第二章　进程的描述与控制

前驱图定义

- 结点：描述一个程序段、进程或一条语句
- 有向边：表示两个结点之间存在的偏序(Partial Order)或前趋关系(Precedence Relation)“→”。

  →={(P_i, P_j)|P_i must complete before P_j may start}

  - 如果(P_i, P_j)∈→,可写成P_i→P_j
  - 称P_i是P_j的直接前趋，而称P_j是P_i的直接后继。
  - 没有前趋的结点称为初始结点(Initial Node)
  - 没有后继的结点称为终止结点(Final Node)。

程序顺序执行时的特征

•顺序性：

每一个操作必在前一操作结束之后才能开始。

•封闭性：

程序运行时独占全机资源，不受外界因素影响。

•可再现性：

若程序执行的环境和初始条件相同，程序重复执行时总可获得相同的结果。

程序并发执行时的特征

- 间断性：多个程序以走走停停的方式并发执行
- 失去封闭性：并发程序共享资源，互相受到影响
- 不可再现性：程序的多次重复执行可能得到不同的结果

典型的进程定义：

(1) 进程是程序的一次执行。

(2) 进程是一个程序及其数据在处理机上顺序执行时所发生的活动。

(3) 进程是程序在一个数据集合上运行的过程，它是系统进行资源分配和调度的一个独立单位。

“进程是进程实体的运行过程，是系统进行资源分配和调度的一个独立单位”。

进程实体

进程在计算机系统中的映像

- **PCB（Process Control Block）**
- **程序段**
- **相关数据段**
- **管理用的用户堆栈和系统堆栈**

进程的特征

- 动态性：进程的实质是进程实体的执行过程。因此，动态性是进程的最基本特征。
- 独立性：独立性是指进程实体是一个能独立运行、独立获得资源和独立接受调度的基本单位。
- 并发性：多个进程实体同存于内存中，且能在一段时间内同时运行；引入进程实体的目的就是并发执行
- 异步性：各进程按各自独立的、不可预知的速度向前推进

进程与程序的区别

- 进程是动态的，程序是静态的：程序是有序代码的集合；进程是程序的执行。通常进程不可在计算机之间迁移；而程序通常对应着文件、静态和可以复制
- 进程是暂时的，程序的永久的：进程是一个状态变化的过程，程序可长久保存
- 进程与程序的组成不同：进程的组成包括程序、数据和进程控制块（即进程状态信息）
- 进程与程序的对应关系：通过多次执行，一个程序可对应多个进程；通过调用关系，一个进程可包括多个程序

进程控制块（Process Control Block，PCB）

概念：
系统为了管理进程设置的一个专门的数据结构，用它来记录进程的外部特征，描述进程的运动变化过程（又称进程描述符、进程属性）

功能：
系统利用PCB来控制和管理进程
PCB是系统感知进程存在的唯一标志

进程与PCB是一一对应的

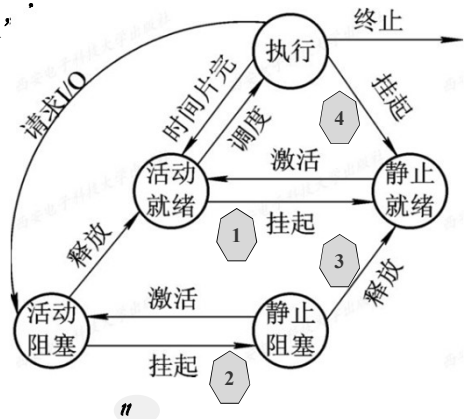
PCB组织方式（2）

- 链接结构：同一状态进程的PCB组成一个链表，不同状态对应多个不同的链表（就绪链表、阻塞链表）
- 索引结构：对具有相同状态的进程，分别设置各自的PCB索引表，表明PCB在PCB表中的地址

进程队列：不同状态进程分别组成队列
运行队列、就绪队列、等待队列

■引入挂起状态后，
原状态称为“活动”，
挂起后称为“静止”

■两种挂起状态：
静止就绪
静止阻塞



操作系统内核支撑功能

(1) 中断处理
中断处理是内核最基本的功能，是整个操作系统赖以活动的基础。
(2) 时钟管理
(3) 原语操作(原子操作)
由若干条指令组成的，用于完成一定功能的一个过程。原语在执行过程中不允许被中断。在系统状态下执行，常驻内存。

进程的创建

1. 进程的层次结构

OS中允许一个进程创建另一个进程

创建进程的进程称为父进程

被创建的进程称为子进程

子进程可继续创建孙进程

UNIX中，进程与其子孙进程共同组成一个进程家族(组)

3. 引起创建进程的事件

内核创建

用户登录

分时系统的用户在终端登录后，如是合法用户，系统将为其创建一个进程，并插入就绪队列

作业调度

在批处理系统中，当作业调度程序调度到某作业时，将该作业装入内存，为它分配资源并创建进程

提供服务

当运行中的用户进程提出某种请求后，系统将专门创建一个进程来提供服务，如打印

应用请求

由应用程序为自己创建进程，以便能并发执行，如输入、计算、输出程序

进程的终止(Termination of Process)

1、正常

进程已经运行完成

2、异常

① 越界错误； ② 保护错； ③ 非法指令；④ 特权指令错； ⑤ 运行超时；⑥ 等待超时；⑦ 算术运算错

3、外界干预

① 操作员或操作系统干预； ② 父进程请求；

③ 父进程终止。

进程的阻塞与唤醒

- 请求系统服务
- 启动某种操作
- 新数据尚未到达
- 无新工作可做

进程的挂起

1、当出现引起进程挂起的事件时

（1）用户进程请求将自己挂起

（2）父进程请求将自己的某个子进程挂起

系统将利用挂起原语suspend( )将指定进程或处于阻塞状态的进程挂起

2、suspend()原语的执行过程

（1）检查被挂起进程的状态，若处于活动就绪状态，便将其改为静止就绪；对于活动阻塞状态的进程，则将之改为静止阻塞

（2）为了方便用户或父进程考查该进程的运行情况而把该进程的PCB复制到某指定的内存区域

（3）若被挂起的进程正在执行，则转向调度程序重新调度

进程同步相关的概念

进程的同步（直接作用）synchronism

指系统中多个进程中发生的事件存在某种时序关系，需要相互合作，共同完成一项任务

表现：进程运行到某一点时要求另一伙伴进程提供消息

未获得消息之前，该进程处于等待态

获得消息后被唤醒进入就绪态

进程的互斥（间接作用）mutual exclusion

各进程要求共享资源，而有些资源需要互斥使用

各进程间竞争使用这些资源

进程同步相关的概念

临界资源 critical resource

系统中一次只允许一个进程使用的资源（互斥/共享）

例示：

(1)系统中硬件如打印机，进程之间只能互斥访问。

(2)软件：变量、数据、表格、队列等。

并发进程对临界资源的访问必须做限制:

不论是硬件临界资源还是软件临界资源，

多个进程必须**互斥**地对它进行访问

进程同步相关的概念

临界区（critical section）

进程中访问临界资源的代码段

互斥实现：

进入区（entry section）:临界区之前执行代码段

- 对临界资源进行检查，判断是否已被访问
- 临界资源未被访问，进入临界区
- 设置临界区被访问

退出区（exit section） :恢复临界区为未被访问标志

同步机制应遵循的规则

空闲让进

无其他进程处于临界区时，允许一个进程进入临界区

忙则等待

已有进程进入临界区时，其他进程必须等待

有限等待

保证有限时间内进入临界区

让权等待

不能进入临界区的进程应立即释放处理机 Busy waiting

硬件同步机制

（1）关中断

进程进入临界区即屏蔽所有中断

```
执行“关中断”指令
   临界区操作
执行“开中断”指令
```

缺点：

成本高

影响系统效率

不适用于多CPU系统

硬件同步机制

（2）Test-and-Set指令

TS指令可看做一个原语，实现对lock锁的操作

```
boolean TS (boolean *lock)
{
    boolean old;
    old = *lock;
    *lock = TRUE;
}
```

Busy waiting

为每个临界区设置一个lock，初值为FALSE

```
while TS(&lock);

critical section;

lock:=FALSE;
```


硬件同步机制

（3）Swap指令——对换指令

```
void  Swap(int *a, int *b)
{
    int  temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

Busy waiting

为每个临界区设置一个lock，初值为FALSE

```
key = TRUE;
do
{
   Swap(&lock,key);
}while(key);
```

临界区

```
lock:=FALSE;
```

信号量机制（Semaphores）

- 1965年，荷兰学者Dijkstra提出
  - 整型信号量
  - 记录型信号量
  - 信号量集
- 广泛应用于单处理机、多处理机、网络
- 信号量是OS提供的管理公有资源的有效手段
- 信号量代表可用资源实体的数量

信号量机制（Semaphores）

1. 整型信号量

- wait和signal操作描述：

原子?

```
wait (S) {
    while (S≤0) ;
    S=S-1;
}
```

```
signal (S) {
    S=S+1;
}
```

- wait(S)和signal(S)是原子操作，执行时不可中断
- 信号量只能通过原语操作来访问，不被进程调度打断
- 缺点：信号量S≤0时“忙等”，未遵循“让权等待”

信号量机制（Semaphores）

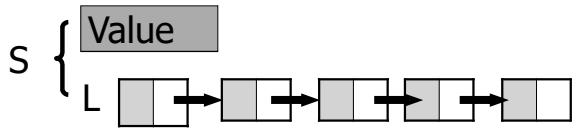
2. 记录型信号量

- “让权等待”
- 无“忙等”
- 资源数目：整型变量value
- 进程链表L：链接访问同一资源的等待进程

信号量机制（Semaphores）

2. 记录型信号量

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



信号量机制（Semaphores）

2. 记录型信号量

- s->value≥0：

  表示系统中可用的资源数量

- S->value<0：

  绝对值表示已阻塞的进程数量

- S->value初值为1时：

  只允许一个进程访问临界资源，是互斥信号量

信号量机制（Semaphores）

3、AND型信号量

将进程在整个运行过程中需要的所有资源，一次性全部地分配给进程，待进程使用完后再一起释放。

- 只要尚有一个资源未能分配给进程，其它所有可能为之分配的资源，也不分配给他。
- 对若干个临界资源的分配，采取原子操作方式：要么全部分配到进程，要么一个也不分配。

- 在wait操作中，增加了一个“AND”条件
  即Swait(Simultaneous wait)

信号量机制（Semaphores）

4、信号量集

- 在记录型信号量机制中，wait(S)和signal(S)操作仅能对信号量施以加1或减1操作，意味着每次只能获得或释放一个单位的临界资源，效率较低
- 在有些情况下，当资源数量低于某下限值时便不予分配。因而，在每次分配之前，都必须测试该资源的数量，看其是否大于等于下限值

信号量机制（Semaphores）

4、信号量集

- Swait(S, d, d)：只有一个信号量S， 允许每次申请d个资源，当资源数少于d时，不分配
- Swait(S, 1, 1)：蜕化为一般的记录型信号量(S＞1时)或互斥信号量(S=1时)
- Swait(S, 1, 0)：特殊且很有用的信号量操作。当S≥1时，允许多个进程进入某特定区；当S变为0后，将阻止任何进程进入特定区，“可控开关”

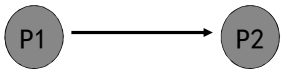
信号量应用

1、进程互斥　　对每一临界资源（区）设一信号量S，初值1
（此时S相当于此临界资源的使用许可证）

```
进程1：                    进程2：
{                          {
    ……                         ……
    P(s)                       P(s)
    临界区;                    临界区;
    V(s)                       V(s)
    ……                         ……
}                          }
```


信号量应用

2、实现前趋关系

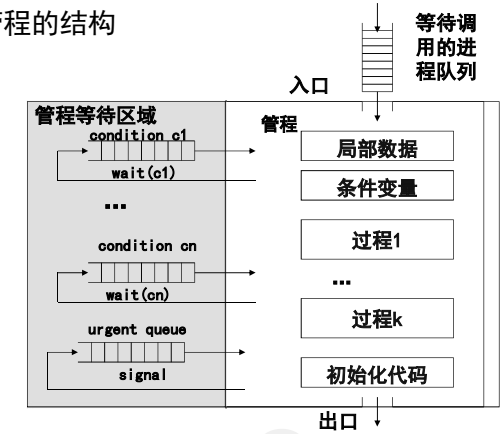


设置一个信号量S，其初值为0,

```
P1;                    P(S);
V(S);                  P2;
```

如此即可实现先执行P1，再执行P2

管程的结构

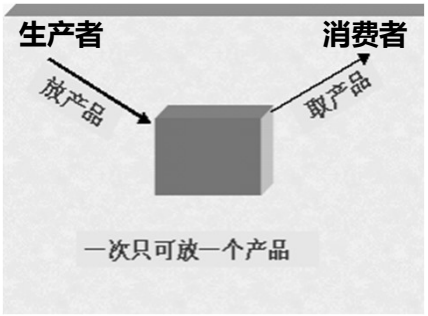


- “生产者—消费者”问题
- “哲学家进餐”问题
- “读者—写者”问题

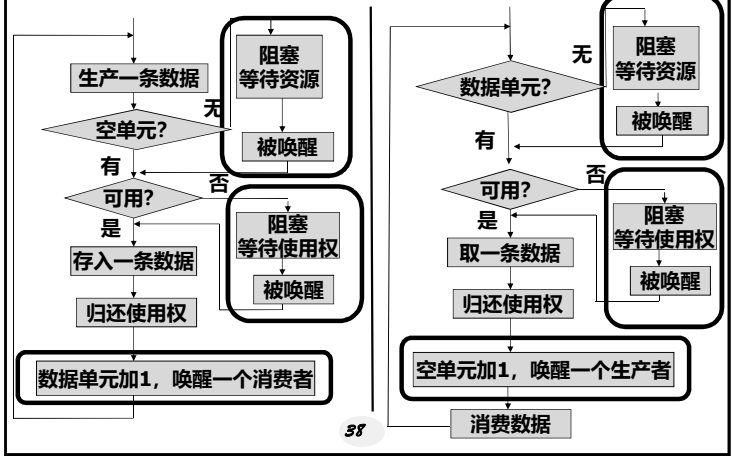
“生产者—消费者”问题——问题描述1

生产者进程生产产品提供给消费者进程消费

- 生产者进程将它所生产的产品放入一个缓冲区中
- 消费者进程可从一个缓冲区中取走产品去消费



“生产者—消费者”问题——求解思想3



“生产者—消费者”问题——记录型信号量1

一个互斥条件：

- 缓冲区一次只能让一个进程访问

  设一互斥信号量**mutex**，初值为1

两个同步条件：

- 缓冲区中至少有一个单元为空时，生产者才送数

  设一信号量**empty**，表示空缓冲区的数量，初值为n

- 缓冲区中至少有一个单元为满时，消费者才取数

  设一信号量**full**，表示满缓冲区的数量，初值为0

“生产者—消费者”问题——记录型信号量2

单缓冲区问题：

empty初值为1，full初值为0

```
P:
while (true) {
    生产一个产品;
    P(empty) ;

    送产品到缓冲区;

    V(full);
};
```

```
C:
while (true) {
    P(full);

    从缓冲区取产品;

    V(empty);
    消费产品;
};
```

“生产者—消费者”问题——记录型信号量4

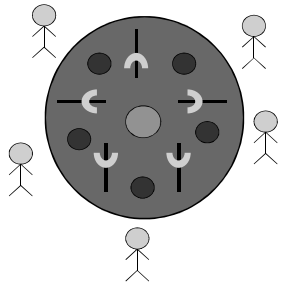
```
int in=0, out=0, counter=0;   item buffer[n];
semaphore empty=n， full=0，mutex=1；

producer(i) {
   var nextp;
   while (TRUE) {
      produce an item in nextp;
      P (empty);
      P (mutex);
      Buffer [in]=nextp;
      in = (in+1) % n;
      V (mutex);
      V (full);
      }
}
```

```
consumer (i) {
   var nextc;
   while (TRUE) {
      P (full);
      P (mutex);
      nextc=buffer [out];
      out= (out+1) % n;
      V (mutex);
      V (empty);
      consume item in nextc;
   }
}
```


“哲学家进餐”问题——问题描述1

五个哲学家围坐圆桌旁
- 桌中央有一盘通心粉
- 每人面前有一只空盘子
- 每两人之间放一只筷子



哲学家的行为
- 思考
- 饥饿
- 吃通心粉

吃通心粉
- 每个哲学家必须拿到两只筷子
- 每个人只能直接从自己的左边或右边去取筷子

“哲学家进餐”问题——问题描述2

```
semaphore chopstick[5]={1,1,1,1,1};

process philosopher ( int i ) {  //i= 0,1,2,3,4
   while(true) {
        think( );
        P(chopstick[i]);
        P(chopstick[(i+1)%5]);
        eat( );
        V(chopstick[i]);
        V(chopstick[(i+1)%5]);
   }
}
```

如果：5人同时拿起左边筷子，再企图拿起右边的筷子时，会如何？

死锁

“哲学家进餐”问题——记录型信号量2

```
semaphore chopstick[5]={1,1,1,1,1};
semaphore room=4;
process philosopher ( int i ) {  //i= 0,1,2,3,4
   while(true) {
        think( );
        P(room);
        P(chopstick[i]);
        P(chopstick[(i+1)%5]);
        eat( );
        V(chopstick[i]);
        V(chopstick[(i+1)%5]);
        V(room);
   }
}
```

“哲学家进餐”问题——AND信号量

```
semaphore chopstick[5]={1,1,1,1,1};

process philosopher ( int i ) {  //i= 0,1,2,3,4
   while(true) {
        think( );
        Swait(chopstick[i],chopstick[(i+1)%5]);
        eat( );
        Ssignal(chopstick[i],chopstick[(i+1)%5]);
   }
}
```


“读者—写者”问题——问题描述1

两组并发进程：读者和写者，共享一个文件F

要求：

- ❖允许多个读者同时执行读操作
- ❖任一写者在完成写操作之前不允许其它读者或写者工作
- ❖写者执行写操作前，应让已有的写者和读者全部退出

“读者—写者”问题——问题描述2

	读 者	写 者
无读者 无写者	√	√
读者在读 无写者等	√	X
读者在读 有写者等	√	X
无读者 有写者写	X	X

“读者—写者”问题——问题描述3

- Writer进程和其它Reader进程互斥
  设互斥信号量Wmutex
- 设置整型变量Readercount表示在读进程数
- Readercount是一个可被多个Reader进程访问的临界资源，为它设置互斥信号量Rmutex

```
semaphore   rmutex=1, wmutex=1;
int         readcount=0;
```

“读者—写者”问题——记录型信号量1

```
    int  readcount=0;
void reader( ) {
  while(TRUE)  {
    P(rmutex);
    if  (readcount==0)
          P(wmutex);
    readcount++;
    V(rmutex);
    perform read operation;
    P(rmutex);
    readcount- -;
    if  (readcount==0)
          V(wmutex);
    V(rmutex);
  }
}
```

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

void writer() {
  while(TRUE) {
    P(wmutex);
    perform write operation;
    V(wmutex);
  }
}

void main()
{
  cobegin
    reader(); writer();
  coend
}
```


“读者—写者”问题——记录型信号量2

```
void reader( ) {
  while(TRUE)  {
    P(S);
    P(rmutex);
    if  (readcount==0)
          P(wmutex);
    readcount++;
    V(rmutex);
    V(S);
    perform read operation;
    P(rmutex);
    readcount- -;
    if  (readcount==0)
          V(wmutex);
    V(rmutex);
  }
}
```

```
int  readcount=0;
semaphore   rmutex=1, wmutex=1;
semaphore   S=1;
//在写者到达后封锁读者

    void writer() {
      while(TRUE) {
        P(S);
        P(wmutex);
        perform write operation;
        V(wmutex);
        V(S);
      }
    }
```


“读者—写者”问题——信号量集1

- 增加“最多只允许RN个读者同时读”的限制
  - 引入了一个信号量L，并赋予其初值为RN
- 执行wait(L，1，1)操作来控制读者的数目
  - 有读者进入，执行wait(L，1，1），L值减1
  - 有RN个读者进入后，L减为0
  - 第RN + 1个读者进入时，阻塞

“读者—写者”问题——信号量集2

```
semaphore L=RN, mx=1;

void reader( ) {
  while(TRUE)  {
    Swait(L,1,1);

    Swait(mx,1,0);

    perform read operation;

    Ssignal(L,1);
  }
}
```

```
void writer() {
  while(TRUE) {
    Swait(mx,1,1，L,RN,0);
    perform write operation;
    Ssignal(mx,1);
  }
}

        void main()
        {
          cobegin
            reader(); writer();
          coend
        }
```

概念(1)

- 并发进程交互的基本要求：同步和通信
- 进程竞争资源时要实施互斥
- 互斥是一种特殊的同步
  - 实质上需要解决好进程同步问题
- 进程同步是一种进程通信
  - 通过修改信号量，进程之间建立起联系，相互协调运行和协同工作

分类(2)

1. 共享存储器系统(Shared-Memory System)
2. 管道(Pipe)通信：又名共享文件通信
3. 消息传递系统(Message Passing System)

（1）直接通信方式：消息缓冲通信

（2）间接通信方式：又称为信箱通信方式

4. 客户机-服务器系统(Client-Serve System)

线程引入(1)

进程模型是基于下面两个独立的概念：

- **资源分配的单位**

操作系统实施保护功能，以防止进程之间发生可能的冲突

- **调度的单位**

一个进程可能通过一个或多个程序（段）的执行轨迹执行，形成一条**进程内执行流，或控制流**。

线程引入(4)

- 进程这两个特点是可以相互独立的
- **操作系统将这两个属性分别赋予了两个不同实体**
- **拥有资源所有权的仍称为进程**
- **调度的单位称为线程**，或轻量级进程。

线程状态

- 执行状态

表示线程正获得处理机而运行

- 就绪状态

指线程已具备了各种执行条件，一旦获得CPU便可执行

- 阻塞状态

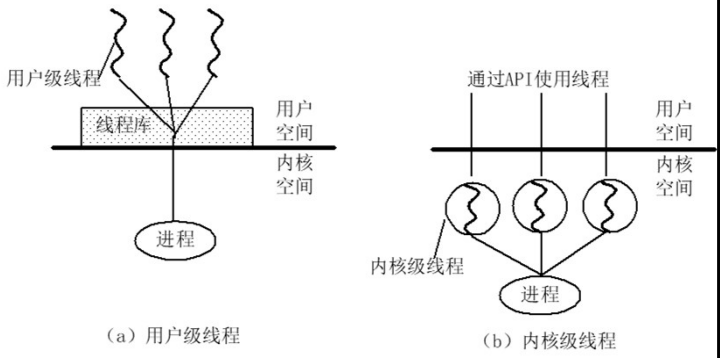
指线程在执行中因某事件而受阻，处于暂停执行状态

线程控制块TCB

- 线程标识符
- 状态参数通常有这样几项：

① 寄存器组：PC、PSD和通用寄存器
② 堆栈：保存有局部变量和返回地址
③ 线程运行状态：描述线程运行状态
④ 优先级：描述线程执行的优先程度
⑤ 线程专有存储器：保存线程局部变量拷贝
⑥ 信号屏蔽：对某些信号加以屏蔽。

线程实现方式



（a）用户级线程　　（b）内核级线程