第3章 进程

孙承杰

E-mail： sunchengjie@hit.edu.cn

哈工大计算学部人工智能教研室

2023年秋季学期

第3章 进程

- 主要内容
  - 3.1 进程的引入
  - 3.2 进程的概念
  - 3.3 进程的描述/表达

    描述进程的状态变迁、数据结构、进程控制块
  - 3.4 进程操作

    进程创建、进程阻塞、进程唤醒、进程终止
  - 3.5 进程调度

    调度队列、调度程序、上下文切换
  - 3.6 进程间通信

    远程过程调用、基于消息传递、基于共享内存、消息队列
  - 3.7 进程的可执行映像

3.1 进程的引入

为什么要引入进程?

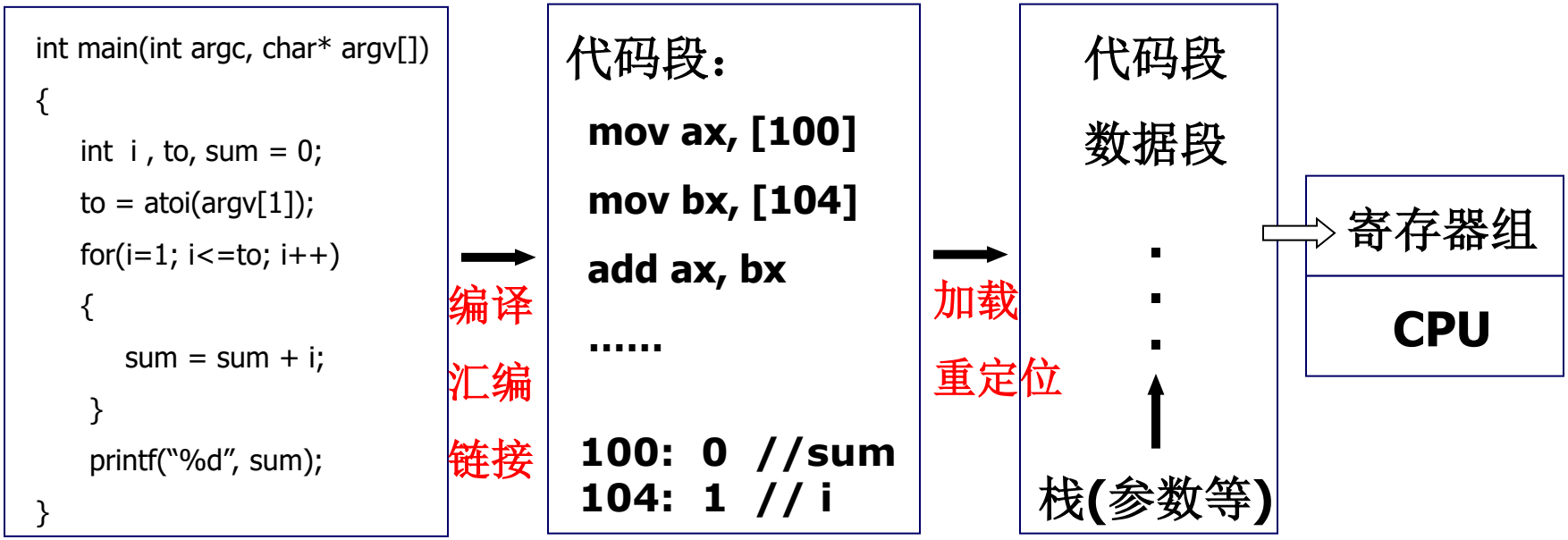
回顾OS发展

(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 (5)对称多处理	← 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

进程概念 – OS发展的需要和必然

3.1 进程的引入

问题1:程序是如何运行起来的？



源代码

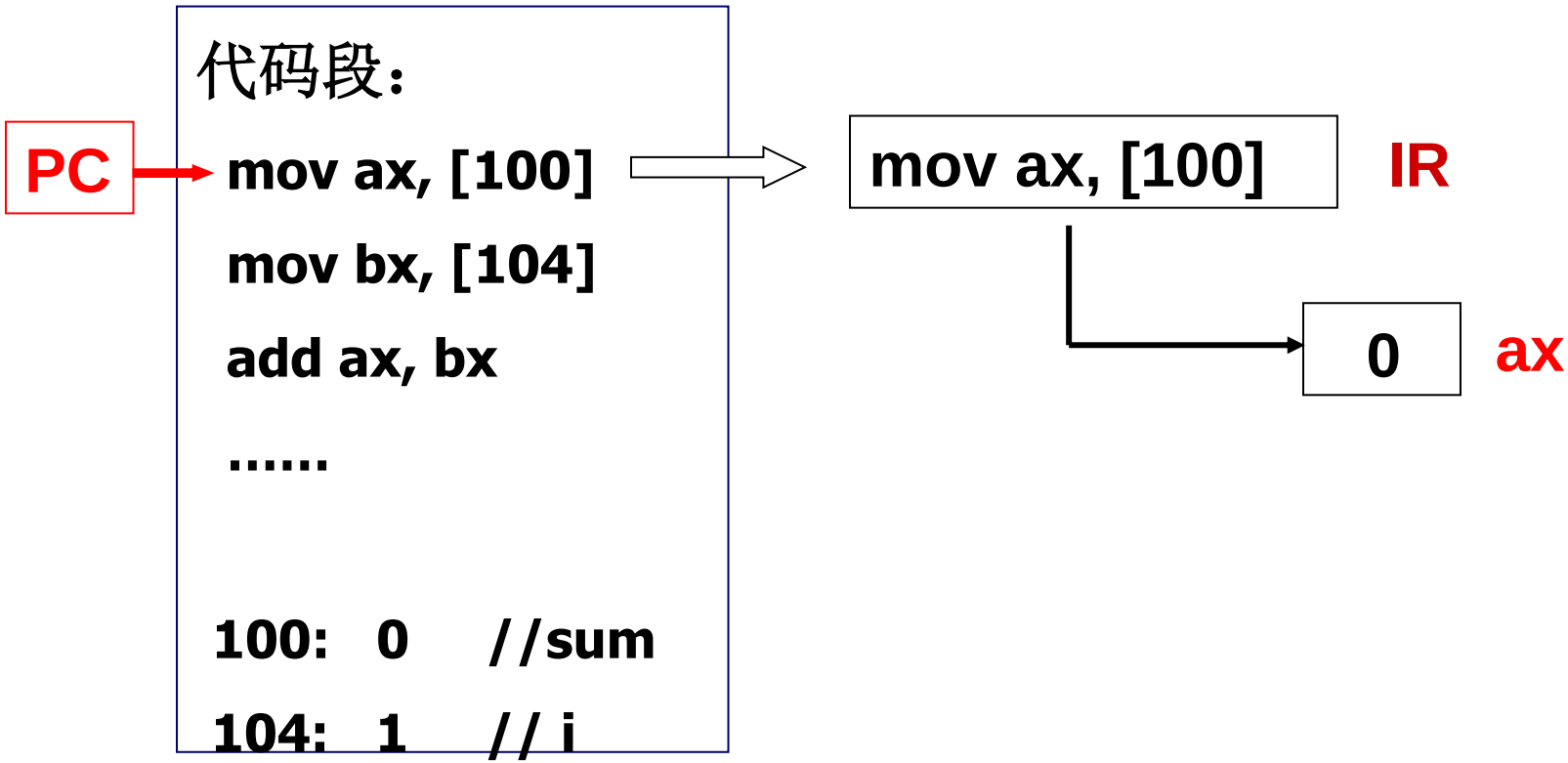
可执行程序（文件）
(在磁盘上，静态存放)

程序在内存中运行

思考：可执行文件的格式与内存布局的关系？

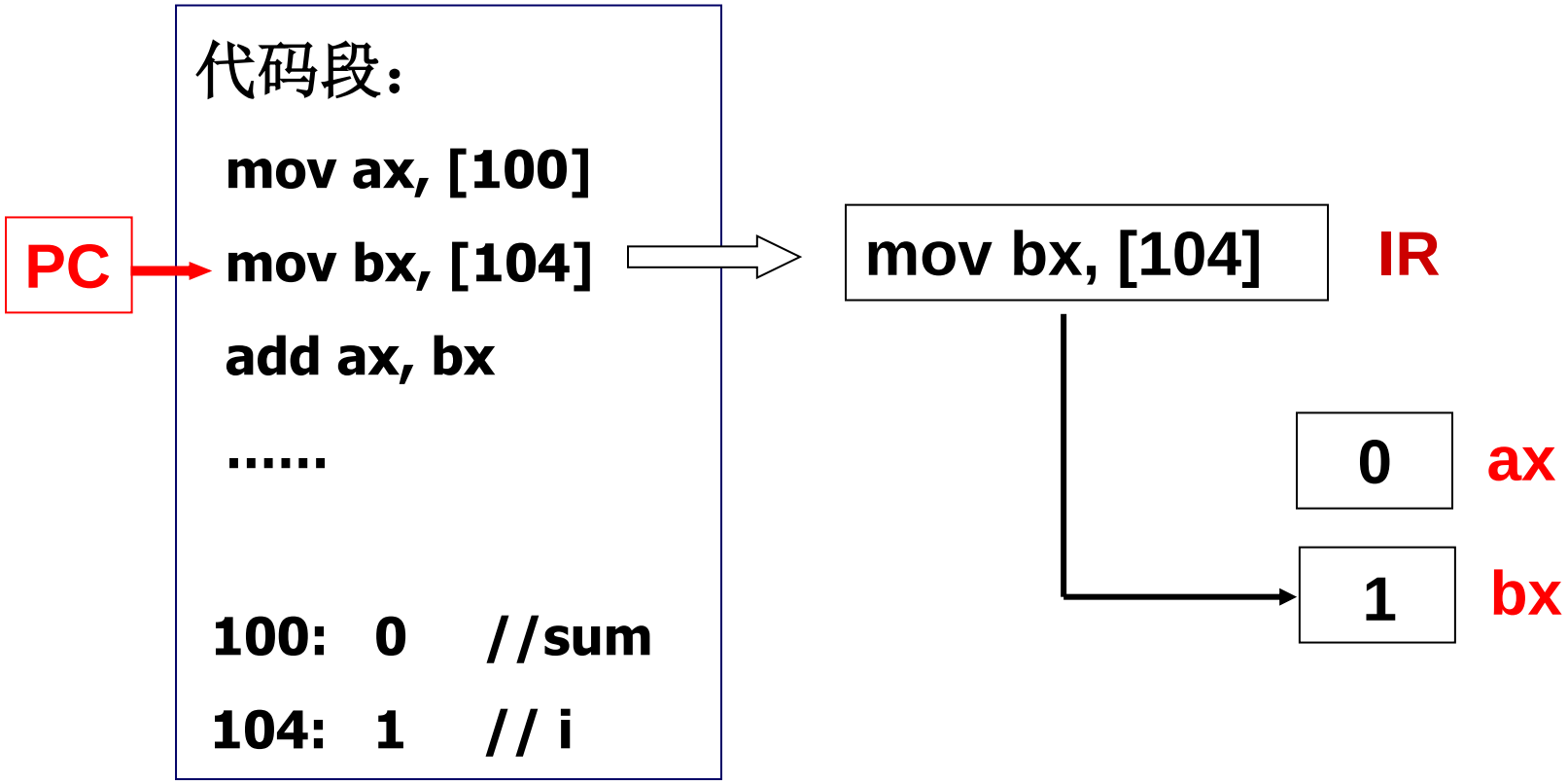
3.1 进程的引入

程序执行的细节



3.1 进程的引入

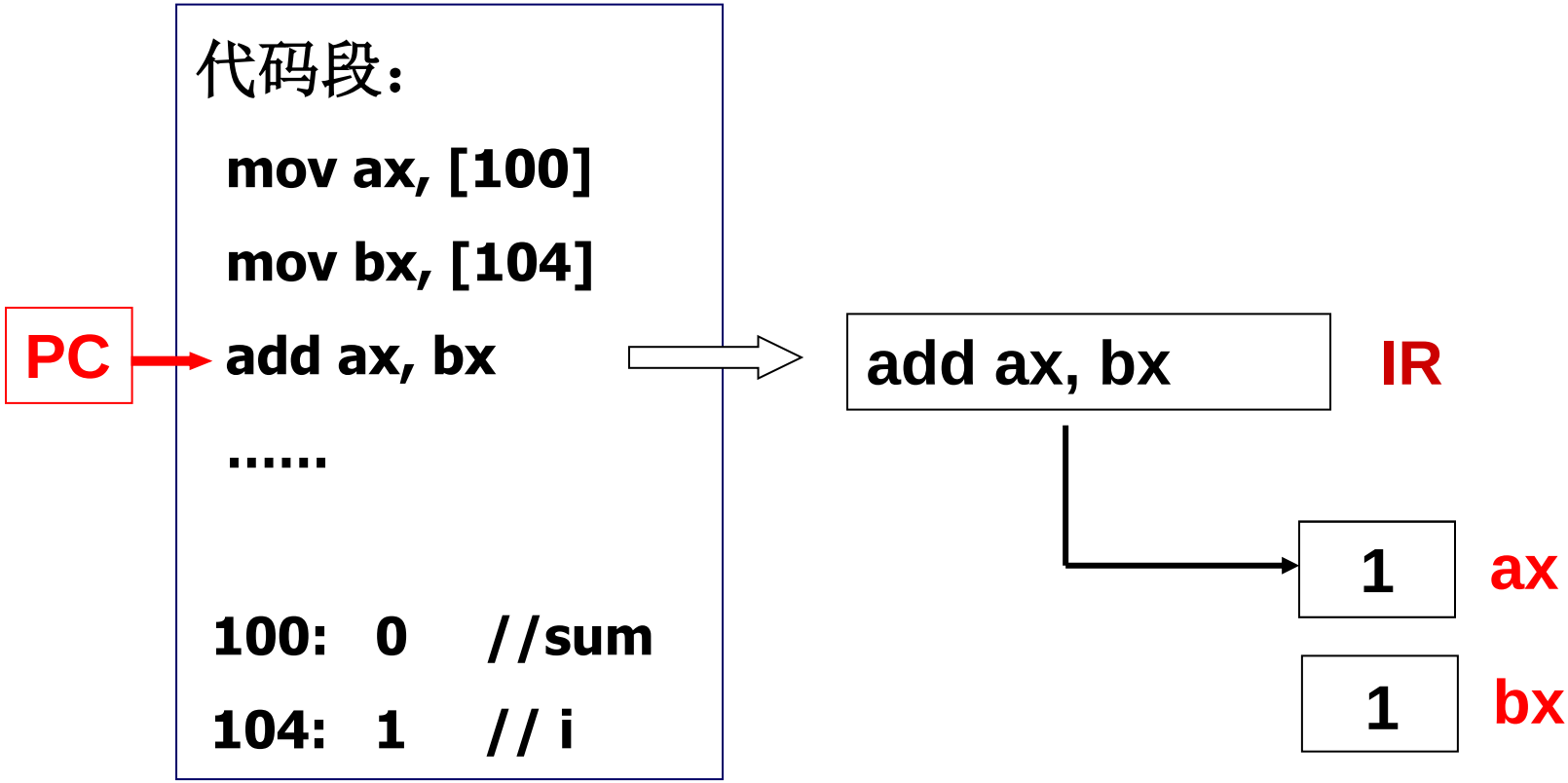
程序执行的细节（1）



3.1 进程的引入

程序执行的细节(2)

- 程序执行的关键在于PC的变化!



3.1 进程的引入

问题2:计算机中只有一个程序在执行吗?

用户的需求:

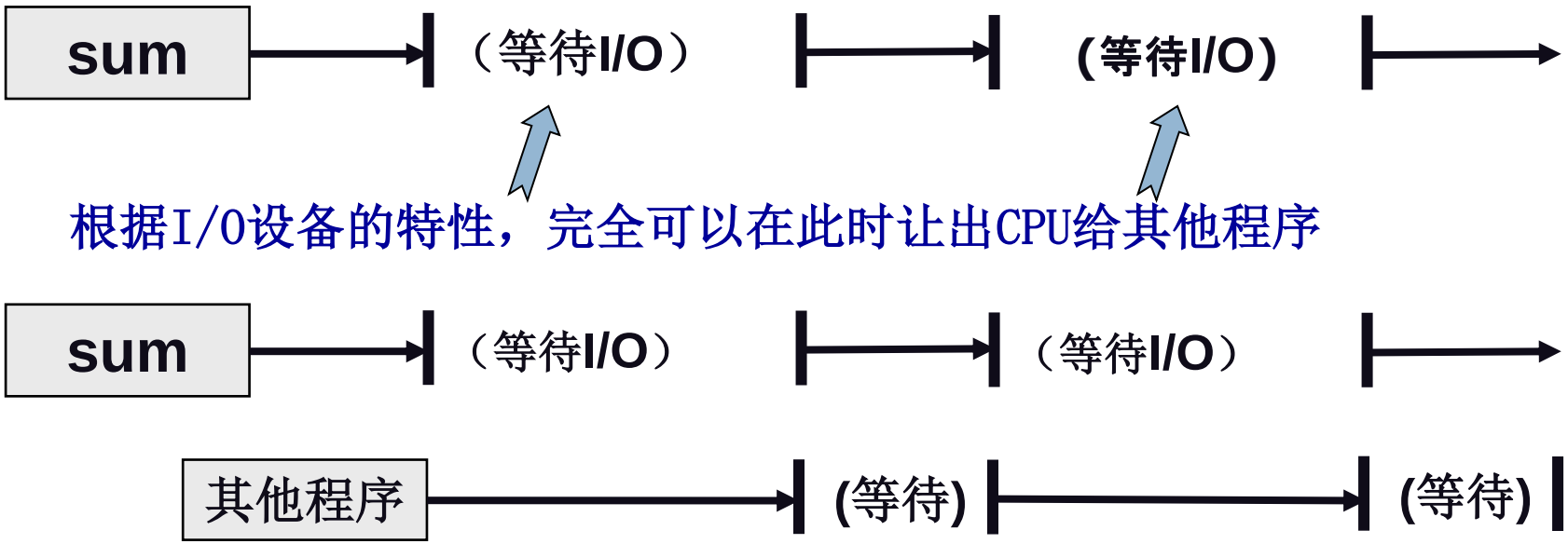
- 多个用户共享一台计算机
- 单个用户利用计算机同时做多个事情

使得计算机必须能够同时执行多个程序!

3.1 进程的引入

问题3:计算机系统的基本特征?

- CPU的速度相比其他设备要快的多!每个应用不停访问IO与其他进程交互
- 中断技术、DMA技术可以使I/O等设备工作过程几乎脱离CPU的干预!

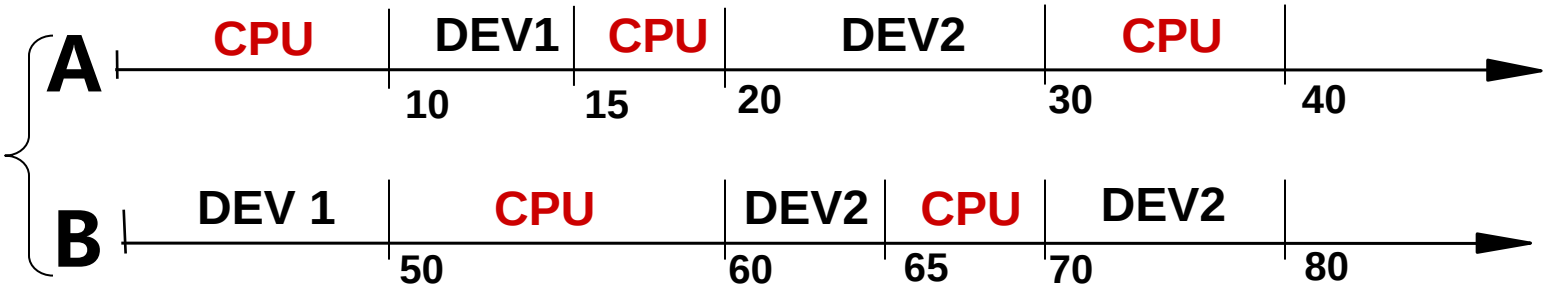


计算机可以同时在执行多个程序(交替执行)!

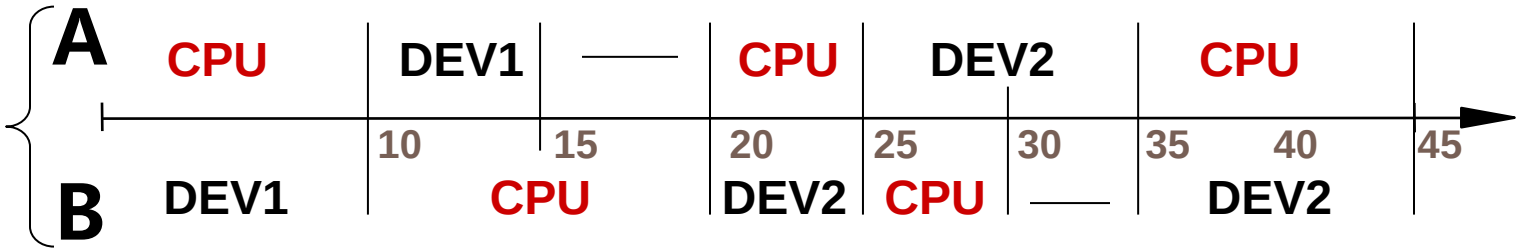
3.1 进程的引入

问题4:一个程序vs.多个程序，效率？

一次执行一个程序：



多个程序并发执行：



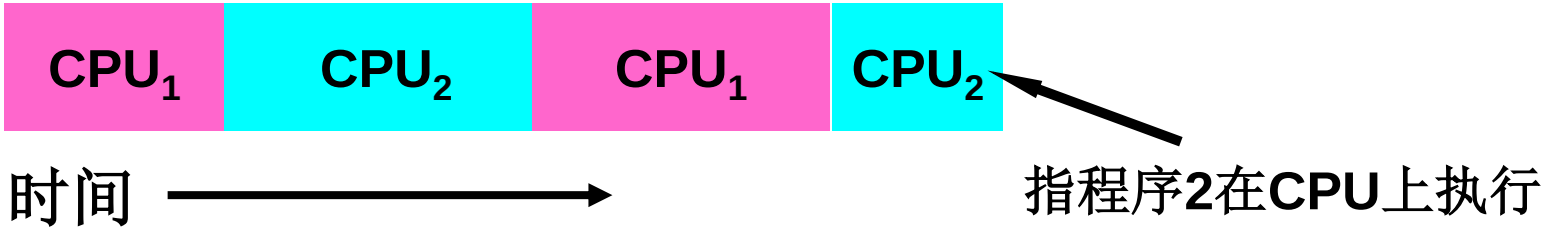
	一次一个程序	多个程序
CPU利用率	40/80=50%	40/45=89%
DEV1利用率	15/80=18.75%	15/45=33%
DEV2利用率	25/80=31.25%	25/45=56%

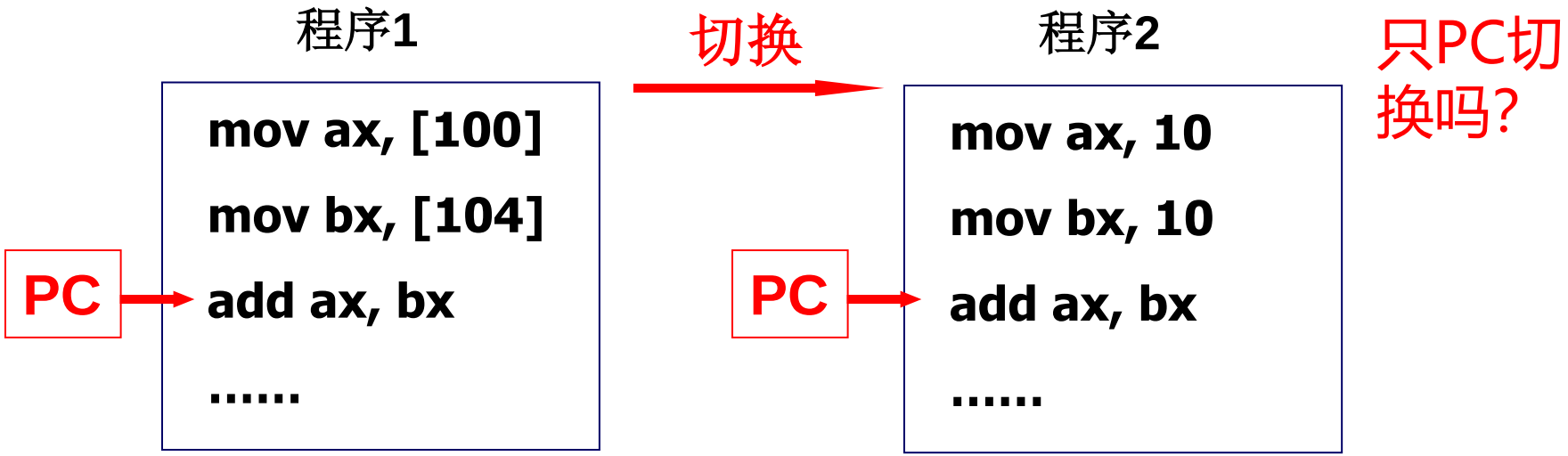
3.1 进程的引入

问题5:并发过程程序如何切换？

并发：用一个CPU交替地“同时”执行多个程序

- 并发是指同时出发，交替执行(和并行不同)

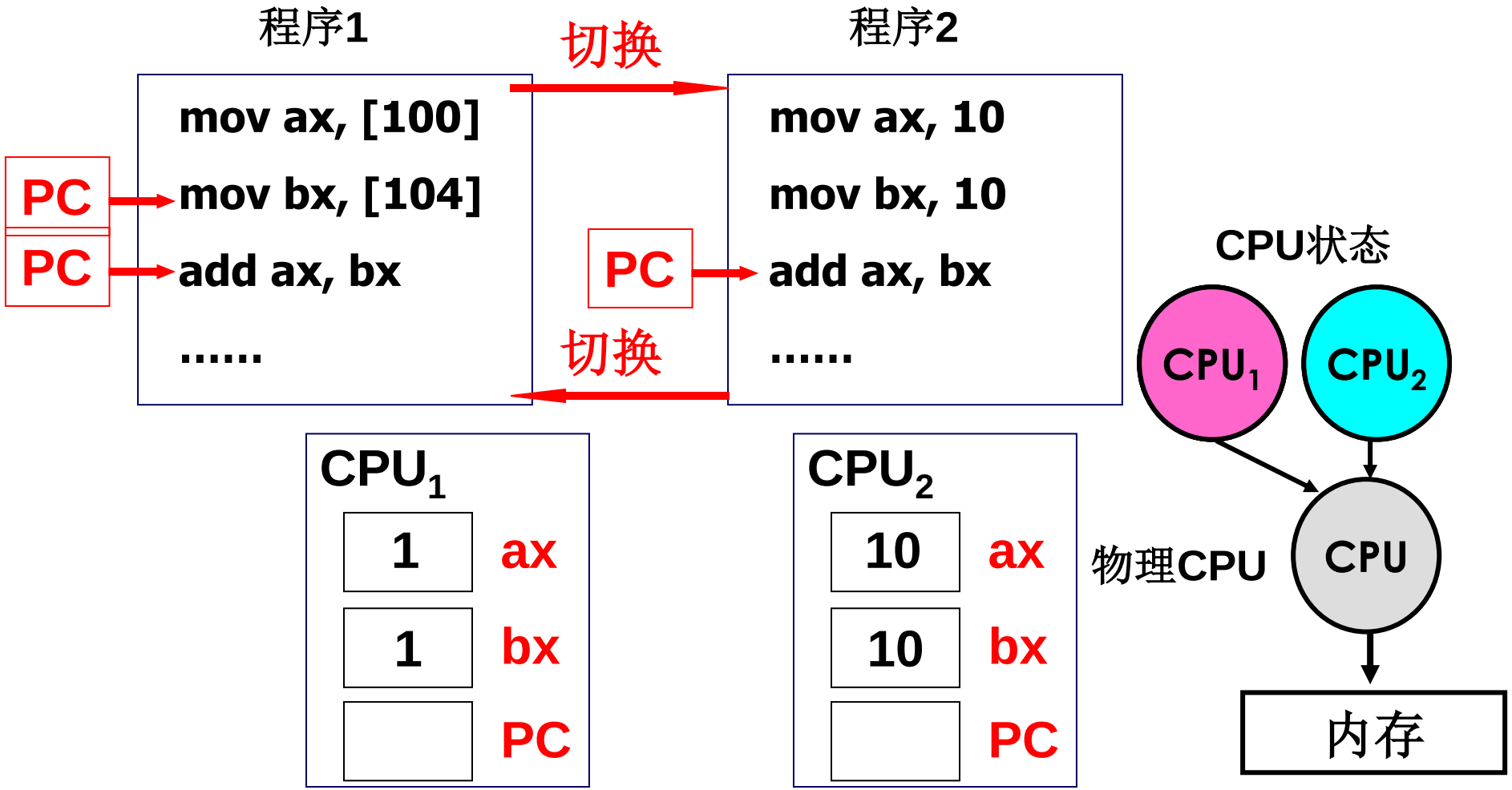




3.1 进程的引入

问题5:并发过程程序如何切换？

- 先保存执行现场，后切换执行现场!



3.1 进程的引入

为什么要引入进程?

总结前面的内容，可以看到：

- 多道程序并发执行是提高系统资源利用率(特别是CPU)的有效途径
- 多道程序竞争CPU进行轮换执行是提高多用户响应速度的有效方法
- 计算机的客观实际支持多道程序并发的实现
- 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！

3.2 进程的概念

- 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*）

计算机中正在运行的程序的一个实例（*Instance*）

可以分配给处理器并由处理器执行的一个实体（*Entity*）

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（*Unit*）

进程概念让许多事情豁然开朗!

用Process Explorer查看进程的表述映像、硬件、状态和上下文

3.2 进程的概念

- 日常生活中“进程”例子：
  - 装修工程： 装修队同时承接N个家庭房屋装修工程
  - 下棋过程： 一对N的“车轮大战”

每个装修工程 -- 一个进程

装修队 -- CPU

每盘对弈棋局 -- 一个进程

聂卫平 -- CPU

3.2 进程的概念

进程的5个基本特征：

（1）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）

（2）并发性　引入进程的目的就是为了使多个程序并发执行，以提高资源利用率（主要CPU）

（3）独立性　进程是一个程序运行的基本单位，也是系统进行资源分配和调度的基本单位

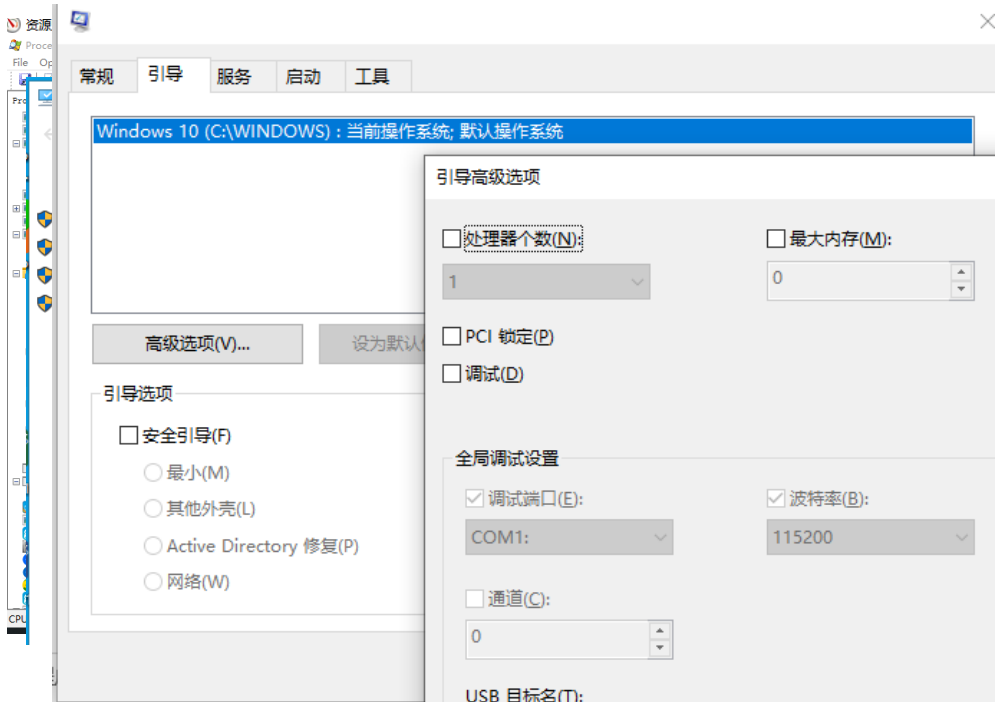
（4）异步性　进程以各自独立的、不可预知的速度向前推进

（5）交往性（交互）进程在运行过程中可能会与其它进程发生直接或间接的相互作用

3.2 进程的概念

认识进程相关项：任务管理、资源监视器、进程浏览器、系统配置（config或msconfig）、计算机属性

- 与进程相关的管理有哪些：磁盘、文件、内存、IO、进程交互、缓存
- 基本信息/详细信息
- 打开/最小化一个ppt，在资源监视器中观查磁盘使用情况。观察工作集、IO和虚拟内存
- CPU分配与调度
- 进程树
- 线程与处理器调度(idle)
- 线程栈
- 程序加载器
- 磁盘与交换分区虚拟内存



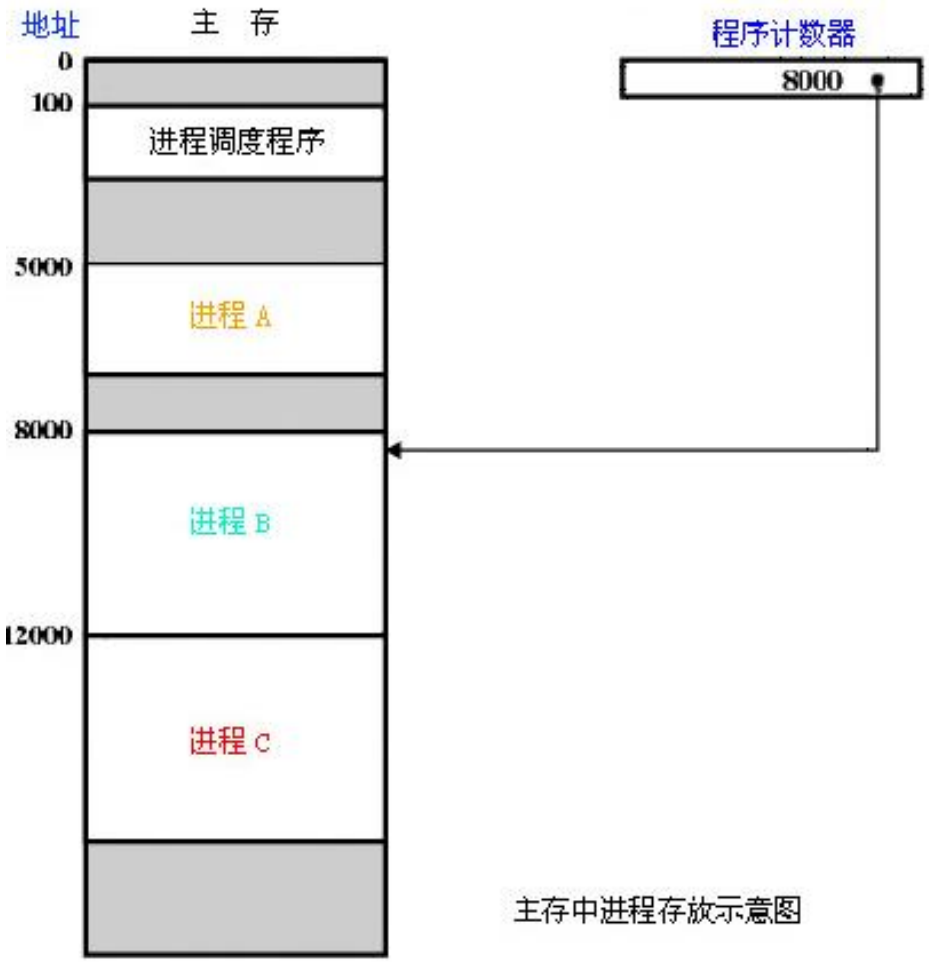
3.3 进程的描述与表达

如何描述/表达进程？

- 进程状态及其变迁
- 进程的数据结构
- 进程控制块

3.3 进程的描述与表达

3.3.1 进程状态及其变迁



主存中进程存放示意图

3.3 进程的描述与表达

3.3.1 进程状态及其变迁

进程轨迹 ——
进程执行的指令序列

(a)进程A轨迹	(b)进程B轨迹	(c)进程C轨迹
5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004	8004	12004
5005	8005	12005
5006	8006	12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

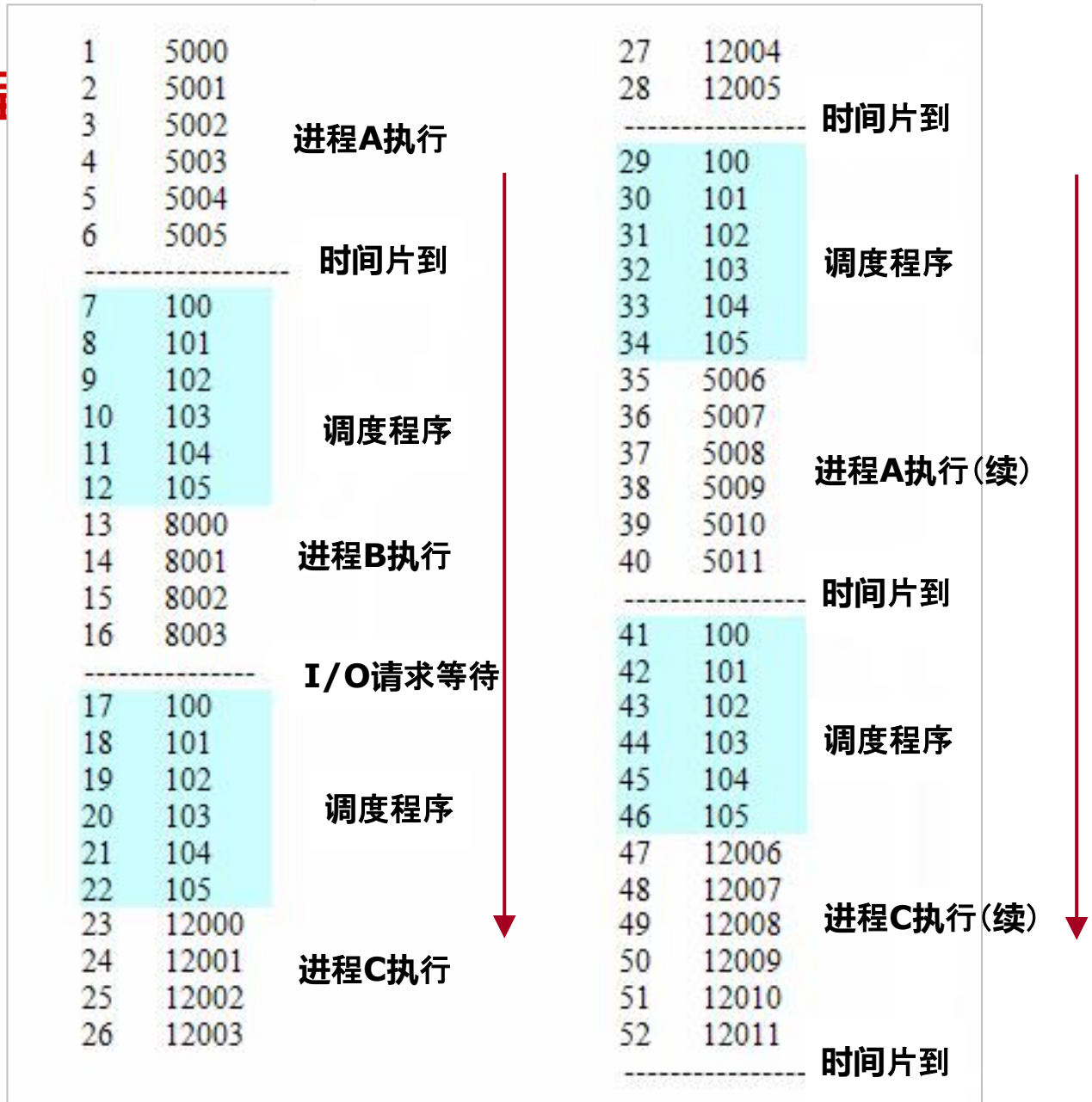
5000 = 进程A的程序起始地址
8000 = 进程B的程序起始地址
12000 = 进程C的程序起始地址

3.3 进程的描述与表达

3.3.1 进程

进程ABC组合轨迹

-- CPU调度轨迹



3.3 进程的描述与表达

3.3.1 进程状态及其变迁

- 进程执行时的间断性，决定了进程可能具有多种状态
- 运行中的进程至少具有以下三种基本状态

（1）就绪状态 – 在某时刻，进程已获得除处理机以外的所有资源，一旦分到了处理机就可以立即执行

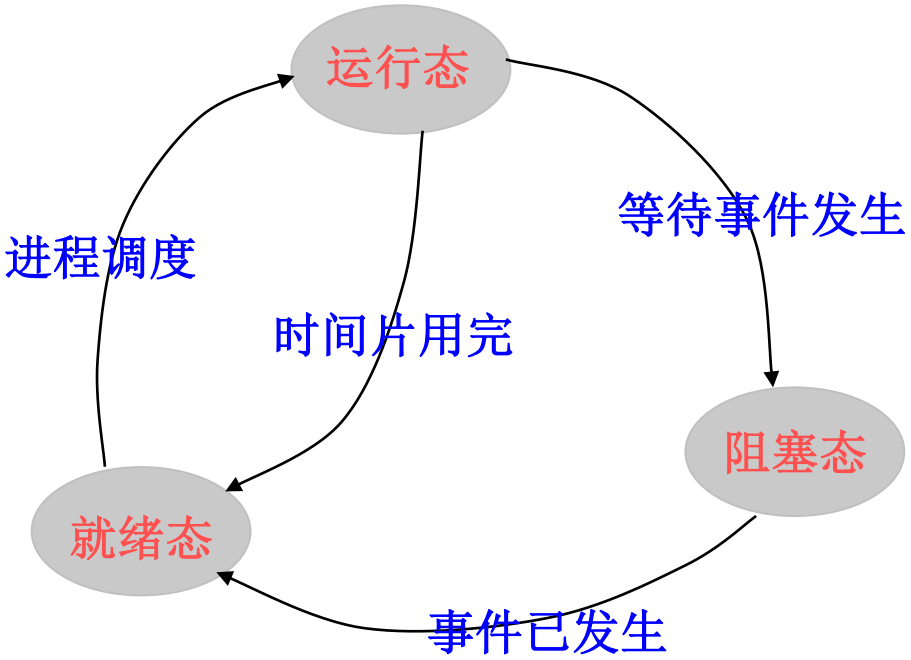
（2）运行状态 – 进程已经获得必要资源，并占有处理机运行

（3）阻塞状态 – 正在执行的进程，由于发生某事件而暂时无法执行下去

- 进程是有生命周期的，它的状态随着自身的推进和外界条件的变化而发生变化
- 可以用一个进程状态变化图来说明系统中每个进程可能具备的状态，以及这些状态发生变化的可能原因

3.3 进程的描述与表达

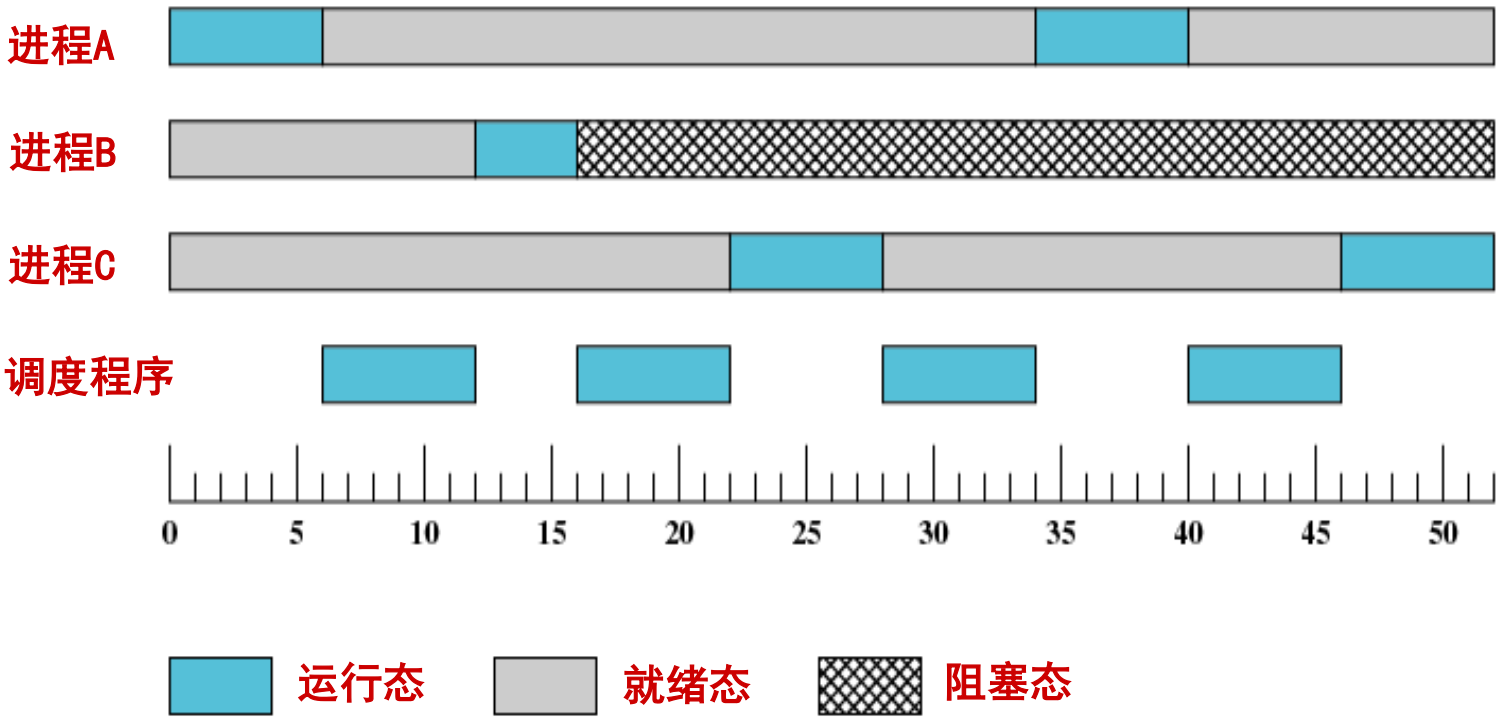
进程三状态变迁图



进程状态变化图（三状态）

3.3 进程的描述与表达

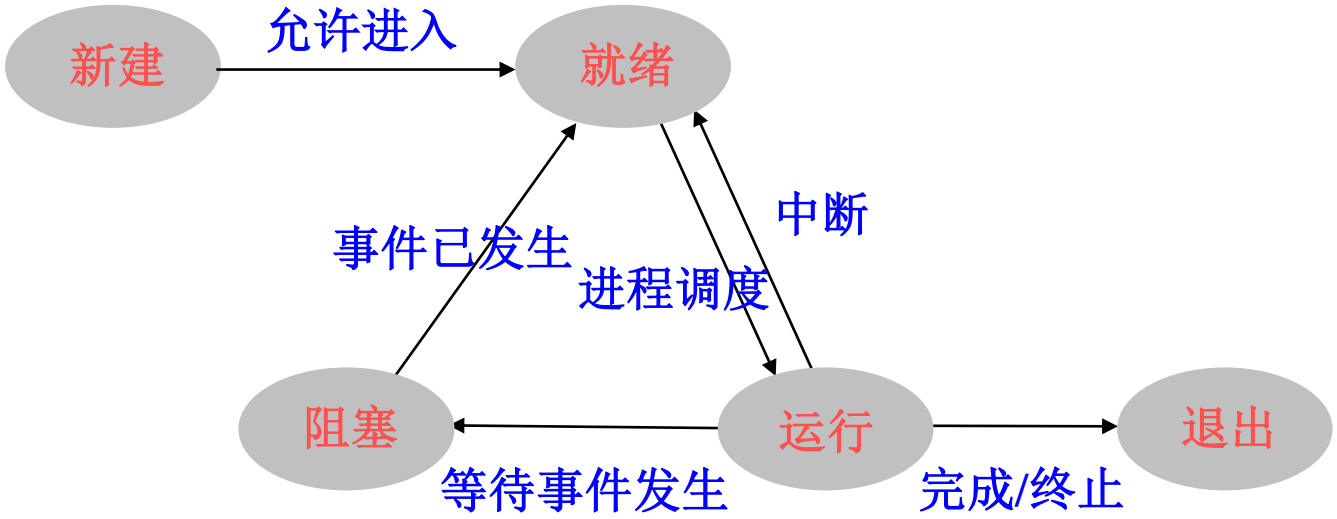
进程三状态变迁图



进程状态变化序列图举例（三状态）

3.3 进程的描述与表达

进程五状态变迁图



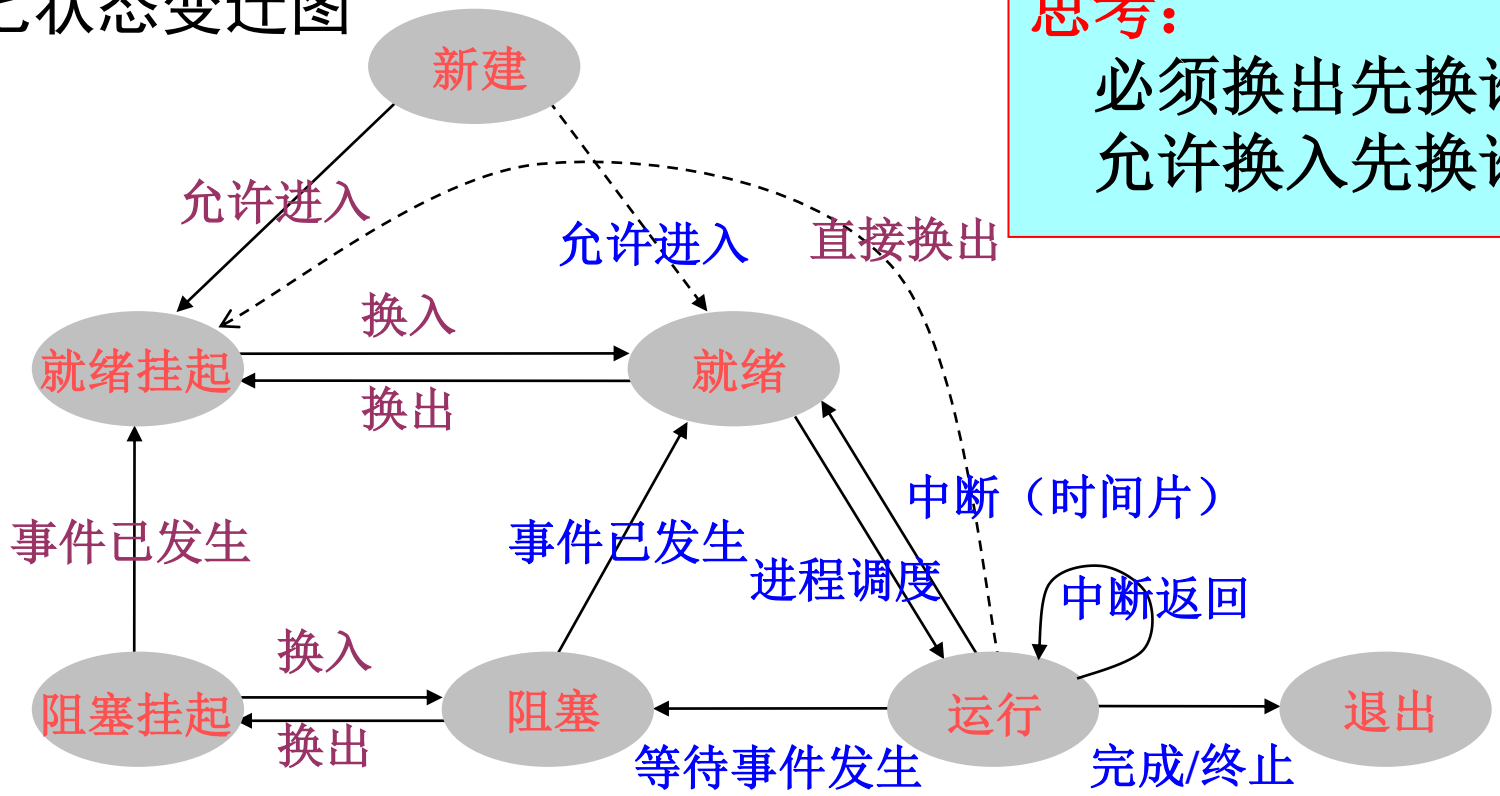
进程状态变化图（五状态）

新建状态 – 至少建立PCB，但进程相关的其他内容可能未调入主存

退出状态 – 进程已经终止，但资源等待父进程或系统回收

3.3 进程的描述与表达

进程七状态变迁图



进程状态变化图（七状态）

思考：
必须换出先换谁？
允许换入先换谁？

挂起状态 – 引入主存←→外存的交换机制，虚拟存储管理的基础

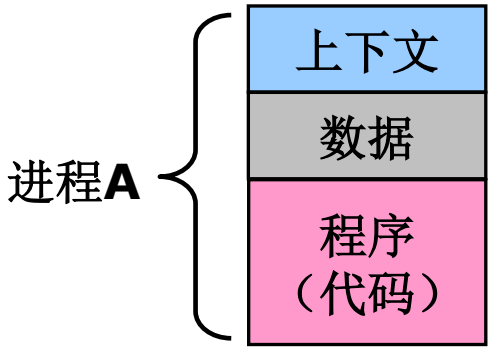
3.3 进程的描述与表达

3.3.2 进程的数据结构

对单个进程，操作系统建立“进程映像”进行描述和表达

必须包含3部分内容：

- 一段可执行的程序（代码）
- 程序所需的相关数据（变量、栈、堆、缓冲区等）
- 记录程序执行的上下文环境或运行时信息（**PCB**）



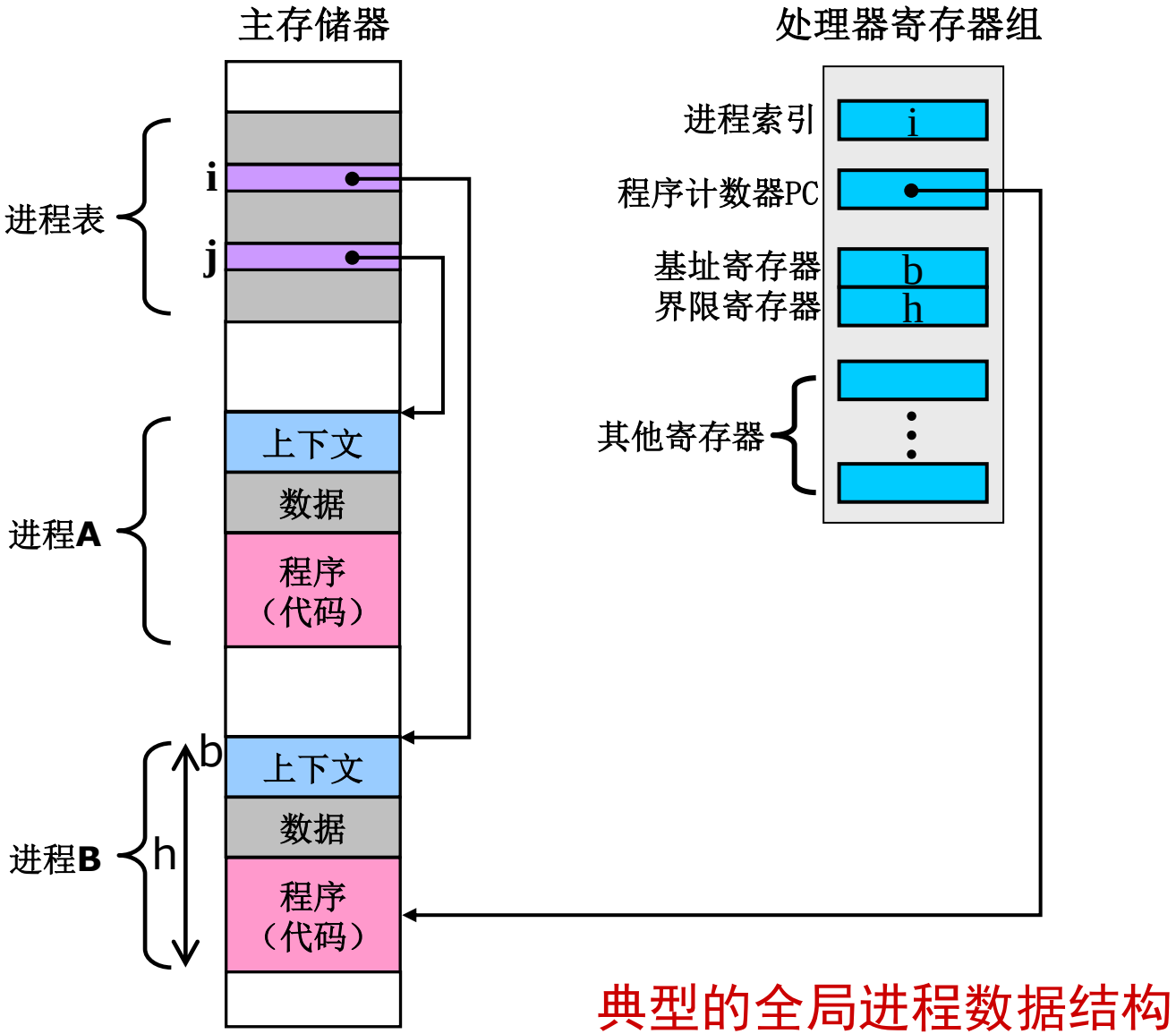
对所有进程的管理，操作系统建立特定数据结构：

进程索引表 - 进程目录

进程映像存储区

进程链表队列

3.3 进程的描述与表达



典型的全局进程数据结构

3.3 进程的描述与表达

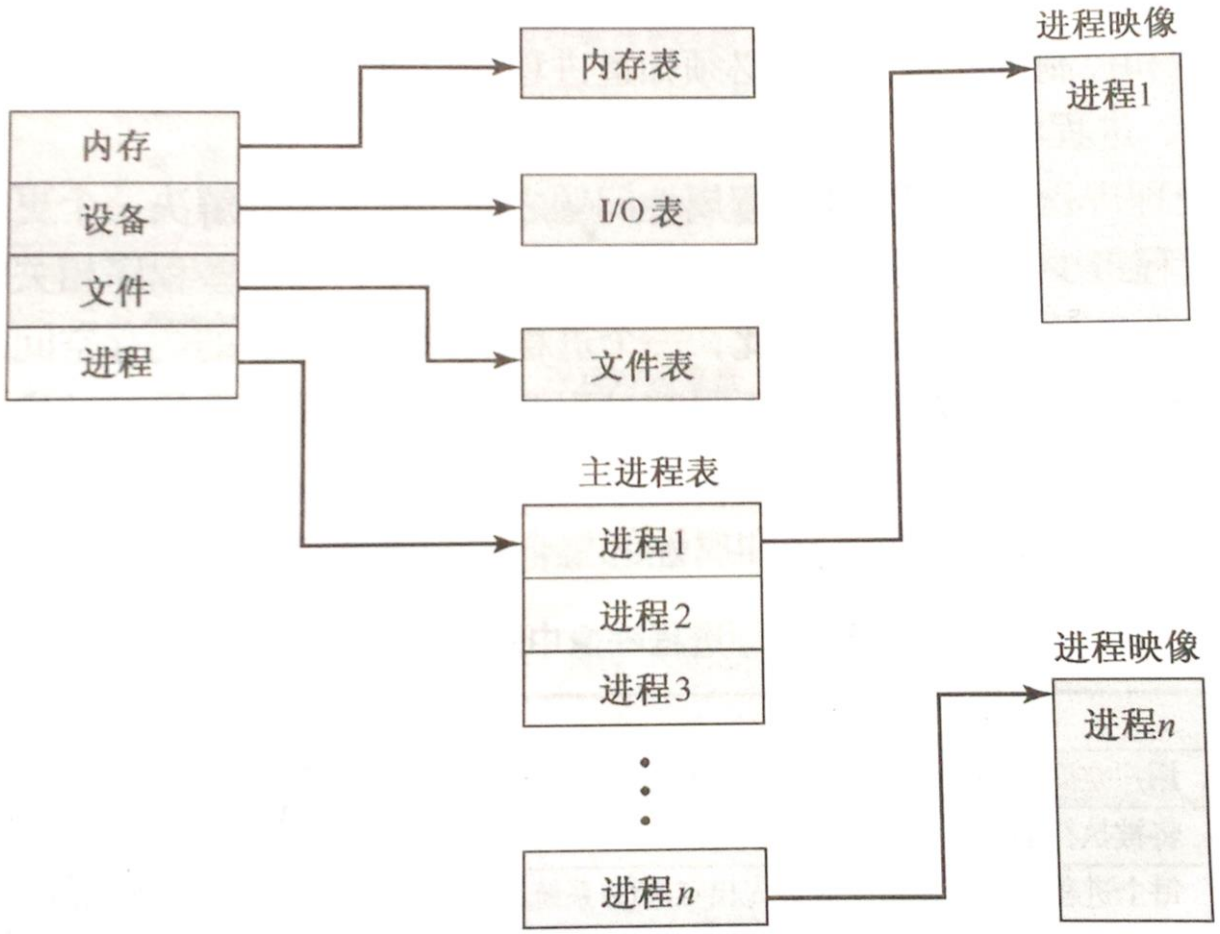


图 3.11　操作系统控制表的通用结构

3.3 进程的描述与表达

3.3.3 进程控制块PCB

PCB – Process Control Block，是一个线性的数据结构
用来描述和记录进程的动态变化信息，是进程的灵魂

进程控制块PCB：

- 记录程序执行的上下文环境
- 为了描述和控制进程的运行，系统为每个进程定义了一个数据结构 — 进程控制块（PCB）
- 系统创建一个进程，就是由系统为某个程序设置一个PCB；进程执行完成时，由系统收回其PCB，该进程便消亡了
- 系统将根据PCB而感知进程的存在，故PCB是进程存在的唯一标志
- 通过PCB可以访问到进程的所有信息

3.3 进程的描述与表达

进程控制块PCB的内容：

不同操作系统中，PCB的内容不尽相同，但通常包括：

(1) 进程标识符 - 确认进程的唯一标识(PID)

(2) 进程当前状态 - 进程调度程序分配处理机（处理器）的依据

(3) 进程队列/栈指针 - 记录PCB链表中上/下一个PCB的地址；进程或线程的栈地址管理指针

(4) 程序地址范围 - 开始地址与结束地址（地址空间）

(5) 进程优先级 - 反映进程要求CPU的紧迫程度

(6) CPU现场保护区 - 记录让出处理机时的CPU现场信息（寄存器组）

(7) 通信信息 - 记录与其他的进程的信息交换情况

(8) 家族联系 - 记录父进程的PID

(9) 占有资源清单 - 打开文件列表、所需资源及已分配资源清单

3.3 进程的描述与表达

进程控制块中的典型元素（1/3）

Process Identification	
Identifiers	• Identifier of this process. • Identifier of the process that created this process (parent process). • User identifier.
Processor State Information	
User-Visible Registers	A user-visible register is one that may be referenced by means of the machine language that the processor executes while in user mode. Typically, there are from 8 to 32 of these registers, although some RISC implementations have over 100.
Control and Status Registers	• **Program counter:** Contains the address of the next instruction to be fetched. • **Condition codes:** Result of the most recent arithmetic or logical operation (e.g., sign, zero, carry, equal,overflow). • **Status information:** Includes interrupt enabled/disabled flags, execution mode
Stack Pointers	Each process has one or more last-in-first-out (LIFO) system stacks associated with it. A stack is used to store parameters and calling addresses for procedure and system calls. The stack pointer points to the top of the stack.

3.3 进程的描述与表达

- 进程控制块中的典型元素（2/3）

Process Control Information	
Scheduling and State Information	• **Process state:** Defines the readiness of the process to be scheduled for execution (e.g., running, ready, waiting, halted). • **Priority:** One or more fields may be used to describe the scheduling priority of the process. In some systems, several values are required (e.g., default, current, highest allowable). • **Scheduling-related information:** This will depend on the scheduling algorithm used. Examples are the amount of time that the process has been waiting and the amount of time that the process executed the last time it was running. • **Event:** Identity of event the process is awaiting before it can be resumed.

3.3 进程的描述与表达

进程控制块中的典型元素（3/3）

Process Control Information	
Data Structuring	A process may be linked to other process in a queue, ring, or some other structure. For example, all processes in a waiting state for a particular priority level may be linked in a queue. A process may exhibit a parent–child (creator–created) relationship with another process. The process control block may contain pointers to other processes to support these structures.
Interprocess Communication	Various flags, signals, and messages may be associated with communication between two independent processes. Some or all of this information may be maintained in the process control block.
Process Privileges	Processes are granted privileges in terms of the memory that may be accessed and the types of instructions that may be executed. In addition, privileges may apply to the use of system utilities and services
Memory Management	This section may include pointers to segment and/or page tables that describe the virtual memory assigned to this process.
Resource Ownership and Utilization	Resources controlled by the process may be indicated, such as opened files. A history of utilization of the processor or other resources may also be included; this information may be needed by the scheduler.

3.3 进程的描述与表达
UCOS任务控制块结构的主要成员

```
OS_STK        *OSTCBStkPtr; /*当前任务栈顶的指针*/
struct os_tcb *OSTCBNext;   /*任务控制块的双重链接指针*/
struct os_tcb *OSTCBPrev;   /*任务控制块的双重链接指针*/
OS_EVENT      *OSTCBEventPtr; /*事件控制块的指针*/
void          *OSTCBMsg;    /*消息的指针*/
INT16U         OSTCBDly;    /*任务延时*/
INT8U          OSTCBStat;   /*任务的状态字*/
INT8U          OSTCBPrio;   /*任务优先级*/
INT8U          OSTCBX;      /*用于加速进入就绪态的过程*/
INT8U          OSTCBY;
INT8U          OSTCBBitX;
INT8U          OSTCBBitY;
```

3.3 进程的描述与表达
linux0.11任务控制块结构的主要成员

```
struct task_struct {
    long state;                 //任务的运行状态（-1 不可运行，0 可运行(就绪)，>0 已停止）。
    long counter;               // 任务运行时间计数(递减)（滴答数），运行时间片。
    long priority;              // 运行优先数。任务开始运行时 counter=priority，越大运行越长。
    long signal;                // 信号。是位图，每个比特位代表一种信号，信号值=位偏移值+1。
    struct sigaction sigaction[32];  // 信号执行属性结构，对应信号将要执行的操作和标志信息。
    long blocked;               // 进程信号屏蔽码（对应信号位图）。
    int exit_code;              // 任务停止执行后的退出码，其父进程会来取。
    unsigned long start_code;   // 代码段地址。
    unsigned long end_code;     // 代码长度（字节数）。
    unsigned long end_data;     // 代码长度 + 数据长度（字节数）。
    unsigned long brk;          // 总长度（字节数）。
    unsigned long start_stack;  // 堆栈段地址。
    long pid;                   // 进程标识号(进程号)。
    long father;                // 父进程号。
    long pgrp;                  // 进程组号。
    long session;               // 会话号。
    long leader;                // 会话首领。
    unsigned short uid;         // 用户标识号（用户 id）。
    unsigned short euid;        // 有效用户 id。
    unsigned short suid;        // 保存的用户 id。
    unsigned short gid;         // 组标识号（组 id）。
    unsigned short egid;        // 有效组 id。
    unsigned short sgid;        // 保存的组 id。
    long alarm;                 // 报警定时值（滴答数）。
    long utime;                 // 用户态运行时间（滴答数）。
    long stime;                 // 系统态运行时间（滴答数）。
    long cutime;                // 子进程用户态运行时间。
    long cstime;                // 子进程系统态运行时间。
    long start_time;            // 进程开始运行时刻。
    unsigned short used_math;   // 标志：是否使用了协处理器。
    int tty;                    // 进程使用 tty 终端的子设备号。-1 表示没有使用。
    unsigned short umask;       // 文件创建属性屏蔽位。
    struct m_inode * pwd;       // 当前工作目录 i 节点结构指针。
    struct m_inode * root;      // 根目录 i 节点结构指针。
    struct m_inode * executable; // 执行文件 i 节点结构指针。
    unsigned long close_on_exec; // 执行时关闭文件句柄位图标志。（参见 include/fcntl.h）
    struct file * filp[NR_OPEN]; // 文件结构指针表，最多 32 项。表项号即是文件描述符的值。
    struct desc_struct ldt[3];  // 局部描述符表。0-空，1-代码段 cs，2-数据和堆栈段 ds&ss。
    struct tss_struct tss;      // 进程的任务状态段信息结构。
```

进程独立的地址空间，其拥有的资源和数据怎样与其他进程共享？进程切换时切的内容？

3.4 进程操作

- 进程操作的职责是对进程生命周期中的状态变迁实施有效的管理
- 进程操作的主要功能包括：
  - 进程的创建
  - 进程的撤销
  - 进程的阻塞
  - 进程的唤醒
- 进程挂起、阻塞等状态变化属于进程调度的事情
- 进程操作功能是由操作系统的内核来实现的
- 进程操作功能是通过执行各种原语来实现的
  - 原语是由若干条机器指令构成的，用于完成某一特定功能的一段程序
  - 原语在执行期间不可分割，所以原语操作具有原子性

3.4 进程操作

3.4.1 进程创建

进程创建是由创建原语实现的

当需要进程时，就可以建立一个新进程

被创建的进程称为子进程，建立进程的进程称为父进程

创建原语的主要功能是为被创建进程形成一个PCB，并填入相应的初始值

进程创建主要操作过程：

（1）先向系统申请一个空闲PCB结构

（2）再根据父进程所提供的参数将子进程的PCB初始化

（3）将此PCB插入就绪队列（或就绪挂起/创建队列）

（4）最后返回一个进程的标识号

典型进程创建系统调用fork()

3.4 进程操作

3.4.2 进程撤销

进程撤销是由撤销原语实现的

一个进程在完成其任务后，应予以撤销，以便及时释放其所占用的各类资源

撤销操作可采用两种撤销策略：

- 一种策略是只撤销一个具有指定标识符的进程
- 另一种策略是撤销指定进程及其子孙进程

3.4 进程操作

3.4.2 进程撤销

进程撤销的主要操作过程：

（1）先从PCB集合中找到被撤销进程的PCB

（2）若被撤销进程正处于运行状态，则应立即停止该进程的执行，设置调度标志，以便进程撤销后将处理机分给其他进程

（3）对后一种撤销策略，若被撤销进程有子孙进程，还应将该进程的子孙进程予以撤销

（4）对于被撤销进程所占有的资源，或者归还给父进程，或者归还给系统

（5）最后撤销它的PCB

典型进程撤销系统调用exit()

3.4 进程操作

3.4.3 进程阻塞与唤醒

阻塞原语的作用是将进程由执行状态转为阻塞状态

唤醒原语的作用则是将进程由阻塞状态变为就绪状态

阻塞操作阻塞一个进程的过程：

（1）由于该进程正处于执行状态，故应先中断处理机和保存该进程的CPU现场

（2）然后将该进程插入到等待该事件的队列中

（3）再从就绪进程队列中选择一个进程投入运行

- 对处于阻塞状态的进程，当该进程期待的事件出现或完成时，将阻塞进程唤醒，使其进入就绪状态。
- 由发现者进程或OS调用唤醒原语

唤醒时机和唤醒者？

典型进程阻塞系统调用sleep()、wait()
典型进程唤醒系统调用wakeup()

3.4 进程操作

3.4.3 进程阻塞与唤醒

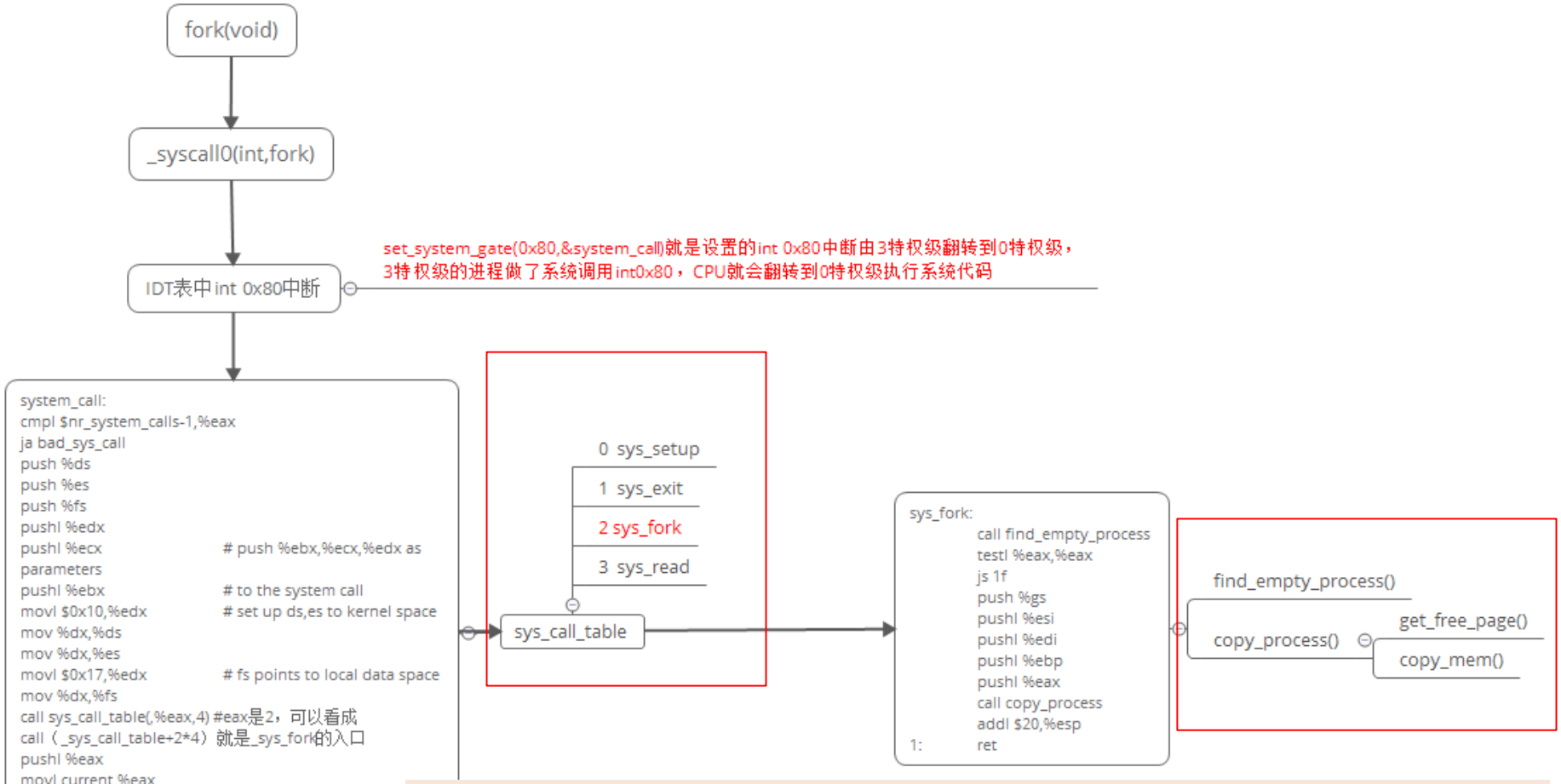
注意：

（1）一个进程由执行状态转变为阻塞状态，是该进程自己调用阻塞原语去完成的

（2）而进程由阻塞状态到就绪状态，是另一个发现者进程（或中断服务程序） 或OS调用唤醒原语实现的，一般这个发现者进程与被唤醒进程是合作的并发进程（通过使用信号量的P/V操作）

Linux fork()

3.4 进程操作

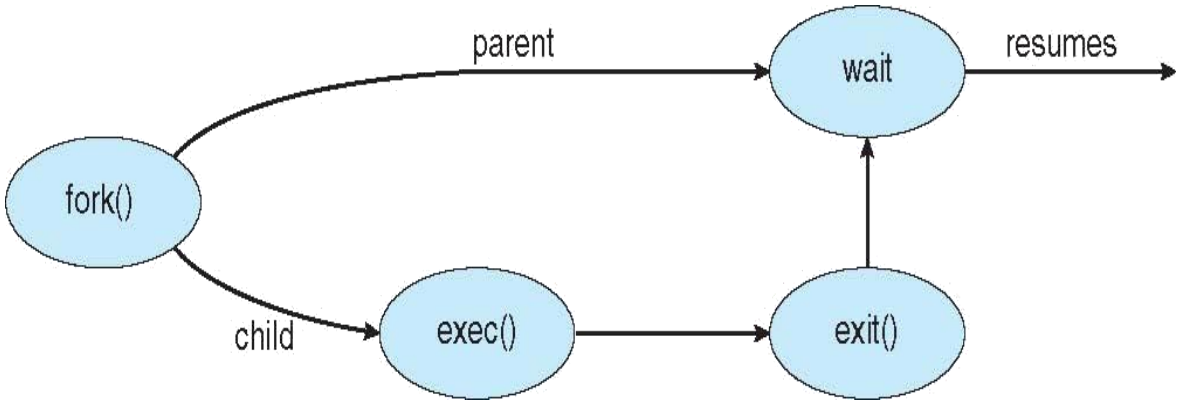


sys_fork做的事：（1）find_empty_process():在task[64]中为进程申请一个空闲位置并获取进程号；（2）copy_process（）：

- 为新进程创建task_struct,将原先进程的task_struct的内容复制给新进程
- 给新进程分配页表，并复制原先进程的页表到新进程
- 共享原先进程的文件
- 设置新进程的GDT项
- 将新进程设置成就绪态，参与进程调度

3.4 进程操作

Linux fork()




```
int main()
{
  pid_t pid;

  pid = fork(); /* fork a child process */
  if (pid < 0)  /* error occurred */
    {  printf("Fork failed!\n");
       exit(-1);
    }
  else if(pid == 0) /* child process */
          {  execlp("/bin/ls",“ls",NULL);
          }
        else /* parent process */
          {  wait(NULL); /* wating the child process to complete */
             printf("The child process complete!\n");
             exit(0);
          }
}
```

3.4 进程操作

Linux fork()：创建与父进程相同的子进程

Parent

```
int main()
{
    pid_t pid;
    char *message;
    int n;
    pid = fork();
    if (pid < 0) {
        perror("fork failed");
        exit(1);
    }
    if (pid == 0) {
        message = "This is the child\n";
        n = 6;
    } else {
        message = "This is the parent\n";
        n = 3;
    }
    for(; n > 0; n--) {
        printf(message);
        sleep(1);
    }
    return 0;
}
```

Child

```
int main()
{
    pid_t pid;
    char *message;
    int n;
    pid = fork();
    if (pid < 0) {
        perror("fork failed");
        exit(1);
    }
    if (pid == 0) {
        message = "This is the child\n";
        n = 6;
    } else {
        message = "This is the parent\n";
        n = 3;
    }
    for(; n > 0; n--) {
        printf(message);
        sleep(1);
    }
    return 0;
}
```

3.4 进程操作

Linux exec()

- 用fork创建子进程后执行的是和父进程相同的程序（但有可能执行不同的代码分支），子进程往往要调用一种exec函数（族）以执行另一个程序。
- 当进程调用一种exec函数时，该进程的用户空间代码和数据完全被新程序替换，从新程序的入口开始执行。
- 调用exec并不创建新进程，所以调用exec前后该进程的id并未改变。

为什么这么做？相同的进程，不同的进程

实例：打开几次记事本，查看进程的PID。再打开多个PPT文件，查看进程PID。

3.5 进程调度

- 3.5.1 进程队列
- 3.5.2 调度程序
- 3.5.3 进程切换

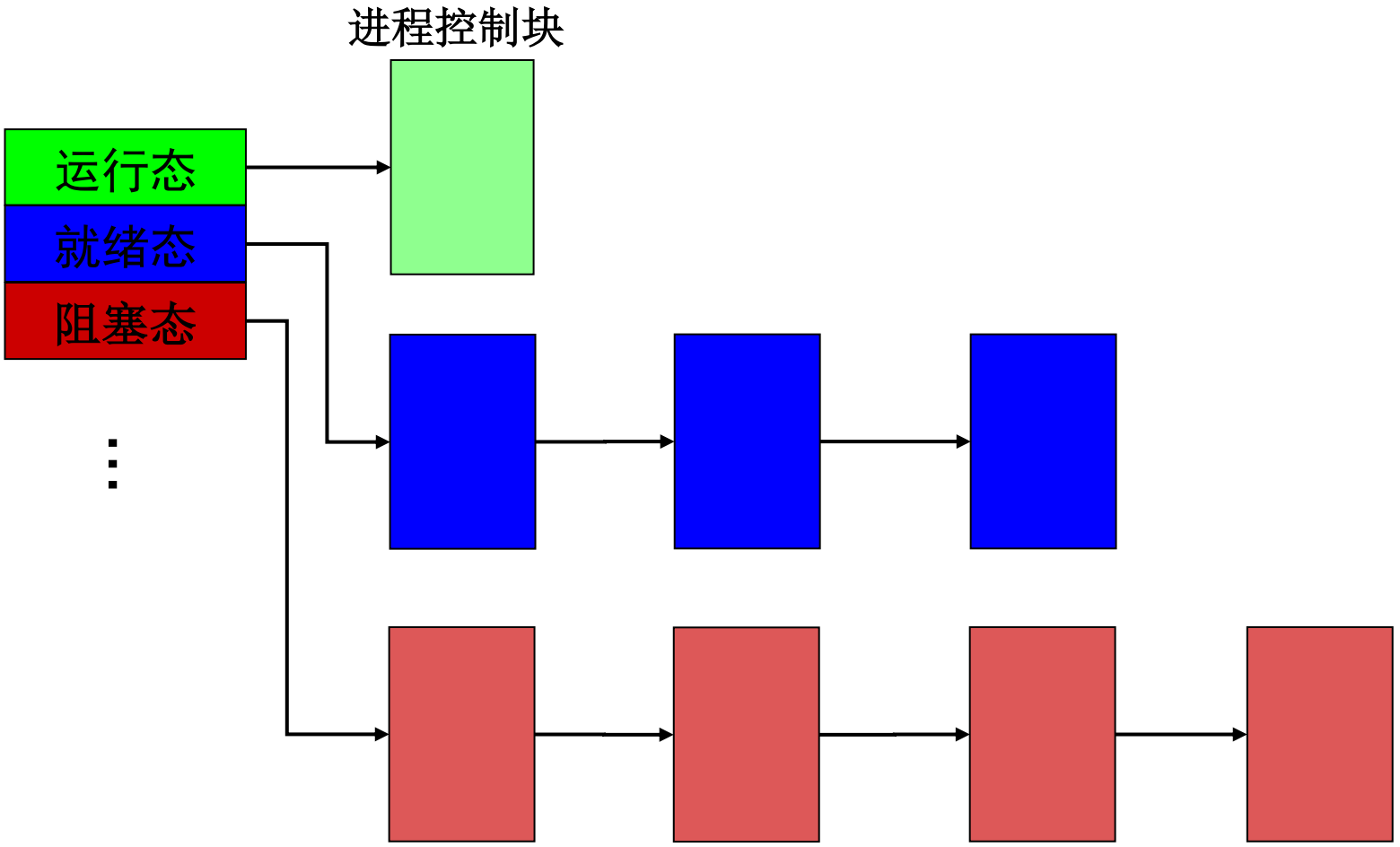
3.5 进程调度

3.5.1 进程队列

为了管理和调度进程，操作系统将具有相同属性或特征的进程保存到队列数据结构中—进程队列

- 进程队列通常以链表队列形式实现
- 进程队列的元素是PCB（Linux称Task_struct）
- 进程队列的分类：
  - 按进程状态：就绪、阻塞、运行、创建、终止
  - 按进程等待的设备：磁盘、键盘
  - 按进程等待的事件：鼠标双击、鼠标右键
  - 按进程等待的信号量、消息队列等

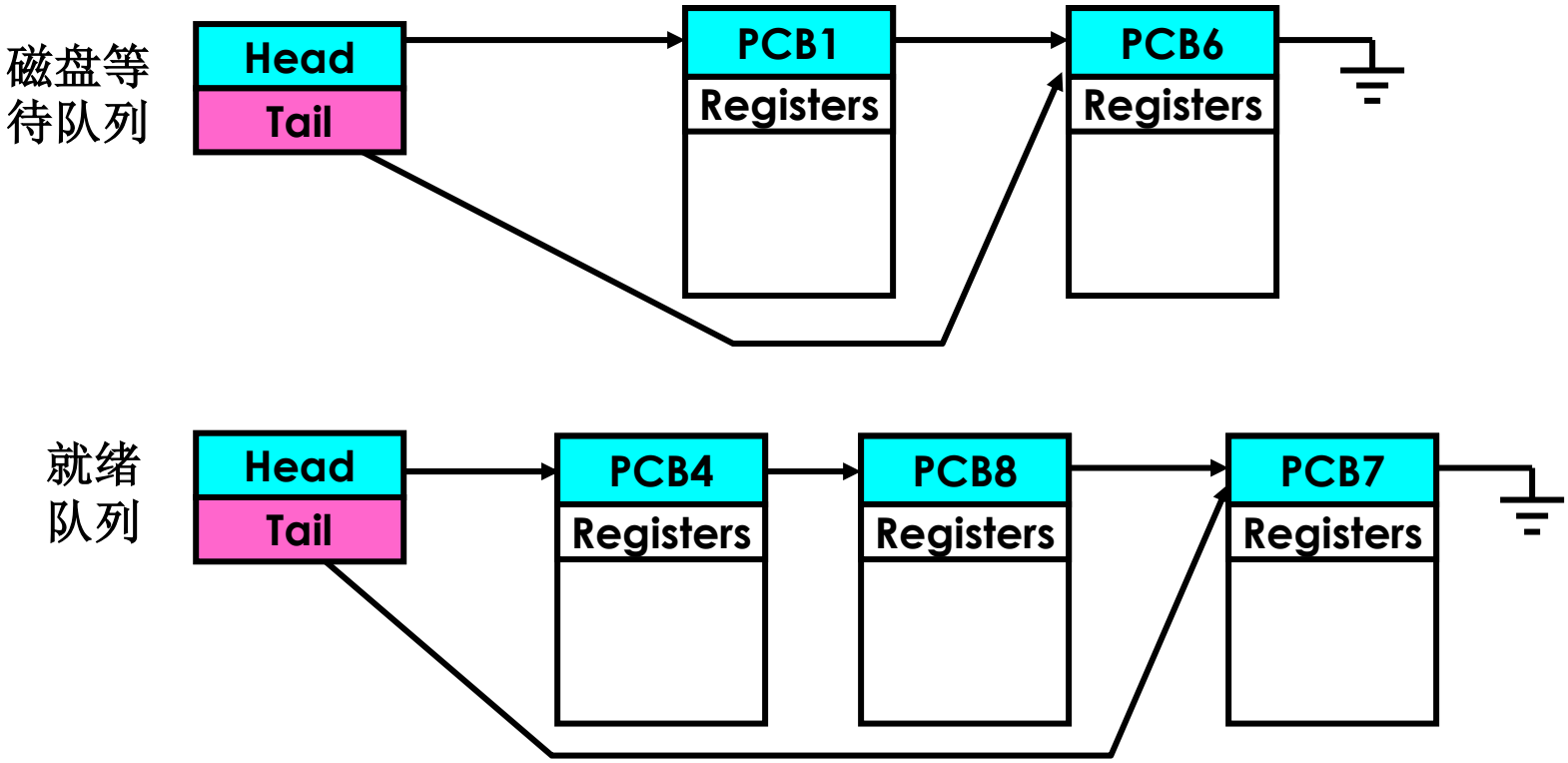
3.5 进程调度



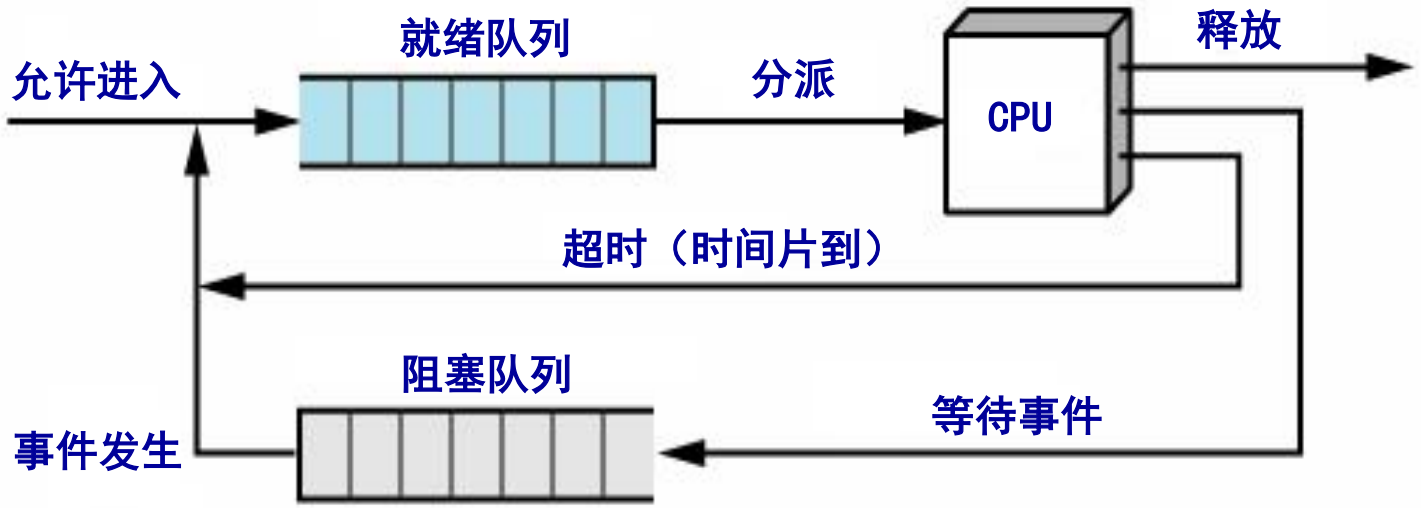
按进程状态划分的链表队列结构

3.5 进程调度

- 进程队列实现示意图

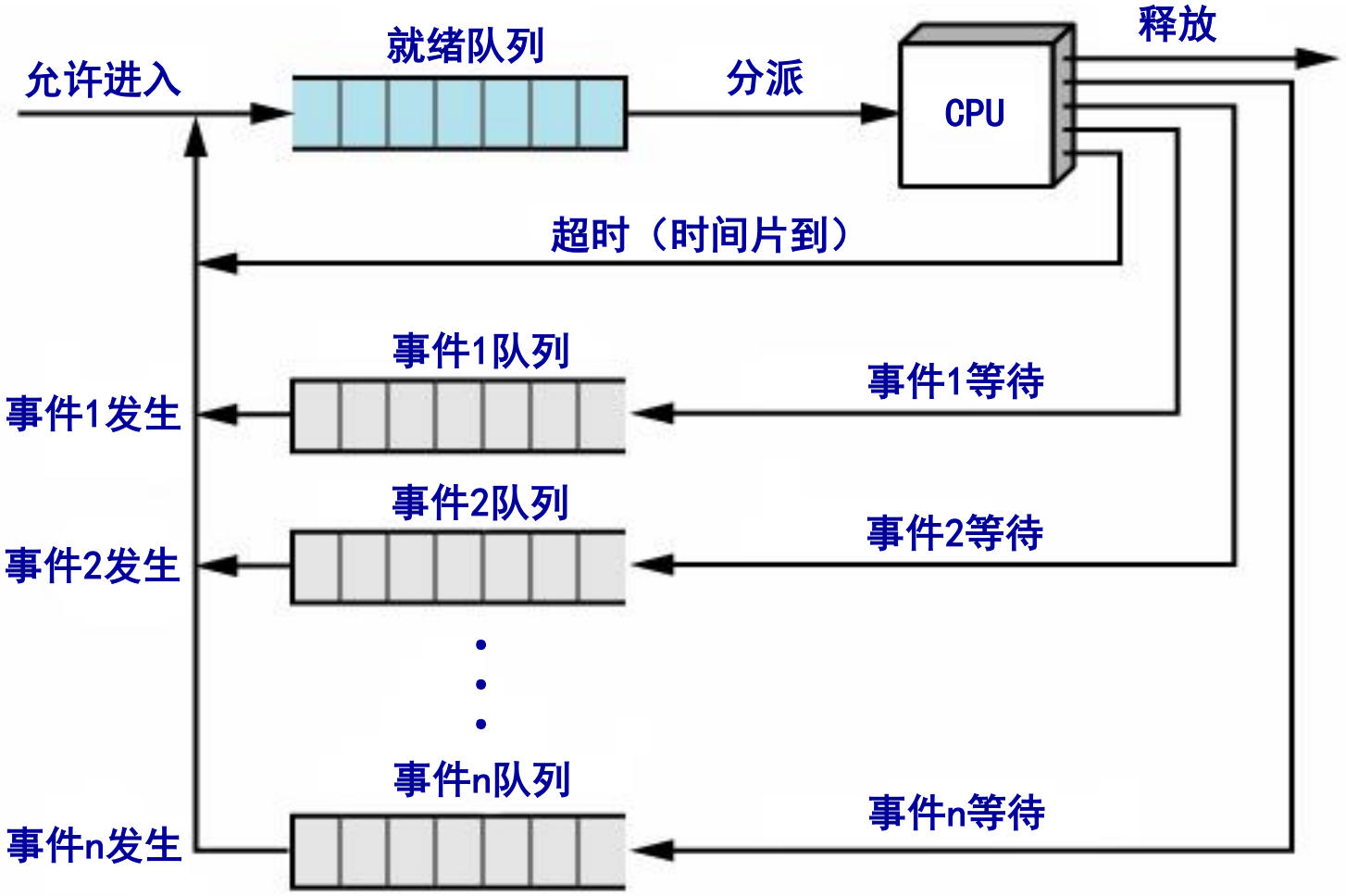


3.5 进程调度



单一的阻塞队列

3.5 进程调度



多条阻塞队列

3.5 进程调度

3.5.2 调度程序

按照一定的规则安排选择。调度种类：

- 长期调度程序（作业调度程序）：

  从磁盘缓冲池中选择进程；*批处理或大规模分布式*

- 中期调度程序：

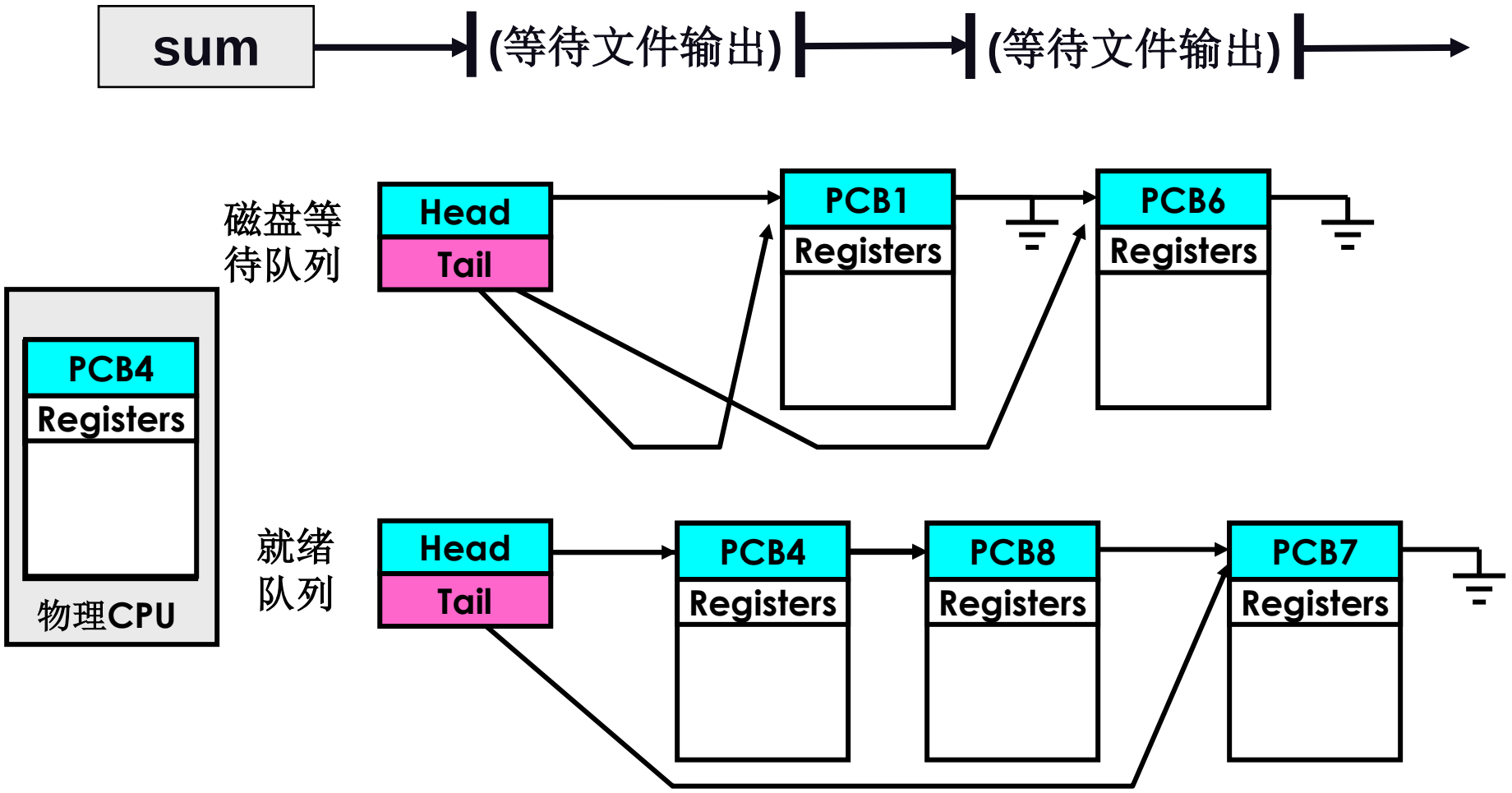
  内外存“交换”进程（换入换出）

- 短期调度程序（CPU调度程序）：

  选择就绪态的进程并为其分配CPU

3.5 进程调度

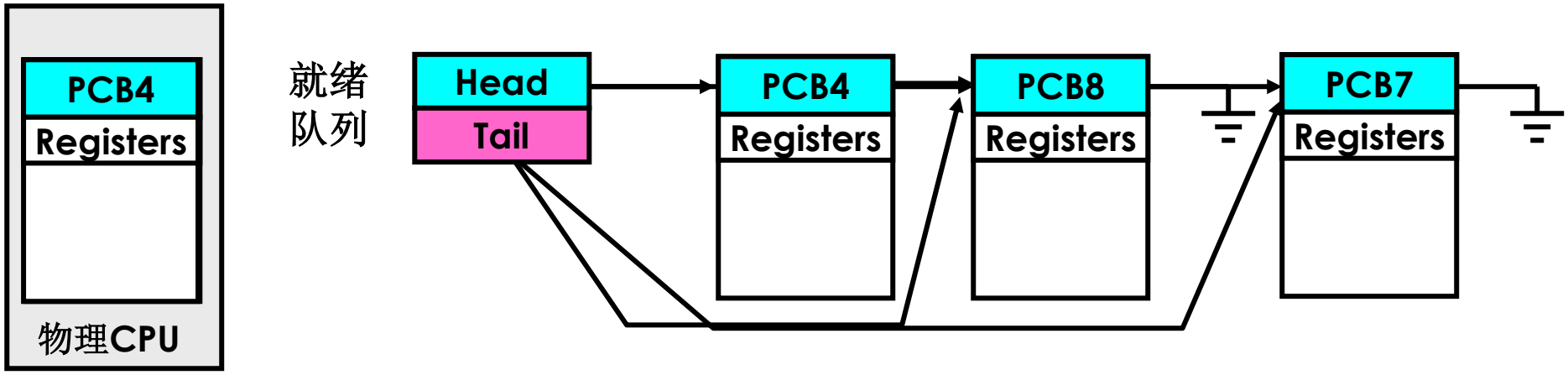
调度情况1： 因等待某些事件而让出CPU



3.5 进程调度

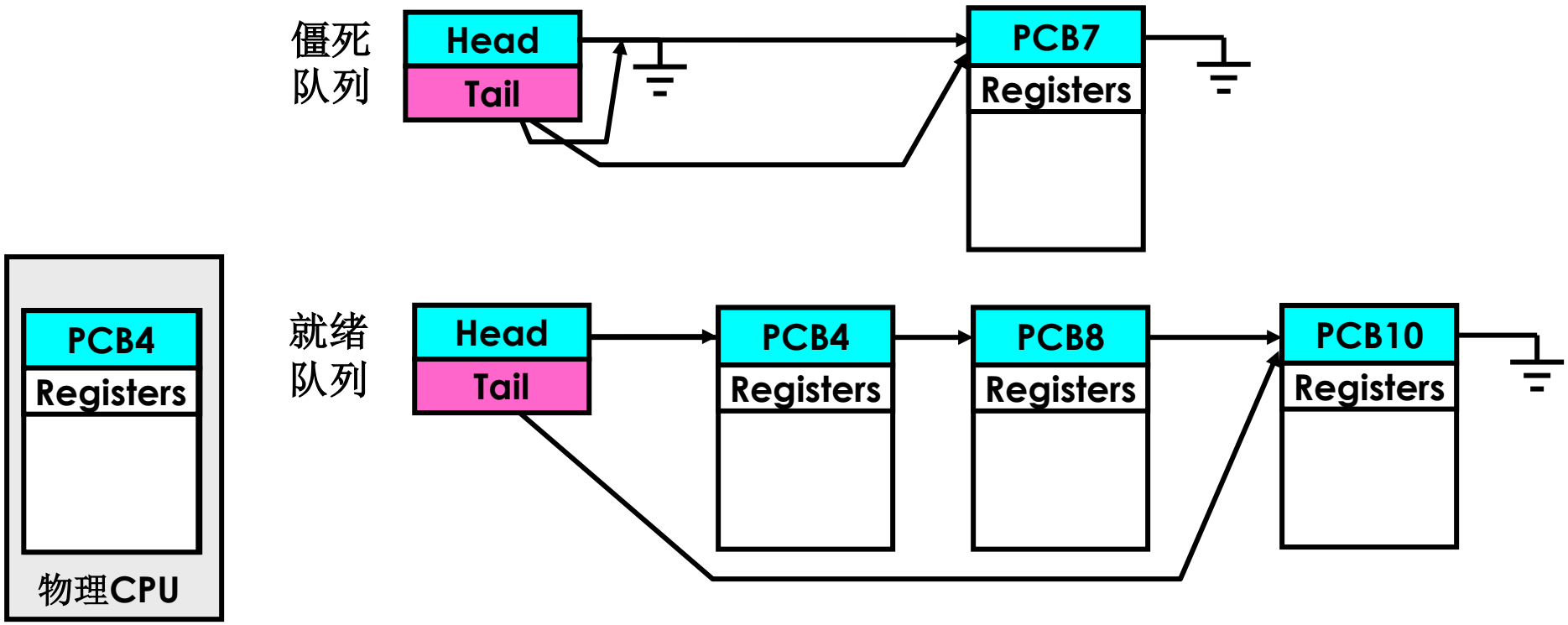
调度情况2：规定的时间片到了

调度情况3：出现了优先级更高的进程



3.5 进程调度

调度情况4：进程的任务完成了，自动终止退出



3.5 进程调度

- 进程调度：（1）进程队列迁移；
  （2）下一个进程选取；
  （3）进程切换

想一想！第（2）步应该有很多种选取策略

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

3.5 进程调度

单核CPU调度算法

- FCFS调度
- 优先级调度
- SJF调度(shortest-job-first，SJF)
- RR调度(轮转法)
- 多级队列调度
- 多级反馈队列调度（各队列间任务可以移动）

调度器选出一个进程，进而分配CPU资源。单个CPU没问题，多个CPU时如何分配?

3.5 进程调度

- **多核CPU调度算法**
  - **全局队列调度**
    - 操作系统维护一个全局的任务等待队列。
    - 当系统中有一个CPU核心空闲时，操作系统就从全局任务等待队列中选取就绪任务开始在此核心上执行。
    - 这种方法的优点是CPU核心利用率较高。
  - **局部队列调度。**
    - 操作系统为每个CPU内核维护一个局部的任务等待队列。
    - 当系统中有一个CPU内核空闲时，便从该核心的任务等待队列中选取恰当的任务执行。
    - 这种方法的优点是任务基本上无需在多个CPU核心间切换，有利于提高CPU核心局部Cache命中率。
  - **主流多核CPU操作系统采用的是基于全局队列的任务调度算法（结合处理器亲和度）**

分布式/云OS调度？ 分类，分组，多级队列（全局、局部）

3.5 进程调度

3.5.3 上下文切换

- 当CPU切换到另一个进程时，需要保存当前进程的状态信息并恢复另一个进程的状态信息，进程状态信息保存在PCB中 – 上下文切换
- 上下文切换时间，CPU时间是浪费的，该指标越小越好
- 进程切换还需保存其他相应的信息，如与内存息息相关的段表、页表等
- 实时操作系统(RTOS)：中断响应、上下文切换、任务调度的时间要求是确定的。

3.5 进程调度

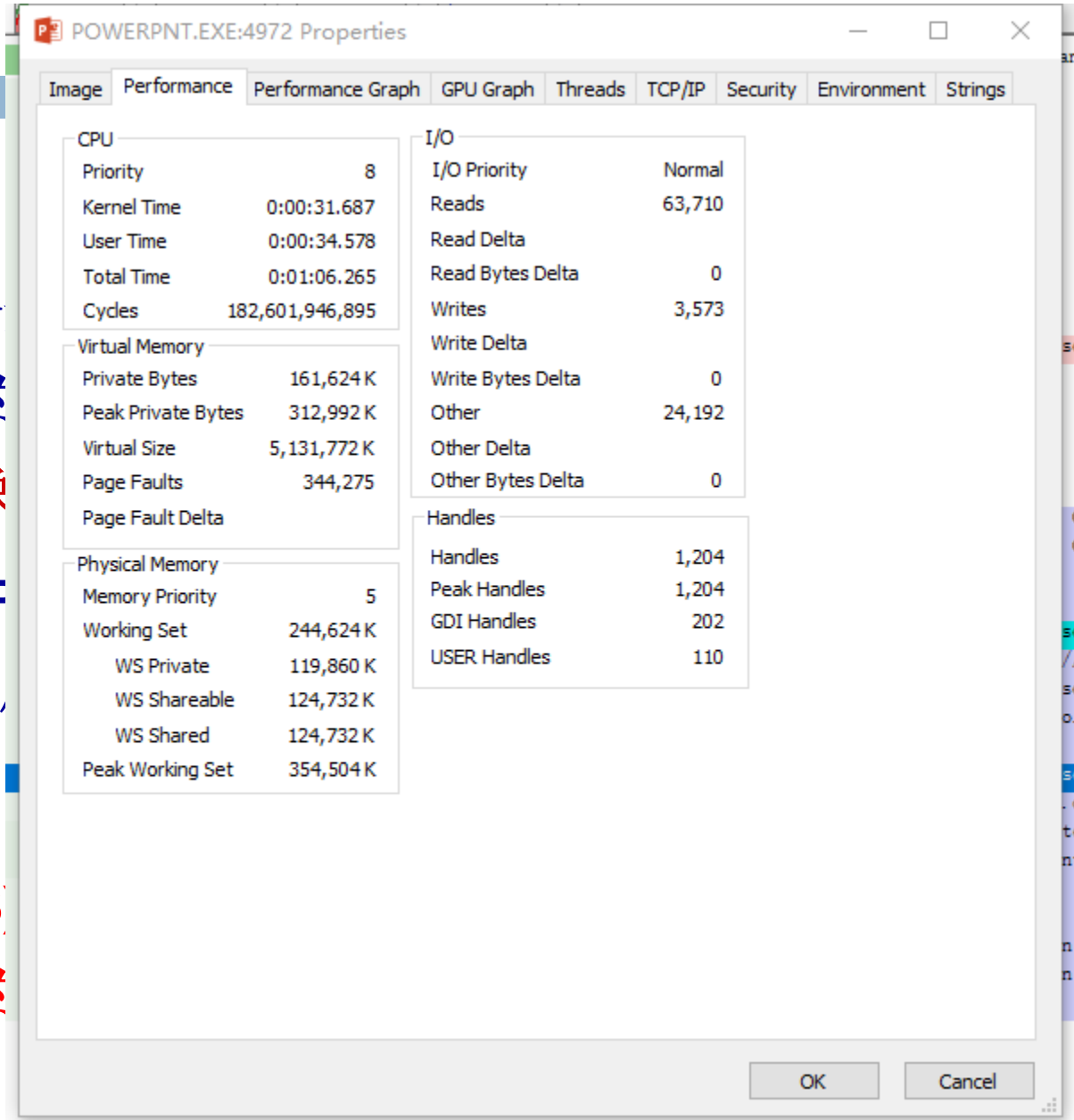
3.5.3 上下文切换

●当CPU切换到另一个
息并恢复另一个进程的
PCB中 - 上下文切换

●上下文切换时间，CP

●进程切换还需保存其
段表、页表等

●实时操作系统(RTOS
度的时间要求是确定的



3.6 进程间通信-IPC

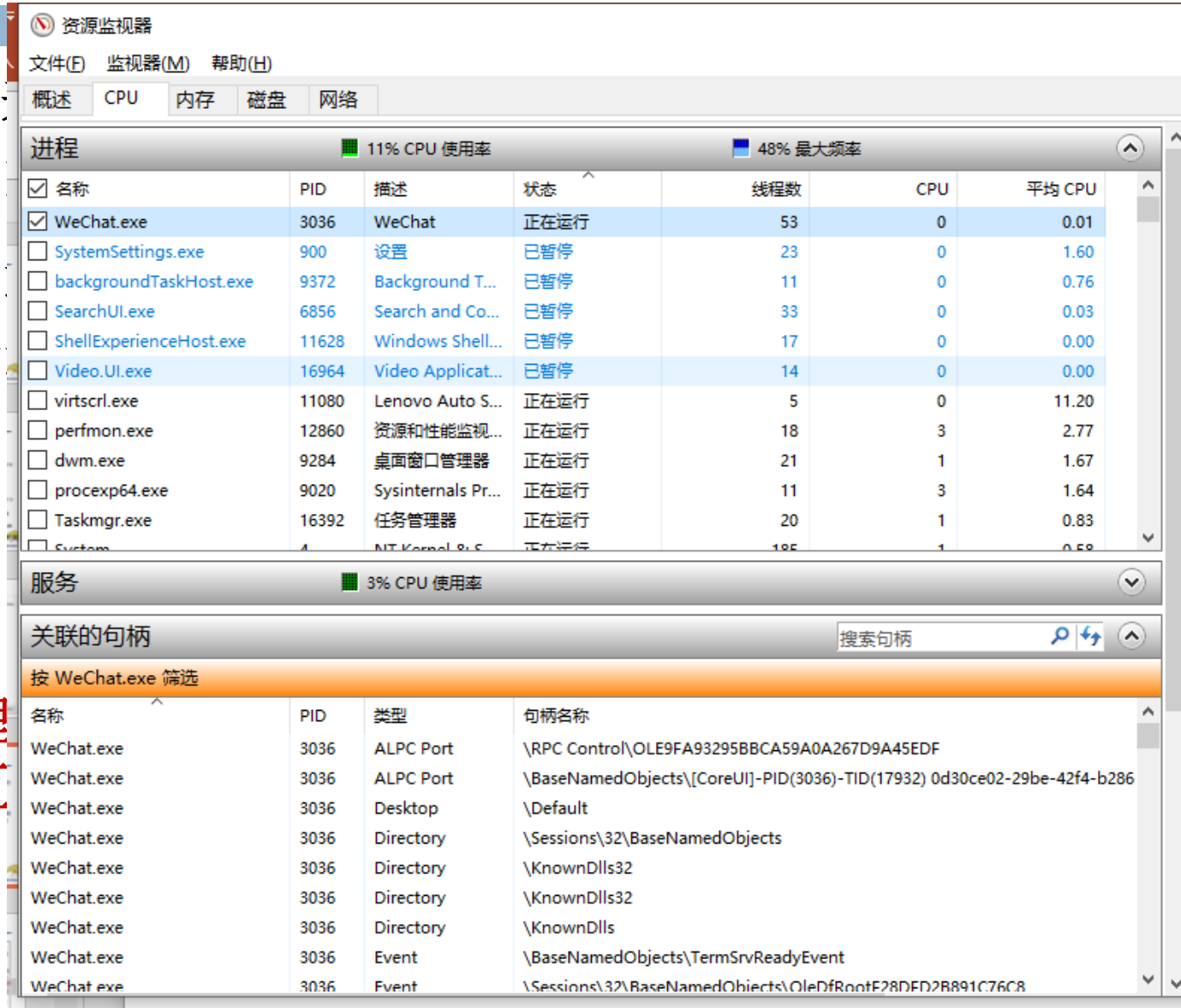
- 操作系统内并发的进程可以是独立的，也可以是协作的
- 独立进程：不能影响其他进程或不能被其他进程影响
- 协作进程：影响其他进程或被其他进程影响
- 进程协作的核心就是共享信息（即进程间通信），达到分工和协作，以提高完成任务的效率或方便用户的目的

观察任务管理器—>资源监视器—>一个进程的关联句柄，查看进程间交互使用的IPC

3.6 进程间通信-IPC

- 操作系统内
- 独立进程：
- 协作进程：
- 进程协作的
  分工和协作，

观察任务管理
看进程间交互



3.6 进程间通信-IPC

- 进程间通信（IPC，Inter-Process Communication），指至少两个进程或线程间传送数据或信号的一些技术或方法。
- IPC资源：支持进程间通信的操作系统资源，如消息队列、事件、信号量、管道、邮箱等。
- 进程是计算机系统分配资源的基本单位。每个进程都有自己的一部分独立的系统资源，彼此是隔离的。为了能使不同的进程互相访问资源并进行协调工作，才有了进程间通信。
- 协作的进程可以运行在同一计算机上或网络连接的不同计算机上。进程间通信技术包括消息传递（直接、间接）、共享内存和远程过程调用。

3.6 进程间通信-IPC

- 进程通信的应用场景
  - 数据传输：一个进程需要将它的数据发送给另一个进程。
  - 共享数据：多个进程想要操作共享数据，一个进程对共享数据的修改，别的进程应该立刻看到。
  - 通知事件：一个进程需要向另一个或一组进程发送消息，通知它（它们）发生了某种事件（如进程终止时要通知父进程）。
  - 资源共享：多个进程之间共享同样的资源。为了作到这一点，需要内核提供锁和同步机制（比如共享一个IO口）。
  - 进程控制：有些进程希望完全控制另一个进程的执行（如Debug进程），此时控制进程希望能够拦截另一个进程的所有陷入和异常，并能够及时知道它的状态改变。

3.6 进程间通信-IPC

- 进程间通信按照本地和远程还可以划分为：
  - （1）本地过程调用LPC，用在多任务操作系统中，使得同时运行的任务能互相会话。
  - （2）远程过程调用RPC，类似于LPC，只是面向网上分布式的主机内的进程。

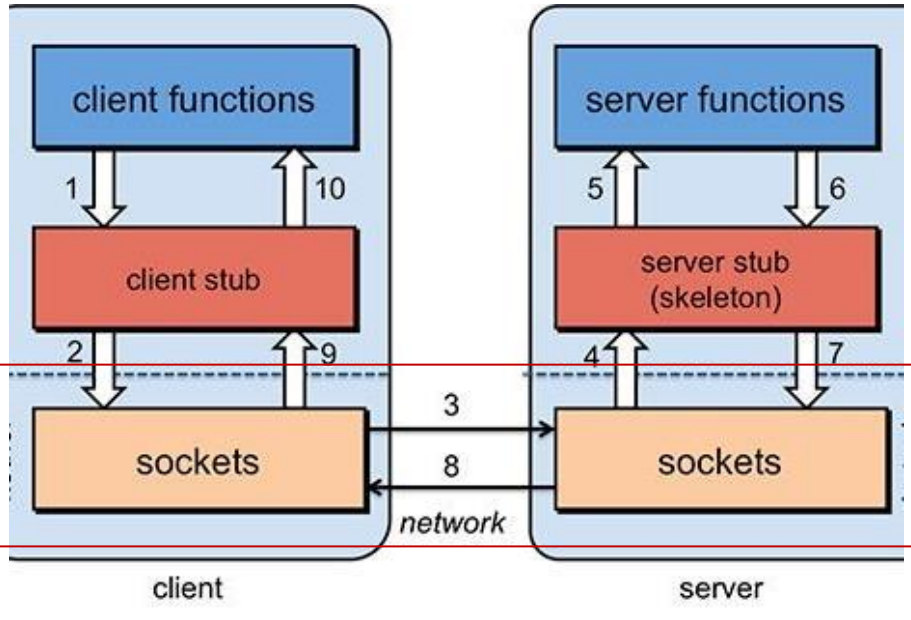
3.6 进程间通信-RPC

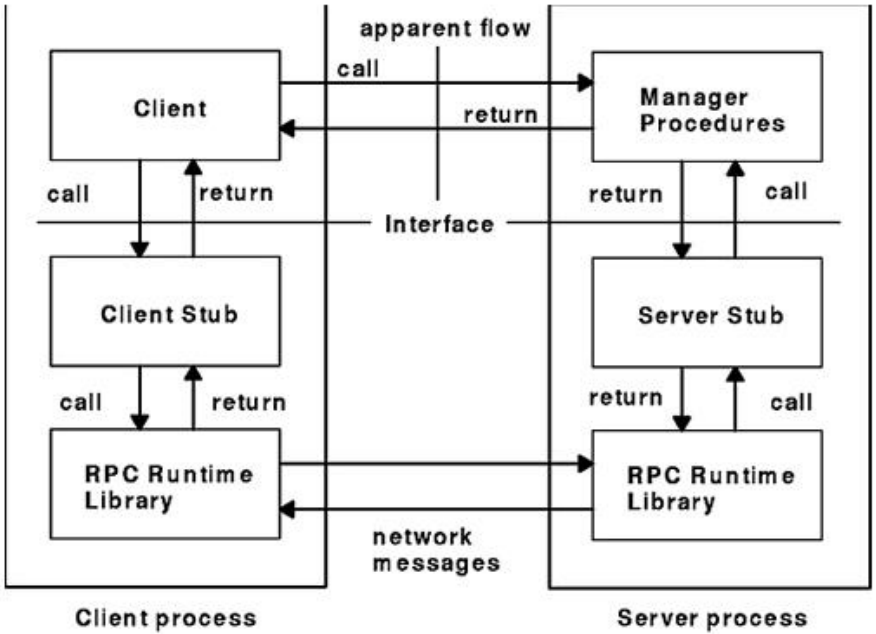
- **RPC 的全称是 Remote Procedure Call，远程过程调用，是一种进程间通信方式。**
- 它允许进程调用另一个地址空间（通常是共享网络的另一台机器上）的过程或函数（以进程方式存在），而不用程序员显式编码这个远程调用的细节。即无论是调用本地接口/服务的还是远程的接口/服务，本质上编写的调用代码基本相同。
- 比如两台服务器A，B，一个应用部署在A服务器上，想要调用B服务器上应用提供的函数或者方法，由于不在一个内存空间，不能直接调用，这时候就可以应用RPC框架来解决

3.6 进程间通信-RPC

RPC框架设计的Client和Server模式(C/S)

- RPC采用客户机/服务器模式。请求程序就是一个客户机，而服务提供程序就是一个服务器。首先，调用进程发送一个有进程参数的调用信息到服务进程，然后等待应答信息。
- 在服务器端，进程保持睡眠状态直到调用信息的到达为止。当一个调用信息到达，服务器获得进程参数，计算结果，发送答复信息，然后等待下一个调用信息，最后，客户端调用过程接收答复信息，获得进程结果，然后调用执行继续进行。





3.6 进程间通信-RPC

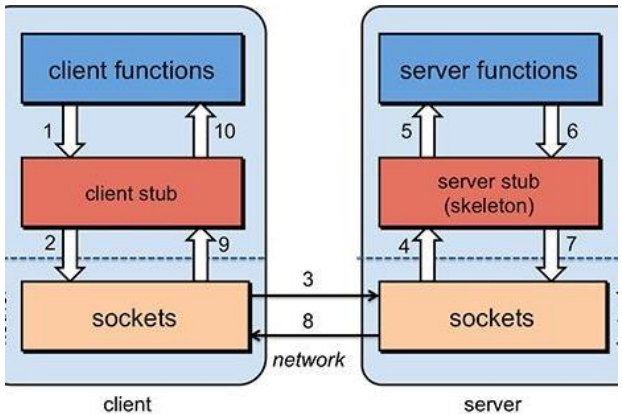
ID存在哪？功能部署到哪？

- **RPC框架设计要做到的最基本的三件事：**

1、**服务端如何确定客户端要调用的函数**；在远程调用中，需要在客户端和服务端分别维护一个**{函数 <–> Call ID}**的对应表， ID在所有进程中都是唯一确定的。客户端在做远程过程调用时，附上这个ID，服务端通过查表，来确定客户端需要调用的函数，然后执行相应函数的代码。

2、**如何进行序列化和反序列化**；客户端和服务端交互时将参数或结果转化为字节流在网络中传输，那么数据转化为字节流的或者将字节流转换成能读取的固定格式时就需要进行序列化和反序列化，序列化和反序列化的速度也会影响远程调用的效率（采用特定的格式XML、JSON)。

3、**如何进行网络传输（选择何种网络协议）**；多数RPC框架选择TCP作为传输协议，也有部分选择HTTP。如gRPC使用HTTP2。不同的协议各有利弊。TCP更加高效，而HTTP在实际应用中更加的灵活(XML,JSON)。



3.6 进程间通信-RPC

- **RPC实现的9个步骤：**

1.服务消费方（client）以本地调用方式调用服务；

2.client stub接收到调用后将方法、参数等组装成网络传输消息体；

3.client stub找到服务地址，并将消息发送到服务端；

4.server stub收到消息后进行解码；

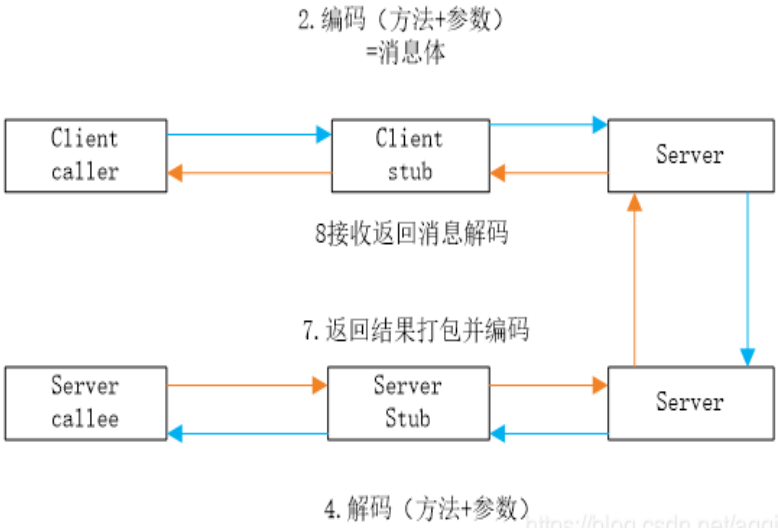
5.server stub根据解码结果调用本地的服务；

6.本地服务执行并将结果返回给server stub；

7.server stub将返回结果打包成消息并发送至消费方；

8.client stub接收到消息，并进行解码；

9.服务消费方得到最终结果。



3.6 进程间通信-RPC

基于HTTP的RPC框架

- RPC OVER HTTP：Microsoft RPC-over-HTTP 部署允许RPC 客户端安全和有效地通过Internet 连接到RPC 服务器程序并执行远程过程调用。
- **Microsoft RPC-over-HTTP 运行在微软IIS上。**它接受来自Internet 的RPC 请求，在这些请求上执行认证，检验和访问检查，**如果请求通过所有的测试，RPC 代理将请求转发给执行真正处理的RPC 服务器。**通过RPC over HTTP，RPC 客户端不和服务器直接通信，它们使用RPC 代理作为中间件。

XML RPC是什么?
使用http协议,使用xml文本的方式传输命令和数据的rpc方式

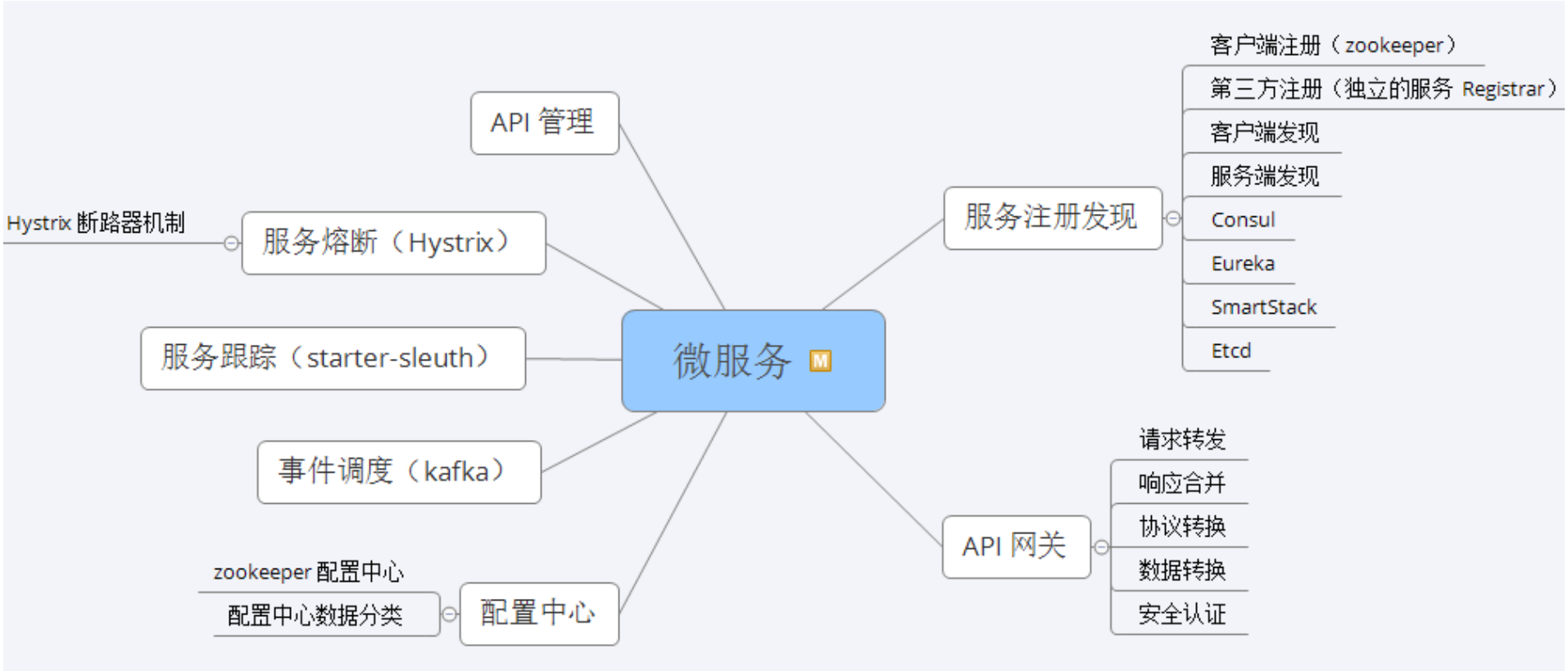
3.6 进程间通信-RPC

- 目前流行的开源RPC框架
  - Dubbo是阿里巴巴开源的一个极为出名的RPC框架。远程接口是基于Java Interface，并且依托于spring框架方便开发。可以方便的打包成单一文件，独立进程运行，和现在的微服务概念一致。
  - gRPC是Google最近公布的开源软件，基于最新的HTTP2.0协议，并支持常见的众多编程语言。框架是基于HTTP协议实现的，底层使用到了Netty框架的支持。

●随着远程调用服务越来越多，服务间的关联越来越复杂，服务的配置和管理越来越复杂。因此需要服务注册和服务发现管理中心，让服务动态的部署和位置管理变得透明。微服务管理框架诞生

ZooKeeper是一个分布式的，开放源码的分布式应用程序协调服务，是Google的Chubby一个开源的实现，是Hadoop和Hbase的重要组件，**对基于RPC微服务提供管理，功能包括：配置维护、域名服务、分布式同步、组服务等。**

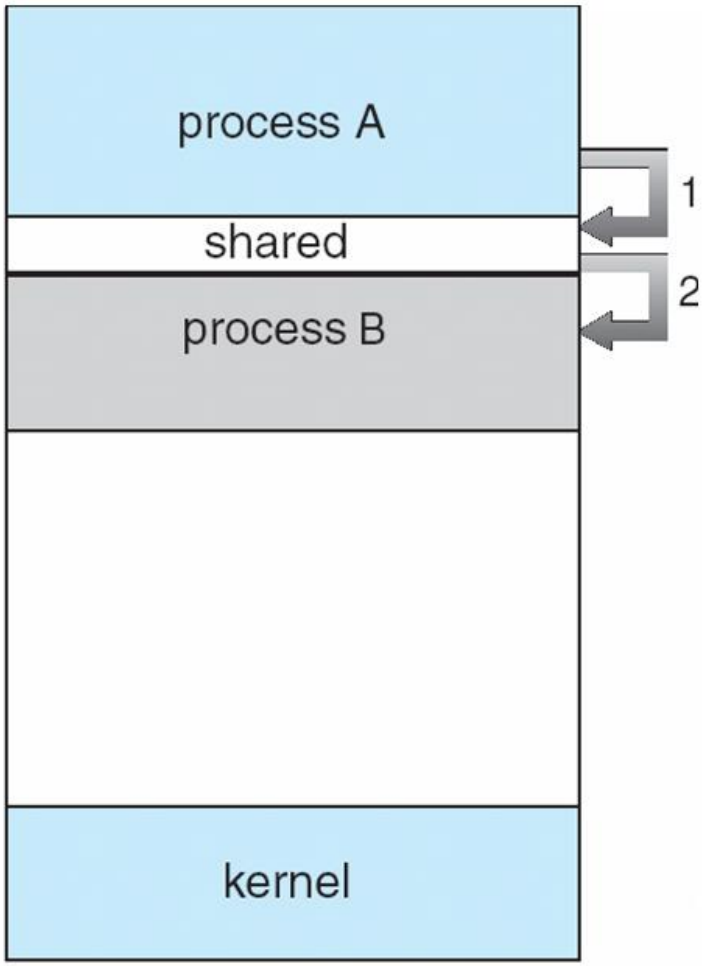
3.6 进程间通信-RPC



，是Google的Chubby一个开源的实现，是Hadoop和Hbase的重要组件，**对基于RPC微服务提供管理，功能包括：配置维护、域名服务、分布式同步、组服务等。**

3.6 进程间通信-LPC

3. 6. 1 共享内存

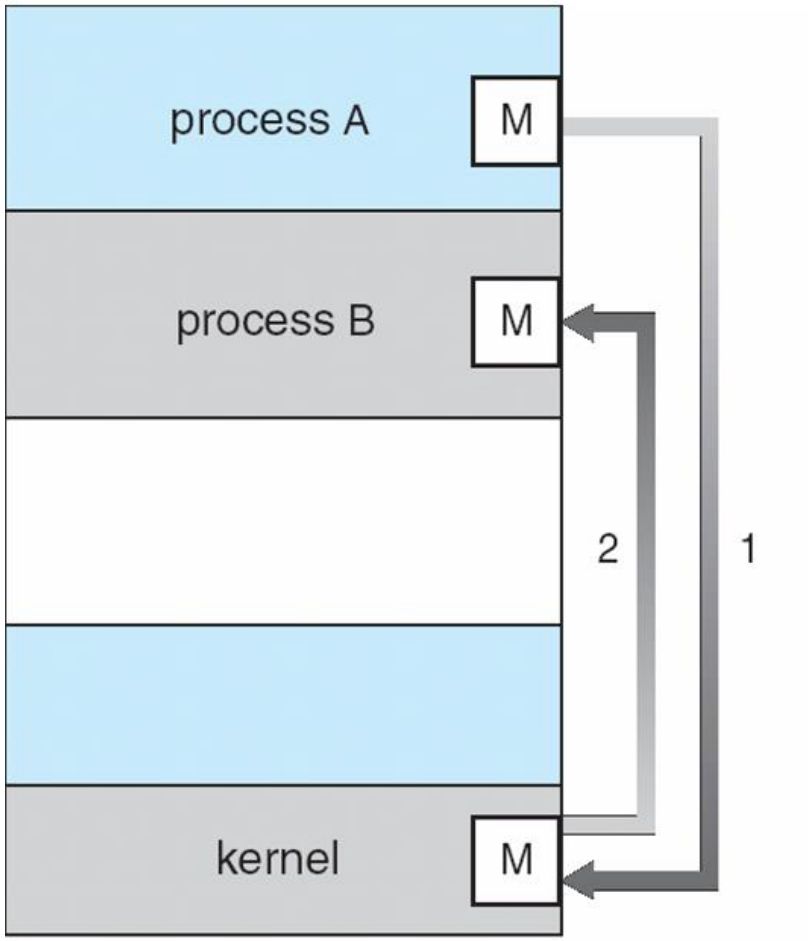


具有各自独立地址空间的进程进行通信时可以：

1. **ftok（）** - 创建共享内存区唯一号
2. **shmget（）** - 获得共享内存区域
3. **shmat（）** - 将共享内存区域映射到进程的空间内
4. 不同进程创建相同名字的共享内存区域，可以同时访问同一段内存区域，从而实现进程间通信。

3.6 进程间通信-LPC

3.6.2 消息传递-直接通信



需要明确给出通信的接收者和发送者。

进程P与Q之间消息传递，通过系统调用：

send (*P, message*)
发送一个消息给进程P

receive(*Q, message*)
从进程Q接收一个消息

3.6 进程间通信-LPC

3. 6. 2 消息传递-间接通信

进程之间通过邮箱\端口\对象\消息队列等来发送和接收消息：

send (*A*, *message*)
发送一个消息到邮箱A

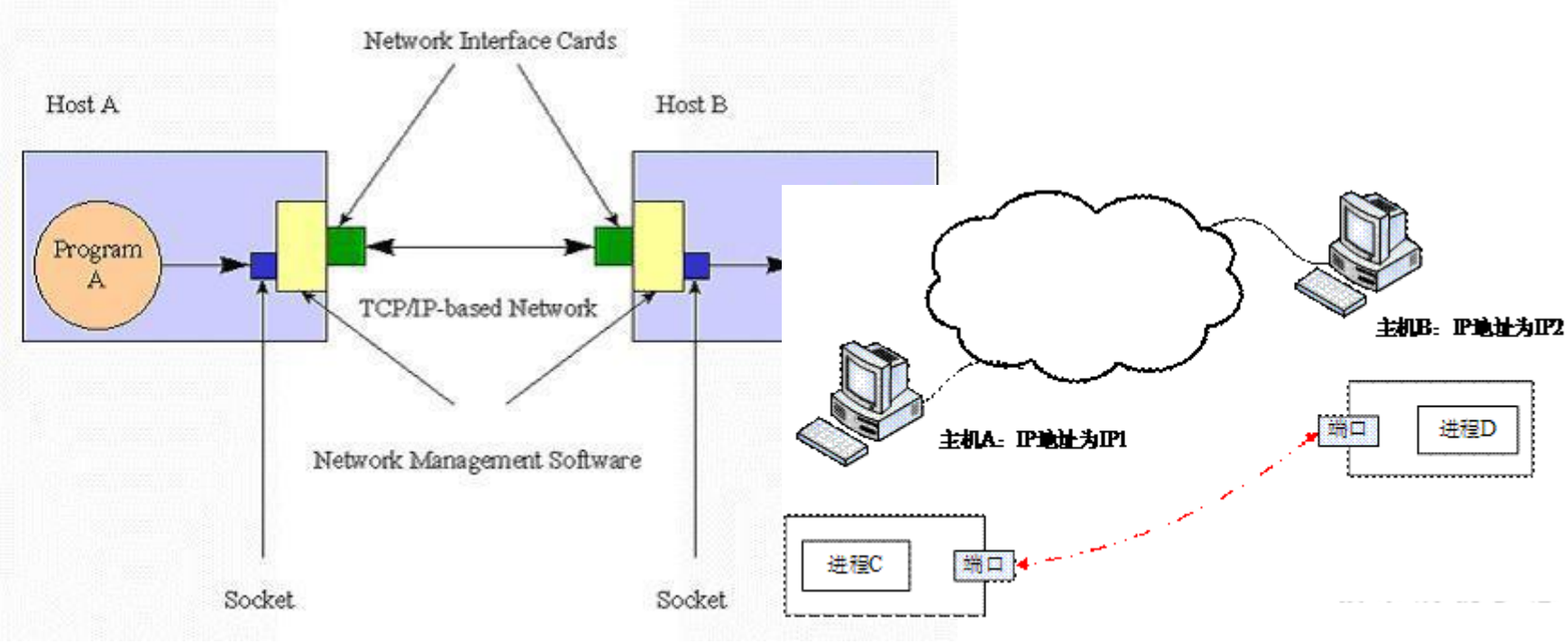
receive(*A*, *message*)
从邮箱A接收一个消息

3.6 进程间通信-linux

3.6.3 Linux进程间通信方法

共享内存，消息传递：管道、消息队列、信号量、套接字等。

套接字：除了在异地的计算机进程间以外，套接口同样适用于本地同一台计算机内部的进程间通信。（查看任务管理器——资源监视器网络部分）



3.6 进程间通信

3.6.3 Linux进程间通信方法

共享内存，消息传递：管道、消息队列、信号量、套接口。

管道：管道是进程间通信中最古老的方式，它包括**无名管道**和**有名管道**两种，前者用于父进程和子进程间的通信，后者用于运行于同一台机器上的任意两个进程间的通信。

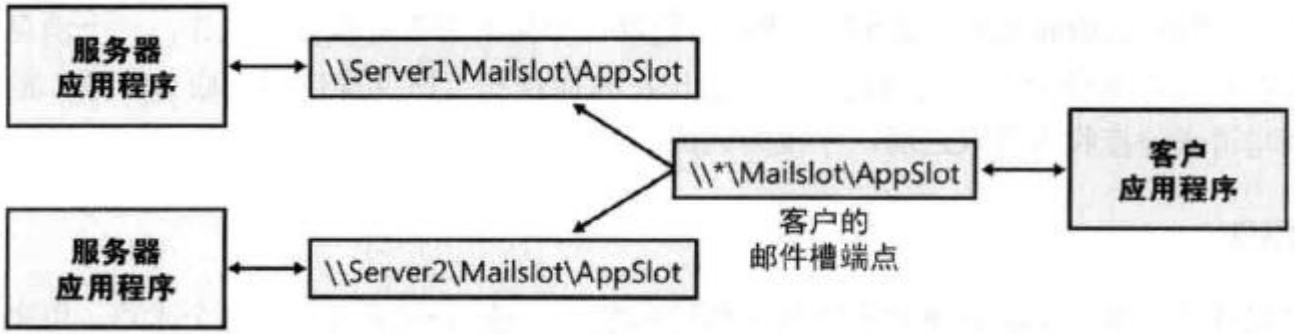
----需要读写进程对管道进行访问权限和顺序控制，类似于文件。

消息队列：与命名管道类似，管道多以（设备）文件形式组织和使用。但消息队列以单个消息为管理对象。

3.6 进程间通信-windows

3.6.4 Windows进程间通信方法

1. 文件映射
2. 共享内存
3. 匿名管道
4. 命名管道



5. 邮件槽（提供了不可靠的单向数据传输，但是邮件槽支持多播）
6. 剪贴板
7. 动态数据交换
8. 对象连接与嵌入
9. 动态连接库
10. 远程过程调用
11. Sockets
12. WM_COPYDATA消息

3.6 进程间通信-VxWorks

3.6.5 Vxworks任务间通信方法

1. 共享内存（Shared Memory）
2. 互斥操作（Mutual Exclusion）
3. 信号量（Semaphore）
4. 消息队列（Message Queue）
5. 管道（Pipe）

VxWorks 操作系统是美国WindRiver公司于1983年设计开发的一种嵌入式实时操作系统（RTOS）。它以其良好的可靠性和卓越的实时性被广泛地应用在通信、军事、航空、航天等高精尖技术及实时性要求极高的领域中，如卫星通讯、军事演习、弹道制导、飞机导航等。在美国的 F-16、FA-18战斗机、B-2 隐形轰炸机和爱国者导弹上，甚至连1997年4月在火星表面登陆的火星探测器、2008年5月登陆的凤凰号，和2012年8月登陆的好奇号也都使用到了VxWorks上。

3.6 进程间通信-VxWorks

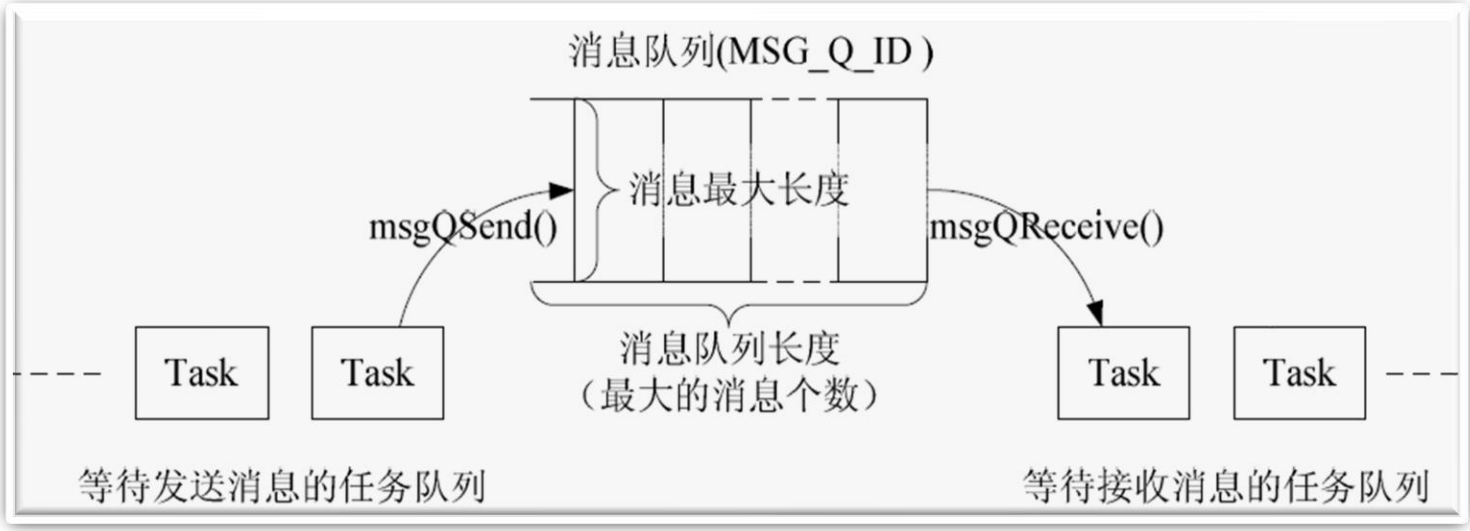
3.6.5 Vxworks消息队列（Message Queue）

1.消息队列用于多任务Task之间，或中断服务程序ISR向任务间传递信息。

2.多个任务（ISR)向同一个消息队列发送消息或接收消息,ISR只能发送。

3.ISR只能用NO_WAIT方式发送消息。

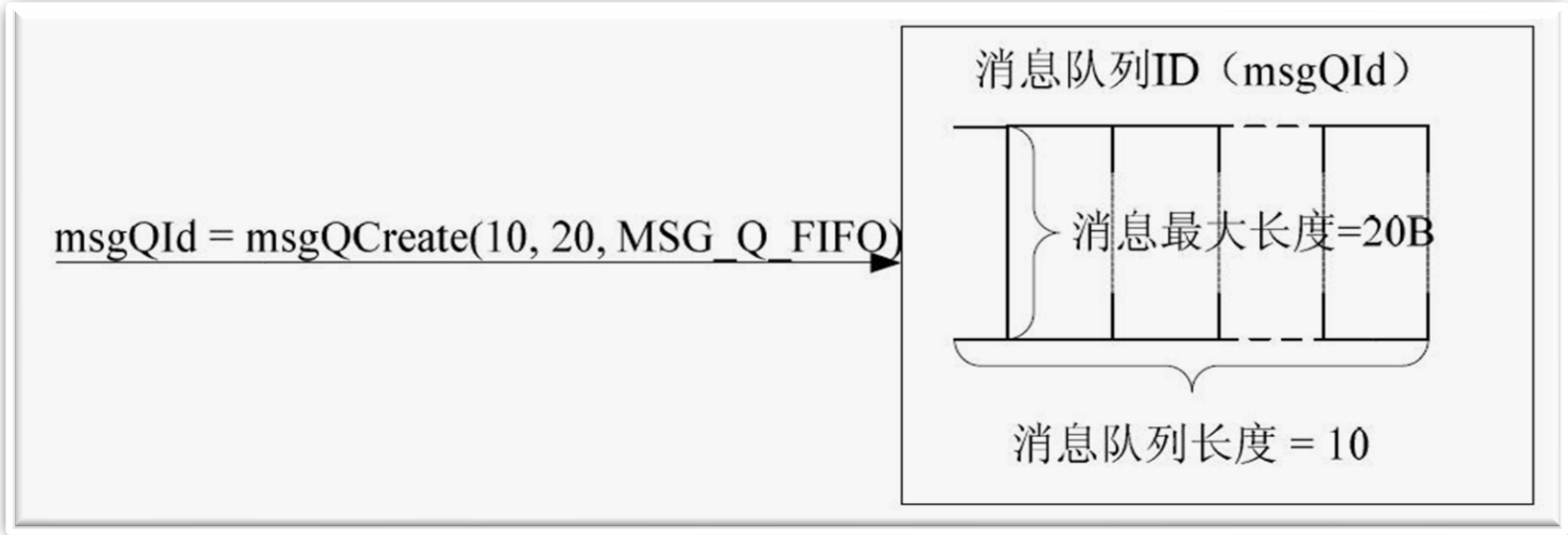
队列读写操作方式：不等，等一定时间，一直等待



3.6 进程间通信-VxWorks

3.6.5 Vxworks消息队列——msgQCreate

```
MSG_Q_ID msgQCreate
(
    int maxMsgs, /* 消息队列长度（最大消息个数） */
    int maxMsgLength, /* 消息队列中消息的最大长度*/
    int options /* FIFO/PRIORITY 等待Task排队方式*/
)
```



3.6 进程间通信-VxWorks

3.6.5 Vxworks消息队列——msgQSend、msgQReceive

```
STATUS msgQSend (
    MSG_Q_ID msgQId, /* 消息队列ID */
    char *buffer, /* 指向要发送的消息的指针*/
    UINT nBytes, /* 消息长度*/
    int timeout, /* 超时时间（tick）/no-wait/forever */
    int priority /* MSG_PRI_NORMAL 或MSG_PRI_URGENT */
)
int msgQReceive (
    MSG_Q_ID msgQId, /* 消息队列ID */
    char *buffer, /* 指向接消息的收缓冲的指针*/
    UINT maxNBytes, /* 接收缓冲的字节数*/
    int timeout /* 超时时间（tick）/no-wait/forever */
)
```

3.6 进程间通信-VxWorks

3.6.5 Vxworks消息队列——msgQSend、msgQReceive

两个函数的timeout参数说明：

•1）对于msgQSend，当消息队列满时，发送者被阻塞，等待时间最长为**timeout**个tick。

•2）对于msgQReceive，当消息队列为空时，接受者被阻塞，等待时间最长为**timeout**个tick。

其中timeout：

•1）为NO_WAIT，则不等待，立即返回。

•2）为WAIT_FOREVER，则一直等待下去，知道队列为非满（对于发送）后非空（对于接受）

•3）为某个整形，则等待timeout个ticks，超过时限，则返回。

3.6 进程间通信-VxWorks

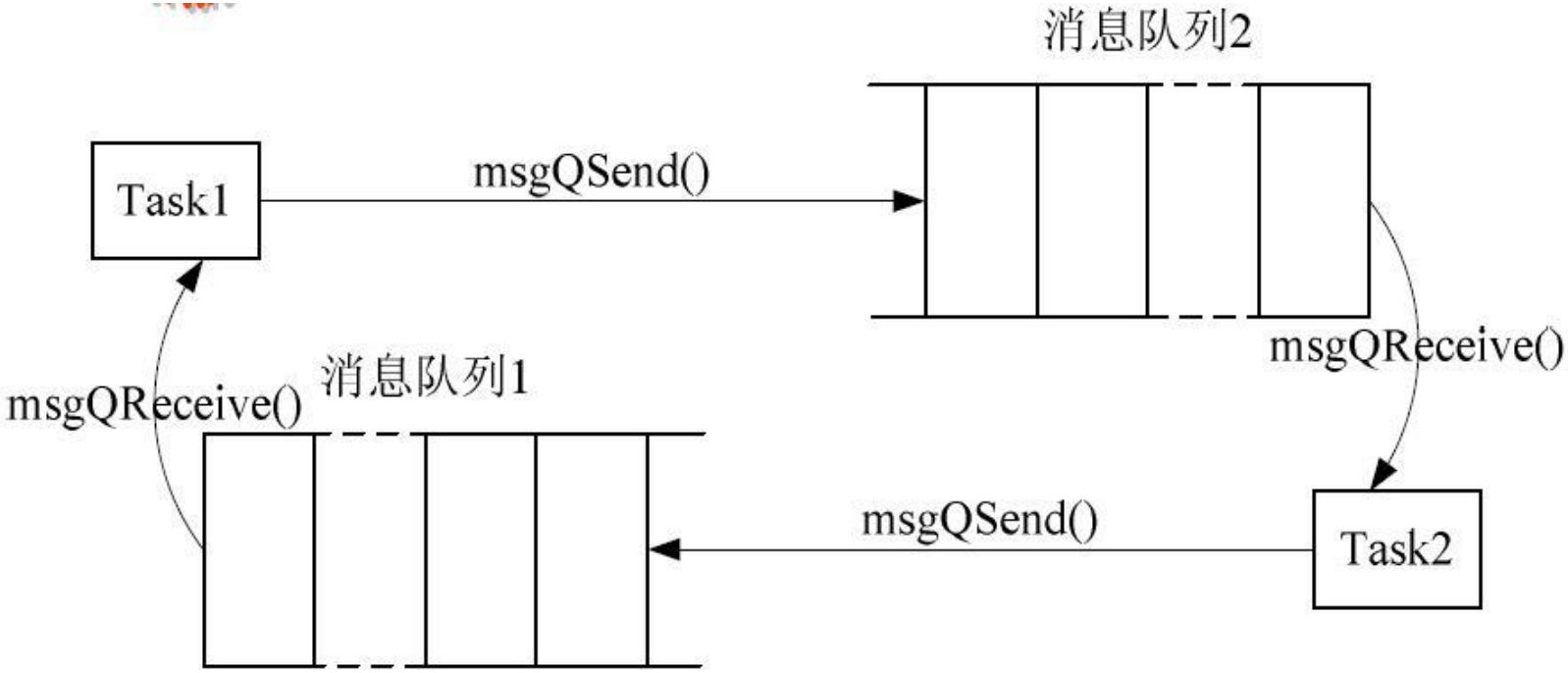
3.6.5 Vxworks消息队列——msgQDelete

```
STATUS msgQDelete (
    MSG_Q_ID msgQId /* 需要删除的消息队列的ID */
)
```

一旦msgQDelete()操作执行完毕，阻塞在该消息队列上的Task，包括阻塞在发送队列、接收队列上的Task，都将被唤醒(Unpend)，而消息队列的ID从此不再有效。

3.6 进程间通信-VxWorks

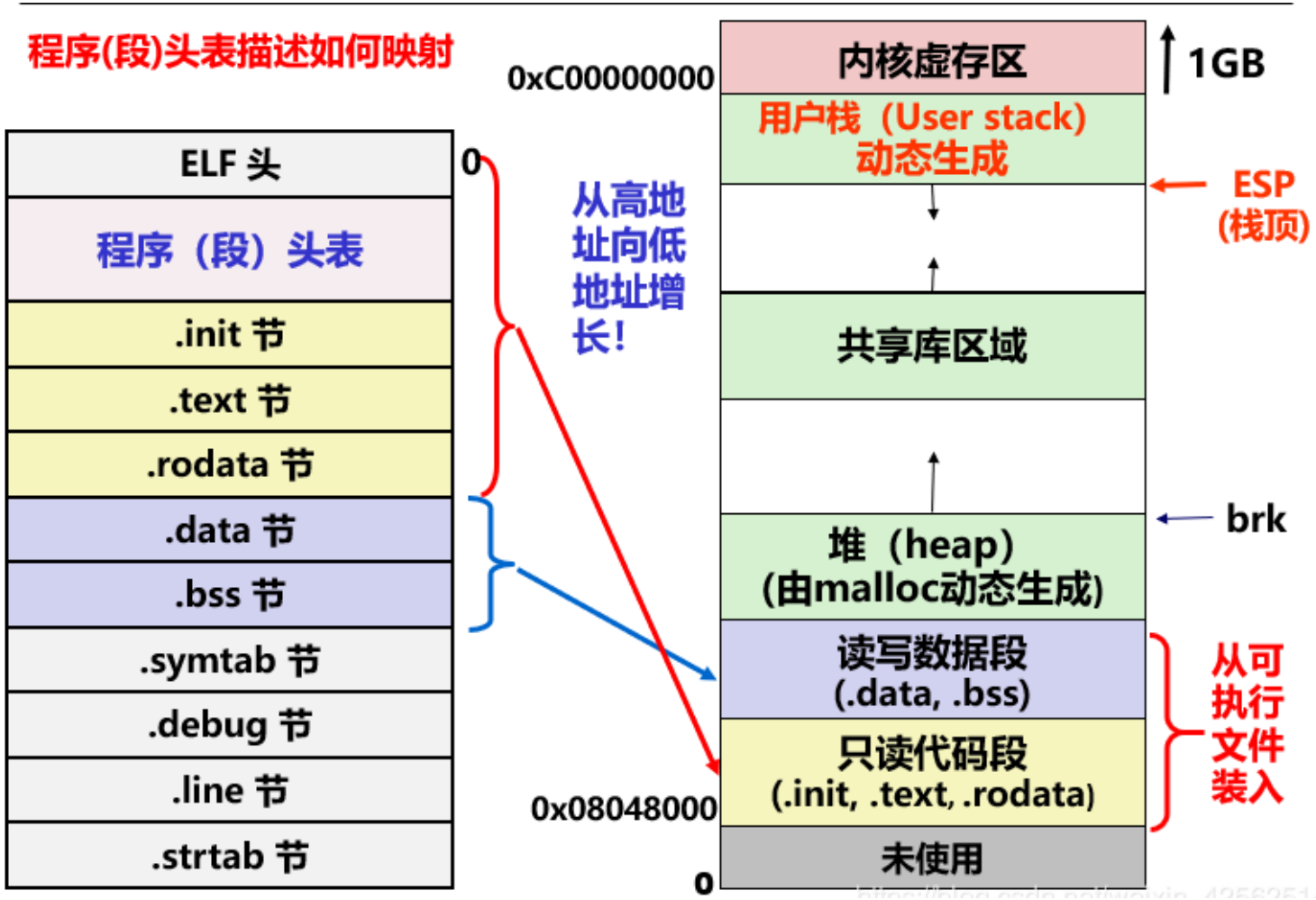
3.6.5 Vxworks消息队列



3.7 进程的可执行映像

- 进程是*运行*的*程序*!
  - 考虑两件事：进程的可执行程序（文件）格式？如何创建进程/如何加载可执行文件？
  - 静态链接、动态链接？ (实例-打开微信可执行文件)
    - 编译原理：什么是链接器、加载器和解释器？

3.7 进程的可执行映像


程序(段)头表描述如何映射
ELF 头
程序（段）头表
.init 节
.text 节
.rodata 节
.data 节
.bss 节
.symtab 节
.debug 节
.line 节
.strtab 节
0
从高地址向低地址增长!
0xC0000000
内核虚存区
1GB
用户栈（User stack）
动态生成
ESP
(栈顶)
共享库区域
brk
堆（heap）
(由malloc动态生成)
读写数据段
(.data, .bss)
只读代码段
(.init, .text, .rodata)
从可执行文件装入
0x08048000
未使用
0

3.7 进程的可执行映像

- 进程是*运行*的*程序*！是可执行文件在内存中的映像
- 代码变成可执行文件的过程：编译、汇编、链接
- 在Linux下，默认的可执行文件/动态库文件（共享目标文件）/目标文件（可重定位文件）/核心转储文件都采用同一种文件格式，称之为ELF文件格式（Linux也支持其他格式的可执行文件，如out）。前三项格式略有不同：
  - 可执行文件没有section header table。
  - 目标文件没有program header table。
  - 动态库文件俩个 header table 都有

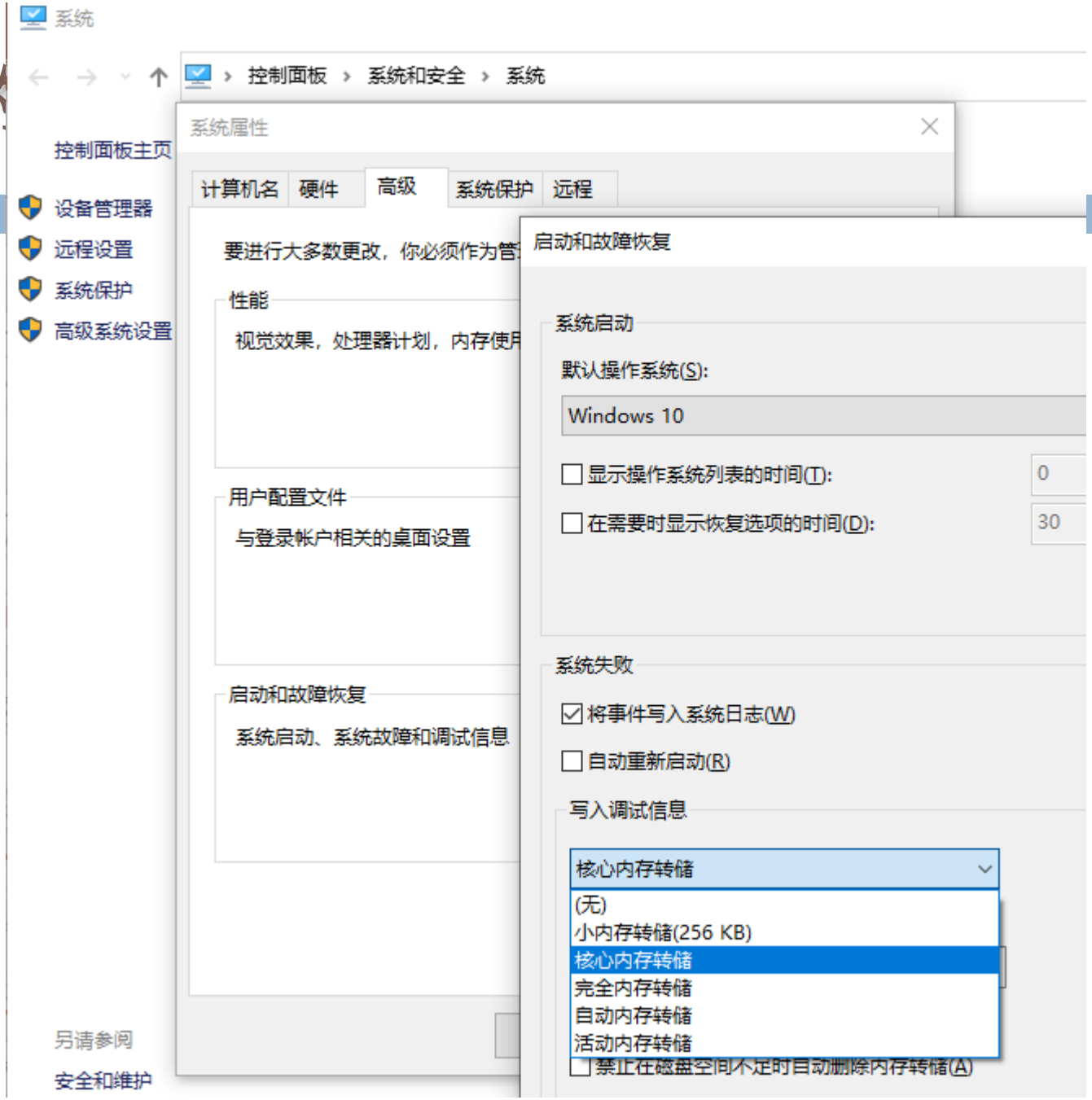
链接器在链接时需要section header table 查看目标文件各个section的信息，然后对各个目标文件进行链接，加载器在加载可执行程序时需要program header table ,它根据这个表把相应的段加载到相应的虚拟内存（虚拟地址空间）中。

3.7 进程的可执行映像

- ELF文件标准里面把系统中采用ELF格式的文件归为四类：

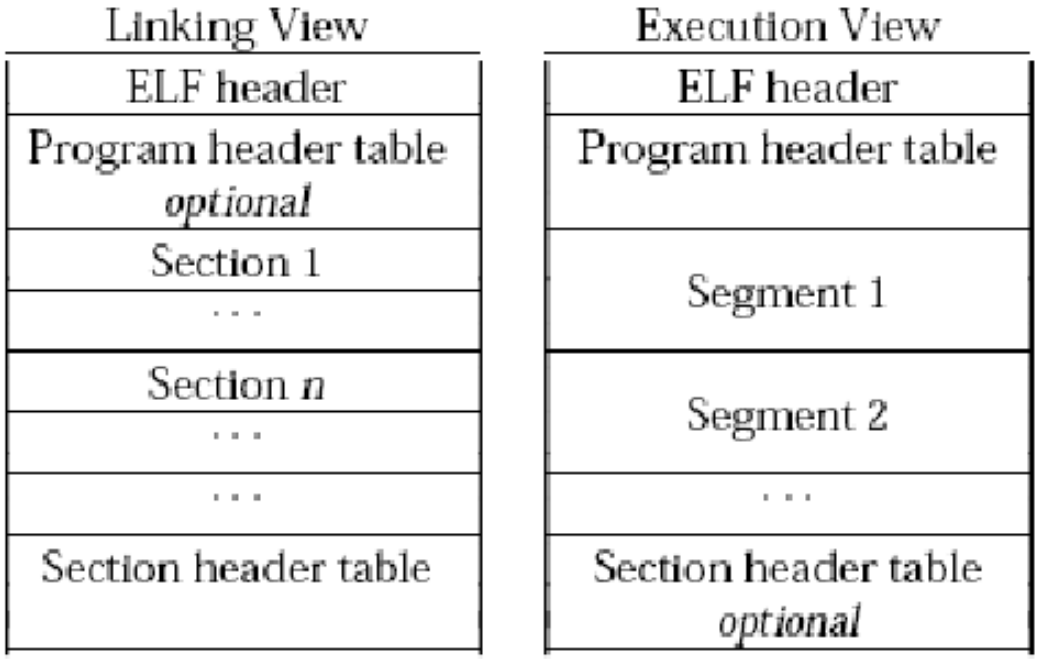
ELF文件类型	说明	实例
可重定位文件 (Relocatable File)	这类文件包含了代码和数据，可以被用来链接成可执行文件或共享目标文件，静态链接库也属于这一类	linux的.o windows的.obj
可执行文件 (Executable File)	这类文件包含了可以直接执行的程序，它的代表就是ELF可执行文件，它们一般都没有扩展名	比如/bin/bash文件； window的.exe
共享目标文件 (Shared Object File)	这种文件包含了代码和数据，可以在以下两种情况下使用。一种是连接器可以使用这种文件跟其他的可重定位文件和共享目标文件链接，产生新的目标文件。第二种是动态链接器可以将几个这种共享目标文件与可执行文件结合，作为进程映像的一部分运行。	linux的.so， 如/lib/glibc-2.5.so windows的DDL
核心转储文件 (Core Dump File)	当进程意外终止时，系统可以将该进程的地址空间的内容及终止时的一些其他信息转储到核心转储文件。	linux下的core dump

3.7 进程的


系统
控制面板 › 系统和安全 › 系统
控制面板主页
设备管理器
远程设置
系统保护
高级系统设置
另请参阅
安全和维护
系统属性
计算机名
硬件
高级
系统保护
远程
要进行大多数更改，你必须作为管
性能
视觉效果，处理器计划，内存使
用户配置文件
与登录帐户相关的桌面设置
启动和故障恢复
系统启动、系统故障和调试信息
启动和故障恢复
系统启动
默认操作系统(S):
Windows 10
显示操作系统列表的时间(T):
0
在需要时显示恢复选项的时间(D):
30
系统失败
将事件写入系统日志(W)
自动重新启动(R)
写入调试信息
核心内存转储
(无)
小内存转储(256 KB)
核心内存转储
完全内存转储
自动内存转储
活动内存转储
禁止在磁盘空间不足时自动删除内存转储(A)

3.7 进程的可执行映像

- 每种操作系统都规定了可以运行文件格式（可执行文件）；
- **ELF(linux)\PE(windows)\COFF(unix)\OUT(linux)\等**



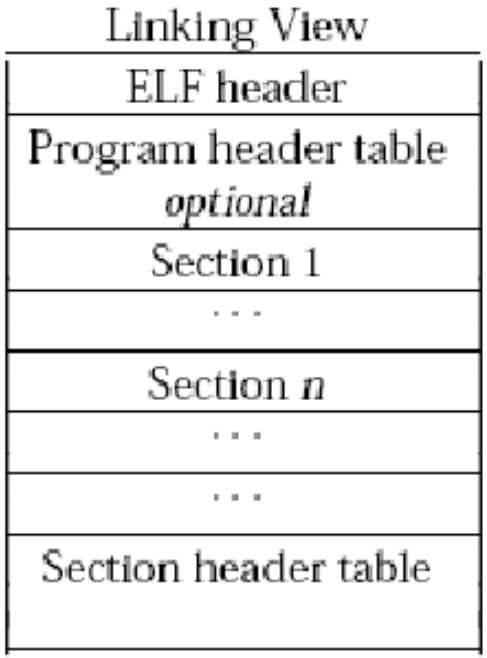
考虑两个问题：程序中的函数变量最后都变成对地址的访问

1）全局变量、静态局部变量、局部变量、常量的特点？

2）调试程序的时候如何能够查看变量值、打断点跟踪、以及查看函数调用关系等？需要什么支持？软件的Debug和Release版？

3.7 进程的可执行映像

- 每种操作系统都规定了可以运行
- ELF(linux)\PE(windows)\COFF



符号表

- 符号表用来体现**作用域**与**可见性信息**
- **符号表的作用**：
  ① 收集符号属性；（词法分析）
  ② 上下文语义的合法性检查的依据；（语法分析）
  ③ 作为目标代码生成阶段地址分配的依据；（语义分析）
- 符号表中语言符号可分为**关键字**（保留字）**符号**，**操作符符号**及**标识符符号**
- 符号表中的**标识符**一般设置的属性项目有：
  ① 符号名
  ② 符号的类型
  ③ 符号的存储类别
  ④ 符号的作用域及可视性
  ⑤ 符号变量的存储分配信息
  ⑥ 符号的其它属性
- 实现符号表的**常用数据结构**
  - 一般的线性表：如：数组，链表，等
  - 有序表：查询较无序表快，如可以采用折半查找
  - 二叉搜索树
  - Hash表
- 开/闭作用域

考虑两个问题：程序中的函数变量

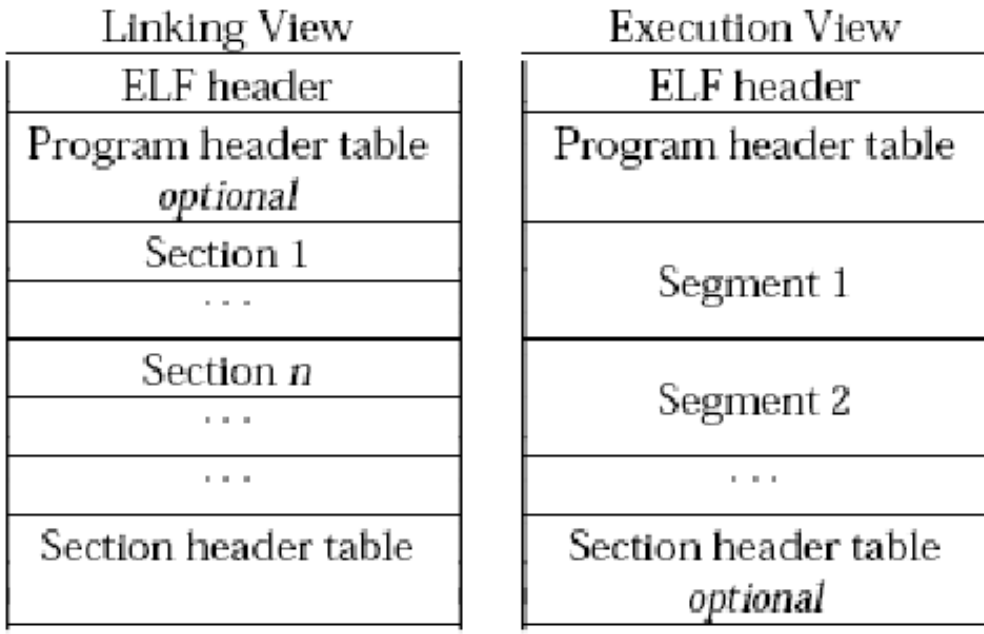
1）全局变量、静态局部变量、局

2）调试程序的时候如何能够查看

调用关系等？需要什么支持？软件的Debug和Release版？

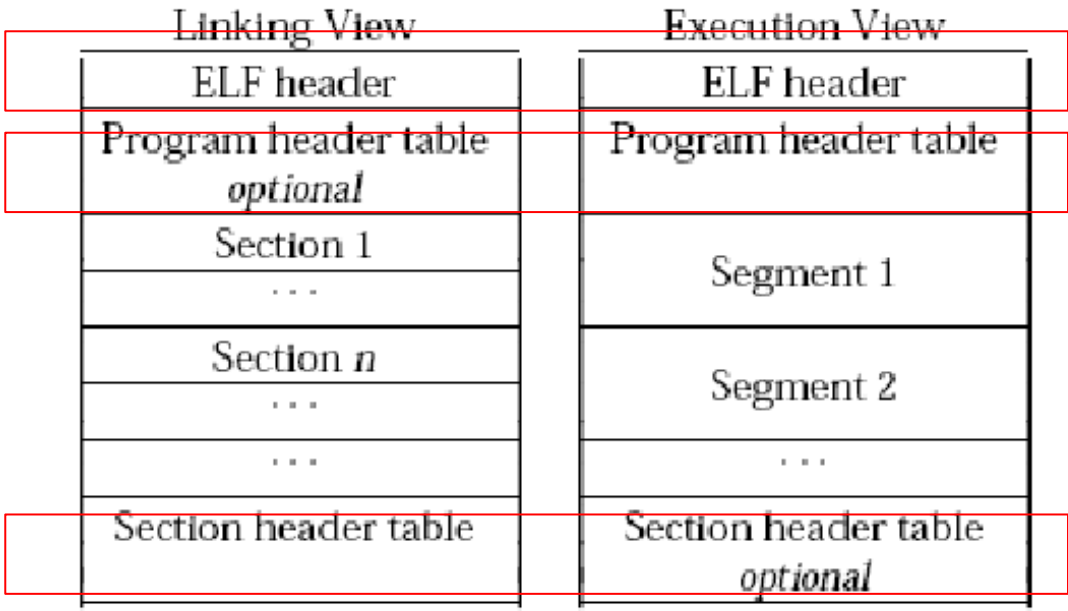
3.7 进程的可执行映像

- 左边是ELF的链接视图，可以理解为是目标代码文件的内容布局。右边是ELF的执行视图，可以理解为可执行文件的内容布局。
- 目标代码文件的内容是由section组成的，而可执行文件的内容是由segment组成的。目标代码文件中的section和section header table中的条目是一一对应的。section的信息用于链接器对代码重定位。
- 写汇编程序时，.text，.bss，.data这些关键字指的是section，比如.text，告诉汇编器后面的代码放入.text section中。



3.7 进程的可执行映像

- 而文件载入内存执行时，是以segment组织的，每个segment对应ELF文件中program header table中的一个条目，用来建立可执行文件的进程映像。
- 我们通常说的，**代码段、数据段是segment，目标代码中的section会被链接器组织到可执行文件的各个segment中**。.text section的内容会组装到代码段中，.data, .bss等节的内容会包含在数据段中。
- 在目标文件中，program header不是必须的，我们用gcc生成的目标文件也不包含program header。
- 解析ELF文件的工具readelf。比如对一个目标代码文件sleep.o执行**readelf -S sleep.o**



3.7 进程的可执行映像

ELF文件头

- ELF文件中的主要内容为**program header和section header**。可通过program header来获得每个segment的属性，通过section header获得每个section的属性。
- ELF header的定义可以在 /usr/include/elf.h 中找到。Elf32_Ehdr是32位 ELF header的结构体。Elf64_Ehdr是64位ELF header的结构体。（加载时内存中建立对应数据结构）
- 64位和32位只是个别字段长度不同，比如 Elf64_Addr 和 Elf64_Off 都是64位无符号整数。而Elf32_Addr 和 Elf32_Off是32位无符号整数。这导致ELF header的所占的字节数不同。32位的ELF header占52个字节，64位的ELF header占64个字节。

```
typedef struct
{
  unsigned char e_ident[EI_NIDENT];     /* Magic number和其它信息 */
  Elf32_Half    e_type;                 /* Object file type */
  Elf32_Half    e_machine;              /* Architecture */
  Elf32_Word    e_version;              /* Object file version */
  Elf32_Addr    e_entry;                /* Entry point virtual address */
  Elf32_Off     e_phoff;                /* Program header table file offset */
  Elf32_Off     e_shoff;                /* Section header table file offset */
  Elf32_Word    e_flags;                /* Processor-specific flags */
  Elf32_Half    e_ehsize;               /* ELF header size in bytes */
  Elf32_Half    e_phentsize;            /* Program header table entry size */
  Elf32_Half    e_phnum;                /* Program header table entry count */
  Elf32_Half    e_shentsize;            /* Section header table entry size */
  Elf32_Half    e_shnum;                /* Section header table entry count */
  Elf32_Half    e_shstrndx;             /* Section header string table index */
} Elf32_Ehdr;
```

```
typedef struct
{
  unsigned char e_ident[EI_NIDENT];     /* Ma
  Elf64_Half    e_type;                 /* Ob
  Elf64_Half    e_machine;              /* Ar
  Elf64_Word    e_version;              /* Ob
  Elf64_Addr    e_entry;                /* En
  Elf64_Off     e_phoff;                /* Pr
  Elf64_Off     e_shoff;                /* Se
  Elf64_Word    e_flags;                /* Pr
  Elf64_Half    e_ehsize;               /* EL
  Elf64_Half    e_phentsize;            /* Pr
  Elf64_Half    e_phnum;                /* Pr
  Elf64_Half    e_shentsize;            /* Se
  Elf64_Half    e_shnum;                /* Se
  Elf64_Half    e_shstrndx;             /* Se
} Elf64_Ehdr;
```

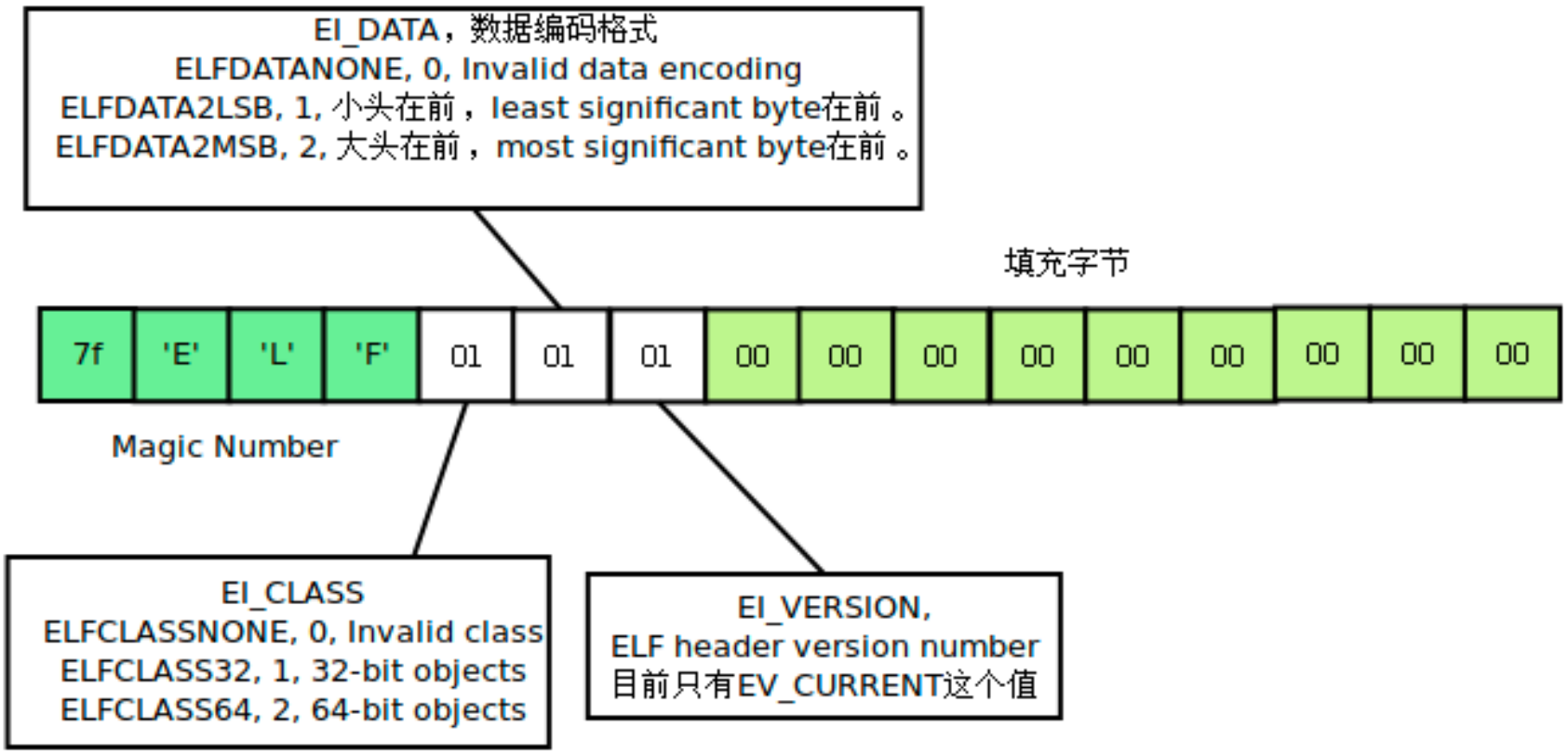
3.7 进程的可执行映像

成员	描述		可能取值		
e_ident	标志		见下表		
e_type	ELF 文件的类型		值	宏	描述
			1	ET_REL	目标文件
			2	ET_EXEC	可执行文件
			3	ET_DYN	动态库
			4	ET_CORE	核心转储文件
e_machine	硬件体系架构		值	宏	描述
			3	EM_386	Intel 80386
			40	EM_ARM	ARM
e_version	文件版本		值固定为 1。		
e_entry	入口点虚拟地址		根据实际情况而定。		
e_phoff	program header	表相对文件偏移量	根据实际情况而定。		
e_phentsize		大小	值固定为 32 个字节。		
e_phnum		数量	根据实际情况而定。		
e_shoff	section header	表相对文件偏移量	根据实际情况而定。		
e_shentsize		大小	值固定为 40 个字节。		
e_shnum		数量	根据实际情况而定。		
e_flags	硬件体系架构特定参数		对于 Intel 80386 来说其值为 0。		
e_ehsize	ELF 文件头的大小		值固定为 52 个字节。		
e_shstrndx	该项在 section header 表中的索引		根据实际情况而定。		

3.7 进程的可执行映像

- 前16个字节的e_ident：readelf -h <elffile>可以读取文件的ELF header信息
- 前4个字节是ELF的幻数Magic Number，固定为7f 45 4c 46。
- 第5个字节指明ELF文件是32位还是64位的。
- 第6个字节指明了数据的编码方式，即little endian或是big endian。little endian低位字节在前，即低位字节在低位地址，比如0x7f454c46，存储顺序就是46 4c 45 7f。big endian就是高位字节在前，即高位字节在低位地址，比如0x7f454c46，在文件中的存储顺序是7f 45 4c 46。
- 第7个字节指明了ELF header的版本号，目前值都是1。
- 第8-16个字节，为0。

3.7 进程的可执行映像


EI_DATA，数据编码格式
ELFDATANONE, 0, Invalid data encoding
ELFDATA2LSB, 1, 小头在前，least significant byte在前。
ELFDATA2MSB, 2, 大头在前，most significant byte在前。
填充字节
7f
'E'
'L'
'F'
01
01
01
00
00
00
00
00
00
00
00
00
Magic Number
EI_CLASS
ELFCLASSNONE, 0, Invalid class
ELFCLASS32, 1, 32-bit objects
ELFCLASS64, 2, 64-bit objects
EI_VERSION,
ELF header version number
目前只有EV_CURRENT这个值

3.7 进程的可执行映像

ELF文件中的program header

- program header用于描述segment的特性，目标文件（也就是文件名以.o结尾的文件）不存在program header，因为它不能运行。
- 一个segment包含一个或多个现有的section，相当于从程序执行的角度来看待这些section。program header使用以下的数据结构来定义：

```
1  typedef struct
2  {
3    Elf32_Word    p_type;         /* Segment type */
4    Elf32_Off     p_offset;       /* Segment file offset */
5    Elf32_Addr    p_vaddr;        /* Segment virtual address */
6    Elf32_Addr    p_paddr;        /* Segment physical address */
7    Elf32_Word    p_filesz;       /* Segment size in file */
8    Elf32_Word    p_memsz;        /* Segment size in memory */
9    Elf32_Word    p_flags;        /* Segment flags */
10   Elf32_Word    p_align;        /* Segment alignment */
11 } Elf32_Phdr;
```

3.7 进程的可执行映像

- p_offset表示该segment相对**ELF文件开头的偏移量**，
- p_filesz表示该segment在**ELF文件中的大小**，
- p_memsz表示该**segment加载到内存后所占用的大小**。
- p_filesz和p_memsz的大小只有在少数情况下不相同，如包含.bss section的segment，因为.bss section在ELF文件中不占用空间，但在内存中需要占用相应字节大小的空间。
- 通过p_offset和p_filesz两个成员就可以获得相应segment中的所有内容，所以这里就不再需要section header的支持，但需要ELF文件头中的信息来确定program header表（每个program header的大小相同）的开头位置，因此ELF文件头（它包含在第一个LOAD segment中）也要加载到内存中。

```
1  typedef struct
2  {
3    Elf32_Word    p_type;         /* Segment type */
4    Elf32_Off     p_offset;       /* Segment file offset */
5    Elf32_Addr    p_vaddr;        /* Segment virtual address */
6    Elf32_Addr    p_paddr;        /* Segment physical address */
7    Elf32_Word    p_filesz;       /* Segment size in file */
8    Elf32_Word    p_memsz;        /* Segment size in memory */
9    Elf32_Word    p_flags;        /* Segment flags */
10   Elf32_Word    p_align;        /* Segment alignment */
11 } Elf32_Phdr;
```

3.7 进程的可执行映像

- **3、p_vaddr表示segment的虚拟地址，p_paddr表示它的物理地址。在现代常见的体系架构中，很少直接使用物理地址，所以这里p_paddr的值与p_vaddr相同。嵌入式系统的编译器里依旧使用物理地址。**
- 4、p_align表示segment的对齐方式（也就是p_vaddr和p_offset对p_align取模的余数为0）。它的可能值为2的倍数，0和1表示不采用对齐方式。p_align的对齐方式不仅针对虚拟地址（p_vaddr），在ELF文件中的偏移量（p_offset）也要采用与虚拟地址相同的对齐方式。PT_LOAD类型的segment需要针对所在操作系统的页对齐。
- 5、p_flags表示segment的标志。PF_X表示segment可执行，PF_W表示可写，PF_R表示可读。它的可能取值如下所示：

```
1 #define PF_X          (1 << 0)    /* Segment is executable */
2 #define PF_W          (1 << 1)    /* Segment is writable */
3 #define PF_R          (1 << 2)    /* Segment is readable */
4 #define PF_MASKOS     0x0ff00000  /* OS-specific */
5 #define PF_MASKPROC   0xf0000000  /* Processor-specific */
```

3.7 进程的可执行映像

- 段头表（Section header Table）
  - 它描述了各个段的信息，比如段的段名、长度、在文件中的偏移、读写权限等等。段表是以一个元素是“ELF32_Shdr”结构体的数组。数组中的每一个元素对应一个段。

```
typedef struct
{
  Elf32_Word    sh_name;//段名存放在字符串表中，这里的指段名字符串在字符串表中的偏移
  Elf32_Word    sh_type;//段的类型
  Elf32_Word    sh_flags;//段的标志位
  Elf32_Addr    sh_addr;//如果该段最终被加载，表示加载后的虚拟地址，否则为0
  Elf32_Off     sh_offset;//表示该段在文件中的偏移
  Elf32_Word    sh_size;//段的长度
  Elf32_Word    sh_link;//段的链接信息
  Elf32_Word    sh_info;//段的链接信息
  Elf32_Word    sh_addralign;//段地址对齐
  Elf32_Word    sh_entsize;//对于有固定大小项的段，指的是项的长度
} Elf32_Shdr;
```

3.7 进程的可执行映像

ELF文件的加载

- 在Linux里首先在用户界面，bash(shell)进程会调用**fork()**系统调用创建一个新的进程，然后新的进程调用**execve()**系统调用执行指定的ELF文件，原先的bash进程继续返回等待刚才启动的新进程结束，然后继续等待用户输入命令。
- 在内核中，execve()系统调用相应的入口是sys_execve()，sys_execve()进行一些参数的检查复制之后，调用**do_execve()**。
- do_execve()检查被执行的文件的格式，调用search_binary_handle()查找合适的**可执行文件装载处理过程**。
- elf可执行文件的装载处理过程叫做**load_elf_binary()**，主要的工作是：

3.7 进程的可执行映像

- elf可执行文件的装载处理过程叫做load_elf_binary()，主要的工作：
  - (1) 检查ELF可执行文件格式的有效性，比如魔数，程序头表中段(segment)的数量；
  - (2) 查找动态链接的“.interp”段，设置动态连接器路径；
  - (3) 根据ELF可执行文件的程序头表的描述，对ELF文件进行映射，比如代码，数据，只读数据；
  - (4) 初始化ELF进程环境，在进程堆栈中保存命令行参数，环境变量，及辅助信息数组(Auxiliary Vector)，辅助信息数组为动态连接器提供可执行文件各个段的信息和程序的入口地址。

3.7 进程的可执行映像

- elf可执行文件的装载处理过程叫做load_elf_binary()，主要的工作：
  - (5) 将系统调用的返回地址改成ELF可执行文件的入口点，这个入口点取决于程序的链接方式。对于静态链接的ELF可执行文件，这个程序入口就是ELF文件的文件头中的e_entry所指的地址，对于动态链接的ELF可执行文件，程序的入口点是动态链接器，动态链接器会被加载。
  - (6) 当load_elf_binary()执行完毕，返回至do_execve()再返回至sys_execve()时，上面第5步中已经把系统调用的返回地址改成了被装载的ELF程序的入口地址了，所以当sys_execve()系统调用从内核态返回用户态时，EIP寄存器直接跳转到ELF程序的入口地址，于是新的程序开始执行，ELF可执行文件装载完成。

3.7 进程的可执行映像

动态链接的步骤

- (1) 动态链接器的自举：**动态链接器是一个特殊的共享对象。**首先，它是静态链接的，不依赖其他任何共享对象；其次，动态链接器本身所需要的静态和全局变量的重定位工作都由它本身完成。对于第二个条件，动态链接器在启动时，有一段非常精巧的代码可以完成这项艰巨的工作，同时又不用到静态和全局变量，这样一段启动代码叫做自举。动态链接器的入口即为自举代码的入口。
- **(2) 装载共享对象：**
  - 完成自举后，动态链接器将可执行文件和其**本身的符号表**都合并到**全局符号表**中，然后开始**寻找可执行文件所依赖的共享对象。可执行文件的.dynamic段中的DT_NEEDED入口下，记录了该可执行文件所依赖的共享对象。**
  - 动态链接器将所有依赖的**共享对象装载到内存，并将符号表合并到全局符号表中**，所以当所有的共享对象都被装载进来的时候，全局符号表中包含了进程中所有动态链接所需要的符号。

3.7 进程的可执行映像

- 动态链接的步骤
  - 重定位和初始化：链接器遍历可执行文件和共享对象的重定位表，将它们的**GOT/PLT中的每个需要重定位的位置进行修正**。因为动态链接器已经拥有了进程的全局符号表，所以这个修正过程比较容易。
  - 重定位完成之后，如果共享对象有.init段，**那么动态链接器会执行.init段中的代码**，用以实现共享对象特有的初始化过程。之后动态链接器将进程的控制权转交给程序入口并开始执行。

3.7 进程的可执行映像

关于.got段

- 由于**共享库的装载地址是不固定的**，为了保持代码段的地址无关性，**代码段中对静态和全局变量的访问都通过got段来周转**，当要对变量进行操作时，从got段相应入口下读取该变量的地址。**每个变量的地址对应got段的一个入口，共享库被装载前，got为全0，当动态链接器将该共享库装载进内存时，会将每个变量的绝对地址写入got段。**
- 对于共享库中定义的全局变量，链接时，连接器会在可执行文件的bss段中创建该变量的副本，当共享模块被装载时，**如果某个全局变量在可执行文件中拥有副本，那么动态链接器会把got中的地址指向该副本，这样该变量在运行时的实例只有一个。**
- 如果变量在共享模块中被初始化，那么动态链接器还要将该初始化的值复制到可执行文件的副本中，如果该全局变量在可执行文件中没有副本，那么got中相应地址就指向模块内部的该变量的副本。因为**共享库中的全局变量在每个进程中都拥有一个副本，所以多个进程对该全局变量的操作不会相互干扰。**

3.7 进程的可执行映像

- 关于.plt段
  - 动态链接相比静态链接，虽然节约了内存空间，但是**使用动态链接的程序却没有使用静态链接的程序速度快**，这是因为**动态链接下，全局变量，静态变量，函数调用都是通过got来间接跳转，运行速度必定会变慢**；
  - 其次**动态链接是运行时链接，即程序运行时，连接器会加载共享库，进行重定位工作，而在程序的运行过程中，很多函数根本不会用到，如果一开是就把所有的函数链接好显然是一种是一种浪费，所以便催生了延迟绑定技术(Lazy Binding)**，基本思想是，**函数第一次使用时才进行重定位，否则不为其重定位。**
  - plt(procedure linkage table)是实现延迟绑定技术的一段精巧指令序列，当函数第一次执行时，会调用_dl_runtime_resolve()函数来为函数符号进行重定位，**每个函数的地址在.got.plt下对应一个入口。当函数第二次执行时，只需从.got.plt相应的入口获取该函数的地址即可。**

3.7 进程的可执行映像

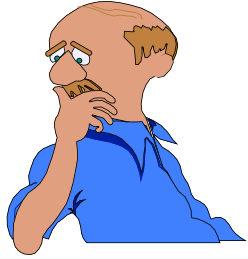
ELF中与动态链接相关的段

- 1 .interp： **动态链接器在操作系统中的存储位置**不是由系统配置决定，也不是由环境参数指定，而是**由 ELF 文件中的 .interp 段指定。**该段里保存的是一个字符串，这个字符串就是可执行文件所需要的动态链接器的位置，**常位于 /lib/ld-linux.so.2。**
- 2 .dynamic： 该段中保存了动态链接器所需要的基本信息，是一个结构数组，**可以看做动态链接下 ELF 文件的“文件头”。存储了动态链接会用到的各个表的位置等信息。**
- 3 .dynsym： 该段与 “.symtab”段类似，但**只保存了与动态链接相关的符号**，很多时候，ELF文件同时拥有 .symtab 与 .synsym段，其中 .symtab 将包含 .synsym 中的符号。该符号表中记录了动态链接符号在动态符号字符串表中的偏移，与.symtab中记录对应。

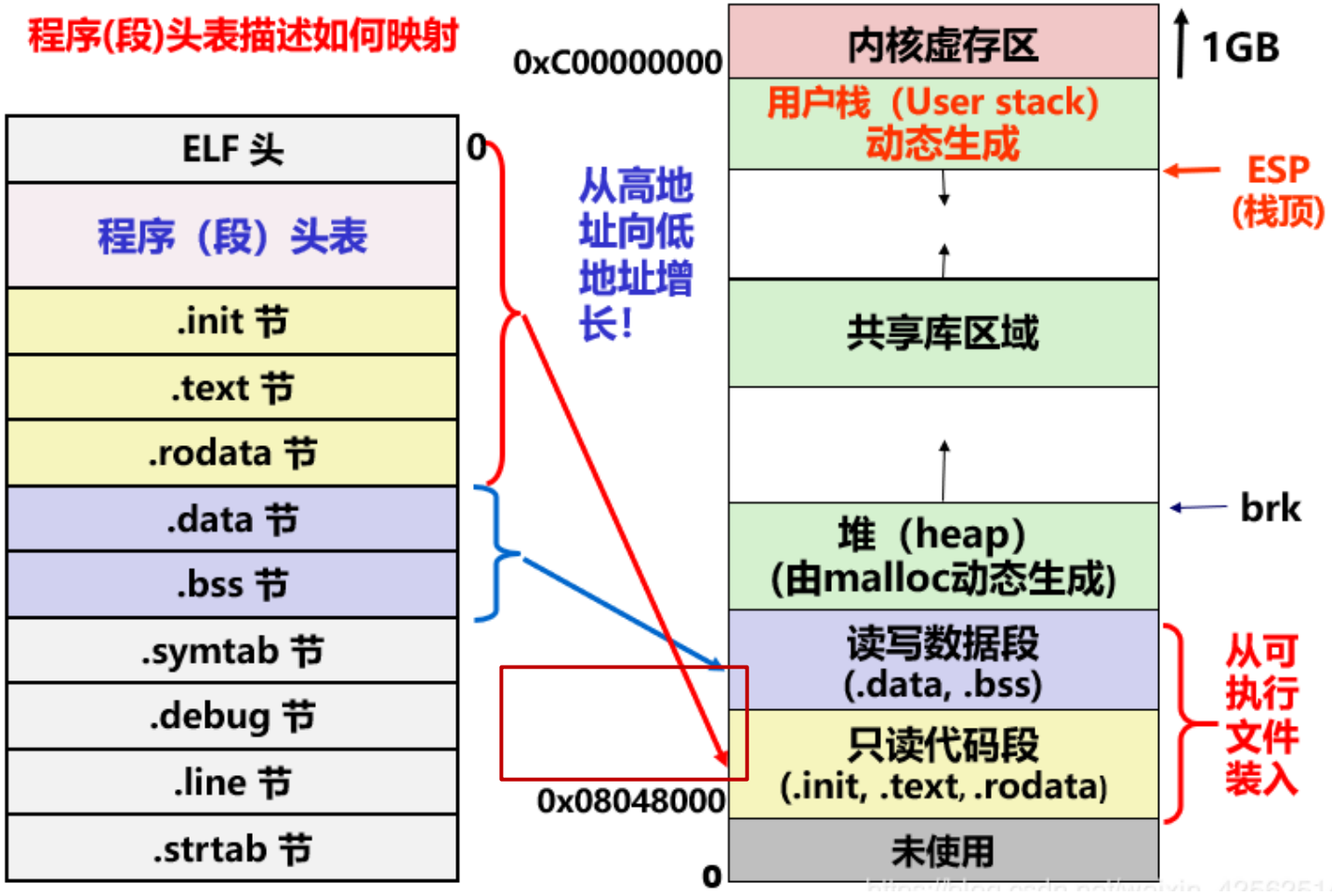
3.7 进程的可执行映像

- ELF中与动态链接相关的段
  - 4 .dynstr： 该段是 .dynsym 段的辅助段，.dynstr 与 .dynsym 的关系，类比与 .symtab 与 .strtab 的关系
  - 5 .hash： 在动态链接下，需要在程序运行时查找符号，为了加快符号查找过程，增加了辅助的符号哈希表，功能与 .dynstr 类似
  - 6 .rel.dyn： 对数据引用的修正，其所修正的位置位于 “.got”以及数据段（类似重定位段 "rel.data"）
  - 7 .rel.plt： 对函数引用的修正，其所修正的位置位于 “.got.plt”

3.7 进程的可执行映像

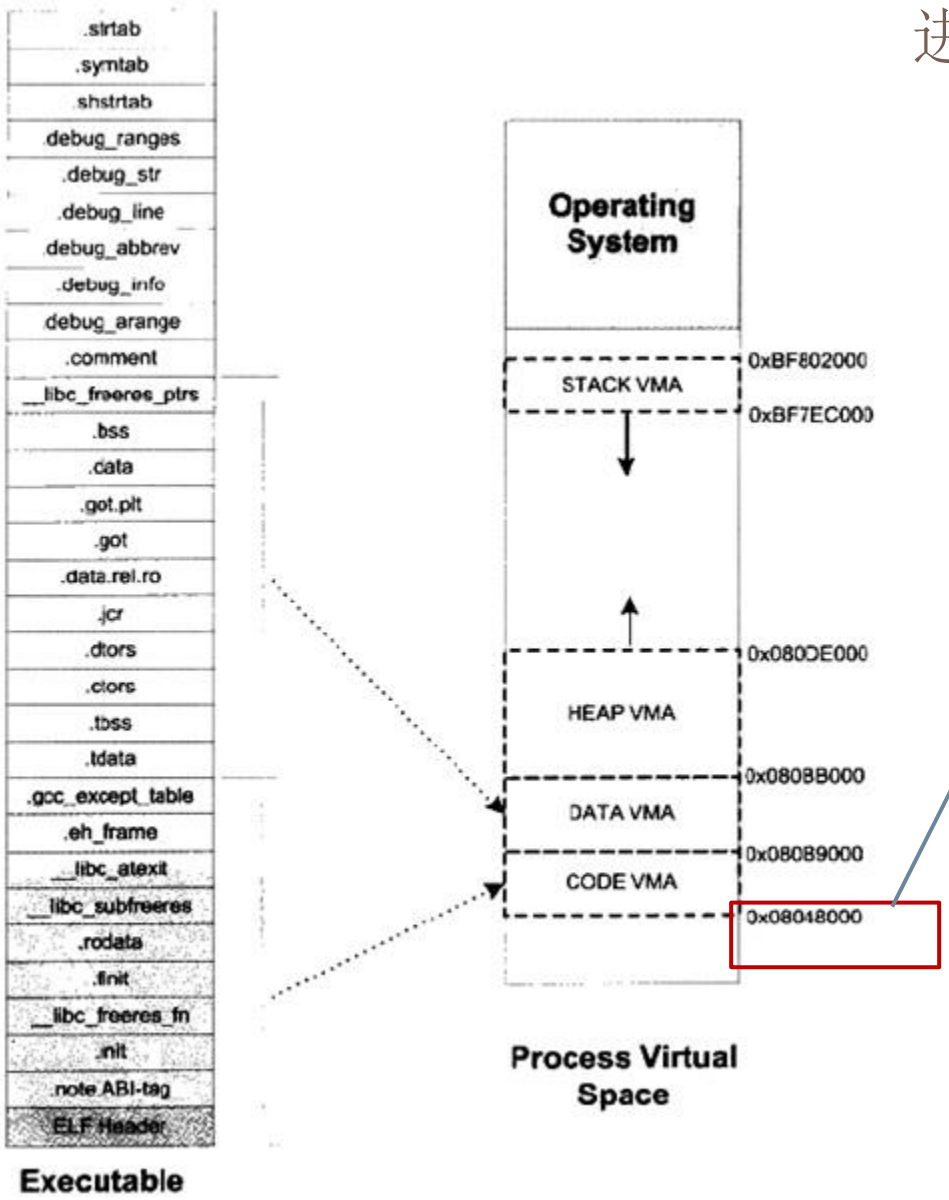


- 程序被加载到内存后的布局是什么？进程的内存映像



3.7 进程的可执行映像

进程的内存映像



所有应用的虚拟地址空间都是一样的?

3.7 进程的可执行映像

在linux中使用命令：ld –verbose

进程地址0x0804 8000产生？

GNU ld version 2.19.51.0.14-34.fc12

Supported emulations:

elf_i386

i386linux

elf_x86_64

using internal linker script:

==========

/* Script for -z combreloc: combine and sort reloc sections */

OUTPUT_FORMAT("elf32-i386", "elf32-i386",

"elf32-i386")

OUTPUT_ARCH(i386)

ENTRY(_start)

SEARCH_DIR("/usr/i686-redhat-linux/lib"); SEARCH_DIR("/usr/local/lib"); SEARCH_DIR("/lib"); SEARCH_DIR("/usr/lib");

SECTIONS

{

/* Read-only sections, merged into text segment: */

PROVIDE (__executable_start = SEGMENT_START("text-segment", 0x08048000)); . = SEGMENT_START("text-segment",

0x08048000) + SIZEOF_HEADERS;

.....

后面略

这个地址是虚拟地址，不是物理地址！！

3.7 进程的可执行映像

- 是不是所有的操作系统都如此？
- 上述过程需要MMU的支持（虚拟存储）；
- 在没有虚存的情况，操作系统与应用一般都统一编译成一个可执行映像，因此对应用程序*没有加载过程*。即编译时的虚地址与运行时物理地址要相同。

- 在虚存管理一章深入学习！！

进程总结

计算机解决问题 ⇒ 执行程序

执行中的程序和静止程序存在很大区别 ⇒ 引出进程

CPU太快 ⇒ 引出并发 ⇒ 进一步深化了进程

进程走走停停 ⇒ 状态转化 ⇒ 现场切换 ⇒ 进程调度

进程相互干扰 ⇒ 进程保护 ⇒ 地址空间 ⇒ 线程

进程相互协作 ⇒ 进程间消息通信⇒ 进程同步、死锁

用户希望操纵进程 ⇒ 进程创建等

进程可执行映像有特定的格式，与内存布局有直接相关