

# 第四章：线程（Thread）

---

1. 概述
2. 多线程模型
3. 线程库
4. 多线程问题

Note : These lecture materials are based on the lecture notes prepared by the authors of the book titled *Operating System Concepts*.

# 第一节、线程概述

为了提高CPU的使用率，目前我们想到的办法是多道程序，即当一个进程阻塞时（比如发生I/O请求的时候）切换到另一个进程运行。

1. 这确实能提高CPU的使用率，但伴随着一些代价，比如上下文切换。
2. 那么，有没有更好的办法既能提高CPU的使用率又能减少代价？

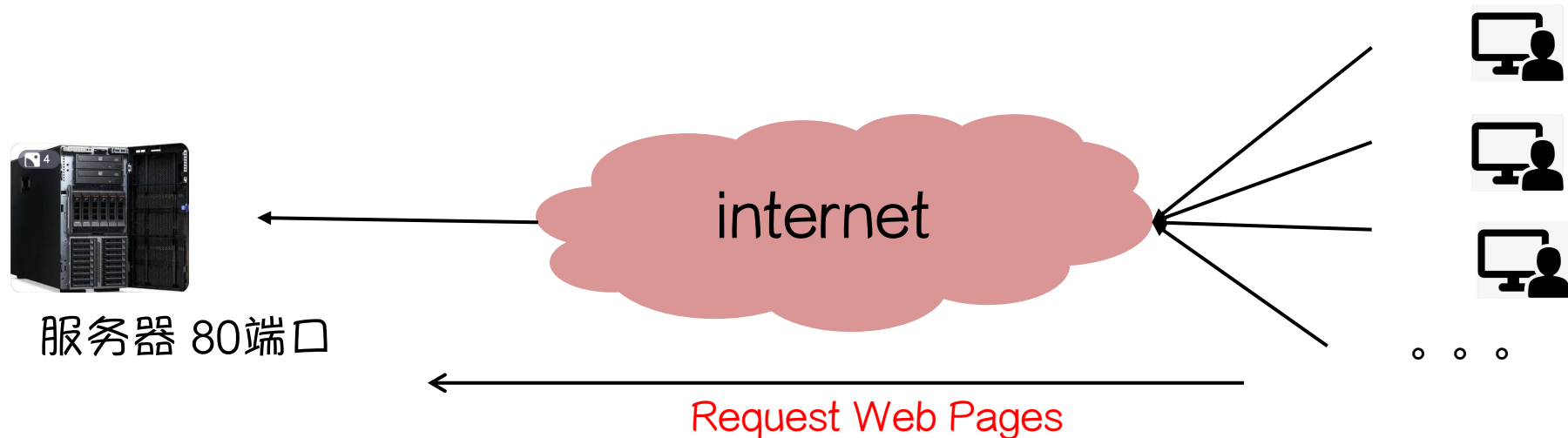
# 1. 概述

## 做料理的任务

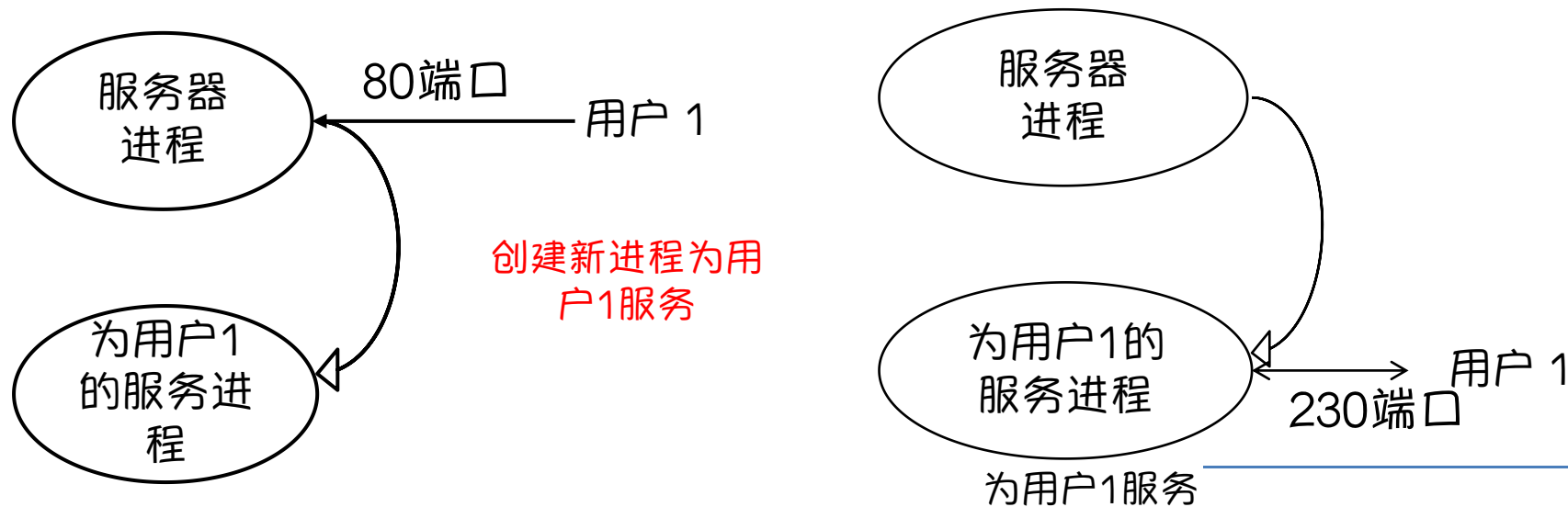


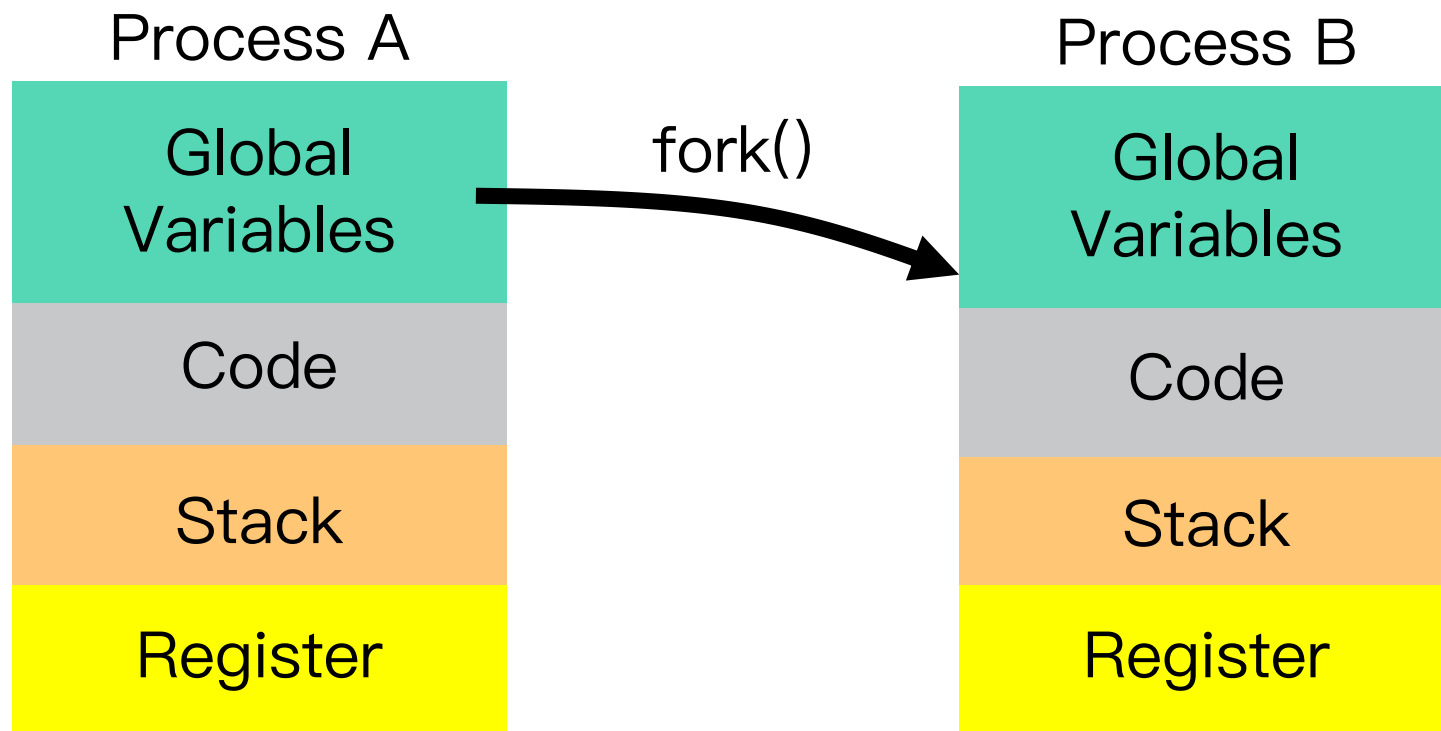
1. 切菜
2. 煮水
3. 调味

进程



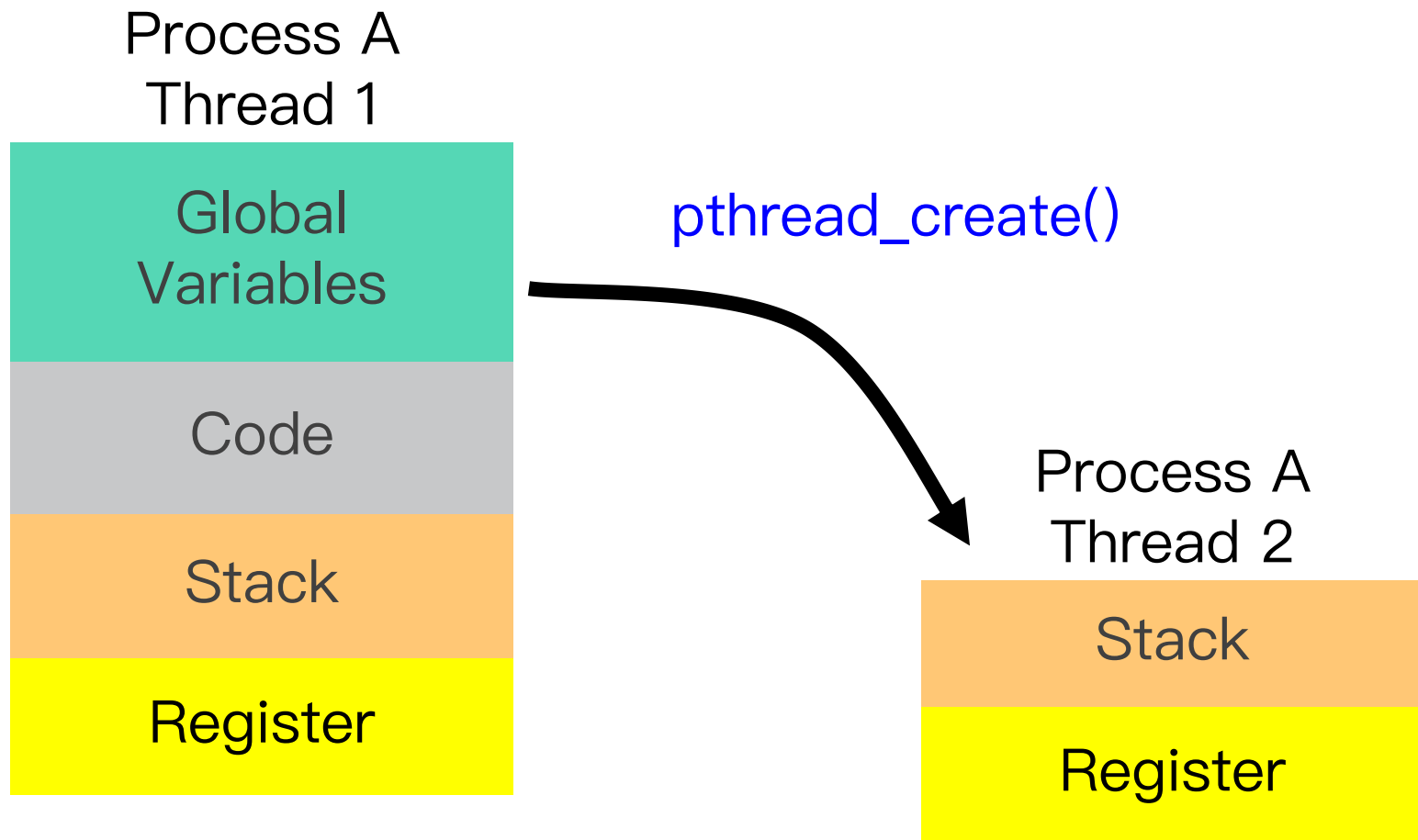
## 服务程序的多进程实现方法





创建进程是重量级的操作过程，而且进程之间的上下文切换也是一个代价比较高的操作，因为PCB以及很多数据需要交换。

重量级：heavy-weight process



相对创建进程，创建线程比较简单，  
只需要复制栈和寄存器的内容

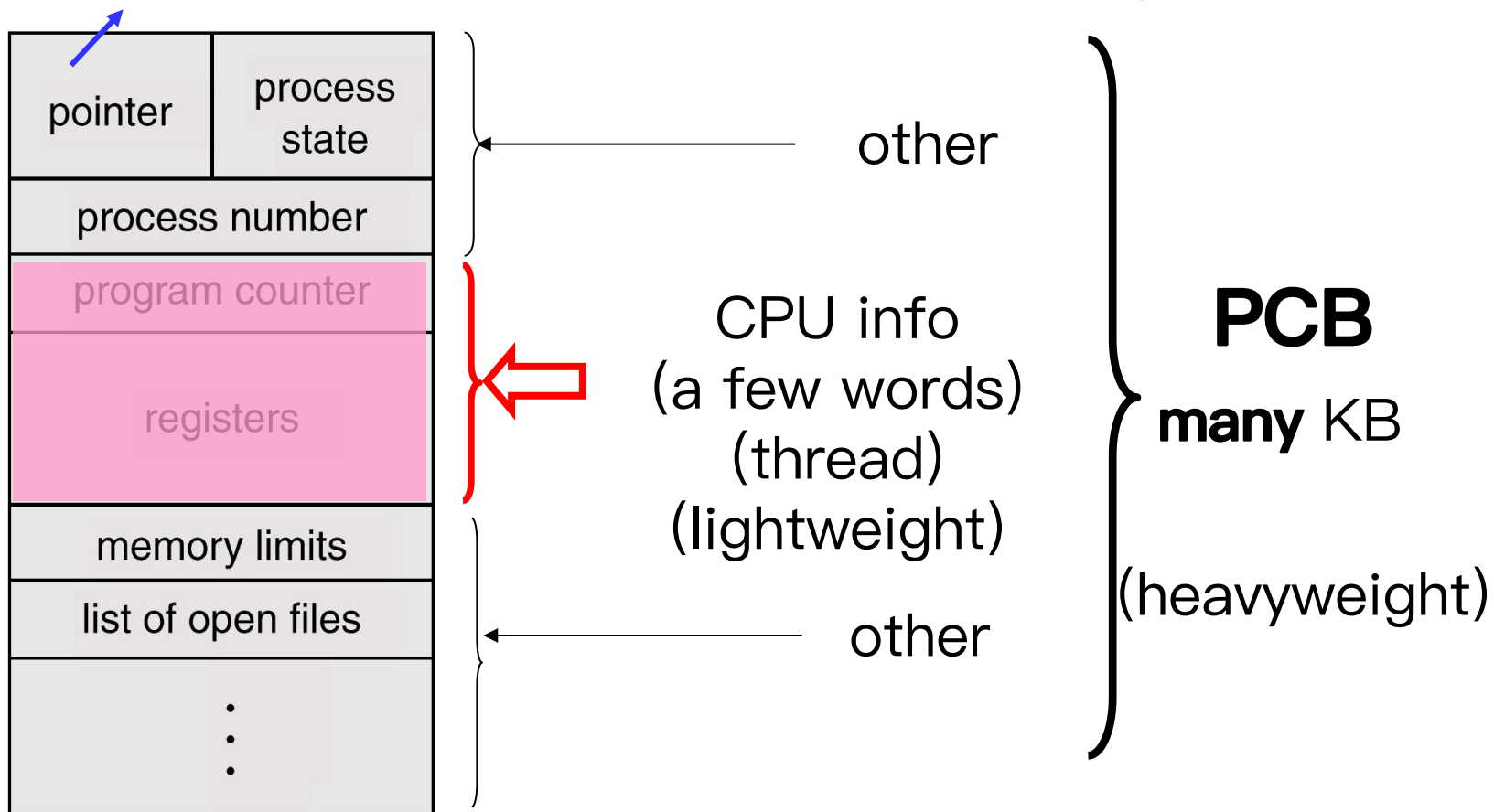
- 大部分现代应用软件都是多线程，线程是运行在应用进程里的、**比进程更小的运行单位**
- 一个应用程序大部分由多种任务来组成

例如，网页浏览软件

1. **客户端**：一个浏览器软件由显示内容、处理数据、拼写检查、网络处理等多个任务来组成。
2. **网页服务器程序**：由监听用户请求（注意，每个用户的请求都不一样）、负责处理用户请求的任务来组成

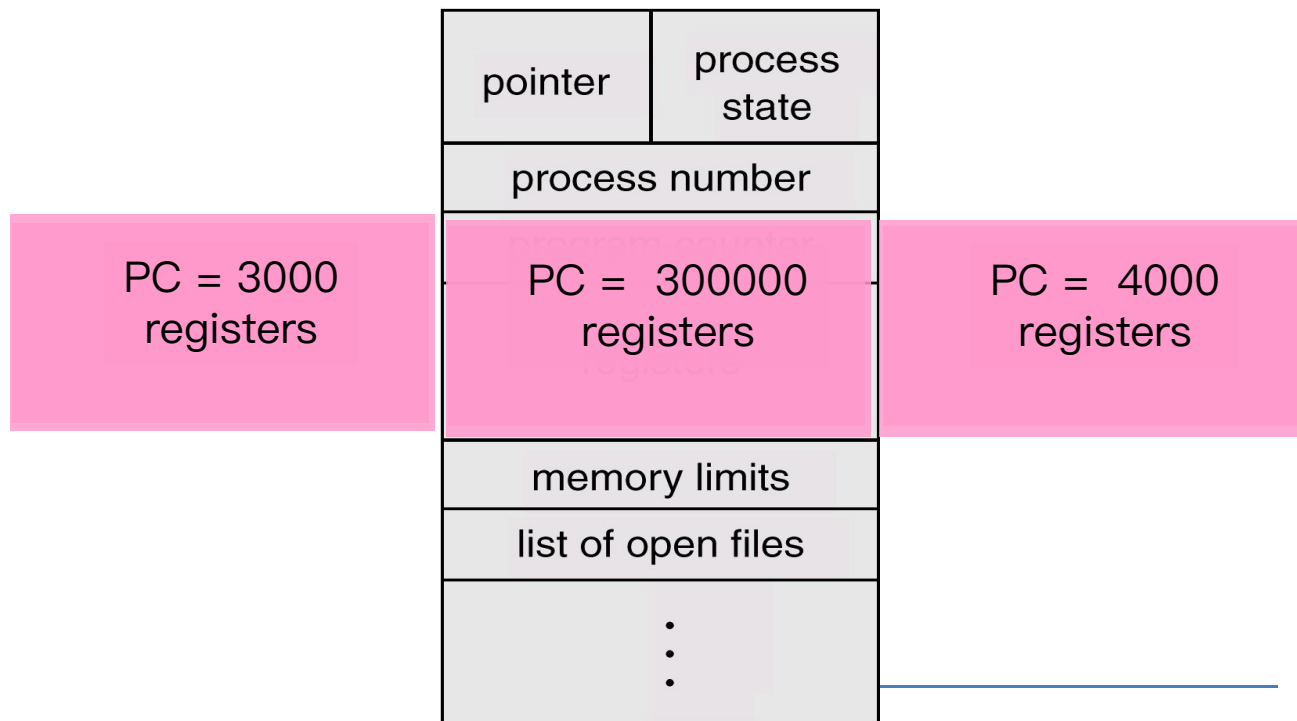


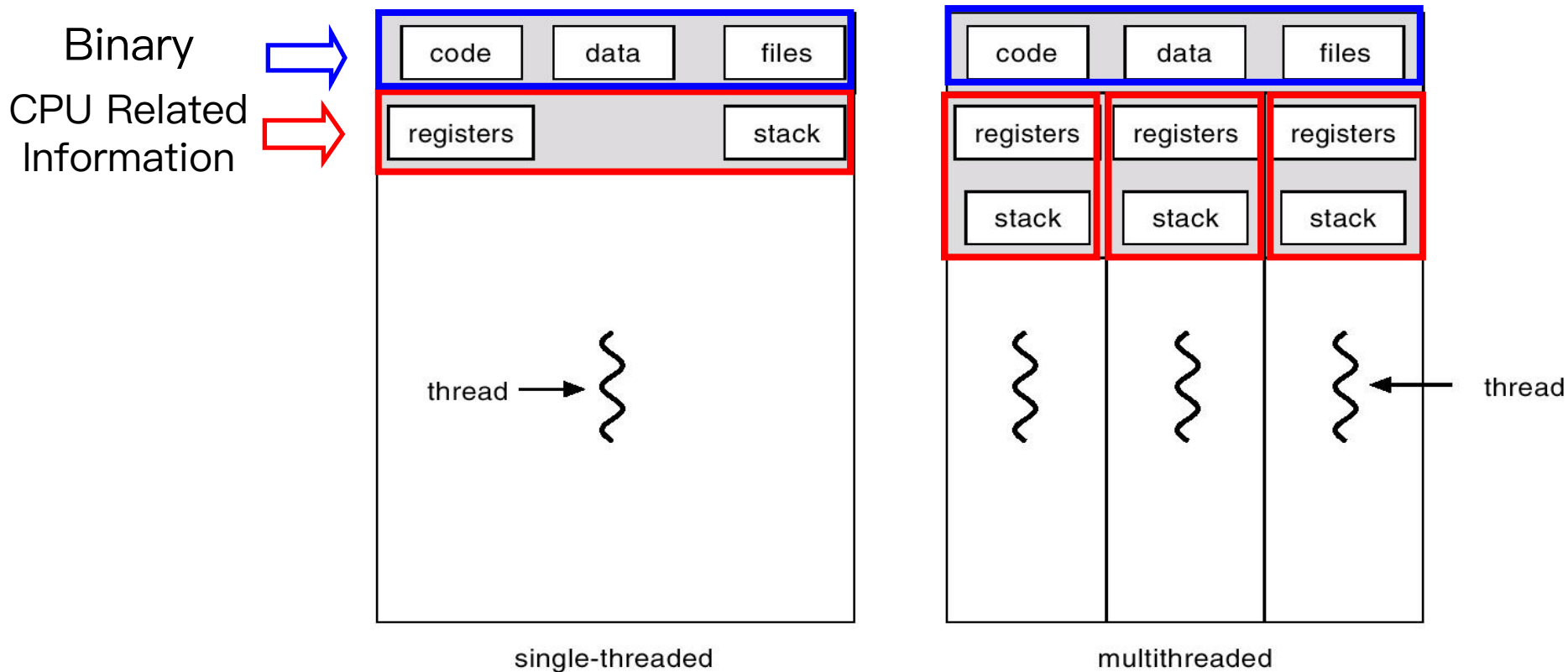
Let them all share same code in memory



一个进程可以拥有多个线程，而且线程之间共享以下内容：代码段，全局变量、打开的文件标识符、工作环境，包括当前目录，用户权限等

### One Process with Three Threads





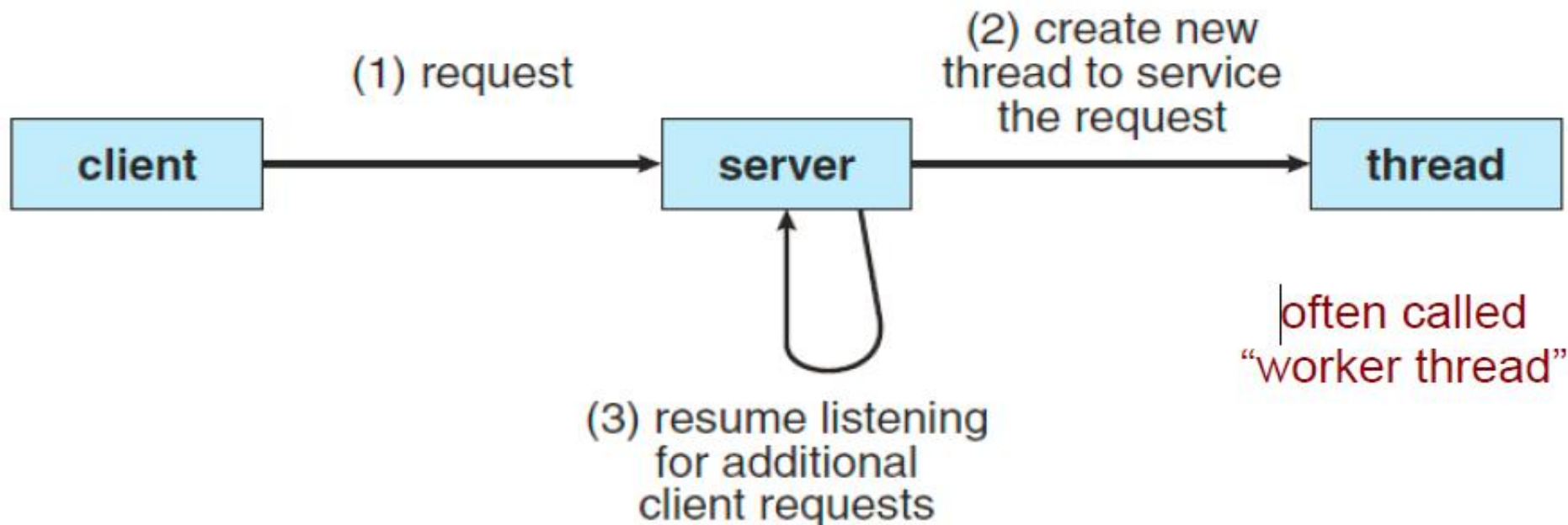
1. 每个线程是独立的调度对象

2. 一个进程可以拥有单个或多个线程，并共享内存空间

所以，**线程的运行是进程内部的执行轨迹**，是进程内部执行指令的跟踪

### 多线程编程优点

1. **响应速度快**：即使部分线程发生阻塞，其他线程可以继续运行
2. **资源共享**：共享所属进程的内存和资源
3. **经济**：创建和切换线程更经济
4. **多处理器体系结构的利用**：多线程可以在多处理器上并行运行



Instead of creating a new process to service a client's request, the OS creates a new thread within the server process, called worker thread. This incurs much less overhead than in the case of creating a new process. We may craft a group of threads in advance so that one of them may be assigned to an incoming request upon their arrival. This approach will be discussed in the section on "thread pools".

## 第二节、多线程模型

- 之前，我们学到是操作系统服务的对象，也就是调度的对象是进程。
- 现在，我们需要考虑调度的对象是线程，需要考虑怎么去服务线程。
- 必然，操作系统的服务与线程之间存在某种关系。
- 这个关系我们叫做关系模型。

## 2. 多线程模型

1. 一对一模型(One-to-One Model)
2. 多对一模型(Many-to-One Model)
3. 多对多模型(Many-to-Many Model)
4. 二级模型(Two-Level Model)

sometimes are called hybrid model

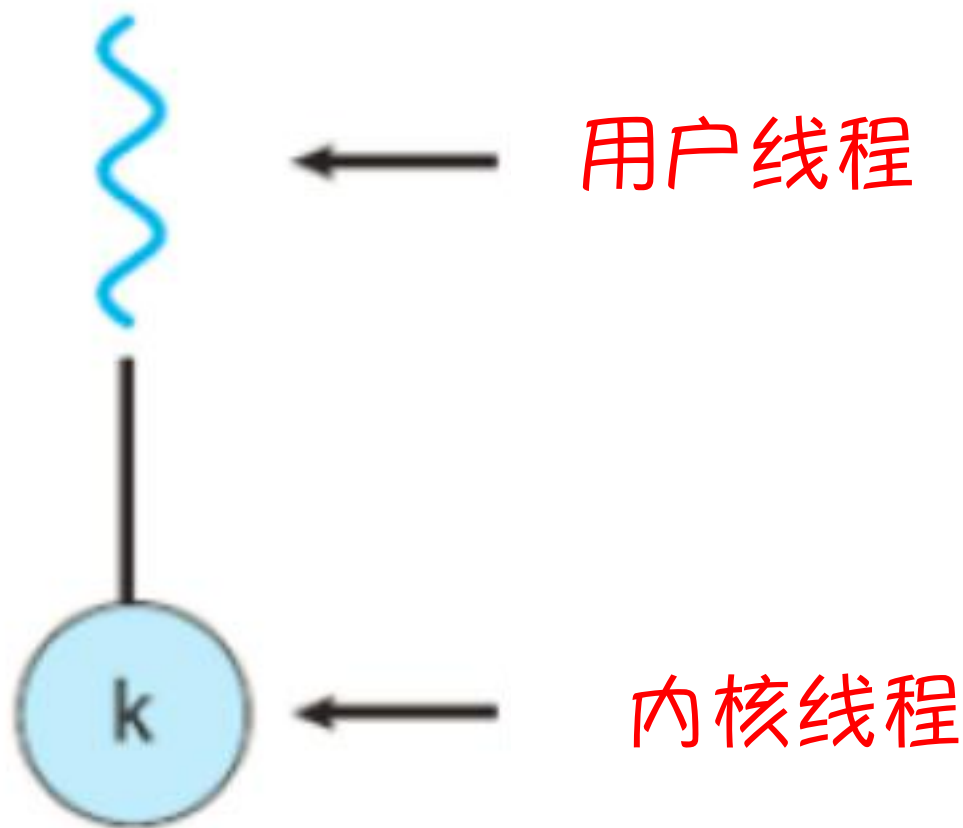


### 1. 用户线程

- I. 用户线程受内核的支持
- II. 用户线程是由用户层的线程库来管理，无需内核管理
- III. 当前主要线程库有  
(1)POSIX Pthreads, (2)Win32 Threads, (3)Java Threads

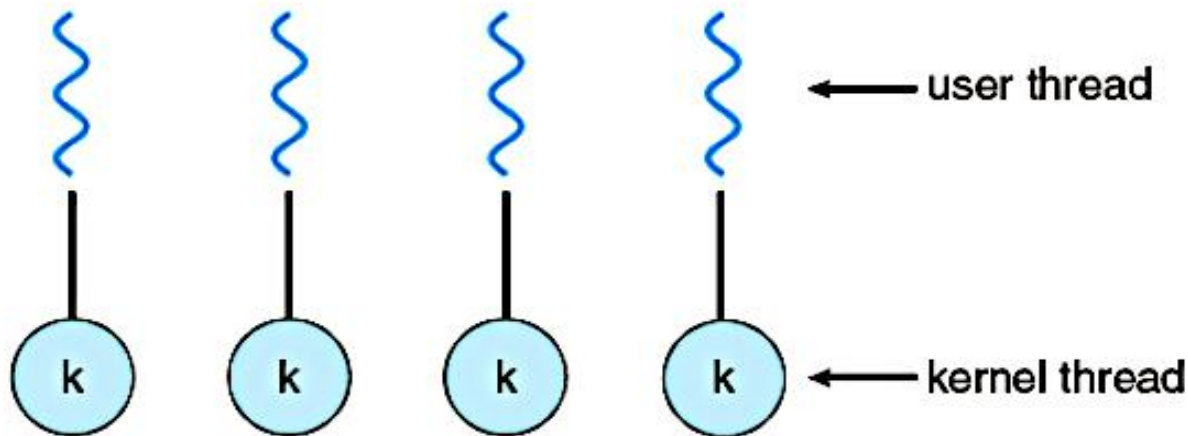
### 2. 内核线程

- I. 内核线程是由内核来管理
- II. 有内核来进行维护和调度
- III. 当前通用的操作系统，如 Windows, Solaris, Linux, Tru64 UNIX, Mac OS X 等都是支持内核线程



- 用户线程和内核线程之间必然会存在的一种关系。有一对一、多对一、多对多关系，建立起来的这种关系我们称为映射。
- 讨论它们之间关系之前，我们先做以下假设
  1. 操作系统支持内核线程
  2. 把内核线程看成是一个进程
  3. 用户线程基于内核线程运行，即用户线程阻塞内核阻塞、内核线程阻塞用户线程也阻塞。

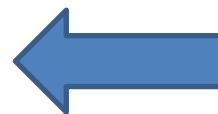
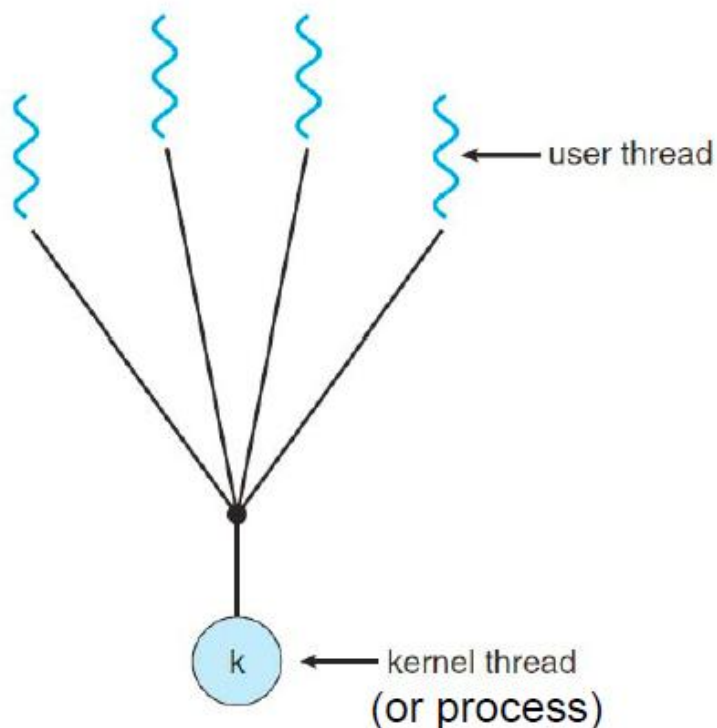
\*被阻塞的线程会进入到等待状态



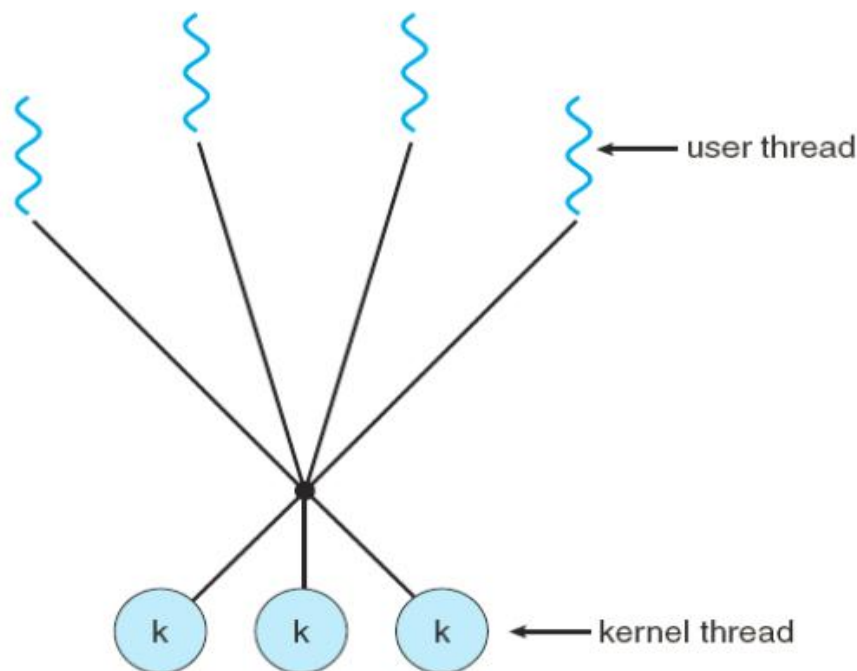
一个用户线程映射到一个内核线程的模型，Linux，Windows  
**优点**

1. 可以提供并发功能， 可以在多个处理器上并行执行
2. 在一个线程执行阻塞系统调用时， 可以调度另一个线程继续执行

- 多个用户线程映射一个内核线程
- 线程由用户空间管理



**多对一模型**

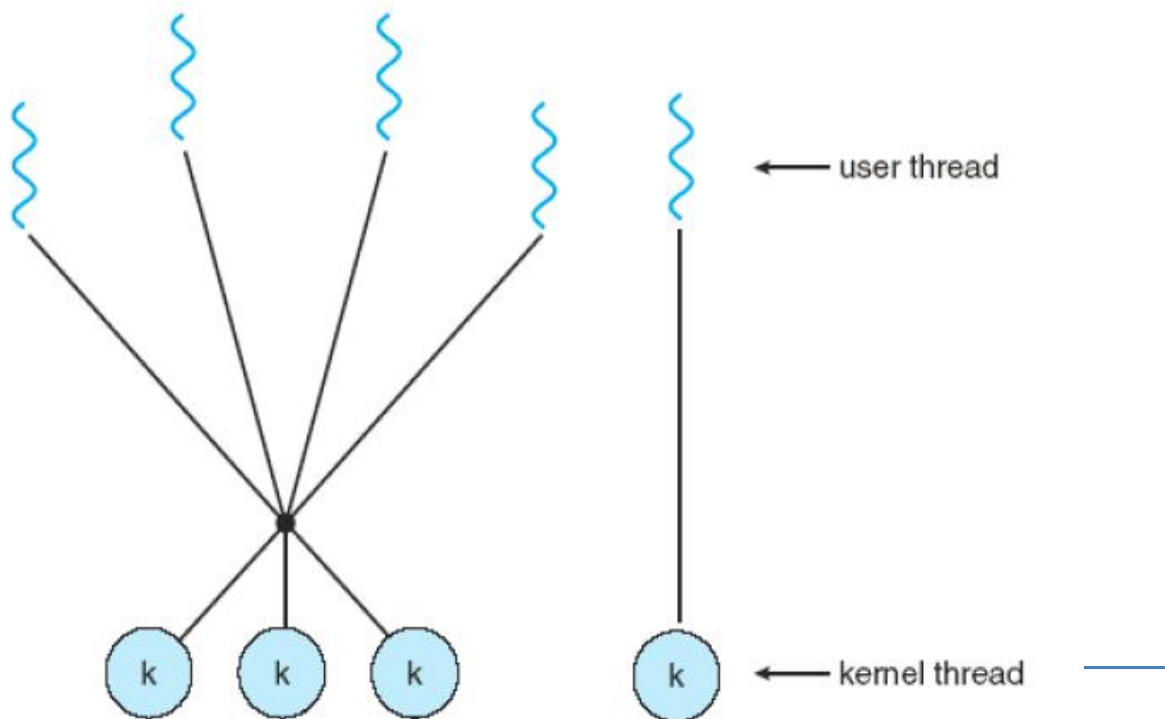


← 多对多模型

多个用户线程映射多个内核线程

1. 一对一模型的唯一的缺点是创建一个用户线程就需要创建一个内核线程，造成内核负担重
2. 多对一模型的缺点是一个用户线程一旦被阻塞，其他用户线程也会阻塞，即针对同一个内核线程服务的用户线程无法提供并发功能
3. 多对多模型没有以上缺点，当一个线程执行阻塞系统调用时，内核能调度另一个线程的执行，但需要提供调度机制

- 多对多模型的变种，即允许用户线程绑定一个特定的内核线程
- Examples: IRIX, HP-UX, Tru64 UNIX, Solaris 8 and earlier





## 第三节、线程库

为程序员提供的、创建和管理线程的 API，主要有以下两种方法来实现

### 1. 非嵌入到内核的方式：

是在用户空间中提供一个没有内核支持的库，此库的所有代码和数据结构都存在于用户空间中

### 2. 嵌入到内核的方式

是操作系统直接支持的内核库，此时所有代码和数据结构都存在于内核中

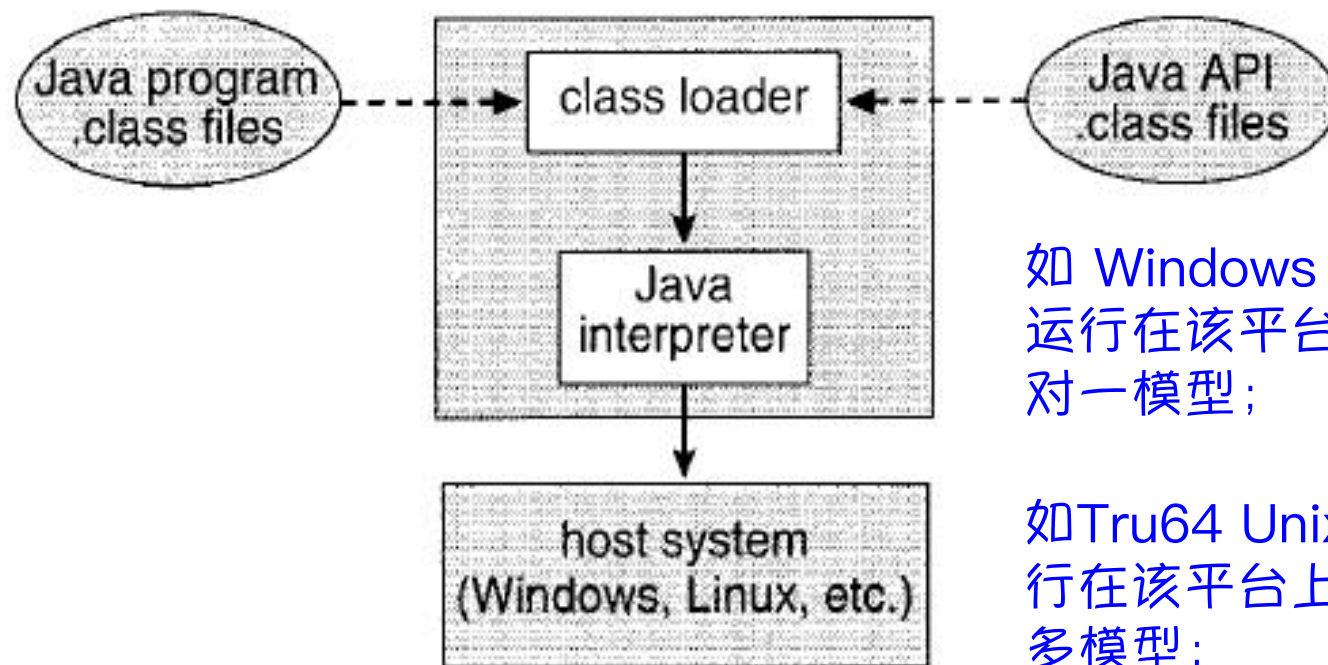
---

宿主操作系统：host operating system

目前，常使用的三种主要线程库有，

1. POSIX Pthread: 提供用户级或内核级的库
2. Win32: 内核级库，嵌入到内核的方式
3. Java: Java 线程 API 通常采用宿主系统上的线程库来实现的

- JVM (Java Virtual Machine)一般在操作系统之上实现，并隐藏了基本的操作系统实现细节
- 从而提供了一种一致的、抽象的环境以允许Java程序能在任何支持JVM的平台上运行
- JVM的规范没有指明Java线程如何被映射到底层的操作系统，而是让特定的JVM实现来决定



如 Windows XP是一对一模型，运行在该平台上的JVM就支持一对一模型；

如Tru64 Unix是多对多模型，运行在该平台上的JVM就支持多对多模型；

Figure 2.17 The Java virtual machine.

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

```
int pthread_create( pthread_t *tidp,    //thread ID point  
                  const pthread_attr_t *attr, //set attribute  
                  (void*)(*start_rtn)(void*), //thread  
                  void *arg);          // thread parameter
```

- 创建线程系统调用。
- 若线程创建成功，则返回0。若线程创建失败，则返回出错编号

```
void pthread_exit()
```

- 终止线程执行

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

```
int pthread_join( pthread_t  thread,  //thread ID  
                 void ** retval); // store thread
```

return value

- 等待一个线程结束
- 成功则返回零， 否则返回错误号

```
int pthread_attr_init( pthread_attr_t * attr );
```

- 初始化线程对象属性
- 成功则返回零， 否则返回错误号

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

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

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

```
void *runner(void *param); /* threads call this function */
```

```
int main(int argc, char *argv[])
```

```
{
```

```
    pthread_t tid; /* the thread identifier */
```

```
    pthread_attr_t attr; /* set of thread attributes */
```

```
    if (argc != 2) {
```

```
        fprintf(stderr, "usage: a.out <integer value>\n");
```

```
        return -1;
```

```
    }
```

```
    if (atoi(argv[1]) < 0) {
```

```
        fprintf(stderr, "%d must be >= 0\n", atoi(argv[1]));
```

```
        return -1;
```

```
    }
```

```
    /* get the default attributes */
```

```
    pthread_attr_init(&attr);
```

```
    /* create the thread */
```

```
    pthread_create(&tid, &attr, runner, argv[1]);
```

```
    /* wait for the thread to exit */
```

```
    pthread_join(tid, NULL);
```

```
    printf("sum = %d\n", sum);
```

```
}
```



```
/* The thread will begin control in this function */  
void *runner(void *param)  
{  
    int i, upper = atoi(param);  
    sum = 0;  
  
    for (i = 1; i <= upper; i++)  
        sum += i;  
  
    pthread_exit(0);  
}
```

Two threads are created:  
1. main() 2. runner()

Figure 4.9 Multithreaded C program using the Pthreads API.

#### ■ 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.

- 线程库编译需要用 `-lpthread` 来链接线程库
- `$ gcc -o thread thread.c -lpthread`

hbpark@hbpark-VirtualBox: ~/src/thread

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

int sum;
void * runner01(void * param); // thread 1
void * runner02(void * param); // thread 2

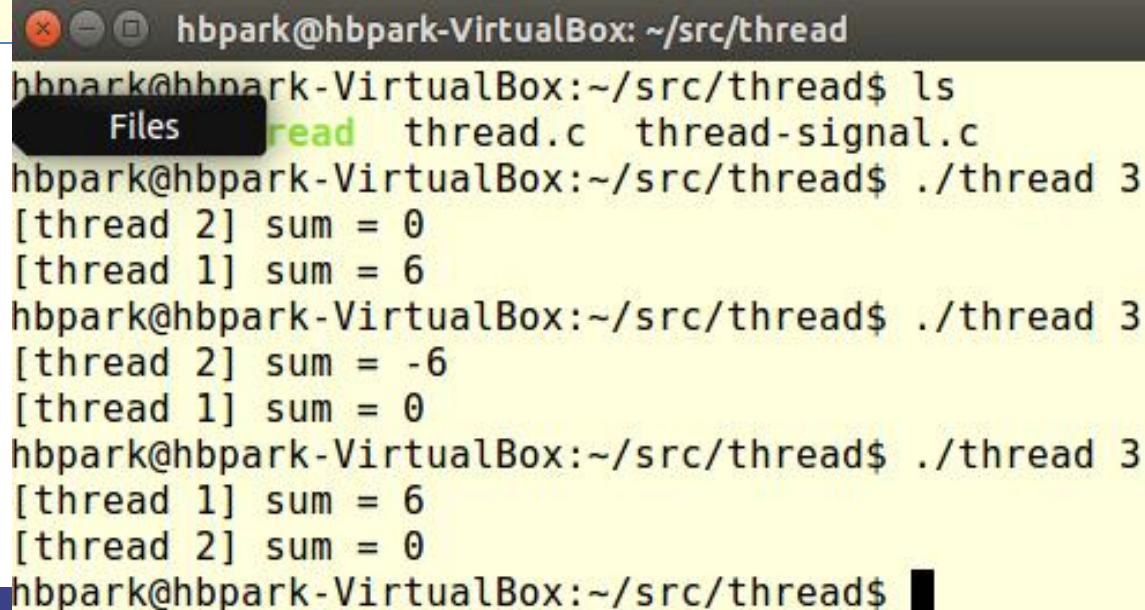
int main (int argc, char * argv[])
{
    sum = 0;
    pthread_t tid01;
    pthread_t tid02;
    pthread_attr_t attr;
    if (argc != 2) {
        fprintf(stderr, "usage: a.out <integer value>\n");
        return -1;
    }
    if (atoi(argv[1]) <= 0){
        fprintf(stderr, "%d must be > 0\n", atoi(argv[1]));
        return -1;
    }
    pthread_attr_init(&attr); // get the default attributes
    pthread_create(&tid01, &attr, runner01, argv[1]);
    pthread_create(&tid02, &attr, runner02, argv[1]);
    pthread_join(tid01, NULL);
    pthread_join(tid02, NULL);

    pthread_exit(0);
}
```

```
void * runner01(void * param)
{
    int i, upper = atoi(param);
    for ( i = 1; i <= upper; i ++ )
        sum += i;
    printf("[thread 1] sum = %d\n", sum);
}
```

```
void * runner02(void * param)
{
    int i, lower = atoi(param);
    for (i = 1; i <= lower; i++)
        sum -= i;
    printf("[thread 2] sum = %d\n", sum);
}
```

## 运行结果



```
hbpark@hbpark-VirtualBox: ~/src/thread
hbpark@hbpark-VirtualBox:~/src/thread$ ls
Files  thread.c  thread-signal.c
hbpark@hbpark-VirtualBox:~/src/thread$ ./thread 3
[thread 2] sum = 0
[thread 1] sum = 6
hbpark@hbpark-VirtualBox:~/src/thread$ ./thread 3
[thread 2] sum = -6
[thread 1] sum = 0
hbpark@hbpark-VirtualBox:~/src/thread$ ./thread 3
[thread 1] sum = 6
[thread 2] sum = 0
hbpark@hbpark-VirtualBox:~/src/thread$
```



- 在上一页代码的主线程和其他线程里加无限循环代码，并运行
- 打开新的 terminal，用 `$ps -ef | grep thread` 命令来查看进程 thread 的PID
- 再用，`$ps -T -p PID` 命令来查看进程内的线程数
- 或者 可以查看 `/proc/pid/status`

```
hbpark@hbpark-VirtualBox: /proc/3764/task$ ps -T -p 3764
```

PID	SPID	TTY	TIME	CMD
3764	3764	pts/1	00:00:00	thread
3764	3765	pts/1	00:02:32	thread
3764	3766	pts/1	00:02:32	thread

### 实验

- 用shell script, 把程序运行1000次,并查看运行结果
- 每次运行结果可能不同, 请把程序改成让线程runner01 先运行结束后, 才能让runner02在运行

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 0; i <= Upper; i++)
        Sum += i;
    return 0;
}

int main(int argc, char *argv[])
{
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;

    if (argc != 2) {
        fprintf(stderr, "An integer parameter is required\n");
        return -1;
    }
    Param = atoi(argv[1]);
    if (Param < 0) {
        fprintf(stderr, "An integer >= 0 is required\n");
        return -1;
    }
}
```

```
/* create the thread */
ThreadHandle = CreateThread(
    NULL, /* default security attributes */
    0, /* default stack size */
    Summation, /* thread function */
    &Param, /* parameter to thread function */
    0, /* default creation flags */
    &ThreadId); /* returns the thread identifier */

if (ThreadHandle != NULL) {
    /* now wait for the thread to finish */
    WaitForSingleObject(ThreadHandle, INFINITE);

    /* close the thread handle */
    CloseHandle(ThreadHandle);

    printf("sum = %d\n", Sum);
}
}
```

Figure 4.11 Multithreaded C program using the Windows API.

## 第四节、多线程程序相关问题



1. 系统调用 `fork()` 和 `exec()` 语义
2. 线程取消
3. 信号处理
4. 线程池
5. 调度程序激活(Scheduler Activation)

- 如果进程中的某一个线程调用 fork(), 那么新进程会复制所有线程呢, 还是只复制调用 fork() 的线程呢?

答: 有的 Unix 系统提供两种形式的 fork() 系统调用, 开发者可以决定只复制一个或者复制全部

- 那么, 什么时候复制全部, 什么时候复制一个好呢?

答: 这取决于创建新进程的目的。请考虑 exec() 系统调用的语义, exec() 语义是用参数所指定的程序替换整个进程。

#### Parent Process

Thread 0:  
printf("I am thread 0\n");  
fork();

Thread 1:  
printf("I am thread 1\n");

Thread 2:  
printf("I am thread 2\n");

Thread 3:  
printf("I am thread 3\n");

#### Child Process

Thread 0:  
printf("I am thread 0\n");  
fork();

只复制调用fork()的线程

### Parent Process

Thread 0:  
printf("I am thread 0\n");  
fork();

Thread 1:  
printf("I am thread 1\n");

Thread 2:  
printf("I am thread 2\n");

Thread 3:  
printf("I am thread 3\n");

### Child Process

```
// Using exec() execute the  
following code.  
void main ()  
{  
    printf("my name is HIT\n");  
}
```

只复制调用fork()的线程

## 结论

1. 如调用exec(), 就复制一个
2. 如不调用exec(), 就复制全部

**目标线程：**线程完成任务之前终止线程的任务，需要取消的线程一般称为目标线程

- 目标线程的取消可能发生“要取消的线程正在更新与其他线程所共享的数据”，为此，提供以下两种取消方式：
  1. **异步取消（Asynchronous Cancellation）**：一个线程立即终止目标线程
  2. **延迟取消（Deferred Cancellation）**：目标线程不断地检查它是否应终止

线程取消请求的实际取消的行为依赖于线程的状态

Mode	State	Type
Off	Disabled	—
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

1. **Off Mode**: 如果系统设置为禁用线程取消的话，线程取消会待定直到可以取消线程为止
2. **Deferred Mode**: 默认状态，延迟取消
3. **Asynchronous Mode**: 异步模式，立即终止目标线程

**取消点 (Cancellation Point)**：当一个线程认定为可以安全取消时，可以安全取消的这个点称为取消点，即 `pthread_testcancel()` 会被调用。例如，在Linux系统中，线程取消是通过信号来处理的

- **信号(signal)**是用来通知进程某个特定事件的发生，这需要操作系统提供一种内核和进程之间的通信机制。信号有以下两种：
    - I. **同步信号(内部信号)**：进程本身事件产生的信号，如访问非法内存、除零等
    - II. **异步信号(外部信号)**：进程之外事件产生的信号，如按CTRL+C键等
  - 信号处理程序是用来处理发生的事件，处理过程如下：
    - 步骤一：由特定事件产生信号
    - 步骤二：这个信号传送给进程
    - 步骤三：信号处理程序处理相应信号
- 信号:signal
  - 信号处理程序:signal handler
  - 默认信号处理程序: default signal handler
  - 用户定义的信号处理程序:User-defined signal handler

信号处理程序有**默认信号处理程序**和**用户定义的信号处理程序**



事件产生信号，传送给进程需要考虑的问题？

情况一：对于单线程进程，No Problem

情况二：对于多线程进程，应该传送给进程的哪个线程？

- I. 发送信号到信号所应用的线程
- II. 发送信号到进程内每个线程
- III. 发送信号到进程某个固定线程
- IV. 规定一个特定的线程一接收进程的所有信号

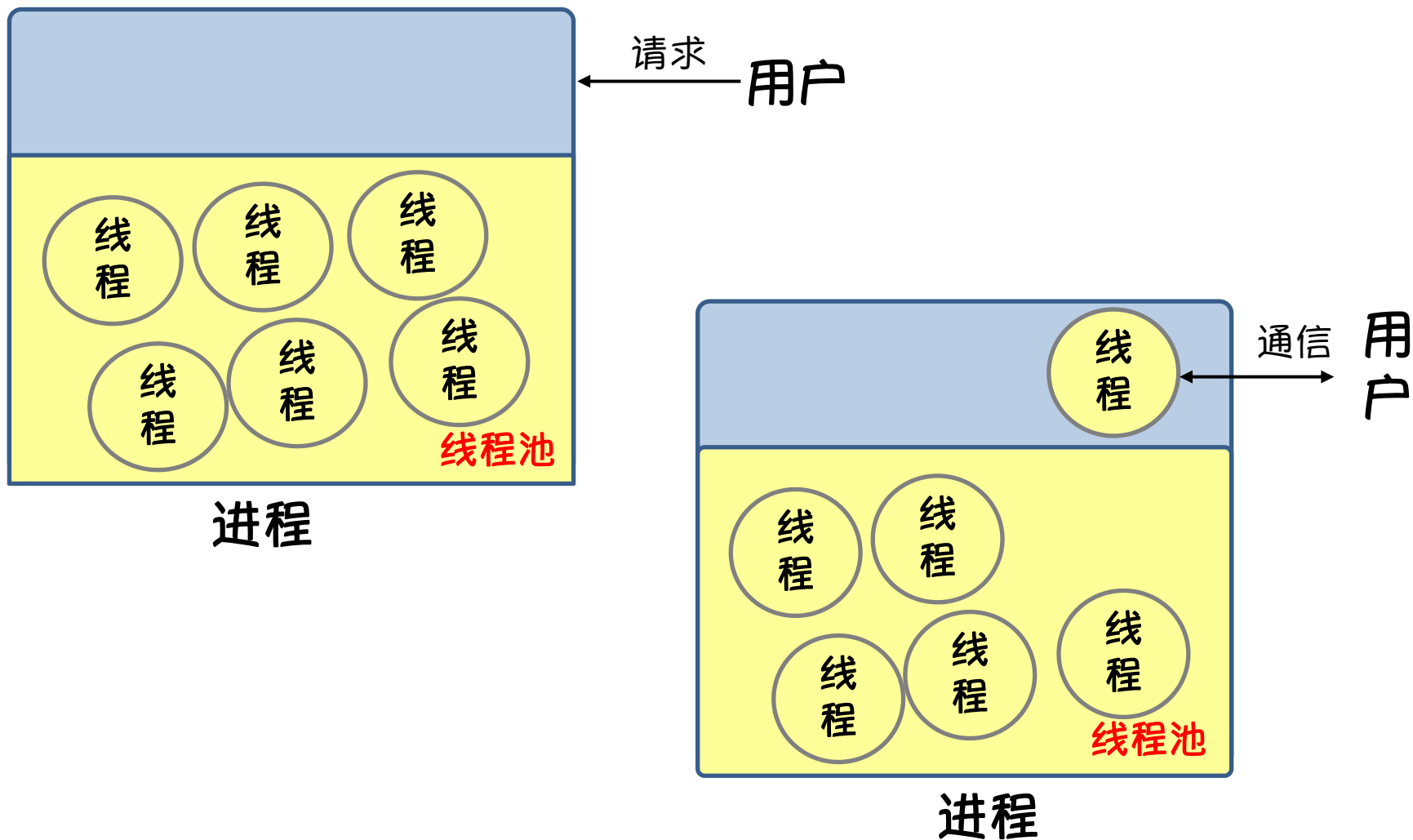
注：发送信号的方法依赖于产生信号的类型。

- 象 CTRL+C 产生的信号必须发送给进程内的每个线程。
- 象 Pthread 提供的 `pthread_kill(pthread_t tid, int signal)` 函数就可以把信号传送给指定的线程

**线程池 (thread pool)** 的主要思想是在进程开始时 (实现) 创建一定数量的线程, 并放入到进程池中等待工作。目的就是提高效率

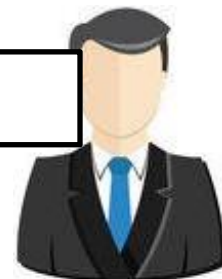
### 主要优点

1. 通常用现有线程处理请求要比等待创建新的线程要快
2. 线程池限制了在任何时候可用线程的数量, 这可以防止大量创建并发线程, 有助于管理





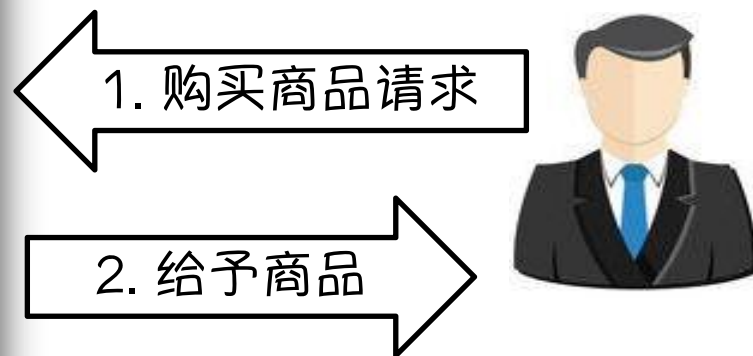
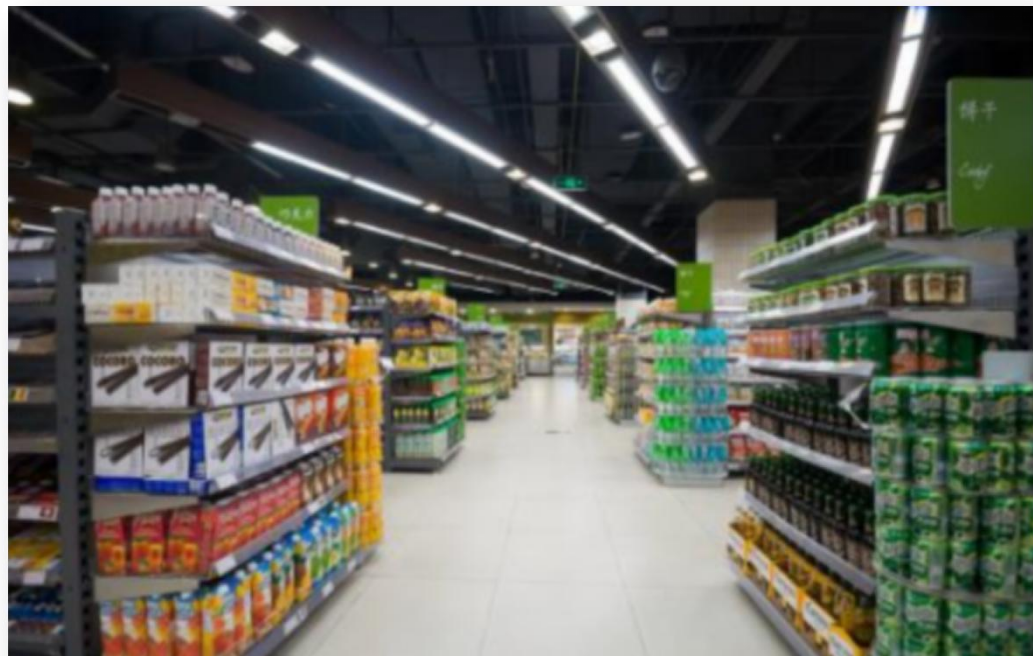
1. 购买商品请求



3. 给予商品

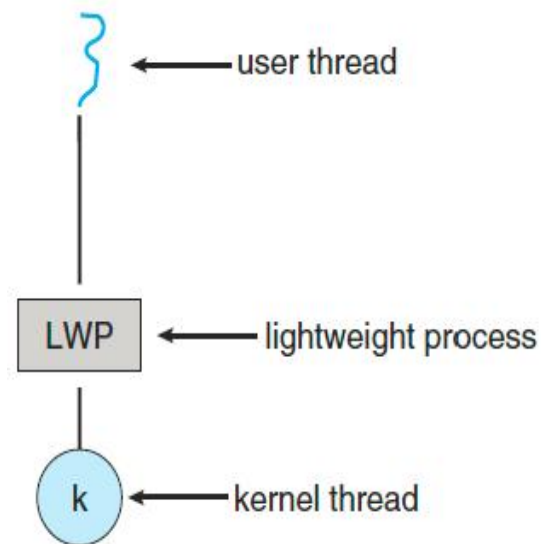
2. 进货





多对多（MM）模型和二层模型中的用户线程和内核线程需要通信，**以便维持内核线程的适当数量**

- I. 用户和内核线程之间设置一种中间数据结构，这种数据结构通常称为轻量级进程（LWP: Light Weight Process）
- II. 对于用户线程库，LWP表现为一种应用程序可以**调度用户线程的虚拟处理器**
- III. 每个LWP与内核线程相连，该内核线程被操作系统调度到物理处理器上运行
- IV. 如果内核线程阻塞，LWP也阻塞，相连用户线程也会阻塞





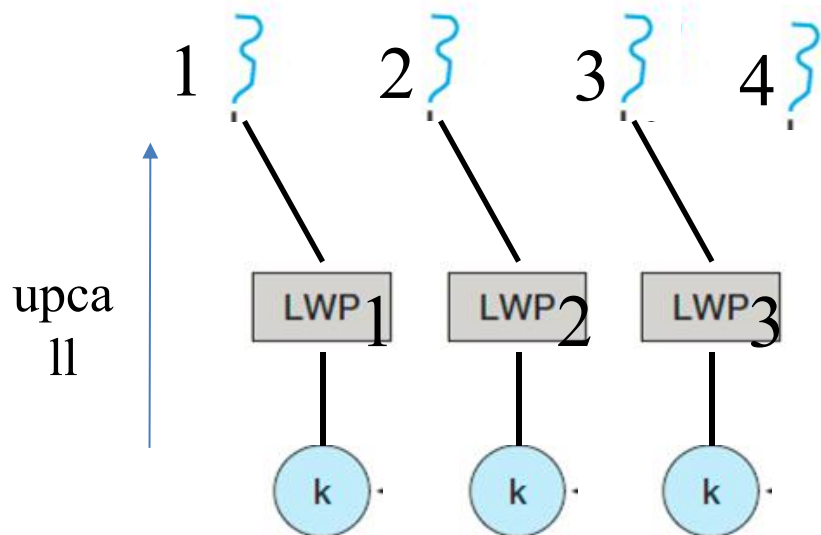
### 调度程序激活：一种解决用户线程库和内核之间通信的方法

- 内核提供一组LWP给应用程序，应用程序可调度用户线程到一个可用的LWP上
- **调度程序激活**提供内核和线程库之间的通信机制，这个通信机制称为 **UPCALL**.
- UPCALL 由具有 UPCALL 处理句柄的线程库处理，而且 UPCALL 处理句柄必须在虚拟处理器上运行
- 这种通信机制允许应用程序保持适当的内核线程数量



### 调度程序激活工作方式及流程

① Thread 1 is blocked

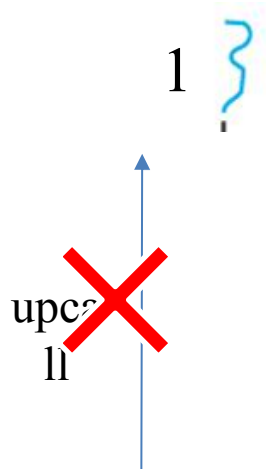


② Kernel informs the application about the blocking of the user thread 1 by upcall

③ Kernel allocates a new LWP

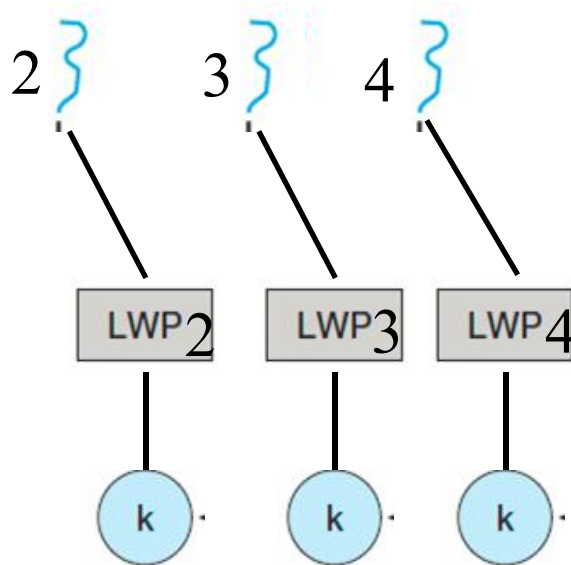
- ④ Upcall handler
1. Identify the blocking thread
  2. Save the state of the blocking thread
  3. Schedule another thread

① Thread 1 is blocked



② Kernel informs the application about the blocking of the user thread 1 by upcall

③ Kernel allocates a new LWP4



④ Upcall handler

1. Identify the blocking thread
2. Save the state of the blocking thread
3. Schedule another thread

# Q&A