# Linux——管道

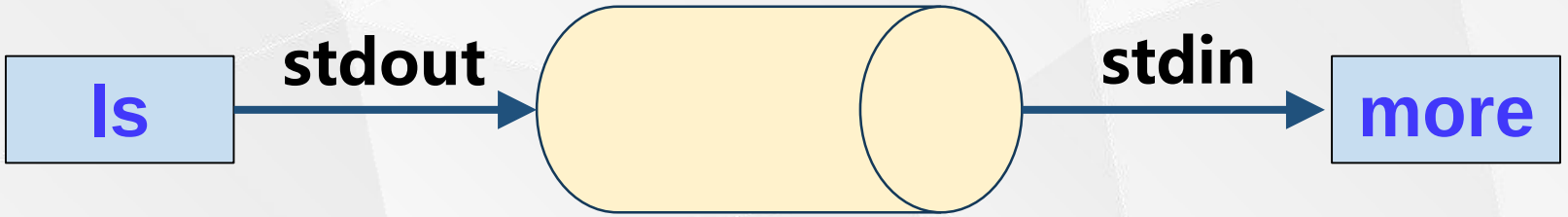


- 管道是进程间基于文件的一种间接通信机制。
- 管道是半双工的，数据只能单向流动。
- 管道对于管道两端的进程而言就是一个文件，写管道时内容添加在文件末尾，读管道则从文件头部读出。
- **无名管道**
  - 无名管道没有名字，只有一组文件描述
  - 子进程从父进程继承文件描述符
  - 只能用于具有亲缘关系的进程间的通信
- **命名管道**
  - 管道有文件名，允许无亲缘关系的进程通过访问管道文件进行通信。

# 管道示例



**$ ls | more**



# 与管道相关的系统调用



- **创建无名管道**：**int pipefd[2]; int pipe(pipefd);**
  - pipefd[0]只能用于读，接收进程应关闭写入端，即close(pipefd[1])。
  - pipefd[1]只能用于写，发送进程应关闭读取端，即close(pipefd[0]) 。
- **将数据写入管道**：**write()**
  - 管道长度受到限制，管道满时写操作将被阻塞，直到管道中的数据被读取。
  - fcntl()可将管道写模式设置为非阻塞模式。
- **从管道读取数据**：**read()**
  - 数据被读取后将自动从管道中清除，若管道中没有数据则读操作将被阻塞。
  - fcntl()可将管道读模式设置为非阻塞模式。
- **关闭管道**：**close()**
  - 所有读端口都关闭时，在管道上进行写操作的进程将收到SIGPIPE信号。
  - 所有写端口都关闭时，在管道上进行读操作的read()函数将返回0

# 与管道相关的系统调用

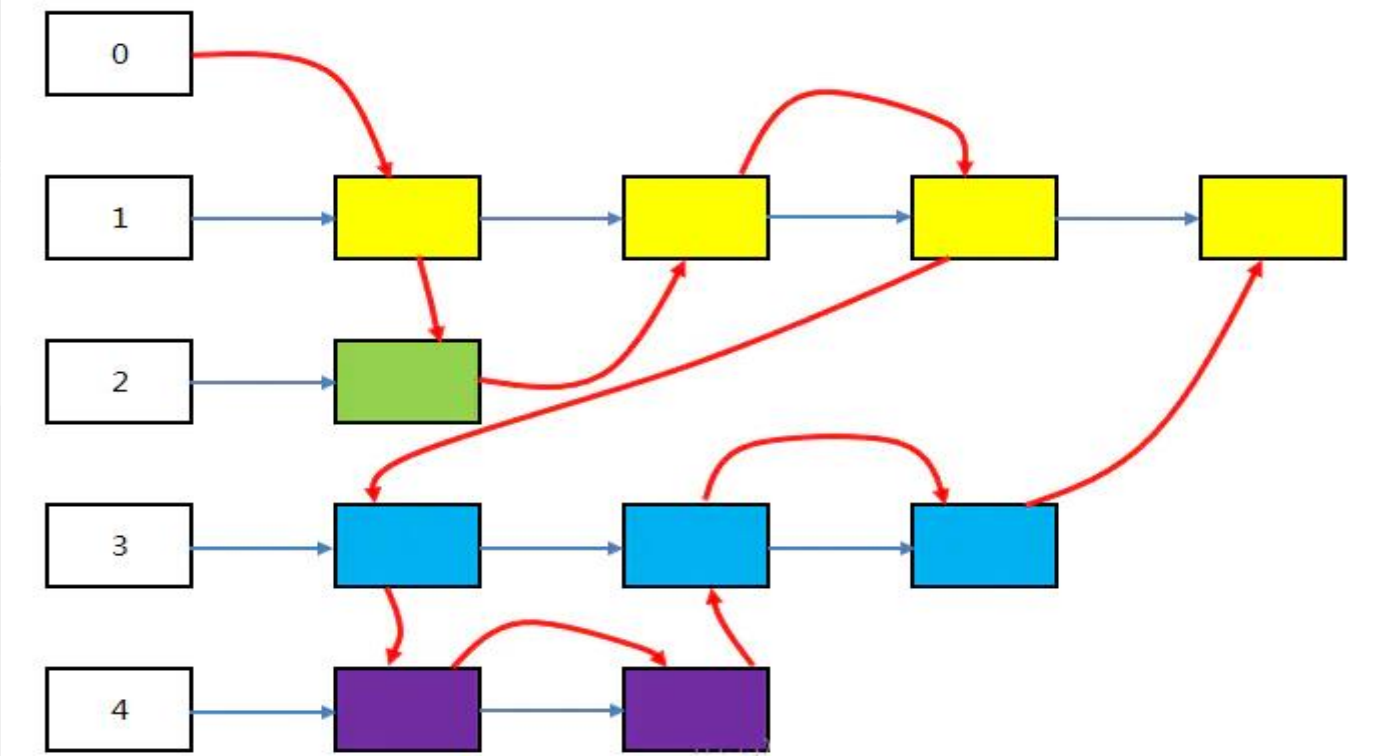


- **创建命名管道**
  - **int mknod(char *pathname, mode_t mod, dev_t dev);**
  - **int mkfifo(char *pathname, mode_t mode);**
- **命名管道的使用**
  - 命名管道必须先调用open()将其打开，其他与无名管道相似
  - **pipe_read = open("./my_pipe", O_RDONLY, 0);**
  - **pipe_write = open("./my_pipe", O_WRONLY, 0);**
  - 以只读方式(O_RDONLY)打开时，进程将被阻塞直到有写方打开管道
  - 以只写方式(O_WRONLY)打开时，进程将被阻塞直到有读方打开管道
- **命名管道的删除**
  - **int unlink(char *pathname);**

# Linux——消息队列



- 消息队列是消息的链接表，存放在内核中并由消息队列标识符标识。
- 每一条消息都有自己的消息类型，每种类型的消息都被对应的链表所维护。



# 消息队列的创建



**int msgget(key_t key, int msgflg);**

① key：键值，多个需要使用同一消息队列的进程用相同的key来创建或获取该消息队列。

② msgflg：flag|mode

- flag：可以是0或者IPC_CREAT(不存在就创建)
- mode：同文件权限一样

# 消息的发送



**int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);**

① msqid：消息队列ID。

② msgp：存放消息的结构体。

```
struct msgbuf {
    long mtype;          // 消息的类型，必须>0
    char mtext[1];       // 消息内容，长度可变
};
```

③ msgsz：要发送的消息的大小，不包括消息的类型占用的4个字节。

④ msgflg：如果是0，当消息队列为满 msgsnd会阻塞；如果是IPC_NOWAIT，当消息队列为满时不阻塞，立即返回。

# 消息的接收



**int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);**

① msqid：消息队列ID。

② msgp：存放消息的结构体。

③ msgsz：要接收的消息的大小，不包括消息的类型占用的4个字节。

④ msgtyp：要接收的消息类型。

- =0：返回消息队列中的第一条消息
- >0：返回消息队列中类型等于msgtyp的第一条消息
- <0：返回其类型小于或等于msgtyp绝对值的最小的一个消息

⑤ msgflg：如果是0，则没有指定类型的消息时会阻塞；如果是IPC_NOWAIT，则不阻塞，立即返回。

# 消息队列的删除



**int msgctl(int msqid, int cmd, struct msqid_ds *buf);**

① msqid：消息队列ID。

② cmd：控制命令， IPC_RMID表示删除。

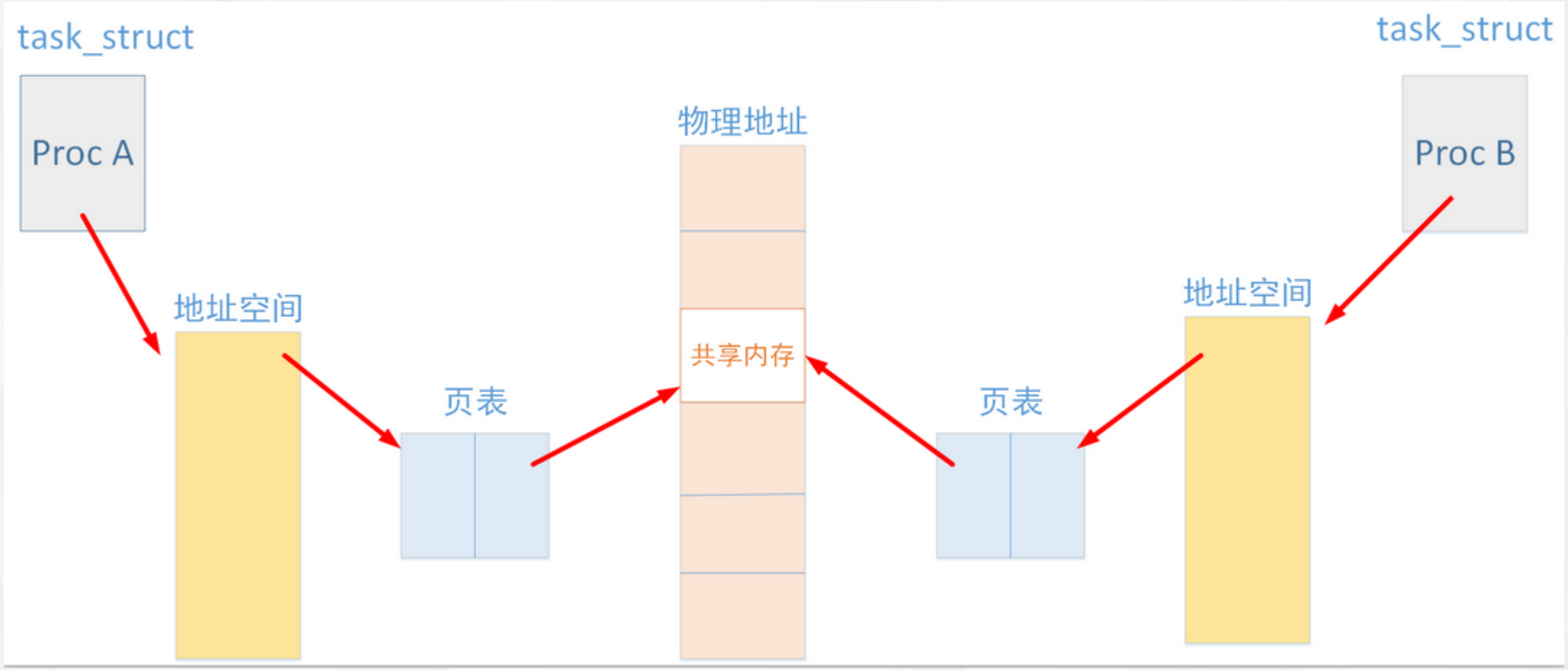
例如：

**msgctl(msqid, IPC_RMID, NULL); //删除ID为msqid的消息队列**

# Linux——共享内存



共享内存是把同一片物理内存区域同时映射到多个进程的地址空间的通信机制。



# 共享内存的创建



**int shmget(key_t key,int size,int shmflg)**

① key：键值，多个需要使用同一共享内存的进程用相同的key来创建或获取该共享内存。

② size：共享内存的大小。

③ shmflg: IPC_CREAT | 0666

# 共享内存绑定



**void *shmat ( int shmid, char *shmaddr, int shmflg)**

① shmid：共享内存句柄，shmget调用的返回值；

② shmaddr：一般用NULL;

③ shmflg：SHM_R | SHM_W

**例：**

**S = (char *)shmat(shmid1,NULL,SHM_R|SHM_W)**

绑定后，对共享内存的操作即转化为对局部变量S的操作。

# 共享内存的释放



**int shmctl(int shmid, int cmd, struct shmid_ds *buf);**

① shmid：共享内存句柄，即shmget函数的返回值。

② cmd：控制命令。

③ buf：一个结构指针，它指向共享内存模式和访问权限的结构。

**例：**

**shmctl(shmid,IPC_RMID,0)**

## 信号量的创建

**int semget(key_t key, int num_sems, int sem_flags)**

① Key：用来允许多个进程访问相同信号量的整数值，它们通过相同的key值来调用semget。

② num_sems：所需要的信号量数目。Semget创建的是一个信号量数组，数组元素的个数即为num_sems。

③ sem_flags：一个标记集合，与open函数的标记十分类似。低九位是信号的权限，其作用与文件权限类似。另外，这些标记可以与 IPC_CREAT进行或操作来创建新的信号量。一般用：IPC_CREAT | 0666

# 信号量的控制



**int semctl(int sem_id, int sem_num, int command, ...)**

① sem_id：是由semget所获得的信号量标识符。

② sem_num：是信号量数组元素的下标，即指定对第几个信号量进行控制。

③ command：要执行的动作，有多个不同的command值可以用于semctl。常用的两个command值为:

- SETVAL：用于为信号量赋初值，其值通过第四个参数指定。
- IPC_RMID：当信号量不再需要时用于删除一个信号量标识。

④ 如有第四个参数，则是union semun，该联合定义如下：

```
union semun {
    int val;
    struct semid_ds *buf;
    unsigned short *array; }
```

# 信号量的操作



**int semop(int sem_id, struct sembuf *sem_ops, size_t num_sem_ops)**

① sem_id：由semget函数所返回的信号量标识符。

② sem_ops：一个指向结构数组的指针，该结构定义如下：

```
struct sembuf {
        short sem_num;    //数组下标
        short sem_op;     //操作，-1或+1
        short sem_flg;    //0
}
```

③ num_sem_ops：操作次数，一般为1

# P操作



```
void P(int semid,int index)
{     struct sembuf sem;
        sem.sem_num = index;
        sem.sem_op = -1;
        sem.sem_flg = 0;              //操作标记：0或IPC_NOWAIT等
        semop(semid,&sem,1);          //1:表示执行命令的个数
        return;
}
```

# V操作



```
void V(int semid,int index)
  {   struct sembuf sem;
        sem.sem_num = index;
        sem.sem_op =  1;
        sem.sem_flg = 0;
        semop(semid,&sem,1);
        return;
}
```

# 线程

# 一个例子



**Word软件**

- 用户交互——接收用户的输入（文档内容或处理命令）
- 格式处理——根据用户的输入调整文档的显示格式
- 自动存盘——每隔一段时间自动将文档存盘

**解决方案：**

- 单进程：任务无法并行处理，用户感受到交互延迟
- 多进程：三个进程处理的是同一个文档，但是地址空间相互隔离。进程间相互通信和共享数据困难，切换开销大。
- 多线程：在进程内部创建多个线程，每个线程完成一个任务。

# 传统的进程模型



- **进程是资源占用的基本单位**：进程拥有主存、I/O通道、I/O设备、文件等系统资源的使用权。
- **进程是调度/执行的基本单位**：操作系统以进程为单位进行处理机的调度。
- 不足：
  - 进程创建开销大
  - 进程切换开销大
  - 进程通信代价大
  - 进程之间的并发性粒度较粗，并发度不高
  - 不适合客户/服务器计算的要求

# 多线程方案的解决思路



- 把进程的两项功能——“独立分配资源”与“被调度执行”分离开来
- 进程作为系统资源分配和保护的独立单位
- 在进程内增加一类实体——**线程**
- 线程作为调度的基本单位
- 同一进程内的线程共享相同的地址空间

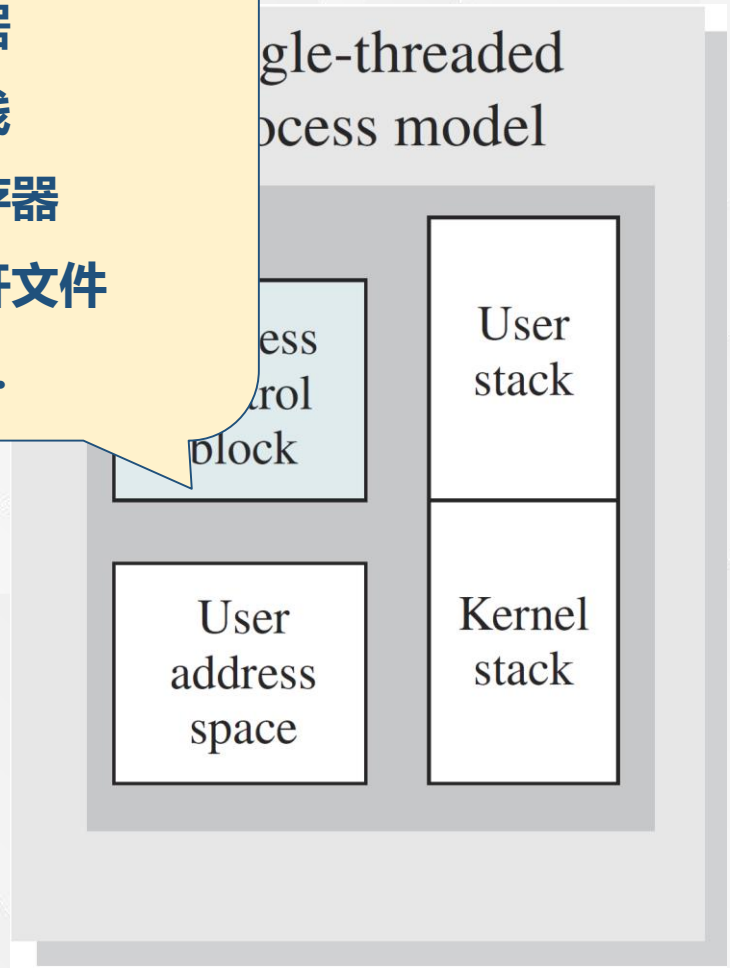
# 线程的定义



- 线程是进程中的一条执行路径；
- 每个线程有自己私用的堆栈和处理机执行环境；
- 单个进程可创建多个同时存在的线程；
- 同一进程中的多个线程共享分配给该进程的内存。

# 单线程模型和多线程模型

- 代码
- 数据
- 堆栈
- 寄存器
- 打开文件
- ……

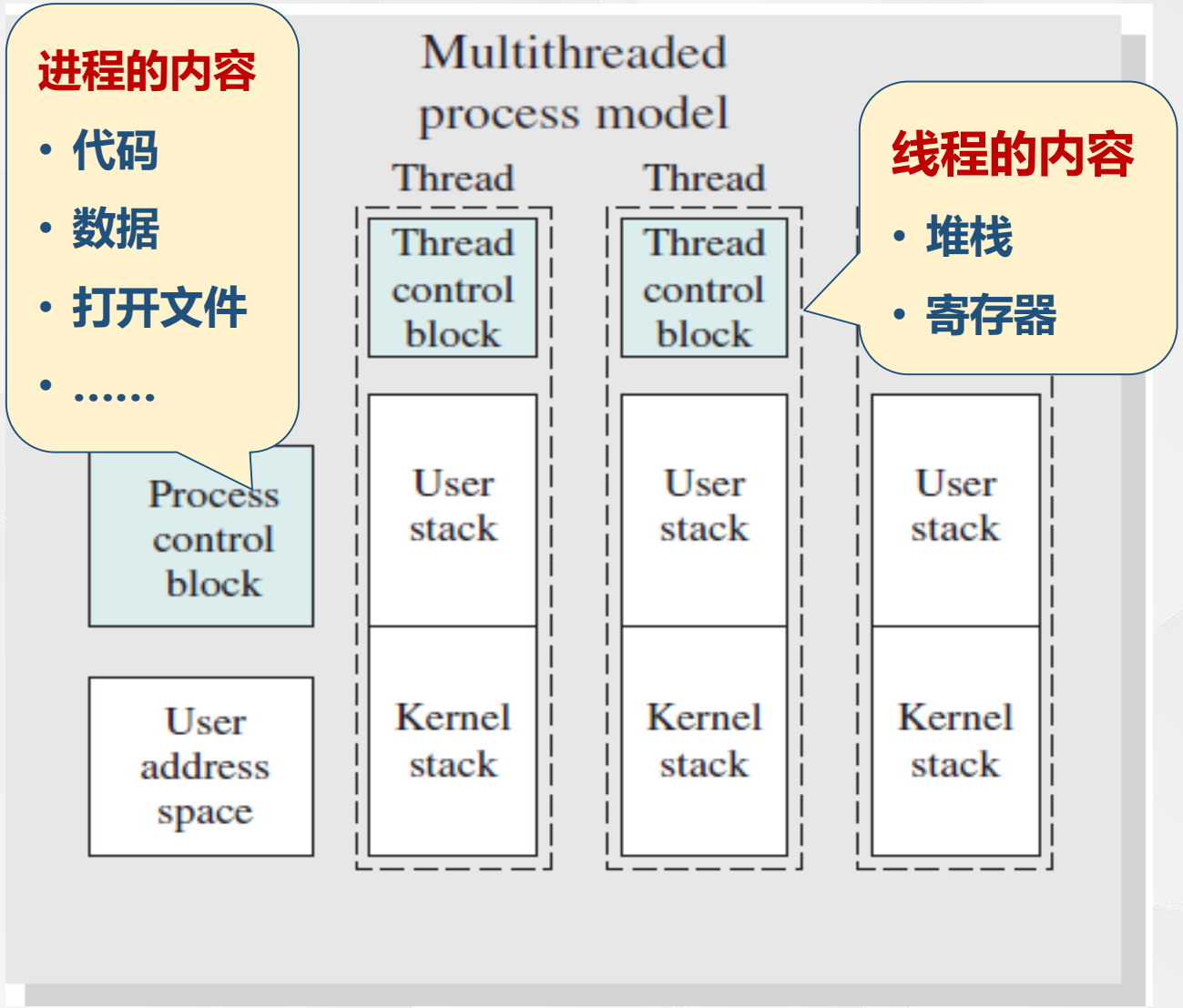



**进程的内容**

- 代码
- 数据
- 打开文件
- ……

**线程的内容**

- 堆栈
- 寄存器



# 线程的特点



**优点：**

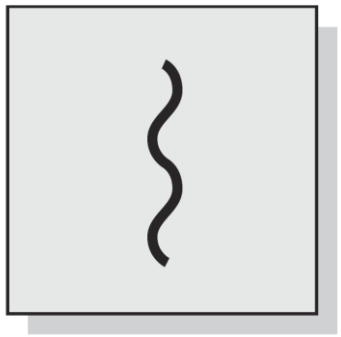
- 创建和撤销一个线程比创建一个进程的开销要小得多（10 ~ 100倍）。
- 因为共享地址空间和数据，线程间通信十分方便且高效。
- 同一个进程的线程间切换速度快（纳秒级）。
- 在进程内创建多线程，可以提高系统的并行处理能力，加快进程的处理速度。

**缺点：**

- 一个线程崩溃，会导致其所属进程内的所有线程崩溃。

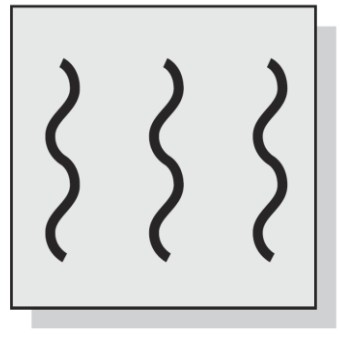
# 不同操作系统对线程的支持





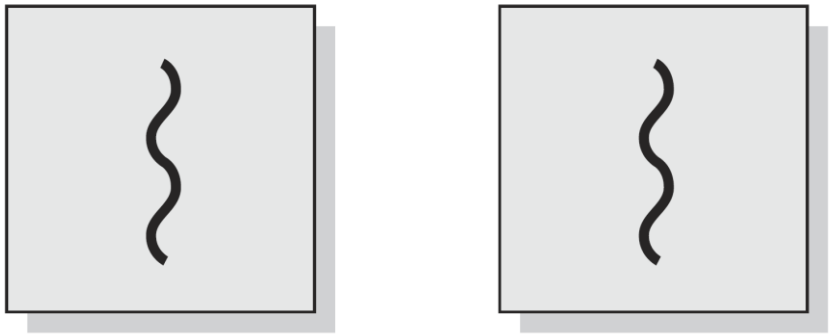
One process
One thread

例：MS-DOS



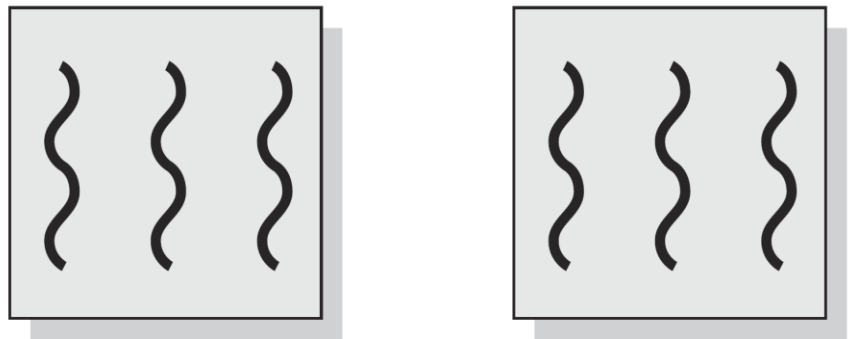
One process
Multiple threads

例：pSOS



Multiple processes
One thread per process

例：传统UNIX

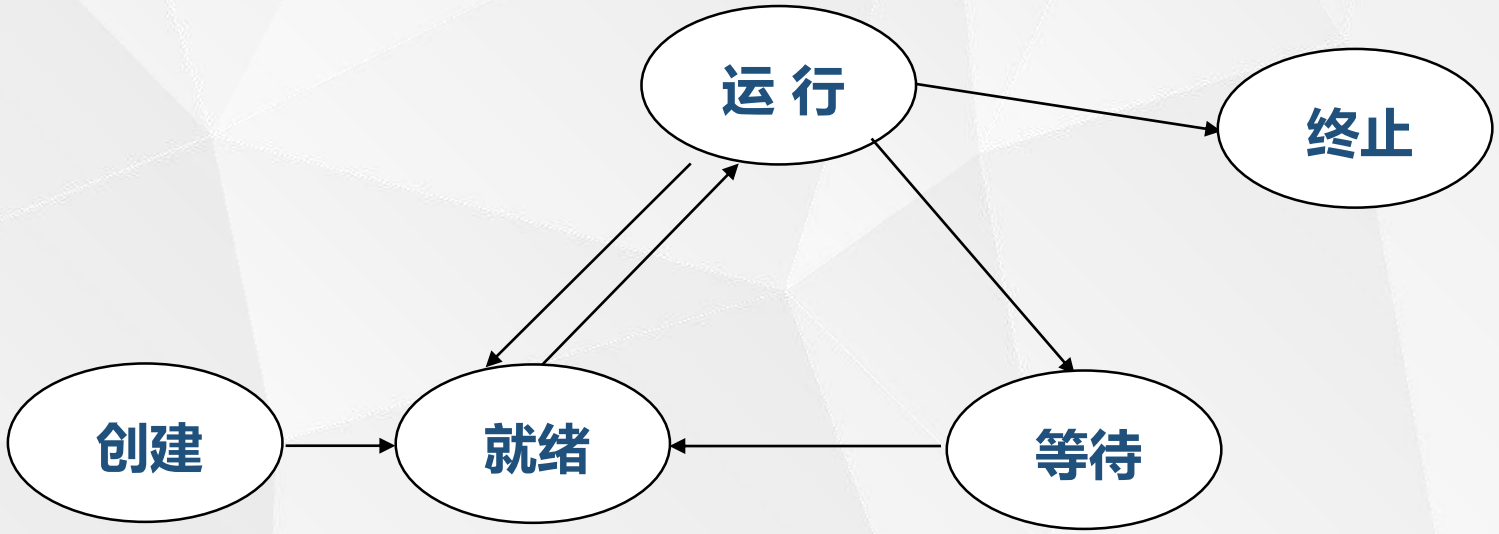


Multiple processes
Multiple threads per process

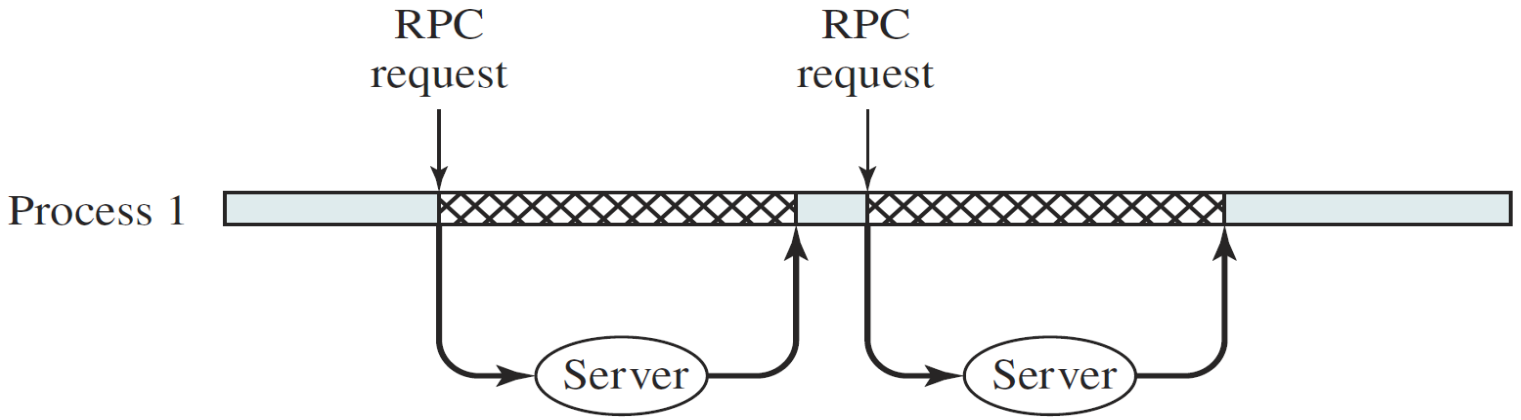
例：现代UNIX

# 线程的状态变迁

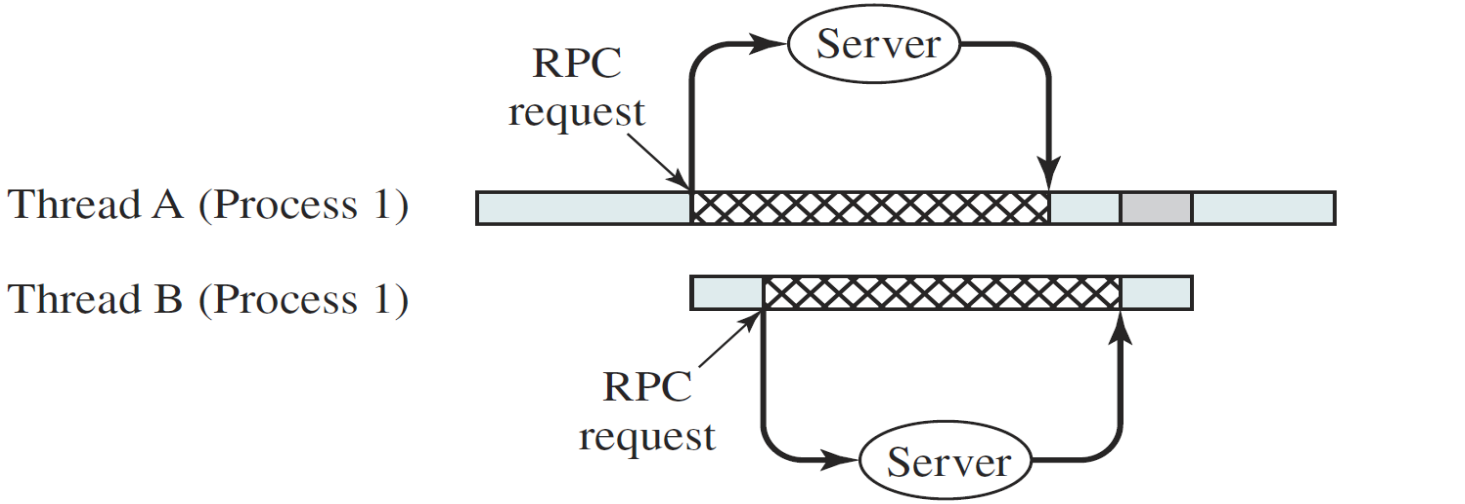




# 采用线程进行远程过程调用

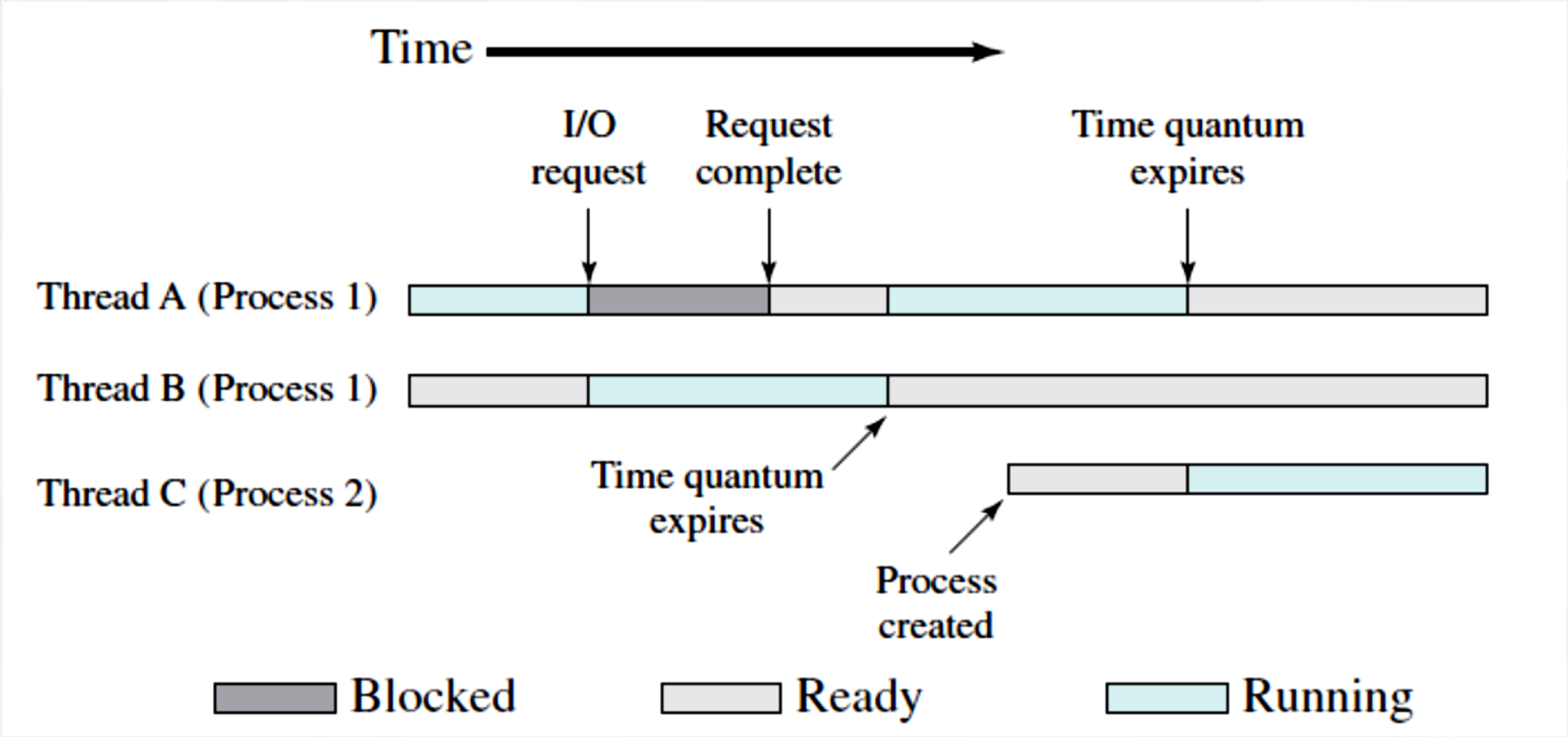



(a) RPC using single thread




(b) RPC using one thread per server (on a uniprocessor)

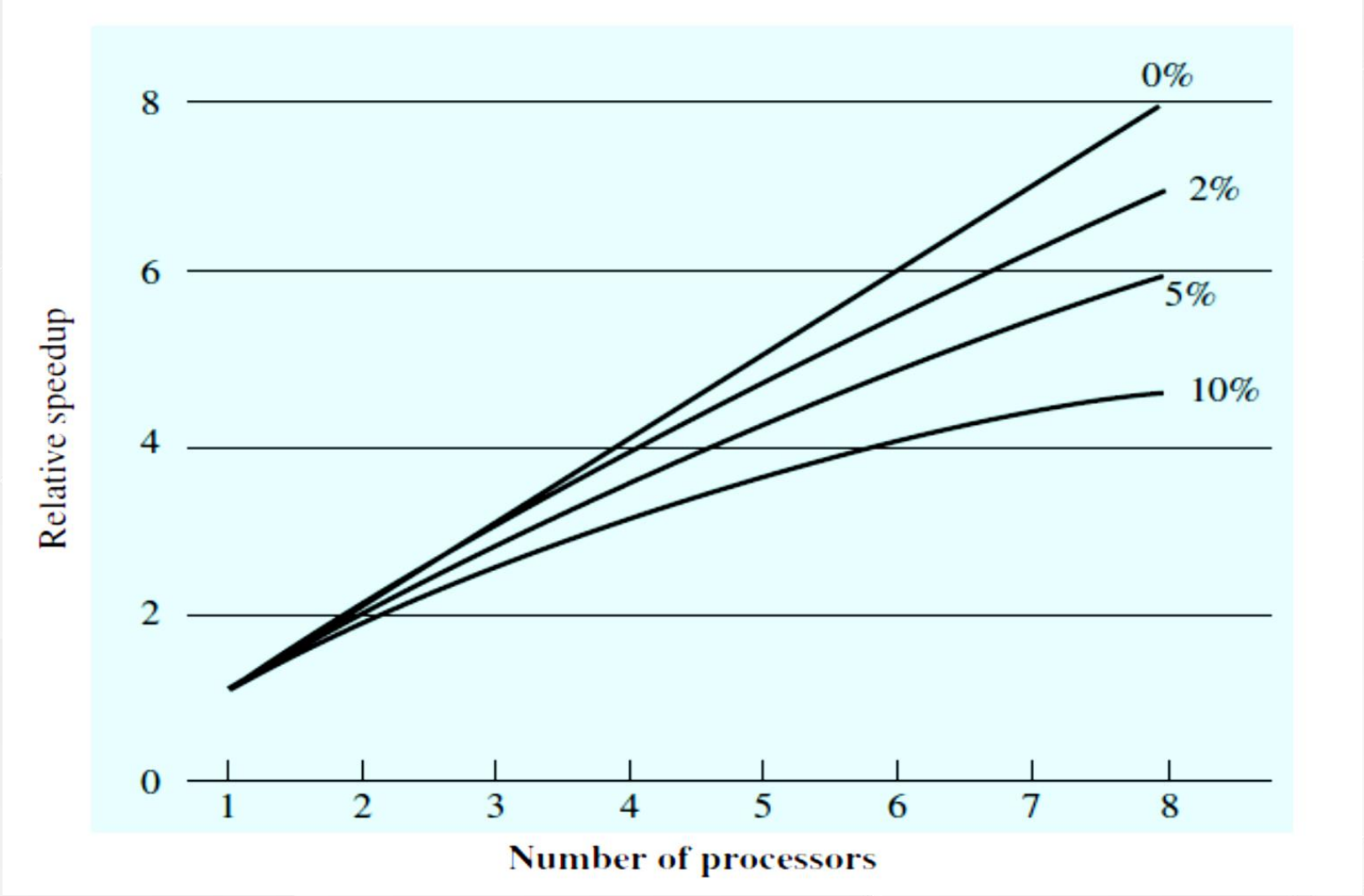
# 两个进程的三个线程交替执行



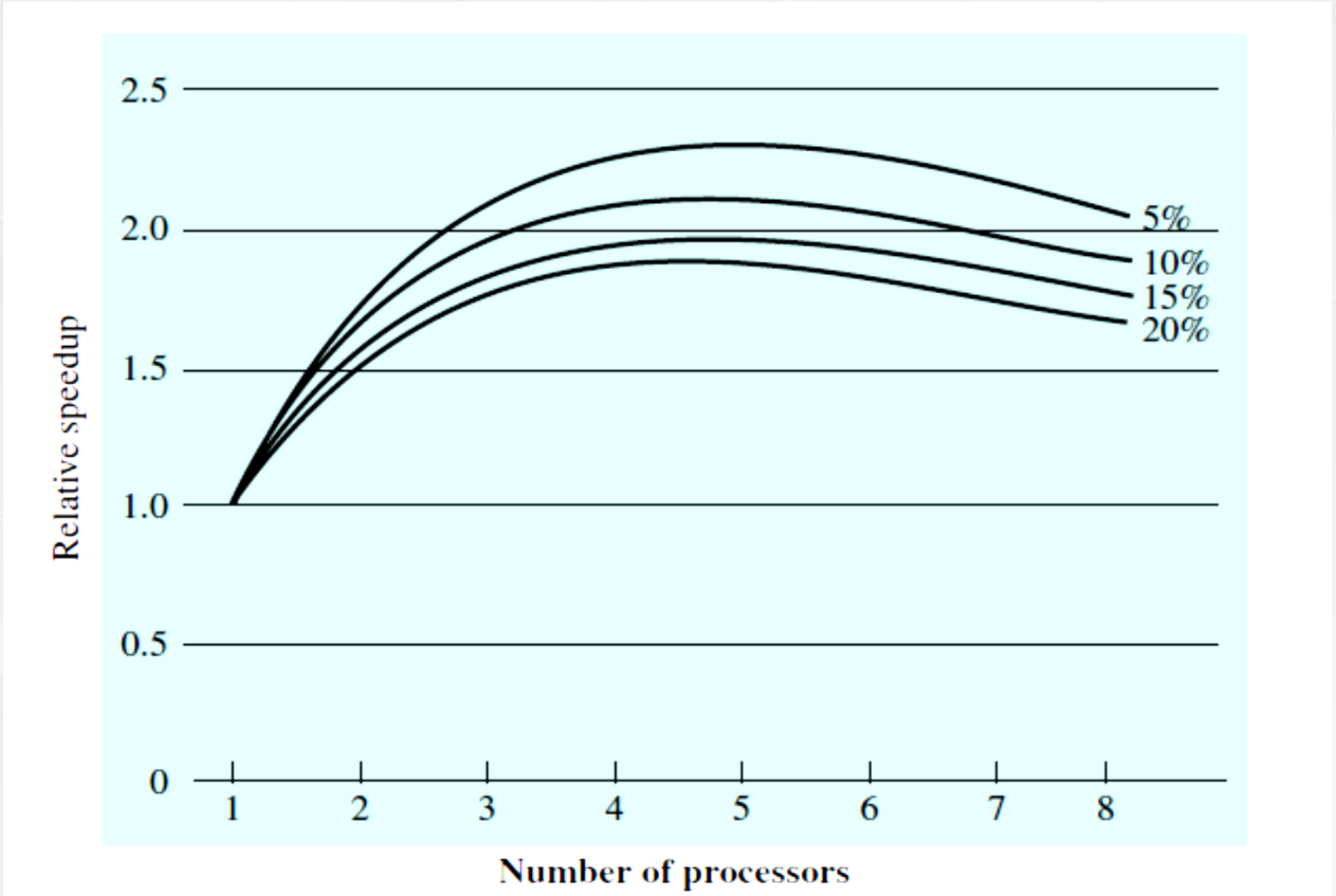
# 多核与多线程



- 应用在多核处理器上所能取得的性能上的好处，实际上取决于它对多处理器资源的利用能力。
- 设 **f** 为程序中能够并行的部分的运行时间在整个程序运行时间中的占比
- **加速比** = 在单处理器上执行程序的时间 / 在N个处理器上执行程序的时间
  = 1/((1-f)+f/N)




串行部分分别为0%，2%，5%以及10%情况下的理想加速比




**考虑实际额外负载的情况下的加速比**

# 线程同步

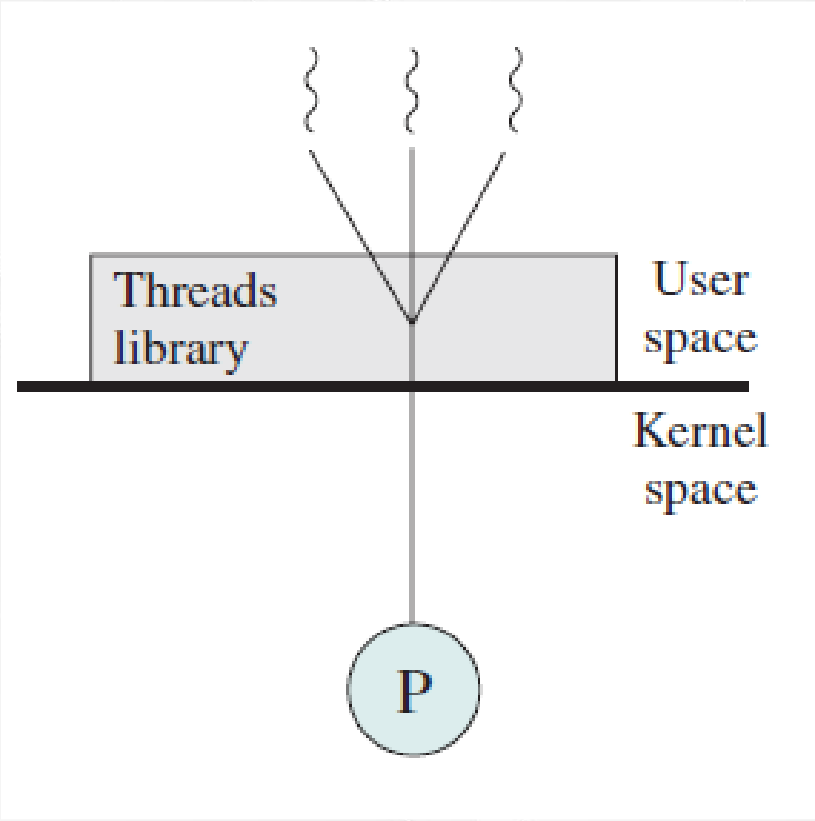


- 对于共享的数据，多线程在访问时需要考虑同步和互斥问题。
- 线程使用的同步机制和方法与进程是一样的。

# 用户级线程



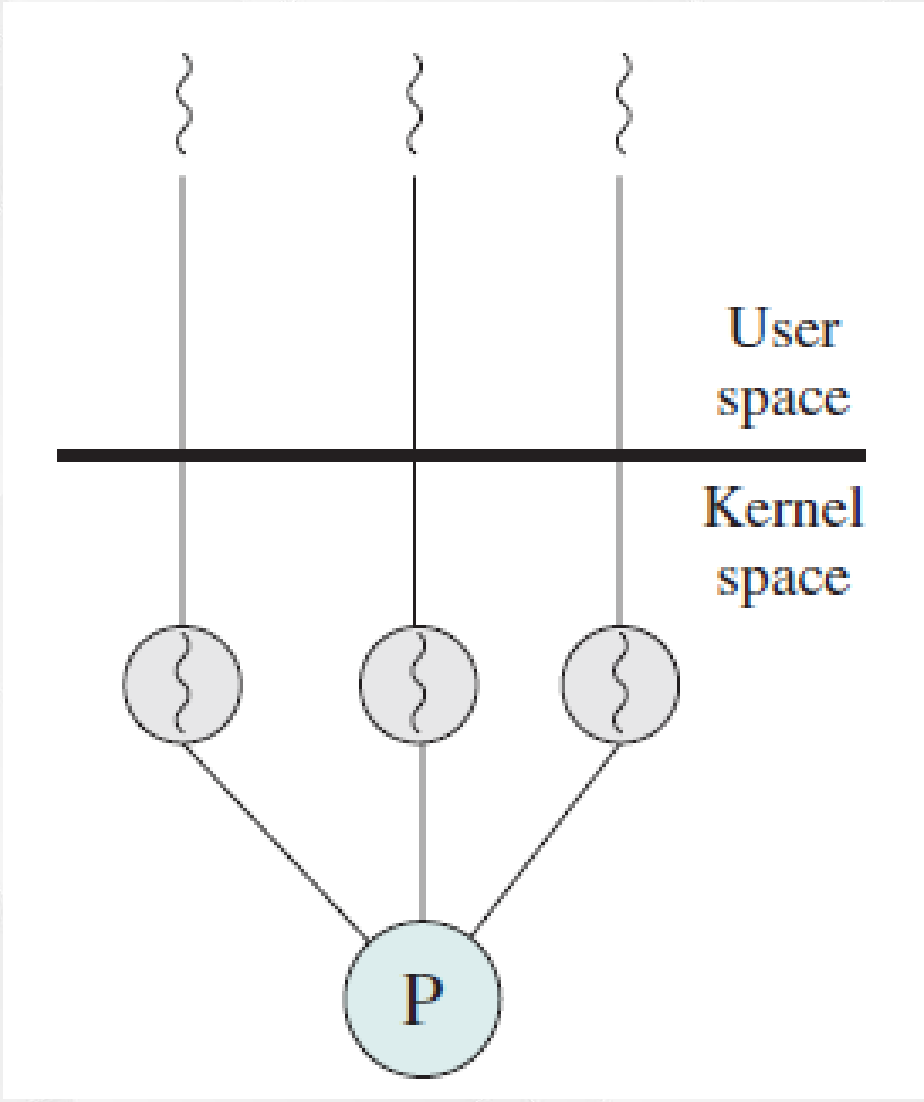
- 所有线程管理的工作都由应用（进程）自己完成，内核不知道线程的存在。
- **例：** POSIX Pthread、Mach-Cthread、Solaris 2 UI-thread
- **缺点：**一个线程阻塞导致整个进程阻塞，且难以利用多核。



# 内核级线程



- 所有线程的管理功能由操作系统内核来完成。
- **例**：Windows NT、Windows 2000
- **缺点**：线程控制时产生处理机状态切换。



# 两种实现方式的对比



| | 用户级线程 | 核心级线程 |
|---|---|---|
| **管理** | 线程库 | 内核 |
| **调度单位** | 进程 | 线程 |
| **切换速度** | 同一进程诸线程间切换，由线程库完成，速度较快 | 由内核完成，速度较慢 |
| **系统调用** | 内核视为整个用户进程的行为 | 内核只视为该线程的行为 |
| **阻塞** | 用户进程 | 线程 |
| **优点** | 线程切换不调用内核，切换速度较快；调度算法可由应用程序决定 | 对多处理器，可同时调度同一进程的多个线程；阻塞发生在线程级 |
| **缺点** | 阻塞发生在进程级<br>难以利用多处理器（多核） | 同一进程内的线程切换速度慢 |

# Linux——pthread_create



```
pthread_create( pthread_t  *thread,
                pthread_attr_t *attr,
                void *(*start_routine)(void *),
                void *arg);
```

**参数：**

- thread指向返回线程标识符的指针；
- attr设置线程属性；
- start_routine线程运行函数地址；
- arg运行函数的参数 。

# Linux——pthread_join



```
int pthread_join ( pthread_t thread,
                   void **thread_return )
```

**参数：**

- thread为被等待的线程标识符；
- thread_return为一个用户定义的指针，它可以用来存储被等待线程的返回值。这个函数是一个线程阻塞的函数，调用它的函数将一直等待到被等待的线程结束为止，当函数返回时，被等待线程的资源被收回。

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

int i=0;
void thread(void)
{
    i = i*2;
    printf("i=%d\n", i);
}
```

**Q：可能的运行结果?**

```
int main(void)
{
    pthread_tid;
    int ret;

    ret=pthread_create(
            &id,
            NULL,
            (void *)thread,
            NULL);
    if (ret!=0){
        printf ("Create pthread error!\n");
        exit (1);
    }
    i=i+5;
    printf("i=%d\n", i);
    pthread_join(id, NULL);
    return (0);
}
```

```
int A=0;
void *subp1() {
    printf("A in thread is %d\n",A);
    A = 10;
    exit(0); //如果线程以exit结束呢?
}
main() {
    pthread_t  t1;
    int  pid;
    pid = fork();
    if (pid==0) {
        printf("A in son process is %d\n",A);
        A=100;
        exit(0);
    }
    wait();
    pthread_create(&t1, NULL, &subp1, NULL);
    pthread_join(t1, NULL);
    printf("A in father process is %d\n",A);
}
```

**Q：运行结果?**

# 第4章 小结

- **进程引入**
  - 程序的顺序执行、定义、特点
  - 程序的并发执行、定义、特点
- **进程概念**
  - 进程定义，进程与程序的区别
  - 进程的状态、状态变迁图
  - 进程描述：PCB的定义与作用，进程的组成
- **进程控制**
  - 进程控制原语：基本进程控制原语（创建、撤销、等待、唤醒）
  - Linux中与进程控制相关的系统调用（fork、exec、exit、wait）
- **进程的相互制约关系**
  - 进程互斥：临界资源、临界区
  - 进程同步

# 第4章 小结（续）

- **进程同步机构：**锁、信号灯
- **进程同步与互斥的实现**
  - 用信号灯的P、V操作实现进程互斥
  - 两类同步问题的解答：合作进程的执行次序、共享缓冲区的合作进程的同步
  - 生产者——消费者问题及解答
- **进程通信**
  - 进程通信的基本概念
  - Linux中的进程通信机制（软中断信号、管道、消息队列、共享内存、信号量）
- **线程**
  - 线程的定义、线程与进程的区别、线程的特点、线程的实现方式
  - Linux的线程库函数