哈爾濱工業大學（深圳）

HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

操作系统 (Operating System)

第二章：进程

陈芳林　副教授

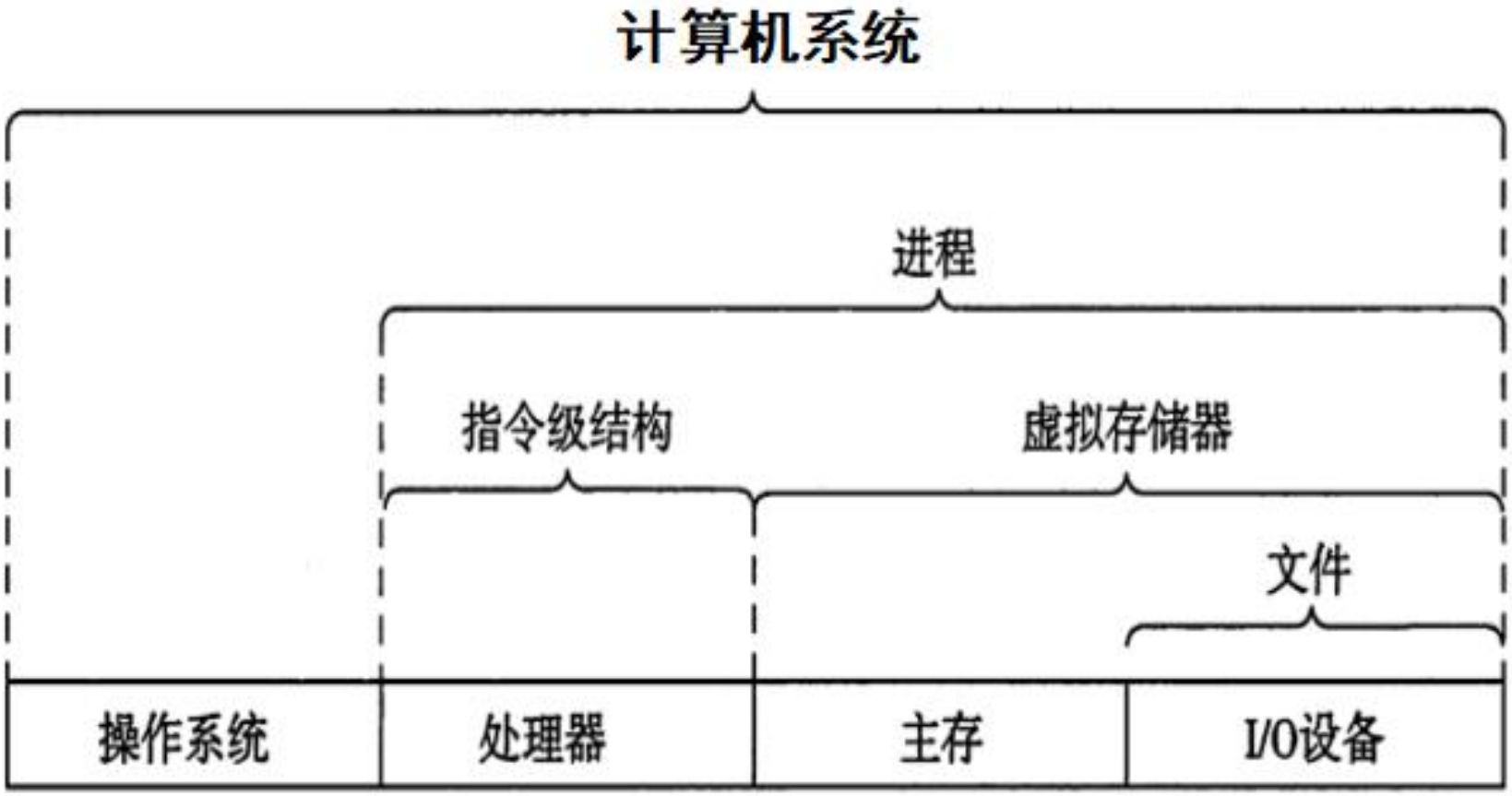
哈尔滨工业大学（深圳）

2024年秋

Email：chenfanglin@hit.edu.cn

计算机系统的抽象/虚拟化

- 1、导论与操作系统结构
- 2、进程与线程
- 3、并发与同步
- 4、处理器调度
- 5、死锁
- 6、内存管理
- 7、虚拟内存
- 8、I/O与存储
- 9、文件及文件系统



主要内容

- 2.1 进程的概念和特征
- 2.2 进程的状态、转换和控制
- 2.3 异常控制流
- 2.4 进程间通信

进程的概念 – OS发展的需要和必然

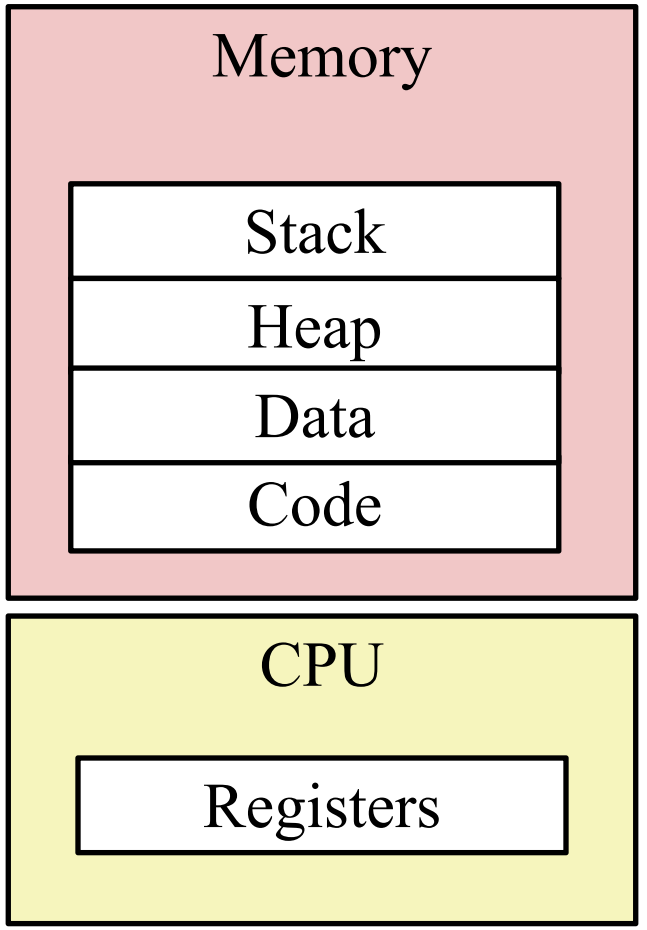
- 回顾OS发展，为什么要引入进程?
  - (1)手工操作 ← 一次装入并运行一个“作业”
  - (2)简单批处理 ← 同时装入多个“程序”，串行执行
  - (3)多道程序批处理 ← 同时装入多个程序，并发执行，仅当等待I/O时切换程序执行
  - (4)分时处理 ← 同时装入多个程序，并发执行，程序切换条件：执行的时间长度(时间片)到 or I/O等待
- 但“作业”、“程序”等概念不能有效描述程序动态轮换执行的过程
- 需要一种统一的方法监视、管理、控制处理器中不同程序的动态执行过程，“进程”的概念被引入！

进程的概念

- “进程”的概念的提出
  - Multics – 1964年，Bell实验室、MIT与GE公司共同开发
  - Multics的设计者首次提出并使用了“进程”的概念
- 进程没有严格的定义，但可以通过不同的角度去描述：
  - 一个正在执行的程序（Program） 代码段
  - 计算机中正在运行的程序的一个实例（Instance） 当前状态
  - 可以分配给处理器并由处理器执行的一个实体（Entity） 系统资源
- 由一个顺序执行的代码段、一个当前状态和一组相关系统资源所刻画的活动单元

进程的描述

- 进程提供给应用程序两个关键抽象
  - 逻辑控制流 (Logical control flow)
    - ✓每个程序似乎独占地使用CPU
    - ✓通过OS内核的上下文切换机制提供
  - 私有地址空间 (Private address space)
    - ✓每个程序似乎独占地使用内存系统
    - ✓OS内核的虚拟内存机制提供



进程的描述--进程控制块

1.什么是进程控制块？

- 描述进程与其他进程、系统资源的关系以及进程在各个不同时期所处的状态的数据结构，称为进程控制块 PCB (process control block) 。

2.进程的组成

- PCB：进程的动态特征，该进程与其他进程和系统资源的关系
- 程序与数据：描述进程本身所应完成的功能

进程控制块PCB

进程的描述--进程控制块

3. 进程控制块内容

包括四个主要部分：进程描述信息，进程控制和管理信息，资源分配清单，处理器相关信息

进程描述信息	进程控制和管理信息	资源分配清单	处理器相关信息
进程标识符（PID）	进程当前状态	代码段指针	通用寄存器值
用户标识符（UID）	进程优先级	数据段指针	地址寄存器值
	代码运行入口地址	堆栈段指针	控制寄存器值
	程序的外存地址	文件描述符	标志寄存器值
	进入内存时间	键盘	状态字
	处理器占用时机	鼠标	
	信号量使用		

XV6中的简化版PCB

```
// Per-process state
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  struct proc *parent;         // Parent process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to
  parent's wait
  int pid;                     // Process ID

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Virtual address of kernel stack
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // Page table
  struct trapframe *tf;        // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

```
// the entire kernel call stack.
struct trapframe {
  /*   0 */ uint64 kernel_satp;   // kernel page table
  /*   8 */ uint64 kernel_sp;     // top of process's kernel stack
  /*  16 */ uint64 kernel_trap;   // usertrap()
  /*  24 */ uint64 epc;           // saved user program counter
  /*  32 */ uint64 kernel_hartid; // saved kernel tp
  /*  40 */ uint64 ra;
  /*  48 */ uint64 sp;
  /*  56 */ uint64 gp;
  /*  64 */ uint64 tp;
  /*  72 */ uint64 t0;
  /*  80 */ uint64 t1;
  /*  88 */ uint64 t2;
  /*  96 */ uint64 s0;
  /* 104 */ uint64 s1;
  /* 112 */ uint64 a0;
  /* 120 */ uint64 a1;
  /* 128 */ uint64 a2;
  /* 136 */ uint64 a3;
  /* 144 */ uint64 a4;
  /* 152 */ uint64 a5;
  /* 160 */ uint64 a6;
  /* 168 */ uint64 a7;
  /* 176 */ uint64 s2;
  /* 184 */ uint64 s3;
  /* 192 */ uint64 s4;
  /* 200 */ uint64 s5;
  /* 208 */ uint64 s6;
  /* 216 */ uint64 s7;
  /* 224 */ uint64 s8;
  /* 232 */ uint64 s9;
  /* 240 */ uint64 s10;
  /* 248 */ uint64 s11;
  /* 256 */ uint64 t3;
  /* 264 */ uint64 t4;
  /* 272 */ uint64 t5;
  /* 280 */ uint64 t6;
};
```

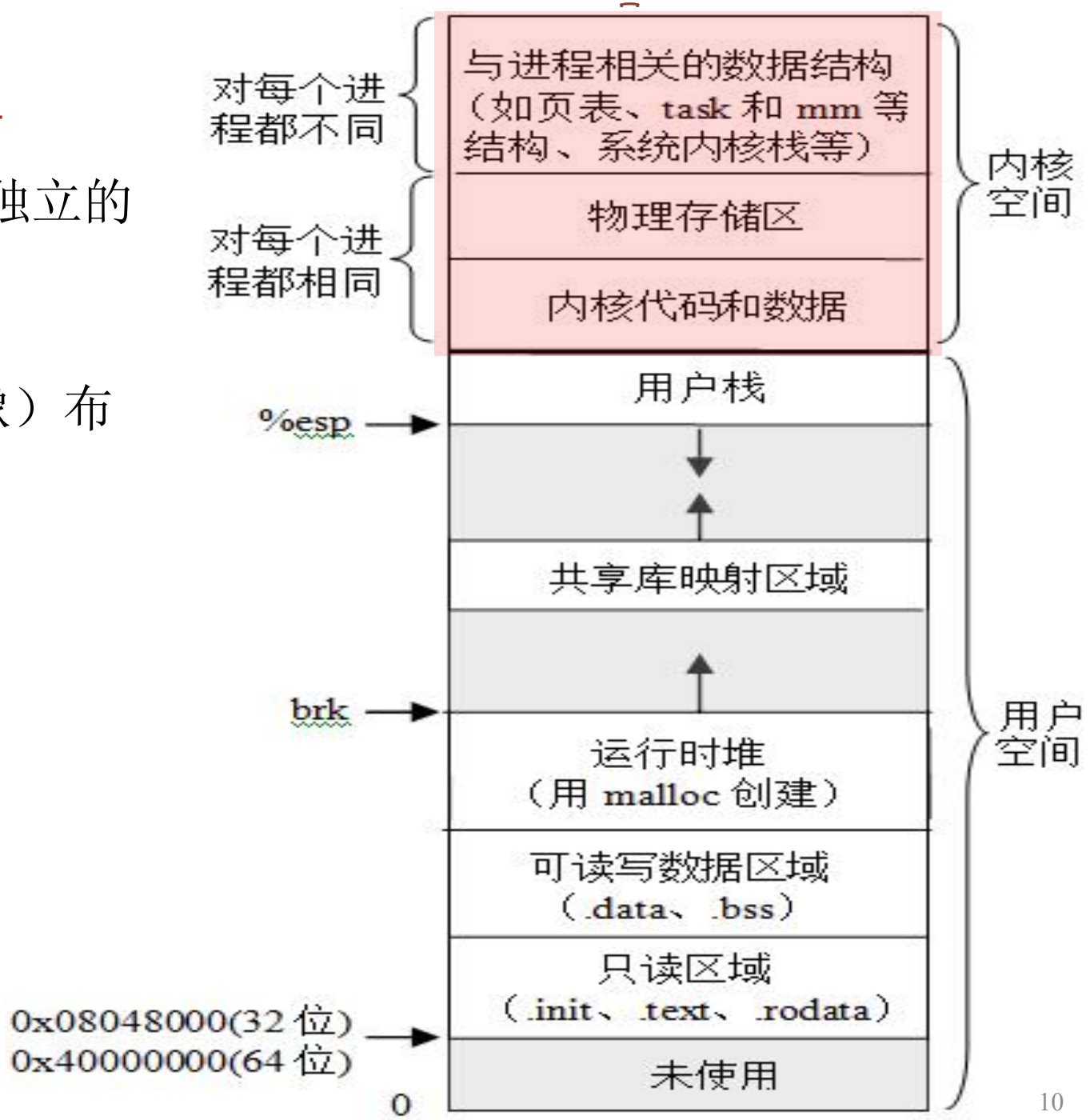
进程的描述--“上下文”

- 进程的物理实体（代码和数据）和支持运行的环境合称为进程的上下文。
- 由进程的程序块、数据块、运行时的堆和用户栈（两者通称为用户堆栈）等组成的用户空间信息被称为用户级上下文；
- 由进程标识信息、进程现场信息、进程控制信息和系统内核栈等组成的内核空间信息被称为系统级上下文；
- 处理器中各寄存器的内容被称为寄存器上下文（也称硬件上下文），即进程的现场信息。
- 在进行进程上下文切换时，操作系统把换下进程的寄存器上下文保存到系统级上下文中的现场信息位置。
- 用户级上下文地址空间和系统级上下文地址空间一起构成了一个进程的整个存储器映像

进程的存储器映像

进程的描述--地址空间

- Linux平台下，每个（用户）进程具有独立的私有地址空间（虚拟地址空间）
- 每个进程的地址空间划分（即存储映像）布局相同（如右图）



进程的特征--5个基本特征

- 动态性 动态特性表现在它因创建而产生，由调度而执行，因得不到资源而暂停执行，最后因完成或撤销而消亡（进程生命周期）
- 并发性 引入进程的目的就是为了使多个程序并发执行，提高资源利用率（主要CPU）
- 独立性 进程是一个能独立运行的基本单位，也是系统进行资源分配和调度的基本单位
- 异步性 进程以各自独立的、不可预知的速度向前推进
- 结构性 进程 = 程序 + 数据 + 进程控制块（PCB）

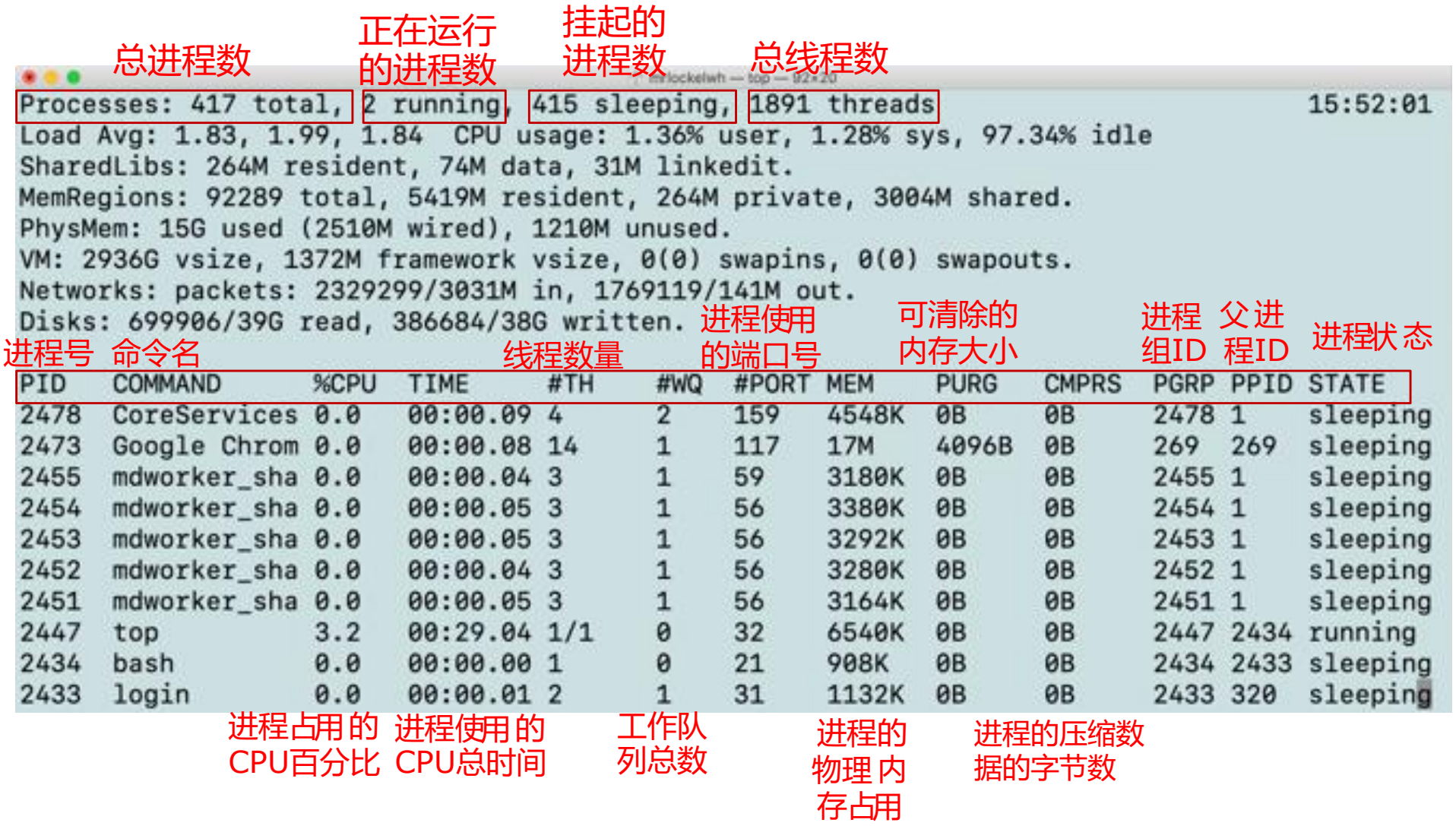
1	2	3
一个可执行的程序	该程序所需的相关数据(变量、工作空间，缓冲区等)	该程序的执行上下文(Context)

进程的特征--进程与程序的区别

- （1）从定义上看，进程是程序处理数据的过程，而程序是一组指令的有序集合
- （2）进程具有动态性、并发性、独立性和异步性等，而程序不具有这些特性；
- （3）从进程结构特性上看，它包含程序、数据和进程控制块（PCB）；
- （4）进程和程序并非一一对应：通过多次执行，一个程序可对应多个进程；通过调用关系，一个进程可执行多个程序。

进程的概念和特征--现代OS的进程

- 使用top命令查看Mac操作系统进程



进程的概念和特征--现代OS的进程

- 使用top –o + state命令对state字段进行排序，可以实时查看到正在运行的进程：

```
Processes: 420 total, 5 running, 415 sleeping, 2142 threads                    01
Load Avg: 4.80, 2.57, 2.07  CPU usage: 8.2% user, 2.40% sys, 89.56% idle
SharedLibs: 280M resident, 70M data, 50M linkedit.
MemRegions: 103377 total, 5784M resident, 221M private, 2779M shared.
PhysMem: 16G used (2184M wired), 366M unused.
VM: 3071G vsize, 1941M framework vsize, 31722(0) swapins, 39210(0) swapouts.
Networks: packets: 53434/27M in, 132300/28M out. Disks: 482419/8109M read, 285490/2532M written.

PID    COMMAND      %CPU TIME     #TH    #WQ  #PORT MEM    PURG   CMPRS PGRP  PPID  STATE
44095  vmrest       0.0  00:00.11 11/1   2    128+  9104K+ 0B     0B    766   779   running
44050  top          3.2  00:00.47 1/1    0    26    4800K+ 0B     0B    44050 33392 running
318    sysmond      0.7  00:23.31 3/1    2/1  28    7368K  0B     464K  318   1     running
245    WindowServer 2.2  03:33.70 9/1    4    3490  304M   82M+   26M   245   1     running
0      kernel_task  1.0  01:40.57 255/12 0    0     16M+   0B     0B    0     0     running
41133  Google Chrom 0.0  00:00.07 13     1    107   14M    8192B  0B    354   354   sleeping
40992  CoreServices 0.0  00:00.34 3      1    164   4368K  0B     0B    40992 1     sleeping
39982  mdworker_sha 0.0  00:00.36 3      1    59    20M    0B     0B    39982 1     sleeping
38080  FMIPClientXP 0.0  00:00.07 3      1    73    6692K  24K    0B    38080 1     sleeping
38077  siriknowledg 0.0  00:00.03 2      1    43    2788K  20K    0B    38077 1     sleeping
38063  com.apple.St 0.0  00:00.01 2      1    31    1352K  0B     0B    38063 1     sleeping
38054  AssetCacheLo 0.0  00:00.04 3      1    58    4004K  0B     0B    38054 1     sleeping
```

主要内容

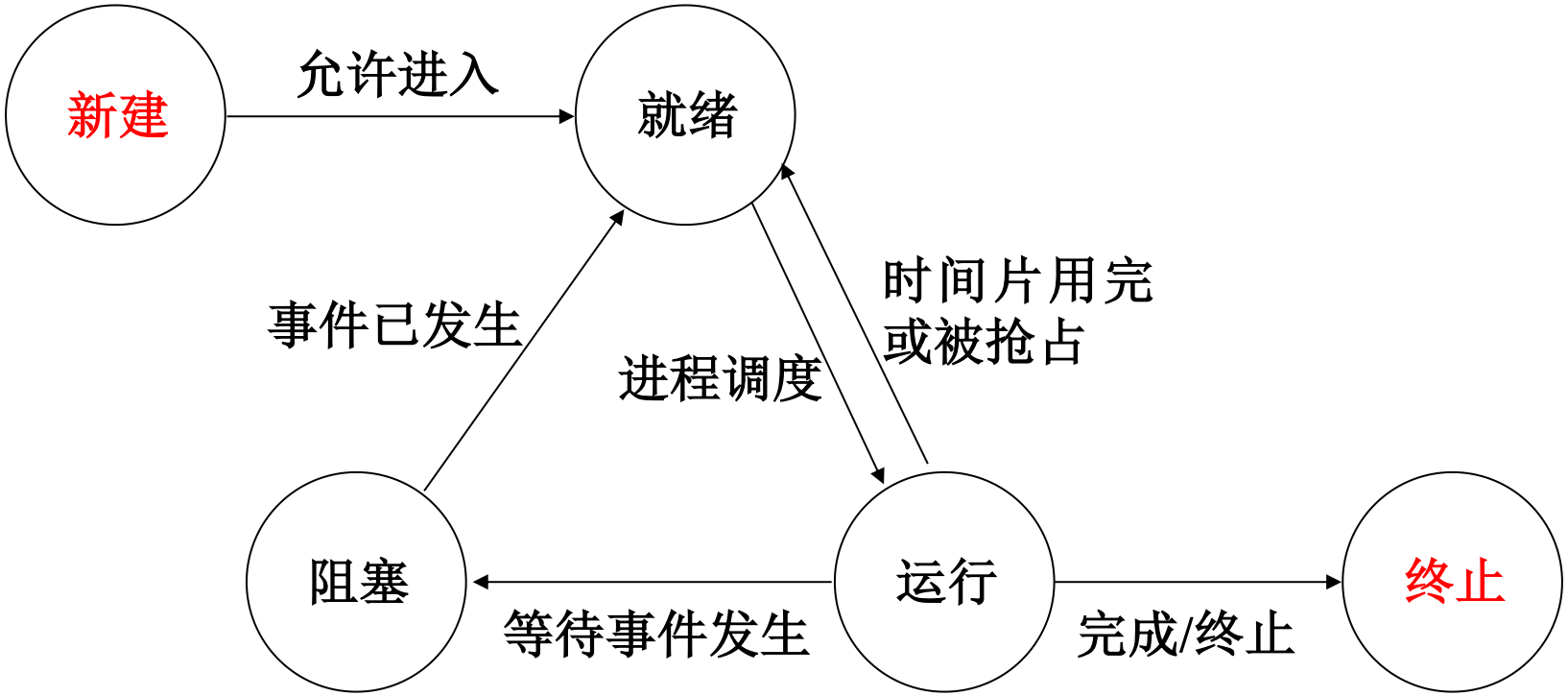
- 2.1 进程的概念和特征
- 2.2 进程的状态、转换和控制
- 2.3 异常控制流
- 2.4 进程间通信

进程的状态、转换和控制--进程的三种基本状态

- （1）就绪状态
  - 进程分配到必要的资源，等待获得CPU执行的状态。 组织成一个或多个就绪队列。即进程获得了除了CPU外所需的一切资源。
- （2）运行状态
  - 进程分配到必要的资源和CPU，在CPU上执行时的状态。
- （3）阻塞状态
  - 又称等待态。程序正在等待某一事件而暂停运行，放弃CPU而处于暂停状态。即使CPU空闲，但是并未等到所需资源时，该进程也不能运行。

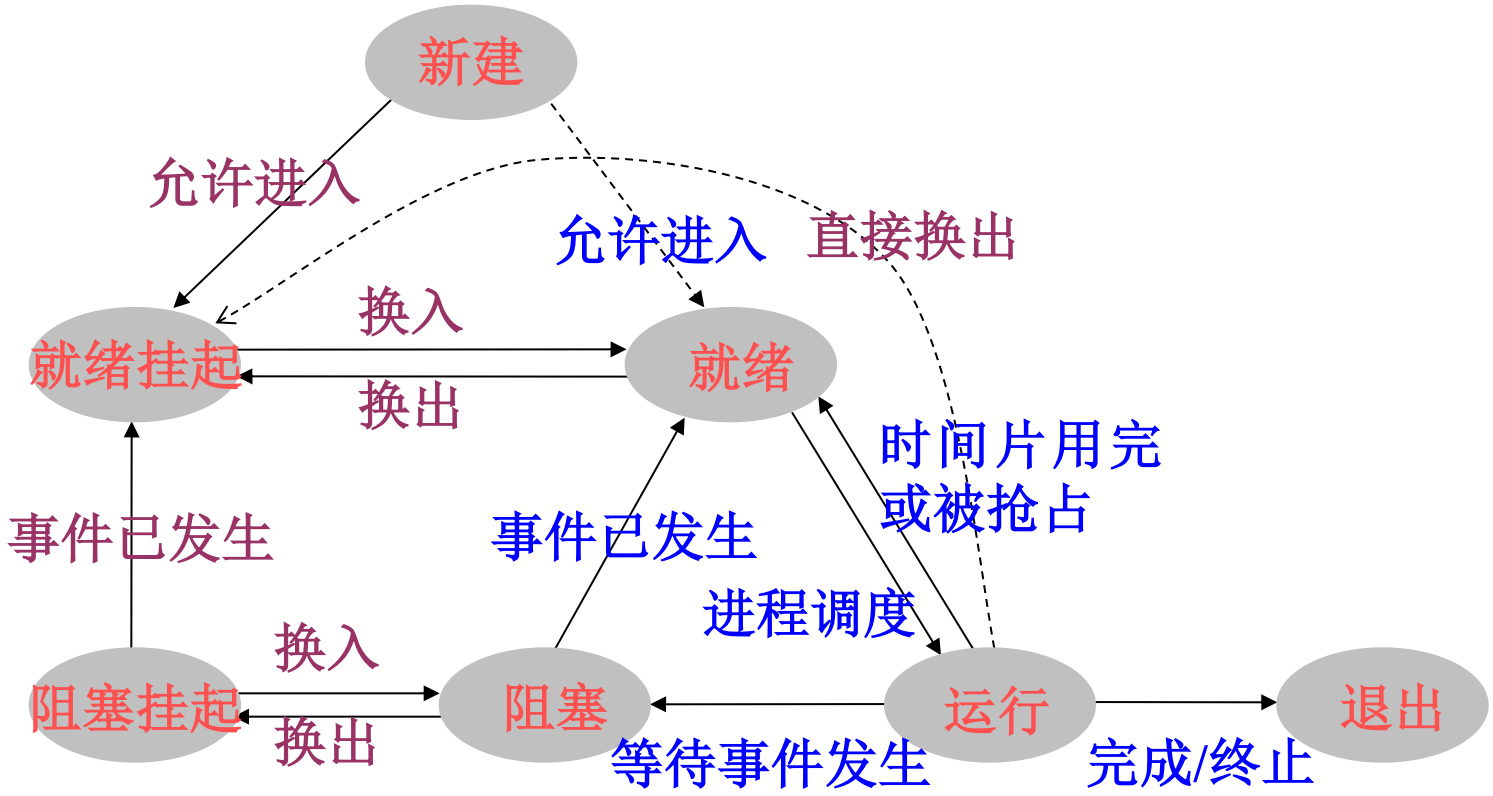
进程的状态、转换和控制--进程状态的转换

- 新建(new) – 至少建立PCB，但进程相关的其他内容可能未调入主存
- 终止(terminated) – 进程已经终止，但资源等待父进程或系统回收



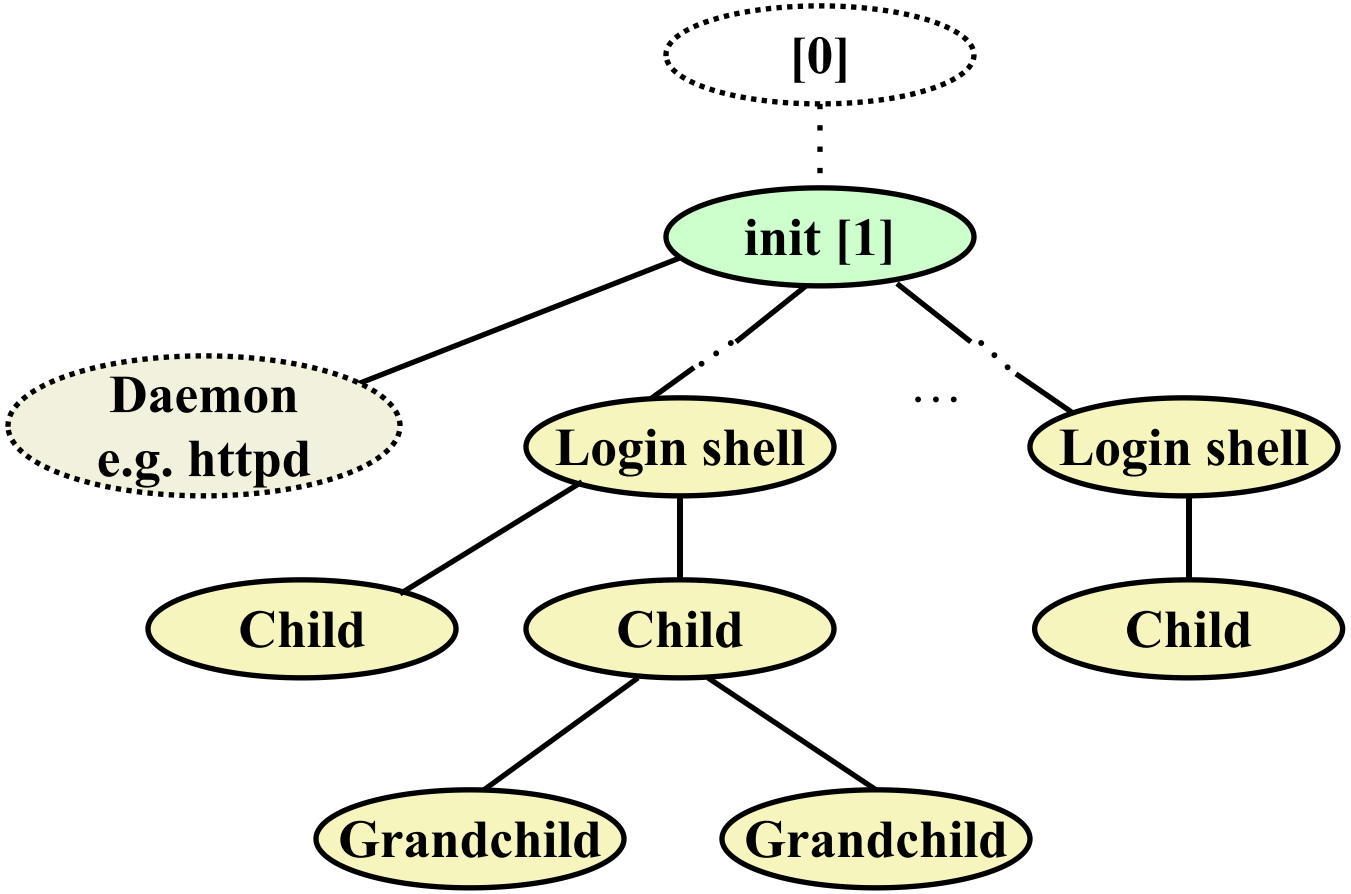
进程状态变化图（五状态）

进程的状态、转换和控制--进程的七状态变迁图



挂起状态 – 引入主存←→外存的交换机制，虚拟存储管理的基础（后面学）

Linux 进程体系



pstree 命令查看进程树

```
[root@host ~]# pstree -p
systemd(1)─┬─NetworkManager(1308)─┬─{NetworkManager}(1332)
           │                      └─{NetworkManager}(1336)
           ├─agetty(1327)
           ├─atop(9792)
           ├─auditd(1242)───{auditd}(1243)
           ├─crond(32113)
           ├─dbus-daemon(1287)
           ├─irqbalance(32354)
           ├─lvmetad(845)
           ├─master(363)─┬─bounce(22284)
           │             ├─cleanup(17834)
           │             ├─local(17307)
           │             ├─pickup(17303)
           │             ├─qmgr(365)
           │             ├─showq(23680)
           │             └─trivial-rewrite(21826)
           ├─mysqld(32450)─┬─{mysqld}(32462)
           │               ├─{mysqld}(32463)
           │               ├─{mysqld}(32464)
           │               ├─{mysqld}(32465)
           │               ├─{mysqld}(32466)
           │               ├─{mysqld}(32467)
           │               ├─{mysqld}(32468)
           │               └─{mysqld}(32469)
```

进程的状态、转换和控制--进程的控制

- **1. 进程的创建：**
  - 允许一个进程创建另一个进程。创建者为父进程，被创建者为子进程。子进程可以继承父进程所拥有的资源，当子进程被撤销时，应将其从父进程那里获得的资源归还给父进程。
- 父进程通过调用fork函数创建一个新的运行状态的子进程
  - int fork(void)
    - ✓ **返回值：返回0给子进程，返回子进程的PID给父进程，出错的话返回-1**
    - ✓ 子进程与父进程的PID不同
  - fork：**调用一次，返回两次值**
    - ✓ 调用者（父进程）：返回子进程的PID
    - ✓ 新建的子进程：返回0

进程的状态、转换和控制--进程的控制

fork1.c

```
8   int main()
9   {
10      pid_t pid_child, pid_parent;
11      int x = 1;
12
13      if ((pid_child = fork()) < 0)
14      {
15          fprintf(stderr, "%s: %s\n", "fork Error!", strerror(errno));
16      }
17      if (pid_child == 0)
18      {
19          /* Child */
20          pid_child = getpid();
21          printf("child : PID=%d x=%d\n", pid_child, ++x);
22          exit(0);
23      }
24      else
25      {
26          /* Parent */
27          pid_parent = getpid();
28          printf("parent: PID=%d x=%d\n", pid_parent, --x);
29          exit(0);
30      }
31  }
```

```
linux> ./fork1
parent: PID=60601 x=0
child : PID=60608 x=2
```

- 并发执行
  - 不能预测父子进程的执行顺序
- 相同但分离的地址空间
  - 在fork执行后，子进程和父进程都拥有一个x=1
  - 但之后对x的更改是互相独立的（copy on write）

2. 进程的终止

- 引起进程终止的事件有：
  - ✓正常结束
  - ✓异常结束
  - ✓外界干预
    - OS
    - 父进程终止
    - 父进程请求
- void exit (int status)
  - ✓终止程序
  - ✓约定: 正常结束时返回 0 , 错误时返回非0值
  - ✓另一种显式设置退出状态的方法是从主程序返回一个整数值
- Exit只会被调用一次，并且不会再次返回。

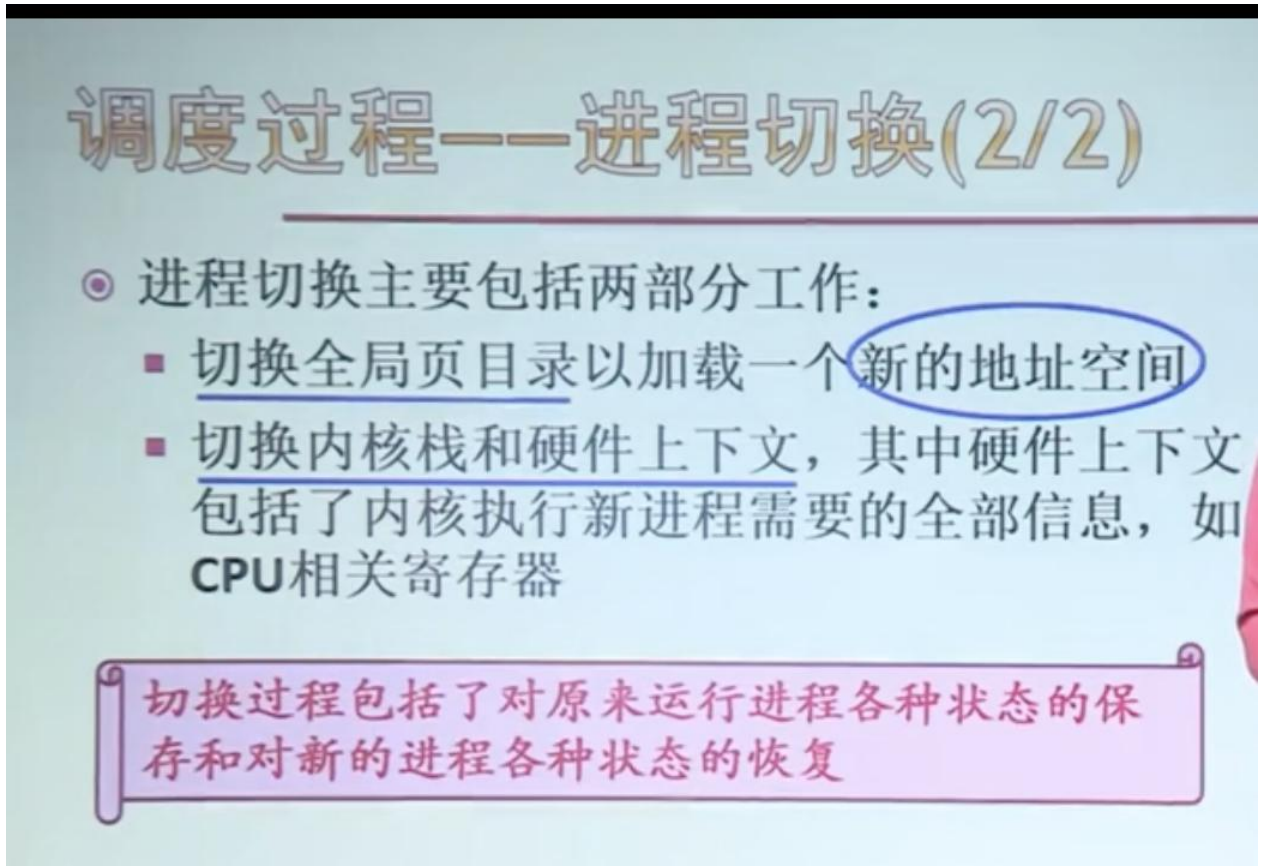
3. 进程的阻塞与唤醒

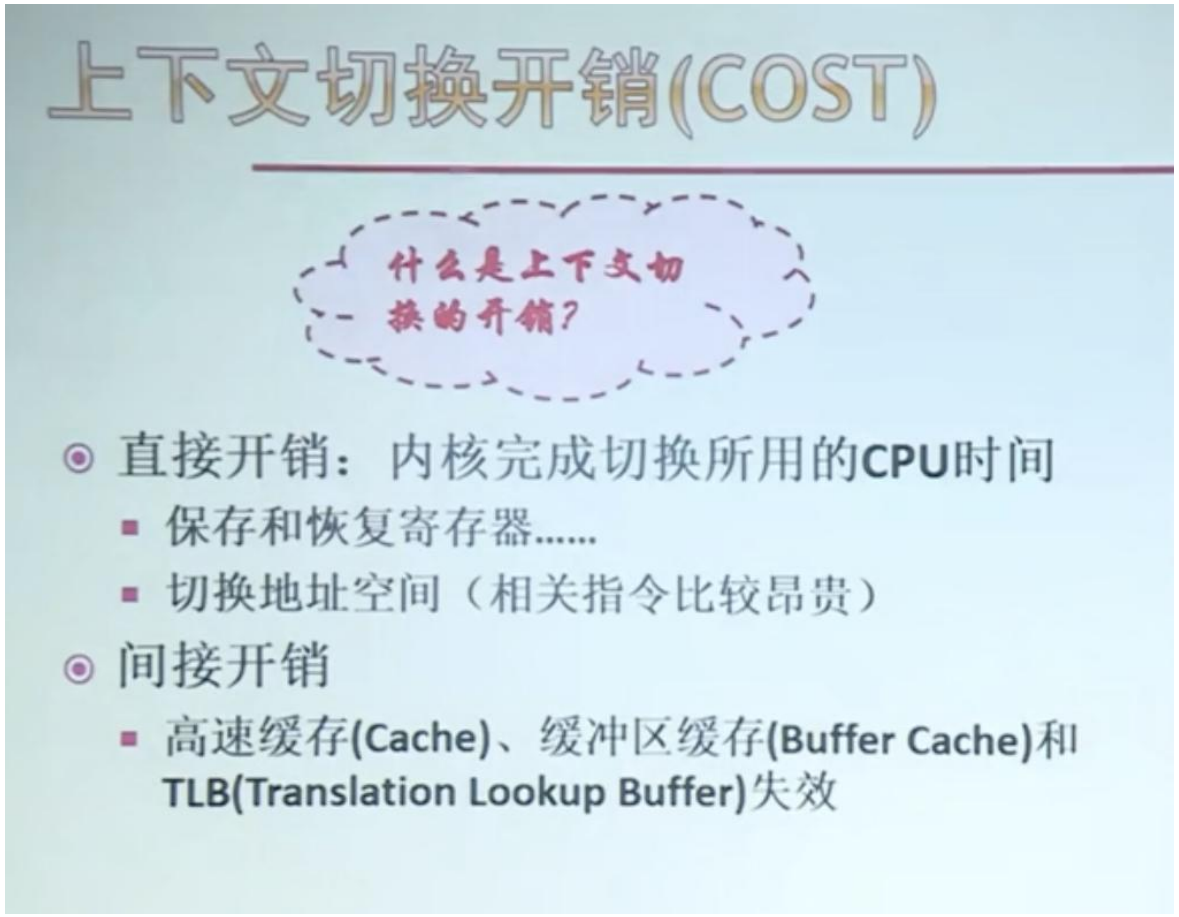
- 正在执行的进程，由于期待的事情未发生，会使自己由运行态变为阻塞态。阻塞是进程自身的一种主动行为。
- 当阻塞等待的条件被满足时，进程会被唤醒进入就绪态。唤醒操作一般是由另一个和被唤醒进程相关的合作的进程实现的。
- 可能引起进程阻塞的事件
  - ✓请求系统服务
  - ✓等待I/O操作
  - ✓等待数据到达
  - ✓无新工作可做：服务进程

4. 进程的切换

- 进程切换是指处理器从一个进程的运行转到另一个进程上运行，在这个过程中，进程的运行环境发生了实质性的变化。进程切换的过程如下：
  - ✓（1）保存处理器上下文，包括程序计数器和其他寄存器
  - ✓（2）更新PCB信息
  - ✓（3）把进程的PCB移入相应的队列，如就绪、在某事件阻塞等队列
  - ✓（4）选择另一个进程执行，并更新其PCB
  - ✓（5）更新内存管理的数据结构
  - ✓（6）恢复处理器上下文

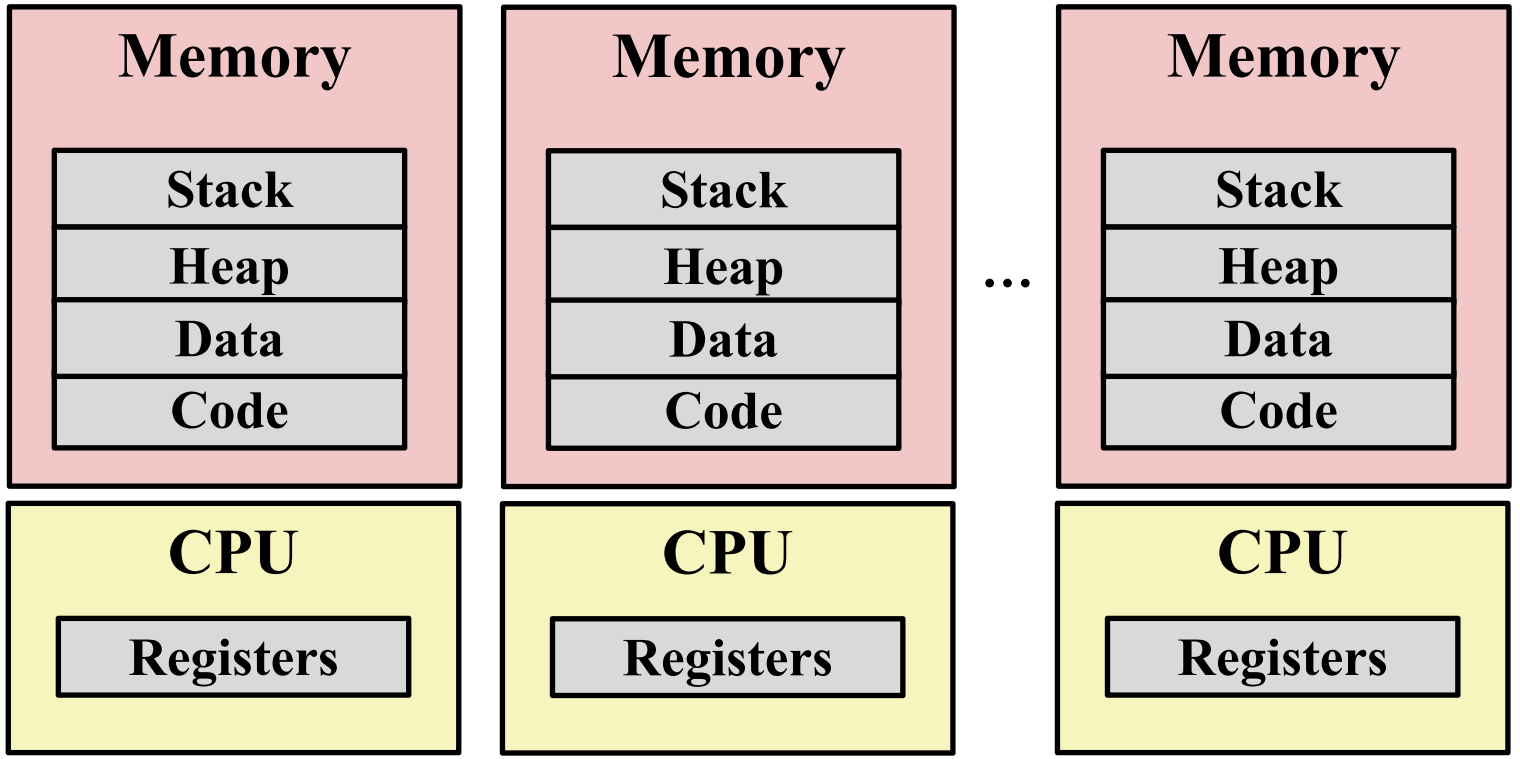
- 进程切换的开销


调度过程——进程切换(2/2)
进程切换主要包括两部分工作：
切换全局页目录以加载一个新的地址空间
切换内核栈和硬件上下文，其中硬件上下文包括了内核执行新进程需要的全部信息，如CPU相关寄存器
切换过程包括了对原来运行进程各种状态的保存和对新的进程各种状态的恢复


上下文切换开销(COST)
什么是上下文切换的开销?
直接开销：内核完成切换所用的CPU时间
保存和恢复寄存器.....
切换地址空间（相关指令比较昂贵）
间接开销
高速缓存(Cache)、缓冲区缓存(Buffer Cache)和TLB(Translation Lookup Buffer)失效

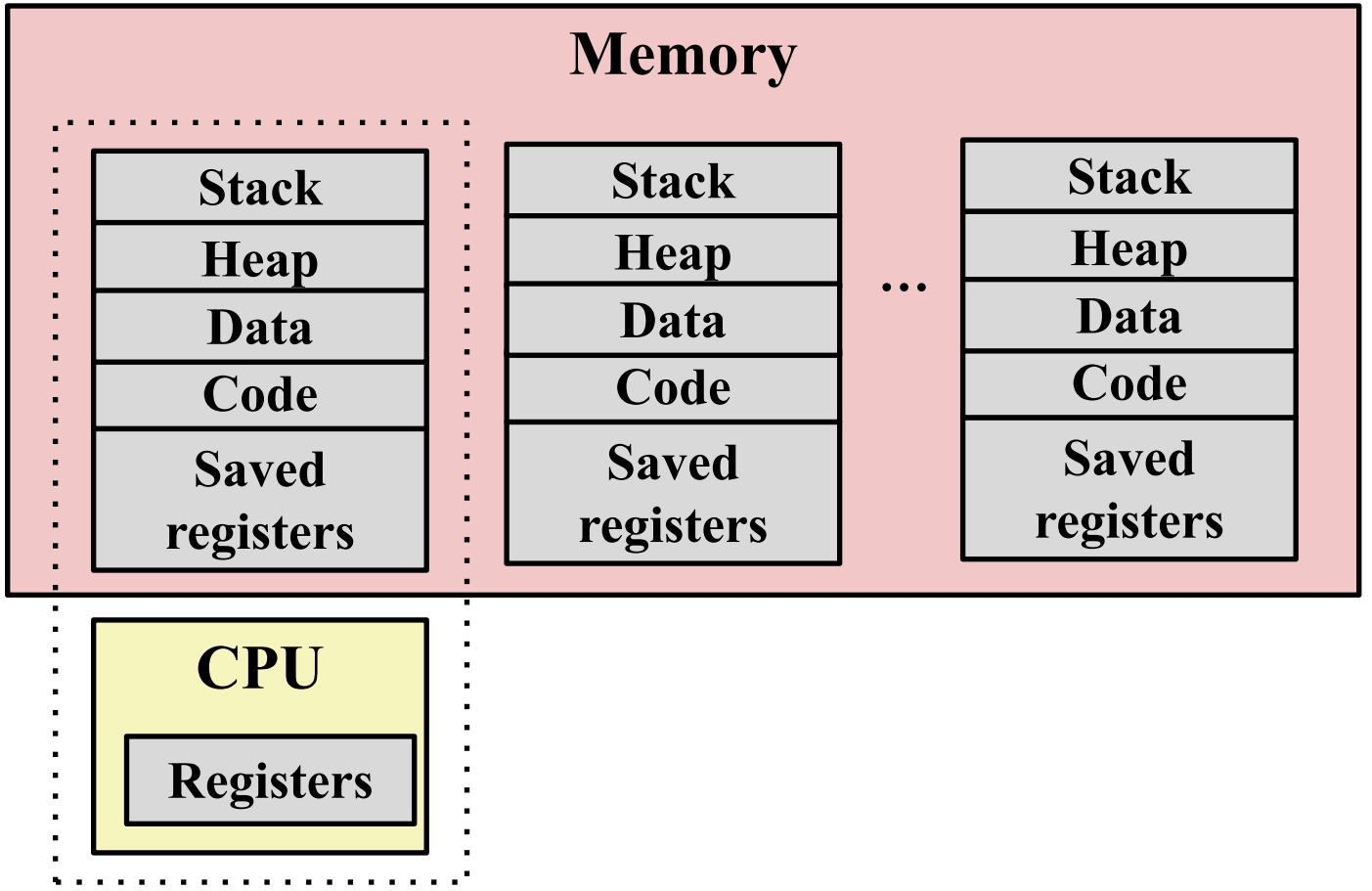
并发处理的假象(Multiprocessing: The Illusion)

- 计算机同时运行许多进程
  - 单/多用户的应用程序
    - ✓Web 浏览器、email客户端、编辑器 …
  - 后台任务(Background tasks)
    - ✓监测网络和 I/O 设备



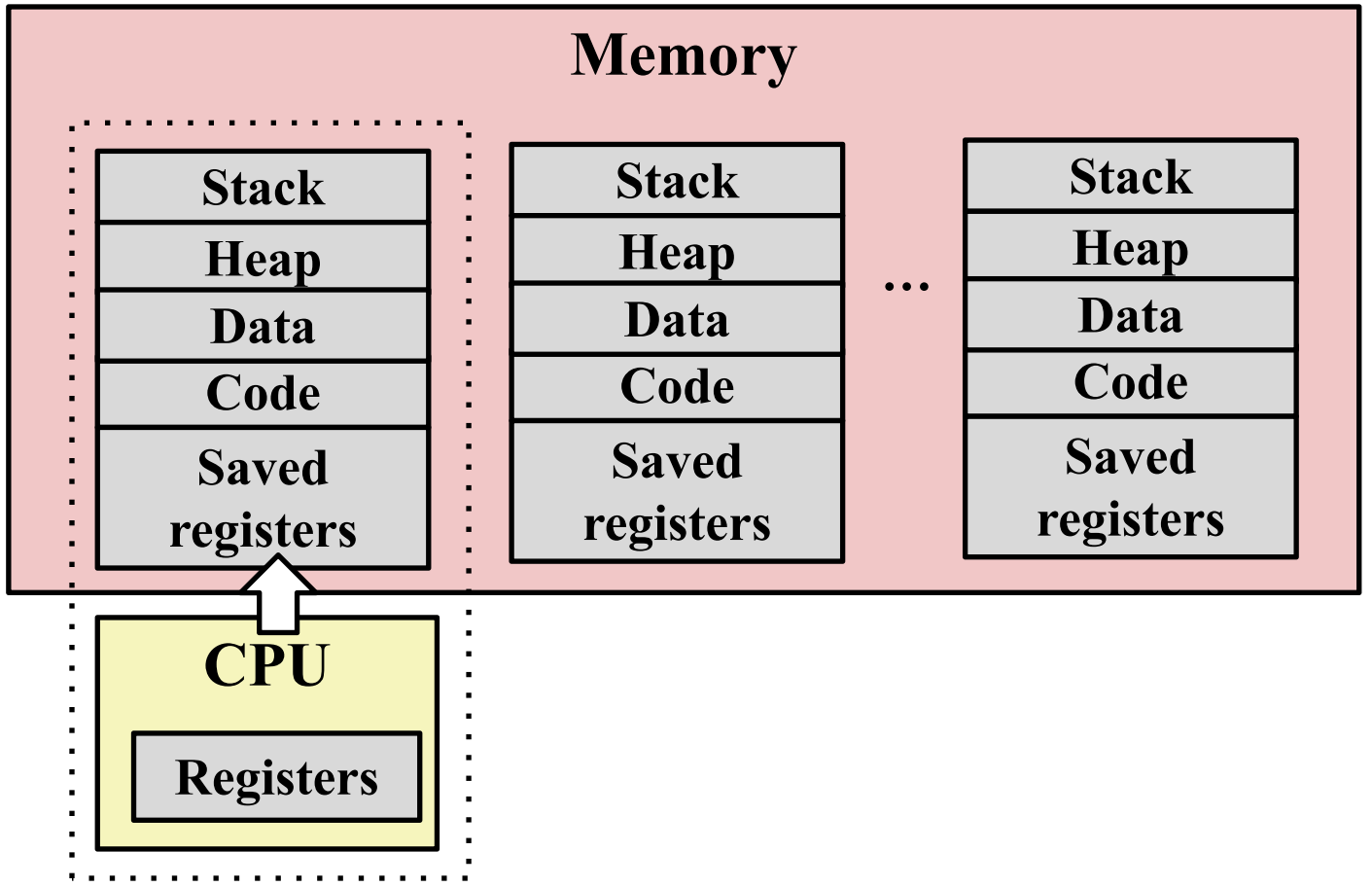
并发处理的真相

- 单处理器在并发地执行多个进程
  - 进程交错执行(多任务)
  - 地址空间由虚拟内存系统管理(后面学习)
  - 未执行进程的寄存器值保存在内存中



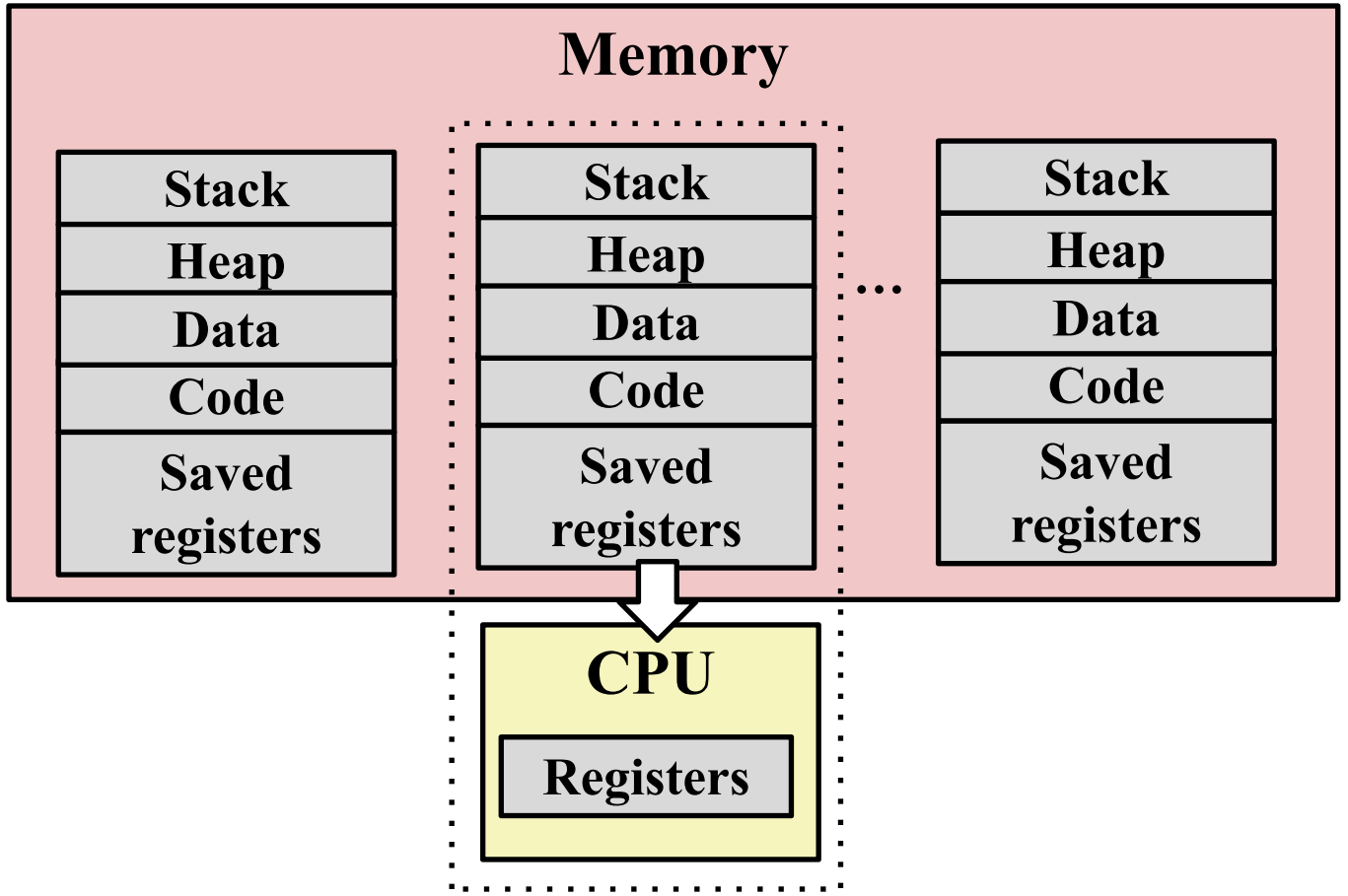
并发处理的真相

- 寄存器当前值保存到内存



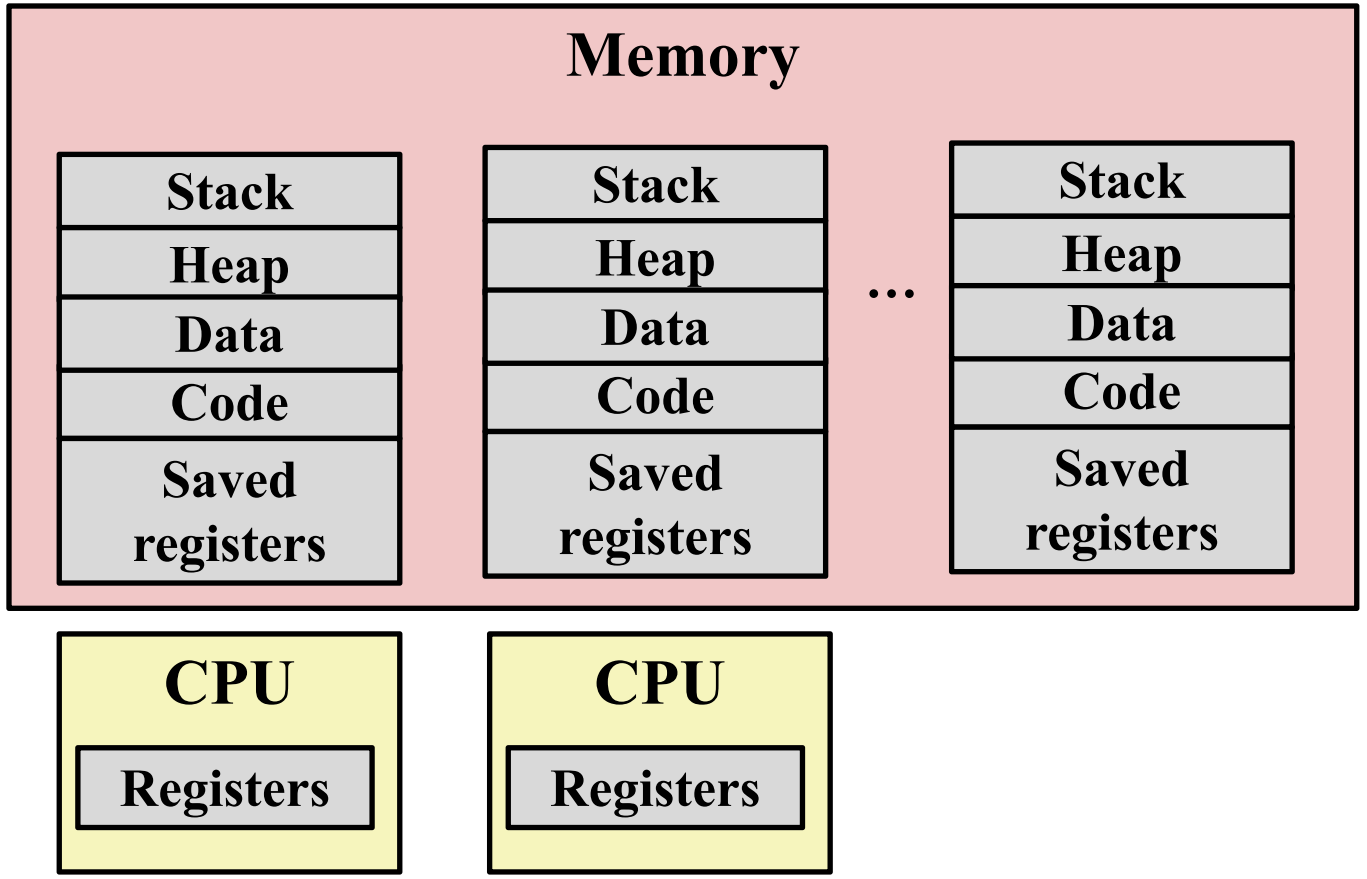
并发处理的真相

- 调度下一个进程执行
- 加载保存的寄存器组，并切换地址空间-上下文切换



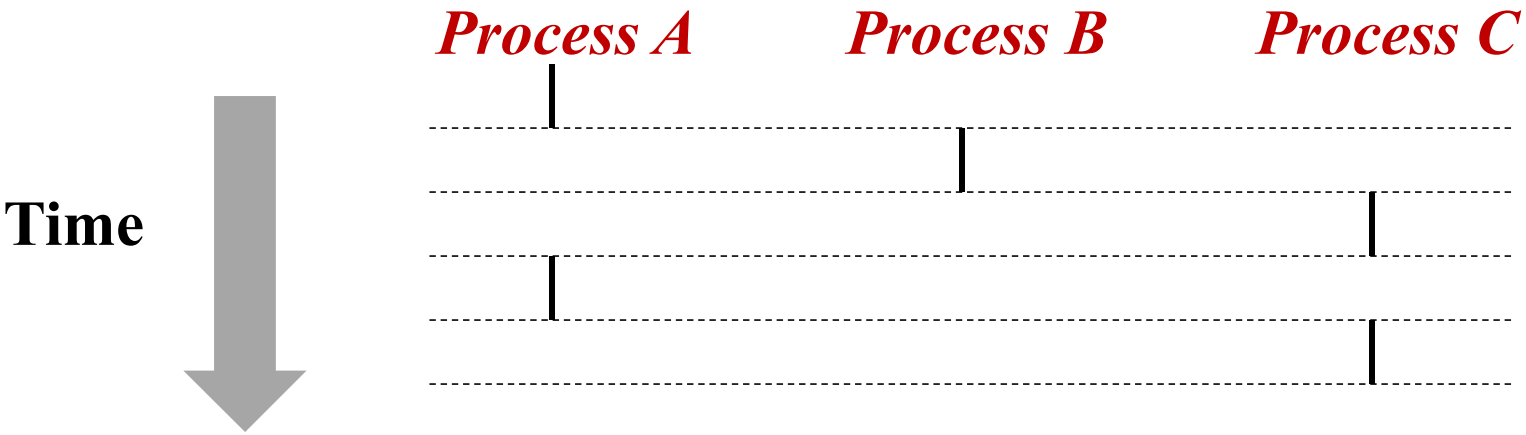
并发处理的真相

- 多核处理器
  - 单个芯片有多个CPU
  - 共享主存、有的还共享cache
  - 每个核可以执行独立的进程，kernel负责处理器的内核调度



并发进程(Concurrent Processes)

- 每个进程是个逻辑控制流
- 如果两个逻辑流在宏观时间上有重叠，则称这两个进程是并发的(并发进程)
- 否则他们是顺序的



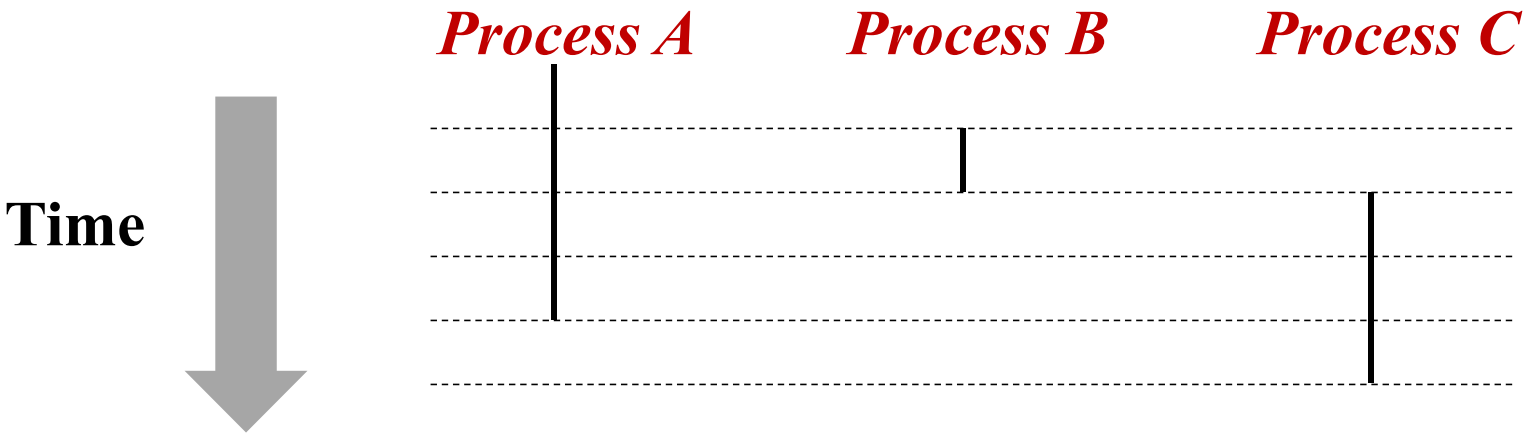
示例（单核CPU）：

A & B, A & C是并发关系

B & C是顺序关系

用户角度看并发进程

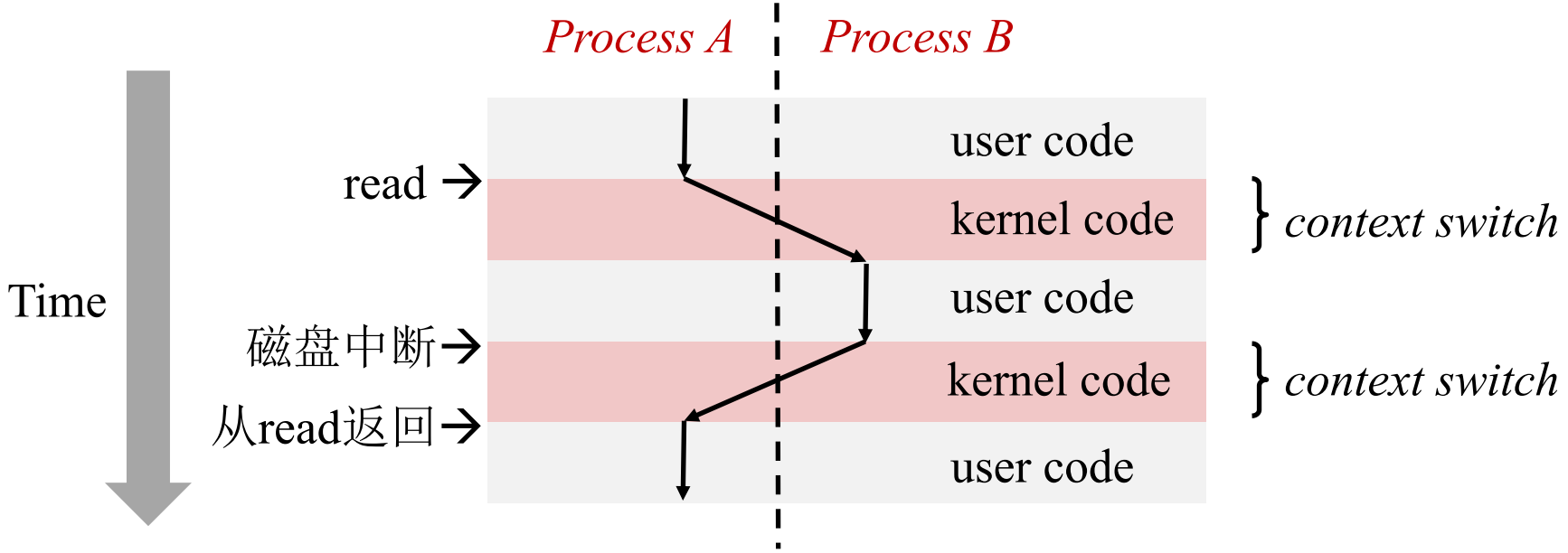
- 虽然，并发进程的控制流物理上是不相交的
- 但在宏观上，可以认为并发进程是并行运行的



- 进程内是串行、进程间是并行—乱序？--不同OS，不一样

进程的状态、转换和控制--进程的控制

- 上下文切换
  - 内核：一块共享的内存驻留操作系统代码
  - 内核并不是一个独立的进程，但是作为其他存在的进程的一部分来运行
  - 控制流的切换传递通过上下文切换进行



进程的状态、转换和控制--进程的一些其他操作

- 1.回收子进程
  - 原因
    - ✓进程终止后仍然消耗着系统资源：如: 返回标志, 变量表等
    - ✓僵尸进程：终止了但还未被回收
  - 回收（ Reaping ）
    - ✓父进程执行回收（using wait or waitpid函数）： 收到子进程退出状态
    - ✓内核删除僵尸子进程
  - 如果父进程不执行回收工作?
    - ✓如果父进程没有回收子进程并终止，则孤儿进程将会由Init进程(PID=1)进行回收

进程的状态、转换和控制--进程的一些其他操作

- 僵尸进程实例

```
void fork4() {
    if(fork() == 0) {
        /* Child */
        printf("Terminating Child, PID
                getpid());
        exit(0);
    } else {
        printf("Running Parent, PID =
                getpid());
        while(1); /* Infinite loop */
    }
}
```

```
linux> ./fork4  &
[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 <defunct>
 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
```

- 从ps结果表明子进程已废止（即成为僵尸进程）
- 杀死父进程的话，子进程将会被init回收

进程的状态、转换和控制--进程的一些其他操作

- 非终止子进程实例（孤儿进程）

```
void fork5()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
               getpid());
        while (1); /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n
               getpid());
        exit(0);
    }
}
```

```
linux> ./fork5 &
[1] 6675
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 forks
 6677 ttyp9    00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9    00:00:00 ps
```

- 即使父进程被终止，子进程仍然可以依然活跃
- 必须显式杀死子进程，否则它将会永远运行

僵尸进程与孤儿进程

- 孤儿进程：一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被init进程(进程号为1)所收养，并由init进程对它们完成状态收集工作。

- 僵尸进程：一个进程使用fork创建子进程，如果子进程退出，而父进程并没有调用wait或waitpid获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中。这种进程称之为僵尸进程。

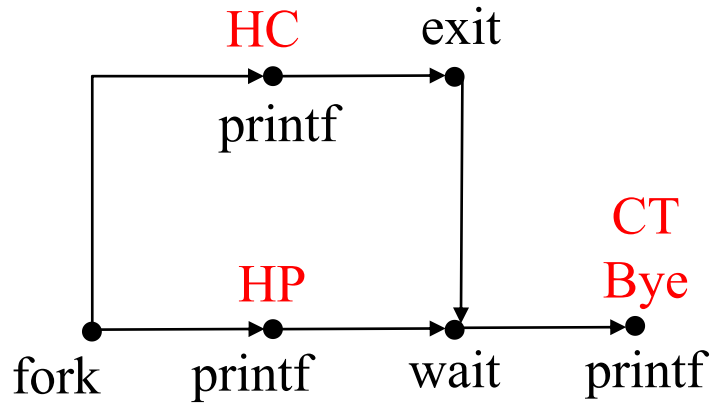
进程的状态、转换和控制--进程的一些其他操作

- wait在父进程中阻塞，等待子进程结束，如果子进程结束，则返回子进程的PID。如果没有子进程则立刻返回-1。

  - pid_t wait(int *status)

```
void fork6() {
    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");
}
```



可能的输出：

HC
HP
CT
Bye

不可能的输出：

HP
CT
Bye
HC

进程的状态、转换和控制--进程的一些其他操作

- wait 的另一个例子
  - 如果多个子进程进行竞争的话，会以任意顺序决定
  - 可以使用宏 WIFEXITED 和 WEXITSTATUS 来得到关于返回状态的信息

```
void fork7() {
    pid_t pid[N];
    int i, child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            exit(100+i); /* Child */
        }
    for (i = 0; i < N; i++) { /* Parent */
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status
%d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```

WIFEXITED(status): 若为正常终止子进程返回的状态，则为真（此参数是查看进程是否是正常退出）

WEXITSTATUS(status): 若WEXITSTATUS非零，提取子进程退出码（查看进程的退出码）

进程的状态、转换和控制--进程的一些其他操作

- waitpid: 等待一个特定的进程，阻塞当前的进程直至特定的进程被终止
  - pid_t waitpid (pid_t pid, int *status, int options)

- (1)当正常返回的时候waitpid返回收集到的子进程的进程ID
- (2)如果设置了选项WNOHANG，而调用中waitpid发现已经没有已经可以退出的子进程可收集，则返回0
- (3)如果调用中出错，则返回-1，这时errno会被设置成相应的值以指示错误所在。

```
void fork8() {
    pid_t pid[N];
    int i, child_status ;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = N-1; i >= 0; i--) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit
status %d\n", wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n",
                    wpid);
    }
}
```

进程的状态、转换和控制--进程的一些其他操作

- 让进程休眠
  - Sleep 函数:
    - ✓阻塞进程一段时间
    - ✓如果请求的时间量已经到了: return 0
    - ✓当被信号中断时，返回剩下的休眠时间秒数
  - Pause 函数:
    - ✓使调用函数进入睡眠状态，直到进程接收到信号为止

进程的状态、转换和控制--进程的一些其他操作

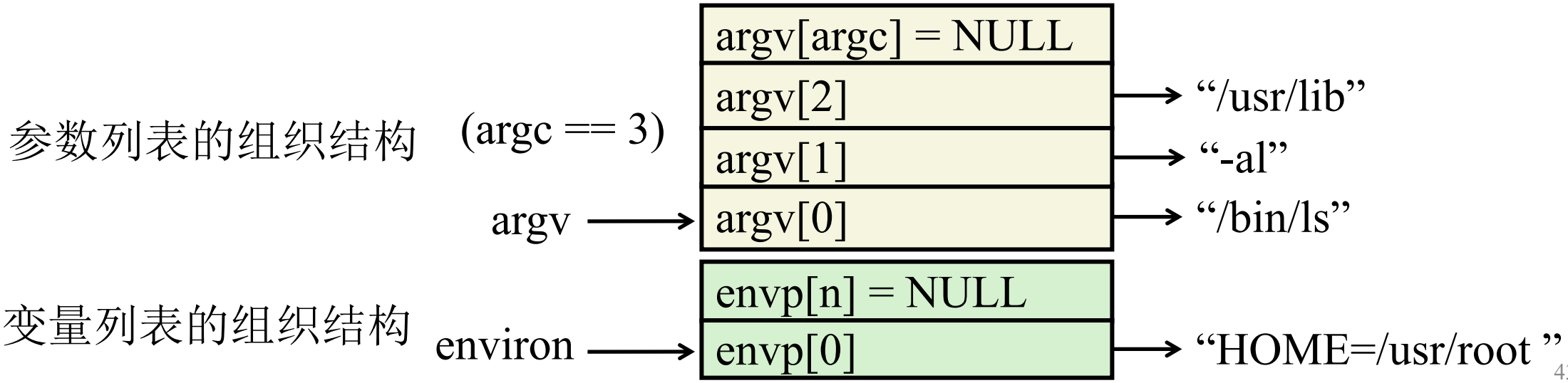
- execve：加载并运行新程序。执行shell脚本，单独的shell命令，或调用其他的程序。
  - int execve(char *filename, char *argv[], char *envp[])
  - 装载并在当前进程中运行:
    - ✓可执行文件 filename：可以是目标文件或脚本文件
    - ✓参数列表 argv：通常 argv[0]==filename
    - ✓环境变量列表 envp
      - “name=value” 字符串 (e.g., USER=droh)
      - getenv, putenv, printenv
  - 覆写 code，data和stack区域
    - ✓保存PID，打开文件和上下文信号
  - 调用一次并且永不返回，发生错误则抛出异常

进程的状态、转换和控制--进程的一些其他操作

- execve举例： int execve(char *filename, char *argv[], char *envp[])
  - 在子进程中使用当前的环境变量执行 “/bin/ls –al /usr/lib”

execve.c

```
int main()
{
    char *argv[]={"/bin/ls","-al","/usr/lib", NULL};
    /*传递给执行文件新的环境变量数组*/
    char *envp[]={"HOME=/usr/root",NULL};
    execve("/bin/ls",argv,envp);
}
```



Shell 程序

- *shell* 是一个交互型应用级程序，代表用户运行其他程序
  - sh　　最早的shell (Stephen Bourne, AT&T Bell Labs, 1977)
  - csh/tcsh　shell的变种
  - bash　　变种、缺省的Linux shell

shellex.c

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

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

shell执行一系列的读/求值步骤

读步骤读取用户的命令行，求值步骤解析命令，代表用户运行

一个简单的Shell程序：eval函数


```
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);
    bg = parseline(buf, argv);
    if (argv[0] == NULL)
        return;   /* Ignore empty lines */

    if (!builtin_command(argv)) {
        if ((pid = Fork()) == 0) {   /* Child runs user job */
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }
        /* Parent waits for foreground job to terminate */父进程等待前台子进程结束
       if (!bg) {            //fg或bg或&
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
        else
            printf("%d %s", pid, cmdline);
    }
    return;
}
```

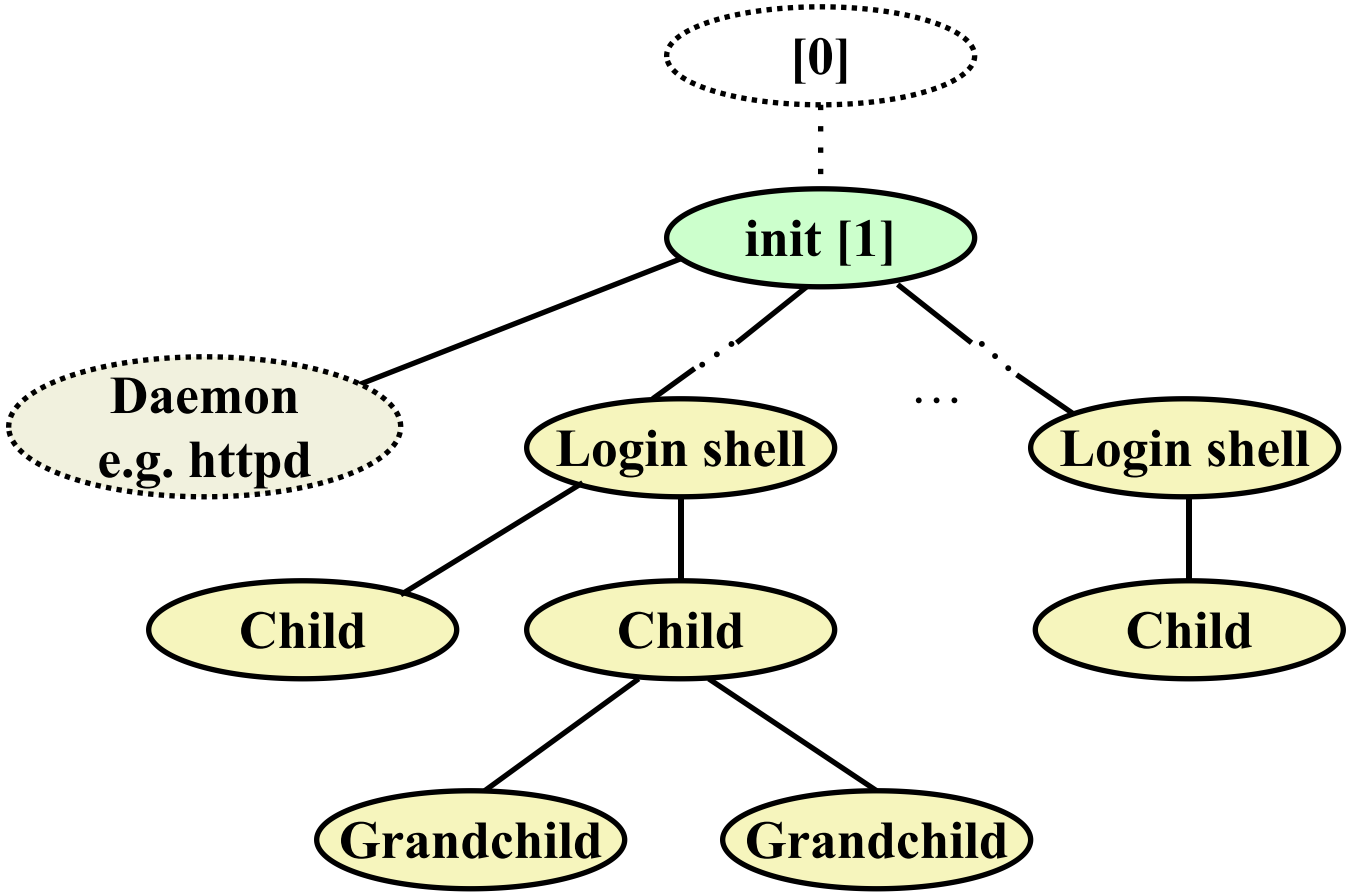
```
/* If first arg is a builtin command, run it and return true */
int builtin_command(char **argv)
{
    if (!strcmp(argv[0], "quit")) /* quit command */
        exit(0);
    if (!strcmp(argv[0], "&"))    /* Ignore singleton & */
        return 1;
    return 0;                     /* Not a builtin command */
}
```

shellex.c

为什么这样设计API

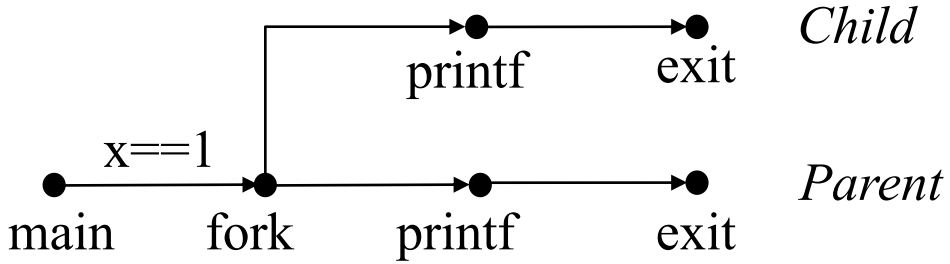
- 进程的创建操作API为什么如此怪异？ fork() 调用1次，返回2次
  - fork()和exec() 组合完成一个新的进程的完整创建过程
  - 为什么不像Windows提供的 CreateProcess()函数一样，完成进程创建
- 原因
  - 调用fork() 之后创建的子进程和父进程保持一致
    - ✓实现父进程与子进程的通信。如：父进程提前为子进程设置某些参数
    - ✓利用这种一致性可以减少创建开销、节省空间。在实际任务中，子进程需要和父进程保持相同，至少部分相同
  - fork()和exec()分离后，系统可以在exec()之前做更多的事情，让shell方便地实现很多有用的功能
    - ✓shell可以利用文件描述符实现重定向
    - ✓shell可以利用pipe()系统调用实现管道

Linux 进程体系（回顾）



进程的状态、转换和控制-- 进程图

- 进程图是刻画并发程序语句偏序关系的一种有效工具
  - 每个节点都是一条语句的执行
  - a → b 表示a在b之前发生
  - 边可以用变量的当前值来做标记
  - printf 可以用输出来进行标记
  - 每个进程图都由一个没有入边的节点开始



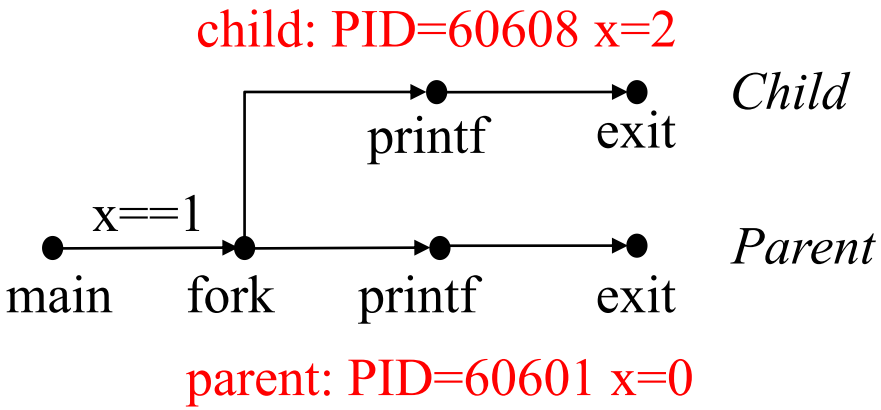
- 图中的任何一个拓扑排序对应于一个可行的全序排列
  - 所有边从左到右指向的顶点的总顺序

进程的状态、转换和控制-- 进程图

fork1.c

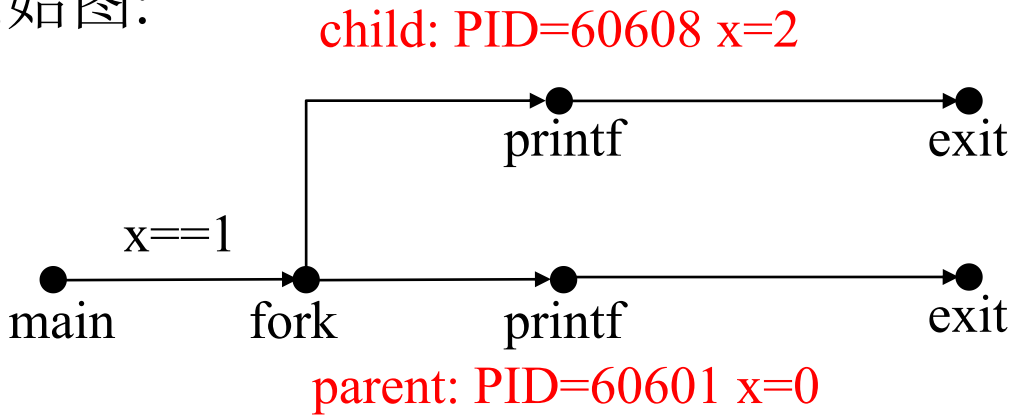
```c
int main()
{
    pid_t pid_child, pid_parent;
    int x = 1;

    if ((pid_child = fork()) < 0)
    {
        fprintf(stderr, "%s: %s\n", "fork Error!", strerror(errno));
    }
    if (pid_child == 0)
    {
        /* Child */
        pid_child = getpid();
        printf("child : PID=%d x=%d\n", pid_child, ++x);
        exit(0);
    }
    else
    {
        /* Parent */
        pid_parent = getpid();
        printf("parent: PID=%d x=%d\n", pid_parent, --x);
        exit(0);
    }
}
```

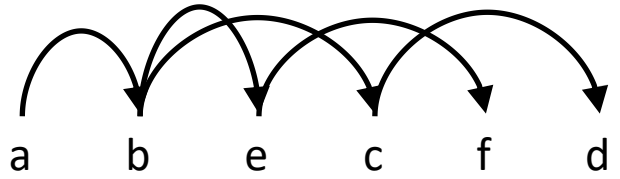


进程的状态、转换和控制-- 进程图

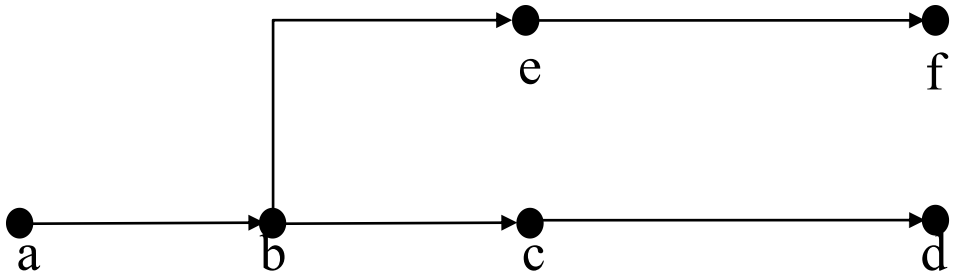
- 原始图:



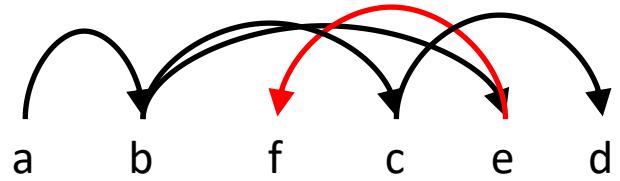
可行的整体顺序：



- 重标记图:



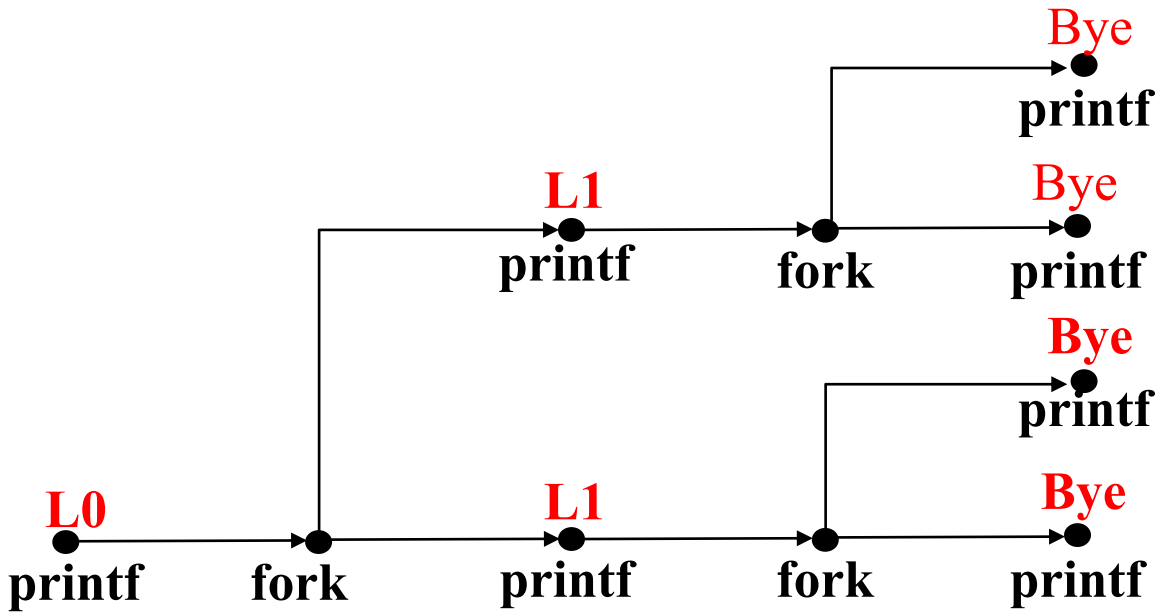
不可行的整体顺序：



进程的状态、转换和控制-- 进程图

- fork实例: 两个连续fork调用

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



可能的输出:

L0
L1
Bye
Bye
L1
Bye
Bye

不可能的输出:

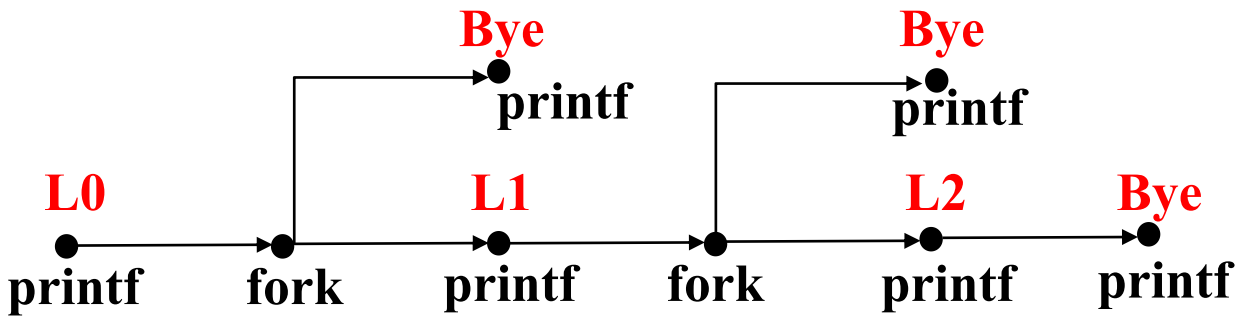
L0
Bye
L1
Bye
L1
Bye
Bye

进程的状态、转换和控制-- 进程图

- fork实例: 父进程的嵌套fork调用

fork3.c

```
void fork3()
{
    printf("L0\n");
    if (fork()!= 0) {
        printf("L1\n");
        if (fork()!= 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```



可能的输出:

L0
L1
Bye
Bye
L2
Bye

不可能的输出:

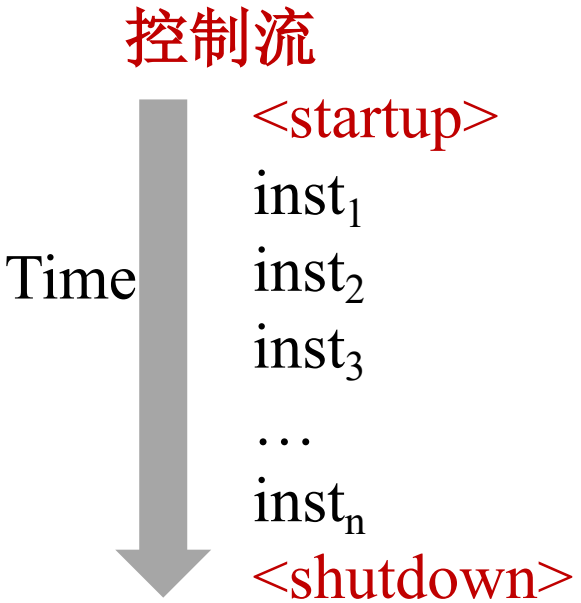
L0
Bye
L1
Bye
Bye
L2

主要内容

- 2.1 进程的概念和特征
- 2.2 进程的状态、转换和控制
- 2.3 异常控制流
- 2.4 进程间通信

异常控制流--CPU的控制流

- 处理器的工作:
  - 从开机到结束运行, 一个cpu所作的仅仅是读取并且执行（或者说是解释）一连串的指令，在某个时刻只会运行一条指令
  - 这样一个指令的运行序列就是cpu的控制流(Control Flow)



每个进程都是个逻辑控制流

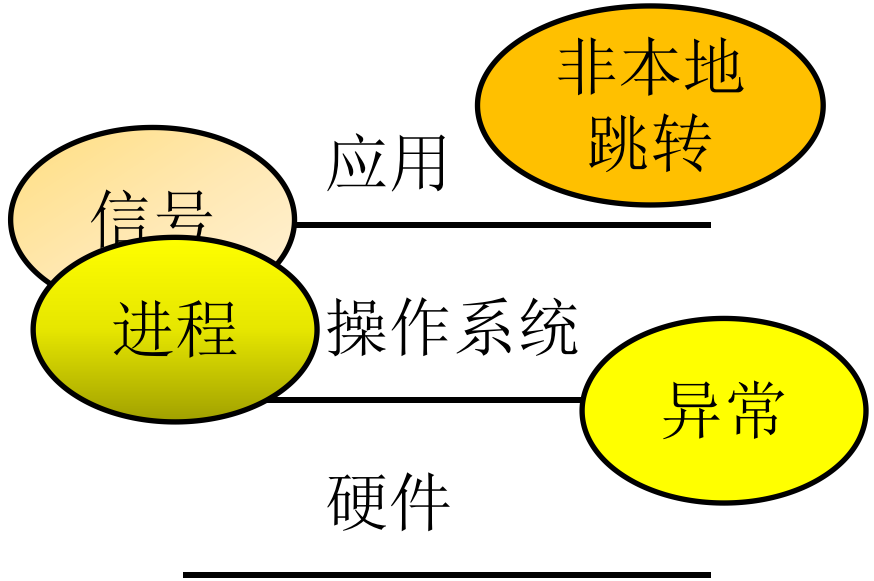
异常控制流--CPU的控制流

- 目前，两种机制可以改变控制流：
  - 跳转和分支
  - 调用和返回

  这样的机制会导致程序状态（program state）发生改变

- 对一个有用的系统来说是不够的：很难改变系统状态（system state），因为
  - 需要从硬盘或者是网络通信处读取数据
  - 运行异常，如零除操作
  - 用户操作中断，如在键盘上按Control-C
  - 系统定时器触发

- 因此，系统需要“异常控制流”机制

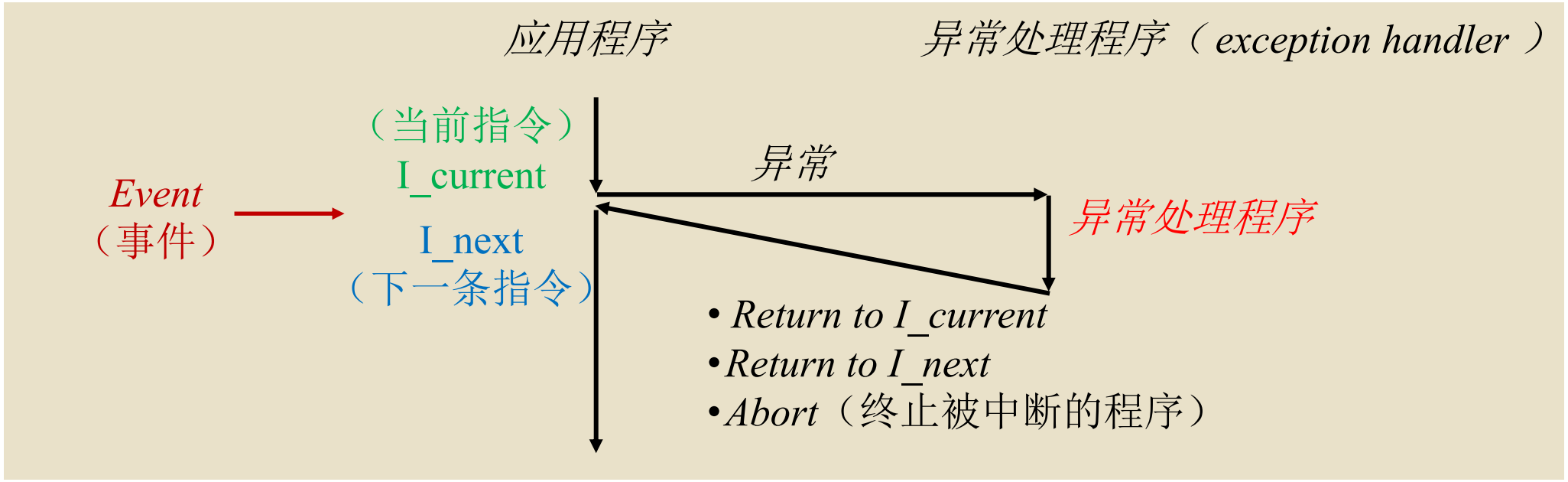


异常控制流--CPU的控制流

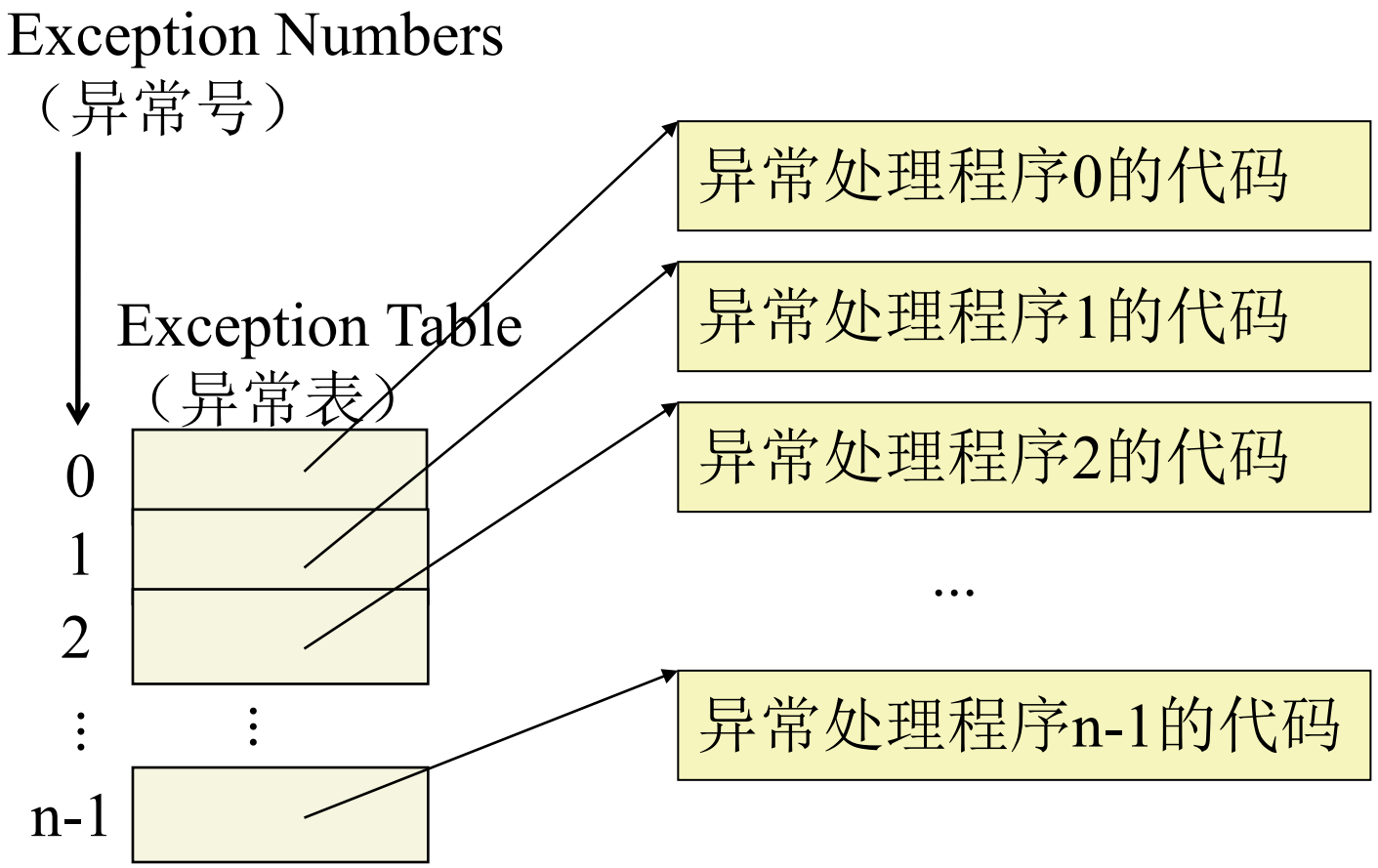
- 存在于计算机系统的各个层次中
  - 低层次机制
    - ✓对系统时间作出响应并改变程序的控制流（会导致系统状态的改变）
    - ✓通过硬件和操作系统软件的组合来实现
  - 高层次机制
    - ✓进程上下文切换（Process context switch）：通过操作系统软件和硬件时钟定时器来实现
    - ✓信号（Signals）：通过操作系统软件来实现(应用层)
    - ✓全局跳转（Nonlocal jumps）: setjmp() and longjmp()：通过C语言运行时库实现

异常控制流--CPU的控制流

- 异常（Exception）是响应某些事件（即处理器状态更改）进而将控制权转移到OS内核（kernel）
  - 内核是OS的内存驻留部分
  - 事件举例: 零除指令, 算术溢出, 缺页, I/O请求, 用户输入中断指令等



异常控制流--CPU的控制流



- 每种类型的异常有唯一的异常号
- k 为异常表的索引(又称为 中断向量)
- 每当异常k发生时，异常处理程序k被调用

异常的类别

类别	原因	异步/同步	返回行为
中断（Interrupt）	来自I/O设备的信号	异步	总是返回到下一条指令
陷阱（Trap）	有意的异常	同步	总是返回到下一条指令
故障（Fault）	潜在的可恢复的错误	同步	可能返回到当前指令
终止（Abort）	不可恢复的错误	同步	不会返回

异步异常控制流--中断异常

- 中断异常：由处理器外部的事件产生
  - 通过设置cpu的中断引脚来触发
  - 中断处理程序返回的位置是下一条指令
- 例如:
  - 时钟中断
    - ✓每过一段时间（如几毫秒），外部定时器芯片触发中断
    - ✓被内核用来从用户程序取回控制权
  - I/O中断
    - ✓在键盘中输入Ctrl-C
    - ✓从网络传输收到包
    - ✓硬盘中的数据到达

同步异常控制流--陷阱异常

- 陷阱异常：由执行当前指令的结果的触发
  - 有意的异常。如：系统调用（ system calls）有一个唯一的ID号，断点
  - 将控制流转移到下一条指令

Number	*Name*	*Description*
0	read	Read file
1	write	Write file
2	open	Open file
3	close	Close file
4	stat	Get info about file
57	fork	Create process
59	execve	Execute a program
60	_exit	Terminate process
62	kill	Send signal to process

同步异常控制流--故障异常

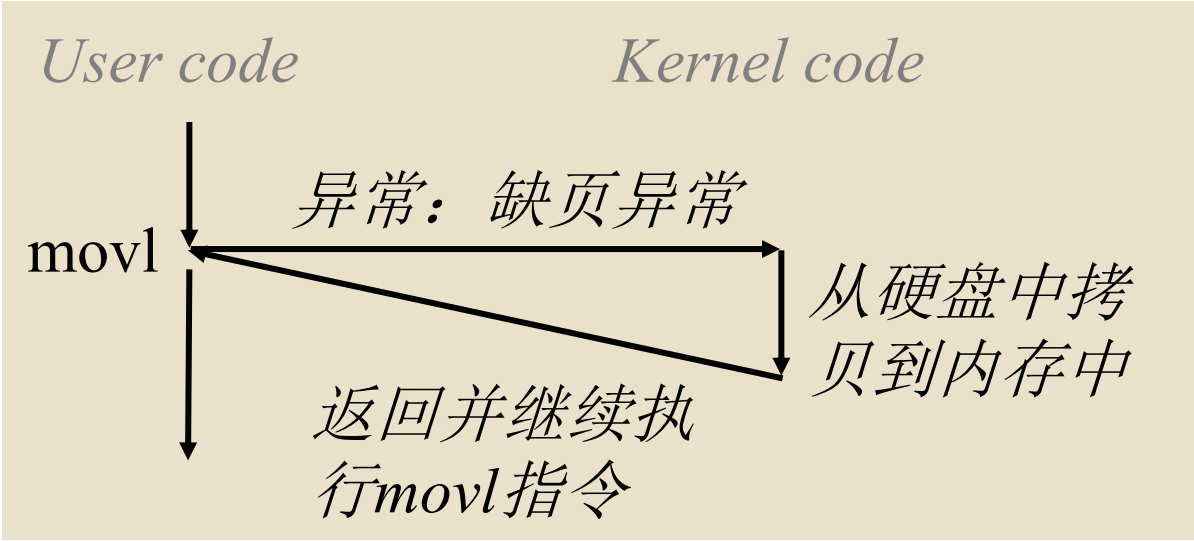
- 故障异常：由执行当前指令的结果的触发
  - 无意的但是可能是可恢复的。
  - 如: 缺页（可恢复），保护故障（protection faults，不可恢复），浮点异常。
  - 重新执行当前指令或者终止。

异常控制流--故障(Fault)举例1:缺页

- 用户写入内存
- 该部分（页) 如今并不在内存中，而在硬盘中，需要调页

```
int a[1000];
main ()
{
     a[500] = 13;
}
```

80483b7: c7 05 10 9d 04 08 0d　　movl　$0xd, 0x8049d10

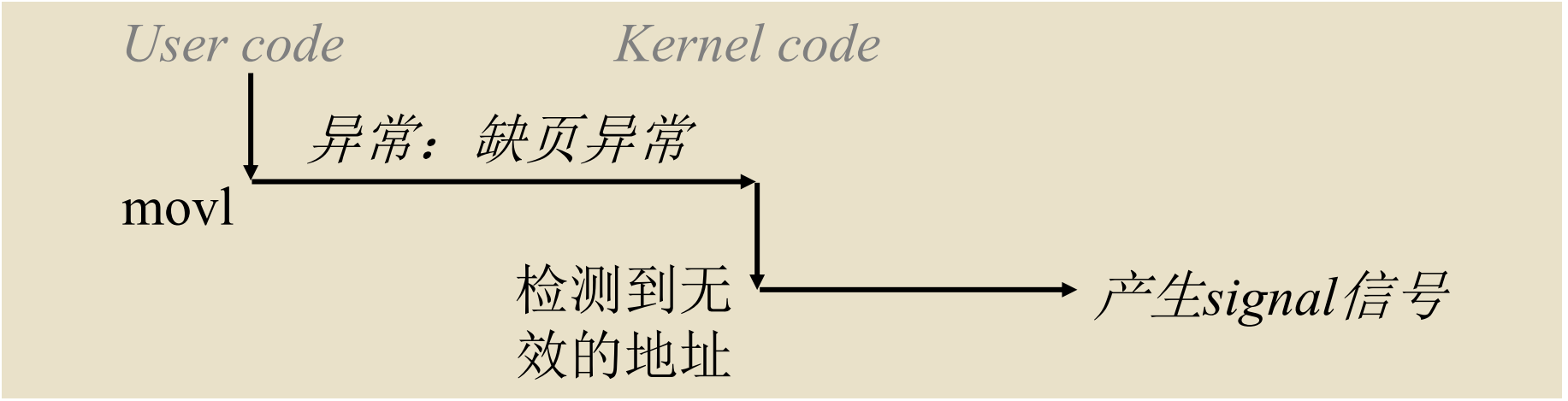


异常控制流--故障（Fault）举例2：无效的内存引用

- 发送 SIGSEGV 信号给用户进程
- 用户进程发生段错误（segmentation fault ）

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

80483b7:　　c7 05 60 e3 04 08 0d　　movl　$0xd, 0x804e360



同步异常控制流--终止异常

- 终止（ Aborts ）
  - 无意的且不可恢复的，如：不合法的指令，奇偶校验错误，机器检查
  - 终止当前程序

主要内容

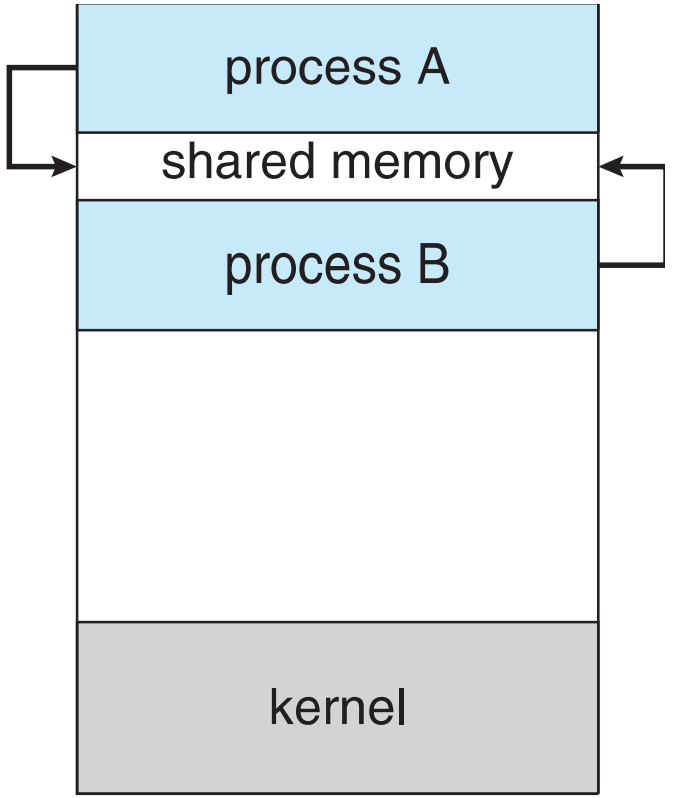
- 2.1 进程的概念和特征
- 2.2 进程的状态、转换和控制
- 2.3 异常控制流
- 2.4 进程间通信

进程间通信

- 进程通信是指进程之间直接以较高的效率传递较多数据的信息交互方式。
- 常用的通信类型
  - 共享内存系统 (Shared-Memory System)
  - 消息传递系统 (Message passing System)
  - 管道(pipe)
  - 套接字
  - 远程过程调用

进程间通信--共享内存系统

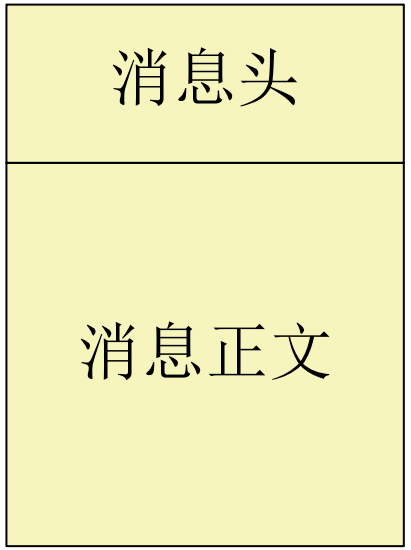
- 通过共享某些数据结构或共享存储器实现进程之间的信息交换。
- 1. 通信过程：　　linux中相应系统调用
  - （1）申请共享存储分区；　　shmget()
  - （2）将共享存储分区映射到本进程地址空间中；　　shmat()
  - （3）进行数据读写；
  - （4）释放共享存储分区；　　shmdt()
  - （5）删除共享存储分区：　　shmctl()



- 2. 共享存储器系统通信方式的特点：
  - （1） 最大的特点是没有中间环节，直接把共享内存映射到不同进程的虚拟地址空间中，进程可直接进行访问，通信直接快速。
  - （2）该通信机制没有提供进程同步机制。

进程间通信—消息传递系统

- 进程间的数据交换是以消息（message，在计算机网络中又称报文）为单位。程序员直接利用系统提供的一组通讯命令（发送原语和接收原语）来实现通信。
  - 在消息通信中，接收方和发送方之间有明确的协议和消息格式 。



消息头：存放消息传输时所需的控制信息。

消息类型：
定长消息
变长消息

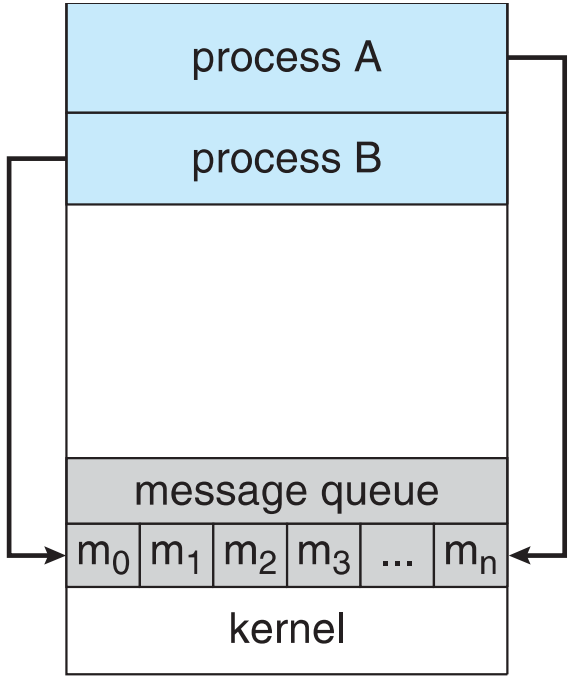
- 消息传递系统实现类型：消息缓冲队列、信箱通信

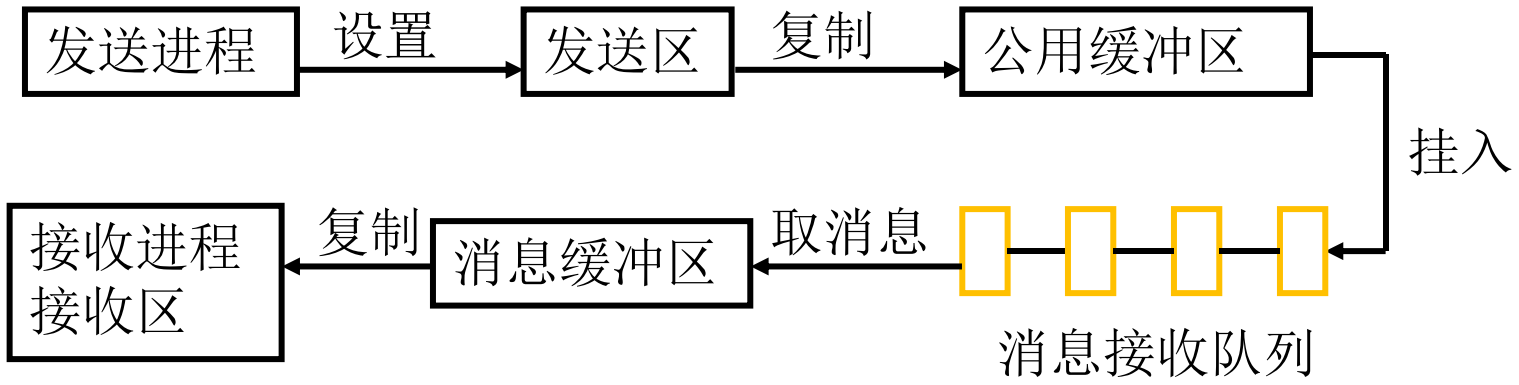
进程间通信—消息传递系统

- 消息传递系统实现类型：消息缓冲队列、信箱通信

  - 消息缓冲队列机制（直接通信）

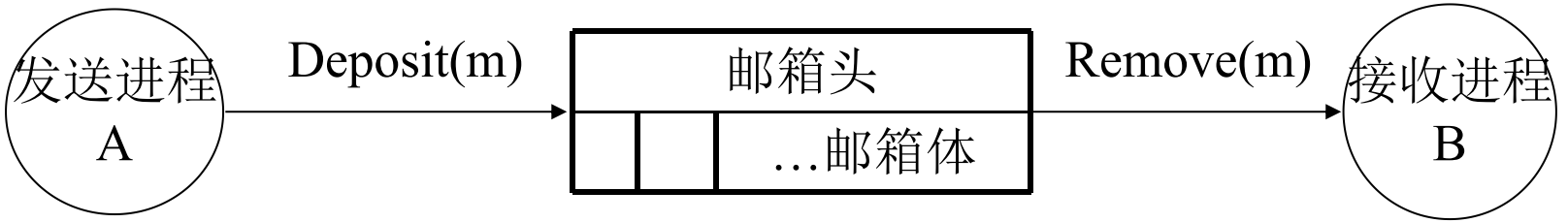
    - ✓发送进程直接将消息发送给接收进程，并将它挂在接收进程的消息缓冲队列上。接收进程从消息缓冲队列中取得消息。





进程间通信—消息传递系统

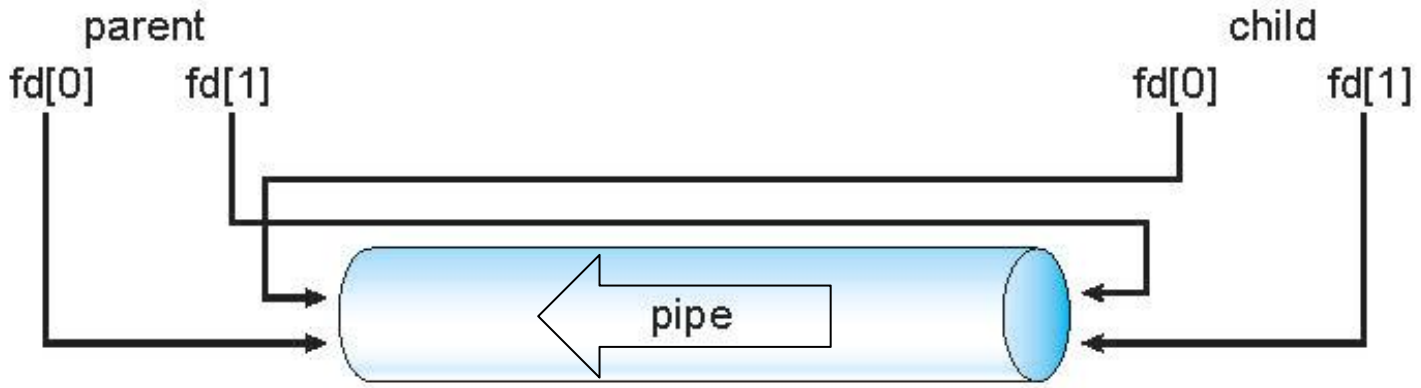
- 消息传递系统实现类型：消息缓冲队列、信箱通信
  - 信箱通信方式（间接通信）
    - ✓发送进程将消息发送到某个中间实体（一般称为信箱）中，接收进程从中取得消息，所以称为信箱通讯方式，相应地系统称为电子邮件系统。
    - ✓信箱通信中，需要定义信箱结构，还包括消息发送和接收功能模块，提供发送原语和接收原语。
      - 邮箱头：邮箱名称、邮箱大小、拥有该邮箱的进程名
      - 邮箱体：存放消息



进程间通信—管道

- 管道用于连接一个读进程和一个写进程，以实现它们之间通信的共享文件（pipe文件，又称为FIFO文件，严格遵循先进先出，不支持文件定位操作）。

- 两种实现机制：
  - 普通管道：int pipe(int fd[2])
    - ✓fd[0]为读出端，fd[1]为写入端。



  - 命名管道：int mkfifo(const char * pathname, mode_t mode)
    - ✓用于任意进程间通信（又称FIFO 通信）

从管道读数据是一次性操作，数据一旦读取，它就从管道中被抛弃，释放空间以便写入更多的数据。

管道只能采用半双工通信，即某一时刻只能单向传输。

要实现双方互动通信，需要定义两个管道。

进程间通信—管道通信应注意的问题

- 为了协调双方的通信，管道通信机制必须提供以下三方面的协调能力：
  - 互斥：一个进程正在对pipe进行读/写操作时，另一进程必须等待。
  - 同步：当写（输入）进程把一定数量的数据写入pipe后，便去睡眠等待，直到读（输出）进程取走数据将其唤醒；当读进程读一空pipe，也应睡眠等待，直至写进程将数据写入管道，才将其唤醒。
  - 对方是否存在：只有确定对方已存在时，才能进行管道通信，否则会造成因对方不存在而无限期等待。

进程同步/通信—不同系统中通信方式

- Unix
  - 管道、消息队列、共享内存、信号量、信号、套接字
- Linux
  - 管道、消息队列、共享内存、信号量、信号、套接字
  - 内核同步机制：原子操作、自旋锁、读写锁、信号量、屏障、BKL
- Windows
  - 同步对象：互斥对象、事件对象、信号量对象、临界区对象、互锁变量
  - 套接字、文件映射、管道命名管道、邮件槽、剪贴板、动态数据交换、对象连接与嵌入、动态链接库、远程过程调用

Summary: 进程表示

- 进程的状态： 新建态、就绪态、运行态、阻塞态、终止态
- 进程控制块对进程管理和控制的作用
- 进程控制块PCB的内容
- 进程间的几种通信方式

例题

- 1.进程是（）

A.与程序等效的概念　　B.并发环境中程序的执行过程

C.一个系统软件　　D.存放在内存中的程序

答案:B

例题

- 2.进程与程序的本质区别是（）

A.内存和外存

B.动态和静态特征

C.共享和独占使用计算机资源

D.顺序和非顺序执行机器指令

答案:B

例题

- 3.进程的三种基本状态不包括（）

A.就绪　　B.执行

C.后备　　D.阻塞

答案:C

解析：

就绪状态：当进程已分配到除CPU以外的所有必要的资源，只要获得处理机便可立即执行，这时的进程状态称为就绪状态。

执行状态：当进程已获得处理机，其程序正在处理机上执行，此时的进程状态称为执行状态。

阻塞状态：正在执行的进程，由于等待某个事件发生而无法执行时，便放弃处理机而处于阻塞状态。例如，等待I/O完成、申请缓冲区不能满足、等待信件(信号)等。

例题

- 4.在进程状态转换时，下面哪一种状态转换时不可能发生的（）

A.就绪态-执行态　　B.执行态-就绪态

C.执行态-阻塞态　　D.阻塞态-执行态

答案:D

解析：

就绪态-执行态：进程的调度

执行态-就绪态：分配的时间片用完

运行态-阻塞态：等待某个事件发生而睡眠，例如，等待I/O完成、申请缓冲区不能满足、等待信件(信号)等。

阻塞态-就绪态：阻塞态因等待事件发生而被唤醒，但不能直接进入执行态，应进入就绪态等待cpu分配时间片。

例题

- 5.某进程在运行过程中需要等待从磁盘上读入数据，此时该进程的状态将（）

A.从就绪态变为运行态　　B.从运行态变为就绪态

C.从运行态变为阻塞态　　D.从阻塞态变为就绪态

答案:C

例题

■ 6. 计算机系统中两个协作进程之间不能用来进行进程间通信的是（）

A. 数据库　　B. 共享内存　　C. 消息传递机制　　D. 管道

答案:A

例题

■ 7. 下列关于管道pipe 通信的叙述中，正确的是（）

A. 一个管道可实现双向数据传输　　B. 管道的容量仅受磁盘容量大小的限制

C. 进程对管道进行读操作和写操作都有可能被阻塞

D. 一个管道只能有一个读进程或一个写进程对其操作

答案:C

➢解析：

A. 管道是单向的（半双工）

B. 实际上，管道是一个固定大小的缓冲区。在linux中，缓冲区大小为4KB。

C. 管道空，read阻塞；管道满，write阻塞

D. 可多个读写（互斥）

例题：进程图


```
void fork4()
{
    printf("L0\n");
    if (fork()== 0) {
        printf("L1\n");
        if (fork()== 0) {
            printf("L2\n");
      }
    }
    printf("Bye\n");
}
```

Hope you enjoyed the OS course!