计算机操作系统

Operating Systems

李　琳

第二章 进程的描述与控制

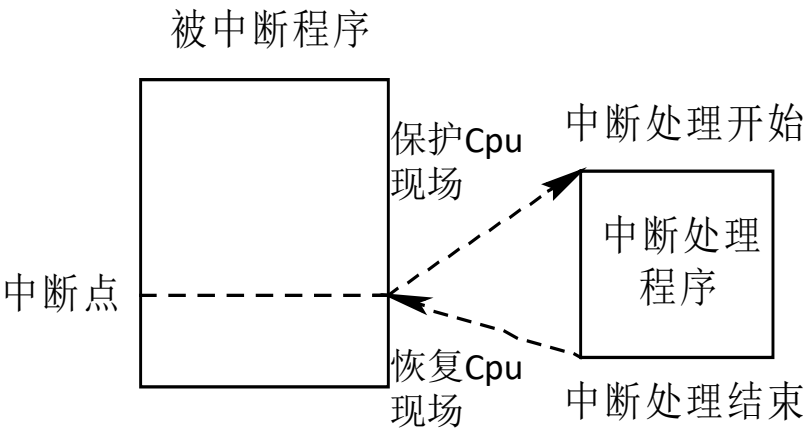
引子：中断和异常

- 中断 (Interrupt)
  - ✓外部硬件设备所产生的信号异步：
  - ✓产生原因和当前执行指令无关，如程序被磁盘读打断
- 异常 (Exception)
  - ✓软件的程序执行而产生的事件
  - ✓包括系统调用（System Call）
  - ✓用户程序请求操作系统提供服务
  - ✓– 同步：产生和当前执行或试图执行的指令相关
- 内核态（管态）与用户态（目态）
  - ✓为了保护操作系统内核的数据与代码，执行应用程序时处理机是处于用户态下。如果需要使用计算机资源或使用OS提供的功能，则需要通过系统调用，转为执行OS内核代码，这个时候处理机是处于系统态或称为内核态下。

引子:中断和异常

系统调用示例：printf() -> write()->sys_write()

CPU上下文环境/现场：是指以指令寄存器IR和指令计数器IP为代表的一组值，体现了CPU某一时刻的状态，且可以还原。



用户地址空间

应用程序

```
#include <stdio.h>
 int main() {
    printf("Hello World!\n");
    return 0;
 }
```

libc

```
write(1, "Hello World!\n" ,13)
{
…
/*参数处理 */
push $__NR_write   /* 第一个参数：系统调用号 */
push $1  /* 第二个参数：file descriptor* /
push  "Hello World!\n"
push  $13
syscall  (不同平台对于的指令不同，例如ARM中的svc, x86中的sysenter, RISC-V中的ecall)
…

}
```

内核地址空间

下陷处理

```
sys_syscall:
…
call syscall_table[__NR_write]
…
```

系统调用处理

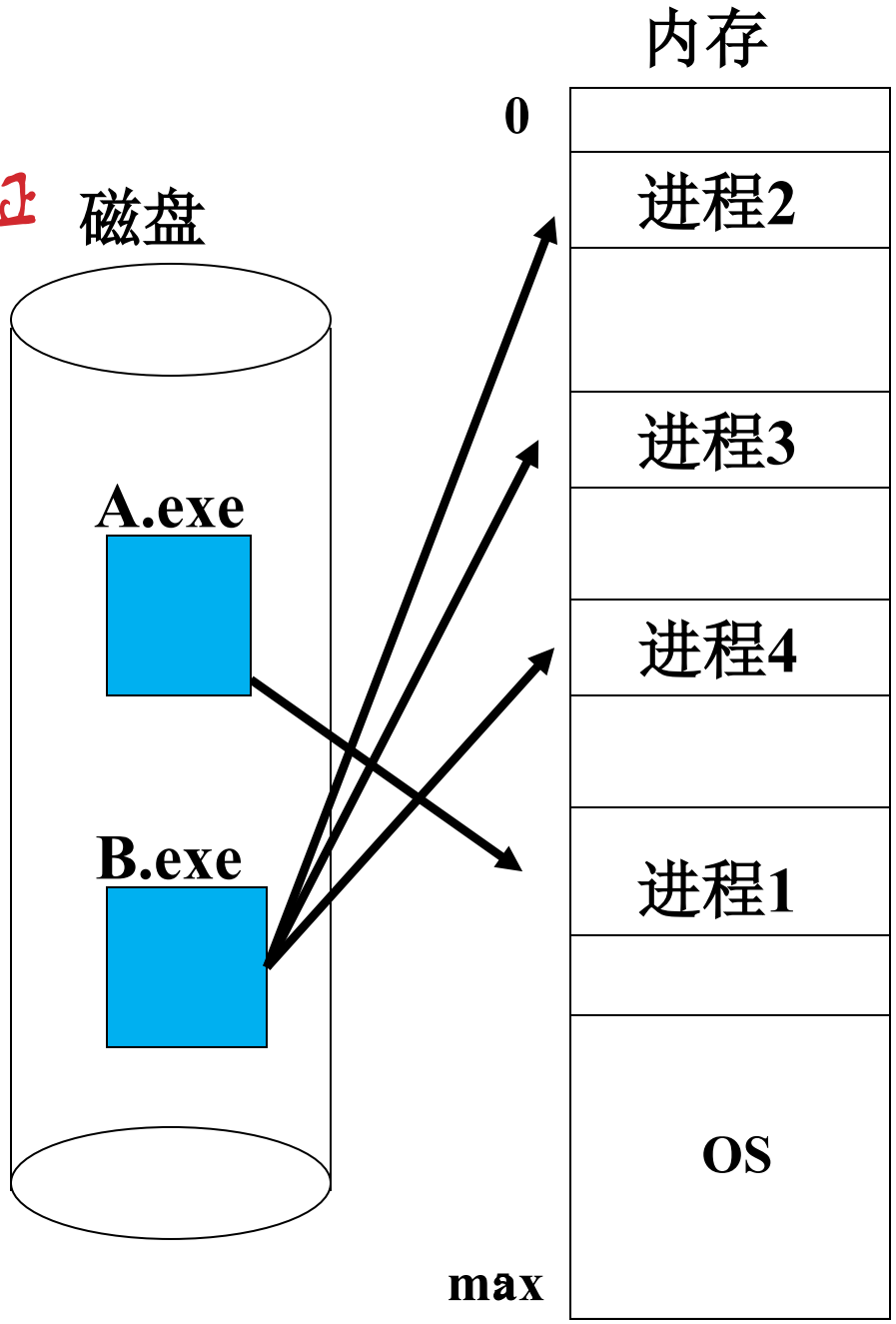
```
sys_write {
…
return error_no;
}
```

2.1 进程的描述

2.1.1 进程的定义和特征

- 进程概念的产生
  - ✓在多道程序环境下，程序的并发执行代替了程序的顺序执行，程序的活动不再处于封闭系统中，从而出现了许多新特征。为此人们引入了新概念—进程

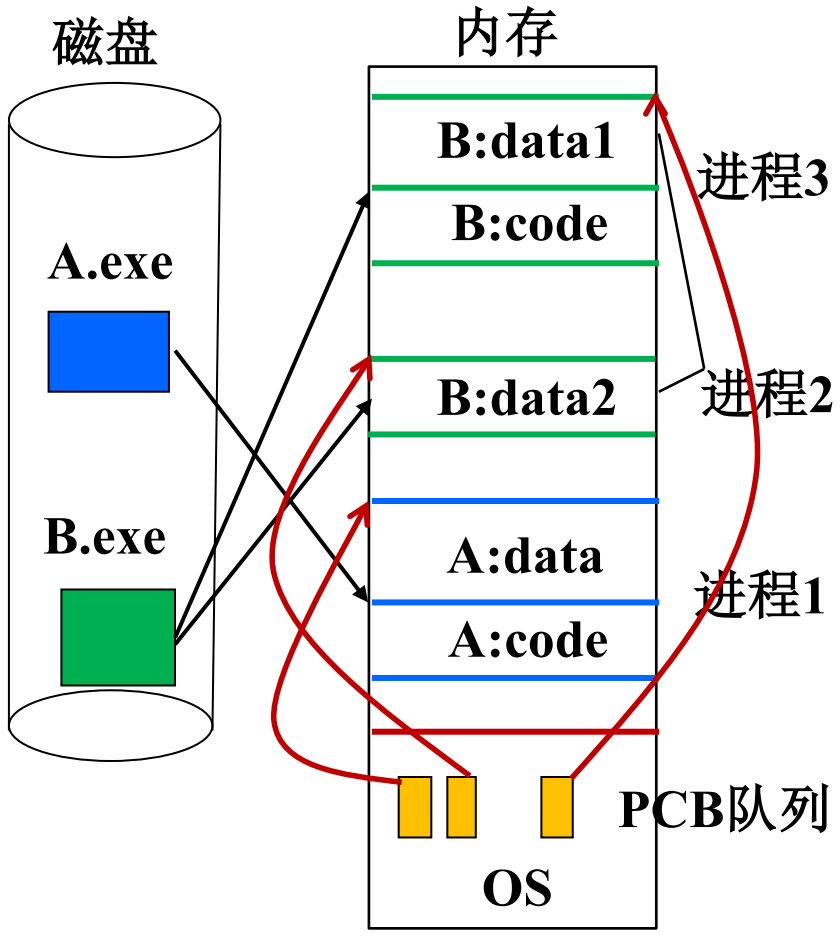
- 动态性：一次执行过程
- 并发性：多道程序环境
- 独立性：资源分配和调度单位
- 异步性：执行顺序未知



2.1 进程的描述

2.1.1 进程的定义和特征

- 进程的结构特征
  - ✓程序段
  - ✓数据段
  - ✓进程控制块（PCB）

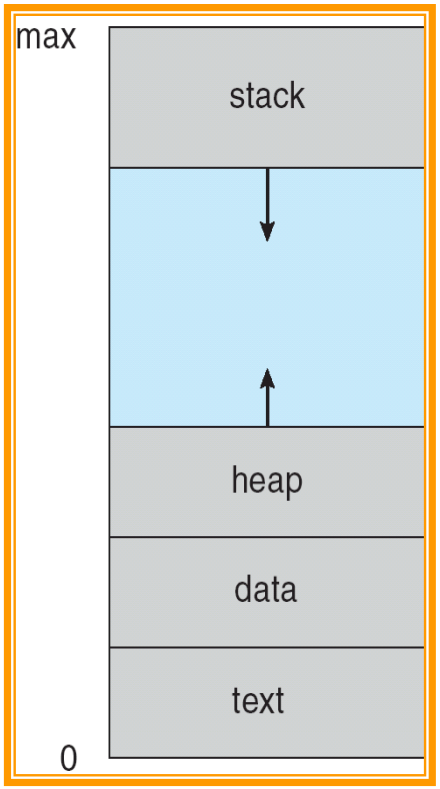


- 进程的定义
  - ✓程序在处理机上的一次执行过程；
  - ✓可以和别的计算并发执行的计算；
  - ✓进程是程序在一个数据集合上运行的过程，是系统进行资源分配和调度的一个独立单位；
  - ✓是一个具有一定功能的程序，是关于某个数据集合的一次运行活动。

2.1 进程的描述

2.1.1进程的定义和特征

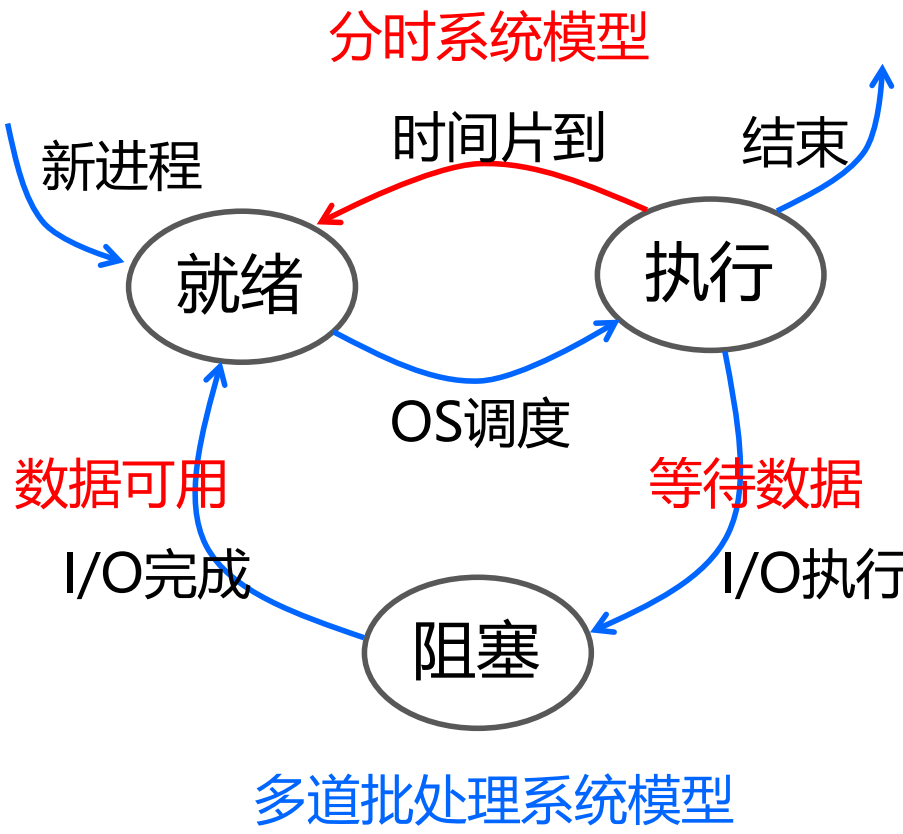
- 作业、程序和进程的区别
- 作业
  - ✓人们提交给计算机处理的若干任务集合.bat
- 程序
  - ✓ 一组完整的、包含的计算机指令、用来执行特定的任务 .exe .dll
- 进程
  - ✓一个具有一定功能的程序关于某个数据集合的一次运行活动



2.1 进程的描述

2.1.1 进程的基本状态及转化

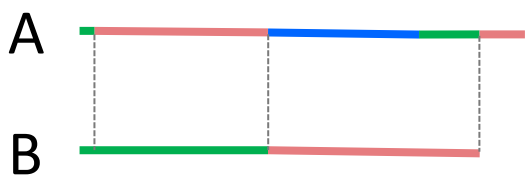
- 进程的三种基本状态
- 就绪状态
  - ✓进程已经获得除处理机之外的所有资源，一旦获得处理机就可以立即执行
- 执行状态
  - ✓一个进程已经获得了必要的资源，正在处理机上运行
- 阻塞状态
  - ✓正在执行的进程，由于发生某事件而暂停无法继续执行，（如等待I/O、等待数据）



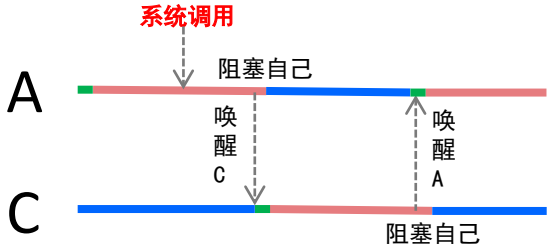
必要的知识概念：中断机制和CPU上下文环境
系统进程和用户进程

示例

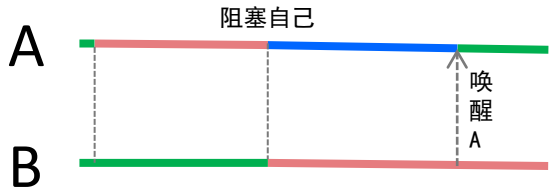
1、用户进程A和B并发运行：A就绪后先被调度执行，B就绪；A因为I/O进入阻塞，B被调度执行；A的I/O结束被唤醒为就绪，B执行结束，A被调度执行（隐藏I/O细节）



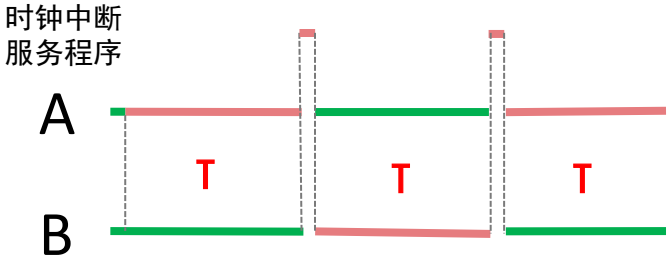
2、用户进程A需要系统进程C提供服务：C处于阻塞，A就绪后被调度执行；系统调用会唤醒C到就绪，阻塞自己；C被调度执行，完成后唤醒A到就绪，阻塞自己；A可能被调度



3、进程A 需要进程B提供数据： A就绪后先被调度执行，B就绪；A执行到需要数据，阻塞自己；B被调度执行到产生数据，唤醒A到就绪；B继续执行



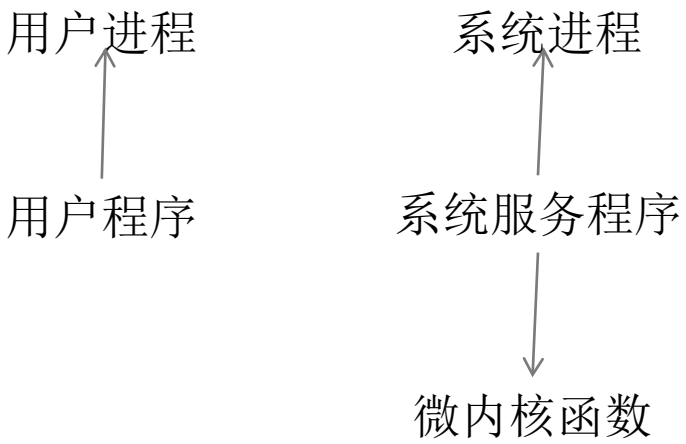
4、用户进程A和B分时执行： A就绪后先被调度执行，B就绪；时钟中断执行中断服务程序，保存A现场，恢复B现场，B被调度执行；时钟中断执行中断服务程序，保存B现场，恢复A现场，A被调度执行；往复轮流。

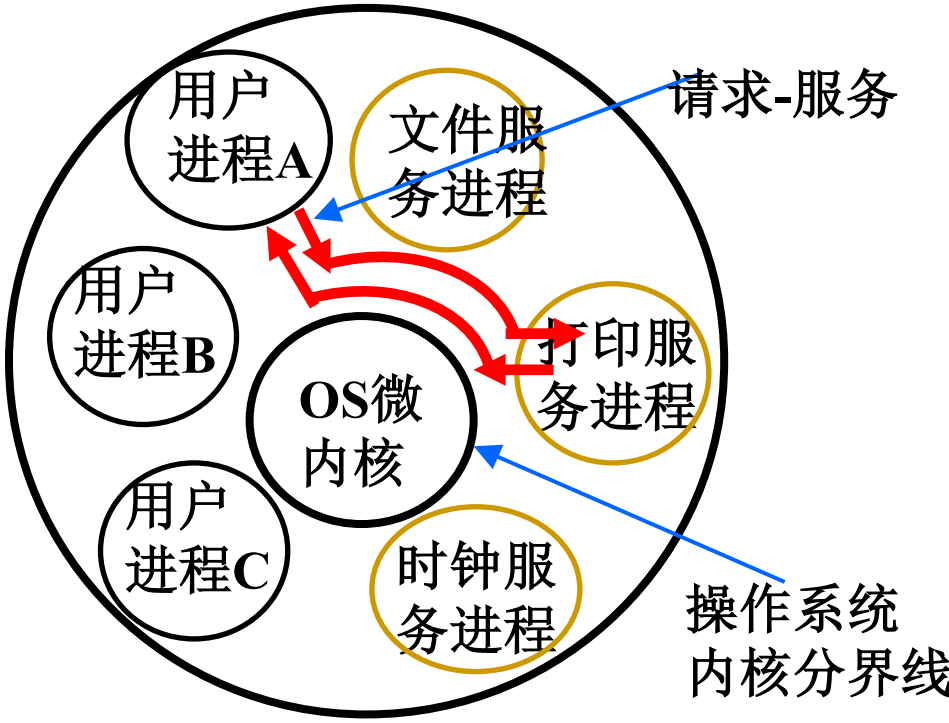


2.1 进程的描述

2.1.1进程的基本状态及转化

- OS一致化的调度模型





试一试

- 某单处理器系统中采用多道程序设计，现有10个用户进程存在，则处于运行、阻塞、就绪的用户进程数量最小和最大值分别可能是多少？

- 以下哪种状态转化是不可能发生的（　）

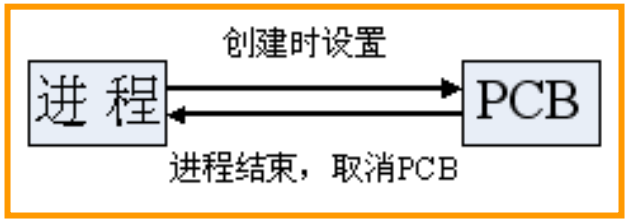
  A、运行—>就绪　　B、运行—>阻塞

  C、阻塞—>就绪　　D、阻塞—>运行

2.1 进程的描述

2.1.2 进程控制块(Process Control Block, PCB)

- 基本概念
  - ✓OS管理进程的数据结构
  - ✓进程控制块是进程存在的唯一标志
  - ✓进程控制块位置：操作系统内核

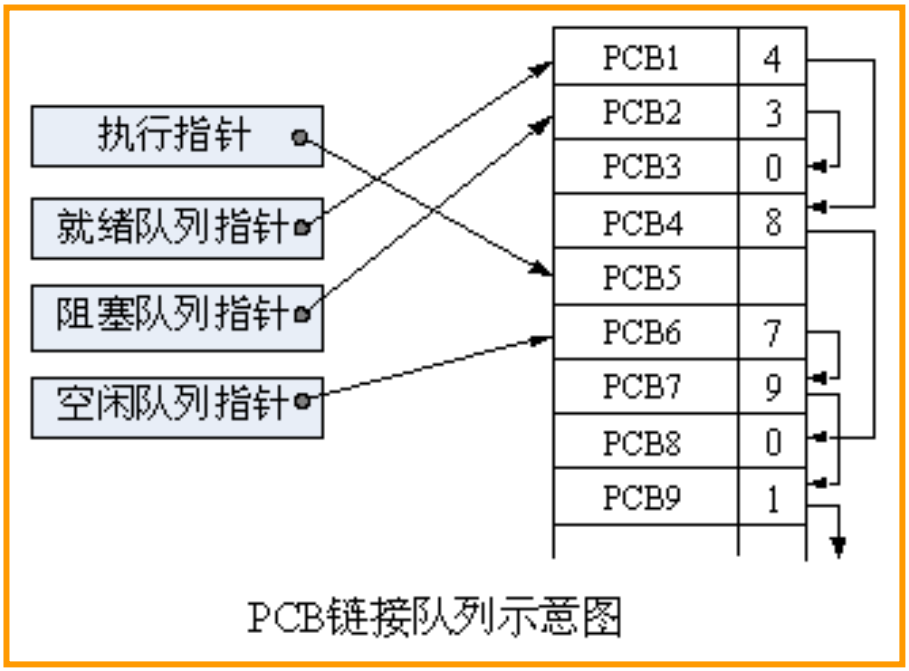


- PCB的内容
  - ✓进程标志符
  - ✓处理机状态
  - ✓进程调度信息
  - ✓进程资源信息

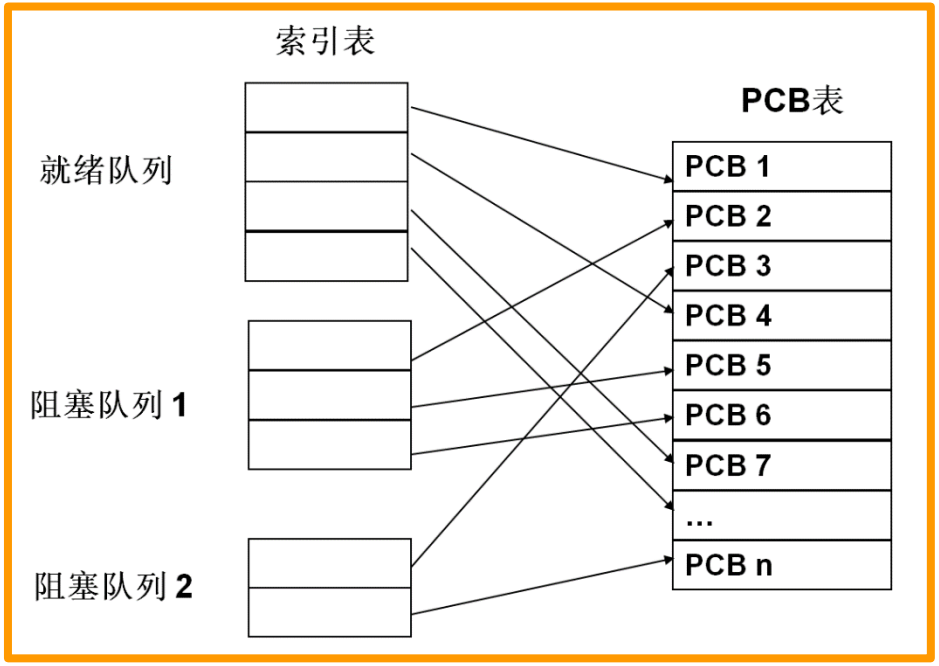
2.1 进程的描述

2.1.2 进程控制块(Process Control Block, PCB)

- PCB的组织形式
  - ✓链接方式
  - ✓索引方式



PCB链接队列示意图



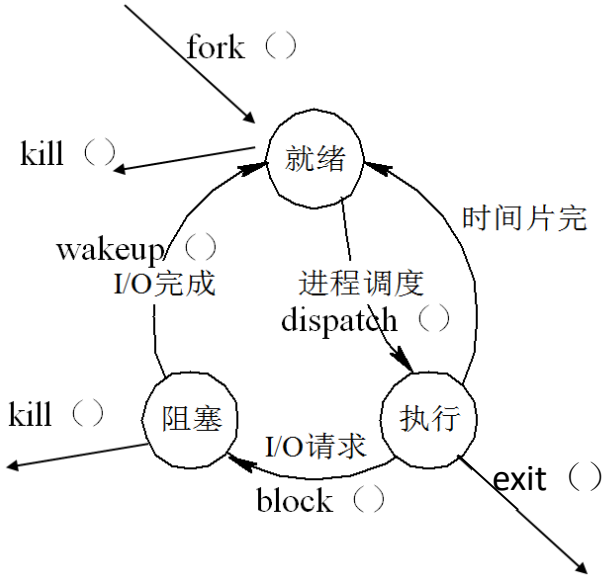
```
/*
*********************************************************************************************************
*                                                TASK CONTROL BLOCK
*********************************************************************************************************
*/

typedef struct os_tcb {
    OS_STK        *OSTCBStkPtr;        /* Pointer to current top of stack                          */

#if OS_TASK_CREATE_EXT_EN
    void          *OSTCBExtPtr;        /* Pointer to user definable data for TCB extension         */
    OS_STK        *OSTCBStkBottom;     /* Pointer to bottom of stack                               */
    INT32U         OSTCBStkSize;       /* Size of task stack (in bytes)                            */
    INT16U         OSTCBOpt;           /* Task options as passed by OSTaskCreateExt()              */
    INT16U         OSTCBId;            /* Task ID (0..65535)                                       */
#endif

    struct os_tcb *OSTCBNext;          /* Pointer to next     TCB in the TCB list                  */
    struct os_tcb *OSTCBPrev;          /* Pointer to previous TCB in the TCB list                  */

#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN || OS_SEM_EN
    OS_EVENT      *OSTCBEventPtr;      /* Pointer to event control block                           */
#endif

#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN
    void          *OSTCBMsg;           /* Message received from OSMboxPost() or OSQPost()          */
#endif

    INT16U         OSTCBDly;           /* Nbr ticks to delay task or, timeout waiting for event    */
    INT8U          OSTCBStat;          /* Task status                                              */
    INT8U          OSTCBPrio;          /* Task priority (0 == highest, 63 == lowest)               */

    INT8U          OSTCBX;             /* Bit position in group  corresponding to task priority (0..7) */
    INT8U          OSTCBY;             /* Index into ready table corresponding to task priority    */
    INT8U          OSTCBBitX;          /* Bit mask to access bit position in ready table           */
    INT8U          OSTCBBitY;          /* Bit mask to access bit position in ready group           */

#if OS_TASK_DEL_EN
    BOOLEAN        OSTCBDelReq;        /* Indicates whether a task needs to delete itself          */
#endif
} OS_TCB;

/*$PAGE*/
/*
*********************************************************************************************************
*                                             GLOBAL VARIABLES
*********************************************************************************************************
```

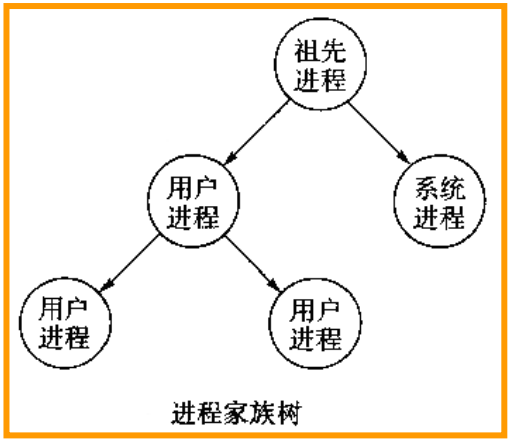
2.2 进程的控制

2.2.1 进程创建

- 引起进程创建的事件
  - ✓用户登录
  - ✓作业调度
  - ✓提供服务。例如：提供打印服务
  - ✓应用请求。例如：用户执行程序
- 进程创建原语 fork（）
- 进程创建过程
  - ✓父进程调用
  - ✓确定父子关系–〉申请空白PCB –〉分配资源 –〉初始化PCB –〉将新进程插入就绪队列–〉进程调度?

在抢占式系统中新进程产生需要重新调度

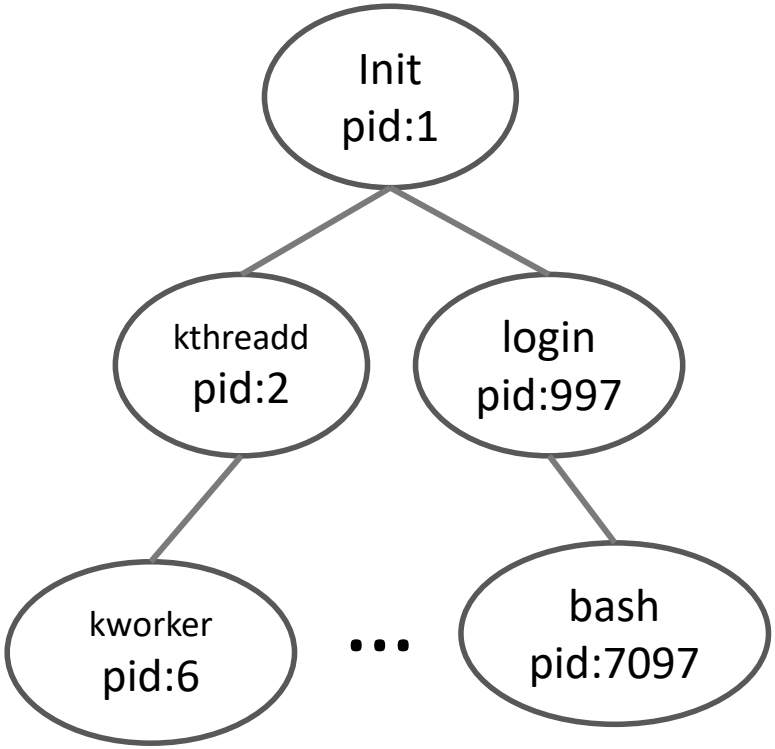




Linux实例：

```
int main () {
    while (1)
      fork();
}
```

如何避免进程创建掠夺资源？



简化的进程树

- 假设先执行父进程，然后再执行子进程
- test.txt中的内容："abcdefghijklmnopqrst…"

```
char str[10];
int fd = open("test.txt", O_RDWR);
if (fork() == 0) {
    ssize_t cnt = read(fd, str, 10);
    printf("Child process: %s\n", (char *)str);
} else {
    ssize_t cnt = read(fd, str, 10);
    printf("Parent process: %s\n", (char *)str);
}
```

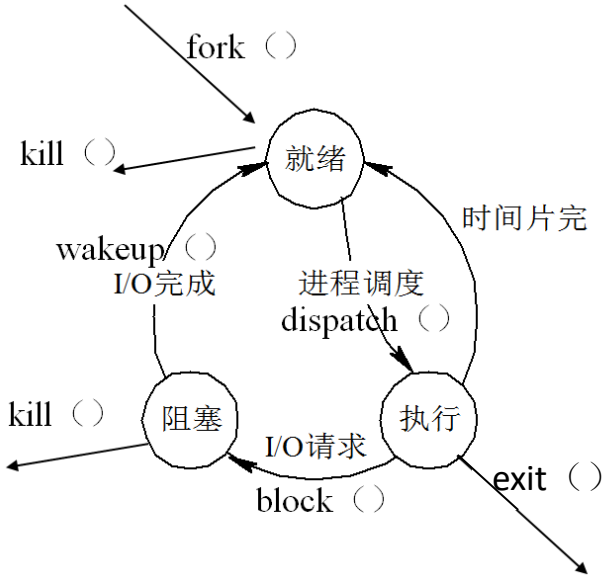
一次调用，二次返回的fork

2.2 进程的控制

2.2.2 进程撤销

- 引起进程撤销的事件
  - ✓正常结束
  - ✓异常结束
  - ✓外界干预
- 进程撤销原语 exit（） kill（）
- 进程撤销过程
  - ✓正常：自己结束时调用
  - ✓非正常：其它进程或父进程调用
  - ✓递归终止子孙进程–〉释放资源–〉从各个队列中弹出–〉回收PCB –〉进程调度?

如果是在执行状态需要重新调度

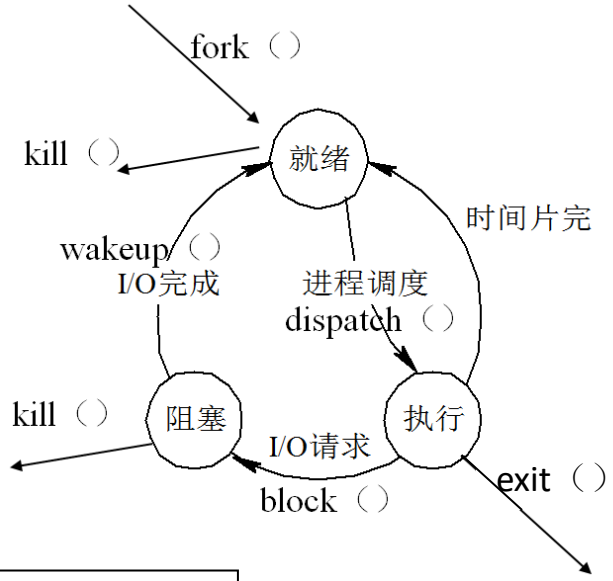


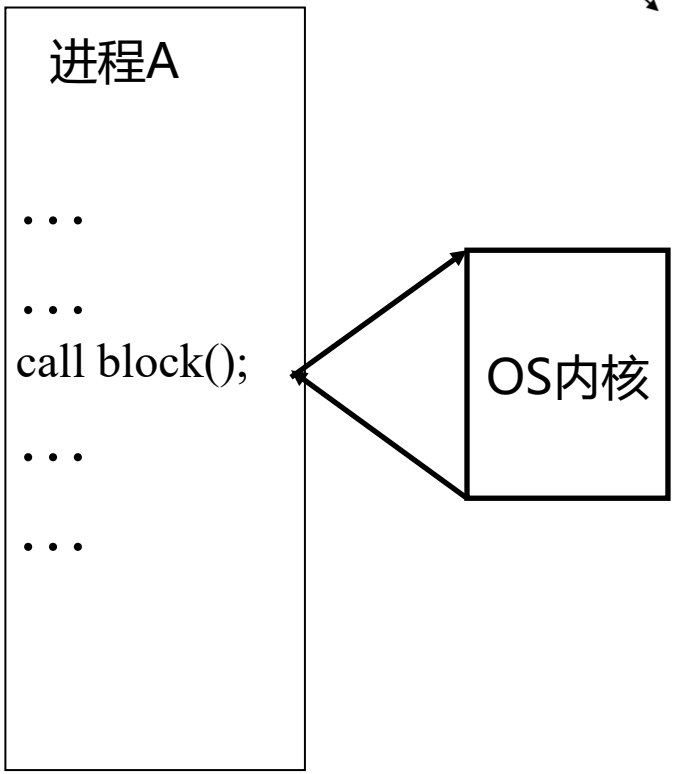
2.2 进程的控制

2.2.3 进程阻塞

- 引起进程阻塞的事件
  - ✓请求系统服务
  - ✓请求操作
  - ✓等待数据、资源
  - ✓等待工作任务
- 进程阻塞原语　block（）
- 进程阻塞过程
  - ✓进程自己阻塞自己
  - ✓插入阻塞队列–〉改变PCB中的状态–〉进程调度?

需要进行进程调度



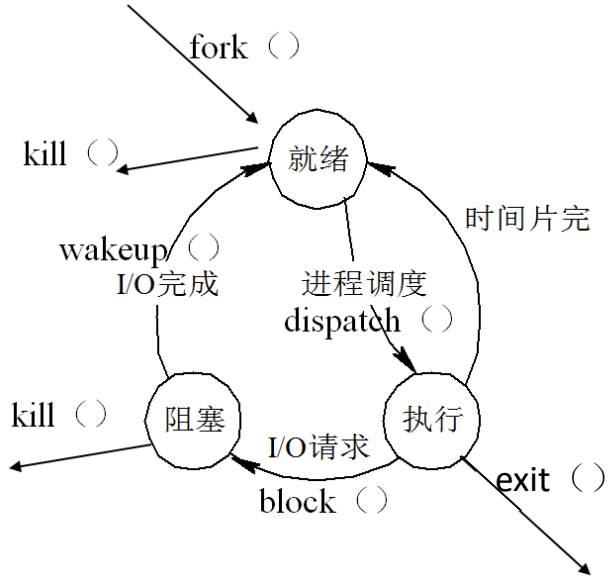


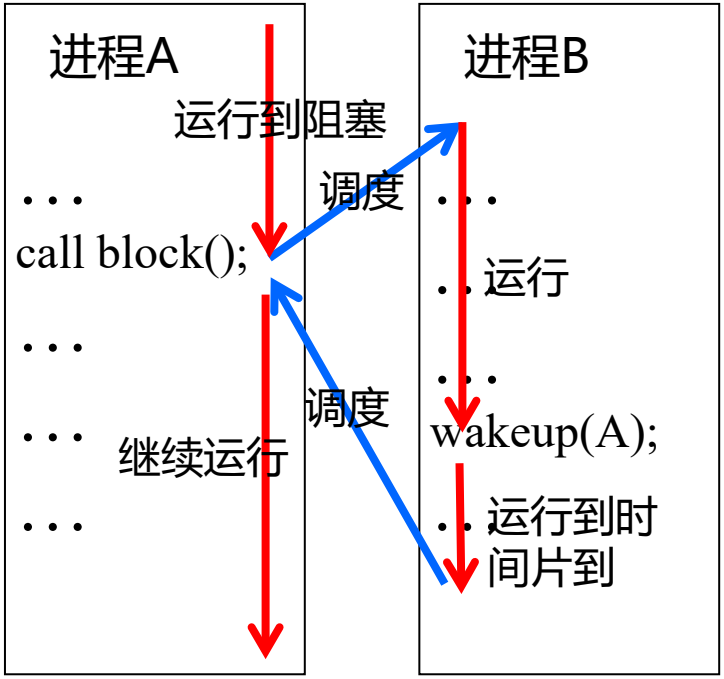
2.2 进程的控制

2.2.4 进程唤醒

- 引起进程唤醒的事件
  - ✓允许系统服务
  - ✓允许操作
  - ✓数据、资源到达
  - ✓工作任务到达
- 进程唤醒原语 wakeup（）
- 进程唤醒过程
  - ✓其它相关进程调用
  - ✓插入就绪队列–〉改变PCB中的状态–〉进程调度?

在抢占式系统中需要重新调度





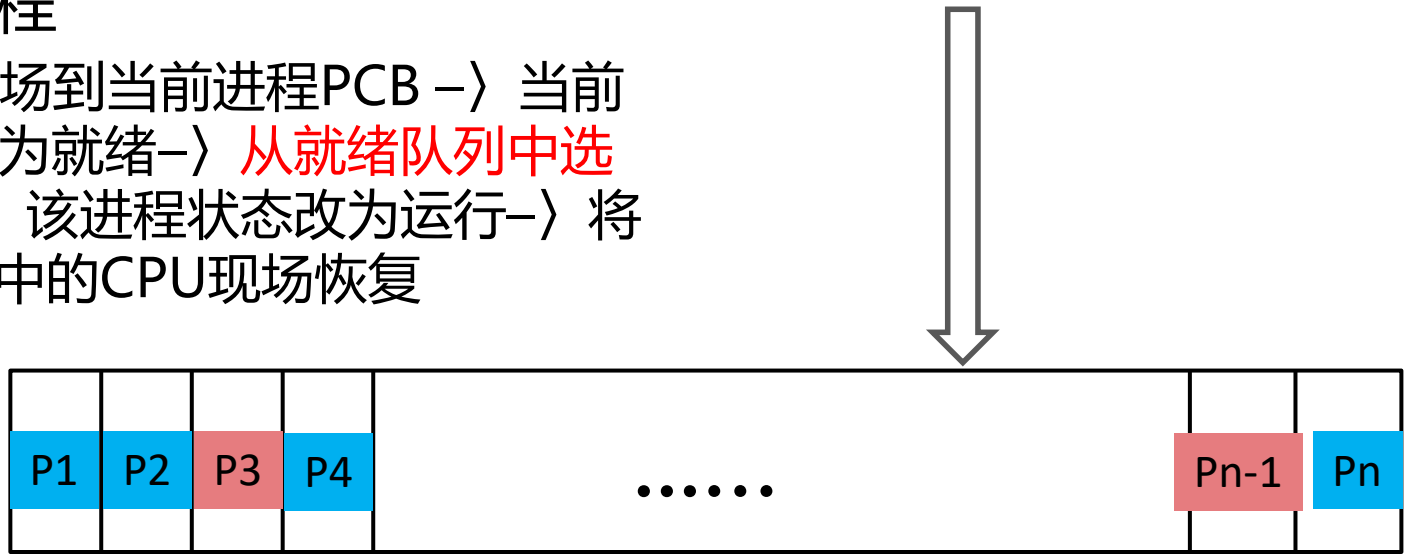
2.2 进程的控制

2.2.5 进程调度

- 引起进程调度的事件
  - ✓正在运行的进程结束或者改变状态
  - ✓时间片到
  - ✓抢占式系统中就绪队列新增了进程
- 进程调度原语 dispatch（）
- 进程调度过程
  - ✓保存CPU现场到当前进程PCB –〉当前进程状态改为就绪–〉从就绪队列中选一个进程–〉该进程状态改为运行–〉将该进程PCB中的CPU现场恢复

现代OS的进程模型

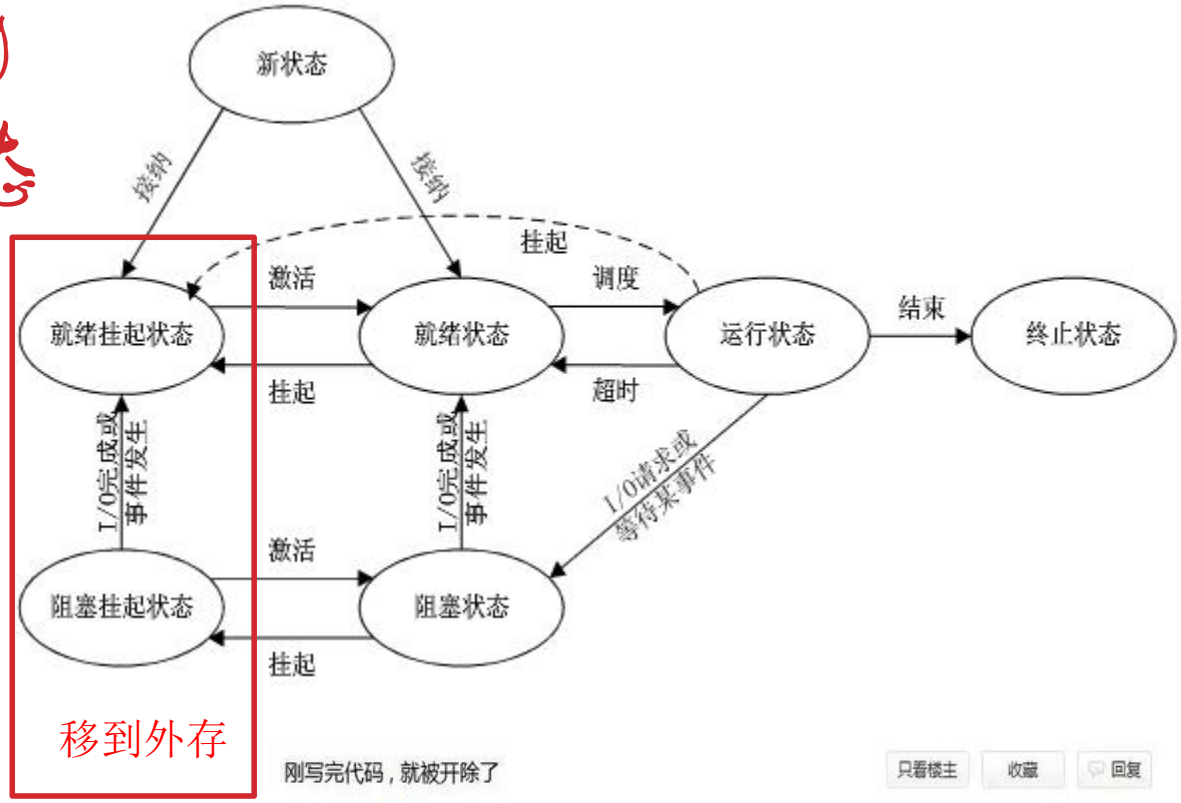
将系统进程和用户进程在同一个架构中进行调度，可以将系统进程的优先级调高，用户进程的优先级调低；系统进程大多为服务型进程，处于阻塞状态。



2.2 进程的控制

2.2.6 挂起状态

- 传统OS中
  - ✓解决内存资源不足问题
  - ✓就绪挂起、阻塞挂起
- 现代OS中
  - ✓应用需求问题
  - ✓暂时不在任务列表
  - ✓实现方式依据不同OS或者更上层平台



试一试

- 以下不可能引起进程调度的是（　）

  A、一个进程完成工作后被撤消

  B、一个进程从就绪状态变成了运行状态

  C、一个进程从阻塞状态变成了就绪状态

  D、一个进程从运行状态变成了的阻塞状态或就绪状态

- 下列选项中，导致创建新进程的操作是（ ）

  1 用户登录成功　2 设备分配　3 启动程序执行

  A. 1和2　B. 2和3　C. 1和3　D.1、2和3

- 一个进程被唤醒，意味着（ ）

  A．该进程重新占有了CPU　　B．进程状态变为就绪

  C．它的优先权变为最大　　D．其PCB移到就绪队列的队首

2.3 线程

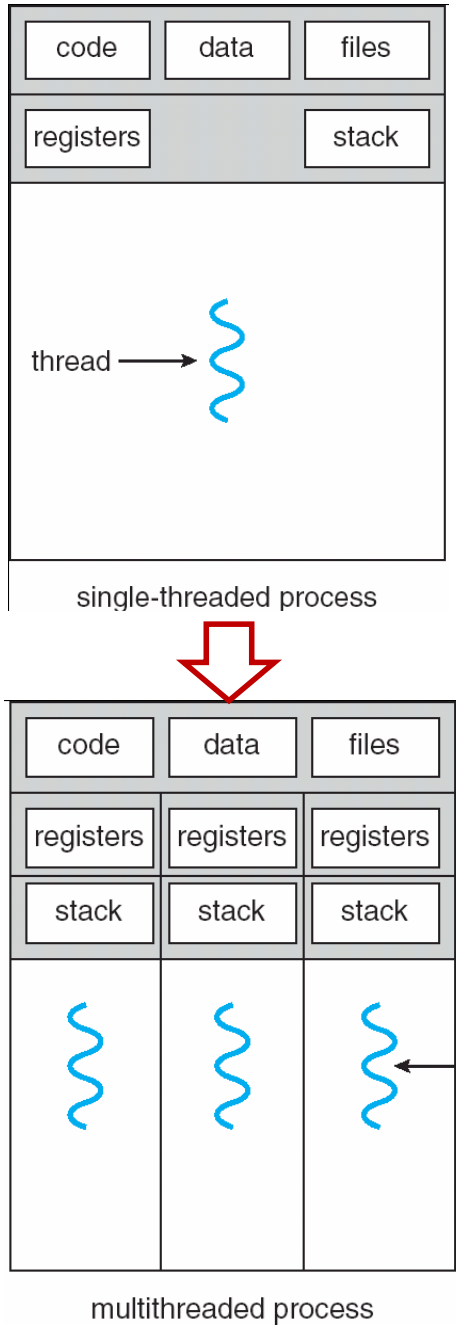
2.3.1 线程的基本概念

- 线程的引入

进程 线程

  - ✓传统的进程拥有资源赋予和调度对象两个功能
  - ✓限制了任务内部的并发度，降低效率；

    例： 复杂的存储、绘制和计算任务同时存在时
  - ✓将进程内部扩展多条执行线索，但同时共享资源
  - ✓将传统进程的两个功能分解
- 线程的概念
  - ✓线程有时候称为轻量级进程
  - ✓是CPU调度和分派的基本单位
  - ✓可并发执行
  - ✓共享进程资源
  - ✓一个进程至少有一个主线程



2.3 线程

2.3.1 线程的基本概念

- 线程的开销
  - ✓线程控制块（TCB)
  - ✓寄存器状态，堆栈，局部变量拷贝
  - ✓线程运行状态（就绪、阻塞、执行）
  - ✓可以创建、撤消另一个线程
  - ✓优先级与信号屏蔽
- 引入线程的优点
  - ✓创建一个新线程花费时间少（结束亦如此）
  - ✓线程的切换比进程的切换花费时间少
  - ✓同一进程内的线程共享内存和文件，因此它们之间相互通信无须调用内核
  - ✓适合多处理机系统（易于并行）

2.3 线程

2.3.1 线程的实现方式

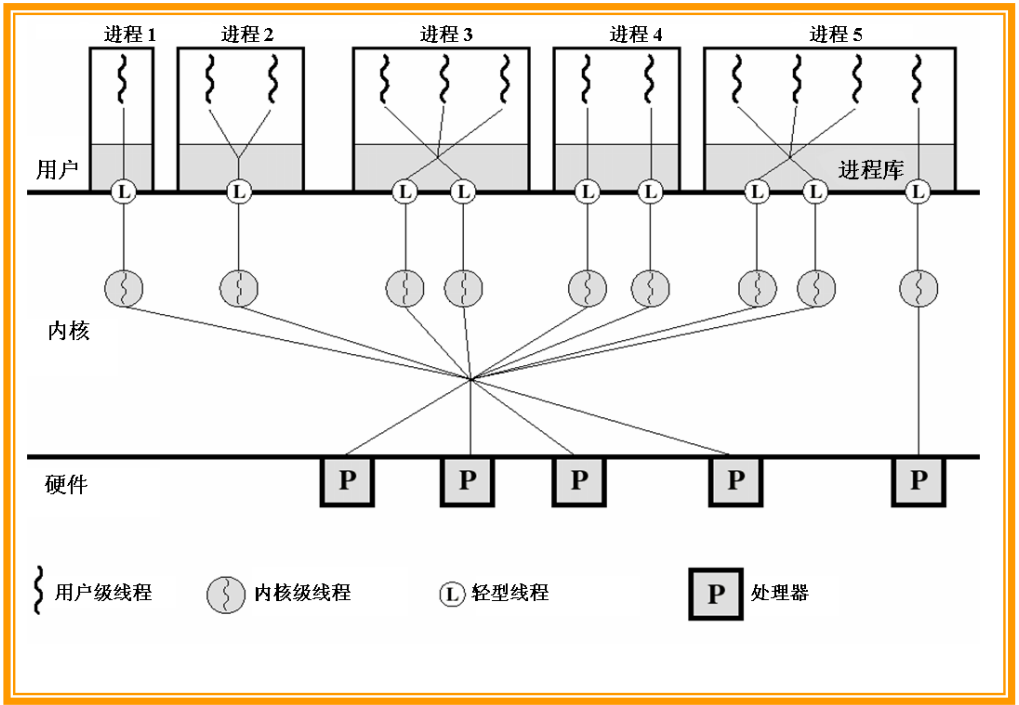
- 内核支持线程
  - ✓内核感知的线程，每个线程都有TCB
  - ✓线程的创建、撤销等均通过系统调用在由内核程序完成
  - ✓以线程为单位进行调度
  - ✓例如： Windows XP及其后的OS、MAC OS
- 用户级线程
  - ✓仅存在于用户空间中，内核并不知道线程的存在
  - ✓线程的创建、撤销等均通过用户的线程包管理，无系统调用
  - ✓还是以进程为单位进行调度，如果不在一个进程中的2个线程切换
  - ✓例如：JAVA的线程（虚拟机）

问题：假如在分时系统中，进程A中包含了一个线程，另一个进程B中含有100个线程；如果采用内核线程，则两个进程获得CPU时间的比例？如果采用用户线程呢？

2.3 线程

2.3.1 线程的实现方式

- 两者结合的方法
  - ✓系统设置一些内核线程，被内核所感知，并调度运行
  - ✓进程中设置用户线程，运行时映射到空闲的内核线程上
  - ✓在用户空间管理线程调度，易于提高效率
  - ✓在内核空间执行线程，易于并行
  - ✓例如：Solaris系统



Linux与WIN32下的线程比较

OS	WIN32	Linux
线程创建	CreateThread	pthread_create
线程终止	执行完成后退出；线程自身调用ExitThread函数即终止自己；其他线程调用TerminateThread函数	执行完成后退出；线程本身调用pthread_exit 退出；其他线程调用函数pthread_cance终止
获取线程ID	GetCurrentThreadId	pthread_self
创建互斥	CreateMutex	pthread_mutex_init
获取互斥	WaitForSingleObject、WaitForMultipleObjects	pthread_mutex_lock
释放互斥	ReleaseMutex	phtread_mutex_unlock
创建信号量	CreateSemaphore	sem_init
等待信号量	WaitForSingleObject	sem_wait
释放信号量	ReleaseSemaphore	sem_post

编程的技巧——多线程

- 多线程下载
- 服务器端的多线程服务
- 显示线程、计算线程与调度线程
- 多核芯片下的并行计算
- 。。。

第一次作业

- 1、简述操作系统的发展过程，特别说明因为何种原因促使操作系统从一种类型向另一种类型转变。
- 2、指出如下的每一个进程状态间的转移是否可能发生。如果可能发生，试举一个可以引起这种转移的例子。
- （1）运行—>就绪 （2）运行—>阻塞 （3）阻塞—>运行 （4）运行—>终止 （5）启动—>运行
- 3、请列出所有可能引发进程调度的事件或者行为。
- 4、在非抢占式的多道批处理系统中有三个进程A\B\C（先后顺序）。其中A计算5s后需要写一个文件，文件操作10s，接着再计算5s结束；B计算8s后需要C传给它一个整数才能继续计算8s结束；C计算6s后后通过共享内存传一个结果给B，然后再计算6s结束。请描述操作系统是如何控制这3个进程在CPU上运行的，说明它们随时间的状态变化。画图说明。