进程控制

Linux
Android
Linux
OpenStack
Mac OS
Windows

进程控制

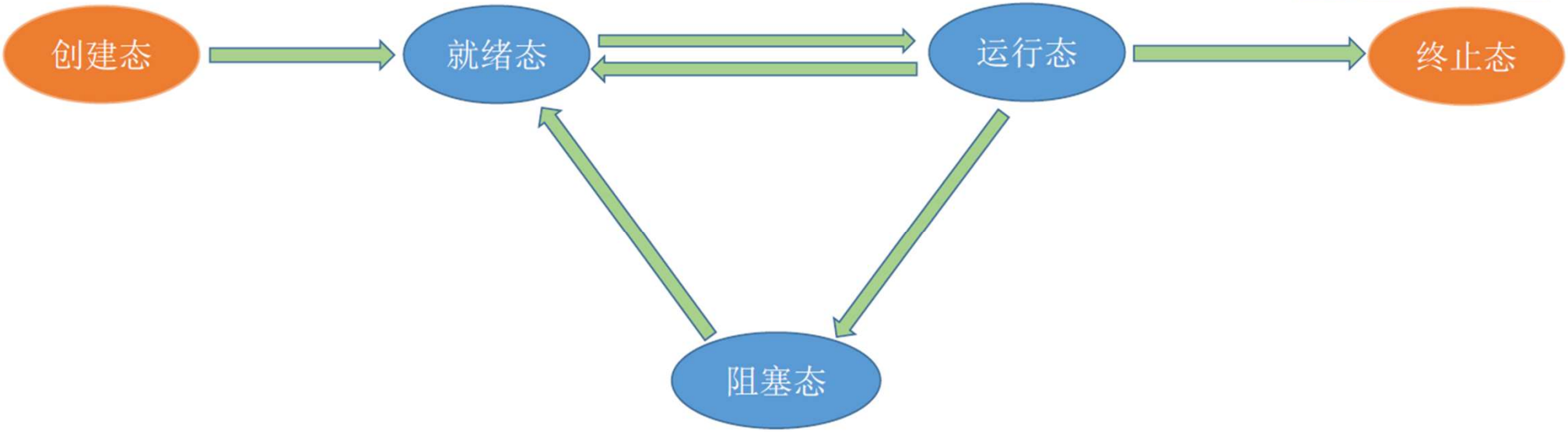
本讲内容

1. 进程控制原语
2. 进程的创建
3. 进程的阻塞和唤醒
4. 进程的撤消
5. 进程的挂起和激活

什么是进程控制？

进程控制的主要功能是对系统中的所有进程实施有效的管理，它具有创建新进程、撤销已有进程、实现进程状态转换等功能。

简化理解：反正进程控制就是要实现进程状态转换

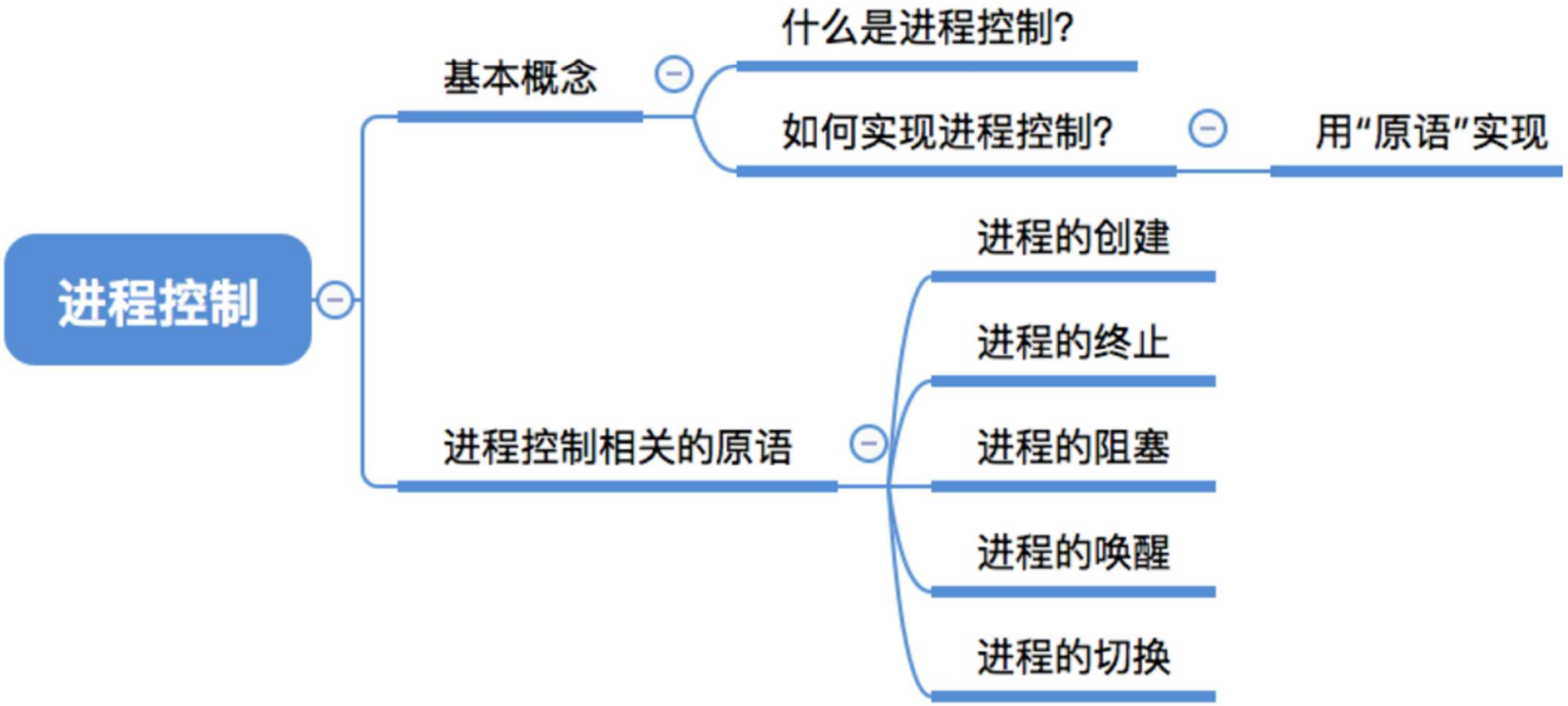


进程控制原语

1 进程的控制

- 含义：系统使用一些具有特定功能的程序段来创建、撤消进程以及完成进程在各状态间转换
- 目标：实现多进程高效率并发执行、协调和共享资源的目的
- 类型：创建进程、阻塞进程、唤醒进程、挂起进程、激活进程和撤销进程等

知识总览


进程控制
基本概念
什么是进程控制?
如何实现进程控制?
用“原语”实现
进程控制相关的原语
进程的创建
进程的终止
进程的阻塞
进程的唤醒
进程的切换

进程控制原语

2 原语

- 进程控制和管理功能是由原语来实现
- 原语是在管态下执行、完成系统特定功能的程序

  管态又叫特权态，系统态或核心态

- 原语执行过程中不允许被中断，强调原子性

  例如说，进程创建，要么创建/要么不创建，不能创建到一半被打断

- 保证原语执行过程中不被中断，采用屏蔽中断的方式

用“原语”实现

如何实现进程控制？

计算机系统的层次结构

原语是一种特殊的程序，它的执行具有原子性。也就是说，这段程序的运行必须一气呵成，不可中断

用“原语”实现

原语的执行具有“原子性”，一气呵成

如何实现进程控制？

思考：为何进程控制（状态转换）的过程要“一气呵成”？

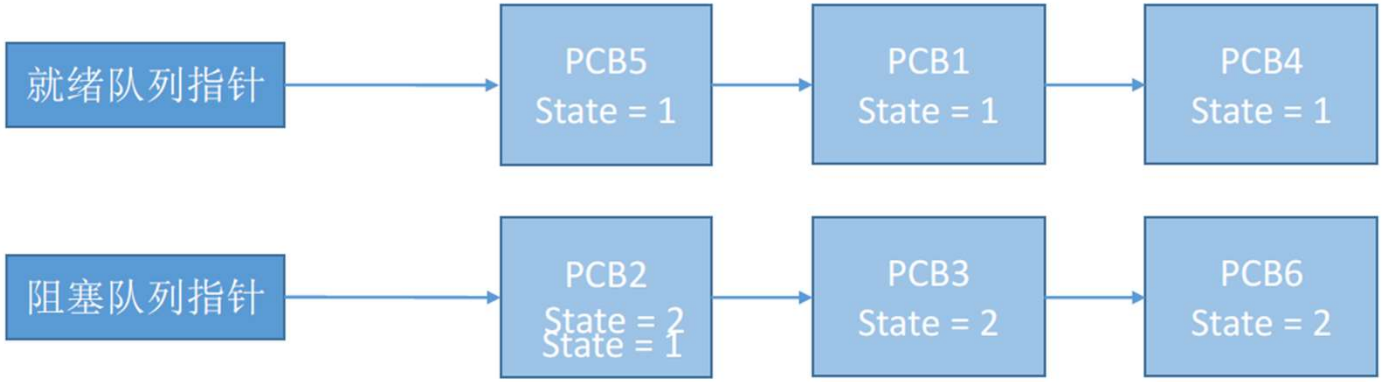
用“原语”实现

如何实现进程控制？

原语的执行具有“原子性”，一气呵成

思考：为何进程控制（状态转换）的过程要“一气呵成”？

Eg：假设PCB中的变量 state 表示进程当前所处状态，1表示就绪态，2表示阻塞态...



假设此时进程2等待的事件发生，则操作系统中，负责进程控制的内核程序至少需要做这样两件事：

①将PCB2的 state 设为 1

②将PCB2从阻塞队列放到就绪队列

完成了第一步后收到中断信号，那么PCB2的state=1，但是它却被放在阻塞队列里

用“原语”实现

如何实现进程控制？

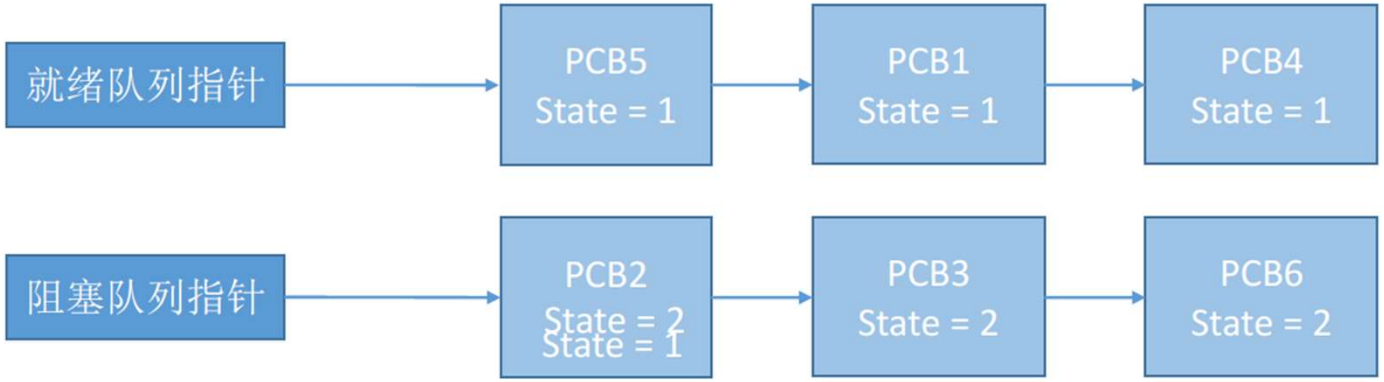
原语的执行具有“原子性”，一气呵成

思考：为何进程控制（状态转换）的过程要“一气呵成”？

如果不能“一气呵成”，就有可能导致操作系统中的某些关键数据结构信息不统一的情况，这会影响操作系统进行别的管理工作

可以用“原语”来实现“一气呵成”啊汪！

Eg：假设PCB中的变量 state 表示进程当前所处状态，1表示就绪态，2表示阻塞态...



假设此时进程2等待的事件发生，则操作系统中，负责进程控制的内核程序至少需要做这样两件事：

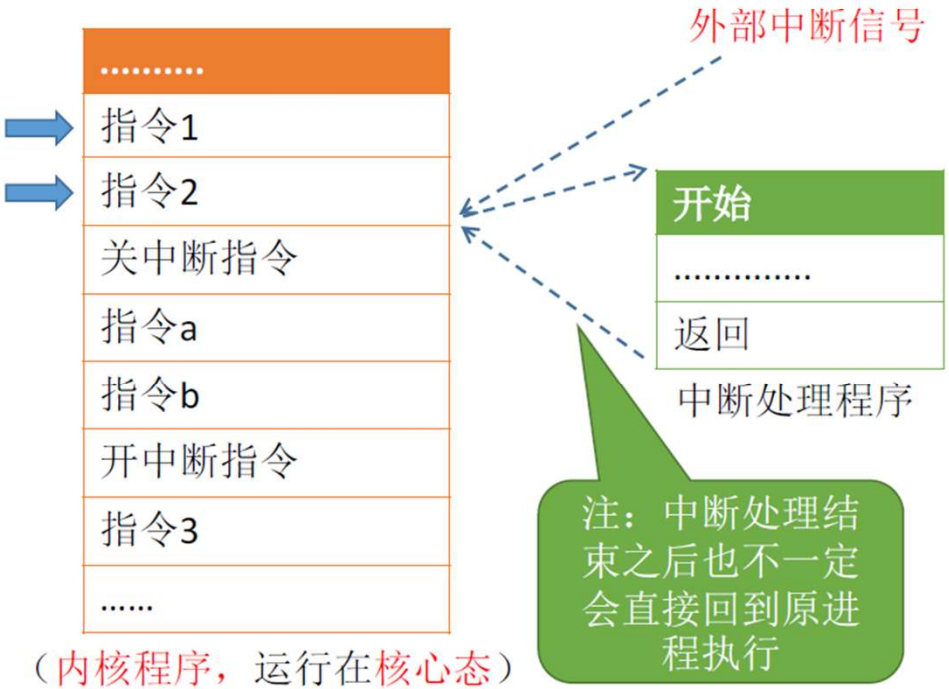
①将PCB2的 state 设为 1

②将PCB2从阻塞队列放到就绪队列

完成了第一步后收到中断信号，那么PCB2的state=1，但是它却被放在阻塞队列里

如何实现原语的“原子性”？

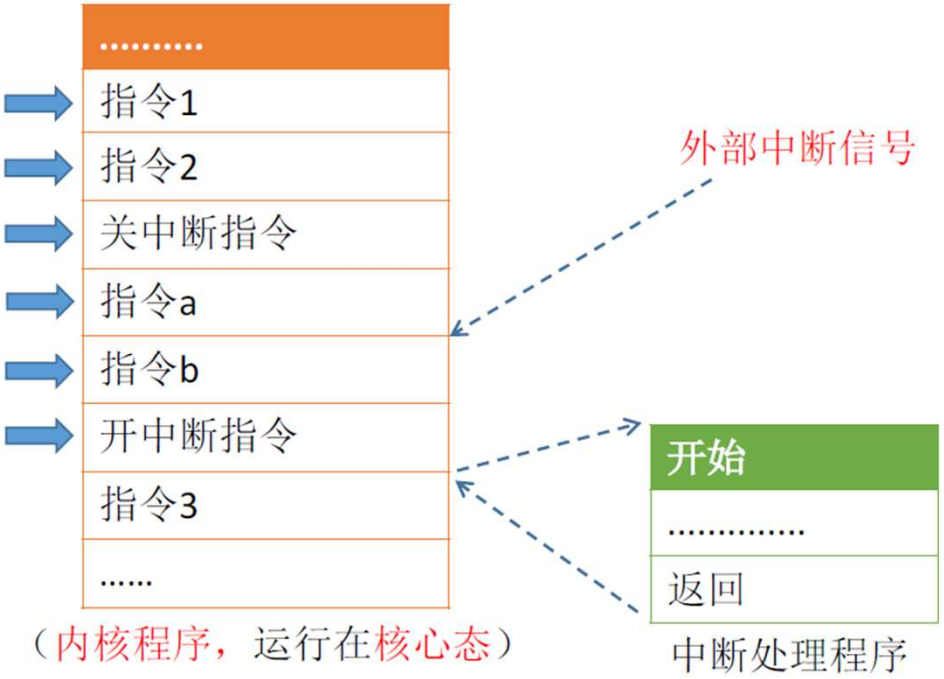
原语的执行具有原子性，即执行过程只能一气呵成，期间不允许被中断。
可以用 “关中断指令”和“开中断指令”这两个特权指令实现原子性



正常情况：CPU每执行完一条指令都会例行检查是否有中断信号需要处理，如果有，则暂停运行当前这段程序，转而执行相应的中断处理程序。

如何实现原语的“原子性”？

原语的执行具有原子性，即执行过程只能一气呵成，期间不允许被中断。
可以用 “关中断指令”和“开中断指令”这两个特权指令实现原子性



（内核程序，运行在核心态）

CPU执行了关中断指令之后，就不再例行检查中断信号，直到执行开中断指令之后才会恢复检查。

这样，关中断、开中断 之间的这些指令序列就是不可被中断的，这就实现了“原子性”

思考：如果这两个特权指令允许用户程序使用的话，会发生什么情况？

进程控制

本讲内容

1. 进程控制原语
2. 进程的创建
3. 进程的阻塞和唤醒
4. 进程的撤消
5. 进程的挂起和激活

进程的创建

1 创建原因 为什么要创建进程、引起进程创建的事件

用户提交作业

就是从外存挑选一个程序，把它放入内存中运行

用户在终端上登录

系统创建服务进程

进程孵化子进程

父进程创建子进程的过程

进程的创建

2 创建过程

1. 申请空白的PCB

- 系统从PCB池中取一个空白PCB

2. 为新进程分配所需资源

- 为新进程的映像分配地址空间，传递环境变量，构造共享地址空间
- 为新进程分配内存等各种资源

进程的创建

2 创建过程

3. 初始化PCB

- 查找辅存，找到进程正文段并装到正文区
- 初始化进程控制块，分配进程标识符，初始化PSW

4.将PCB插入就绪队列

PSW即程序状态字(Program Status Word)

- 将进程加入就绪进程队列，投入运行

新建态-> 就绪态

- 通知操作系统的记账程序、监控程序等

进程的创建

3 Linux进程创建

- 0进程是在系统引导时被创建的；系统初启时，0进程创建1进程；0进程变成对换进程，1进程成为始祖进程。
- 进程利用fork( )创建其子进程，形成一棵进程树；系统中除0进程外的所有进程都是用fork( )创建的。

进程间的关系是树形结构

- fork( ) 的源代码参见：

/USR/SRC/LINUX/KERNEL/FORK.C

进程的创建

3 Linux进程创建

实例

- 父进程创建两个子进程，系统中有一个和两个子进程活动。
- 每一个进程在屏幕上显示不同的字符串：

  子进程P1显示'daughter …’

  子进程P2显示'son ……’

  父进程显示 'parent ……'

进程的创建

3 Linux进程创建

实例

fork( ) 创建一个新进程

系统调用格式： pid=fork( )

参数定义： int fork( )

进程的创建

3 Linux进程创建

实例

fork( )返回值意义

- 0：在子进程中，pid变量保存的fork( )返回值为0，表示当前进程是子进程
- >0：在父进程中，pid变量保存的fork( )返回值为子进程的id值（进程唯一标识符）
- -1：创建失败

进程的创建

3 Linux进程创建

- 如果fork( )调用成功，它向父进程返回子进程的PID，并向子进程返回0，即fork( )被调用了一次，但返回两次值。
- OS在内存中创建一个新进程，所建的新进程是调用fork( )的父进程的副本。
- 子进程继承了父进程的许多特性，并具有与父进程完全相同的用户级上下文。

进程的创建

3 Linux进程创建

实例

这段C代码创建了三个进程：父进程、女儿进程（P1）和儿子进程（P2）。
以下是代码的分析：

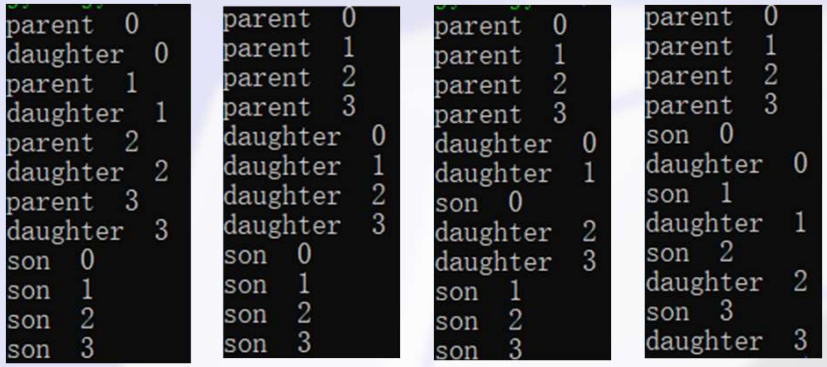
1.父进程使用 fork() 系统调用创建女儿进程（P1）。
2.女儿进程(P1)通过一个循环运行，打印四次 "daughter"。
3.父进程再次使用 fork() 调用创建儿子进程(P2)。
4.儿子进程(P2)通过一个循环运行，打印四次 "son"。
5.父进程通过一个循环运行，打印四次 "parent"。

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  int main( ) {
5    int p1, p2, i;
6    while ((p1=fork()) == -1);          /*创建子进程P1*/
7    if (p1==0) {
8      for (i=0; i<4; i++) {
9        printf ("daughter  %d\n", i);
10     }
11   } else {
12     while ((p2=fork()) == -1);      /*创建子进程P2*/
13     if (p2==0) {
14       for (i=0; i<4; i++) {
15         printf("son  %d\n", i);
16       }
17     } else {
18       for (i=0; i<4; i++) {
19         printf("parent  %d\n", i);
20       }
21     }
22   }
23   return 0;
24 }
```

进程的创建

这些进程并行运行，它们的执行顺序不被保证。程序的输出取决于操作系统对这些进程的调度。每次运行程序时，操作系统可能以不同的顺序调度这些进程，因此导致不同的输出顺序。

此外，如果没有适当的同步机制，使用 fork() 可能导致子进程与父进程并发执行，从而产生交错或混合的输出。如果要确保特定的执行顺序，您需要使用同步机制，比如信号量、互斥锁或其他进程间通信方法。




```
.h>
.h>
d.h>

rk()) == -1);          /*创建子进程P1*/

<4; i++) {
daughter  %d\n", i);


Fork()) == -1);       /*创建子进程P2*/
13      if (p2==0) {
14        for (i=0; i<4; i++) {
15          printf("son  %d\n", i);
16        }
17      } else {
18        for (i=0; i<4; i++) {
19          printf("parent  %d\n", i);
20        }
21      }
22    }
23    return 0;
24  }
```

进程控制

本讲内容

1. 进程控制原语
2. 进程的创建
3. 进程的阻塞和唤醒
4. 进程的撤消
5. 进程的挂起和激活

进程的阻塞和唤醒

1 进程阻塞 运行态 --> 阻塞态（自己）

- 阻塞原语在进程期待某事件发生，但没有发生时，或所需资源尚不具备时，被该进程调用来阻塞自己

引起进程阻塞的事件：1. 需要等待系统分配某种资源

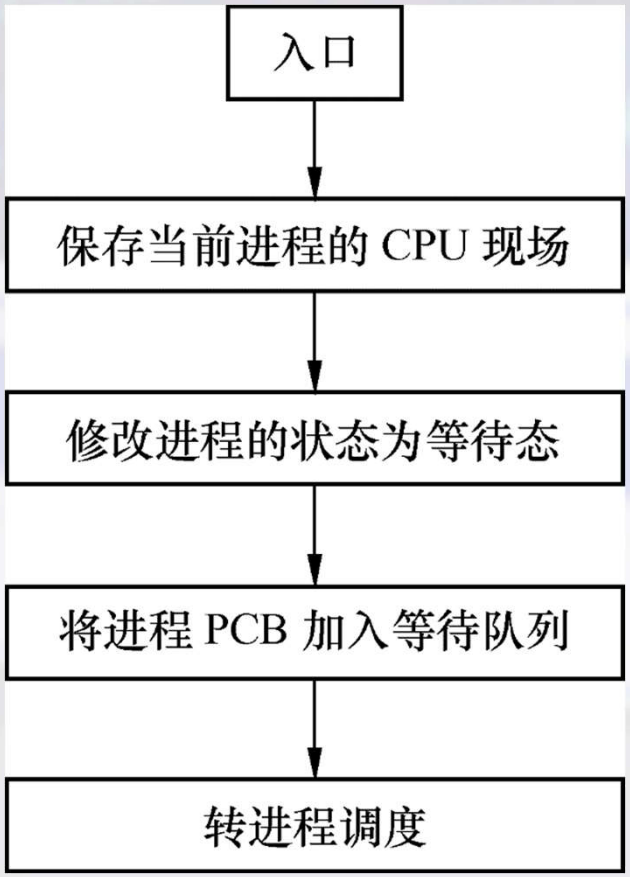
2. 需要等待相互合作的其他进程完成工作

- 阻塞常常是自我阻塞，就是我需要资源，但是资源获不到，然后我就自我阻塞

进程的阻塞和唤醒

1 进程阻塞 运行态 --> 阻塞态

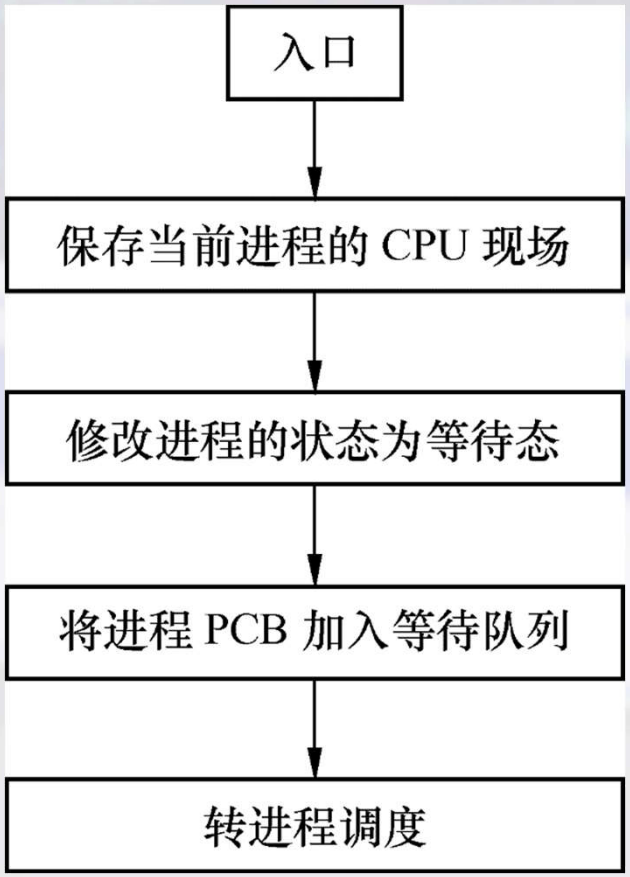
- 阻塞进程时，先中断处理器，保存该进程的CPU现场信息到内存PSW
- 将被阻塞进程置“阻塞”状态后，插入等待队列中



进程的阻塞和唤醒

1 进程阻塞 运行态 --> 阻塞态

- 进程调度程序选择新的就绪进程投入运行
- 完成一个进程切换的过程



进程的阻塞和唤醒

2 进程唤醒 阻塞态 --> 就绪态（其他）

- 引起进程唤醒的事件：等待的事件发生

  一个进程因为什么事件被阻塞，就应该被同一个事件唤醒

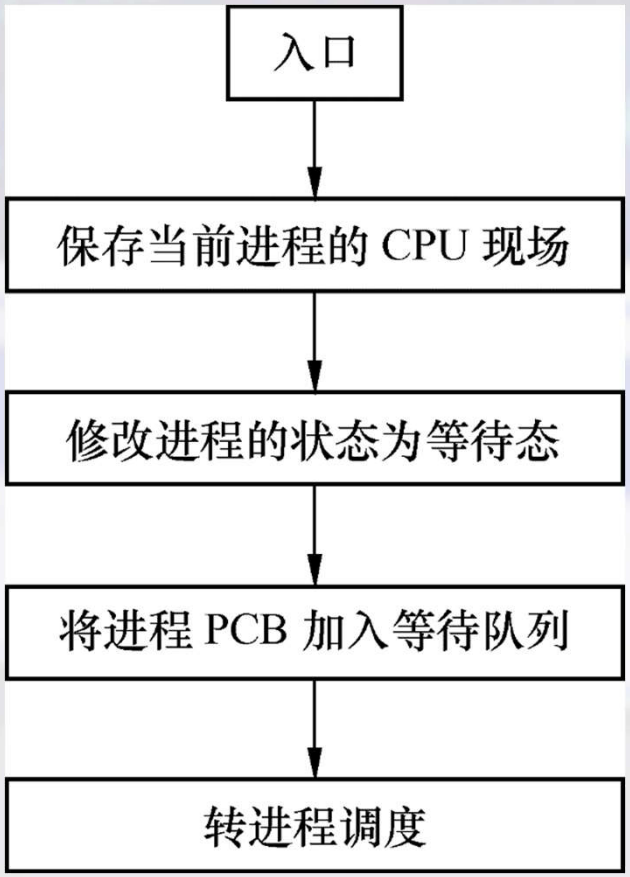
- 系统唤醒进程：系统进程统一控制事件的发生并将事件发生的消息通知等待进程，使得该进程进入就绪队列

- 事件发生唤醒进程：事件发生进程和被唤醒进程之间是合作关系，唤醒原语既可被系统进程调用，也可被事件发生进程调用

进程的阻塞和唤醒

2 进程唤醒 阻塞态 --> 就绪态

- 在等待队列中找到PCB
- 将PCB从等待队列移除，设置进程为就绪态
- 将PCB插入就绪队列，等待被调度



进程控制

本讲内容

1. 进程控制原语
2. 进程的创建
3. 进程的阻塞和唤醒
4. 进程的撤消
5. 进程的挂起和激活

进程的撤消

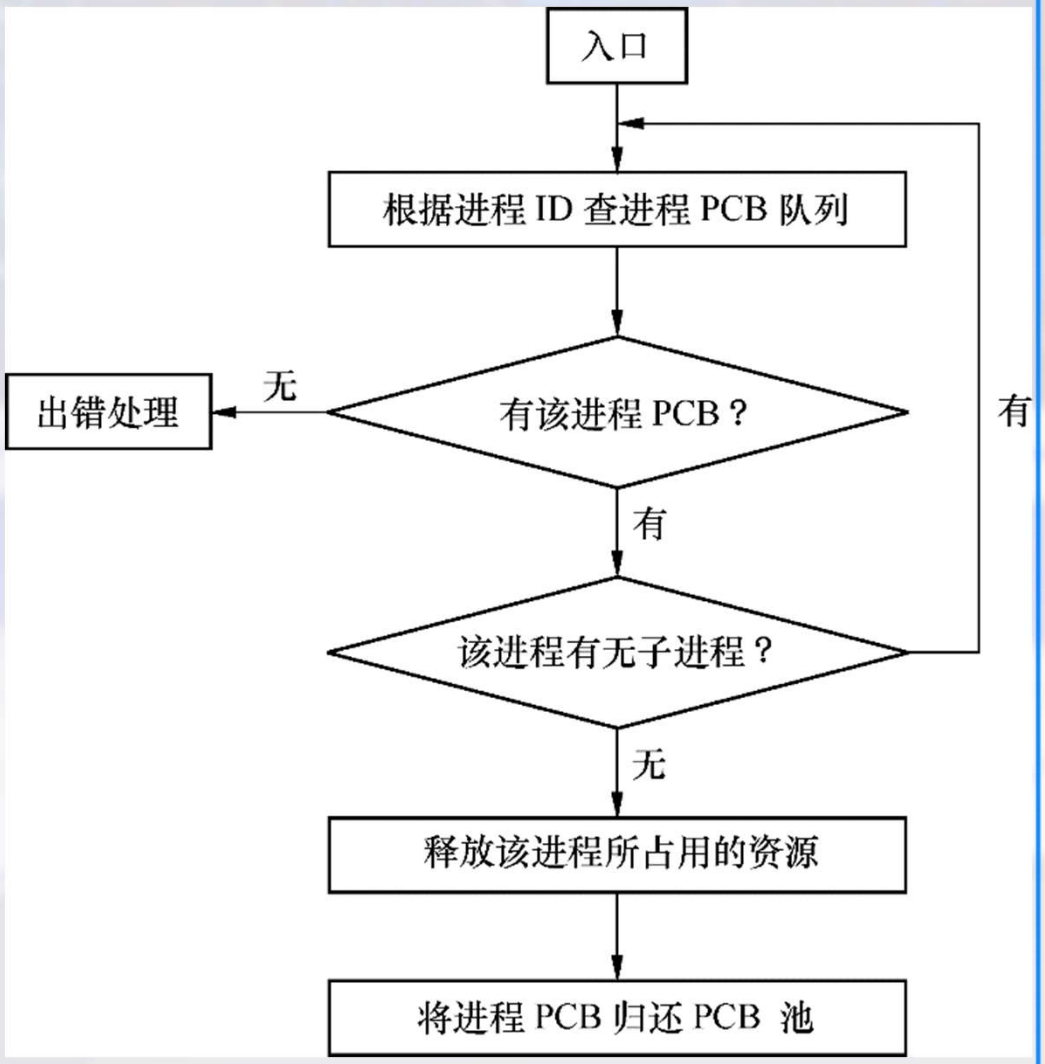
1 撤销缘由 就绪/阻塞/运行态 --> 终止态 --> 无

- 进程已完成所要求的功能而正常终止
- 由于某种错误导致非正常终止
- 祖先进程要求撤消某个子孙进程

进程的撤消

2 撤销步骤

- 根据进程标识号，从相应队列中找到它的PCB
- 将该进程拥有的资源归还给父进程或操作系统
- 若该进程拥有子进程，先撤销子孙进程，以防脱离控制
- 撤销进程出队，将它的PCB归还到PCB池



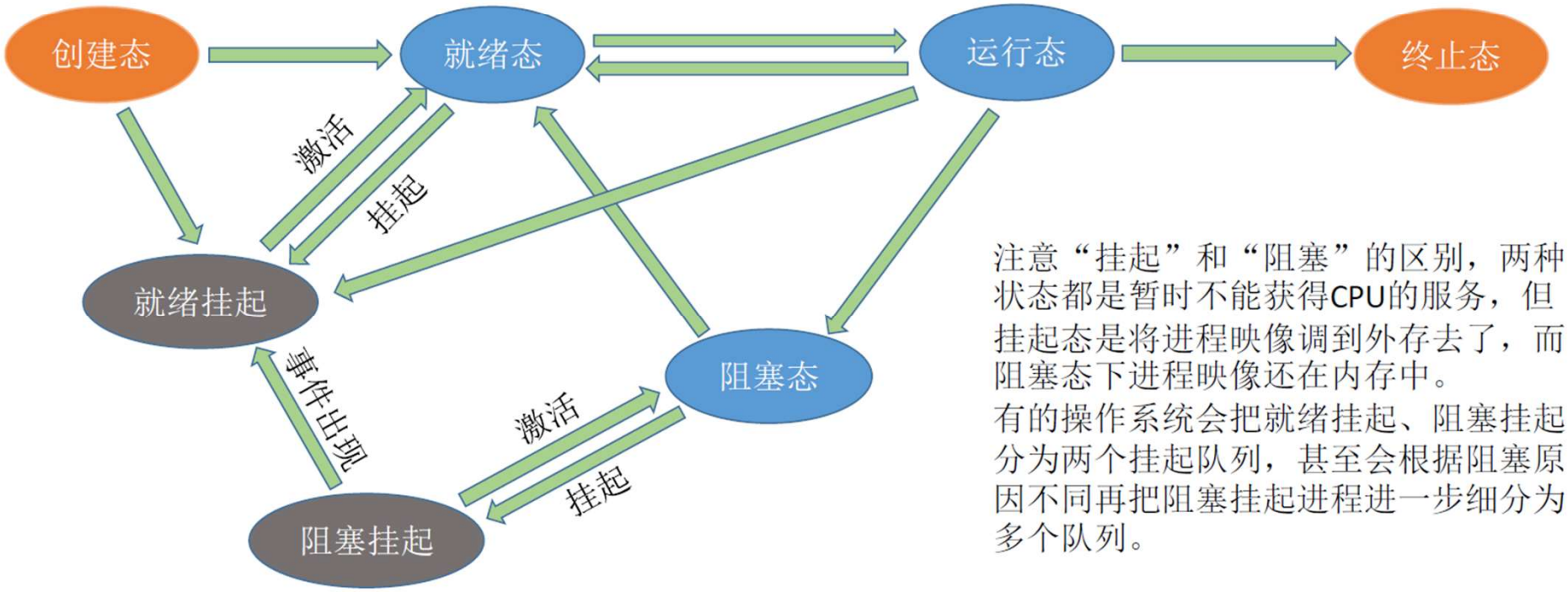
进程控制

本讲内容

1. 进程控制原语
2. 进程的创建
3. 进程的阻塞和唤醒
4. 进程的撤消
5. 进程的挂起和激活

补充知识：进程的挂起态与七状态模型

暂时调到外存等待的进程状态为挂起状态（挂起态，suspend）
挂起态又可以进一步细分为就绪挂起、阻塞挂起两种状态
五状态模型 → 七状态模型

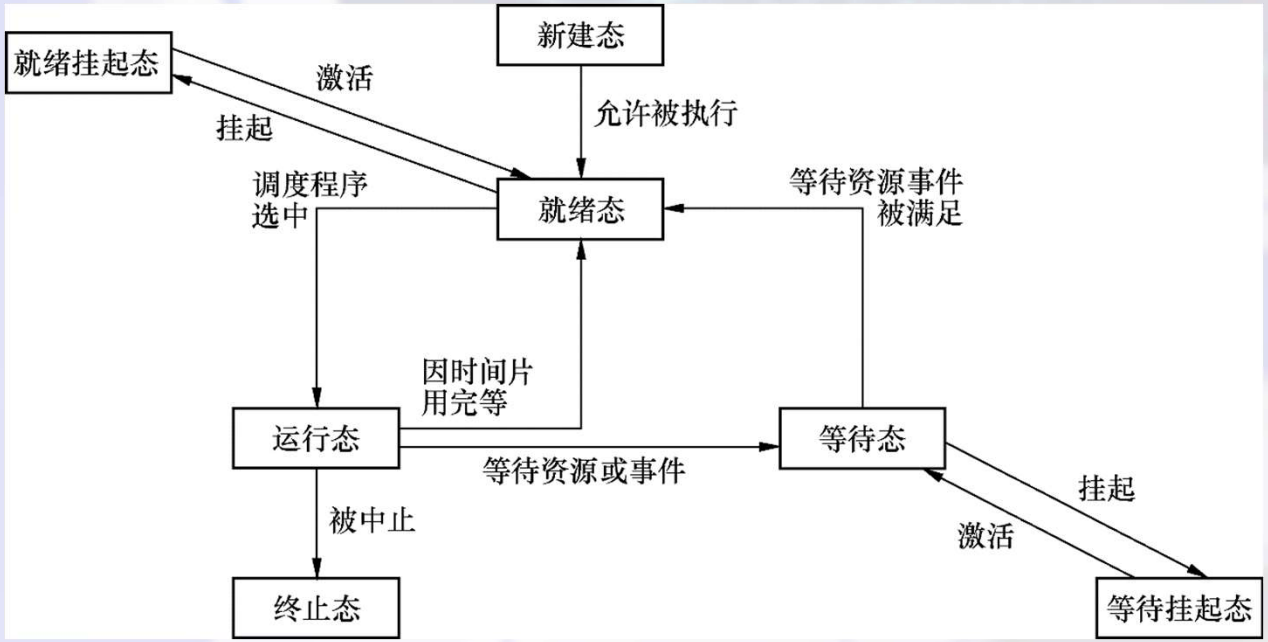


注意“挂起”和“阻塞”的区别，两种状态都是暂时不能获得CPU的服务，但挂起态是将进程映像调到外存去了，而阻塞态下进程映像还在内存中。
有的操作系统会把就绪挂起、阻塞挂起分为两个挂起队列，甚至会根据阻塞原因不同再把阻塞挂起进程进一步细分为多个队列。

进程的状态和转换

3 七种进程状态

五状态进程模型->七状态进程模型

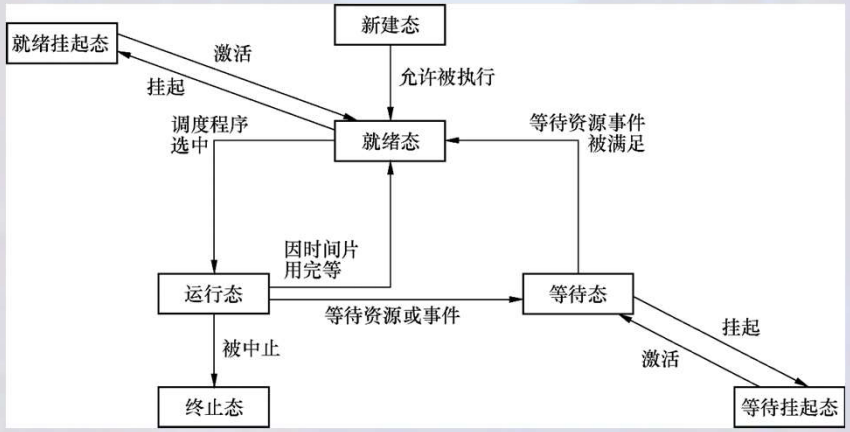


注意“挂起”和“阻塞”的区别，两种状态都是暂时不能获得CPU的服务，但挂起态是将进程映像调到外存去了，而阻塞态下进程映像还在内存中。

有的操作系统会把就绪挂起、阻塞挂起分为两个挂起队列，甚至会根据阻塞原因不同再把阻塞挂起进程进一步细分为多个队列。

进程的挂起和激活

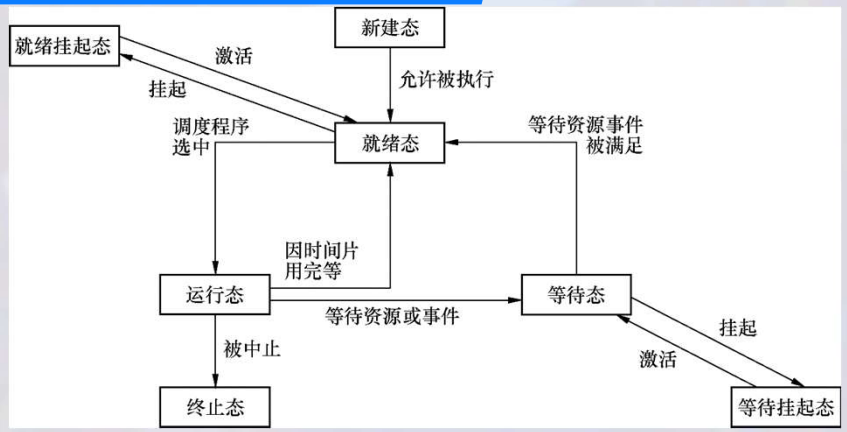
1 挂起过程



- 检查要被挂起进程的状态，若处于就绪态就修改为挂起就绪态，若处于阻塞态，则修改为挂起阻塞态
- 被挂起进程PCB的非常驻部分交换到磁盘对换区
- 不仅要把程序的代码和数据，从内存移到硬盘上，还可能要把进程PCB里一部内容也移到硬盘上
- 为了最大限度的腾出空间，给需要的进程使用

进程的挂起和激活

2 激活过程



- 将进程PCB非常驻部分调进内存，修改状态，挂起等待态改为等待态，挂起就绪态改为就绪态

  把代码和数据调入到内存中

- 挂起原语既可由进程自己也可由其他进程调用，但激活原语却只能由其他进程调用

进程控制

本讲内容

1. 进程控制原语
2. 进程的创建
3. 进程的阻塞和唤醒
4. 进程的撤消
5. 进程的挂起和激活