# 计算机操作系统

# 第2章 进程的描述与控制

2020级互联网专业

# 2.1 进程的引入

**程序的顺序执行要求：**

- 顺序性：程序按次序逐步执行
- 封闭性：独占全机资源，所有资源状态由程序完成
- 可再现性：程序的执行结果不变，且与计算机的执行速度无关

**多道程序系统、分时系统、实时系统**

- 存在多个运行的程序
- 程序执行具有并发性、共享性、异步性

**进程 = 程序（纯代码段 + 数据段） + 程序专用控制数据区**

**程序在并发执行时的特征：**

- 间断性：多个程序相互制约（如输入程序与计算程序的先后关系）
- 失去封闭性：多个程序共享资源(资源状态由多个程序改变)
- 不可再现性：程序的执行结果与计算机的执行速度有关，也与其它程序的执行相关

**并发、共享、异步环境下能正常运行**

- 必须专门设置一个控制数据区，保留程序运行的现场
- 现场信息包括：进程标识、处理机状态、进程调度、进程控制等信息

# 2 .2.1 进程的定义与特征

## 进程的定义

- 进程是程序的一次执行，是进程实体(包括程序段、数据和PCB)的运行过程
- 是系统进行资源分配和调度的独立单位

## 进程的特征

- 结构性：程序段、数据段和进程控制块
- 动态性：动态地创建、执行、撤消
- 并发性：在同一时段内有多个进程运行
- 独立性：独立运行和获得资源的基本单位
- 异步性：异步执行

## 进程与程序的区别

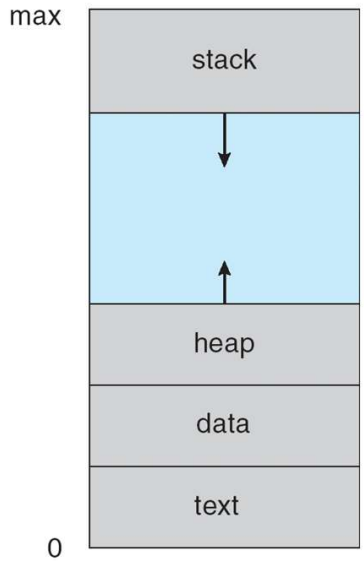
- 进程是动态的，程序是静态的(是指令集合)
- 一个程序可以包含多个进程
- 进程可以描述并发活动，程序则不明显
- 进程执行需要处理机，程序存储需要介质
- 进程有生命周期，程序是永存的

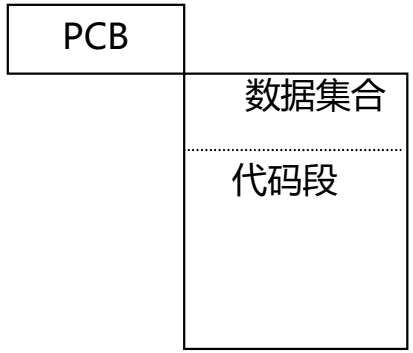
## 系统进程（kernel）

- 在系统态、核心态下运行，拥有一些独占的资源，可被最高优先权优先使用资源
- 可执行一切指令，访问所有寄存器和存储区，进行设备I/O操作

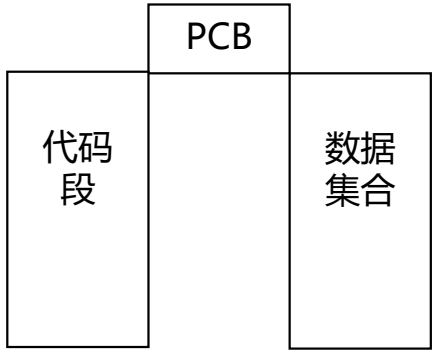
# 2.2.1 进程的结构

- 代码段：描述进程所完成的功能
- 数据集合：程序运行时所需要的数据区
- 进程控制块（PCB）：是标识进程的存在，并刻画进程瞬间特征的数据结构，位于内核

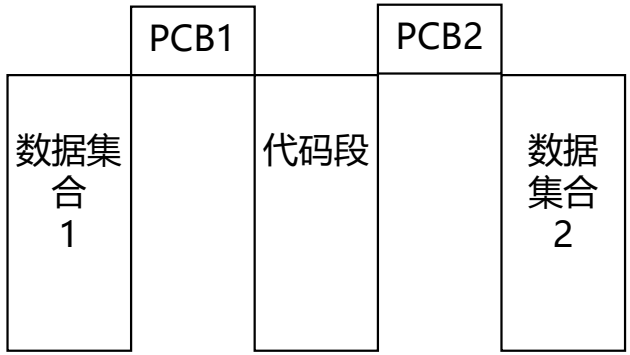



代码段与数据集合放在连续内存中




代码段与数据集合放在内存的不同区域




两个进程有各自的PCB和数据集合，但共享同一个代码段

# 2 .2.2 进程的类型

## 进程的类型

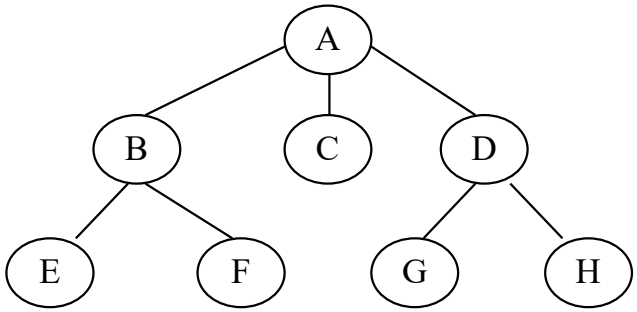
- 父进程：由系统或用户首先创建的进程
- 子进程：由父进程创建的进程
- 父、子进程的关系
  - 进程控制：任何子进程只能由父进程创建及撤消，子进程不能对父进程实施控制
  - 运行方式：可同时执行，也可以等待子进程完成后，父进程再执行
  - 资源共享：子进程可全部或部分共享父进程的拥有的资源

## 进程关系图（树）

- 父子（孙）进程之间构成一棵树形结构，即进程树

## 用户进程

- 在目态（用户态）下运行，通过系统服务请求的手段竞争系统资源
- 不能直接做I/O操作，只能执行规定的指令，访问指定的寄存器和存储区



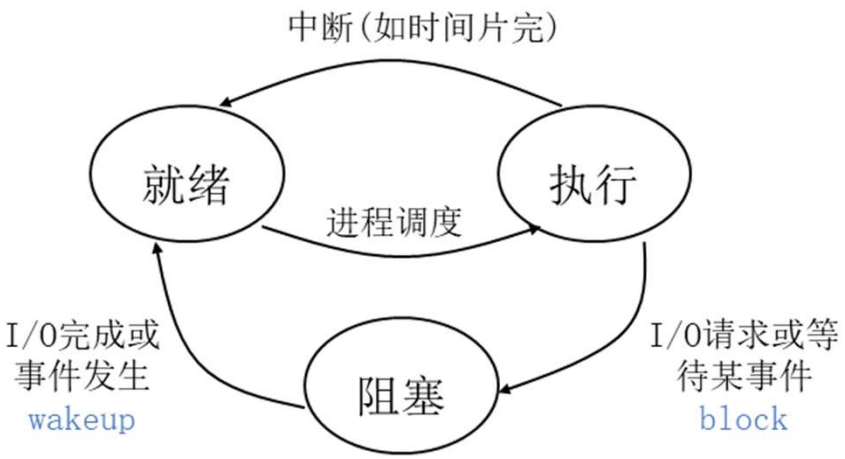
# 2.2.2 进程的基本状态及转换

## 三种基本状态

- 就绪状态：进程获得了除处理机CPU之外的所有资源（可以有多个，排成就绪队列）
- 执行状态：进程的程序正在处理机上执行（单CPU中只有一个进程处于该状态）
- 阻塞状态：因发生某事件（如请求I/O，申请缓存空间等）而暂停执行的状态（也称为等待状态）

## 进程基本状态的转换

- 处于执行状态的进程，可以主动用阻塞原语(block)将其变为阻塞状态
- 处于阻塞状态的进程，可以用唤醒(wakeup) 原语将其变为就绪状态



# 2.2.3 挂起操作和进程状态的转换

## 挂起状态

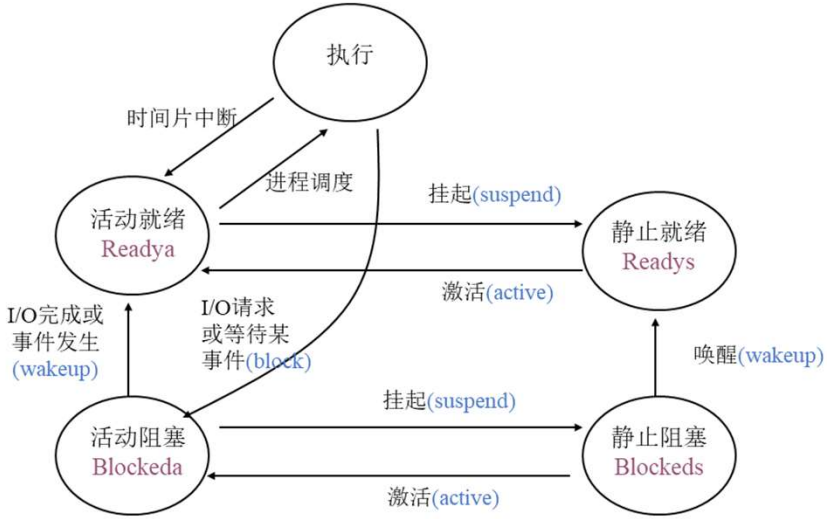
- 挂起状态是一种静止状态，即不能马上投入运行的状态，包括静止就绪状态(Readys)和静止阻塞状态(Blockeds)
- 处于挂起状态的进程可以存放到外存上保留，可以回收这些挂起状态进程的内存、设备等部分资源

## 设置挂起状态的原因

- 终端用户的需要：（调试）
- 父进程的需求：（子进程同步）
- 负荷调节的需要：（减轻负荷）
- 操作系统的需要：（检查资源使用情况）

## 活动状态与挂起状态的转换

- 处于活动状态的进程(Readya, Blockeda)，可以用挂起（Suspend）原语将其变为挂起状态(Readys, Blockeds)
- 处于挂起状态(静止状态)的进程，可以用激活（Active）原语将其变为活动状态



# 2.2.4 进程管理中的数据结构

## OS中用于管理控制的数据结构

- 每个资源、每个进程都对应一个数据结构
  - 用于表征其实体，表中包含资源或进程的标识、描述、状态等信息以及一批指针
- 通过指针，按类型，链接成不同的队列
  - 同类资源或进程的信息表
  - 同一进程所占用的资源信息表

## OS管理的数据结构有4类

- 进程管理表，也称为进程控制块PCB
- 内存表
- 设备表
- 文件表

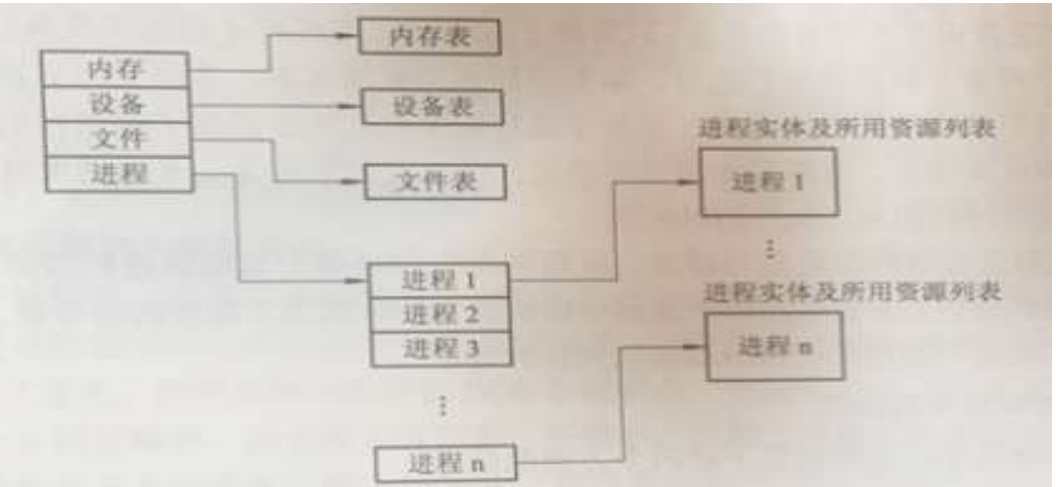


图 2-9 操作系统控制表的一般结构

# 2.2.4 进程管理表（进程控制块PCB）

## PCB的作用

- 描述进程的变化过程
- 记录进程的外部特征
- 记录进程与其他进程的联系
- 是进程存在的唯一标志
- 系统通过PCB控制和管理进程

## PCB的内容

- 进程标识符信息
  - 内部标识符信息（进程唯一序号）；
  - 外部标识符信息（由创建者提供的进程名字）

## PCB的内容

- 处理机状态信息
  - 处理机执行进程的现场信息（通用寄存器、指令计数器、程序状态字PSW、用户堆栈指针等）
- 进程调度信息
  - 进程状态；进程优先级；进程调度所需的其它信息(如进程已用CPU时间)；事件（阻塞原因）
- 进程控制信息
  - 程序和数据的地址
  - 进程同步和通信机制（如消息队列指针等）
  - 资源清单
  - 链接指针（与其它PCB形成一个队列）

# 2.2.4 PCB的组织形式

## 线性表方式

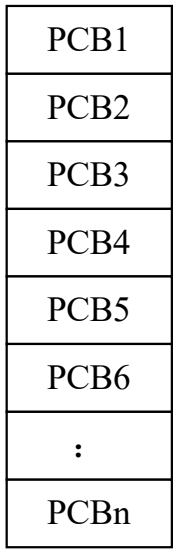
- 将所有PCB不分状态，全部组织在一个连续表中，优点是简单
- 缺点是必须扫描整个PCB表
- 一般在进程较少的系统中使用

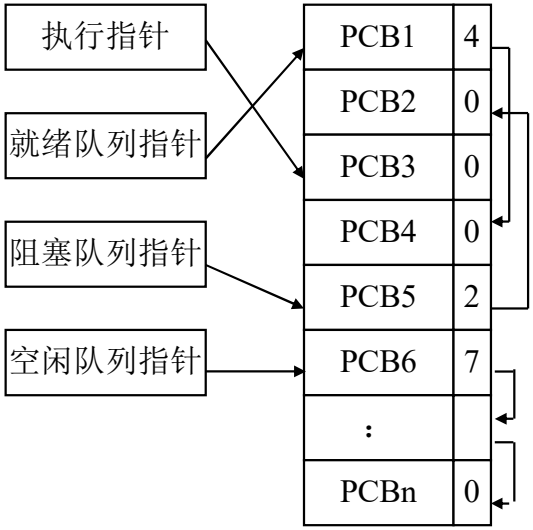
## 链接方式

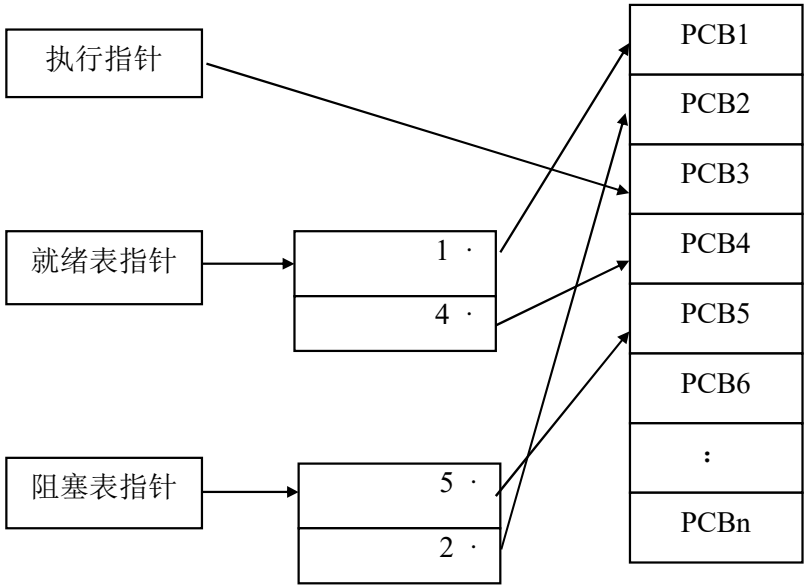
- 把具有相同状态的PCB，链接成一个队列
- 执行队列、就绪队列、阻塞队列、空闲队列

## 索引方式

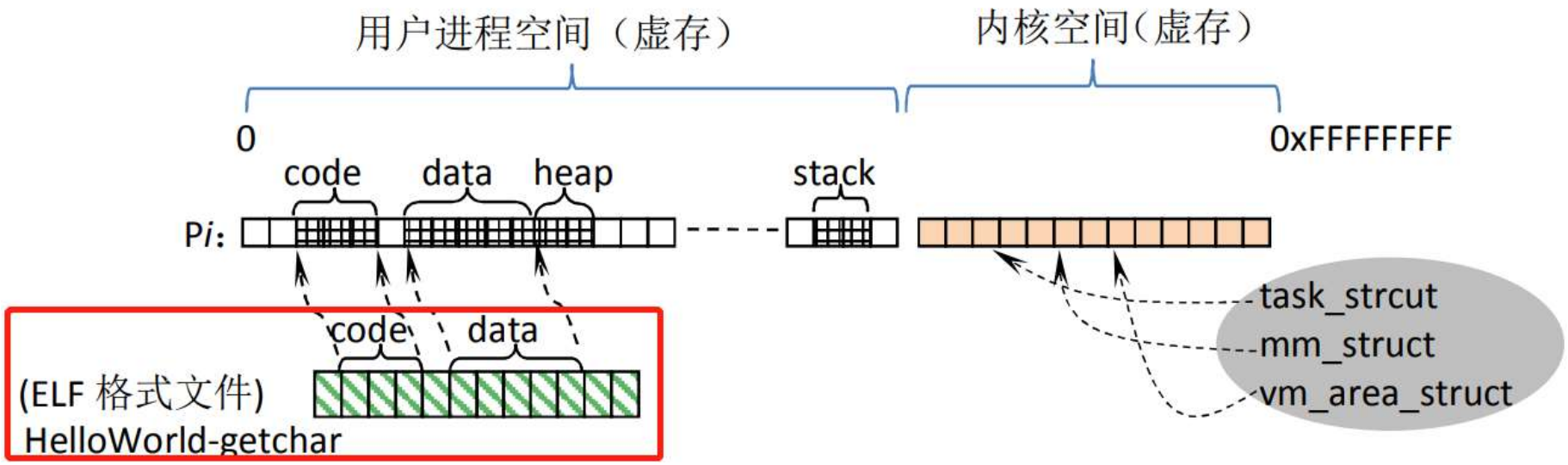
- 把具有相同状态的PCB在PCB表中的首址，按顺序组成一个索引表(如就绪索引表、阻塞索引表)



# Linux进程示例：ELF可执行文件创建进程影像




- 将 HelloWorld 文件的代码部分拷贝到新进程虚存空间的低地址部分（图中 code）数据拷贝到图中 data
- 在用户进程高端位置建立用户态的堆栈（对应图中 stack）
- 在内核空间中分配一些管理数据结构，如PCB（ struct task_struct）和内存描述符（struct mm_struct）等
- 进程映像+管理数据结构构成进程实体，进程可向系统申请各种资源
- 进程实体被调度运行就构成了进程的活动，如果系统中有多个活动的进程，它们各自有独立的进程空间（各自独立地使用自己的 0~0xFFFFFFFF 虚存空间）

# Linux进程PCB——task_struct结构体

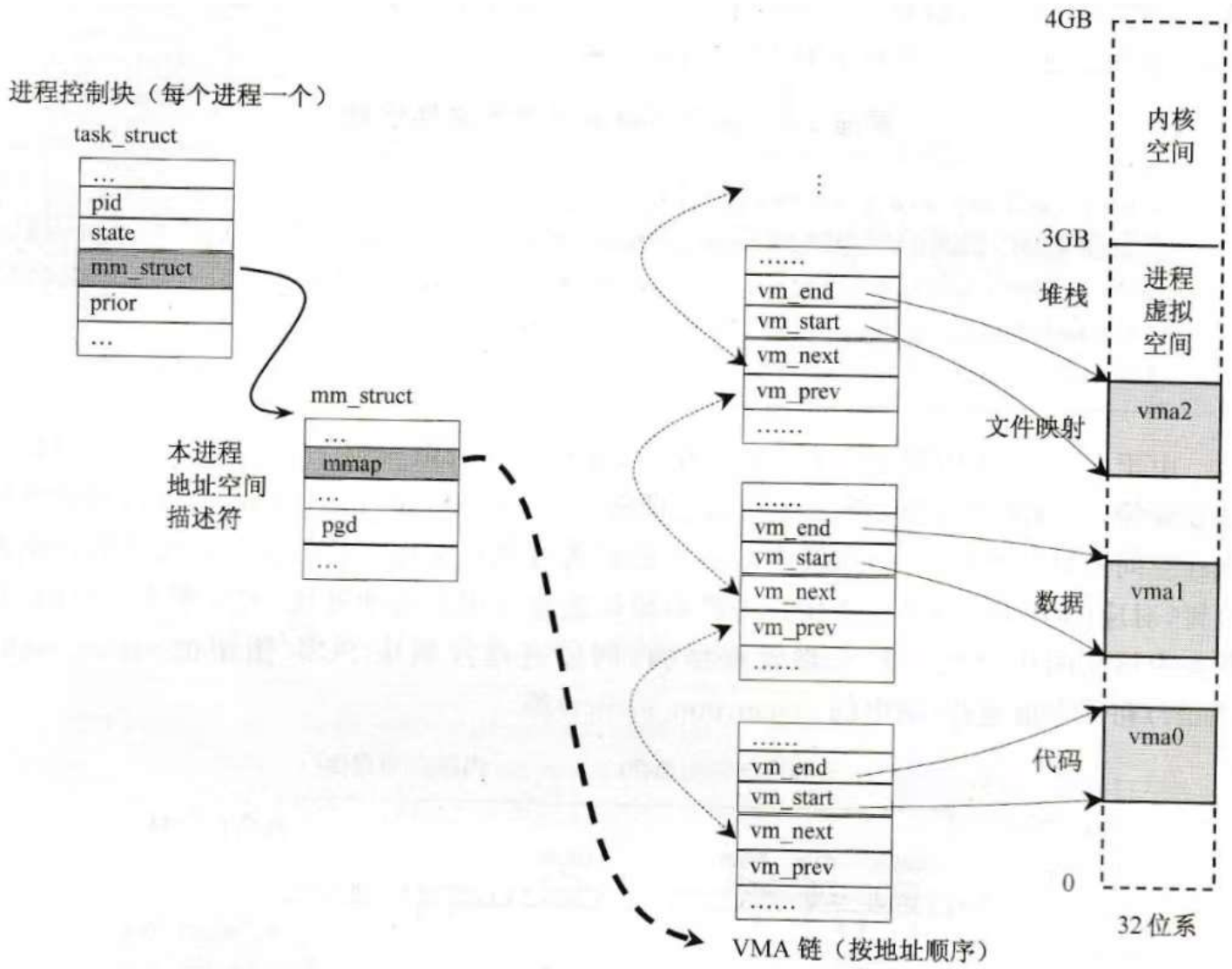



图 2-3　task_struct、mm_struct 和 vm_area_struct 的关系

- task_struct的mm成员描述进程的内存布局
- mm_struct的mm成员管理本进程内部的各个内存区域（代码段、数据段、堆栈区）
- 这些相同属性的连续虚存空间成为VMA（virtual memory area）
- VMA区域使用vm_area_struct结构体描述（用vm_start/vm_end成员变量指出内存区域的起止地址）

# 2.3 进程控制

- 进程控制是进程管理中最基本的功能
  - 创建新进程
  - 终止已完成的进程
  - 将因发生异常情况而无法继续运行的进程置于阻塞状态
  - 负责进程运行中的状态转换
- 进程控制一般由OS的内核中的原语来实现
  - 原语就是一旦开始执行，就不可以被中断，一直连续执行完毕

# 2.3.1 操作系统内核

- 将处理机的执行状态分为系统态和用户态
- 与硬件紧密相关的**中断处理**、设备驱动程序以及运行频率较高的**时钟管理**、**进程调度**和许多模块公用的基本操作都**常驻内存**，称为内核；
  - 系统态具有较高特权，能执行一切指令，访问所有寄存器和存储区
  - 用户态仅能执行规定的指令，访问指定的寄存器和存储区
- 大多数OS都包含**支撑功能**和**资源管理**功能
  - 支撑功能包括中断处理、时钟管理、原语操作（一个操作种的所有动作，要么做就全做，）
  - 资源管理功能包括进程管理、存储器管理、设备管理

# 2.3.2 进程的创建

## 进程的层次结构

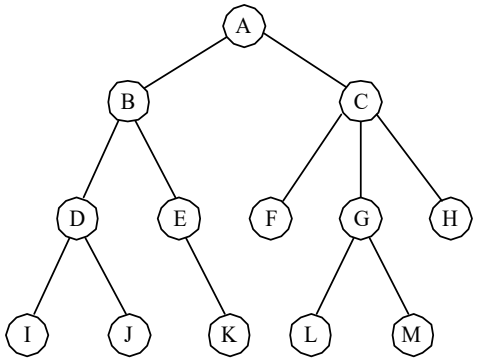
- 创建进程的进程称为父进程，被创建的进程称之为子进程
- 子进程可以创建更多的孙进程
- 子进程可以继承父进程的所有资源（文件、缓冲区等）
- 子进程被撤销时，把资源归还给父进程
- 撤销父进程时，需要撤销所有子进程
- PCB中有家族关系表的数据项

## Windows系统中，不存在层次结构

- 一个进程创建另外的进程时，获得句柄，可以用这个句柄控制被创建的进程
- 进程之间的关系是获得句柄与否、控制与被控制

## 引起创建进程的事件

- 用户登录：用户合法登录后（分时系统）
- 作业调度：某作业被调度后（批处理系统）
- 提供服务：用户调用某些系统调用或命令后
- 应用请求：由用户程序自己创建进程
  - 如Unix中的fork()函数
  - Windows中的CreateProcess()函数



# 2.3.2 进程的创建

## 进程创建原语

```
procedure create (pn, pri, res, fn, args);
begin
   getfreepcb(i);
   if i=NIL then return (NIL);
   i.id := pn; i.priority := pri;  i.resources := res;
   memallocate(datasetsize, add);
   if add = NIL then
   begin
      pcbrelease (i);        return (NIL);
   end;
   i.dataadd := add; i.datasize:= datasetsize;
   datasetinit(i.dataadd, args);    filestate(fn, add, size);
   if add = NIL then
   begin
            memallocate(size, add);
             if add = NIL then
             begin
                memrelease (i.dataadd, i.datasize);          pcbrelease(i);
        return(NIL);
             end
             read(fn, size, add);
           end;
   i.textadd := add; i.textsize : = size;    i.prog := fn;       i.pc := add;
   i. children := 0;  i.parent := EXE;    EXE.children := EXE.children+1;
   i.state := "ready"; i.queue := RQ;
   insert(RQ, i);
   otherinit;
   return(i);
End;
```

**原语说明：**

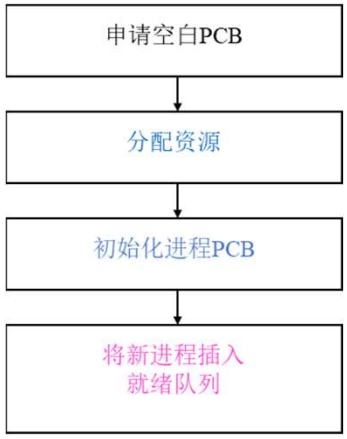
Pn新进程的外部名，pri优先级，res初始资源，fn执行程序文件名，args是fn的参数表

add内存区地址，size内存区大小，RQ为就绪队列，EXE是执行态进程的PCB指针

进程控制由原语操作实现。原语是由若干条指令构成，用于完成一定功能的一个过程；

原语操作是一种“原子操作”；原语操作中的所有动作，要么全做，要么全不做；原语操作是一种不可分割的操作

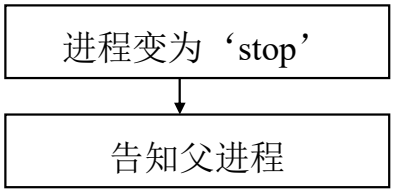
原语操作是OS内核执行基本操作的主要方法



# 2.3.3 进程的终止

□**进程正常结束，**通过停止原语实现

```
procedure halt(i);
begin
   i.state := 'stop';
   send(i.parent, 'completed');
end;
```

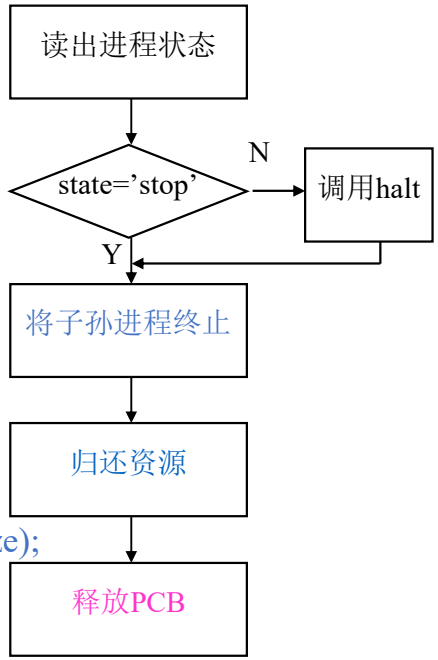



□**异常结束：**出现某些错误和故障而迫使进程终止（如保护错、越界/权、非法指令、超时、错误等）

□**外界干预：**用户或系统干预\父进程请求或终止

□**进程终止的原语**

```
procedure destory(i);
begin
   if i.state <> 'stop' then
      halt(i);
   while i.children > 0 do
   begin
      i.children := i.children – 1;
      findchild(i, child);
      destory(child);
   end;
 memrelease(i.dataadd, i.datasize)
   close(i.prog, t);
   if t = true then
      memrelease(i.textadd, i.textsize);
   resrelease(i);
   remove(i.queue, i);
   pcbrelease(i);
end;
```

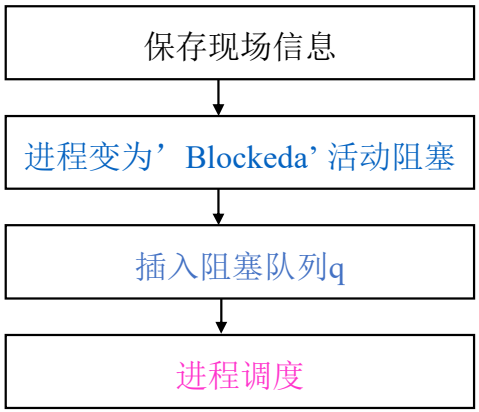



原语说明：
remove将进程移出队列
findchild查找子进程

# 2.3.4 进程的阻塞与唤醒

## 阻塞是主动的，引起进程阻塞的事件

- 请求系统服务（如请求分配I/O）
- 启动某种操作（如启动I/O）
- 新数据尚未到达（如进程通信）
- 无新工作可做（系统进程）




## 进程阻塞原语

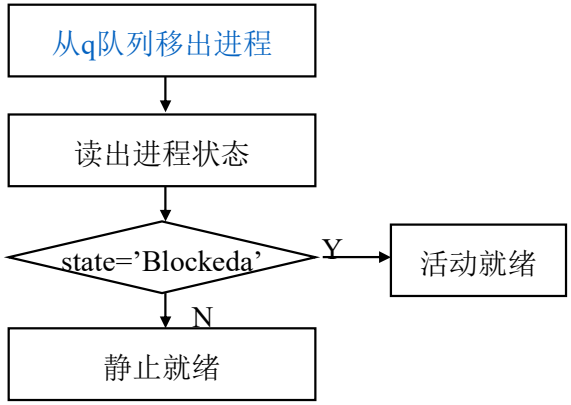
```
procedure block(q);
begin
   save(EXE);
   EXE.state :=  'Blockeda' ;
   EXE.queue := q; 阻塞队列
   insert(q, EXE);
   EXE := NIL;
   scheduler;
end;
```

- 原语说明：
  - insert将进程插入队列

# 2.3.4 进程的阻塞与唤醒

## 引起进程唤醒的事件

- 系统请求实现（如获得I/O资源）
- 某种操作完成（如I/O操作完成）
- 新数据已经到达（如其它进程已将数据送达）
- 又有新工作可做（系统进程）




## 进程阻塞原语

```
procedure wakeup(q);
begin
   outqueue(q, i);
   if i.state =   'Blockeda'
   then i.state =   'Readya'
   else  i.state =   'Readys'  ;
   i.queue := RQ;  就绪队列
   insert(RQ, i);
end;
```

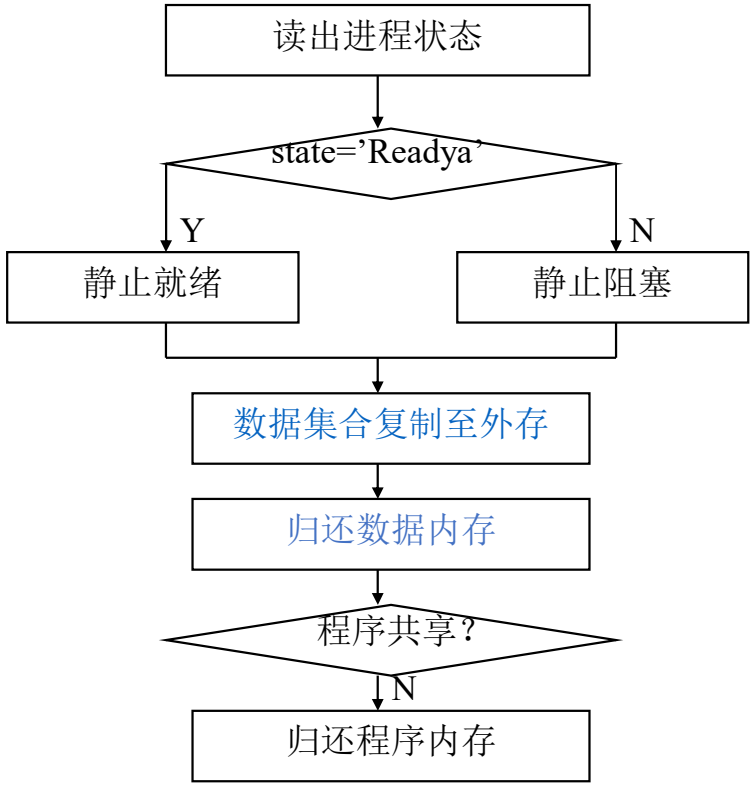
- 原语说明：
  - outqueue从队列移出进程，并返回该进程号
  - Readya活动就绪
  - Readys静止就绪

# 2.3.5 进程的挂起与激活

## 调用挂起原语suspend

```
procedure suspend(i);
begin
   if i.state =  'Readya'
   then i.state= 'Readys'  else
i.state= 'Blockeds' ;
   swapout(i.dataadd, i.datasize, add)
   i.swapadd := add;
   memrelease(i.dataadd, i.datasize);
   close(i.prog, t);
   if t = true then memrelease(i.textadd, i.textsize);
end;
```

- 原语说明
  - swapout将数据集合复制到外存交换区，并返回地址
  - Blockeda活动阻塞
  - Blockeds静止阻塞

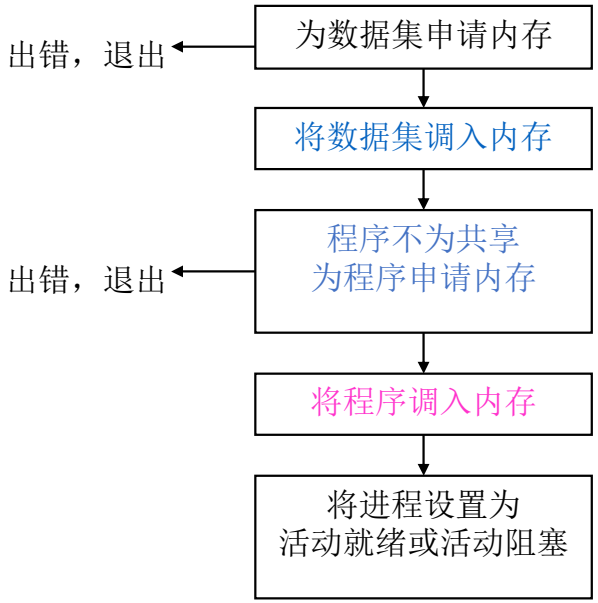


# 2.3.5 进程的挂起与激活

## 调用激活原语active

```
procedure active(i);
begin
   memallocate(i.datasize, add);
   if add = NIL then return(false);
   swapin(i.swapadd, i.datasize, add);
   i.dataadd := add;
   filestate(i.prog, add, size);
   if add = NIL then
   begin
      memallocate(size, add);
      if add = NIL then
 begin
         memrelease(i.dataadd, i.datasize);
         return(false);
      end;
     read(i.prog, size, add);
   end;
   i.textadd := add;
   if i.state =   'Readys'
   then i.state :=   'Readya'
   else  i.state :=   'Blockeda'
   return(true);
end;
```

- 原语说明
  - swapin将数据集合从外存交换区复制到内存



# 示例：linux的Shell命令方式创建和撤销进程

在 shell 命令行输入可执行文件名的方式创建进程。

用脚本方式创建进程，本质上和命名行方式差不多，都是由 shell 创建进程，提前将命名输入到一个脚本文件中，例如编辑一个 shell 脚本，第一行用于指出该脚本需要用/usr/bin/bash 来解释执行，第二行 shell 内部命令 echo 用来显示，第三行 HelloWorld 是外部命令

```
#!/usr/bin/bash
echo Running a shell script!
HelloWorld
```

- 将它保存为 shell-script，用 chmod a+x shell-scrpit 命令为它增加执行权限， ls –l 就发现有“x”执行权限
- 在提示符下输入脚本文件 名以运行该脚本程序，可以看出里面创建并运行了 HelloWorld 进程

命令行上用 kill 命令撤销进程， kill 命令的输入需要进程号，通常先用ps命令查看到进程的进程号 PID，然后再用“kill -PID”命令杀死 PID 指定进程

killall 用于杀死指定进程名的所有进程——它不是用 PID（例如“killall httpd”）

Kill和killall都是通过发送信号给指定的进程，这些信号可以通过 kill –l 或killall-l 列出。其中 9 是无条件终止进进程且不能被进程所忽略

# 示例：使用C 语言库函数fork()创建子进程

fork()函数的头文件是 unistd.h，函数原形为“int fork( void );”

fork 函数被成功调用后将安照父进程的样子复制出一个完全相同的子进程。

- 父进程 fork()函数结束时的返回值是子进程的 PID 编号。新创建的子进程则也是从 fork()返回处开始执行，但不同的是它获得的返回值是 0。
- 父子进程执行上几乎相同，唯一区别是子进程中返回 0 值而父进程中返回子进程 PID。
- 如果 fork()出错则只有父进程获得返回值-1 而子进程未能创建。

子进程是父进程的副本，它将获得父进程数据空间、堆、栈等资源的相同副本。

- 子进程持有的是上述存储空间的“副本”，父子进程间不共享这些存储空间，它们之间共享的存储空间只有代码段

**代码 2-2 fork()示例代码 fork-demo.c**

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

int main(int argc, char ** argv )
{
        int pid = fork();
        if(pid == -1 ) {
                printf("error!\n");
        } else if( pid ==0 ) {
                printf("This is the child process!\n");
                getchar();
        } else {
                printf("This is the parent process! child process id = %d\n", pid);
                getchar();
        }
        return 0;
}
```

**屏显 2-7 fork-demo 的输出**

```
[lqm@localhost ~]$ gcc fork-demo.c -o fork-demo
[lqm@localhost ~]$ fork-demo
This is the parent process! child process id = 14411
This is the child process!
```

然后用 ps 命令和 pstree 命令查看其父子信息，屏显 2-8 表明两个 fork-demo 的进程号分别是 154422 和 1543，其中 15422 进程是 15423 的父进程。

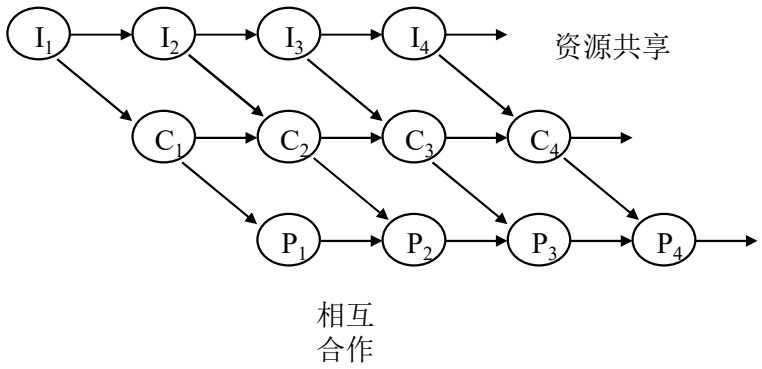
**屏显 2-8 fork-demo 的 ps/pstree 输出**

```
[lqm@localhost ~]$ ps j
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
...
 4066 15422 15422  4066 pts/3    15422 S+    1000   0:00 fork-demo
15422 15423 15422  4066 pts/3    15422 S+    1000   0:00 fork-demo
 4030 15424 15424  4030 pts/2    15424 R+    1000   0:00 ps j
[lqm@localhost ~]$ pstree 15422
fork-demo———fork-demo
```

# 2.4.1 进程同步的基本概念

## 进程之间的同步

- I：输入进程　C：计算进程　P：输出进程
- 相互合作关系（直接相互制约）
  (I→C→P)
- 资源共享关系（间接相互制约）
  ($I_1$→ $I_2$→ $I_3$→)




## 临界资源

- 临界资源是一个时刻只能由一个进程使用的资源
- 硬件资源：许多都属于临界资源，如打印机，磁带机等
- 软件资源：如变量、表格、队列等

## 临界资源使用方法

- 对临界资源的使用采用互斥方式
- 互斥：一个进程使用完之后，另一个进程才能使用

# 2.4.1 进程同步的基本概念

- **假设系统有n个进程，每个进程有一段代码。称为临界区**
  - 进程在执行该区时可能修改公共变量、更新表、写文件等
  - 当一个进程在临界区执行时，其他进程不允许在它们的临界区内同时执行
- **临界区**，进程中访问临界资源的代码段
- **同类临界区**，同一临界资源的不同进程中的临界区
- **进入区**，临界区前检查临界资源使用情况的代码段，
- **退出区**，临界区后面恢复临界资源访问标志的代码段

- **临界资源使用的同步准则**
  - 空闲让进：（提高效率）
  - 忙则等待：（解决互斥）
  - 有限等待：等待进入临界区的要求应在一有限时间满足（以免死等）
  - 让权等待：放弃占用CPU（以免忙等）
- **临界资源的状态判断与设置**
  - 临界资源的状态设置和修改必须是原子操作

# 2.4.1 临界区问题的通用结构和算法

## 临界区问题是设计一个协议来协同进程

- 在进入临界区前，每个进程应请求许可，实现这一请求的代码区段称为**进入区。**
- 临界区之后有**退出区**，然后是其他代码**剩余区**

**进程Pi的通用结构**

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

**进程Pi的算法（Pj表示另一个进程j=1-i）**

```
do {
    while (turn == j);
        critical section
    turn = j;
        remainder section
} while (true);
```

# 2.4.2 硬件同步

- 目前的许多系统为实现临界区代码提供了硬件指令支持。
- 所有的解决方案都是基于锁的思路
  - 就是通过锁保护临界区
- **单CPU**–可以禁中断
  - 当前运行的代码无需抢占即可执行
  - 通常在多处理器系统上效率太低
    - 操作系统使用此方法扩展性不大
  - 关中断的方法不适于多CPU环境
    - 关闭所有CPU中断耗时多，时钟系统紊乱
- 现代机器提供特殊的原子硬件指令
    - 原子=不间断
  - 测试记忆字和设定值
  - 或交换两个记忆字的内容

- 采用**互斥锁**的临界区问题的解决方案

```
do {
        acquire lock 获得锁
                critical section
        release lock 释放锁
                remainder section
} while (TRUE);
```

# 2.4.3 记录型信号量机制

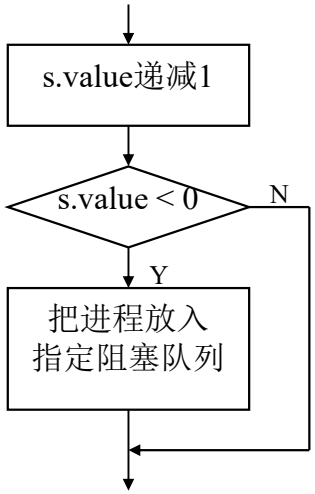
- 信号量变量是由一个整型数和一个阻塞等待进程链表构成的记录型数据结构

```
typedef struct {
    int value;
    struct process_control_block *list;
} semaphore;
```

- 所有对同一个信号量进行操作且进入等待状态的进程，都自动进入阻塞等待（让权等待）状态，并将其挂在该信号量阻塞等待队列L中

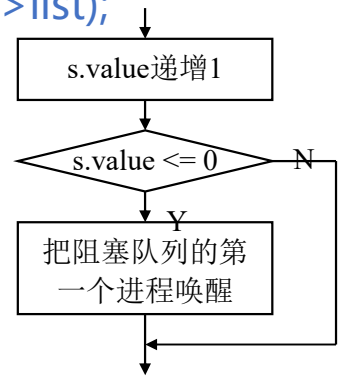
- **P原子操作（wait）**

```
wait(semaphore *s){
    s->value--;
    if (s->value<0)  block(s->list);
}
```




- **V原子操作（signal）**

```
signal(semaphore *s){
    s->value++;
    if (s->value <= 0) wakeup(s->list);
}
```



# 2.4.3无忙等待的记录性信号量的代码实现——续

- **每个信号量都有一个相应的等待队列**
- 等待队列中的每一项，都有两个数据项:
  - 值 (整形)
  - 指向下一个记录的指针
- 有两个操作:
  - **阻塞block** – 把激活的进程放到适当的等待队列
  - **唤醒wakeup** – 从等待队列中移出一个进程，并放到就绪队列中

```
typedef struct{
  int value;
  struct process *list;
  } semaphore;
```

```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();
   }
}
signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```

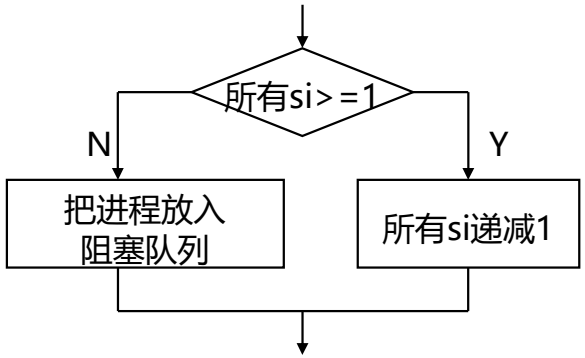
# 2.4.3 AND型信号量机制

将进程运行过程中需要的所有临界资源，一次性地全部分配给进程（即要么全部分配，要么一个也不分配）

进程使用完后再一起释放

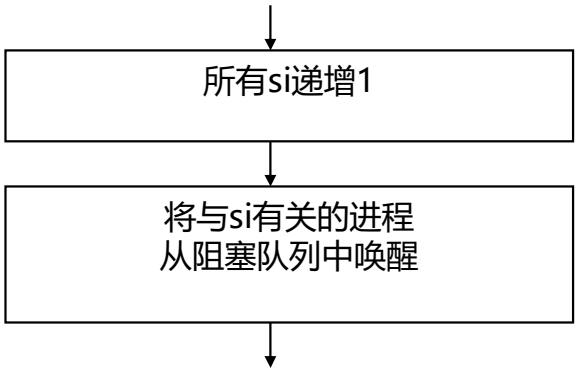
## P原子操作（Swait)

```
Swait(s1, s2, ..., sn) {
   while(TRUE) {
      if(si>=1 && ....&& sn>=1) {
         for (i=1;i<=n;i++) si=--;
         Break;
      }
      else {
         /*把进程放入阻塞队列*/
      }
   }
}
```




## V原子操作（Ssignal)

```
Ssignal(s1, s2, ..., sn) {
   while(TRUE) {
      for (i=1;i<=n;i++) {
         si=++;
         /*将与si有关的进程从阻塞队列中唤醒*/
      }
   }
}
```



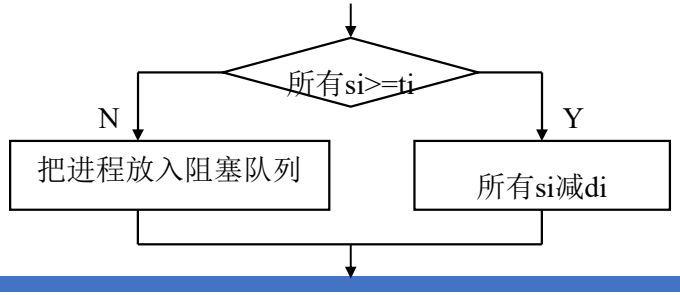
# 2.4.3 信号量集机制

## 一般信号量集机制

- 进程一次运行需要多个（N个）同类临界资源
- 安全起见，现有的资源数量要大于某个（t个）临界资源数量才准予分配

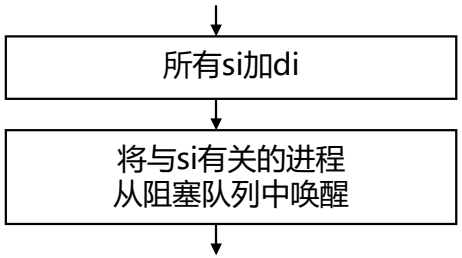
## P原子操作（Swait)

```
procedure Swait(s1, t1, d1; ...; sn, tn, dn);
var s1, s2, ..., sn: semaphore;
begin
   if  (s1 >= t1) and ... and (sn >= tn)
   then  {for i := 1 to n do  si := si – di;
            return true; }
   else  {将程序计数器设置为本函数开始处;
           将进程插入阻塞队列; }
end;
```




## V原子操作（Ssignal)

```
procedure Ssignal (s1, d1; ...; sn, dn);
var s1, s2, ..., sn: semaphore;
begin
   for i := 1 to n do  si := si + di;
   将与si有关的进程从阻塞队列中唤醒
end;
```




## 一般信号量集特例

- Swait(s, d, d):　一个信号量，每次同时分配d个同类资源
- Swait(s, 1, 1):　等效于记录型信号量
- Swait(s, 1, 0):
  - s=1, 允许多个进程进入特定区；
  - s=0, 阻止任何进程进入特定区

# 2.4.4 信号量机制实现前驱关系示例

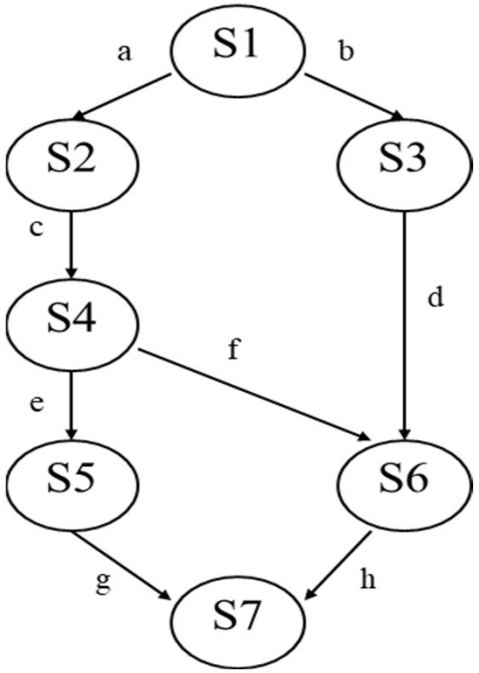
## 利用信号量实现较复杂的进程同步

S1… S7分别为进程P1…P7的执行部分，它们的执行顺序如右图所示

```
semaphore a, b, c, d, e, f, g, h
P1() { S1; signal(a); signal(b); }
P2() { wait(a); S2; signal(c); }
P3() { wait(b); S3; signal(d); }
P4() { wait(c); S4; signal(e); signal(f); }
P5() { wait(e); S5; signal(g); }
P6() { wait(f); wait(d); S6; signal(h);}
P7(){ wait(g); wait(h); S7; }
Main(){
 a.value=b.value= c.value= d.value= e.value=
f.value= g.value= h.value= 0;
Cobegin
                p1();p2(); p3(); p4(); p5() p6(); p7();
Coend
```

## 信号量机制的特性

- wait(s)操作和signal(s)操作必须成对出现(可不在同一进程中)
- 缺少wait(s)不能保证资源互斥使用
- 缺少signal(s)将可能使资源永远得不到释放
- 不存在“忙等”问题



# 2.4.5 管程机制

## 信号量机制的一些问题

- 临界区分散在各进程之中，不便于管理和控制
- 很多临界区的操作是相同的，重复编写，使进程结构不清晰
- 如果编程出现差错，不便于检查，且会带来严生后果

## 管程的定义

- 一个管程定义了一个**数据结构**和能为并发进程所执行（在该数据结构上）的一组**操作**，这组操作能同步进程和改变管程中的数据
- 管程实际上是一种能实现进程同步的特殊的子程序（函数、过程）的集合

## 管程的组成

- 名称：该管程的标识
- 共享变量说明：局部于管程的变量说明（包括特殊同步变量即条件变量）
- 一组过程：对该数据结构进行操作的程序段（相当于临界区代码段）
- 初始化：对局部于管程的数据设置初始值

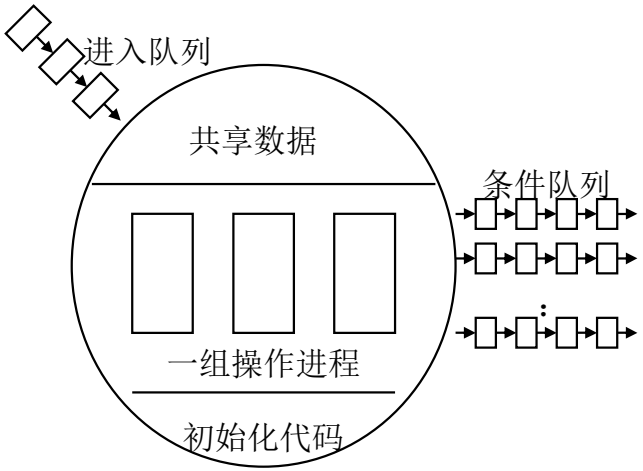
## 管程的语法结构

```
Type 管程名称 = monitor
   共享变量说明语句
   procedure entry P1(...)
      begin   ...   end;
   procedure entry P1(...)
      begin   ...   end;
         :
   begin
      初始化语句
   end;
```

# 2.4.5 管程机制

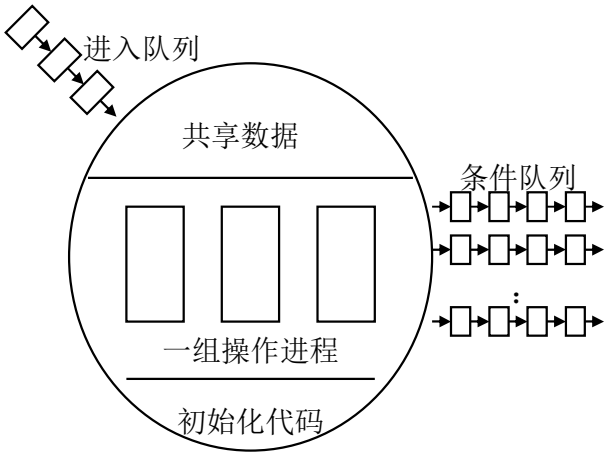
## 进程的互斥访问

- 进程访问临界资源，必须经过管程
- 管程每次只允许一个进程进入
- 即只要某个进程进入了管程，则要么执行完成，要么因故阻塞，然后下一个进程才可以进入管程




## 进程的同步

- 设置条件变量(相当于互斥信号量)
- var x: condition
- 当临界资源已被占用，执行x.wait，将进程挂在x条件变量的阻塞等待队列中
- 当临界资源已空闲，执行x.signal，从x条件变量的阻塞等待队列中，唤醒第一个进程



# 示例：Linux的信号量

Linux 支持 System V IPC 中的信号量集和 POSIX 信号量。

- 前者常用于进程间通信、是基于内核实现的（不随进程结束而消失）；
- 后者是常用于线程间同步、方便使用且仅含一个信号量。
- POSIX 有名信号量和一个文件的路径名相关联，创建后不随进程结束而消失（可用于进程间通信）
- POSIX无名信号量只在进程生命周期内，只能在该进程创建的线程间使用
- 所有 System V IPC信号量函数名字里面没有下划线（例如 semget()），POSIX 信号量函数都下划线（例如 sem_post()）。
- Linux内核内部也有多个并发的执行流，使用内核的信号量，和用户态信号量又不相同。

# 示例：Linux的信号量

POSIX有名信号量的创建使用sem_open()，函数原型如下：

```
#include <semaphore.h>

sem_t *sem_open(const char *name,int oflag,mode_t mode,unsigned int value);
```

sem_wait()来执行 V 操作（减 1）

sem_getvalue()来查看信号量的值

sem_post(sem)对信号量执行 P 操作（增 1）

POSIX无名信号量使用 sem_init()创建，函数原型如下：

```
#include int sem_init(sem_t *sem, int pshared, unsigned int value);
```

互斥量是信号量的一个退化，仅用于并发任务间的互斥访问；

```
mutex m; //增加一个互斥量

pthread_mutex_init(&m,NULL); //初始化互斥量

pthread_mutex_lock(&m); //进入临界区 pthread_mutex_unlock(&m); //推出临界区
```

# 2.5.1 有界缓冲区的同步结构

***n*** 个缓冲区，每个缓冲区可存一个数据项，设以下3个信号量

Semaphore **mutex** initialized to the value 1

Semaphore **full** initialized to the value 0

Semaphore **empty** initialized to the value n

wait(s){while(s<=0);s--;}，signal(s){s++;}

## 生产者进程结构

```
do {
        ...
      /* produce an item in
next_produced */
        ...
      wait(empty);
      wait(mutex);
        ...
      /* add next produced to the
buffer */
        ...
      signal(mutex);
      signal(full);
  } while (true);
```

## 消费者进程结构

```
Do {
      wait(full);
      wait(mutex);
        ...
      /* remove an item from buffer to
next_consumed */
        ...
      signal(mutex);
      signal(empty);
        ...
      /* consume the item in next
consumed */
        ...
  } while (true);
```

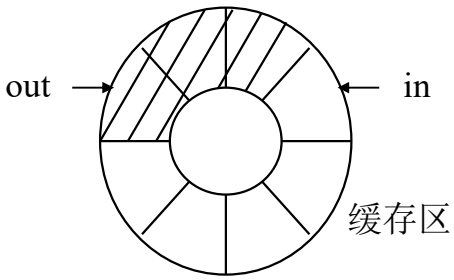
# 2.5.1 生产者消费者同步问题的提出

## 生产者 - 消费者问题描述

- 有一群生产者进程在生产消息，并将此消息提供给另外一群消费者进程去消费
- 生产者进程（多个），产品存放缓冲区（n个），消费者进程（多个）

## 生产者 - 消费者问题举例

- 数据输入进程（生产者）与计算进程（消费者）
- 数据计算进程（生产者）与数据输出（如打印）进程（消费者）

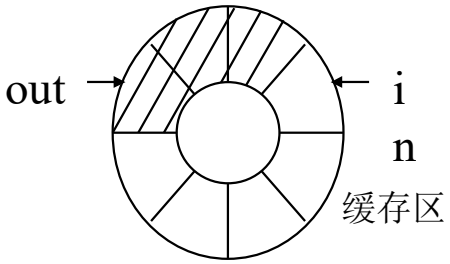



```
Var n, integer;
type item= … ;
var buffer:array [0, 1, …, n-1] of item;
in, out: 0, 1, …, n-1;  //in 填入位置，out取出位置
counter: 0, 1, …, n;  //缓冲区中产品数量

producer: repeat
    …
    produce an item in nextp;
    …
    while counter=n do no-op;
    buffer [in] :  = nextp;
    in:  =in+1 mod n;
    counter:  =counter+1;
   until false;

consumer: repeat
    while counter=0 do no-op;
    nextc:  =buffer [out] ;
    out:  = (out+1) mod n;
    counter:  =counter-1;
    consumer the item in nextc;
   until false;
```

# 2.5.1 用一个信号量的生产者消费者同步

## 生产者 - 消费者关系

- 公用信号量**mutex**：初值为1，实现临界资源（缓冲区）互斥使用。(如果缓冲区只有一个缓冲块，可以只用这一个信号量)
- 生产者私用信号量**empty**：初值为n，指示空缓冲块数目
- 消费者私用信号量**full**：初值为0，指示满缓冲块数目
- 整型量in, out：初值均为0，in指示首空缓冲块序号，out指示首满缓冲块序号




## 用一个信号量的生产者-消费者进程描述

```
var mutex: semaphore := 1
    buffer: array[0, ..., n-1] of item;            定义变量
    in, out: integer: =0, 0        // in == out 缓冲区空，
(in+1)%n == out 缓冲区满

procedure producer;            生产者进程
begin
   repeat
            生产一个产品nextp;
pLoop:   wait(mutex);
            if ((in+1)%==out) {signal(mutex); goto pLoop;}
            buffer(in) : = nextp;
            in := (in + 1) mod n;
           signal(mutex);
   until false;
end;

procedure consumer;                消费者进程
begin
   repeat
cLoop:    wait(mutex);
            if (in == out) {signal(multex); goto cLoop;}
             nextc := buffer(out);
             out := (out + 1) mod n;
             signal(mutex);
             消费一个产品nextc;
   until false;
end;
```

**存在“忙等”问题**

# 2.5.1 用多个信号量的生产者消费者同步

## 多信号量的生产者-消费者进程描述

```
var mutex, empty, full: semaphore := 1, n, 0
    buffer: array[0, ..., n-1] of item;     定义变量
    in, out: integer: =0, 0;

procedure producer;                 生产者进程
begin
   repeat
      生产一个产品nextp;
      wait(empty);    wait(mutex);
      buffer(in) : = nextp;
      in := (in + 1) mod n;
      signal(mutex);    signal(full);
   until false;
end;

procedure consumer;                 消费者进程
begin
   repeat
      wait(full);    wait(mutex);
      nextc := buffer(out);
      out := (out + 1) mod n;
      signal(mutex);    signal(empty);
      消费一个产品nextc;
   until false;
end;
```

## 采用信号量应注意的问题

- 公用（互斥）信号量（mutex）和私用信号量（empty, full）都必须成对出现
- 在进入临界区之前，先对私用信号量（empty, full）进行wait操作，再对公用信号量进行wait操作（不能颠倒，否则易于造成“死锁”）

# 2.5.1 采用AND信号量的生产者-消费者同步

```
var mutex, empty, full: semaphore := 1, n, 0
    buffer: array[0, ..., n-1] of item;
```

定义变量

```
    in, out: integer: =0, 0;

procedure producer;            生产者进程
begin
   repeat
      生产一个产品nextp;
      Swait(empty, mutex);
      buffer(in) : = nextp;
      in := (in + 1) mod n;
      Ssignal(mutex, full);
   until false;
end;
```

```
procedure consumer;      消费者进程
begin
   repeat
      Swait(full, mutex);
      nextc := buffer(out);
      out := (out + 1) mod n;
      Ssignal(mutex, empty);
      消费一个产品nextc;
   until false;
end;
```

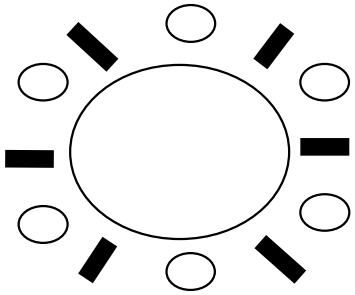
# 2.5.2 哲学家就餐问题描述

## 哲学家进餐问题描述

- 六个哲学家，围着圆桌交替地进行思考和进餐；
- 每次进餐时，必须同时拿到左右两边的两只筷子才能进餐；
- 进餐后，再放下筷子继续思考。

## 哲学家进餐问题所用信号量

- 这是一个典型的同时需要两个资源的例子
- 采用矩阵型信号量chopstick[i]（i=0…5）
- 当chopstick[i]和chopstick[i+1]两个临界资源都取得时，第i个进程（哲学家）可以执行（进餐）



## 采用信号量的哲学家进餐进程描述

var chopstick: array[0,…,5]: semaphore := 1, 1, 1, 1, 1, 1;　　//定义变量

procedure Process;　//哲学家进程
begin
　wait(chopstick[i]);
　wait(chopstick[(i+1)%6]);
　吃饭……
　signal(chopstick[i]);
　signal(chopstick[(i+1)%6]);
end;

## 采用信号量应注意的问题

- 相邻的哲学家最好不要同时拿同方位的筷子
- 至少有一个哲学家与其他哲学家第一次拿的筷子方位不同

# 2.5.2 解决哲学家就餐死锁问题的方法

## 采用信号量的哲学家进餐进程描述

var chopstick: array[0,...,5]: semaphore := 1, 1, 1, 1, 1, 1;　　//定义变量

```
procedure OddProcess;   //奇数哲学家进程
begin
    wait(chopstick[i]);
    wait(chopstick[(i+1)%6]);
    吃饭……
    signal(chopstick[i]);
    signal(chopstick[(i+1)%6]);
end;
```

```
procedure EvenProcessr; //偶数哲学家进程
begin
    wait(chopstick[(i+1)%6]);
    wait(chopstick[i]);
    吃饭……
    signal(chopstick[(i+1)%6]);
    signal(chopstick[i]);
end;
```

## 采用AND信号量解决哲学家进餐问题

var chopstick: array[0, ..., 5] of semaphore := 1, 1, 1, 1, 1, 1;　　//定义变量

```
procedure i;
begin
    repeat
        思考;
        Swait(chopstick[i], chopstick[(i+1) % 6]);
        进餐;
        Ssignal(chopstick[i], chopstick[(i+1) % 6]);
    until false;
end;
```

# 2.5.3 读者写者问题描述

## 读者 - 写者问题描述

- 一个数据文件或记录被多个进程共享，多个只读进程（reader）可以同时访问共享对象，而改写进程（writer）必须与其它进程（包括reader, writer）互斥地访问共享对象的同步问题

## 读者 - 写者问题举例

- 数据库数据的读（读者）操作、写（写者）操作
- 民航订票：查询订票信息（读者）操作，售票人员售出机票（写者）操作

## 读者 - 写者关系

- 互斥信号量wmutex：初值为1，实现Reader与Writer进程间在读或写时的互斥。只要有一个Reader进程在读，便不允许Writer进程去写。
- 读者数目：readcount，初值为0，表示正在读的进程数目
  - 如果readcount=0，读者进程须执行wait(wmutex)，Writer进程可能正在写
  - 读者操作完成之后，如果readcount-1=0，读者进程必须执行signal(wmutex)操作
- 读操作信号量rmutex：初值为1，多个Reader访问，需要对readcount的互斥访问

# 2.5.3 读者-写者问题的同步结构

- 定义变量

```
semaphore rmutex=1, wmutex=1;

int  readcount =0;
```

**The structure of a writer process**

```
do {
    wait(w_mutex); //写者互斥信号量

      ...
    /* writing is performed */

      ...

    signal(w_mutex);

 } while (true);

void main() {
    cobegin
      Reader();Writer();
    coend
```

**The structure of a reader process**

```
do {
   wait(rmutex);    更新read_count互斥
   if (read_count == 1) wait(w_mutex);

   read_count++;    正在读的进程数量
   signal(rmutex);

          ...
   /* reading is performed */

          ...

   wait(rmutex);
   read count--;
   if (read_count == 0) signal(w_mutex);

   signal(rmutex);

} while (true);
```

采用信号量应注意的问题

写（互斥）信号量（wmutex）必须成对出现

在对readcount（所有读者进程共用的临界资源）进行处理之前，必须先进行wait(rmutex)操作；

处理完之后（+1或-1），再执行signal(rmutex)操作

# 2.5.3 采用一般信号量集的读者写者问题

- 采用一般信号量集解决读者 - 写者问题

- var rmutex, wmutex: semaphore := RN, 1 ; 定义变量

```
procedure writer;                    写者进程
  begin
     repeat
          Swait(wmutex, 1, 1; rmutex, RN, 0);
           执行写操作;
           Ssignal(wmutex, 1);
       until false;
  end;
```

```
procedure reader;            读者进程
     begin
        repeat
           Swait(rmutex, 1, 1);
           Swait(wmutex, 1, 0);
           执行读操作
           Ssignal(rmutex, 1);
        until false;
     end;
```

# 2.6.1 进程通信的类型——共享存储器系统

**进程通信的三种类型：**

- 共享存储器系统（无格式）
- 消息传递系统（有格式）
- 管道通信系统（相当于文件）

**进程通信（同步）的三种方式：**

- 发送进程阻塞，接收进程阻塞(没有缓冲区)
- 发送进程不阻塞,接收进程阻塞(如后台打印)
- 发送进程不阻塞，接收进程不阻塞(有多个缓冲区)

**希望通信的进程之间共享的内存区域**

- 主要问题是，当用户进程存取共享内存时，提供容许用户进程同步它们行动的机制

**进程间以共享存储器方式通信**

基于共享数据结构的通信方式

- 由用户程序定义数据结构、申请内存，并同步各进程对该数据结构的访问
- 在用户进程的控制下进行通信，而不是操作系统，如生产者—消费者问题

基于共享存储区的通信方式

- 系统设置一共享存储区，由系统同步各进程对该共享存储区的访问
- 用户需要共享存储区时，到系统中申请，系统分配部分共享存储分区给用户并返回该分区的名字，相关进程按名共享该分区

# 2.6.2 直接消息传递通信

- 进程间的数据交换以消息为单位
- OS直接提供一组命令（原语）实现通信

## 直接消息传递通信的特点：

- 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

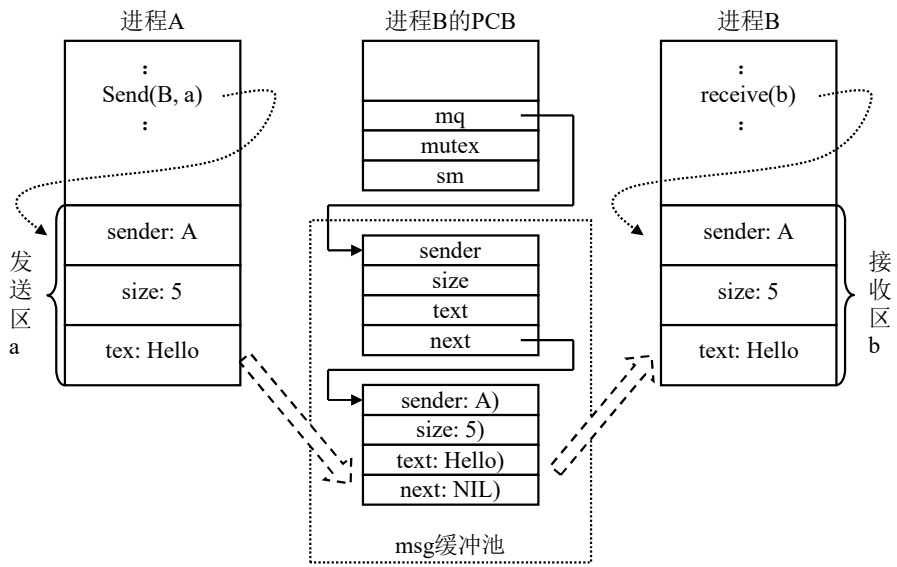
## 消息通信命令

- 直接消息传递通信主要采用两种命令（原语）
- 发送消息原语Send(Receiver, message)
- 接收消息原语Receive(message)

## 直接通信机构

- mq：进程的消息队列首指针
- mutex：进程的消息队列互斥信号量，初值为1
- sm：同步信号量，用于消息队列中的消息计数，初值为0

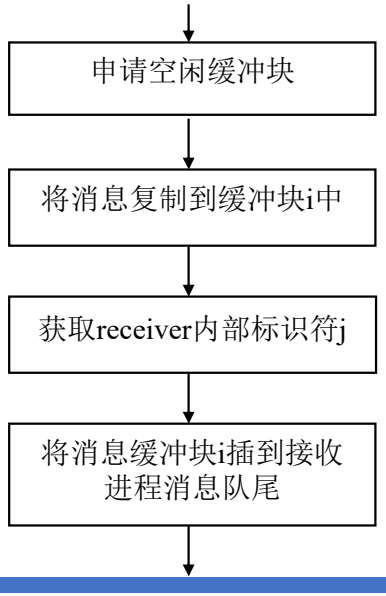
## 直接通信示意图



# 2.6.2 直接消息传递通信的同步原语实现

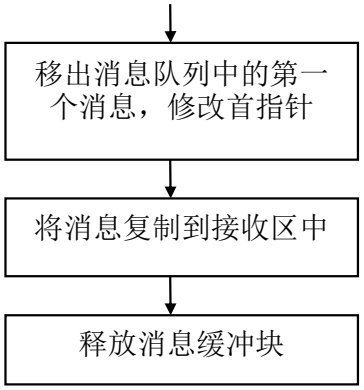
## 直接通信Send原语

```
procedure Send(receiver, a)
begin
   getbuf(a.size, i);
   i.sender = a.sender;  i.size = a.size;
   i.text = a.text;          i.next = NIL;
   getid(PCB set, receiver, j);
   wait(j.mutex);
   insert(j.mq, i);
   signal(j.mutex);
   signal(j.sm);
end;
```




## 直接通信Receive原语

```
procedure Receive(b)
begin
   j = internal name;
   wait(j.sm);  wait(j.mutex);
   remove(j.mq, i);
   signal(j.mutex);
   b.sender = i.sender ; b.size = i.size;
   b.text  = i.text;
   putbuf(i);
end;
```



# 2.6.3 间接消息传递通信

## 1、特点：

- **需要某种共享数据结构的实体作为中介**
- 发送进程将发送给目标进程的消息暂存于该中介中
- 接收进程从中介中，取出发送给自己的消息
- 中介一般称为信箱(mailbox)，因此，间接消息传递通信也称信箱通信

## 2、信箱通信命令

- 信箱创建命令Create(mailbox)
- 信箱撤消命令Delete(mailbox)
- 消息发送命令Send(mailbox, message)
- 消息接收命令Receive(mailbox, message)

## 3、信箱种类

- 私用信箱：由用户进程创建，其它进程只能向该信箱发送消息
- 公用信箱：由操作系统创建，并提供给系统中的所有核准进程使用
- 共享信箱：由某进程创建，并指明共享者名字；所有共享者和创建者进程都可以取走自己的消息

## 4、发送进程与接收进程的关系

- 一对一：在发送与接收进程之间设立专用通信链路
- 多对一：客户/服务器方式
- 一对多：一个发送进程向多个接收进程广播消息
- 多对多：设立一个公用信箱

# 2.6.4 消息传递系统的同步与缓冲

- 传递的消息可以是阻塞的或者非阻塞的
- 阻塞对应于同步
  - 阻塞发送 – 发送者被阻塞，直到消息被接收
  - 阻塞接受 – 接收者被阻塞，直到消息可用
- 非阻塞对应于异步
  - 非阻塞发送 – 发送器发送消息并继续
  - 非阻塞接受 – 接收器接收:
    - 有效消息，或无消息
- 不同的组合可能
  - 如果收发都被阻塞，就会有一个交会

- **生产者和消费者问题的同步变得很简洁**

```
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 */
 }
```

- **链路中附加的消息队列以某种方式实施**
  - 1.零容量 – 链路中没有消息在等待.
    发送者必须等待接收者收到消息
  - 2.有限容量 – 消息队列长度为有限的n
    如果链路已满，发送者应被阻塞
  - 3.无限容量 – infinite length
    发送者从不阻塞

# Linux 的进程通信System V IPC

System V IPC 指的是 引入的三种进程间通信工具:

(1)信号量，用来管理对共享资源的访问

(2)共享内存，用来高效地实现进程间的数据共享

共享内存是由内核提供的一段内存，可以映射到多个进程的续存空间上，从而通过内存上的读写操作而完成进程间的数据共享

(3)消息队列，用来实现进程间数据的传递。

执行 ipcs 命令可以查看到当前系统中所有的 System V IPC 对象

```
--------- 消息队列 ----------
键        msqid      拥有者  权限     已用字节数 消息

------------ 共享内存段 -------------
键        shmid      拥有者  权限     字节     nattch     状态
0x00000000 131072     lqm       600       524288     2          目标
0x00000000 163841     lqm       600       4194304    2          目标
0x00000000 327682     lqm       600       4194304    2          目标
0x00000000 524292     lqm       600       2097152    2          目标

--------- 信号量数组 ----------
键        semid      拥有者  权限     nsems
```

# 示例：消息队列 & 共享内存

创建消息队列，收发消息

```
/* Open the queue - create if necessary */
if((msgqueue_id = msgget(key, IPC_CREAT|0660)) == -1)

switch(tolower(argv[1][0]))
{
   case 's': send_message(msgqueue_id,(struct mymsgbuf*)&qbuf,atol(argv[2]),argv[3]);

   break;
   case 'r': read_message(msgqueue_id, &qbuf, atol(argv[2]));
   break;
   case 'd': remove_queue(msgqueue_id);
   break;
   case 'm': change_queue_mode(msgqueue_id, argv[2]);
   break;
```

某进程创建共享内存
另外的进程映射使用

```
shm_id=shmget(IPC_PRIVATE, BUFSZ, 0666 ) ;     //创建共享内存
system( "ipcs -m"); //执行 ipcs -m 命令，显示系统的共享内存信息

if ( (shm_buf = shmat( shm_id, 0, 0)) < (char *) 0 ){   //映射共享内存到进程空间
system("ipcs -m");                                      //显示共享内存信息
strcpy(shm_buf,"Hello shared memory!\n");
 if ( (shmdt(shm_buf)) < 0 ) {                          //解除共享内存的映射
```

# 管道

进程间的管道通信有两种形式，无名管道用于父子进程间，

命名管可以用于任意进程间— —命名管道在文件系统中有可访问的路径名。

管道通信方式用于单向通信，如果需要双向 通信则建立两条相反方向的管道。

管道实质是由内核管理的一个缓冲区（一边由进程写入，另 一边由进程读出），因此要注意，如果缓冲区满了则写管道的进程将会阻塞。

另外管道内部没 有显式的格式和边界，需要自行处理消息边界，如果多进程间共享还需要处理传送目标等工作。

C 库中的用户态函数 popen()和 pclose()对 pipe()系统调用进行了封装。

pipe()父子进程间使用管道进行通信 的方法，pipe()将通过两个文件描述符（整数）来指代管道缓冲区的读端和写端（代码中用 fds[] 变量记录）。

父进程关闭管道的读端 fds[0]并往管道的写端 fds[1]写出信息，子进程关闭 了管道的写端 fds[1]并从管道的读端 fds[0]读回信息。

# 2.7.2 线程与进程的关系

- 在传统OS中，进程既是CPU调度和分配的基本单位，同时，进程又是拥有资源的独立实体
- 在处理（创建、撤消、调度）进程时，所费开销较大
- 在OS中引入进程，主要是为了提高计算机系统的并发执行能力
- 将进程的两个属性分开，即让进程仅成为拥有资源的单位
- 而让线程成为调度和执行的基本单位，是为了进一步提高并发能力，并有利于多处理机系统的调度
- 线程是进程中的运行实体
- 一个进程可包含多个线程
- 一个进程中至少包含一个线程，称主线程
- 进程相当于线程的载体

# 2.7.3 线程的状态和线程控制块

## 多线程OS中的线程

- 作为系统资源分配的单位
- 可包括多个线程
- 进程不是一个可执行的实体

## 线程的属性

- 轻型进程：线程只拥有程序计数器、堆栈、TCB
- 独立调度和分派的基本单位：线程切换开销少
- 可并发执行：同一进程或不同进程的线程都可并发执行
- 共享进程资源：内存空间、公用变量、信号量等

## 线程的状态

- 执行状态：线程正获得CPU而运行
- 就绪状态：一旦获得CPU就可运行
- 阻塞状态：因等待某一事件而暂停

## 线程的创建和终止

- 创建：新线程由正在执行的线程所创建
- 线程创建一般通过创建函数（或系统调用）来实现，并提供相应的参数，如指向线程主程序的入口指针，堆栈的大小，优先级等
- 终止：线程同样有生命期，当线程完成任务后，会自动终止
- 线程终止还有可能由其它线程实现

# 2.7.4 线程的分类

## 线程的分类

- 内核支持线程：线程的处理均由内核实现，内核为线程保留TCB，并通过TCB感知线程
- 用户级线程：线程的处理不通过内核实现，线程的状态信息等全部存放在用户空间中，内核不能感知用户级线程

## 线程同步－－互斥锁(Mutex)

- 利用开锁/关锁实现互斥访问

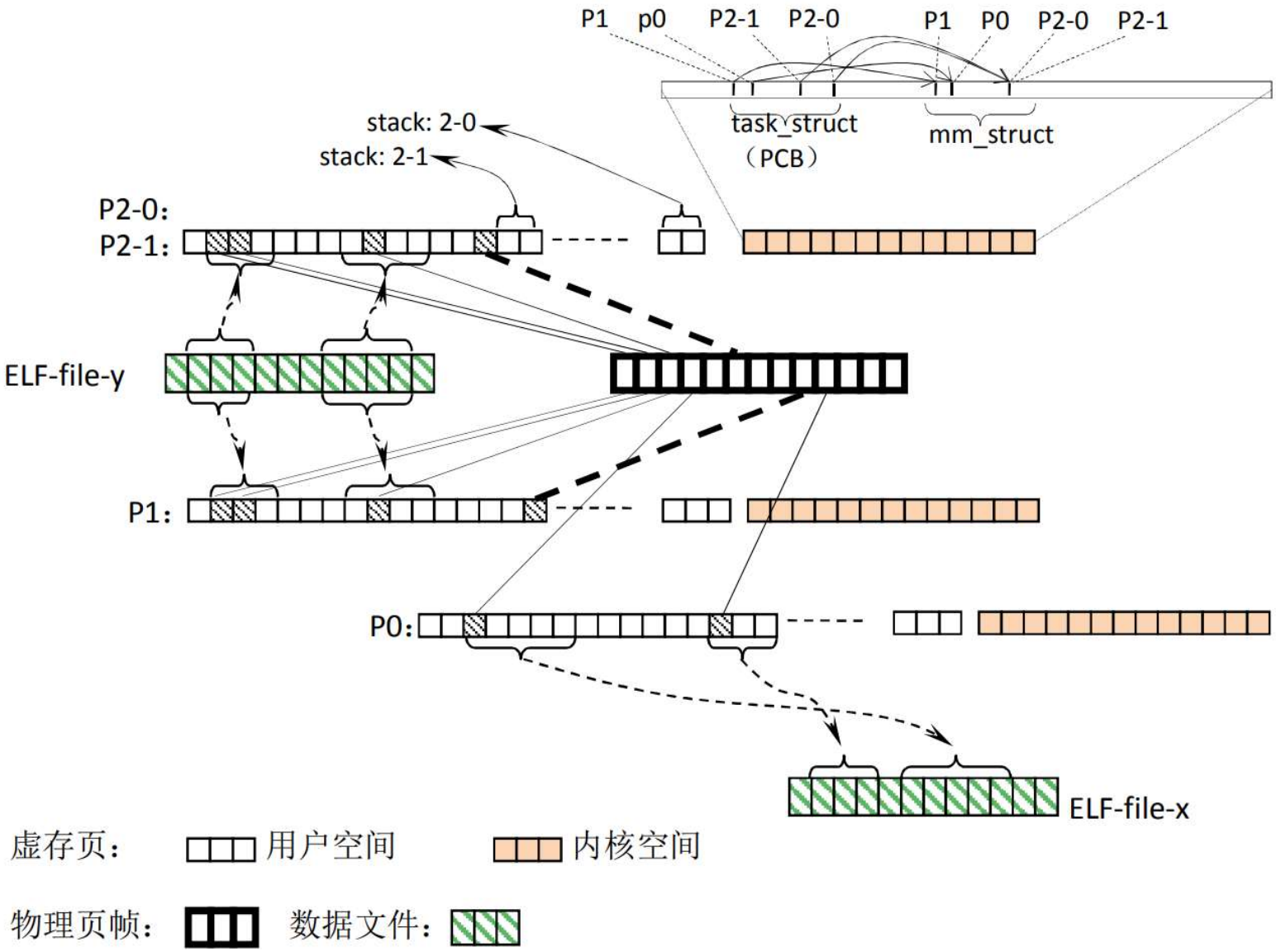
## 线程同步－－条件变量

- 在进入临界区之前，先关上互斥锁，进行判断，是否条件可以满足
- 不能满足时，自行转为等待状态，并打开互斥锁
- 满足条件，则继续执行临界区，完成后，打开互斥锁

## 线程同步－－信号量机制

- 私用信号量：用于同一进程中不同线程之间的同步和通信
- 公用信号量：用于不同进程中的线程之间的同步和通信

# 多进程、多线程并发的地址空间



# 示例：Linux的线程创建

pthread_t id;　　ret=pthread_create(&id,NULL,(void *) thread,NULL);

使用 pthread 库来创建线程，用“gcc –lpthread demo.c –o demo”完成编译并输出 demo 可执行文件，-lpthread 表示编译链接时需要用到 pthread 库。

其中函数 pthread_create()用于创建线程，第三个参数是一个函数指针——用于指定新创建线程将执行哪个函数，第一个参数用于记录新创建进程的标识 ID。

主线程使用 pthread_join(id,NULL)等待指定线程（以 id 为标识的线程，也就是前面刚创建的线程）结束。

Linux 中，/proc/slabinfo 中统计了各种内核数据对象的数量，其中也包括进程控制块的数量。

100个进程创建，运行前后对比，多 100 个 task_struct的数据对象

在进程的主线程存在的前提下创建 100 个线程，则不需要新创建 mm_struct。

创建 100 个线程，由于·共享进程空间，没增加一个 mm_struct。

每个进程需要一个独立的用户台堆栈，需要一个 独立的vm_are_struct 结构体来描述，增加 100 个 vm_area_struct 用于描述线程用户态栈。

如果师100 个进程，每个进程至少需要大约 15 个 vm_area_struct 结构体，100 个进程需要增加大约 1500 个 vm_area_struct