上海交通大学
SHANGHAI JIAO TONG UNIVERSITY

SJTU

L2-1. 进程

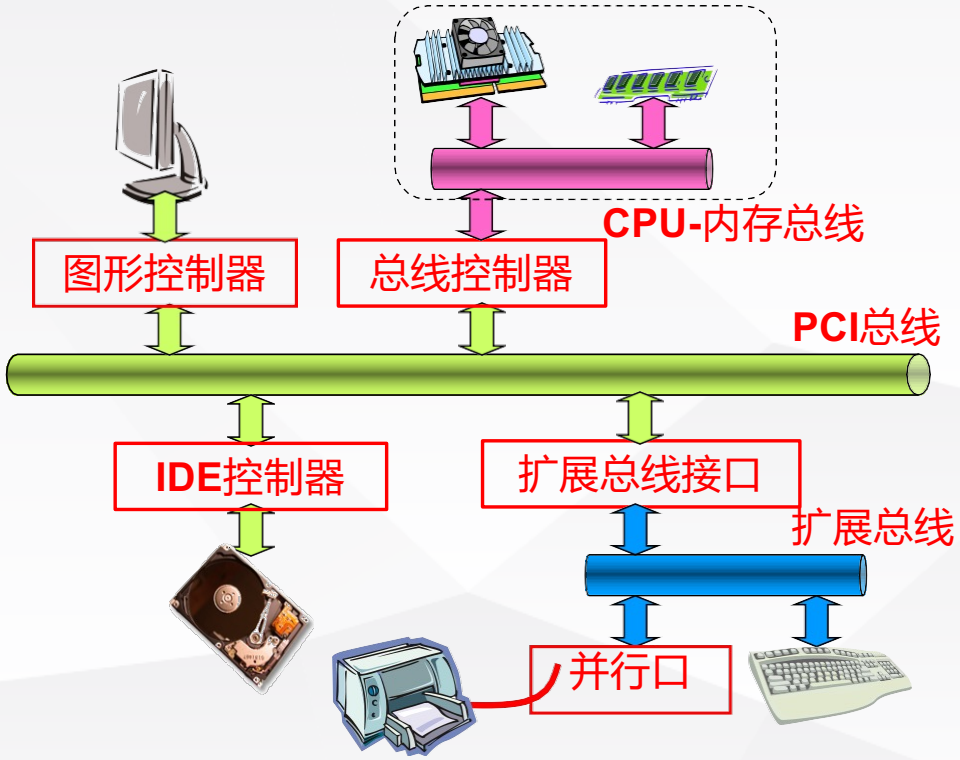
宋卓然

上海交通大学计算机系

songzhuoran@sjtu.edu.cn

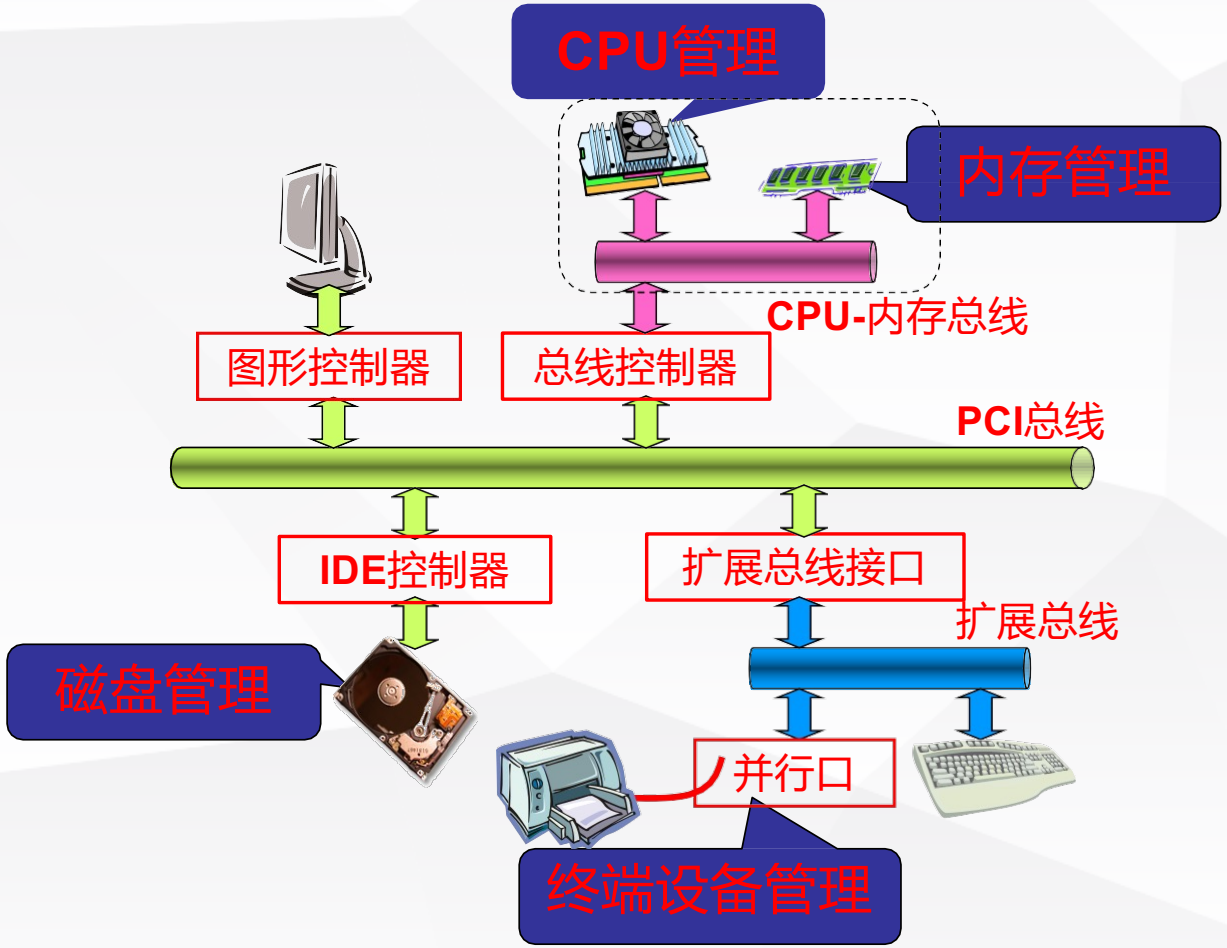
什么是操作系统

- 温故：操作系统要管理硬件，方便我们使用...



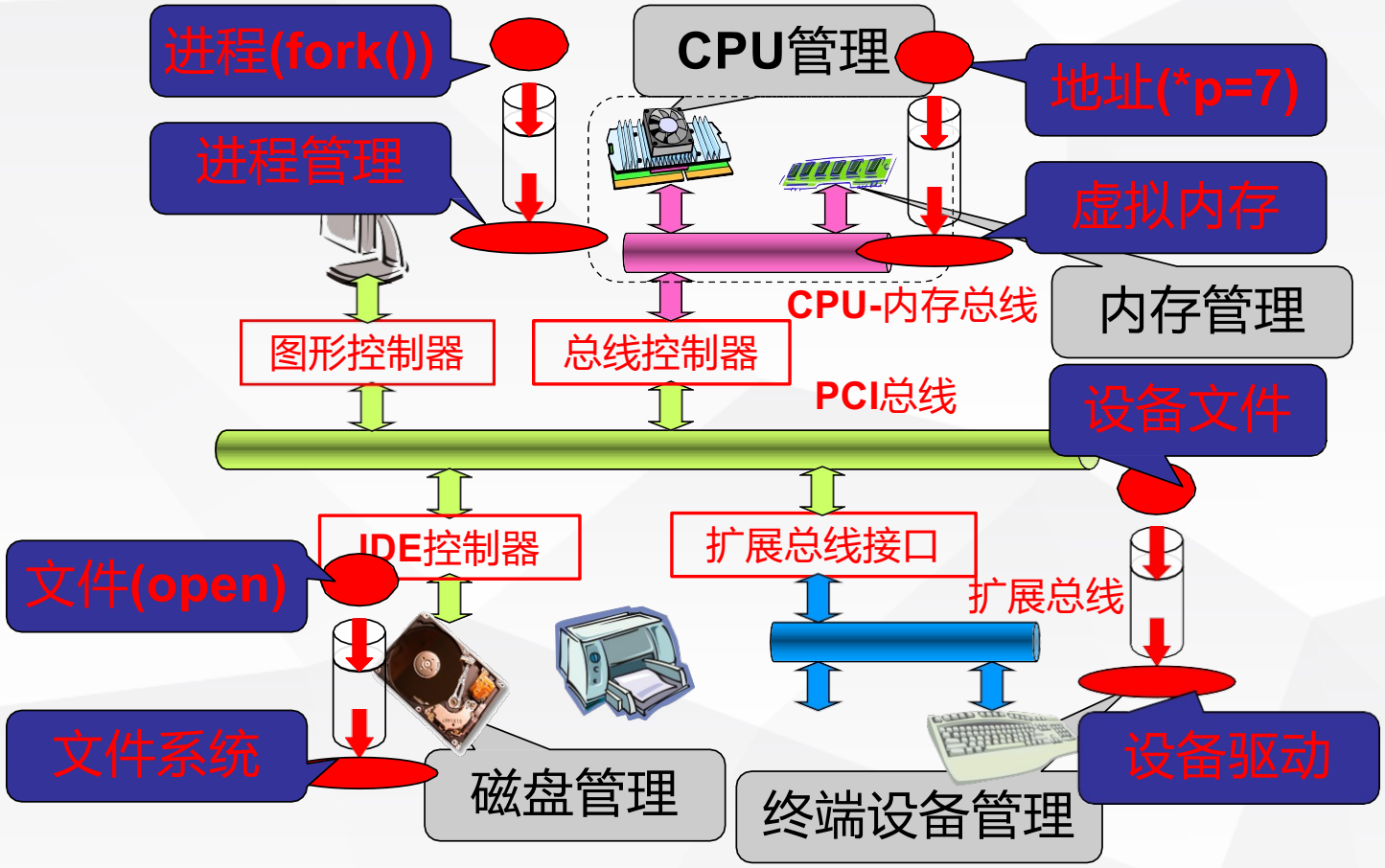
什么是操作系统

- 要管理硬件资源



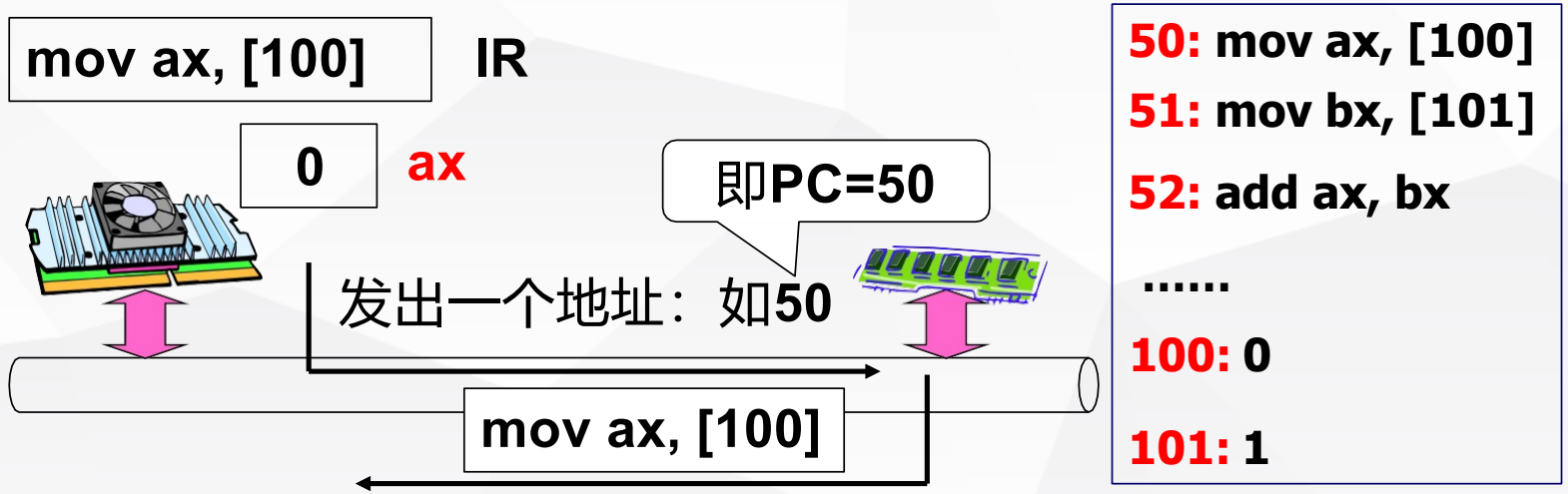
我们要学什么?再具体一些...

- 方便用户使用硬件资源



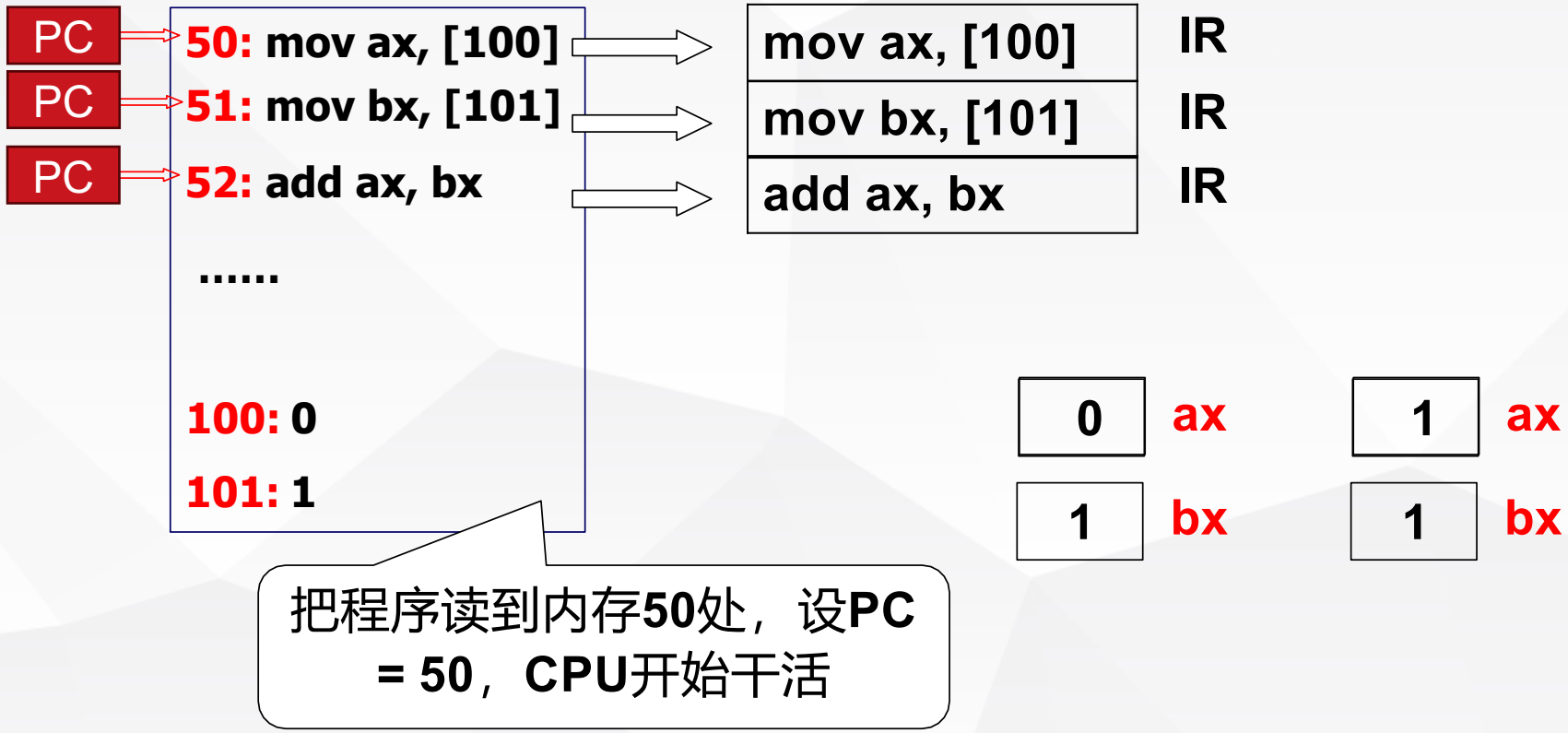
CPU的工作原理

- 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!



这样做有没有问题?

```
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
```

$0.015/10^7$

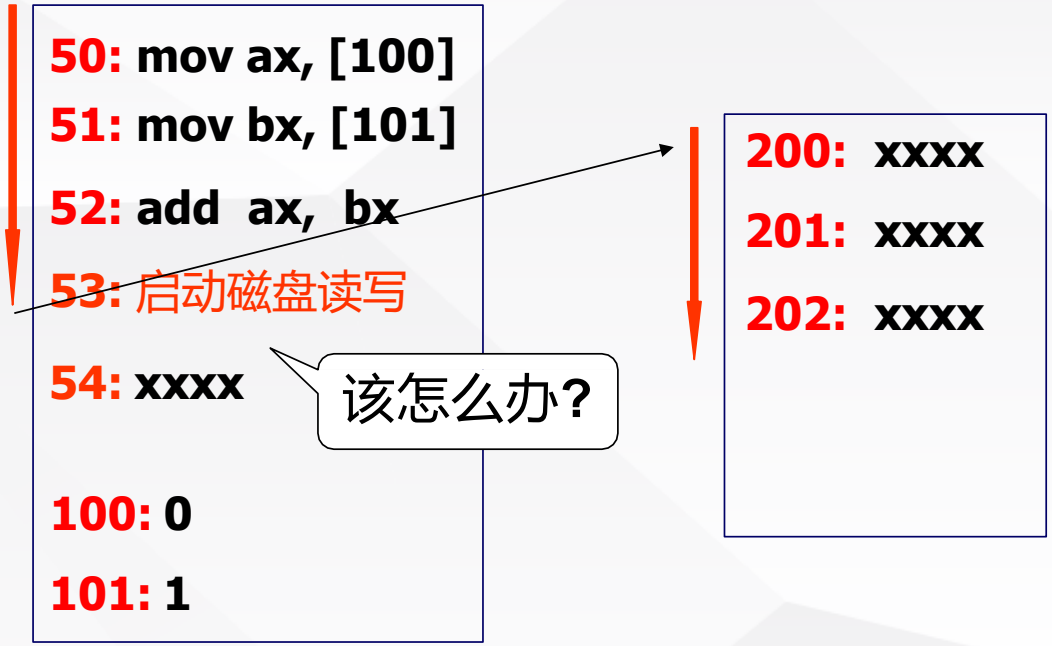
有fprintf

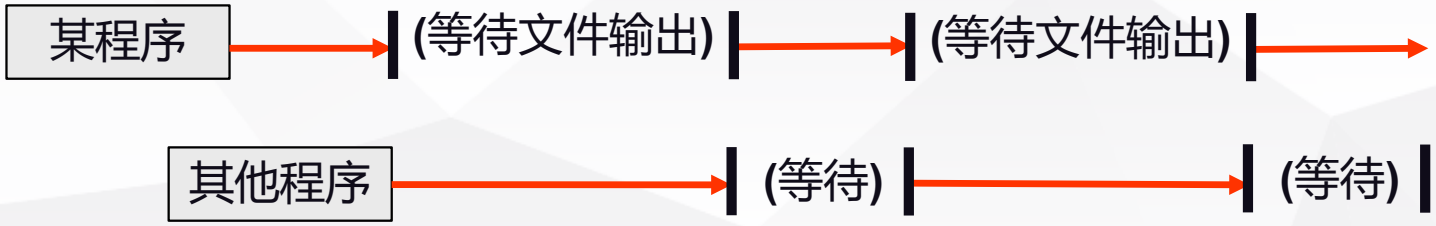
```
C:\>sum 1000
0.859000 seconds
```

$0.859/10^3$

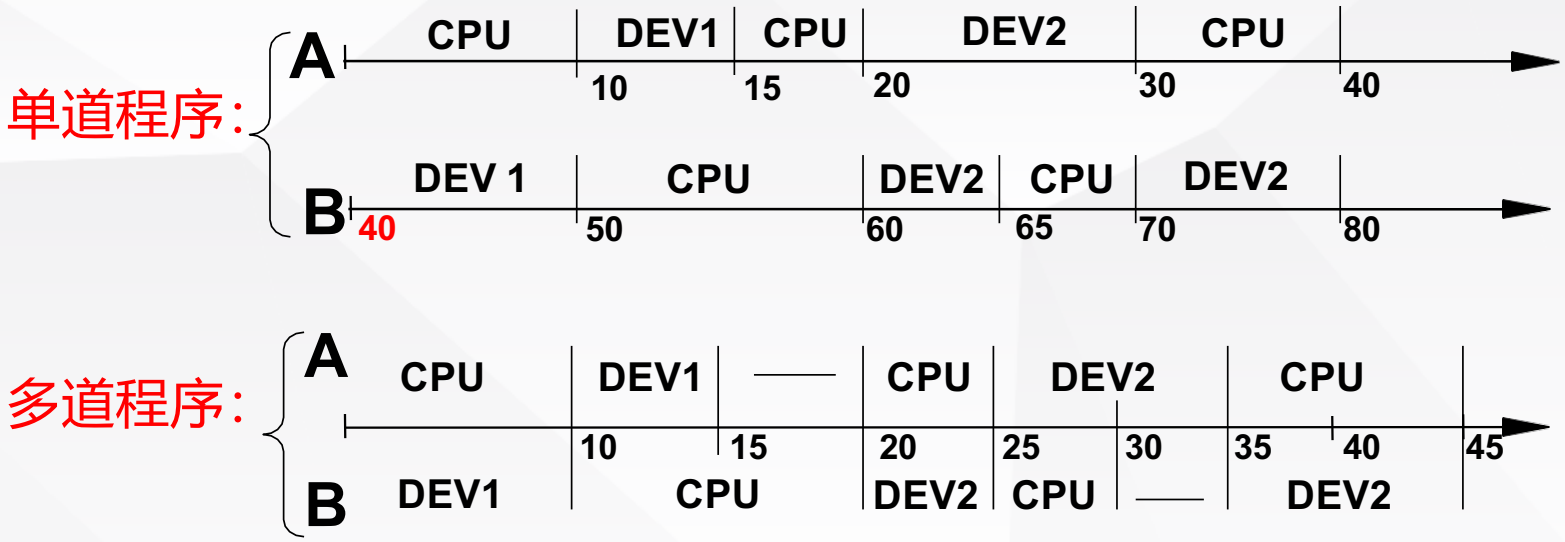
$5.7\times10^5 : 1$

怎么解决?进程切换


50: mov ax, [100]
51: mov bx, [101]
52: add ax, bx
53: 启动磁盘读写
54: xxxx
该怎么办?
100: 0
101: 1
200: xxxx
201: xxxx
202: xxxx

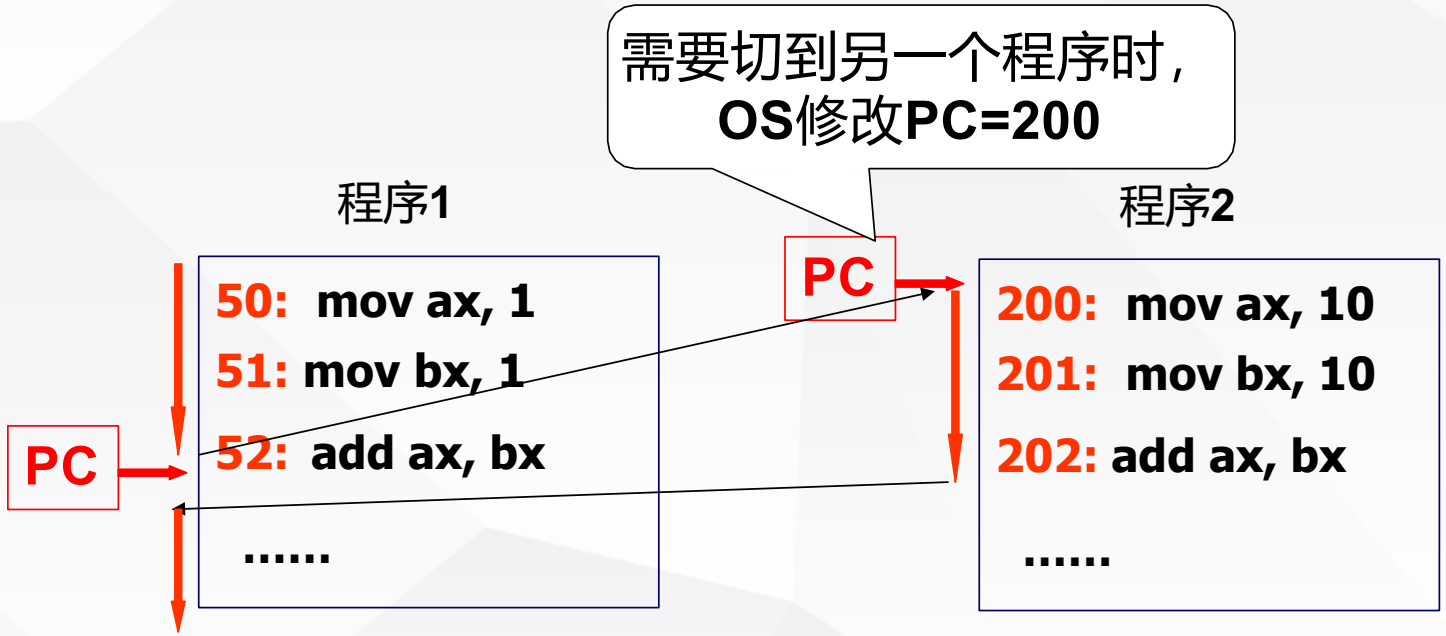

某程序
(等待文件输出)
(等待文件输出)
其他程序
(等待)
(等待)

多道程序、交替执行，好东西啊!



	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%

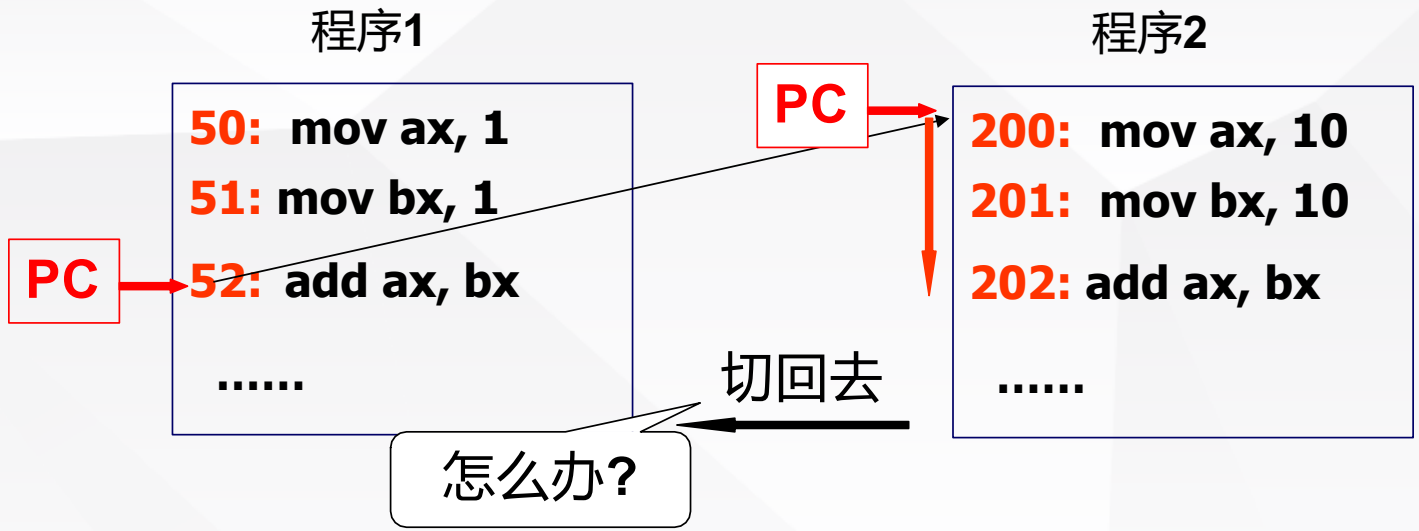
一个CPU面对多个程序？



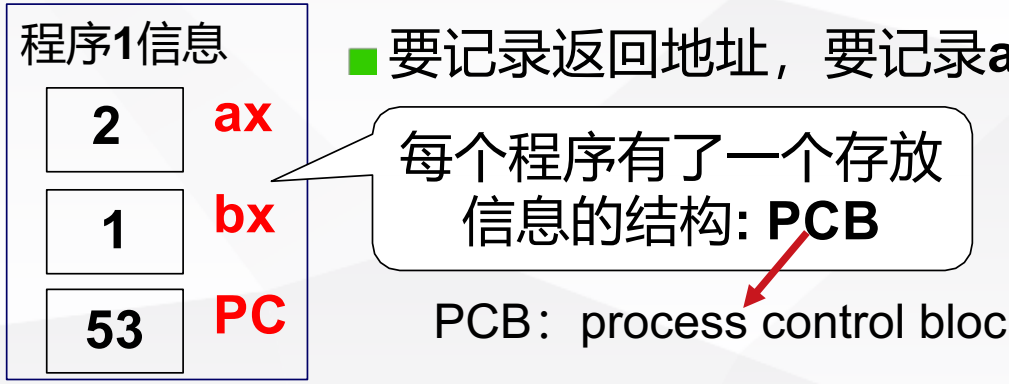
- 一个CPU上交替的执行多个程序：并发
  - 怎么做到？

    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**

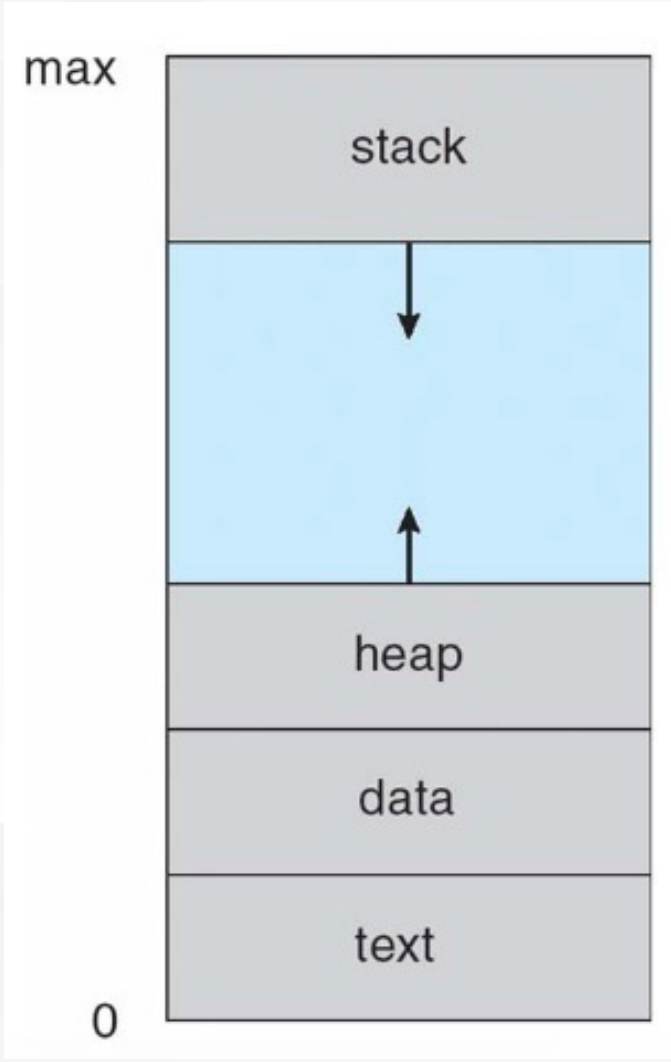
PCB：process control block 进程控制块

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

- 进程用于描述操作系统本身的行为和活动
- 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

进程的概念

- 程序：被动实体，存储在磁盘上包含一系列指令的文件（通常被称为可执行文件，executable file，a.exe）
- 运行程序：./a.exe
  - a.exe将创建一个进程来运行
  - 但a.exe并非一个进程

进程的概念

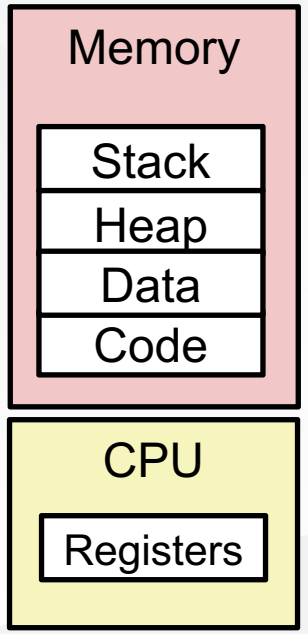
- 每一个进程都具有：

1、“看上去”独立使用CPU

- 当前的活动状态记录（PCB，上下文）
- program counter(当前的程序计数器)
- processor registers（当前的寄存器的状态）

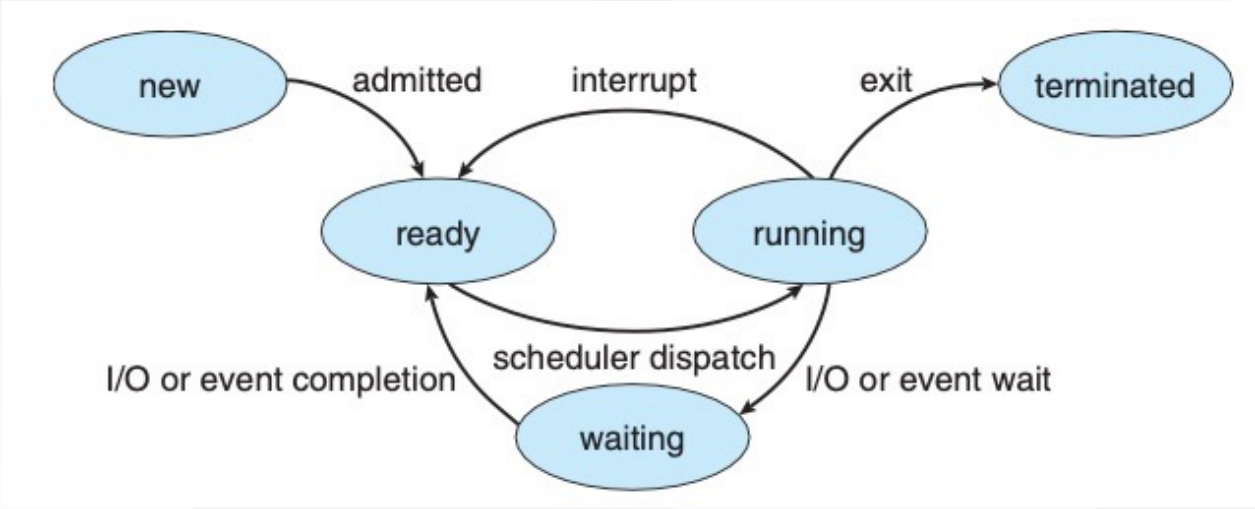
2、Private address space 私有地址空间

- “看上去”拥有整个内存



进程状态

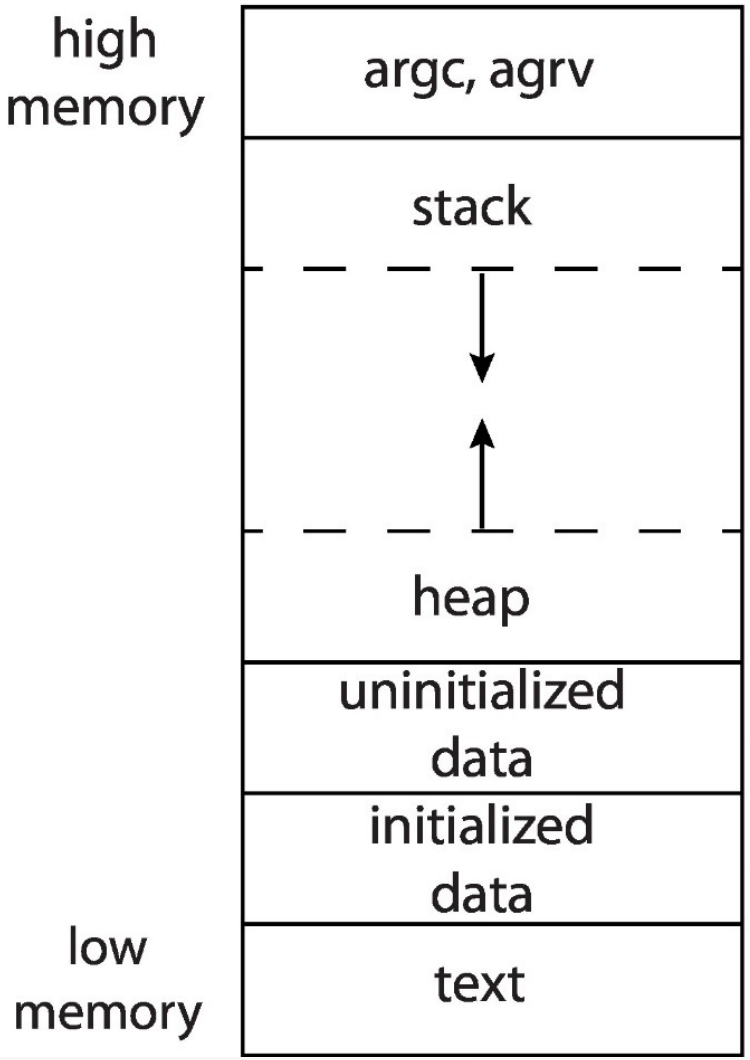
- 进程在执行时可能会改变状态
- 右图可称为进程状态图
- 它能给出进程生存期的清晰描述
- 它是认识操作系统进程管理的一个窗口



New: 刚刚创建
Running: 正在运行
Ready: 准备好被分配运行

Blocked/Waiting: 因为等待某些事件无法运行
Terminated: 执行结束

对于一个C程序进程的内存排布




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

int x;
int y = 15;

int main(int argc, char *argv[])
{
    int *values;
    int i;

    values = (int *)malloc(sizeof(int)*5);

    for(i = 0; i < 5; i++)
        values[i] = i;

    return 0;
}
```

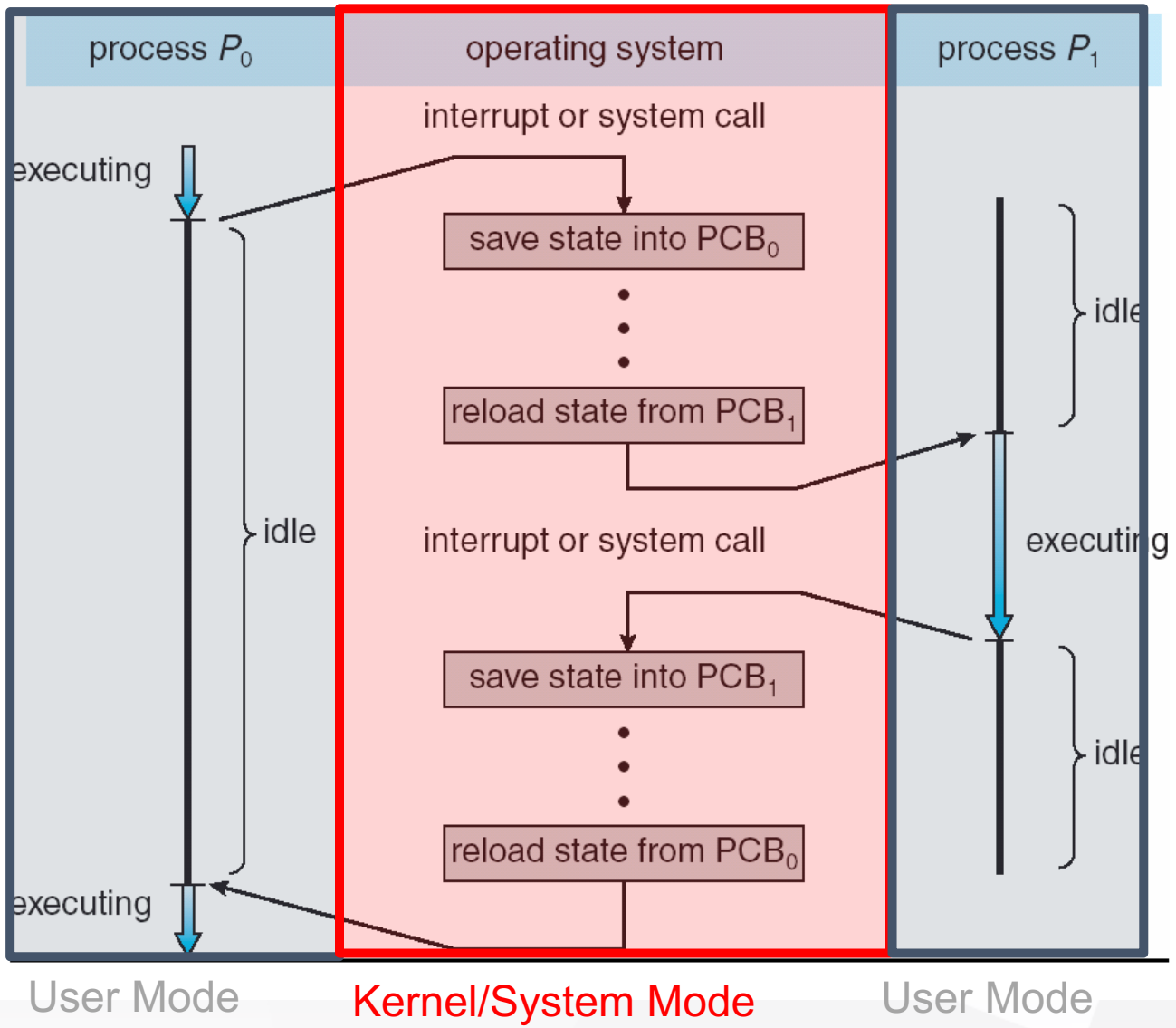
进程调度

- 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
- 切换CPU到另一个进程需要保存当前进程的状态并恢复另一进程的状态，这个过程称为“上下文切换”

```
启动磁盘读写;
pCur.state = 'W';
将pCur放到DiskWaitQueue;
schedule();
```

```
schedule()
{
    pNew = getNext(ReadyQueue);
    switch_to(pCur,pNew);
}
```

进程调度



上下文切换保存：

program counter

accumulators

index registers

stack pointers

general-purpose registers

condition-code information.

Linux的进程表示

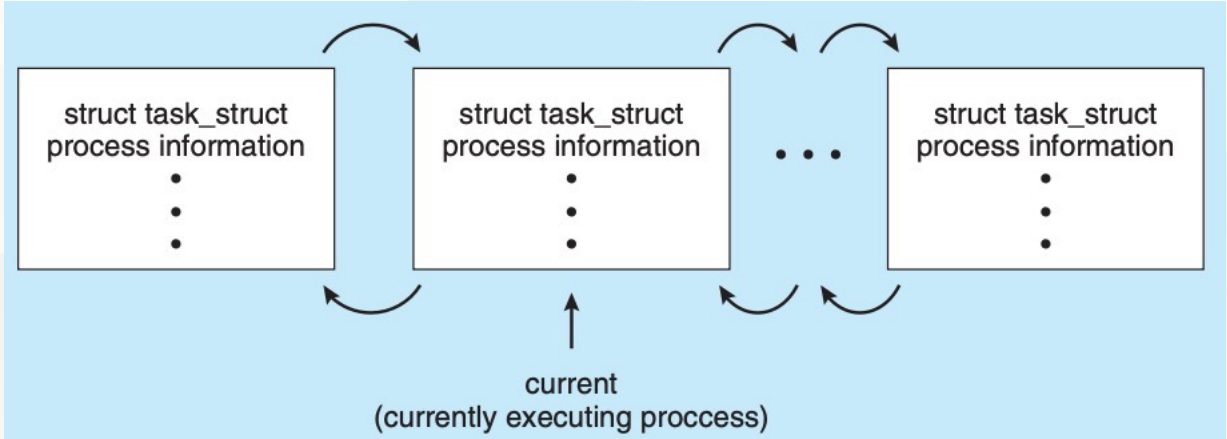
- PCB用C语言结构体task_struct表示，位于内核源代码目录内的头文件<linux/sched.h>

Represented by the C structure **task_struct**

```
pid t pid; /* process identifier */
long state; /* state of the process */
unsigned int time slice /* scheduling information */
struct task_struct *parent; /* this process's parent
*/
struct list_head children; /* this process's children
*/
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this pro */
```

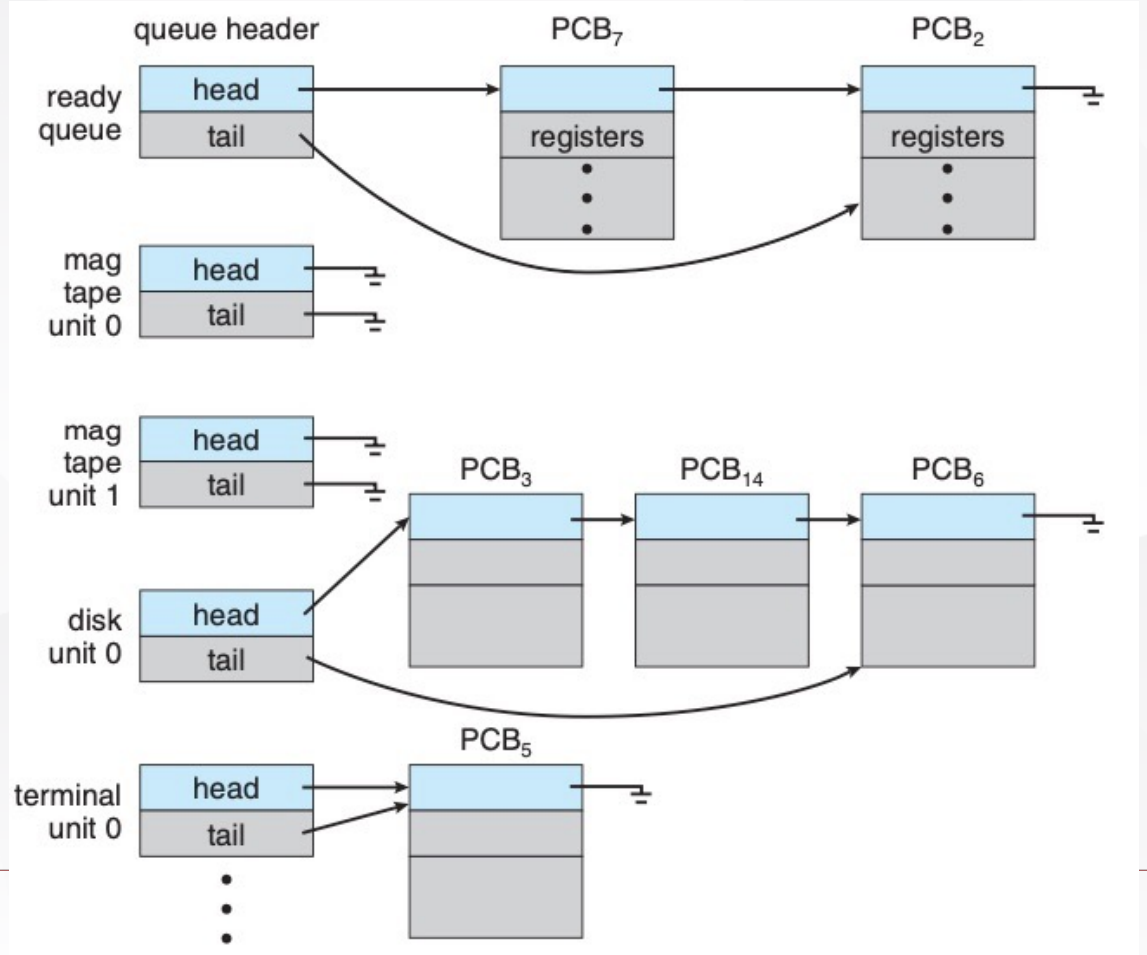
Linux的进程表示

- 所有活动的进程使用双向链表链接task_struct
  - 采用一个指针current指向当前系统正在执行的进程
- 如何修改某进程task_struct的成员？如果要修改进程状态为new_state
  - current->state=new_state;



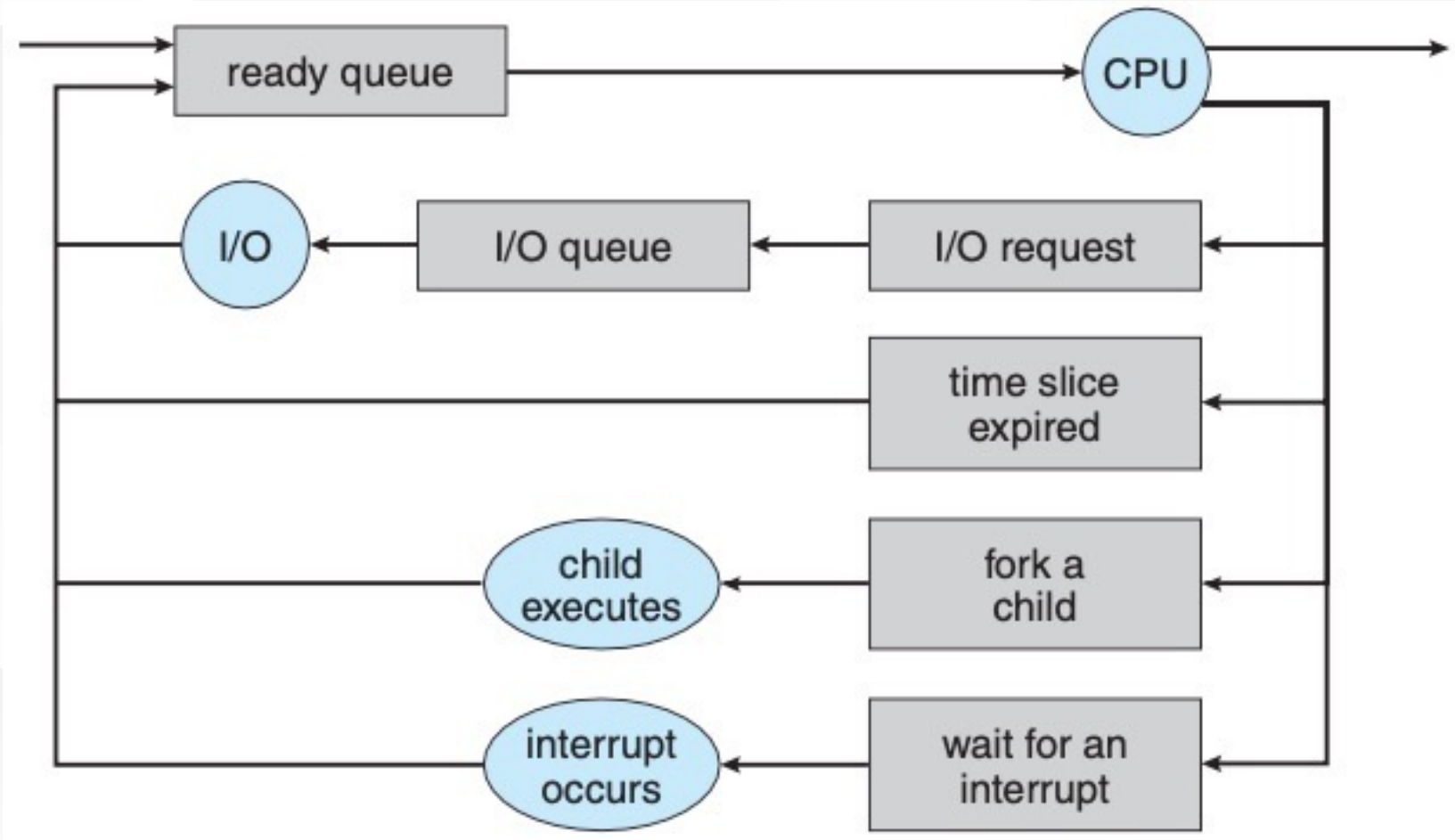
调度队列

- 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

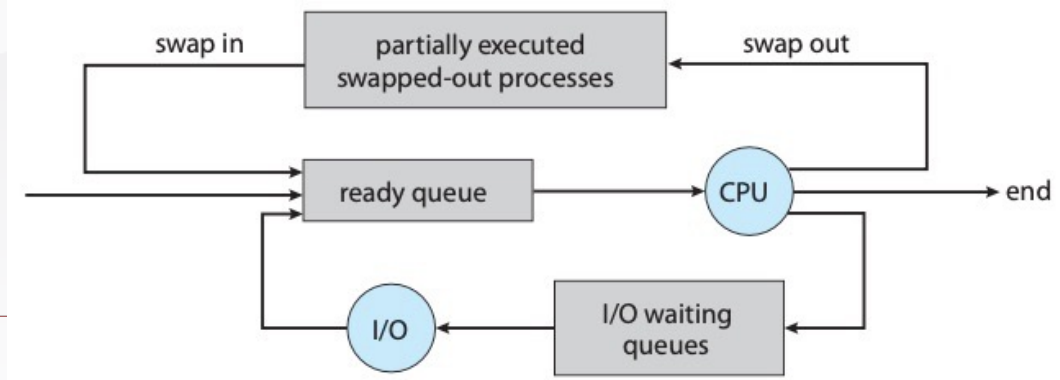

ready queue
CPU
I/O
I/O queue
I/O request
time slice expired
child executes
fork a child
interrupt occurs
wait for an interrupt

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，如果花费过短的时间来调度过长的进程，则浪费过多时间在“调度”上
  - 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

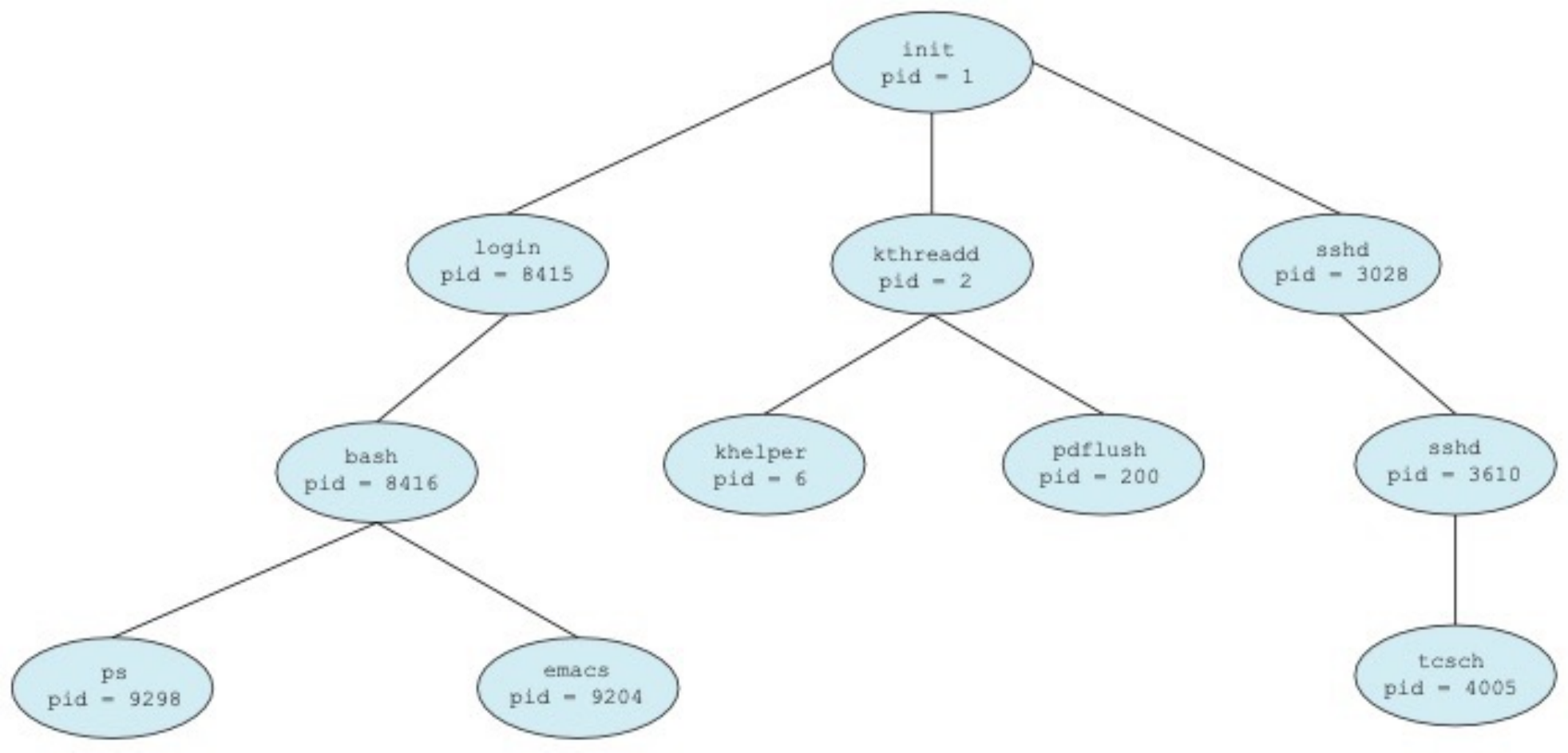


进程创建

- 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

- init为引导/初始程序，kthreadd和sshd为init的子进程，kthreadd负责创建额外进程，执行内核任务，sshd负责管理通过ssh连接到系统的客户端

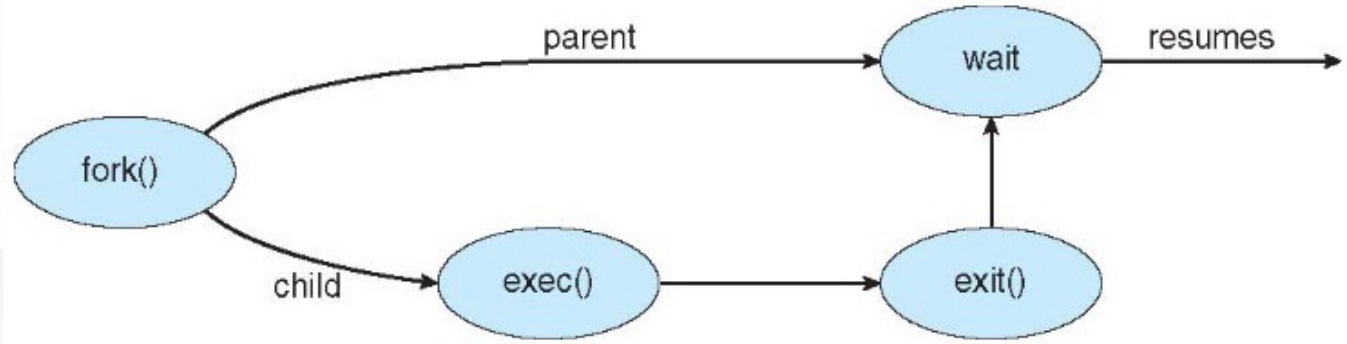


进程创建

- 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




```
#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;
```

fork

- `pid_t fork()` – copy the current process
  - New process has different pid
  - New process contains a single thread
- Return value from **`fork()`**: pid (like an integer)
  - When > 0:
    - Running in (original) Parent process
    - return value is pid of new child
  - When = 0:
    - Running in new Child process
  - When < 0:
    - Error! Must handle somehow
    - Running in original process
- State of original process duplicated in *both* Parent and Child!
  - Address Space (Memory), File Descriptors (covered later), etc…

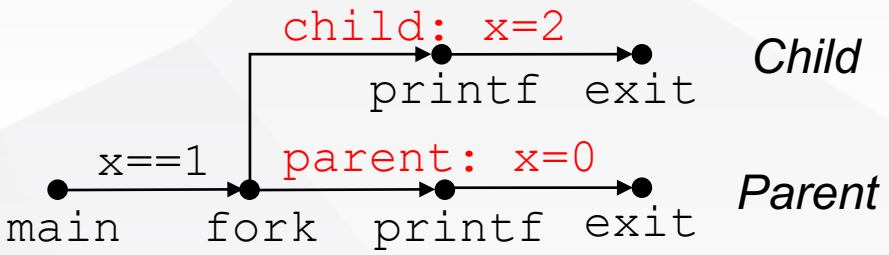
fork

fork.c

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = fork();
    if (pid == 0) {  /* Child */
        printf("child : x=%d\n", ++x);
      exit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```

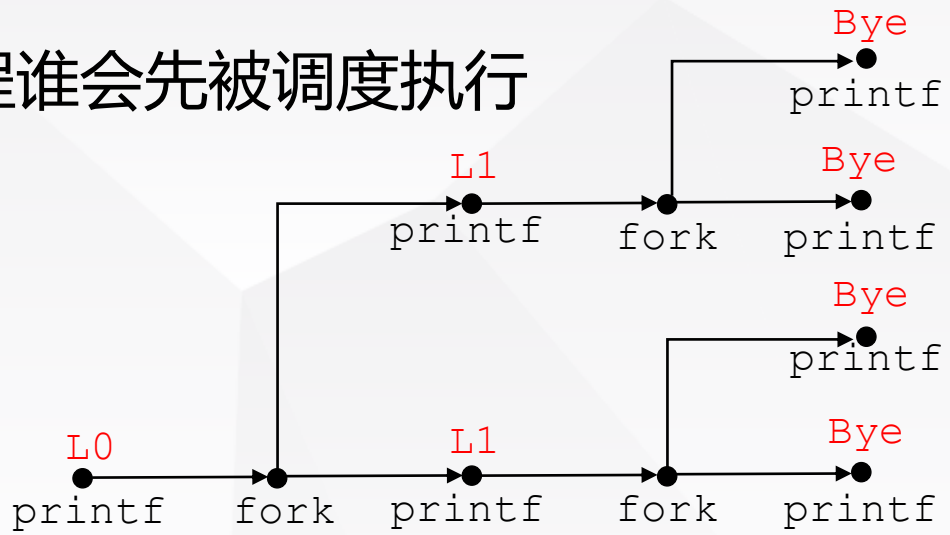



```
linux> ./fork
parent: x=0
child : x=2
```

fork：两个连续的fork

并发执行：无法预测父进程与子进程谁会先被调度执行

```
void fork2()                    forks.c
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



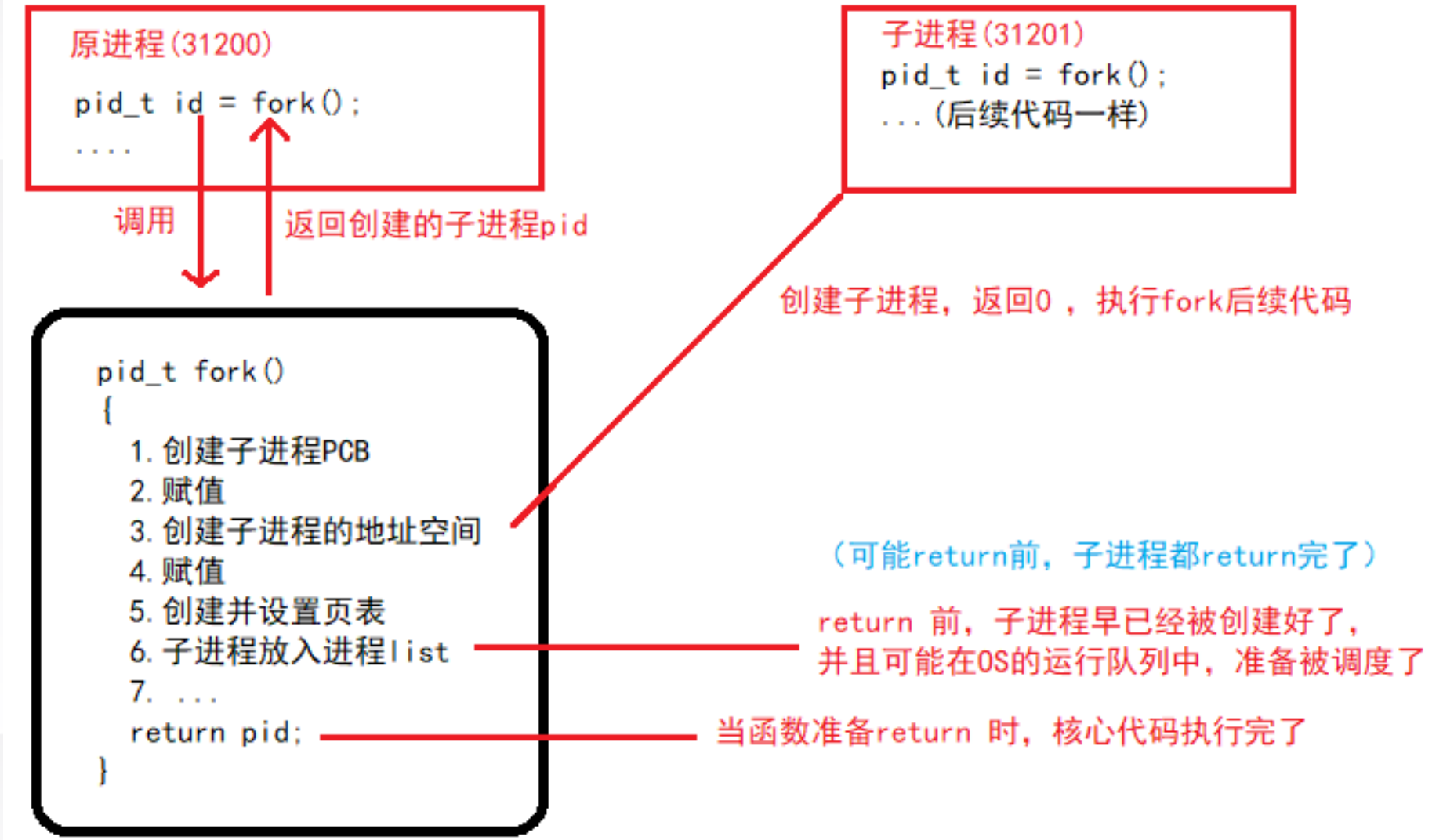
Feasible output:

L0
L1
Bye
Bye
L1
Bye
Bye

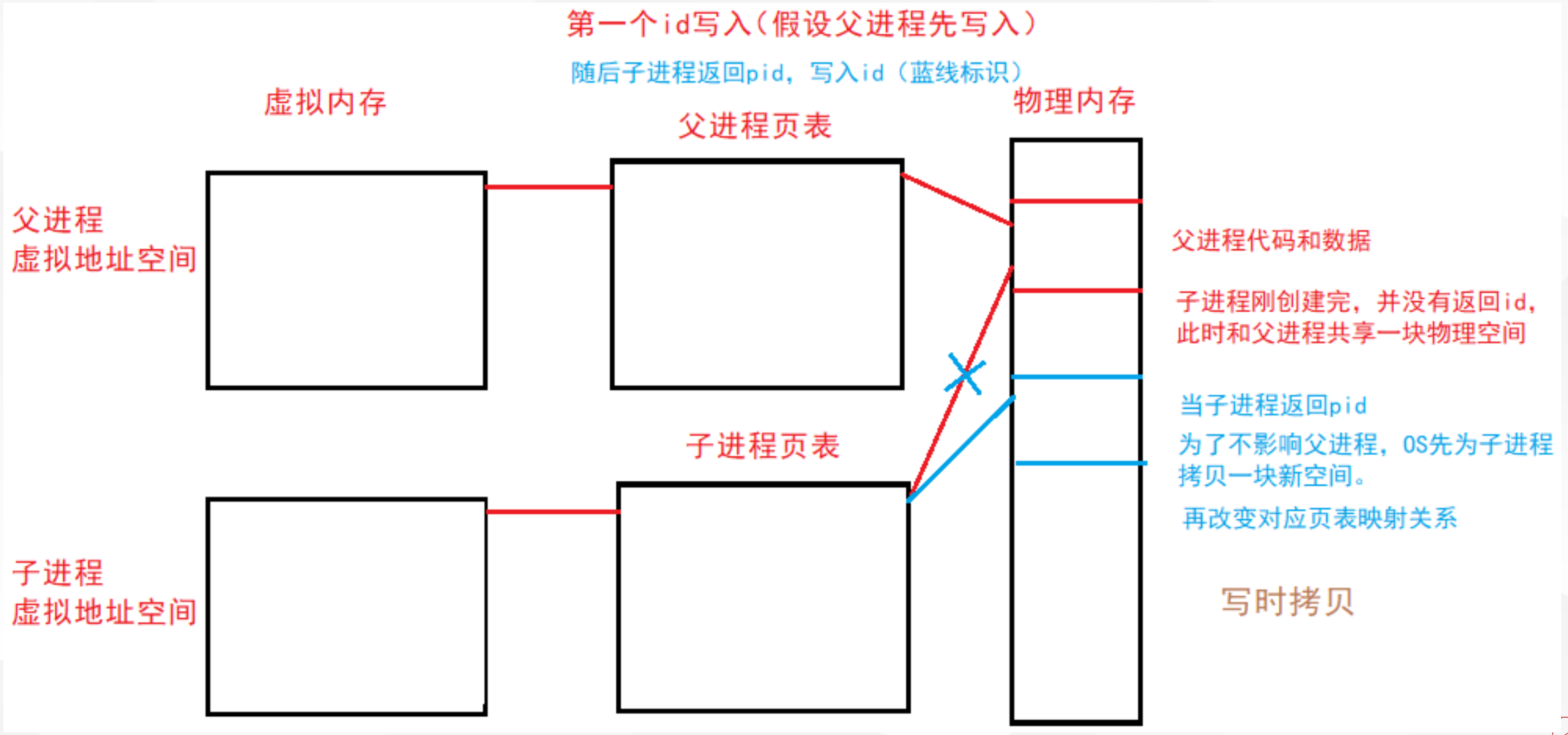
Infeasible output:

L0
Bye
L1
Bye
L1
Bye
Bye

fork的实现细节


原进程(31200)
pid_t id = fork();
....
调用
返回创建的子进程pid
子进程(31201)
pid_t id = fork();
...(后续代码一样)
创建子进程，返回0 ，执行fork后续代码
pid_t fork()
{
1. 创建子进程PCB
2. 赋值
3. 创建子进程的地址空间
4. 赋值
5. 创建并设置页表
6. 子进程放入进程list
7. ...
return pid;
}
（可能return前，子进程都return完了）
return 前，子进程早已经被创建好了，
并且可能在OS的运行队列中，准备被调度了
当函数准备return 时，核心代码执行完了

fork的实现细节

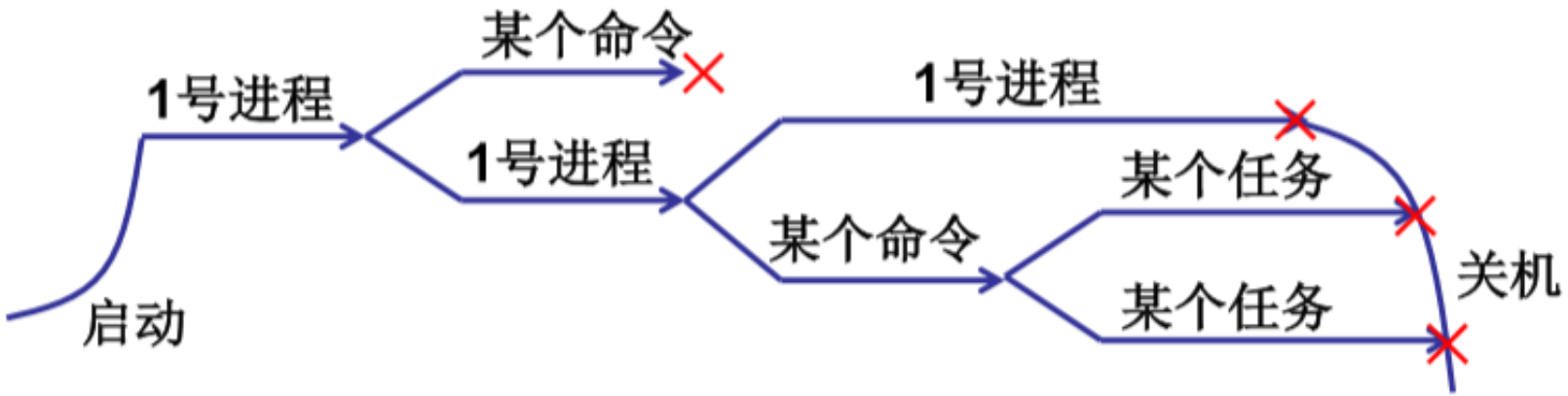

第一个id写入（假设父进程先写入）
随后子进程返回pid，写入id（蓝线标识）
虚拟内存
父进程页表
物理内存
父进程
虚拟地址空间
子进程页表
子进程
虚拟地址空间
父进程代码和数据
子进程刚创建完，并没有返回id，
此时和父进程共享一块物理空间
当子进程返回pid
为了不影响父进程，OS先为子进程
拷贝一块新空间。
再改变对应页表映射关系
写时拷贝

从开机到关机

- main中的fork()创建了第1个进程
  - init执行了shell(Windows桌面)
- shell再启动其他进程

```
int main(int argc, char * argv[])
{   while(1)  { scanf("%s", cmd);
        if(!fork()) {exec(cmd);} wait(); }  }
```

一命令启动一个进程，返回shell再启动其他进程…



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

- 正常退出的方法：
  - 1、在main函数中执行return
  - 2、调用exit函数
- 异常退出的方法：
  - 1、调用abort函数
  - 2、进程收到某个信号，而该信号使进程终止

进程终止

- return和exit的差别
  - return() 代表调用栈的返回，exit()代表一个进程的结束。
  - 从main函数退出，会隐式的调用exit()函数，并将return的返回值传递给exit()
  - 从exit()退出，会先调用退出程序，刷新stdio流缓冲区，使用由status提供的值执行_exit()系统调用
  - return 是关键字，exit()是库函数。

```
int main()
{
        return 0;
        //exit(0);
}
```

进程终止

- 父进程终止子进程的条件：需要知道子进程的标识符pid
- 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

```
int wait(int *child_status)
```

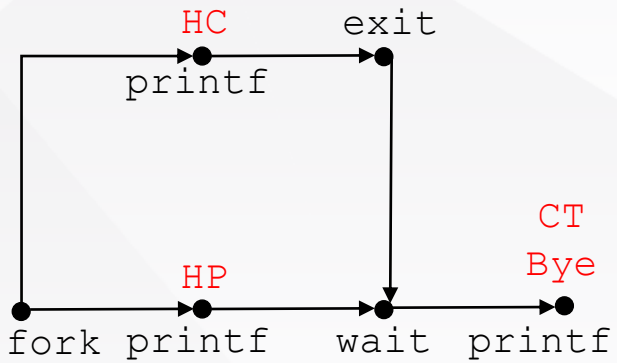
- **调用wait函数的进程会被挂起（阻塞），直到其任意一个子进程结束。**
- 当子进程结束时，该函数会回收子进程所占的资源（PCB等），并返回被回收的子进程的pid。将被回收子进程的状态通过child_status指针参数返回。
- 如果没有运行的子进程，则函数立刻返回-1.

进程终止

```
void fork9() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from
child\n");
      exit(0);
    } else {
        printf("HP: hello from
parent\n");
        wait(&child_status);
        printf("CT: child has
terminated\n");
    }
    printf("Bye\n");
}
```

forks.c



Feasible output:

HC
HP
CT
Bye

Infeasible output:

HP
CT
Bye
HC

进程终止

- 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

僵尸进程

- “僵尸”进程(zombie或defunct)
  - 进程生命期结束时的特殊状态
    - 系统已经释放了进程占用的包括内存在内的系统资源，但仍在内核中保留进程的部分数据结构，记录进程的终止状态，等待父进程来“收尸”
    - 父进程的“收尸”动作完成之后，“僵尸”进程不再存在
  - 僵尸进程占用资源很少，仅占用内核进程表资源
    - 过多的僵尸进程会导致系统有限数目的进程表被用光
  - 僵尸进程的回收：使用**waitpid**()

僵尸进程

```
void fork7() {
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
getpid());
        exit(0); 子进程退出了
    } else {
        printf("Running Parent, PID = %d\n",
getpid());                                forks.c
        while (1)  父进程还没有调用wait()
            ; /* Infinite loop */
    }
}
```

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 forks
 6640 ttyp9    00:00:00 forks <de
 6641 ttyp9    00:00:00 ps
linux> kill 6639
[1]    Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
```

- `ps` shows child process as “defunct” (i.e., a zombie)
- Killing parent allows child to be reaped by `init`
- `Init`会管理所有子进程，并将该子进程回收

waitpid

int waitpid(pid_t pid, int *wstatus, int options)

- 参数pid可以指定要回收的子进程的进程号id
  - pid = -1: 表示要回收任意子进程
  - pid = 0: 回收当前进程同一进程组中的任意一个进程
  - pid > 0: 回收进程号为pid的子进程
- 参数options可以选择在回收时是否阻塞当前进程：
  - 0: 阻塞（等价于wait)
  - WNOHANG：不会阻塞，立即返回。
- 返回值：正常回收则返回回收的子进程号；若没有子进程可回收则返回0；若回收出错则返回-1

waitpid

```
void fork11() {
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */  终止子进程
    for (i = N-1; i >= 0; i--) {
        pid_t wpid = waitpid(pid[i], &child_status, 0); 回收子进程
        if (WIFEXITED(child_status)) 检查子进程是否是通过exit正常退出
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```

waitpid

```
songzhuoran@songzhuorandeMacBook-Air Desktop % gcc fork_example.c -o fork_exam
ple
songzhuoran@songzhuorandeMacBook-Air Desktop % ./fork_example
Child 30702 terminated with exit status 104
Child 30701 terminated with exit status 103
Child 30700 terminated with exit status 102
Child 30699 terminated with exit status 101
Child 30698 terminated with exit status 100
```

waitpid：以不阻塞的方式回收子进程

```c
pid_t pid;
for(int i = 0; i < 5; i++){
    pid = fork();
    if(pid == 0){
        break;
    }
    sleep(1);
}
if(pid > 0){
    while(1){
        printf("父进程, pid:%d\n", getpid());
        int ret = waitpid(-1, NULL, WNOHANG);
        if(ret > 0)
            printf("回收了进程%d\n", ret);
        sleep(1);
    }
}
else{
    printf("子进程, pid:%d, ppid:%d\n", getpid(), getppid());
    sleep(getpid() % 10 + 3);
    exit(0);
}
```

getppid获取父进程pid

```
子进程, pid:7557, ppid:7556
子进程, pid:7559, ppid:7556
子进程, pid:7572, ppid:7556
子进程, pid:7582, ppid:7556
子进程, pid:7583, ppid:7556
父进程, pid:7556
父进程, pid:7556
父进程, pid:7556
回收了进程7572
父进程, pid:7556
回收了进程7582
父进程, pid:7556
父进程, pid:7556
回收了进程7557
父进程, pid:7556
回收了进程7583
父进程, pid:7556
父进程, pid:7556
回收了进程7559
父进程, pid:7556
父进程, pid:7556
父进程, pid:7556
```

自己编写一个简单的Shell程序

```
int main()
{
    char cmdline[MAXLINE]; /* command line */
    while (1) {
        /* read */
        printf("> ");
        fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin))
            exit(0);

        /* evaluate */
        eval(cmdline);
    }
}
```

存储用户输入的命令行

无限循环，持续等待用户输入

打印提示符，表示程序等待用户输入命令

使用 fgets() 函数从标准输入（键盘）读取用户输入的命令行，并存储在 cmdline 中

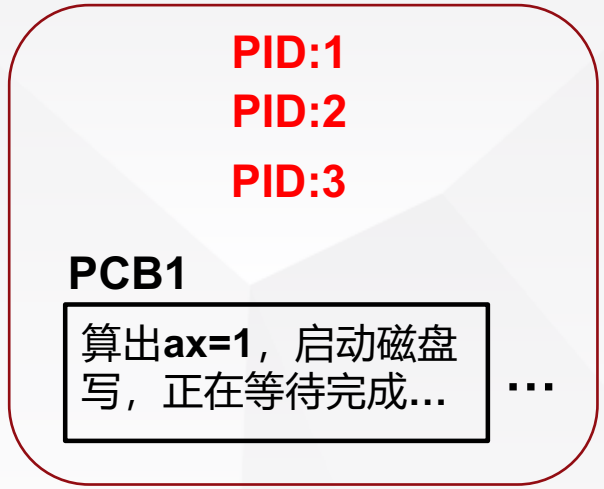
检查是否达到了文件结束标志（End of File），如果是，则退出程序

调用 eval() 函数来处理用户输入的命令

```
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* Argument list execve() */
    char buf[MAXLINE];   /* Holds modified command line */
    int bg;              /* Should the job run in bg or fg? */
    pid_t pid;           /* Process id */
    strcpy(buf, cmdline);  将用户输入的命令行复制到 buf 中
    bg = parseline(buf, argv);  解析用户输入的命令行，并将解析后的参数存储在 argv 数组中
    if (argv[0] == NULL)  如果用户输入的命令为空（即只有换行符），则直接返回，忽略这个空命令
        return;   /* Ignore empty lines */
    if (!builtin_command(argv)) {                                   创建子进程来执行用户输入的命令
        if ((pid = fork()) == 0) {   /* Child runs user job */
            if (execve(argv[0], argv, environ) < 0) { 在子进程中执行用户输入的命令
                printf("%s: Command not found.\n", argv[0]);
                exit(0); 子进程退出
            }
        }
        /* Parent waits for foreground job to terminate */
      if (!bg) { waitpid(pid, &status, 0) };
    }
    return;
}
```

多个进程使用CPU的图像

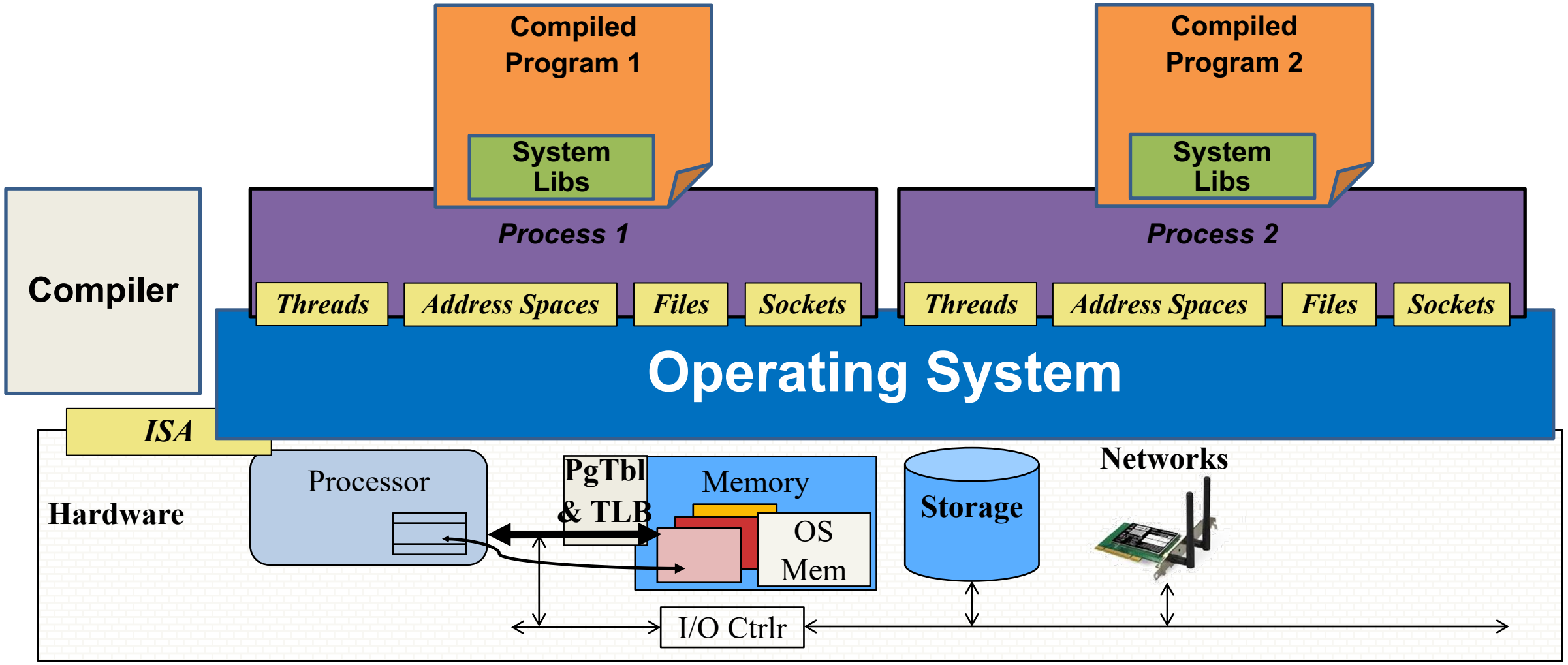
- 如何使用CPU呢？
  - 让程序执行起来
- 如何充分利用CPU呢？
  - 创建多个程序，交替执行
- 启动了的程序就是进程，所以是多个进程推进
  - 操作系统只需要把这些进程记录好（PCB）、要按照合理的次序推进(分配资源、进行调度)
  - 这就是多进程图像…



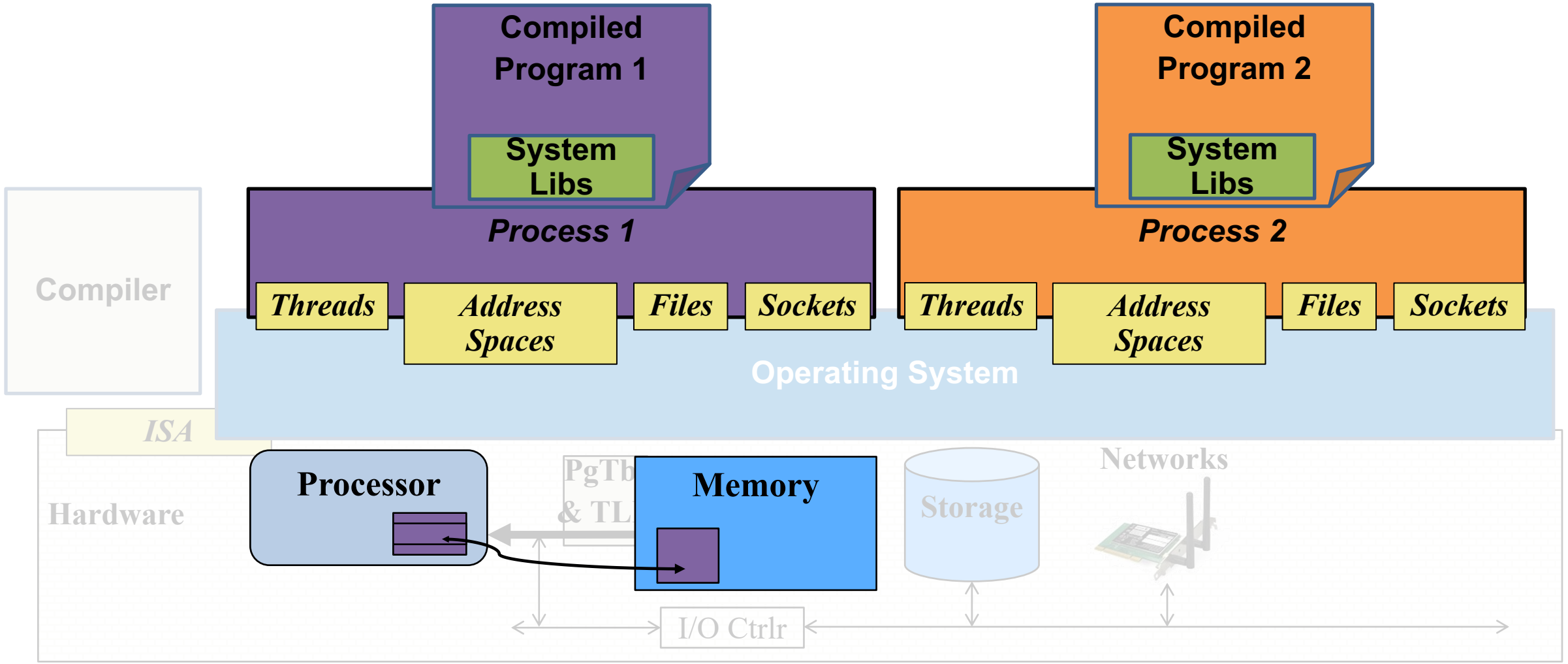
交替的三个部分：队列操作+调度+切换

- 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

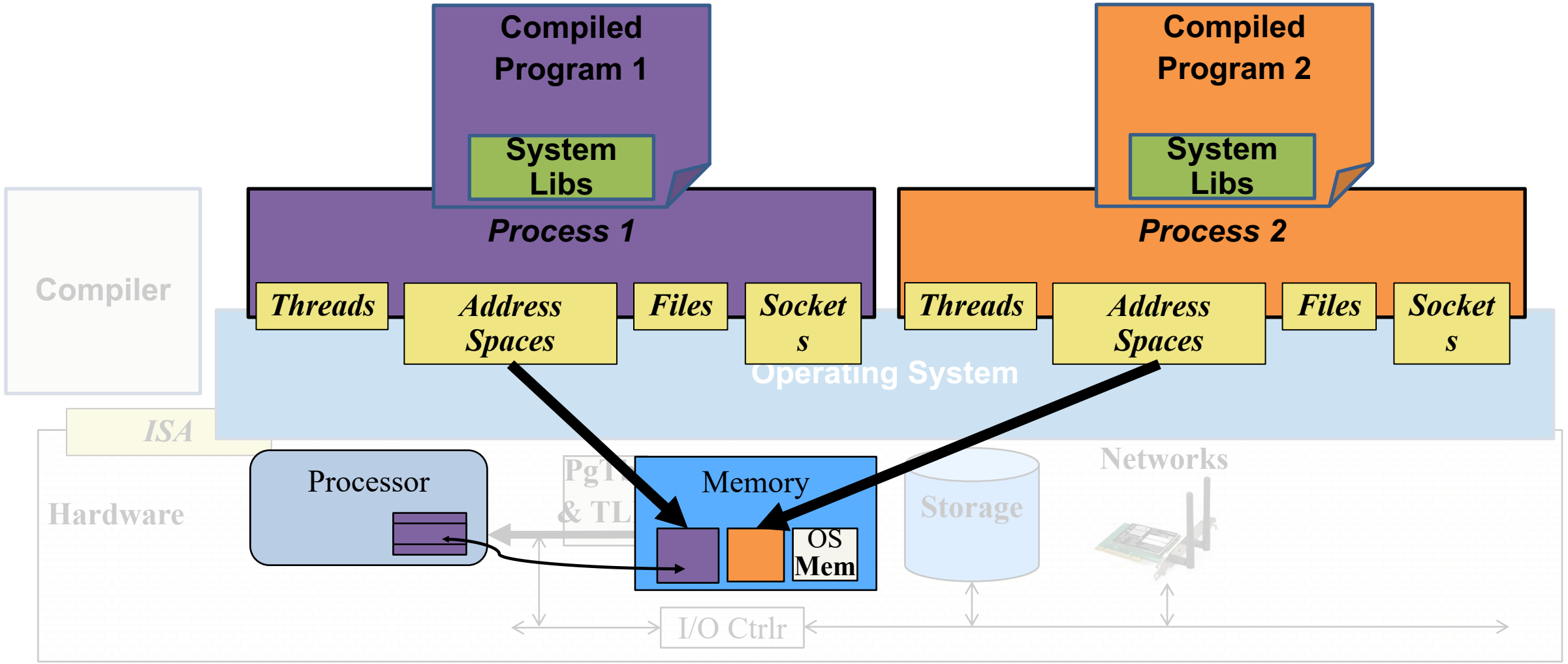
Operating System's View of the World



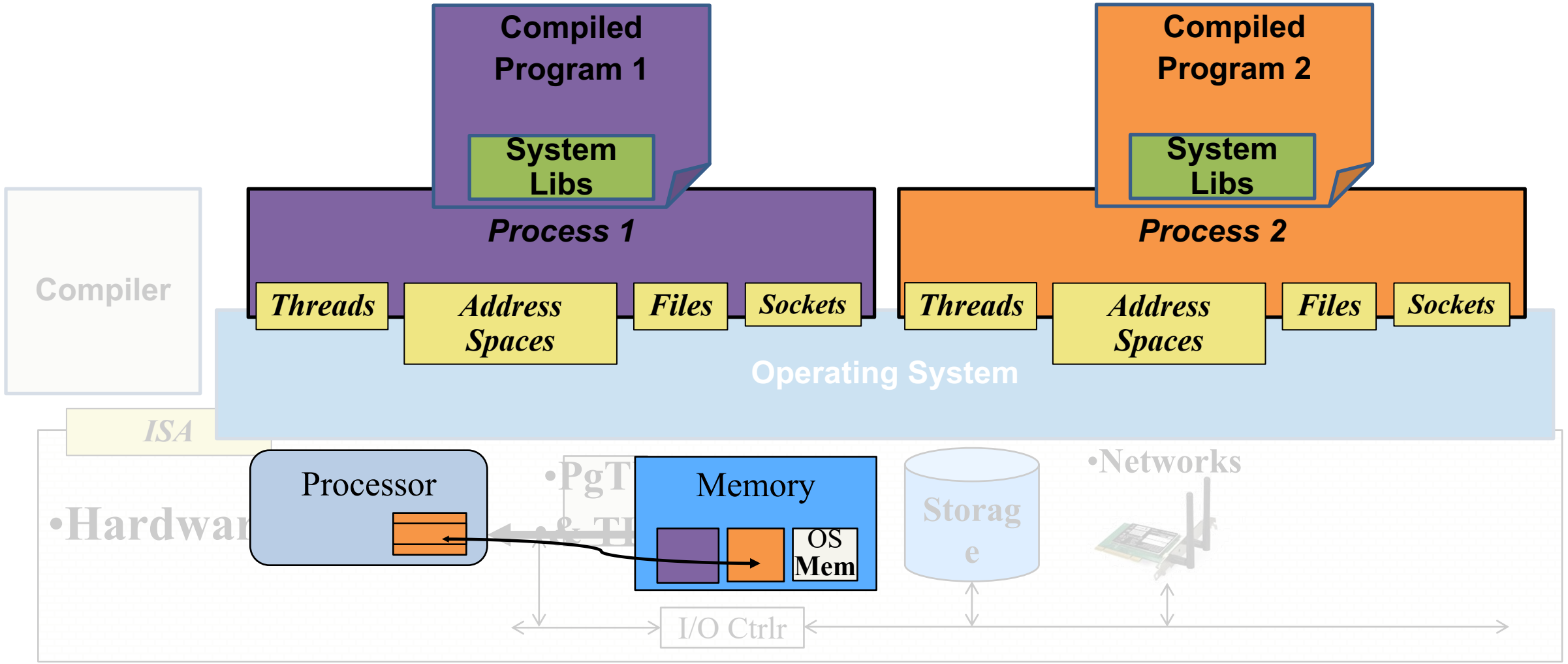
OS Basics: Running a Process



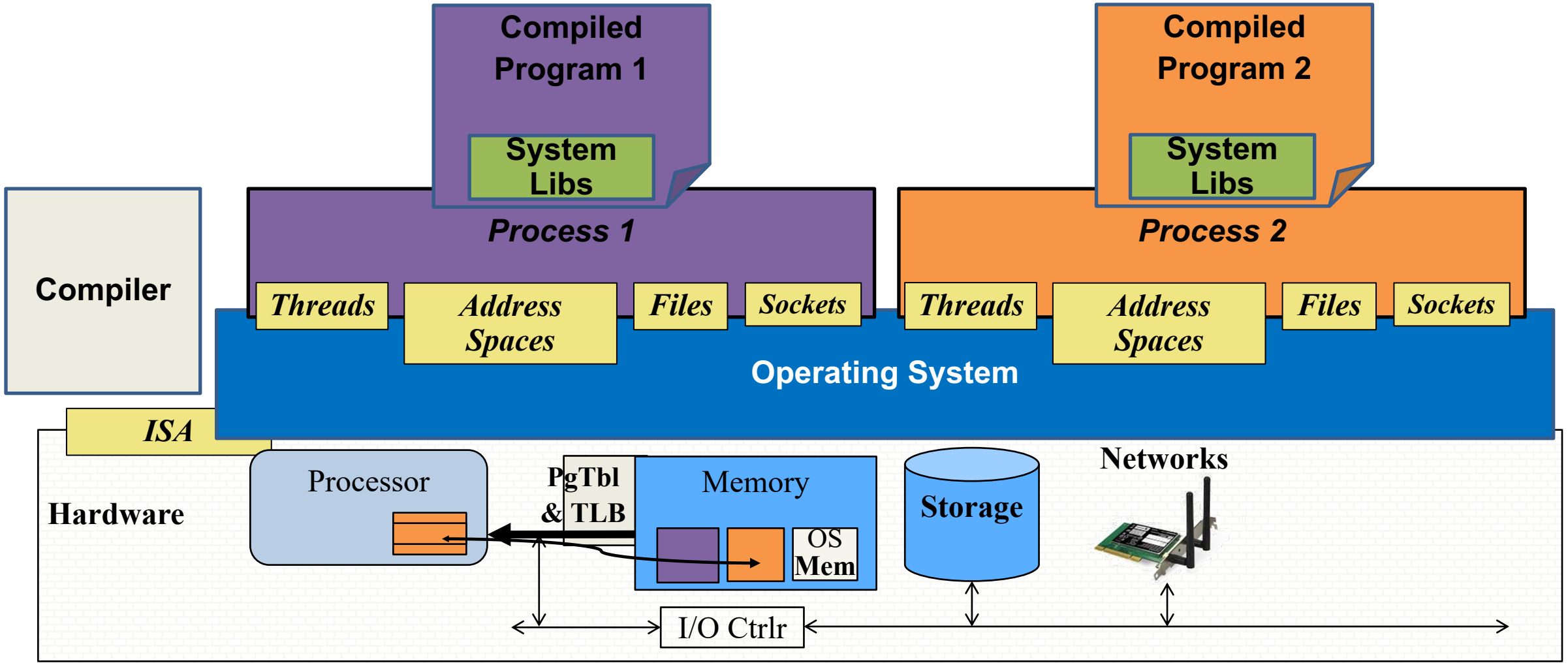
OS Basics: Switching Processes



OS Basics: Switching Processes



OS Basics: Switching Processes



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

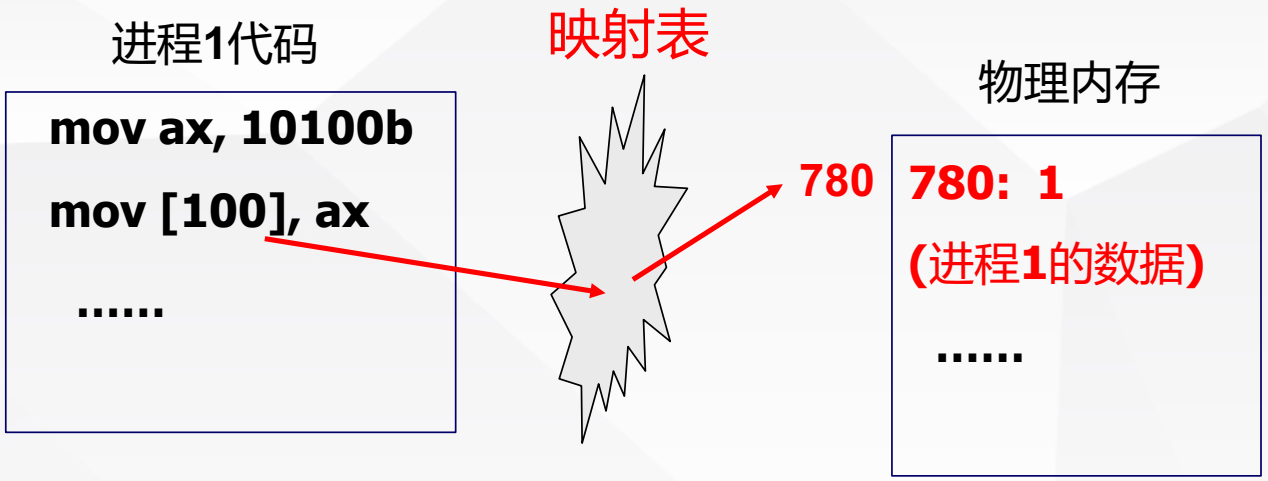
```
mov ax, 10100b
mov [100], ax
......
```

进程2代码

```
mov [100], 10200b
......
```

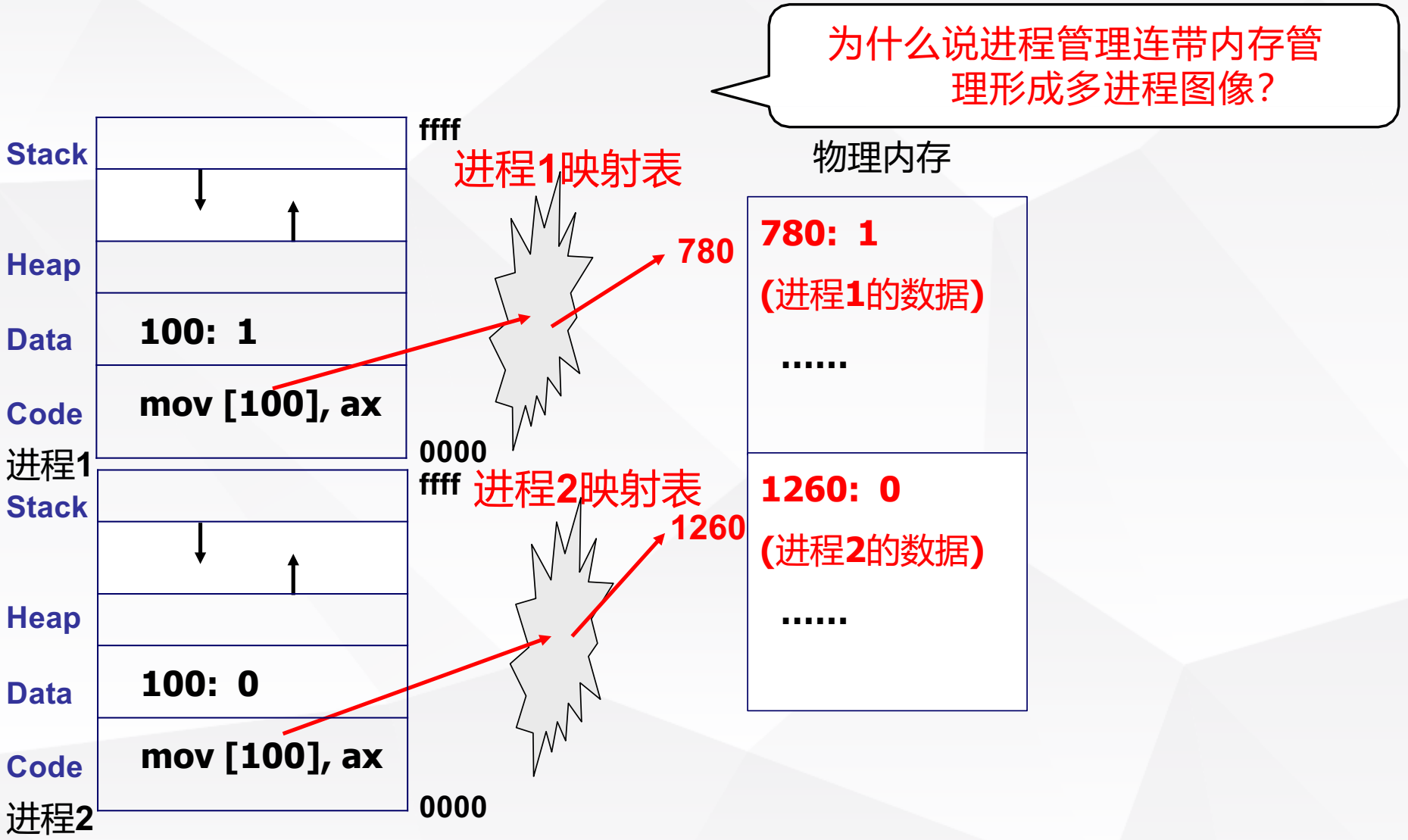
- 解决的办法：限制对地址100的读写
- 多进程的地址空间分离：内存管理的主要内容

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
进程1映射表
进程2映射表
Stack
Heap
Data
Code
进程1
100: 1
mov [100], ax
ffff
0000
780
780: 1
(进程1的数据)
.....
Stack
Heap
Data
Code
进程2
100: 0
mov [100], ax
ffff
0000
1260
1260: 0
(进程2的数据)
.....

从纸上到实际：生产者-消费者实例

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

生产者进程

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

消费者进程

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. 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;
}
```