上海交通大学
SHANGHAI JIAO TONG UNIVERSITY

SJTU

L2-1. 进程

宋卓然

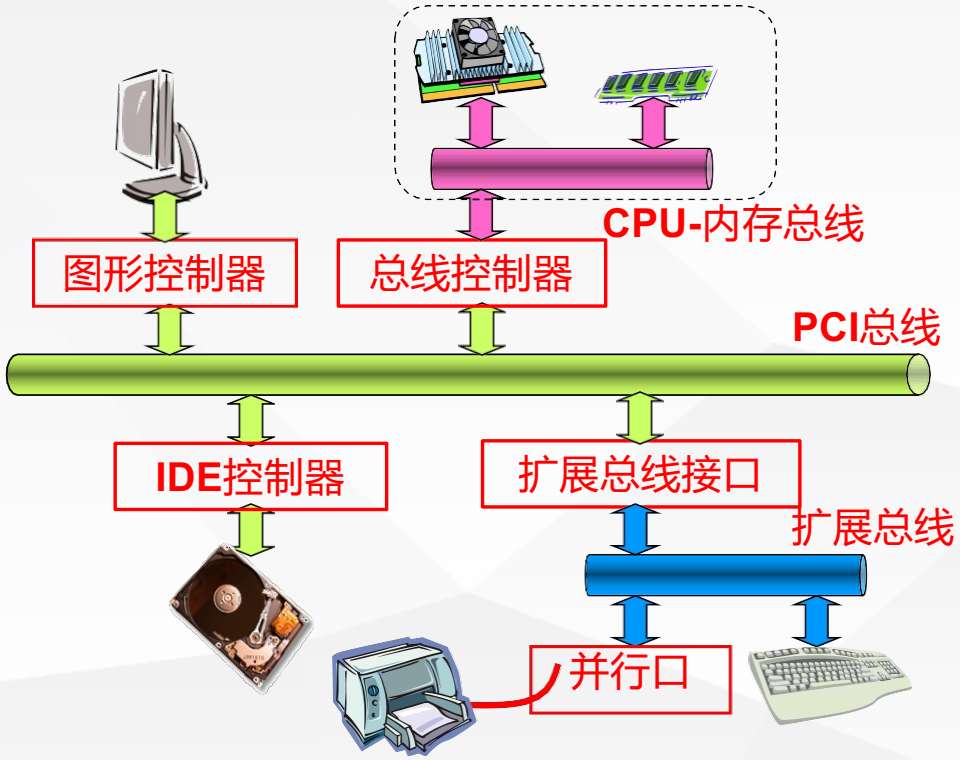
上海交通大学计算机系

songzhuoran@sjtu.edu.cn

饮 水 思 源 • 爱 国 荣 校

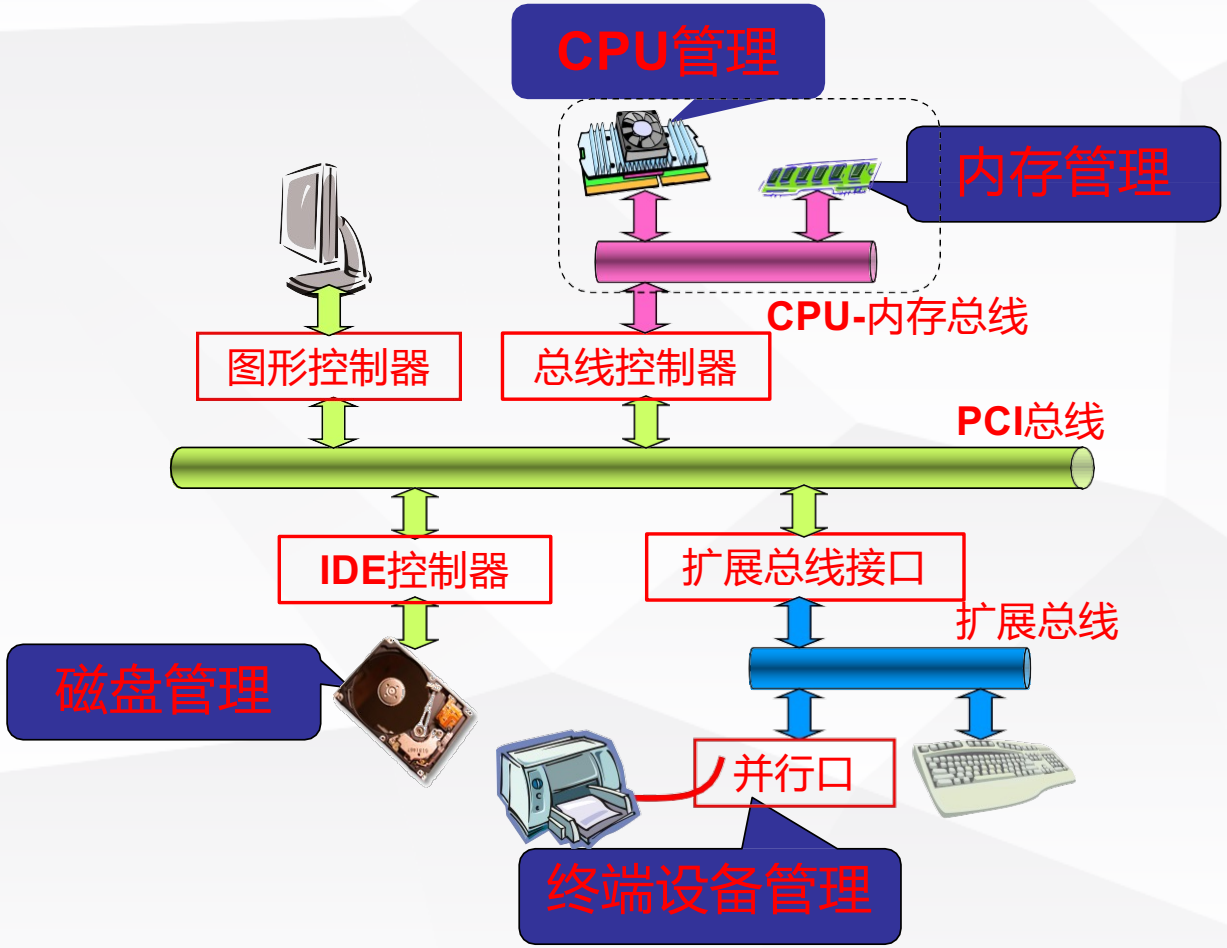
什么是操作系统

- 温故：操作系统要管理硬件，方便我们使用...



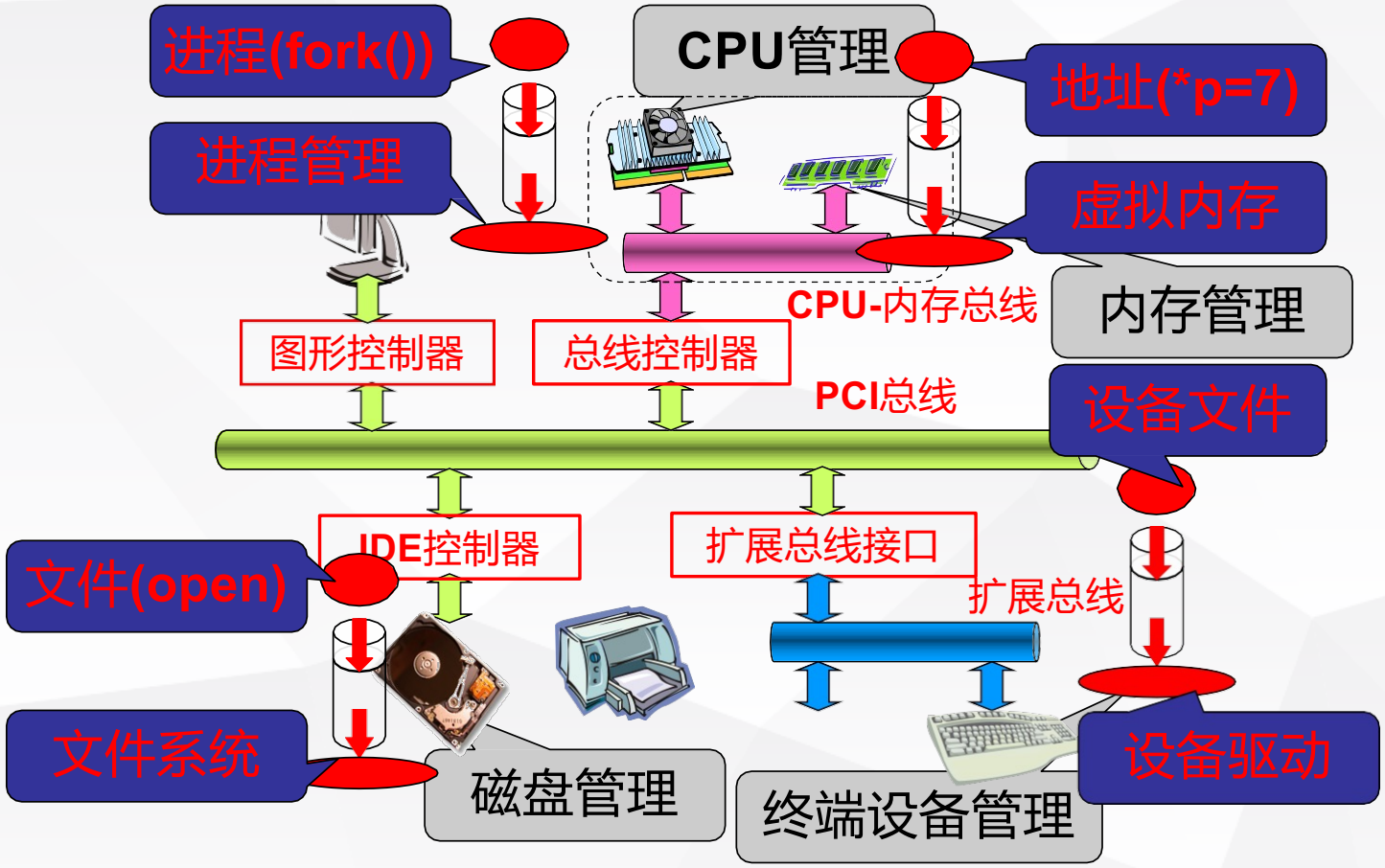
什么是操作系统

- 要管理硬件资源



我们要学什么?再具体一些...

- 方便用户使用硬件资源

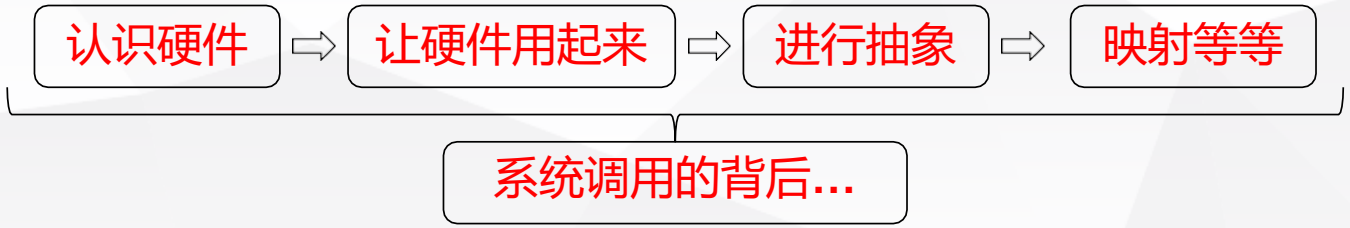


我们要学什么?再再具体一些...

CPU管理
- 认识CPU->CPU管理的直观想法-> CPU到进程的抽象->多进程基本结构->多进程相关问题

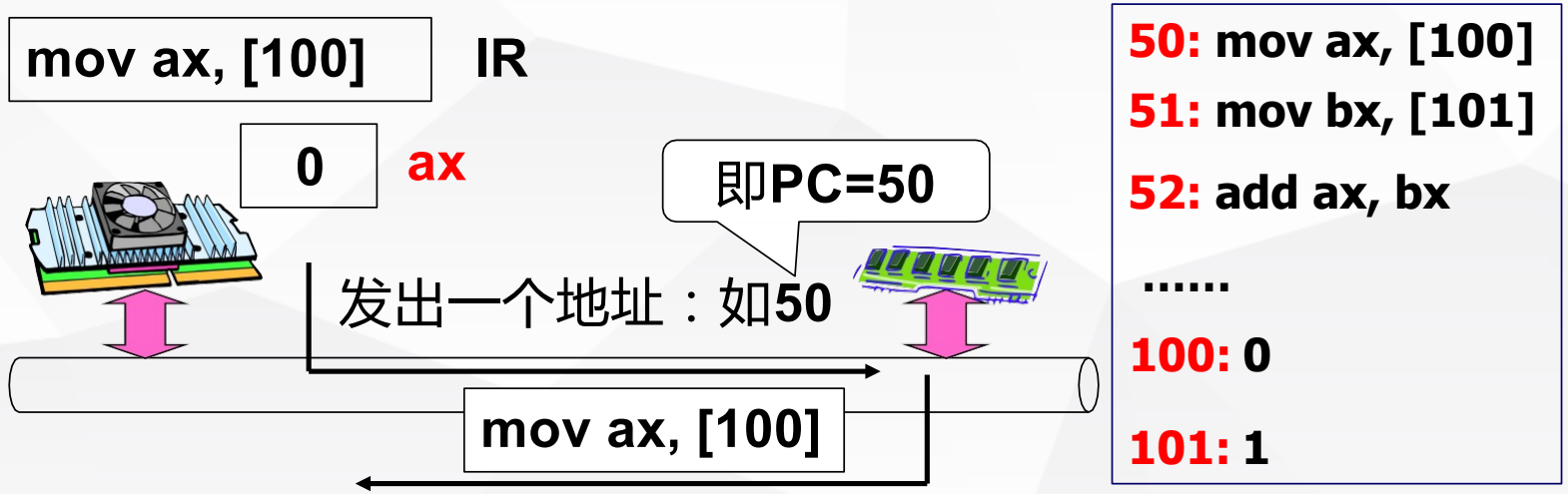
内存管理
- 认识内存->内存管理的直观想法->物理地址到虚拟地址->进程虚拟内存如何产生?

文件管理
- 认识设备认识设备->设备使用的基本结构->从设备到文件的抽象->open、read、write的背后?



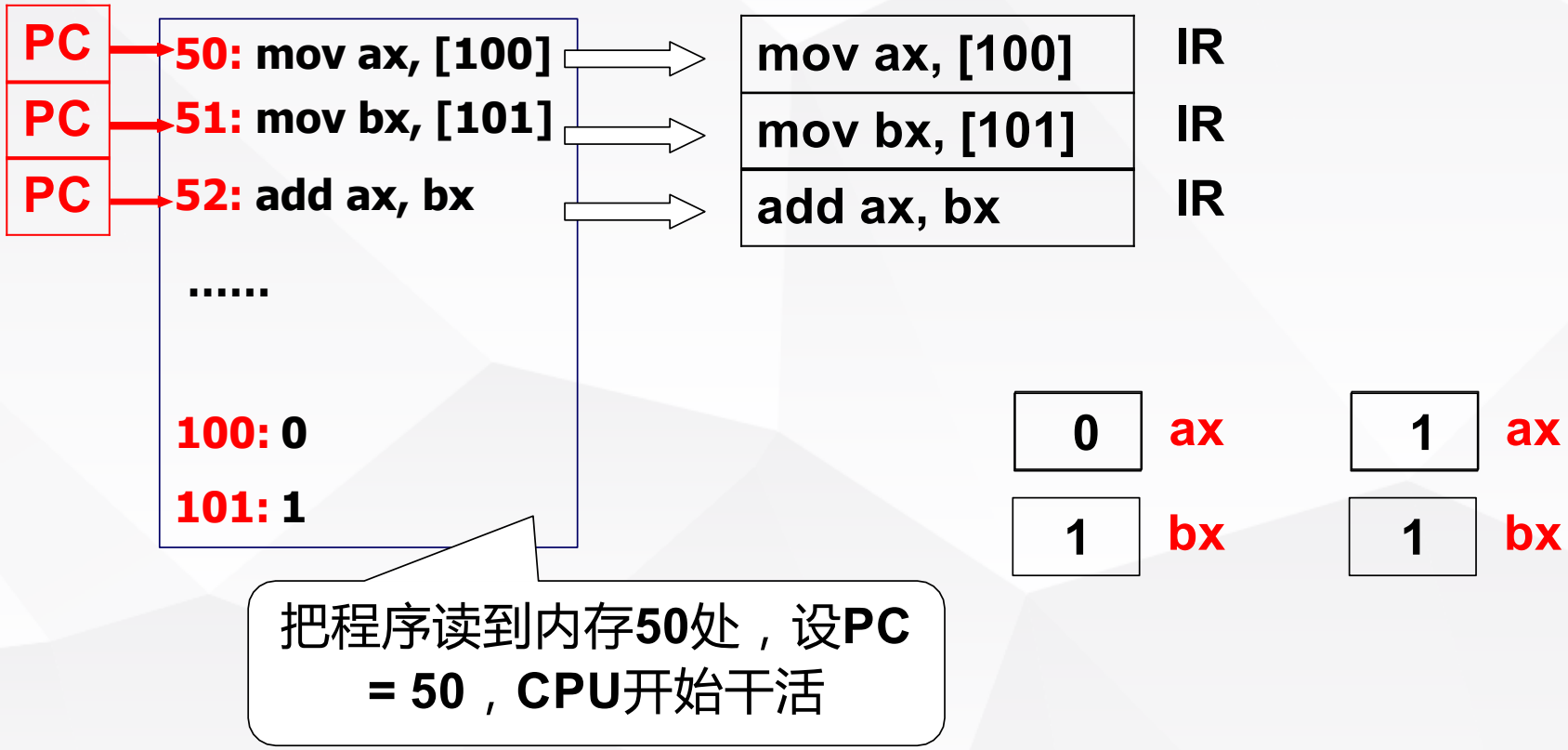
CPU的工作原理

- CPU上电以后发生了什么？
  - 自动的取指—执行
- CPU怎么工作?
- CPU怎么管理?



管理CPU的最直观方法

- 设好PC初值就完事!



看看这样做有没有问题？提出问题

```
int main(int argc, char* argv[])
{
    int  i , to, *fp, sum = 0;
    to = atoi(argv[1]);
    for(i=1; i<=to; i++)
    {
        sum = sum + i;
        fprintf(fp,"%d", sum);
     }
}
```

fprintf用一条其他计算语句代替

```
C:\>sum 10000000
0.015000 seconds
```

$0.015/10^7$

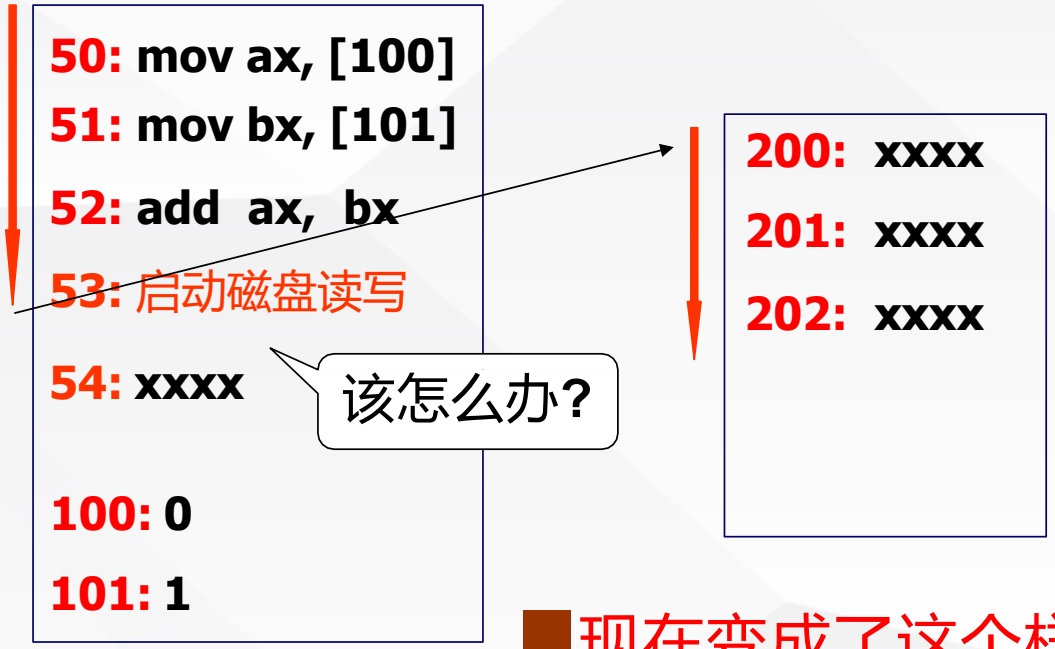
有fprintf

```
C:\>sum 1000
0.859000 seconds
```

$0.859/10^3$

$5.7\times10^5 : 1$

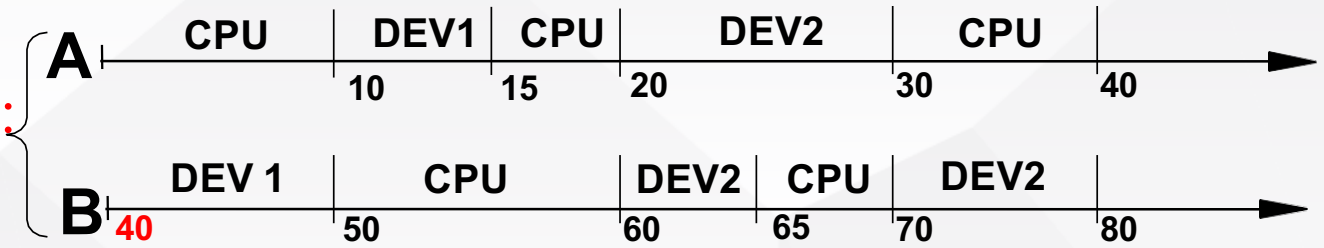
怎么解决?进程切换



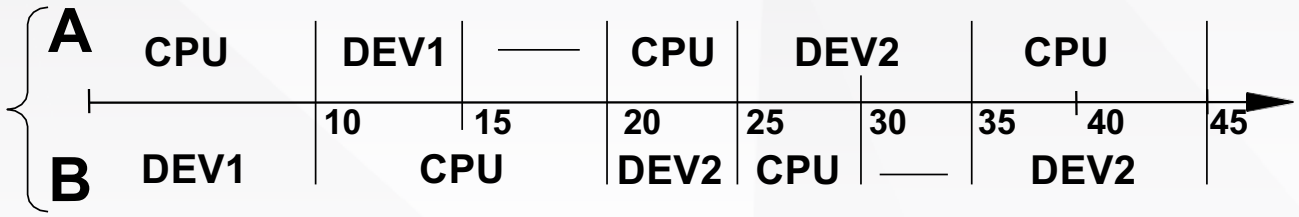
- 现在变成了这个样子

多道程序、交替执行，好东西啊!

单道程序：

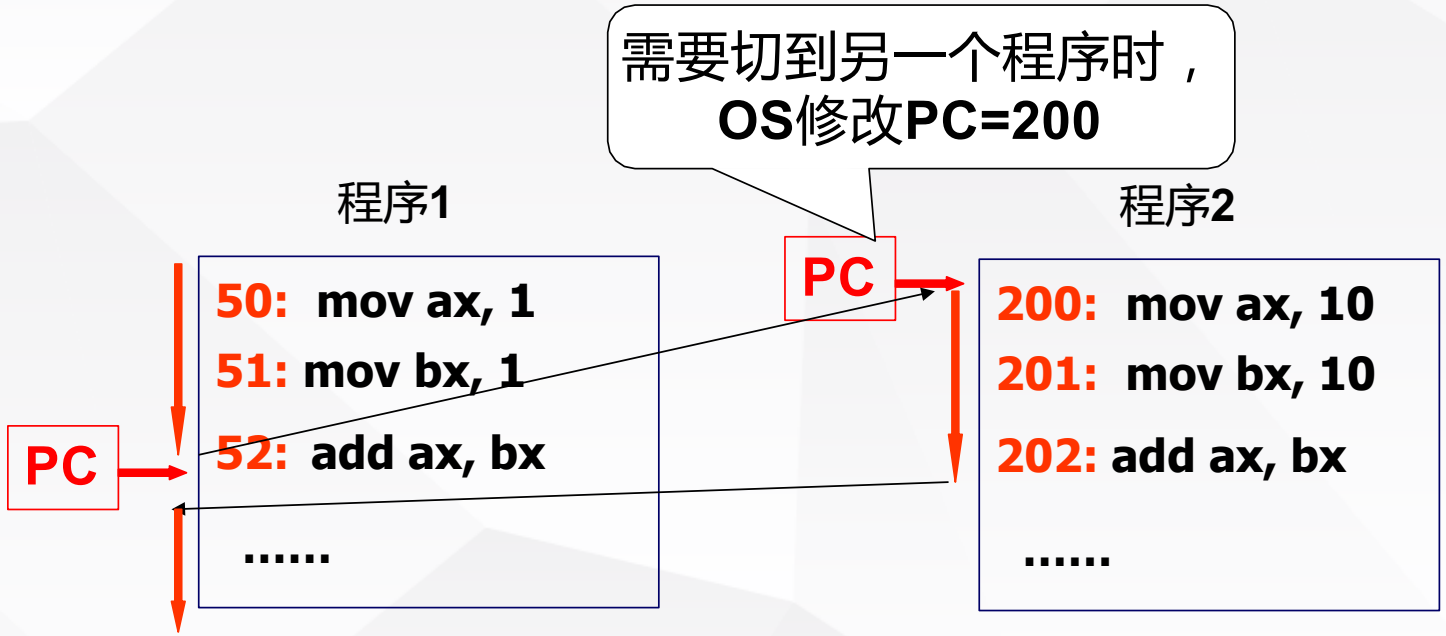


多道程序：



	单道程序	多道程序
CPU利用率	40/80=50%	40/45=89%
DEV1利用率	15/80=18.75%	15/45=33%
DEV2利用率	25/80=31.25%	25/45=56%

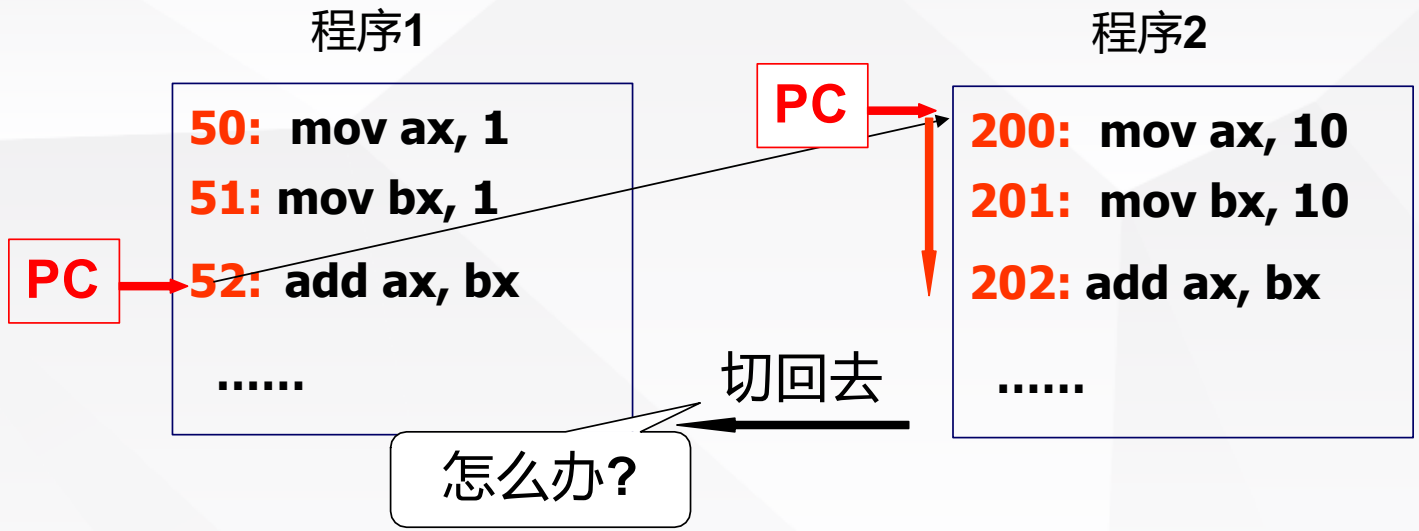
一个CPU面对多个程序？



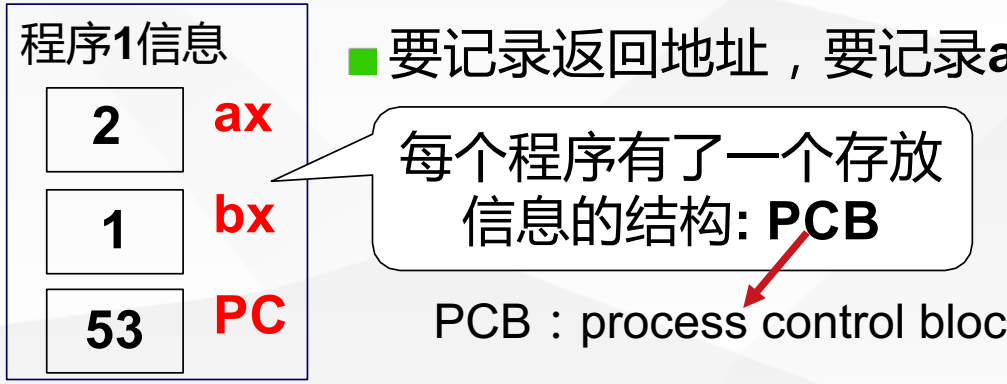
- 一个CPU上交替的执行多个程序：并发
  - 怎么做到？

    PC进行切换

修改寄存器PC就行了吗？



- 运行的程序和静态程序不一样了…



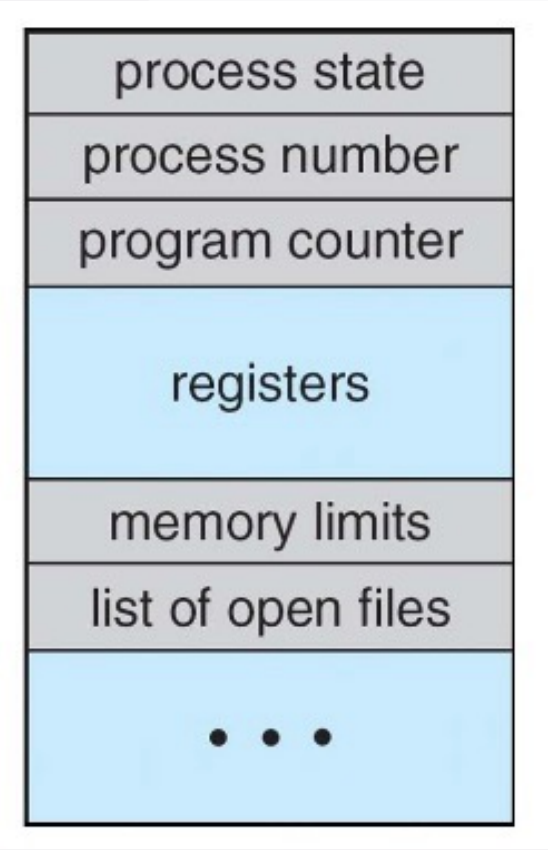
- 要记录返回地址，要记录ax…

每个程序有了一个存放信息的结构: **PCB**

PCB：process control block 进程控制块

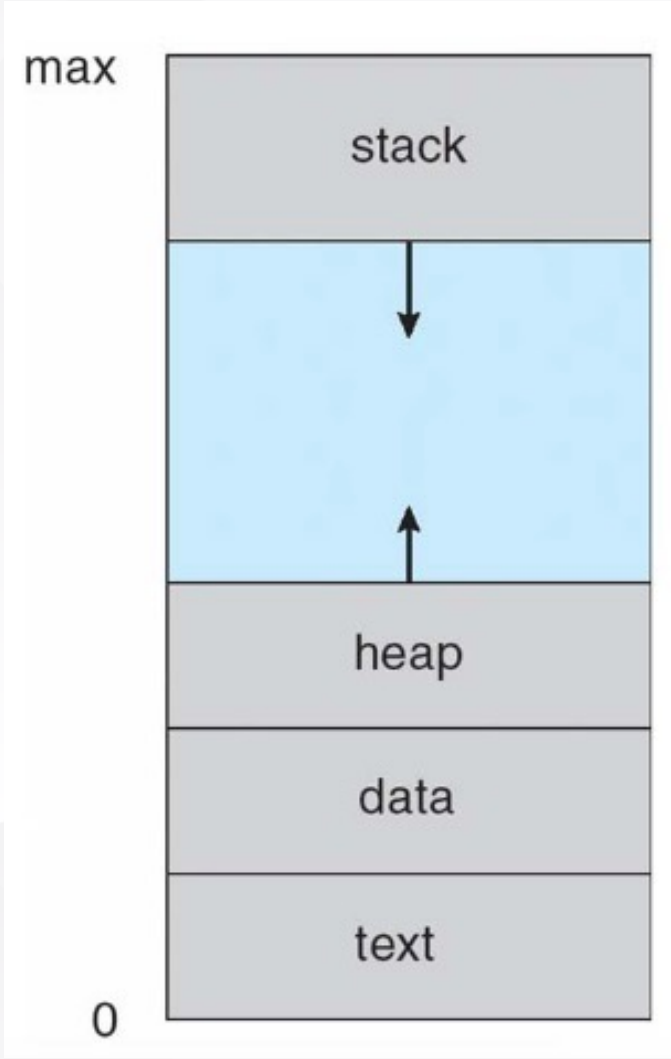
PCB概念

- PCB：process control block 进程控制块，用于表示进程
  - 进程状态
  - 程序计数器（program counter，PC）
  - CPU寄存器
  - CPU调度信息：进程优先级、调度队列的指针
  - 内存管理信息：基地址、页表、段表
  - 记账信息：CPU时间、实际使用时间
  - I/O状态信息：分配给进程的I/O设备列表



进程的概念

- 在执行中的程序
  - 程序代码 text section
  - 全局变量 data section
  - 局部变量 stack
  - 动态分配的内存 heap

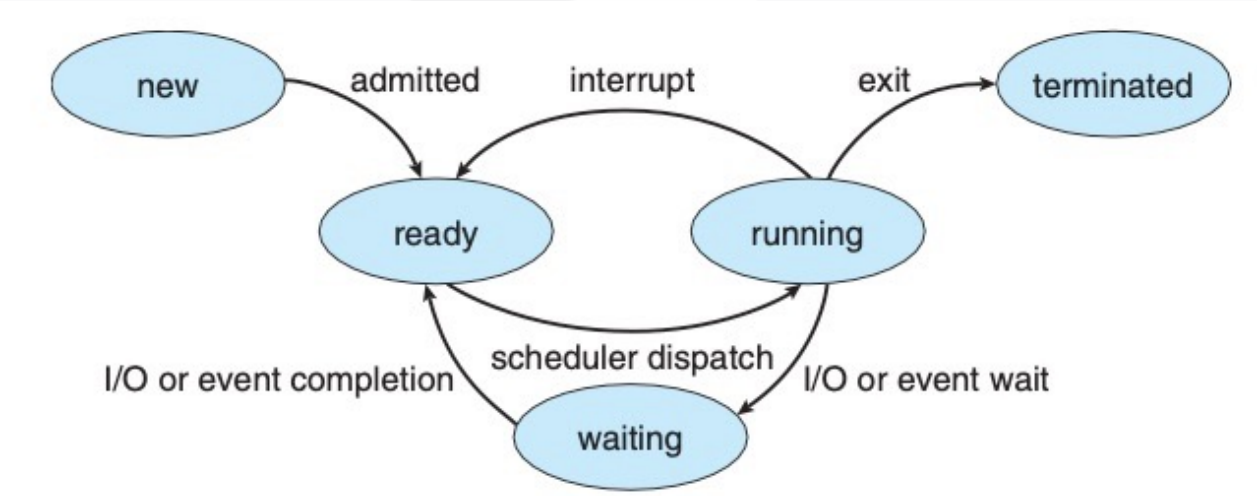


进程的概念

- 进程不是程序
- 程序是被动实体，例如磁盘中的文件
- 进程是活动实体，具有一个程序计数器用于表示下个执行命令和一组相关资源
  - 对于可执行文件，可通过：1.双击；2.命令行，加载其到内存，成为进程
- 虽然两个进程可以与同一程序相关联，但是当作两个单独的执行序列
  - 虽然代码文本段相同，但堆、数据等均不相同

进程状态

- 进程在执行时可能会改变状态
  - 新的
  - 运行
  - 等待
  - 就绪
  - 终止

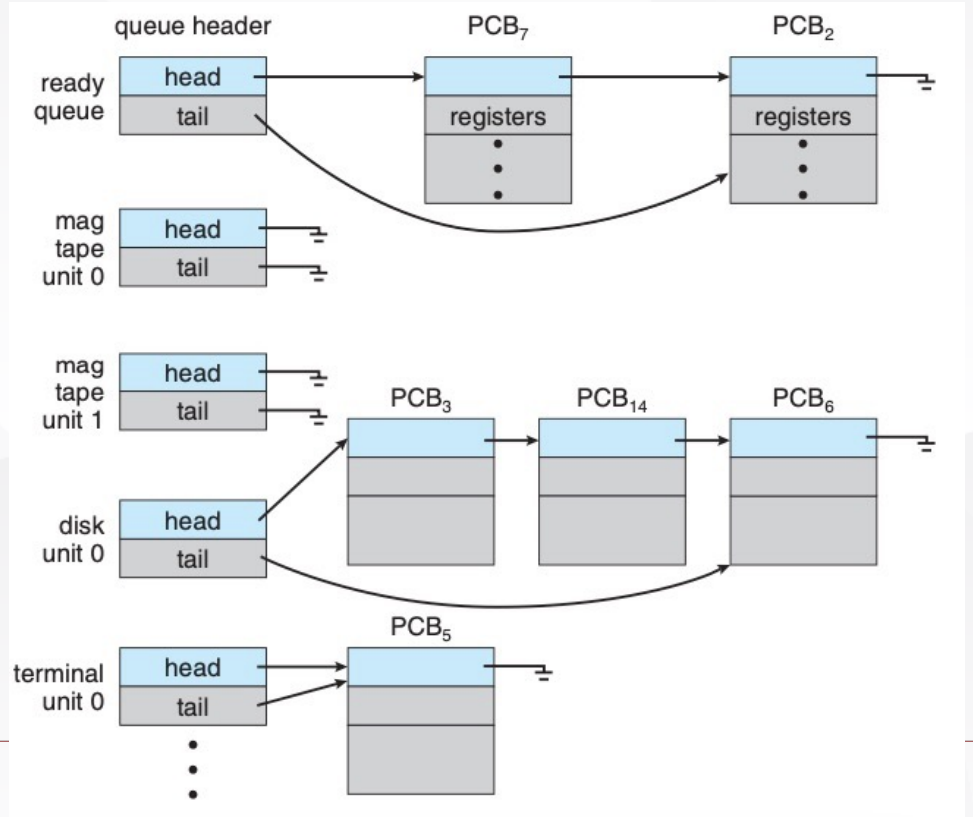


进程调度

- 多道程序设计的目标是， 无论何时都有进程运行， 从而最大化CPU 利用率
- 进程调度器选择一个可用的进程到CPU上执行

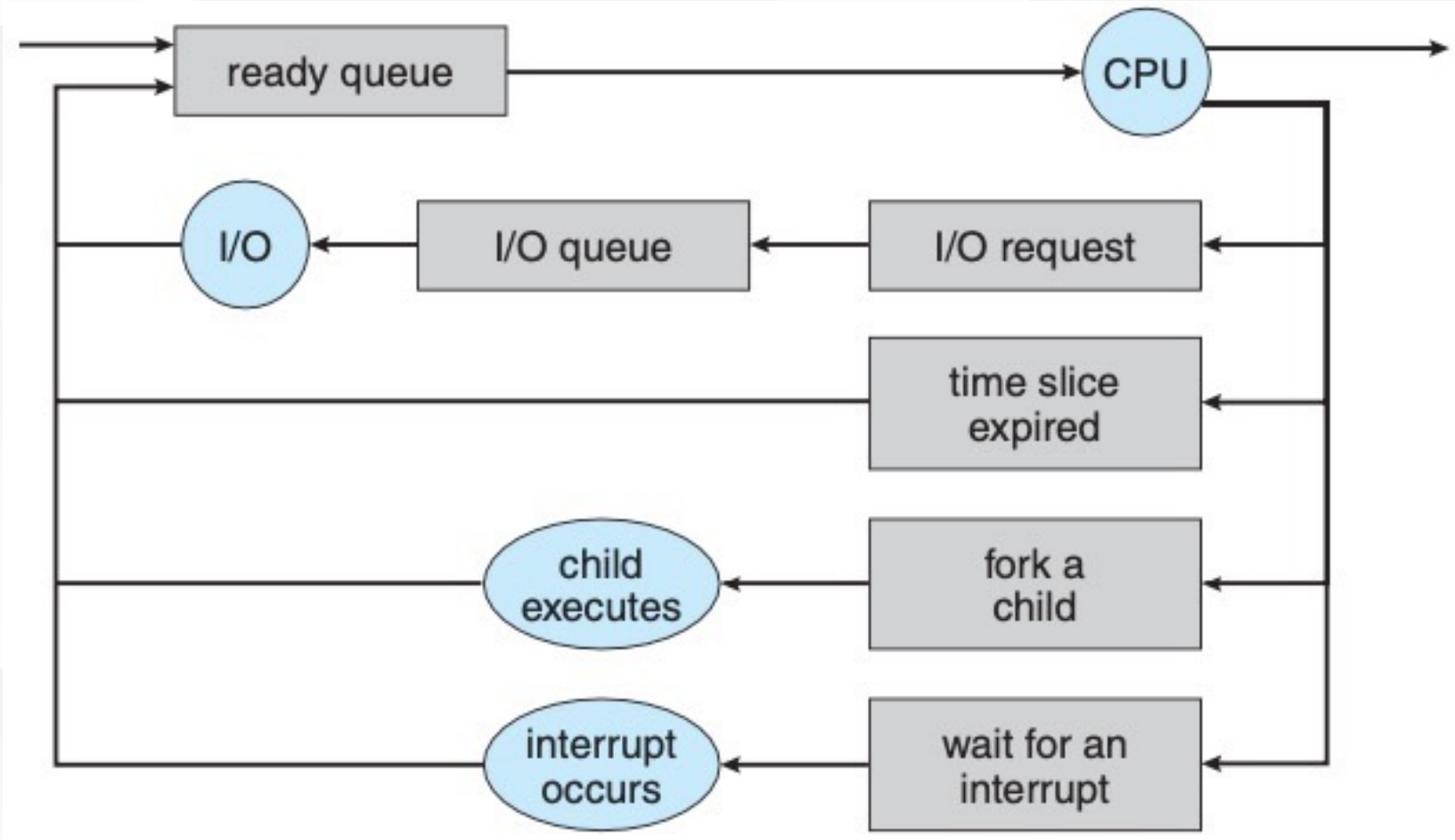
调度队列

- 进程在进入系统时， 会被加到作业队列，这个队列包括系统内的所有进程
  - 就绪队列：驻留在内存中、就绪的进程保存在就绪队列中
  - 设备队列：等待特定设备的进程列表



调度队列

- 队列图

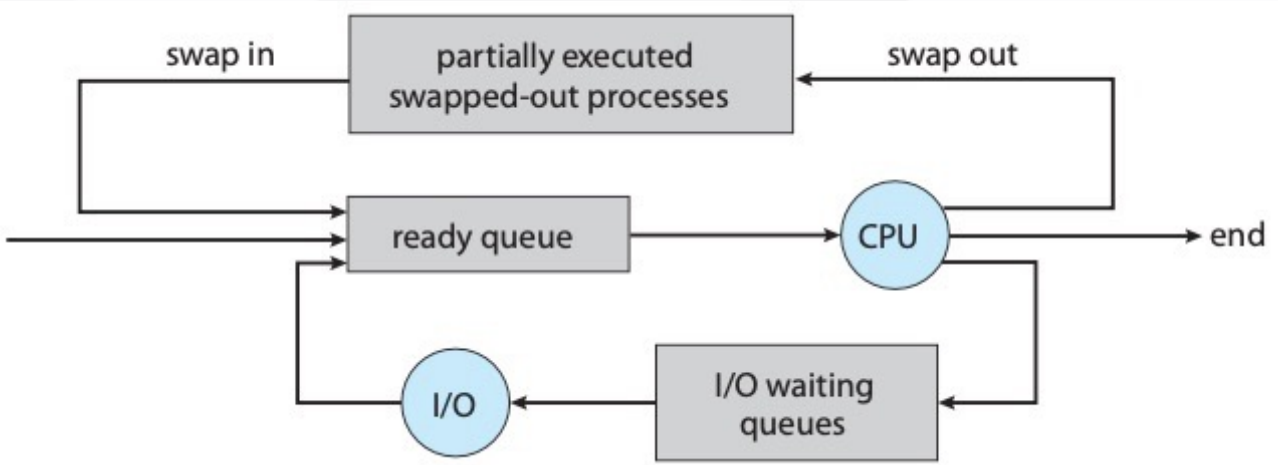

ready queue
CPU
I/O
I/O queue
I/O request
time slice expired
child executes
fork a child
interrupt occurs
wait for an interrupt

调度程序

- 长期调度程序
  - 对于批处理系统，提交的进程多于可以立即执行的。 这些进程会被保存到大容量存储设备(通常为磁盘)的缓冲池，以便以后执行
  - 从该池中选择进程，加到内存，以便执行
- 短期调度程序
  - 从准备执行的进程中选择进程，并分配CPU

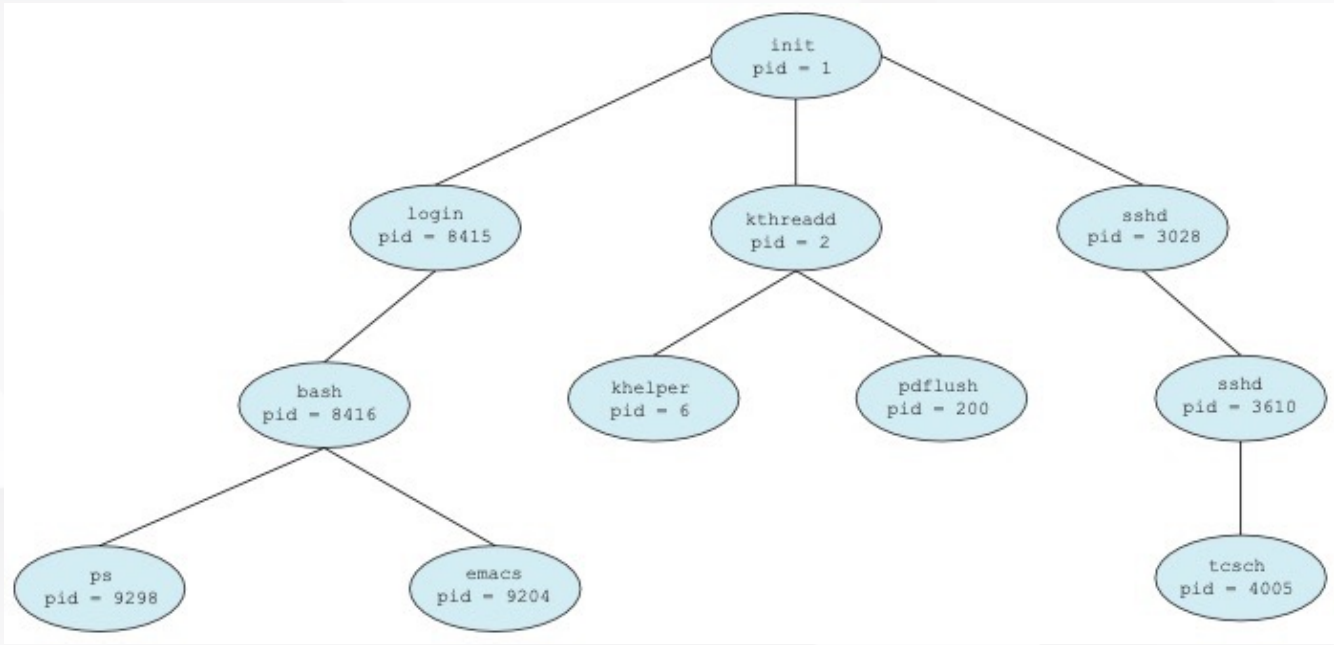
调度程序

- 长期/短期的区别在于执行频率
  - 短期调度程序需要频繁调度进程
  - 长期调度程序只需要在进程离开系统时承担调度任务，可以花费时间谨慎选择，例如将I/O密集型和CPU密集型的进程合理组合
- 引入“中期调度程序”，改善进程组合
  - 可将进程从内存中移出，之后，进程可被重新调入内存，并从中断处继续执行



进程创建

- 新进程可以再创建其他进程（父进程、子进程），从而形成进程树
- 进程可以使用进程标识符 pid（process identifier）来进行识别和管理
  - Linux操作系统的一 个典型进程树

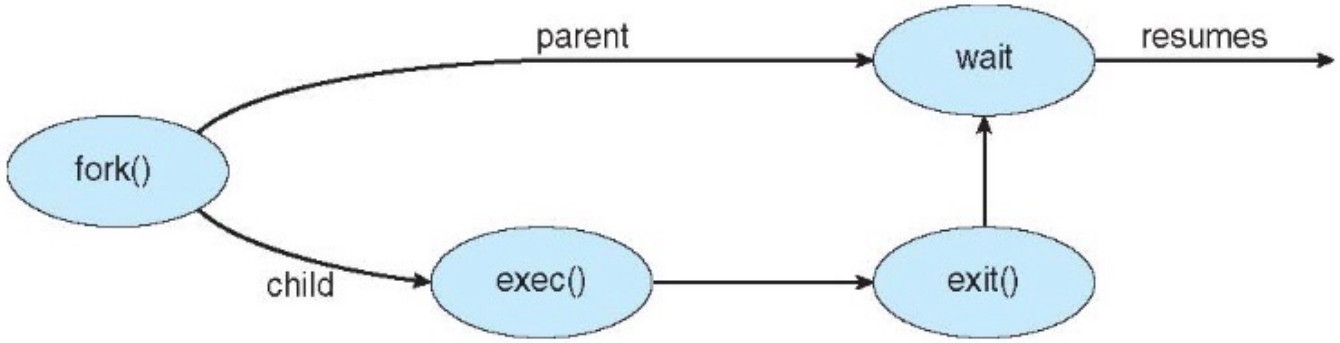


进程创建

- 资源共享
  - 父子进程共享所有资源
  - 子进程共享父进程的部分资源
  - 各自独享
- 执行
  - 并发
  - 父进程等待子进程完成
- 地址空间
  - 子进程复制父进程 UNIX
  - 子进程加载另一个新程序 Windows

UNIX系统实例

- fork，创建新进程
- 子进程使用exec 系统调用，用新程序来取代进程的内存空间
- wait系统调用，等待子进程完成
- 查询**Linux man page**




```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
    int i = 1;
    pid_t  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
          fprintf(stderr, "Fork Failed");
          return 1;
    }
    else if (pid == 0) { /* child process */
          printf( "This is child.");
    }
    else { /* parent process */
          /* parent will wait for the child */
          wait (NULL);
          printf ("Child Complete.");
    }
    return 0;
```

Windows系统实例

- createprocess，创建新进程
- createprocess( )在进程创建时，要求将一个特定程序加载到子进程的地址空间
- 传入特定参数
  - mspaint.exe

```
int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
     "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
     NULL, /* don't inherit process handle */
     NULL, /* don't inherit thread handle */
     FALSE, /* disable handle inheritance */
     0, /* no creation flags */
     NULL, /* use parent's environment block */
     NULL, /* use parent's existing directory */
     &si,
     &pi))
    {
       fprintf(stderr, "Create Process Failed");
       return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

进程终止

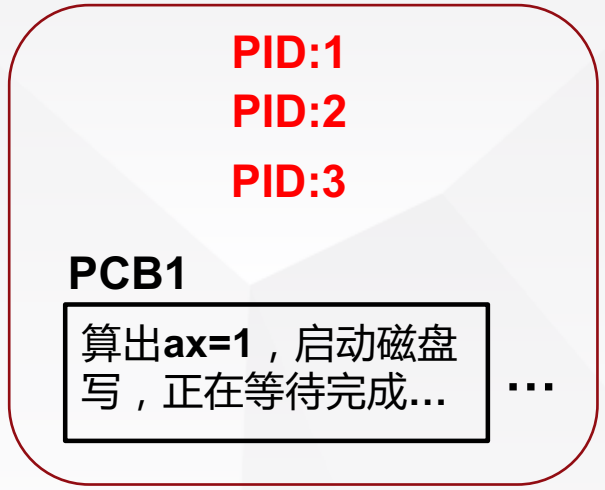
- 通过系统调用exit()，终止进程，释放资源（物理和虚拟内存、打开文件和I/O缓冲区）
- 父进程终止子进程的原因
  - 子进程使用了超过它所分配的资源
  - 分配给子进程的任务，不再需要
  - 父进程正在退出，而且操作系统不允许无父进程的子进程继续执行
- 有些系统不允许子进程在父进程终止的情况下存在。如果一个进程终止，则它的所有子进程也应终止，被称为“级联终止”

进程终止

- 父进程可以通过调用wait()，等待子进程的终止
  - 这个系统调用也返回终止子进程的标识符，这样父进程能够知道哪个子进程已经终止了
- 僵尸进程
  - 一个子进程在父进程还没有调用wait()的情况下退出，这个子进程就是僵尸进程。一般而言僵尸只是短暂存在。一旦父进程调用了wait ()，僵尸进程就被释放
- 孤儿进程
  - 一个父进程退出，它的一个或多个子进程还在运行，子进程将成为孤儿进程。Linux和UNIX的处理方法是将init进程作为孤儿进程的父进程

多个进程使用CPU的图像

- 如何使用CPU呢？
  - 让程序执行起来
- 如何充分利用CPU呢？
  - 创建多个程序，交替执行
- 启动了的程序就是进程，所以是多个进程推进
  - 操作系统只需要把这些进程记录好（PCB）、要按照合理的次序推进(分配资源、进行调度)
  - 这就是多进程图像…



交替的三个部分：队列操作+调度+切换

- 就是进程调度，一个很深刻的话题
- **FIFO?**
  - FIFO显然是公平的策略
  - FIFO显然没有考虑进程执行的任务的区别
- **Priority?**
  - 优先级该怎么设定?可能会使某些进程饥饿

多进程图像：多进程如何影响？

- 多个进程同时在存在于内存会出现下面的问题

进程1代码

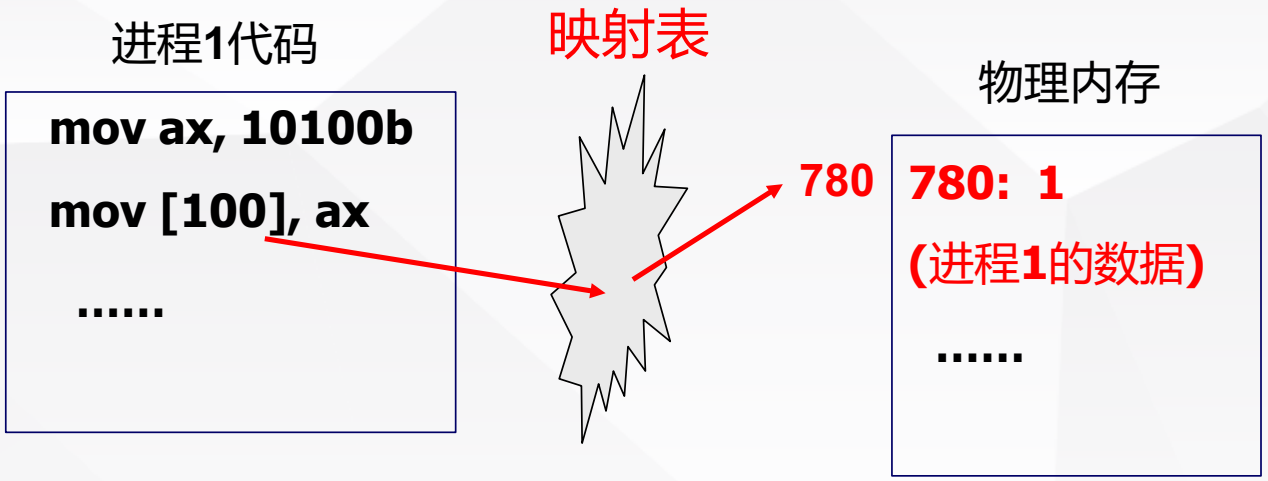
```
mov ax, 10100b
mov [100], ax
  ......
```

进程2代码

```
100: 00101
......
```

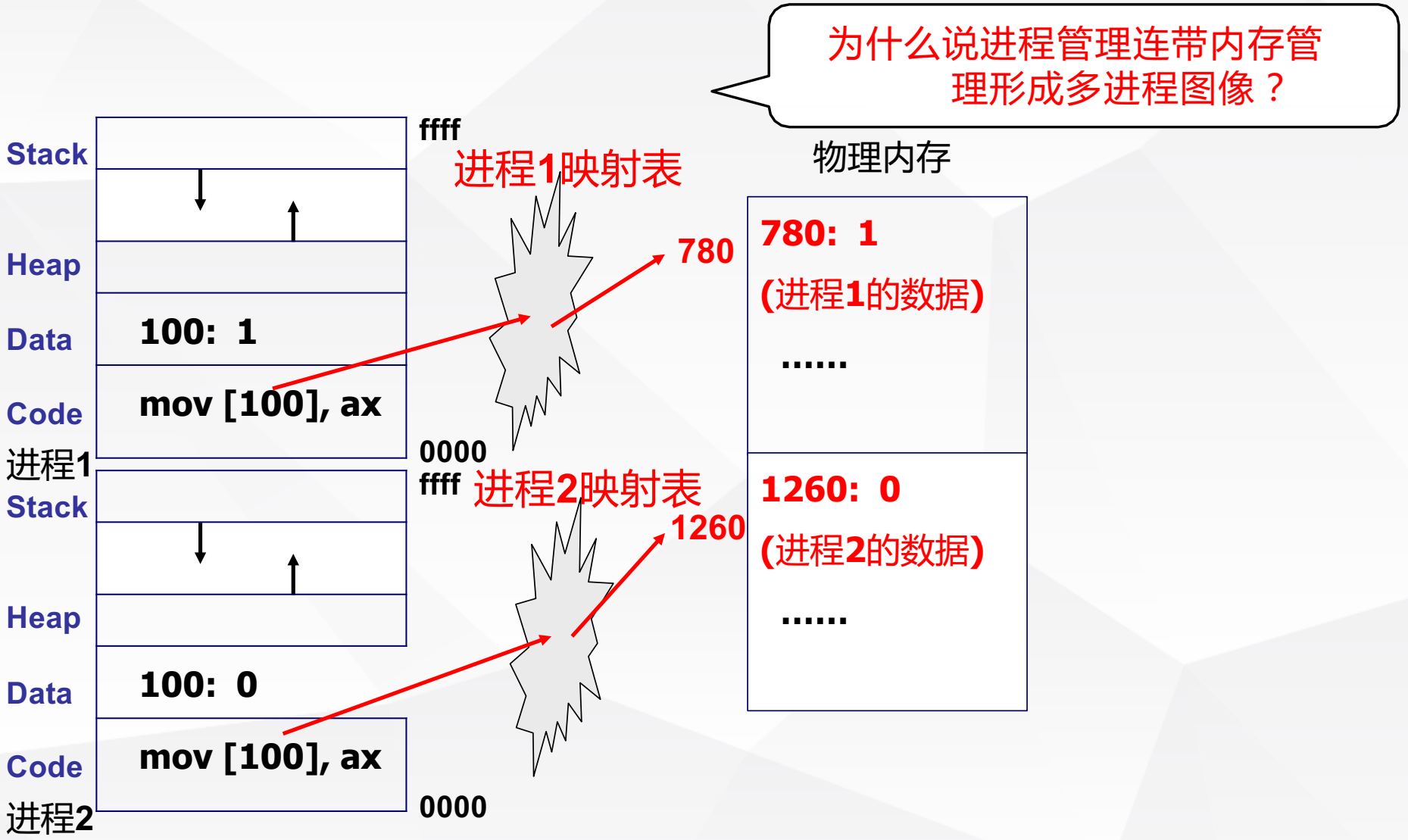
- 解决的办法：限制对地址100的读写
- 多进程的地址空间分离：内存管理的主要内容

进程执行时的100...



- 进程1的映射表将访问限制在进程1范围内
- 进程1根本访问不到其他进程的内容
- 内存管理...

进程带动内存的使用


为什么说进程管理连带内存管理形成多进程图像？
物理内存
Stack
Heap
Data
100: 1
Code
mov [100], ax
进程1
ffff
0000
进程1映射表
780
780: 1
(进程1的数据)
.....
Stack
Heap
Data
100: 0
Code
mov [100], ax
进程2
ffff
0000
进程2映射表
1260
1260: 0
(进程2的数据)
.....

从纸上到实际：生产者-消费者实例

生产者进程

```
while (true) {
  while(counter== BUFFER_SIZE)
      ;
  buffer[in] = item;
  in = (in + 1) % BUFFER_SIZE;
  counter++;
}
```

消费者进程

```
while (true) {
  while(counter== 0)
      ;
  item = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  counter--;
}
```

共享数据

```
#define BUFFER_SIZE 10
typedef struct { . . . } item;
item buffer[BUFFER_SIZE];
int in = out = counter = 0;
```

两个合作的进程都要修改counter

共享数据

```
int counter=0;
```

生产者进程

```
counter++;
```

消费者进程

```
counter--;
```

初始情况

```
counter = 5;
```

生产者P

```
register = counter;
register = register + 1;
counter = register;
```

消费者C

```
register = counter;
register = register - 1;
counter = register;
```

一个可能的执行序列

```
P.register = counter;
P.register = P.register + 1;
C.register = counter;
C.register = C.register - 1;
counter = P.register;
counter = C.register;
```

发生错误！

核心在于进程同步(合理的推进顺序)

- 写counter时阻断其他进程访问counter

一个可能的执行序列

```
P.register = counter;
P.register = P.register + 1;
C.register = counter;
C.register = C.register - 1;
counter = P.register;
counter = C.register;
```

生产者P

给counter上锁

```
P.register = counter;
P.register = P.register + 1;
```

消费者C

检查counter锁

生产者P

```
counter = P.register;
```

给counter开锁

消费者C

给counter上锁

```
C.register = counter;
C.register = C.register - 1;
counter = C.register;
```

给counter开锁

课堂习题

1.论述长期调度和短期调度的差异

2. 如图所示的程序创建了多少个进程（包括初始的父进程）？

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

int main()
{
    /* fork a child process */
    fork();

    /* fork another child process */
    fork();

    /* and fork another */
    fork();

    return 0;
}
```