign a specific thread to receive all signals for the process



# Threading Issues (Cont.)

## 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

## Thread-specific data

Create Facility needed for data private to thread

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)

## Scheduler activations

Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application

# 总结

## Process

1. 独立
2. 携带相对更多的状态信息
3. 有独立的地址空间
4. 通过IPC(进程间通信机制)通信
5. 上下文切换速度较慢

## Thread

1. 作为进程的子集存在
2. 共享进程的状态、内存、资源
3. 共享进程的地址空间
4. 线程之间通信更方便
5. 在同一个进程空间内的上下文切换, 速度更快