第3章 进程

目录

3.1 进程的引入

- 为了描述并发程序执行时的特征，引入了进程。

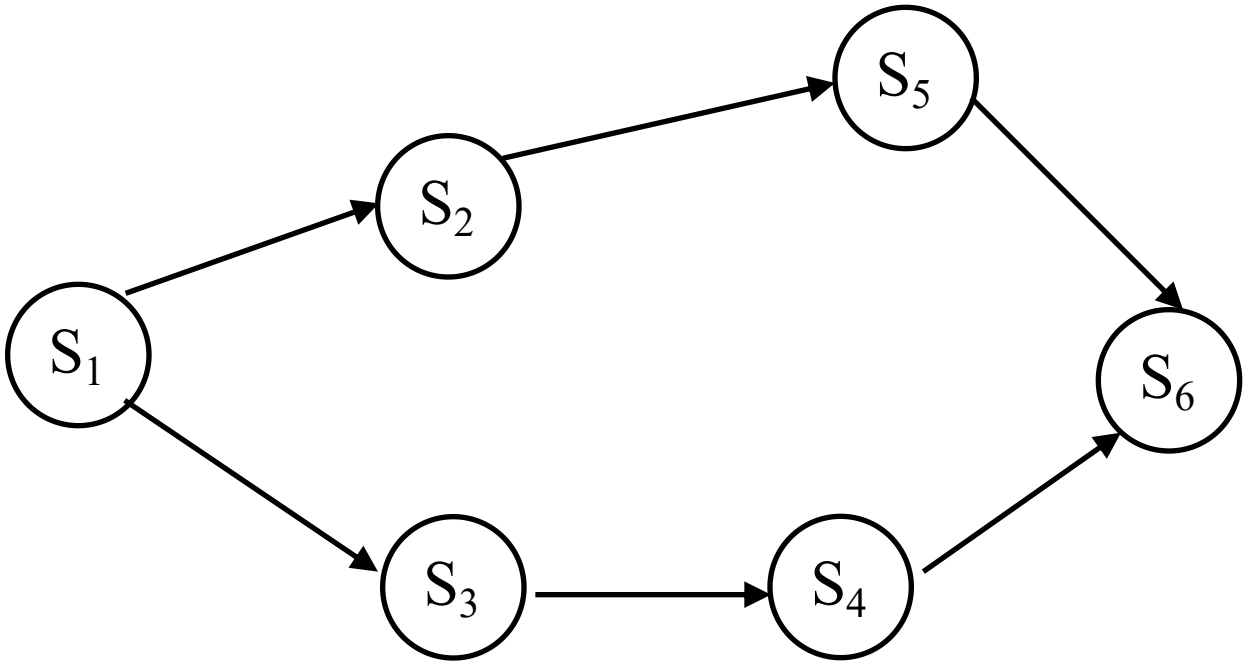
1、 前趋图

- 前趋图是一个有向无循环图，用于描述程序、程序段或语句执行的先后次序。
- 图中的每个结点可以表示一条语句、一个程序段或一个进程，结点间的有向边表示两个结点之间存在的前趋关系“→”：
- $\rightarrow=\{(P_i, P_j) \mid P_i$必须在$P_j$开始执行之前完成$\}$

前趋图中的各类结点

- 如果$(P_i, P_j) \in \rightarrow$，可以写成$P_i \rightarrow P_j$，则称$P_i$是$P_j$的直接前趋，$P_j$是$P_i$的直接后继。
- 若存在一个序列$P_i \rightarrow P_j \rightarrow \ldots \rightarrow P_k$，则称$P_i$是$P_k$的前趋。在前趋图中，没有前趋的结点称为初始结点，没有后继的结点称为终止结点。

前趋图例子



2、 程序的顺序执行

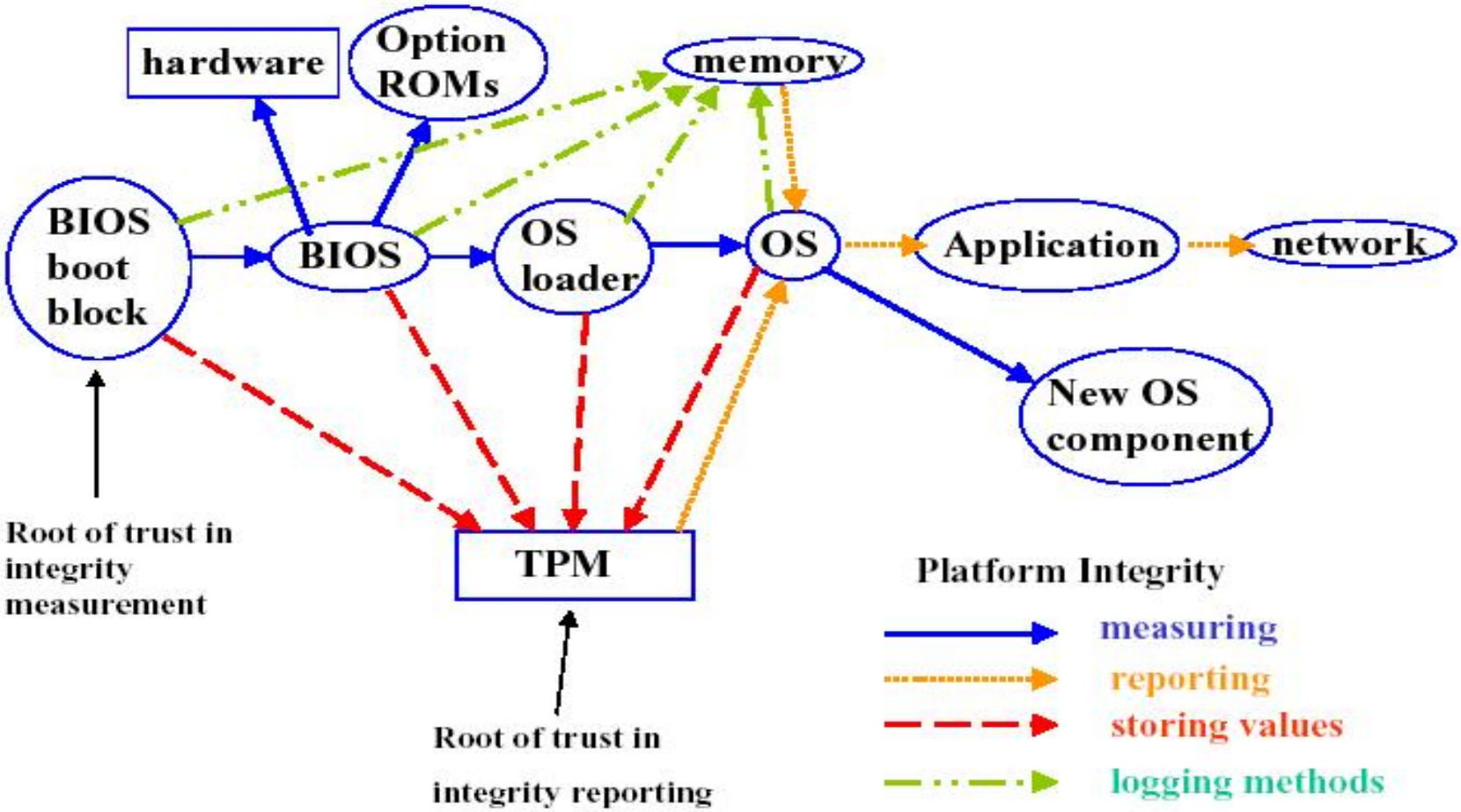
- 一个程序通常由若干个程序段所组成，它们必须按照某种先后次序来执行，仅当前一个操作执行完后才能执行后继操作，这类计算过程就是程序的顺序执行过程。
- 例如：先输入→再计算→最后打印输出，即：I→C →P。

程序顺序执行时的特征

1. 顺序性：处理机的操作严格按照程序所规定的顺序执行，即每一个操作必须在下一个操作开始之前结束。
2. 封闭性：程序一旦开始运行，其执行结果不受外界因素影响。
3. 可再现性：只要程序执行时的初始条件和执行环境相同，当程序重复执行时，都将获得相同的结果。

可信计算的启动信任链

可信PC信任链体系图

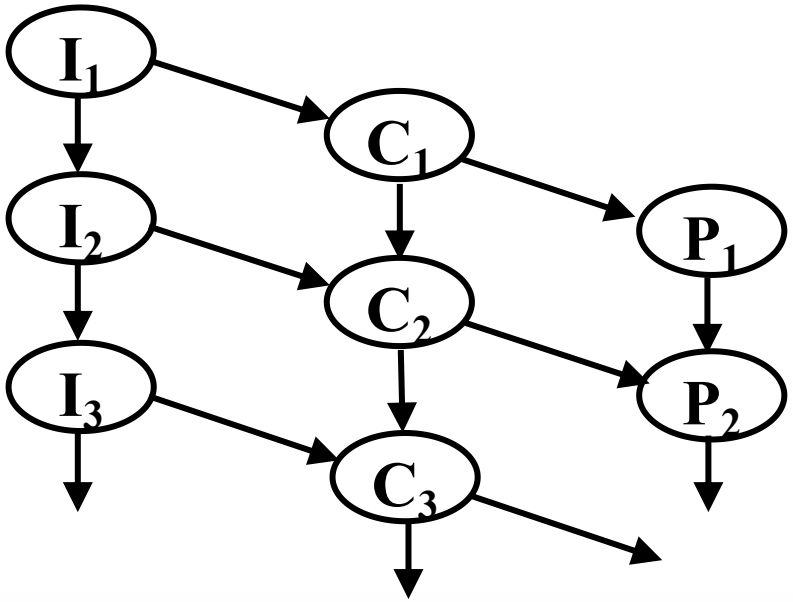


3、 程序的并发执行

- 程序的并发执行是指若干个程序（或程序段）同时在系统中运行，这些程序（或程序段）的执行在时间上是重叠的，一个程序（或程序段）的执行尚未结束，另一个程序（或程序段）的执行已经开始。

程序并发执行例子

- 进程1、2、3并发执行。对每个进程而言，其输入、计算和输出这三个操作必须顺序执行。它们之间存在如下先后关系：
  - **I_1先于C_1和I_2， C_1先于P_1和C_2， P_1先于P_2**
  - **I_2和C_1， I_3、 C_2和P_1可以并发。**



程序并发执行时的特征

1. 间断性：并发程序具有“执行---暂停----执行”这种间断性的活动规律。
2. 失去封闭性：多个程序共享系统中的资源，这些资源的状态将由多个程序来改变，致使程序之间相互影响。
3. 不可再现性：在初始条件相同的情况下，程序的执行结果依赖于执行的次序。

不可再现性的例子

- 程序并发执行时可能出现与时间有关的错误。
- 例
  - 进程1：r1=x;　　进程2：r2=x;
  - r1++;　　r2++;
  - x=r1;　　x=r2;
- 设在两进程运行之前，x的值为0。则两进程运行结束后，x值可为：

1　　2

4、程序并发执行的条件

如何解决时间有关的错误？

- 几个定义：
  - 读集：语句执行期间要引用的变量集合，记为
    $R(S_i)=\{a_1, \ldots, a_m\}$
  - 写集：语句执行期间要改变的变量集合，记为
    $W(S_i)=\{b_1, \ldots, b_n\}$

Bernstein条件

- Bernstein条件能保证两个程序段S_i和S_j并发执行而不会产生与时间有关的错误：
  ① $R(S_i) \cap W(S_j) = \{\}$
  ② $R(S_j) \cap W(S_i) = \{\}$
  ③ $W(S_j) \cap W(S_i) = \{\}$
  - 条件1与条件2，保证了两个程序段之间不会引起读数据的干扰，两次读取数据之间，存储器数据不会发生改变
  - 条件3保证了两个程序段之间不会造成写数据的干扰，写数据结果不会丢失
- 满足以三个条件即可满足并发执行的程序可满足封闭性和可再现性

例

- 考虑下面是条语句：

  S1：a=x+y　　S2：b=z+1

  S3：c=a-b　　S4：d=c+1

- R(S1)={x,y}　R(S2)={z}　R(S3)={a,b}

  W(S1)={a}　W(S2)={b}　W(S3)={c}

- 因$R(S1)\cap W(S2)\cup R(S2)\cap W(S1)\cup W(S1)\cap W(S2)=\{\}$，故S1和S2可以并发执行 。

- 因$R(S2)\cap W(S3)\cup R(S3)\cap W(S2)\cup W(S3)\cap W(S2)=\{b\}$，故S2和S3不能并发执行 。同理，S1和S3之间也不可以并发执行。

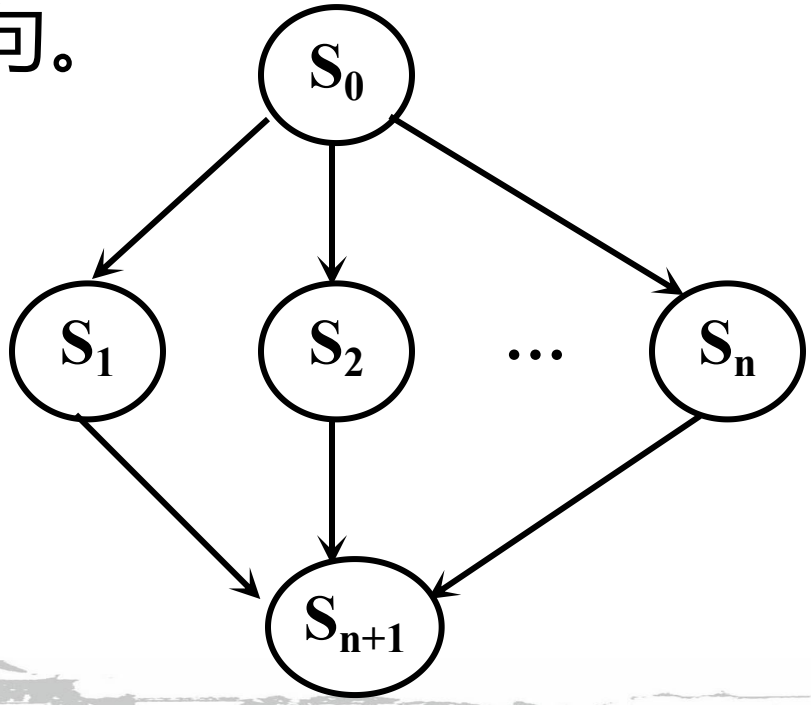
并发语句的描述方式

- cobegin

  S_1; S_2; …S_n;

  coend

- 对应的前趋图如右，其中S_0和S_{n+1}分别是cobegin和coend语句前后的两条语句。



5、进程的引入

- 操作系统为什么要引入进程概念?
  - 原因1-刻画系统的动态性，发挥系统的并发性，提高资源利用率。
  - 原因2-它能解决系统的“共享性”，正确描述程序的执行状态。

3.2 进程定义与描述

- 进程最初由20世纪60年代提出，由MULTICS和IBM TSS/360首先引入
- 进程有多种定义，下面列举一些有代表性的定义：
  ① 进程是程序在处理器上的一次执行过程。
  ② 进程是可以和别的计算并行执行的计算。
  ③ 进程是程序在一个数据集合上运行的过程，是系统进行资源分配和调度的一个独立单位。
  ④ 进程是一个具有一定功能的程序关于某个数据集合的一次运行活动。（1978年全国操作系统学术会议）。

进程

- 操作系统引入进程的概念
  - 从理论角度看，是对正在运行的程序过程的抽象；
  - 从实现角度看，是一种数据结构，目的在于清晰地刻划动态系统的内在规律，有效管理和调度进入计算机系统主存储器运行的程序。

进程的特征

① **动态性**：进程是程序的一次执行过程。进程是有“生命”的，动态性还表现为它因创建而产生，因调度而执行，因无资源而暂停，因撤消而消亡。而程序是静态实体。

② **并发性**：多个进程实体同时存在于内存中，能在一段时间内同时运行。

③ **独立性**：在传统OS中，进程是独立运行的基本单位，也是系统分配资源和调度的基本单位。凡是未建立进程的程序，很难作为独立单位参与调度和运行。

④ **异步性**：也叫**制约性**，进程以各自独立的不可预知的速度向前推进。

⑤ **结构性**：进程实体由程序段、数据段及进程控制块组成，又称为进程映像。

进程与程序的关系

① 进程是动态概念，程序是静态概念；进程是程序在处理机上的一次执行过程，而程序是指令的集合。

② 进程是暂时的，程序是永久的。进程是一个状态变化的过程；程序可以长久保存。

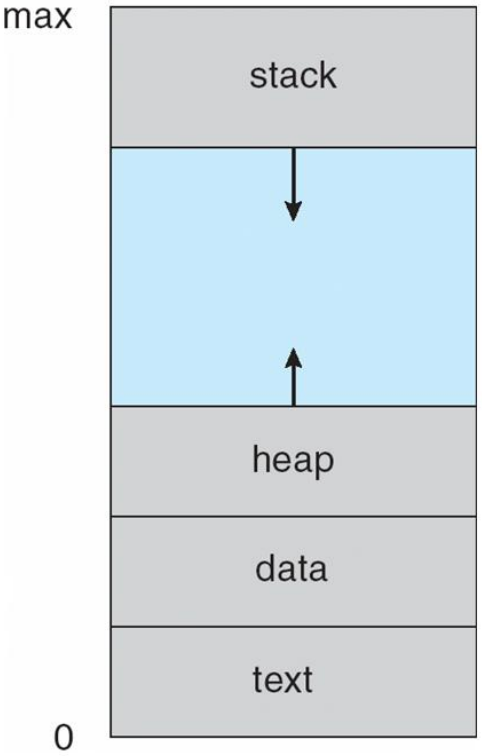
③ 进程与程序的组成不同。进程的组成包括代码段、数据段和进程控制块。

④ 进程与程序是密切相关的。一个程序可以对应多个进程；一个进程可以包括多个程序，进程和程序不是一一对应的。

⑤ 进程可以创建新进程，而程序不能形成新程序。

进程的描述

- 进程映象：
  - 某时刻进程的内容及其状态的集合
- 组成
  - 进程控制块：每个进程捆绑一个控制块，标识信息、现场信息、控制信息
  - 内存中的进程(虚拟地址空间)：
    - 代码段：被执行的程序
    - 数据段：全局变量
    - 栈：保存临时数据，如参数、返回地址
    - 堆：动态分配的内存
  - 核心栈：每个进程捆绑一个核心栈，用于态工作时的现场保护



进程的上下文

- **进程的上下文**
  - 操作系统中把**进程物理实体和支持进程运行的环境**合称为进程上下文。
  - 当系统调度新进程占有处理器时，新老进程随之发生**上下文切换**。进程的运行被认为是在上下文中执行。

- **进程上下文组成**
  - 用户级上下文：
    - 正文(程序)、数据和共享存储区、用户栈组成，占用进程的虚拟地址空间。
  - 寄存器级上下文
    - 程序状态寄存器、指令计数器、栈指针，控制寄存器、通用寄存器等组成
  - 系统级上下文
    - 进程控制块、主存管理信息、内核栈等组成

进程控制块(PCB)

- PCB是描述和管理进程的数据结构。它是进程实体的一部分
  - 操作系统创建PCB
  - 操作系统通过PCB感知进程的存在，PCB是进程存在的唯一标志
  - 操作系统通过PCB了解进程状态：执行情况、进程让出处理器后所处的状态、断点信息等
  - 操作系统通过PCB来调度、控制和管理进程
- 操作系统会执行如下操作：
  - 创建PCB
  - 修改PCB
  - 访问PCB
  - 回收PCB

PCB的组成

PCB一般包括三类信息：

1. 进程标识信息
   - ① 进程标识符：惟一标识进程的一个标识符或整数。
   - ② 家族关系：指明本进程与家族的关系，如父子进程标识。
   - ③ 用户标识符：表明进程的所有者
2. 进程现场信息
   - ① 保留进程在运行时存放在处理器中的各种信息
   - ② 包括：程序计数器、通用寄存器、程序状态字、栈指针等

PCB的组成

3. 进程控制信息

① 进程当前状态：说明进程当前所处状态。

② 进程队列指针：用于记录PCB队列中下一个PCB的地址。

③ 进程优先级：反映进程获得CPU的优先级别。

④ 通信信息：进程与其他进程所发生的信息交换。

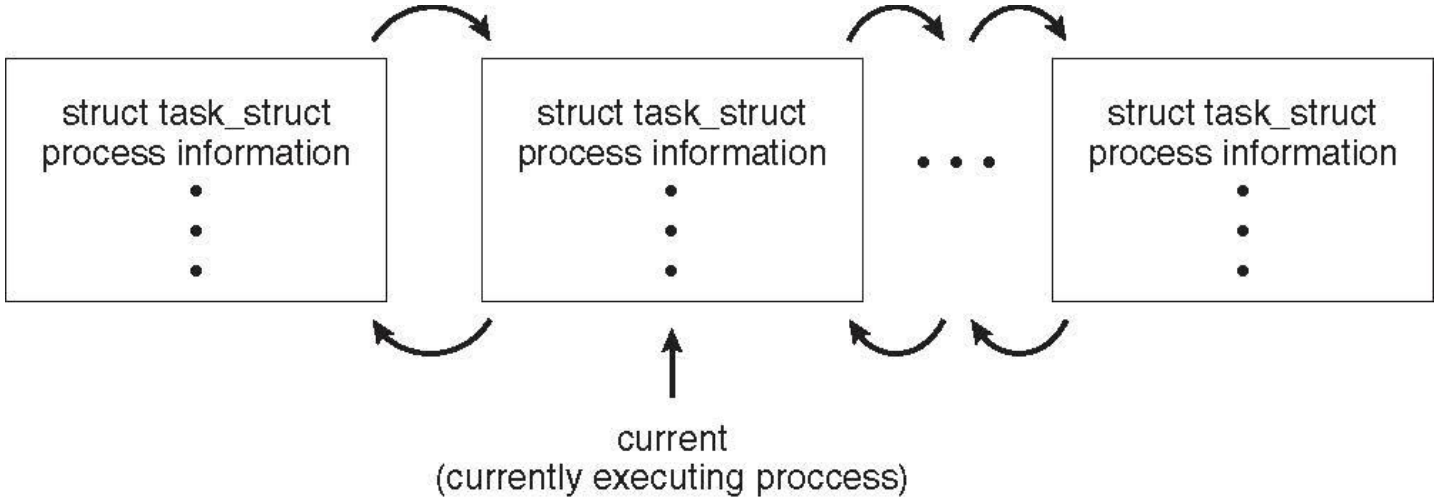
⑤ 程序和数据地址：定位进程的程序和数据在内存或外存中的存放地址。

⑥ 资源清单：列出进程所需资源及当前已分配资源，如打开的文件，使用CPU记录

Linux的进程表示

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 process */
```



3.3 进程的状态及转换

- 为了刻画进程的动态特征，可以将进程的生命期划分为一组状态，用这些状态来描述进程的活动过程。
  - 进程因创建而产生，因撤销而消亡
  - 进程可占用处理器执行
  - 进程可运行而无法分配处理机
  - 进程因等待事件而无法进入空闲处理机
  - 进程的状态可切换

1、 进程的三种基本状态

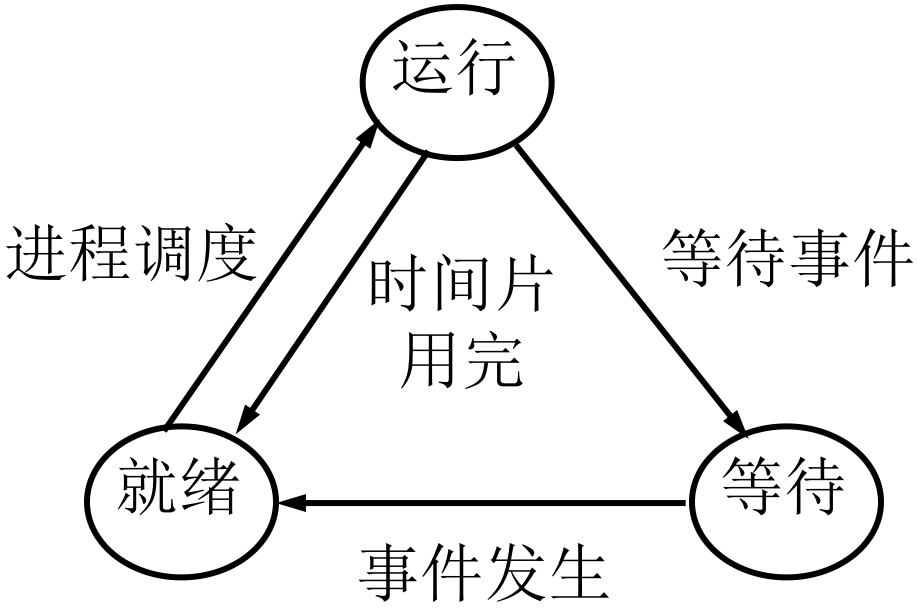
- 三态模型：一个进程至少应有以下三种基本状态：
  - 就绪状态
  - 执行状态
  - 阻塞状态

1、 进程的三种基本状态

- 就绪状态(Ready)： 进程已获得除处理机以外的所有资源，一旦分配了处理机就可以立即执行。
- 执行状态(Running)： 又称运行状态。一个进程获得必要的资源并正在处理机上执行。
- 阻塞状态(Blocked)： 又称等待状态(wait)、睡眠状态(sleep)。正在执行的进程，由于发生某事件而暂时无法执行下去（如等待输入/输出完成）。这时即使把处理机分配给该进程，它也无法运行。

进程状态转换图

- 思考：1个处理器，N个进程，处于运行态的进程个数可以是多少？处于就绪态和等待态的进程可以是多少？



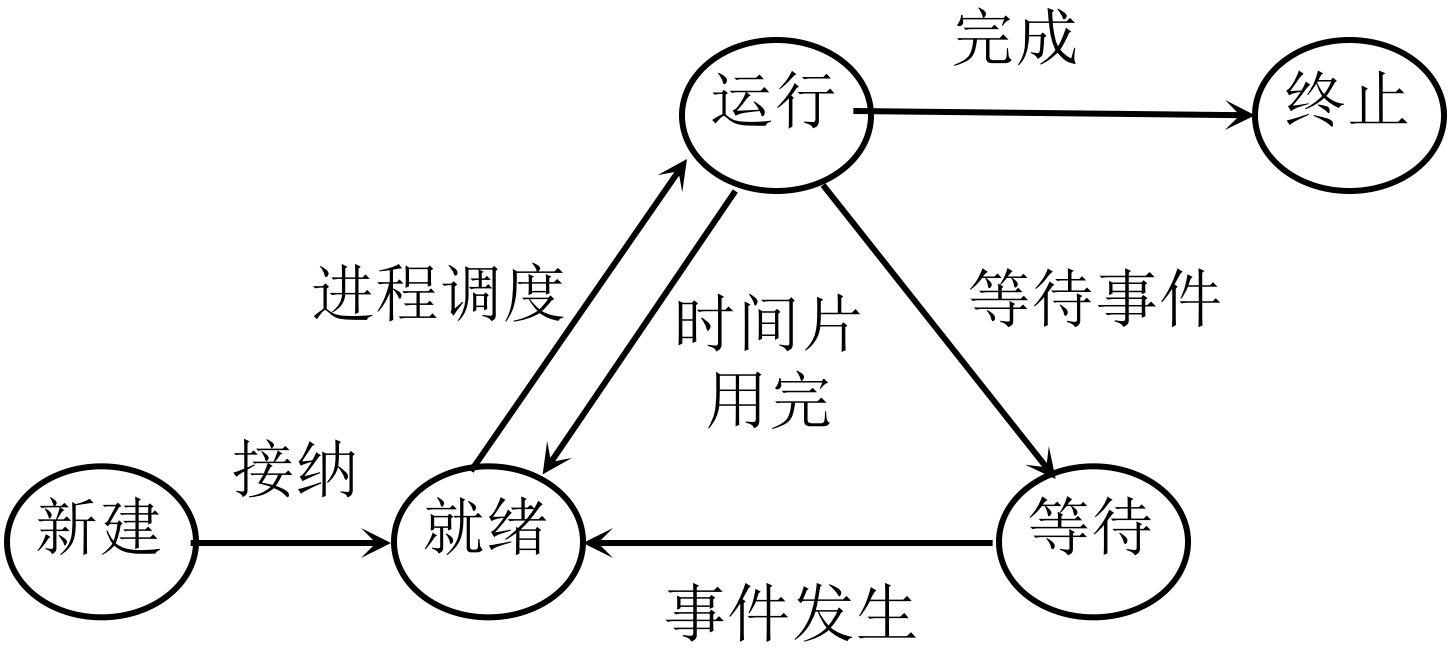
进程转换的原因

- 运行态→等待态：等待使用资源；如等待外设传输；等待人工干预。
- 等待态→就绪态：资源得到满足；如外设传输结束；人工干预完成。
- 运行态→就绪态：运行时间片到；出现有更高优先权进程。
- 就绪态→运行态：CPU 空闲时选择一个就绪进程。

2、 五态模型

- 在许多系统中又增加了两种状态：
  - 新建状态（new）：进程刚刚建立，但还未进入就绪队列。又称创建状态。
  - 终止状态（exit）：当一个进程正常或异常结束，操作系统已释放它所占用的资源，但尚未将它撤消时的状态，又称退出状态。

五态模型的进程状态转换图



进程转换的原因

- NULL→新建态：执行一个程序，创建一个子进程。
- 新建态→就绪态：当操作系统完成了进程创建的必要操作，并且当前系统的性能和虚拟内存的容量均允许。
- 运行态→终止态：当一个进程到达了自然结束点，或是出现了无法克服的错误，或是被操作系统所终结，或是被其他有终止权的进程所终结。
- 终止态→NULL：完成善后操作。
- 就绪态→终止态：未在状态转换图中显示，但某些操作系统允许父进程终结子进程。
- 等待态→终止态：未在状态转换图中显示，但某些操作系统允许父进程终结子进程。

状态转换的有关说明

- 大多数状态**不可逆转**，如等待不能转换为运行。
- 状态转换大多为被动进行，但运行→等待是主动的。
- 一个进程在一个时刻只能处于上述状态之一。

3、 进程的挂起状态

- 由于进程的不断创建，系统资源已不能满足进程运行的要求，就必须人为将某些进程切换到静止状态，因此需要把某些进程挂起（suspend），对换到磁盘镜像区中，暂时不参与进程调度，起到平滑系统操作负荷的目的。

进程挂起的原因

- 进程挂起的原因主要有以下6个方面：
  - ① 进程竞争资源，导致系统资源不足，负荷过重，需要挂起部分进程以调整系统负荷,保证系统的实时性或让系统正常运行。
  - ② 定期执行的进程（如审计、监控、记账程序）对换出去，以减轻系统负荷。
  - ③ 系统中的进程均处于等待状态，需要把一些阻塞进程对换出去，腾出足够内存装入就绪进程运行。

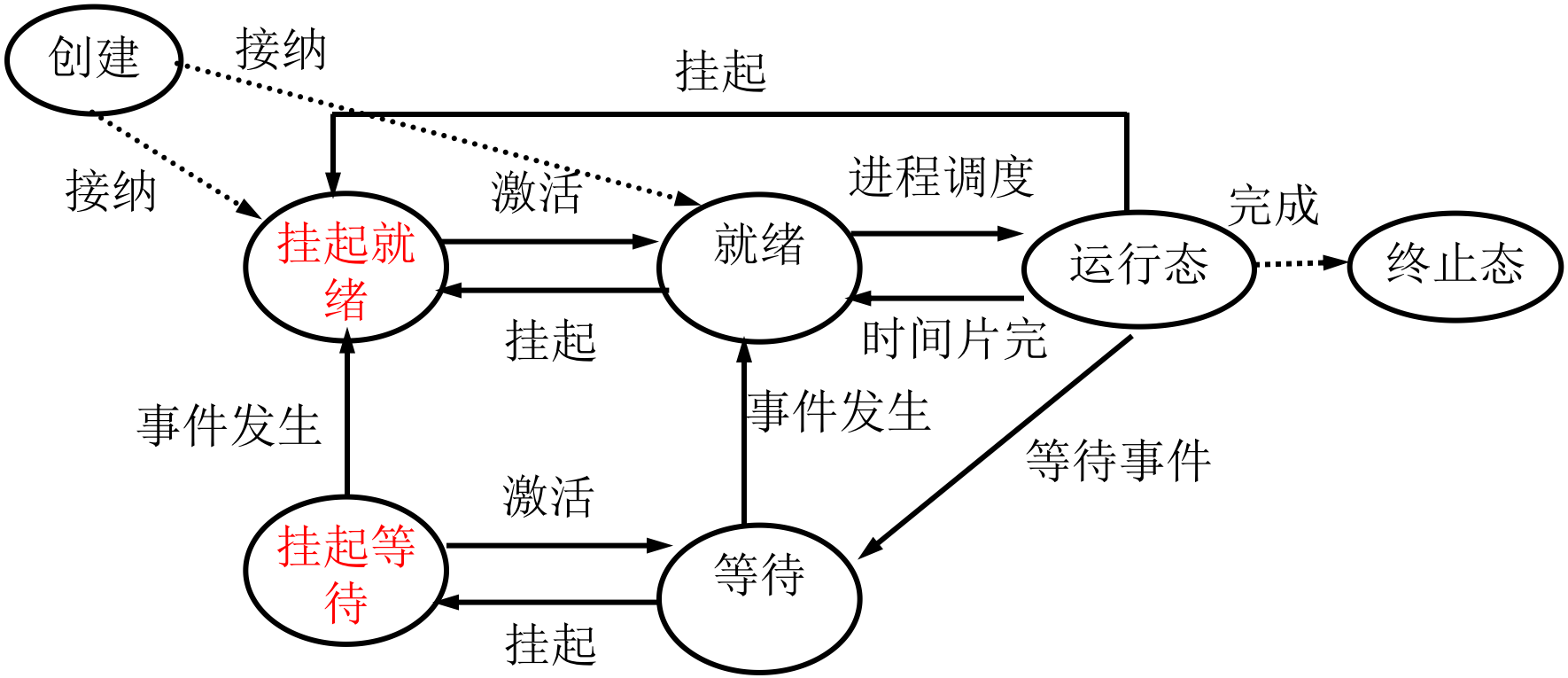
进程挂起的原因 （cont.）

- 进程挂起的原因有：
  - ④ 用户要求挂起自己的进程，以便进行某些调试、检查和改正。
  - ⑤ 父进程要求挂起后代进程，以进行某些检查和改正。
  - ⑥ 操作系统需要挂起某些进程，检查运行中资源使用情况，以改善系统性能;或当系统出现故障或某些功能受到破坏时，需要挂起某些进程以排除故障。

进程的挂起状态

- 基于上述原因，需引入两个新的状态：
  - 挂起就绪态（ready,suspend）：进程具备运行条件但目前在外存中，只有当它被对换到主存才能被调度执行
  - 挂起等待态（blocked,suspend）：进程正在等待某一个事件且在外存中。

七态模型的进程状态转换图



进程转换的原因

- 等待态→挂起等待态：操作系统根据当前资源状况和性能要求，可以决定把等待态进程对换出去成为挂起等待态。
- 挂起等待态→挂起就绪态：引起进程等待的事件发生之后，相应的挂起等待态进程将转换为挂起就绪态。
- 就绪态→挂起就绪态：操作系统根据当前资源状况和性能要求，也可以决定把就绪态进程对换出去成为挂起就绪态。
- 挂起就绪态→就绪态：当内存中没有就绪态进程，或者挂起就绪态进程具有比就绪态进程更高的优先级，系统将把挂起就绪态进程转换成就绪态。

进程转换的原因

- 挂起等待态→等待态：当一个进程等待一个事件时，原则上不需要把它调入内存。但是在下面一种情况下，这一状态变化是可能的。当一个进程退出后，主存已经有了一大块自由空间，而某个挂起等待态进程具有较高的优先级并且操作系统已经得知导致它阻塞的事件即将结束，此时便发生了这一状态变化。
- 运行态→挂起就绪态：运行态进程被抢占 CPU，，而此时主存空间不够，从而可能导致运行态进程转化为挂起就绪态。另外处于运行态的进程也可以自己挂起自己。
- 新建态→挂起就绪态：考虑到系统当前资源状况和性能要求，可以决定新建的进程将被对换出去成为挂起就绪态。

挂起进程的特征

- 可以把一个挂起进程等同于不在主存的进程，因此挂起的进程将不参与进程调度直到它们被对换进主存。
- 一个挂起进程具有如下特征：
  - 1. 该进程不能立即被执行。
  - 2. 挂起进程可能会等待一个事件，但所等待的事件是独立于挂起条件的，事件结束并不能导致进程具备执行条件。
  - 3. 进程进入挂起状态是由于操作系统、父进程或进程本身阻止它的运行。
  - 4. 结束进程挂起状态的命令只能通过操作系统或父进程发出。

3.4 进程的组织

- 系统中有许多进程，为了能对它们进行有效的管理，应将PCB组织起来。
- 常用的组织方式有：
  - 线性方式
  - 链表方式
  - 索引方式

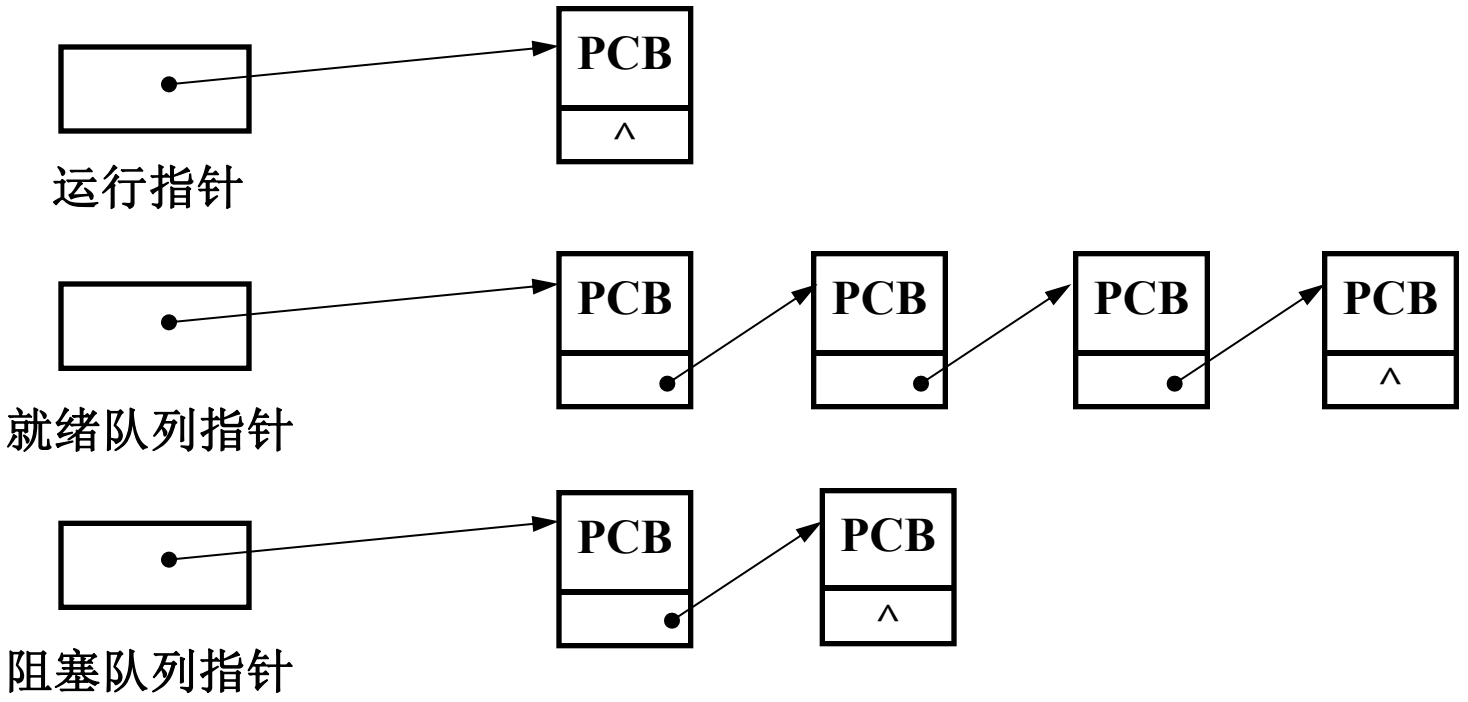
线性方式

- 线性方式：将PCB顺序存放在一片连续内存中。

PCB_1	PCB_2	PCB_3	…	PCB_{n-2}	PCB_{n-1}	PCB_n

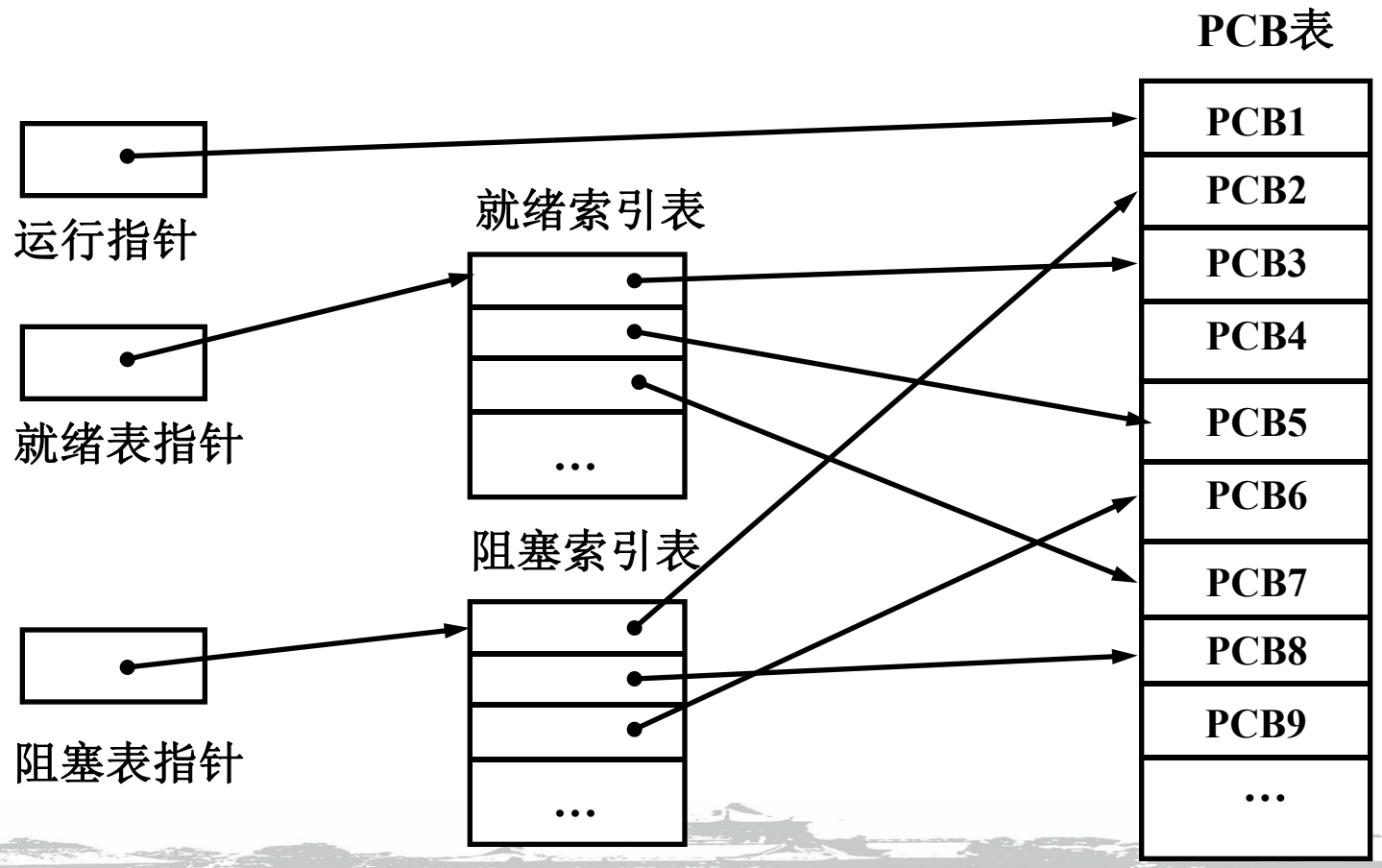
链接方式

- 链接方式：将同一状态的PCB组成一个链表。



索引方式

- 索引方式：将同一状态的进程归入一个索引表，再由索引指向相应的PCB



3.5 进程控制和管理

- 进程控制的职能是对系统中的所有进程实施有效的管理。
- 常见的进程控制功能有进程创建、撤消、阻塞与唤醒等。
- 这些功能一般由操作系统内核原语来实现。

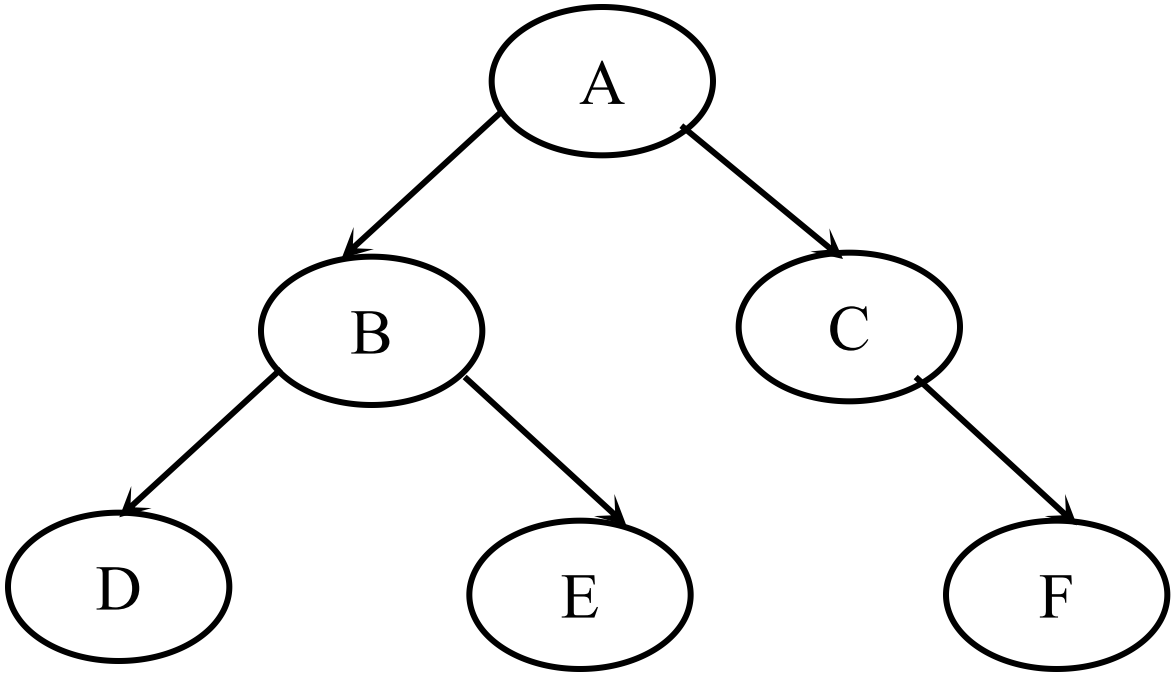
原语

- 原语是由若干条机器指令构成的，用以完成特定功能的一段程序，这段程序在执行期间不可分割。

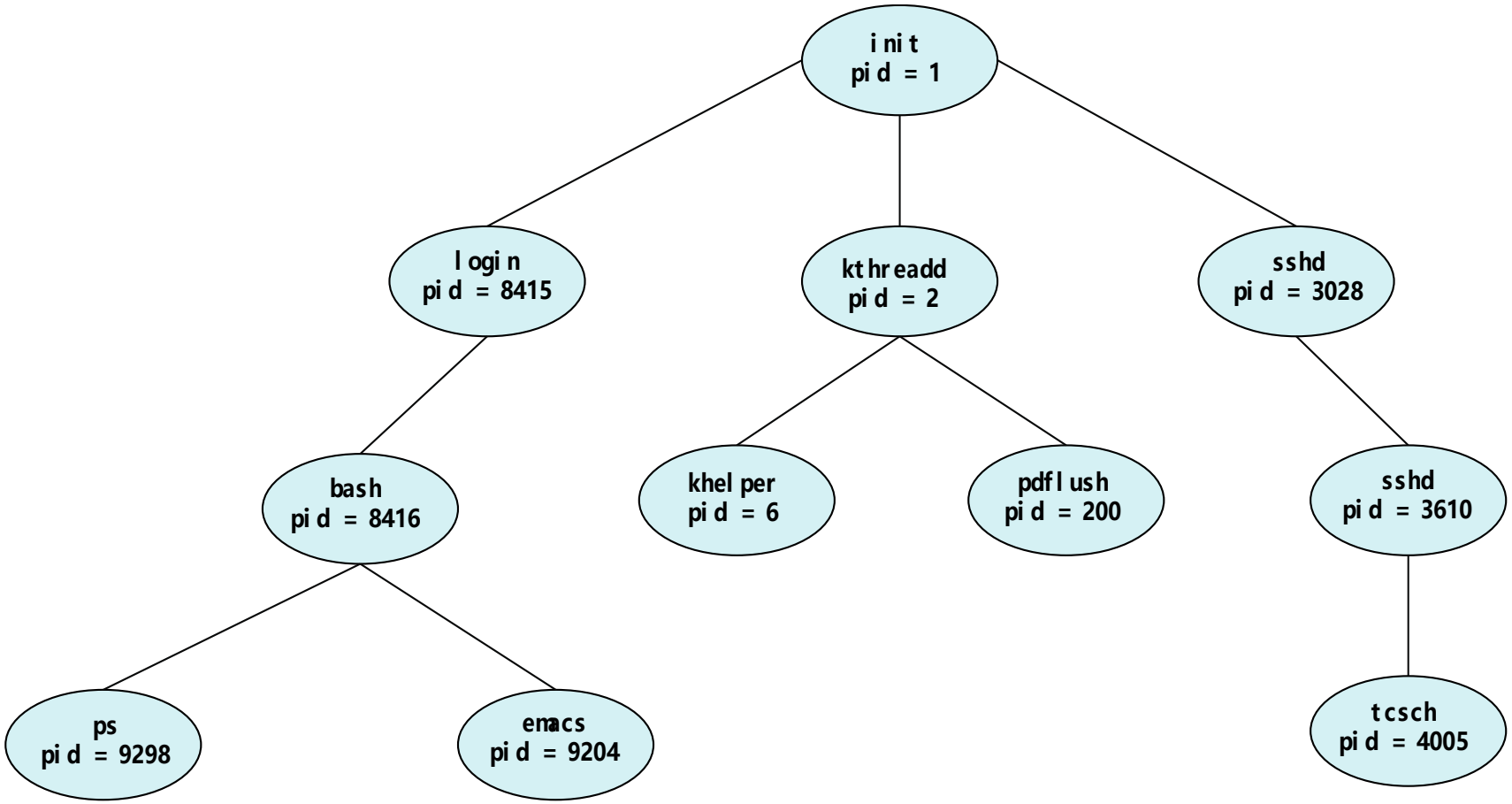
1、进程创建

- 为描述进程之间的创建关系，引入了进程图。
- 进程图又称进程树或进程家族树，是描述进程家族关系的一棵有向树。
- 进程图中的结点表示进程，若进程A创建了进程B，则从结点A有一条边指向结点B，说明进程A是进程B的父进程，进程B是进程A的子进程。

进程图例



Linux进程图的例子



进程创建原语

- **导致进程创建的原因有：**
  - 系统初始化
    - 启动OS时，通常会创建若干个前台和后台进程
  - 正在运行的程序执行了创建进程的系统调用
    - 正在运行的程序创建一个或多个进程协助其工作
  - 用户请求创建一个新进程
    - 交互式系统中，用户执行命令或启动程序
  - 一个批处理作业的初始化
    - 作业调度选择中一个作业

创建原语的主要功能

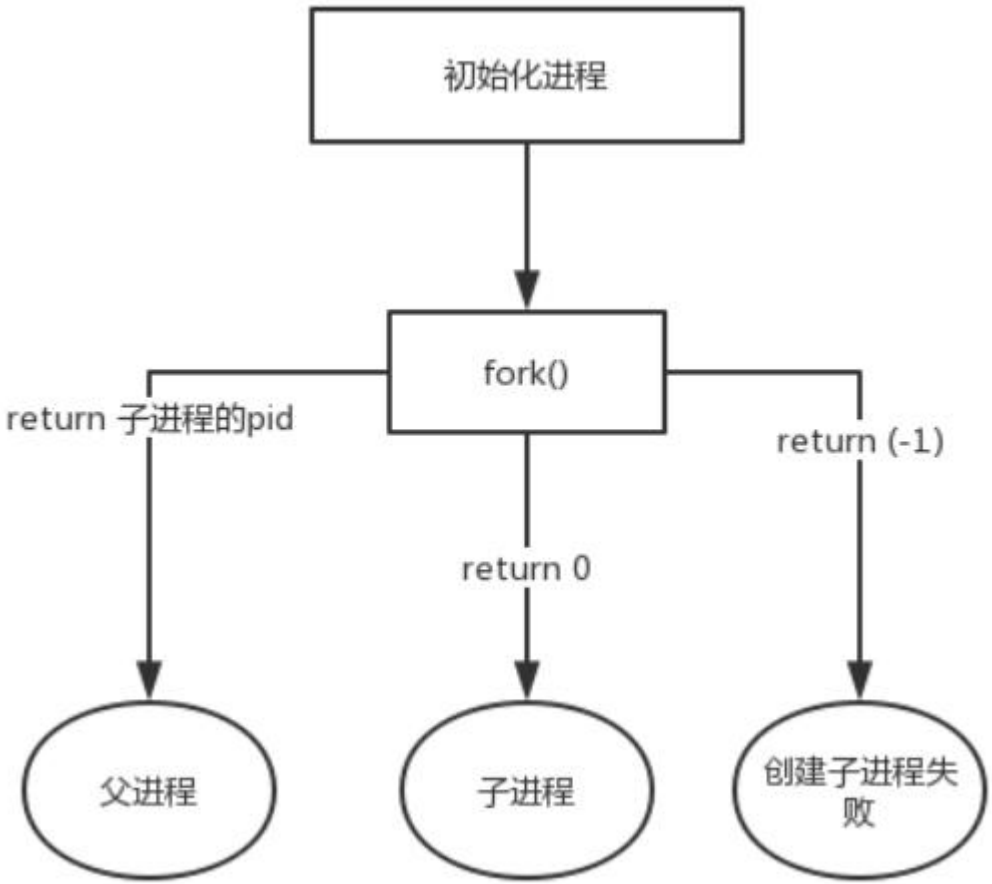
- 进程创建原语的功能是创建一个新进程，其主要操作过程如下：
  ① 向系统申请一个空闲PCB。
  ② 为新进程分配资源。如分配内存空间。
  ③ 初始化新进程的PCB。在其PCB中填入进程名、家族信息、程序和数据地址、进程优先级、资源清单及进程状态等。
  ④ 将新进程的PCB插入就绪队列。

进程创建

- **父子进程执行方式**
  - 父进程和子进程并发执行
  - 父进程等待，直到某个子进程或全部子进程执行完毕
- **父子资源共享原则**
  - 共享全部资源
  - 子进程共享父进程的部分资源
  - 父子进程不共享资源

进程创建

- **地址空间**
  - 子进程复制父进程空间内容（复制品）
  - 子进程装入另一个程序运行
- **Linux例子**
  - fork: 创造的子进程是父进程的完整副本，**复制**了父亲进程的所有资源
  - vfork：创建的子进程与父进程**共享**数据段,而且由vfork()创建的子进程将先于父进程运行
  - clone：允许子进程**有选择的**共享父进程资源

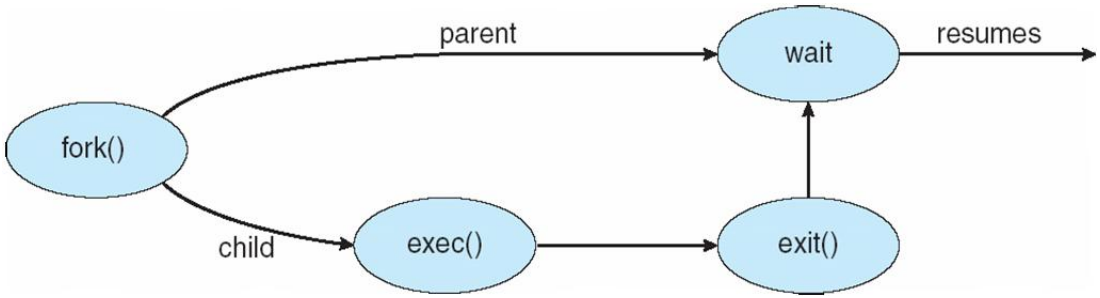

初始化进程
fork()
return 子进程的pid
return 0
return (-1)
父进程
子进程
创建子进程失败

创建进程的例子1

```
int main( ) {
    int i,p_id;
    if ( (p_id = fork( )) = = 0 ) {
        /* 子进程程序 */
        for (i = 1; i < 3; i ++ )
            printf("This is child process\n");
      } else if (p_id = = -1){
    printf( “fork new process error!\n” );
    exit(-1);
    } else {
          /* 父进程程序*/
     /*p_id 为新创建子进程的进程ID*/
      for ( i = 1; i < 3; i ++ )
        printf("This is parent process\n");
     }
  }
```

创建进程的例子2

```
int main( ) {
      pid_t pid;
      pid = fork();
      if (pid < 0){
            printf( "....." );
            exit(-1);
            }
       else if (pid ==0){
            execlp( "/bin/ls" ," ls" , NULL);
            }
        else{
            wait(NULL);
           printf( "Child Complete" );
           exit(0);
            }
   }
```



创建进程的例子3

```
int value = 5;
int main() {
  pid_t pid;
  pid = fork();
  if (pid == 0) { /* child process */
    value += 15;
  return 0;
  }
  else if (pid > 0) { /* parent process */
     wait(NULL);
     printf("PARENT: value = %d",value); /* LINE A */
     return 0;
   }
}
```

请问这个程序LINA A 打印信息？

课堂练习

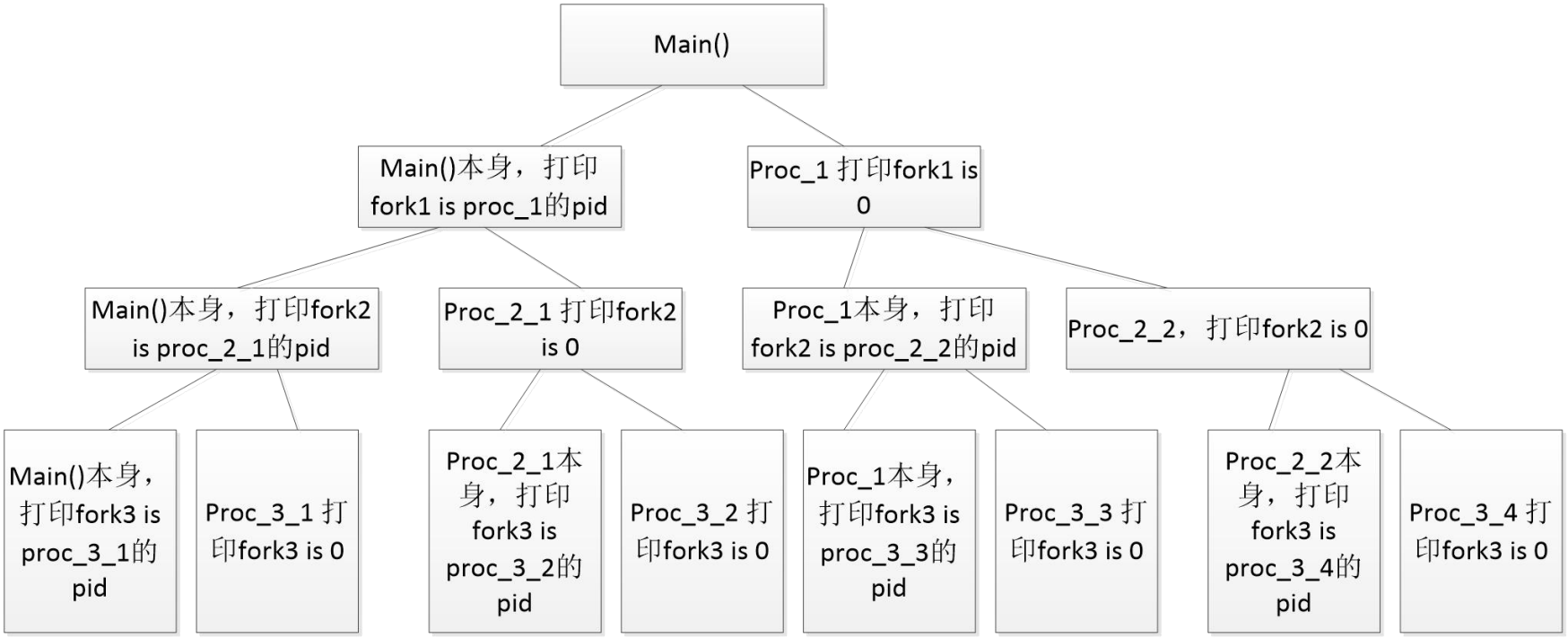
在Linux系统中运行以下程序

```
int main(){
   printf("fork1 is:%d\n", fork());
   printf("fork2 is:%d\n", fork());
   printf("fork3 is:%d\n", fork());
   }
```

请问这个程序最多可以输出多少行打印信息？产生多少个进程（含main函数本身）？请绘制进程家族树来说明。

解答：

- 包括自身，可共创建8个进程，输出信息14行，进程创建的过程图如下：



2、进程撤销

- **引起进程撤销(终止)的原因有：**
  - 正常结束
    - 自愿的，调用exit
  - 异常结束：超时、内存不足、地址越界、算术错、I/O故障、非法指令等。
    - 自愿出错退出；非自愿的出错退出（严重错误）
  - 外界干预：包括操作员或系统干预，父进程请求。
    - 非自愿，被其他进程kill

撤消原语采用的两种策略

- 撤消原语采用的两种策略：
  - 撤消指定标识符的进程
  - 撤消指定进程及其所有子孙进程
- 下面给出后一种撤消策略的功能描述。

撤消原语的主要功能

- 撤消原语的功能是撤消一个进程，其主要操作过程如下：
  ① 从系统的PCB表中找到被撤消进程的PCB。
  ② 检查被撤消进程的状态是否为执行状态，若是则立即停止该进程的执行，设置重新调度标志。
  ③ 检查被撤消进程是否有子孙进程，若有子孙进程还应撤消该进程的子孙进程。
  ④ 回收该进程占有的全部资源并回收其PCB。

3、进程阻塞与唤醒

- **引起进程阻塞及唤醒的事件：**
  - 请求系统服务。如请求分配打印机，但无空闲打印机则进程阻塞；当打印机重又空闲时应唤醒进程。
  - 启动某种操作并等待操作完成。如启动I/O操作，进程阻塞；I/O完成则唤醒进程。
  - 等待合作进程的协同配合。如计算进程尚未将数据送到缓冲区，则打印进程阻塞；当缓冲区中有数据时应唤醒进程。
  - 系统进程无新工作可做。如没有信息可供发送，则发送请求阻塞；当收到新的发送请求时，应将阻塞进程唤醒。

阻塞原语的主要功能

- 阻塞原语的主要功能是将进程由执行状态转为阻塞状态。其主要操作过程如下：
  ① 停止当前进程的执行；
  ② 保存该进程的CPU现场信息；
  ③ 将进程状态改为阻塞，并插入到相应事件的等待队列中；
  ④ 转进程调度程序，从就绪队列中选择一个新的进程投入运行。

唤醒原语的主要功能

- 当进程等待的事件发生时，由发现者进程将其唤醒。
- 唤醒原语的主要功能是将进程唤醒，其主要操作过程如下：
  ① 将被唤醒进程从相应的等待队列中移出；
  ② 将进程状态改为就绪，并将该进程插入就绪队列；
  ③ 转进程调度或返回。

阻塞与唤醒的关系

- 一个进程由执行状态转变为阻塞状态，是这个进程自己调用阻塞原语去完成的。
- 进程由阻塞状态转变为就绪状态，是另一个发现者进程调用唤醒原语实现的。
- 一般发现者进程与被唤醒进程是合作的并发进程。

4、进程的挂起与激活

- 挂起原语和激活原语都有多种实现方式如：
  - 把发出挂起原语的进程自身挂起
  - 挂起具有指定标识符的进程
  - 把某进程及其子孙进程挂起
  - 激活一个具有指定标识名的进程
  - 激活某进程及其子孙进程
- 下面以挂起或激活具有指定标识符的进程为例，说明这两种原语的主要功能。

挂起原语的主要功能

- 挂起原语的主要功能是将指定进程挂起，算法思想如下：
  ① 到PCB表中查找该进程的PCB；
  ② 检查该进程的状态，若为执行则停止执行并保护CPU现场信息，将该进程状态改为挂起就绪；
  ③ 若为活动阻塞，则将该进程状态改为挂起阻塞；
  ④ 若为活动就绪，则将该进程状态改为挂起就绪；
  ⑤ 若进程挂起前为执行状态，则转进程调度，从就绪队列中选择一个进程投入运行。

激活原语的主要功能

- 激活原语的主要功能是将指定进程激活。其算法思想如下：
  ① 到PCB表中查找该进程的PCB。
  ② 检查该进程的状态。若状态为挂起阻塞，则将该进程状态改为活动阻塞。
  ③ 若状态为挂起就绪，则将该进程状态改为活动就绪。
  ④ 若进程激活后为活动就绪状态，可能需要转进程调度。

3.6 进程间通信

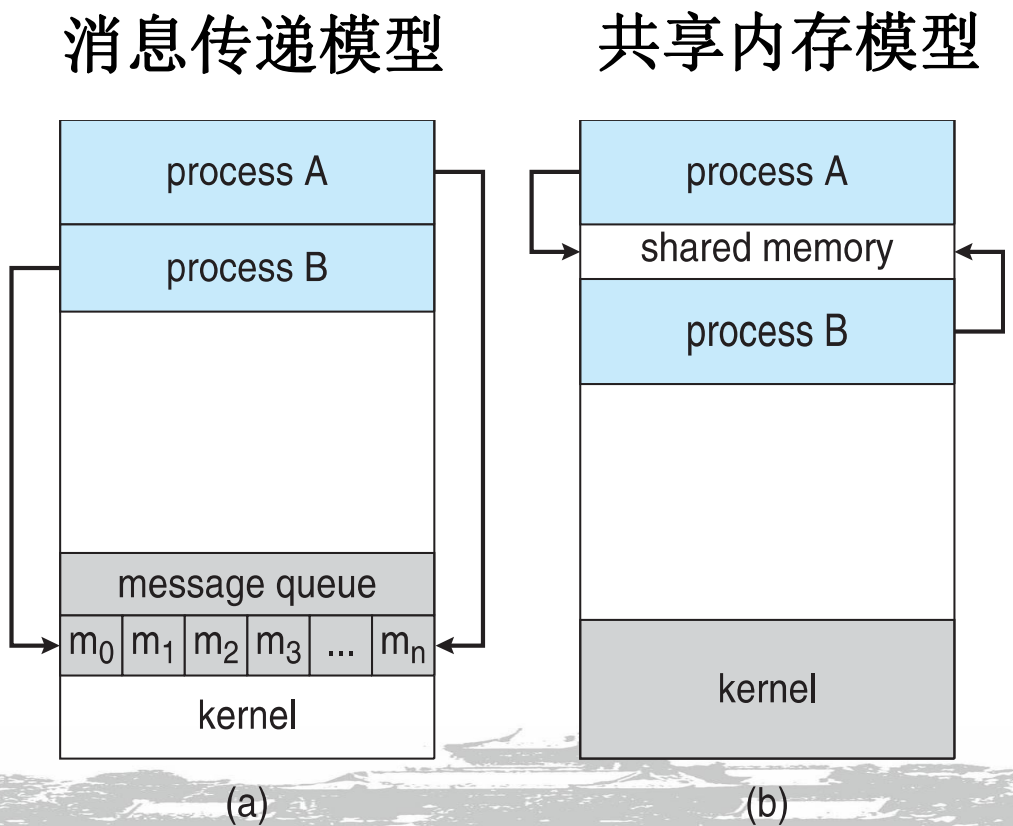
- 进程间通信机制（interprocess communication, IPC）
  - 指进程之间的信息交换，是进程之间的一种协作机制
- 进程协作的原因:
  - 信息共享：多用户交换感兴趣的信息
  - 提高运算速度：任务分割，并行执行
  - 系统模块化要求：将系统按照模块化划分
  - 方便性：单用户同时处理多个任务

进程间通信

- 进程间通信的三个问题
  - 一个进程和其他进程协作时，如何把消息传递给其他进程
  - 如何确保两个或多个进程，在关键活动中不出现交叉，即相互之间不会出现互相干扰
  - 如果进程执行过程中对数据的访问存在先后的关联顺序，如何确保访问顺序的正确性

进程间通信的基本形态

- 两类最基本的IPC模型
  - 消息传递模型，Message passing
  - 共享内存模型，Shared memory



进程间通信方式的发展(1)

- UNIX发展历史中，AT&T的Bell UNIX与加州大学伯克利分校的BSD是两大主力。
  - Bell致力于改进传统的进程IPC，形成了SYSTEM V IPC机制。
  - BSD在改进IPC的同时，把网络通信规程(TCP/IP)实现到UNIX内核中，考虑把计算机上的进程间通信纳入更广的网络范围的进程间通信，这种努力结果出现了socket通信机制。

进程间通信方式的发展 (2)

- STSTEM V IPC进程间通信机制，包括：消息队列、共享内存和信号量。
  - 消息队列：允许一个进程向其他进程发送消息
  - 共享内存：让多个进程可共享它们的部分虚地址空间
  - 信号量：允许若干进程通过它来同步协作地运行

进程间通信的类型

- 消息传递
  - 信号机制(Signal)
  - 消息传递机制(Message Passing)
  - 共享文件机制(Shared File)
  - 信号量机制(Semaphore)
  - 套接字通信(Socket)
- 共享内存
  - 共享内存机制(Shared memory)

1 消息传递机制

- 在消息传递机制中，进程间的数据交换以消息为单位，程序员直接利用系统提供的一组通信命令（原语 send, receive）来实现通信。
- 消息传递系统因其实现方式不同可分为：
  - 直接通信方式：发送进程将消息发送到接收进程，并将其挂在接收进程的消息队列上;接收进程从消息队列上取消息。例如：消息缓冲通信
  - 间接通信方式：发送进程将消息发送到信箱，接收进程从信箱中取消息。

消息传递机制的特点

- 消息传递系统支持
  - 实现进程间的信息交换
  - 实现进程间的同步
- 任何时刻发送
  - 一个正在执行的进程可在任何时刻向另一个正在执行的进程发送消息;
- 任何时刻请求
  - 一个正在执行的进程可在任何时刻向正在执行的另一个进程请求发出消息。
- 因此消息传递机制与进程的同步紧密相关

（1） 消息缓冲通信

- 消息缓冲通信是直接通信方式的一种实现。
- 发送或接收消息的进程必须指出信件发给谁或从谁那里接收消息
- 原语send（P，消息）：把一个消息发送给进程P
- 原语receive（Q，消息）：从进程Q接收一个消息

消息缓冲通信的实现思想

- 为了实现消息通信，发送进程应先在自己的工作区中设置一个发送区，把欲发送的消息填入其中，然后再用发送原语将其发送到接收进程，将其挂在接收进程的消息队列上。
- 接收进程调用接收原语从自己的消息队列中摘下第一个消息，并将其内容复制到自己的消息接收区内。

消息缓冲通信机制中的数据结构

- 消息缓冲区的数据结构如下：

```
struct message
 {
        sender;   发送者进程标识符
        size;     消息长度
        text;     消息正文
        next;     指向下一个消息缓冲区的指针
 }
```

所有发给同一进程的消息缓冲区构成该进程的消息队列（消息链、消息缓冲队列）

消息缓冲通信机制中的数据结构（续）

- 在进程的PCB中增加涉及通信的数据结构：

mq	消息队列队首指针
mutex	消息队列互斥信号量，初值为1
sm	消息队列资源信号量，初值为0

- 发送进程的工作区要开辟发送区：

id	接收进程标识符
size	消息长度
text	消息正文

- 接收进程的工作区要开辟接收区：

id	发送进程标识符
size	消息长度
text	消息正文

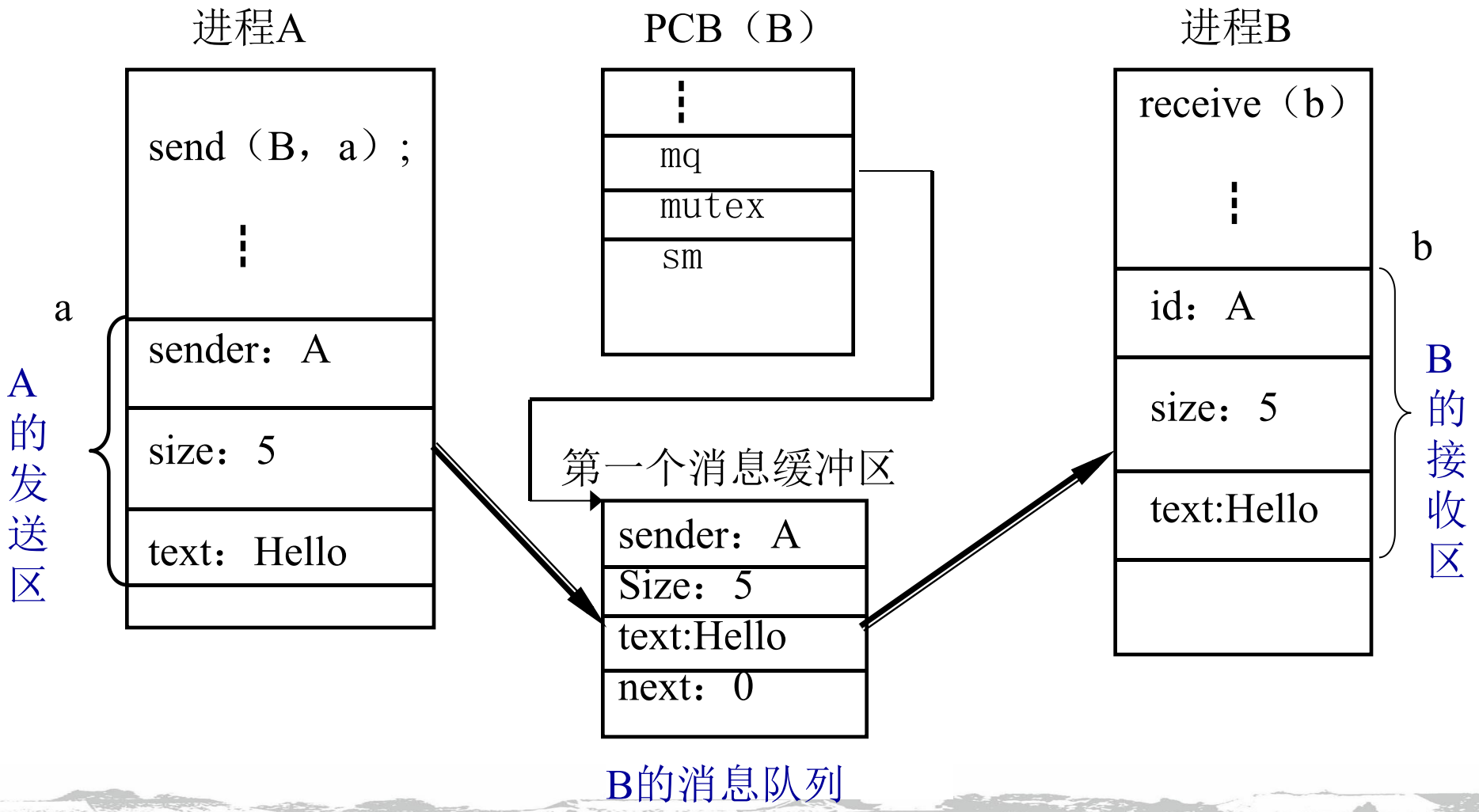
发送原语描述

```
procedure send(receiver, a) /*receiver为接收者标识号，
    a为发送区首址*/
{
        向系统申请一个消息缓冲区；
        将发送区a中的消息复制到该消息缓冲区中；
        找到接收进程的PCB；获得接收者进程的内部标识符j;
        wait(j.mutex);          /*互斥使用消息队列*/
        把消息挂到接收进程消息队列的尾部；
        signal(j.mutex);
        signal(j.sm);           /*接收进程消息队列长度加1*/
}
```

接收原语描述

```
void  receive(b)  /* b为接收区首址 */
{
        wait(j.sm);   /* j为调用进程的内部标识符 */
        wait(j.mutex);
        将消息队列中的第一个消息移出;
        signal(j.mutex);
        将消息复制到接收区b;
}
```

两个进程进行通信的过程



（2）信箱通信

- **信箱**通信方式是一种间接消息通信方式，进程之间通信需要通过共享数据结构实体--信箱来进行。
- 信箱是一种数据结构，其中存放信件。
- 信箱逻辑上分成信箱头和信箱体两部分。
  - 信箱头中存放有关信箱的描述。
  - 信箱体由若干格子组成，每格存放一个信件，格子的数目和大小在创建信箱时确定。

- 消息在信箱中可以安全地保存，只允许核准的目标用户随时读取。
  - 信箱可由操作系统创建，也可由用户进程创建，创建者是信箱的拥有者。据此，可把信箱分为以下三类。
    - 私用邮箱
    - 公用邮箱
    - 共享邮箱

信件的格式问题

- 单机系统中信件的格式可以分直接信件（又叫定长格式）和间接信件（又叫变长格式）。
- 网络环境下的信件格式较为复杂，通常分成消息头和消息体，前者包括了发送者、接收者、消息长度、消息类型、发送时间等各种控制信息；后者包含了消息内容。

信箱通信原语

- 信箱通信原语包括：
  - 信箱的创建和撤消：
  - 消息的发送和接收：
    - Send(mailbox,message);
    - Receive(mailbox,message);

消息通信中的同步问题

- 进程间的消息通信使用send和receive来进行，这些原语的实现有不同的设计选项。
- 消息传递可以是阻塞或非阻塞的（也称为同步或异步）

消息通信中的同步问题（Cont.）

- 对于发送进程来说，它在执行发送原语后有两种可能选择：
  - 发送进程阻塞，直到这个消息被接收进程接收到，这种发送称为阻塞发送。
  - 发送进程不阻塞，继续执行，这种发送称为非阻塞发送。

消息通信中的同步问题(Cont.)

- 对于一个接收进程来说，在执行接收原语后也有两种可能选择：
  - 如果一个消息在接收原语执行之前已经发送，则该消息被接收进程接收，接收进程继续执行。
  - 如果没有正在等待的消息，则该进程阻塞直到有消息到达，称为阻塞接收；或者该进程继续执行，放弃接收的努力，称为非阻塞接收。

消息通信中的同步问题(Cont.)

- 三种常用的组合方式：
  - 阻塞发送，阻塞接收。
  - 非阻塞发送、阻塞接收。
  - 非阻塞发送、非阻塞接收。

阻塞发送，阻塞接收
严格意义的顺序执行：生产者—消费者问题

```
message next_produced;
      while (true) {
 /* produce an item in next produced */
      send(next_produced);
      }
```

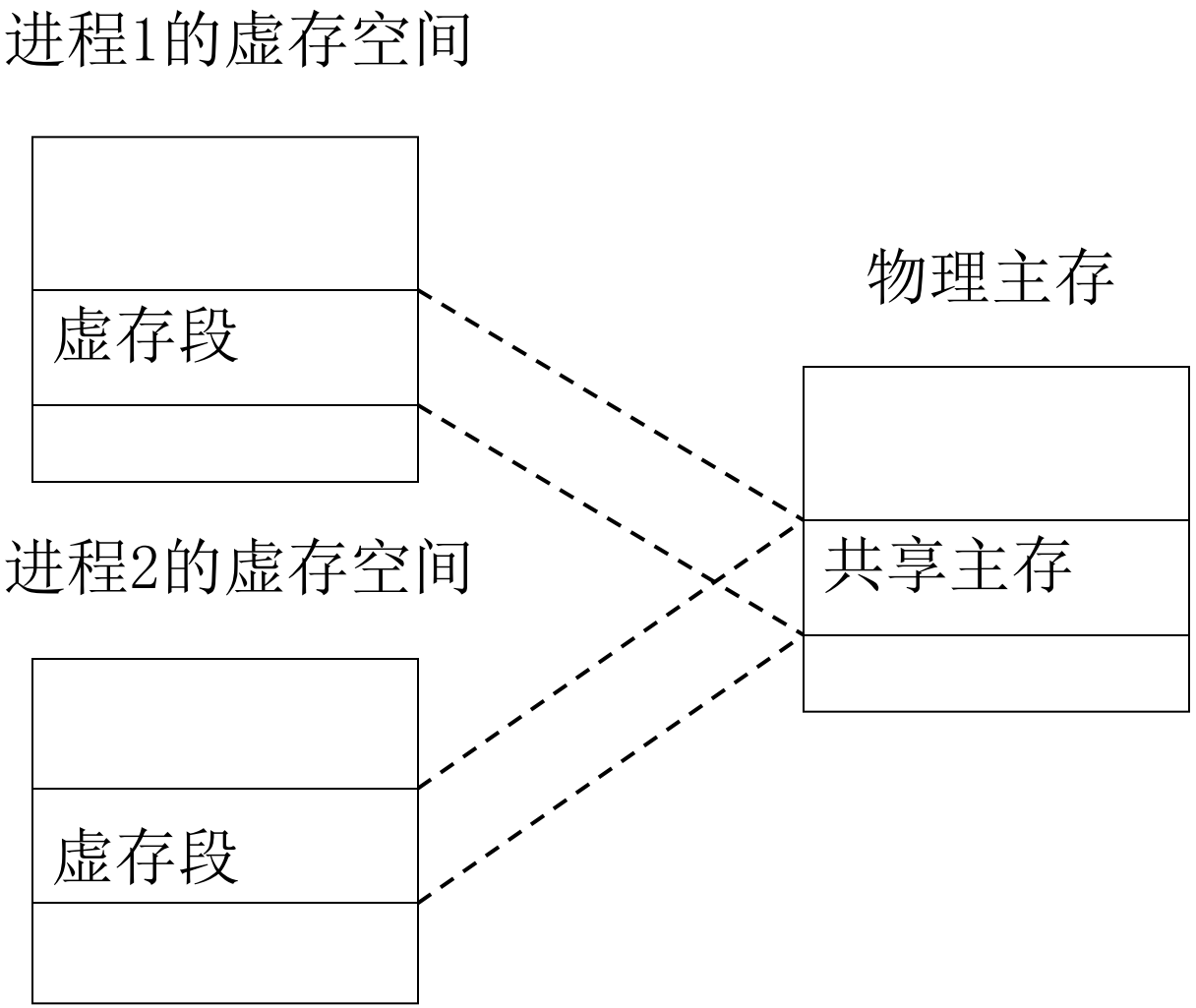
```
message next_consumed;
while (true) {
   receive(next_consumed);

   /* consume the item in next consumed */
}
```

2 共享存储区机制

- 相互通信的进程共享某些数据结构或共享存储区。
  1. 基于共享数据结构的通信方式
     - 诸进程通过共用某些数据结构交换信息。如生产者-消费者问题，共享了一个缓冲池
  2. 基于共享存储区的通信方式
     - 在存储器中划出一块共享存储区，诸进程可通过对共享存储区进行读或写来实现通信。包括建立共享存储区、附接及断接。

共享存储区通信机制



生产者-消费者问题

- 协作进程的通用范例—生产者消费者问题

- 生产者进程生产信息供消费者进程消费。可以使用两种缓冲区：
  - 无界缓冲 unbounded-buffer
  - 有界缓冲 bounded-buffer

生产者-消费者问题

- 下述变量由生产者-消费者共享：

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

生产者进程

```
Producer :
   item nextProduced;
   while (true)
  {
      produce an item in nextProduced ;
       while (((in + 1) % BUFFER_SIZE) == out)
           ; /* do nothing */
       buffer[in] = nextProduced;
       in = (in + 1) % BUFFER_SIZE;
    }
```

消费者进程

```
Consumer :
    item nextConsumed;
    while (true)
   {  while (in == out)
            ; /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
       consume the item in nextConsumed ;
    }
```

3 管道通信机制

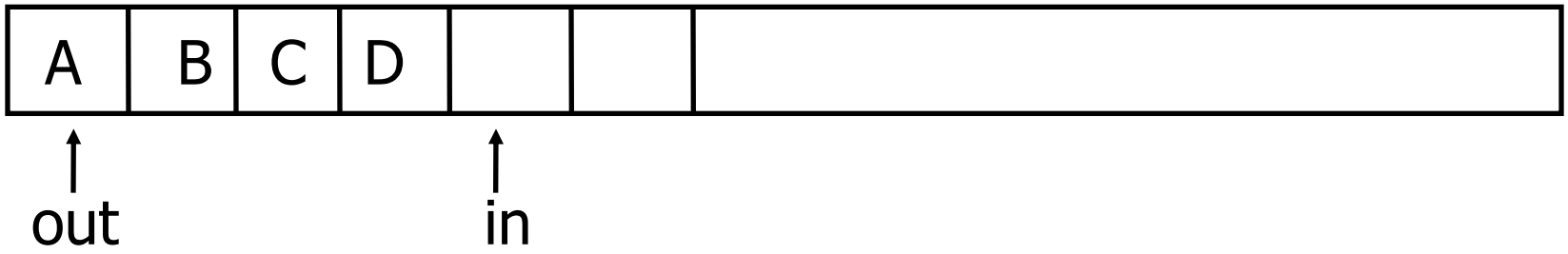
- 管道(pipeline)
  - 是连接读写进程的一个特殊文件
  - 允许进程按先进先出方式传送数据,也能使进程同步执行操作。
  - 发送进程以字符流形式把大量数据送入管道，接收进程从管道中接收数据，所以叫**管道通信。**
- 管道的实质是一个共享文件，基本上可借助于文件系统的机制实现，包括（管道）文件的创建、打开、关闭和读写。

管道

- 管道机制应提供以下三方面的协调能力：
  - **互斥**：进程对通信机构的使用应该互斥，一个进程正在使用某个管道写入或读出数据时，另一个进程就必须等待。
  - **同步**：管道长度有限，发送信息和接收信息之间要实现正确的同步关系，当写进程把一定数量的数据写入管道，就去睡眠等待，直到读进程取走数据后，把它唤醒。
  - **存在**：发送者和接收者双方必须能够知道对方是否存在，如果对方已经不存在，就没有必要再发送信息。

管道通信示意图1

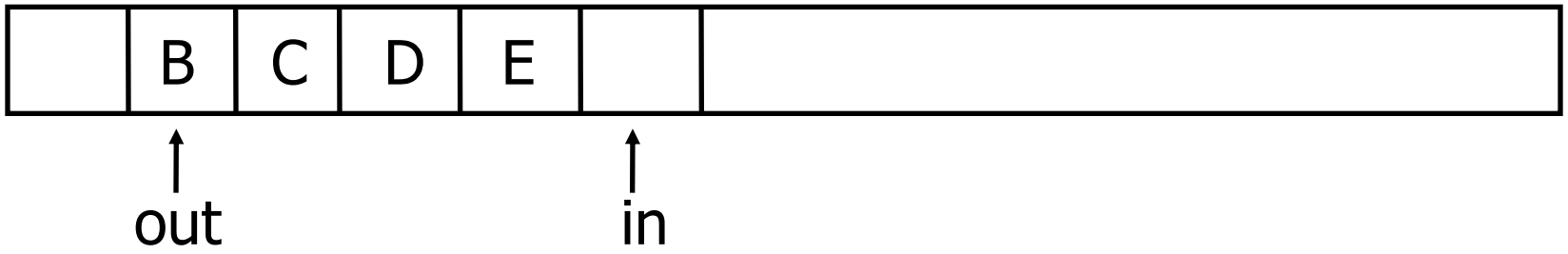
- 初始时，其长度为4，系统将管道看成一个循环队列。按先进先出的方式读写。



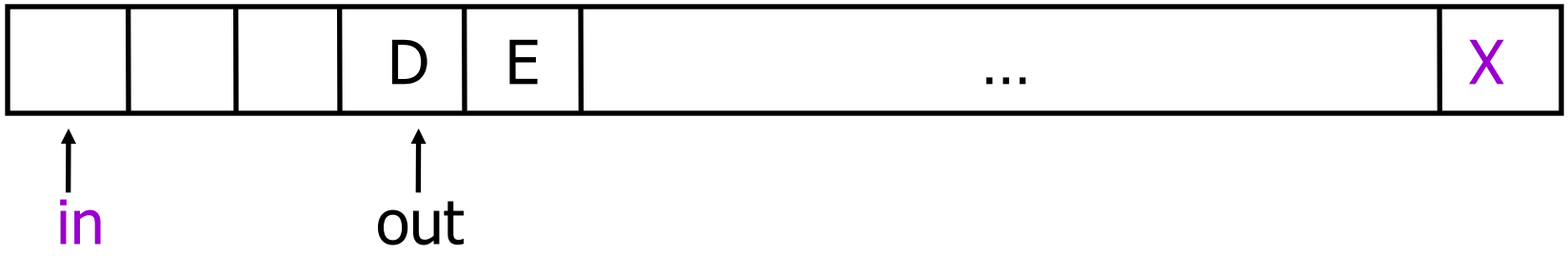
- 写入字符E后，管道长度为5

管道通信示意图2

- 读一个字符后，管道长度为4



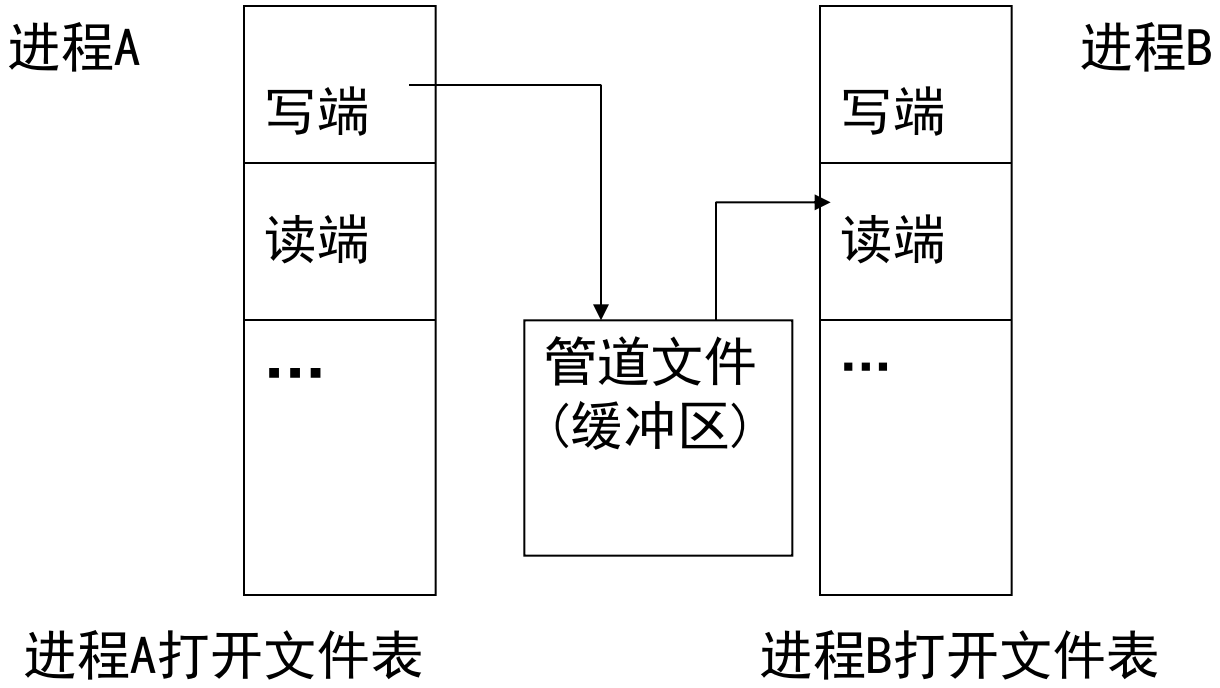
  - 若管道容量为n且in=n时，再写入一个字符，则in移到管道的另一端。



匿名管道的特点

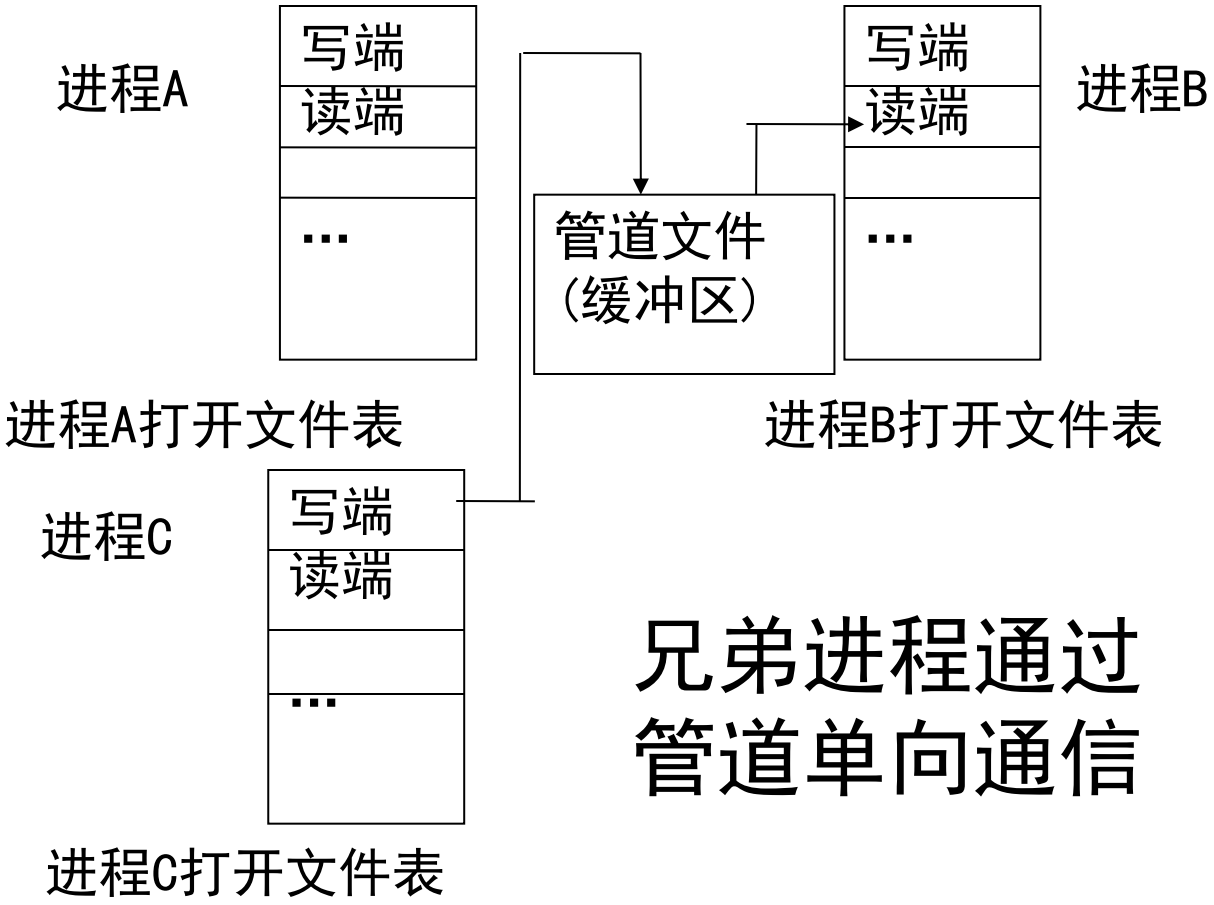
- 匿名管道是半双工的，数据只能向一个方向流动；要求双向通信时，需要建立两个匿名管道。
- 只能用于具有亲缘关系的进程通信，亲缘关系指的是具有共同祖先，如父子进程或者兄弟进程之间。
- 匿名管道对于管道两端的进程而言，就是一个文件，但它不是普通文件，而是一个只存在于主存中的特殊文件。
- 一个进程向管道中写入的内容被管道另一端的进程读出。写入的内容每次都添加在管道缓冲区的末尾，并且每次都是从缓冲区的头部读出数据。

父子进程通过管道传递信息



父子进程通过
管道单向通信

兄弟进程通过管道传送信息



有名管道

- 又称FIFO，克服只能用于具有亲缘关系的进程之间通信的限制。
- FIFO提供一个与路径名的关联，以FIFO的文件形式存在于文件系统中，通过FIFO不相关的进程也能交换数据。
- FIFO遵循先进先出，对管道及FIFO的读总是从开始处返回数据，对它们的写则把数据添加到末尾。

小作业2

1、 课本P103 3.5 (画图说明进程创建关系)

2 、课本P103 3.7

3、 结合以下代码回答下面的问题：

```
int main()
{
      if (!fork()) {printf("%c\n",'A');}
      printf("%c\n",'B');
      fork();
      printf("%c\n",'C');
}
```

（1）该代码正确运行产生的输出会是什么？

（2）该代码共创建了几个进程（含main函数本身）？ 画图说明进程创建关系。 并在图上标出输出相应字符的位置。

大作业2

- 在大作业1要求安装的国产操作系统环境下，实现第三章课后编程项目1 or 项目2