进程同步与通信

主讲教师：赵霞

提问

- 程序和进程的区别是什么？
- 如何理解进程的并发执行？
- 在进程的并发执行过程中，会发生什么问题呢?

进程同步与通信

- 目的与要求
  - 理解互斥问题的硬件实现方法
  - 掌握信号量机制
  - 使用信号量解决进程同步互斥问题的方法
- 重点与难点
  - PV原语的实现及使用
- 作业
  - 4.13
  - 制作本章思维导图

阅读与思考

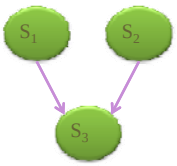
- 教材
  - 第4章
- Operating System Concepts (6th edition)
  - Chapter 7 Process Synchronization
- Modern Operating System (2nd edition)
  - Section 2.3

进程同步与通信

- 并发执行实现
- 进程的同步与互斥
- 消息传递原理

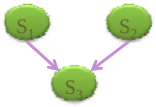
并发执行实现

- 程序：
  - 指令或语句序列，体现了某种算法
  - 所有程序是顺序的
- 程序的执行有两种方式
  - 顺序执行
    - S_1和S_3
    - S_2和S_3
  - 并发执行
    - S_1和S_2



并行编程方法

- 在顺序程序中加入并行语句——Dijkstra
  - Parbegin; $S_1, S_2, \cdots; S_n$; Parend;
- 对这三个任务的描述
  - Parbegin
  - S_1;
  - S_2;
  - Parend
  - S_n;



- 并发语句不能描述某些并发优先关系

进程同步与通信

- 并发执行实现
- 进程的同步与互斥
- 消息传递原理

提问

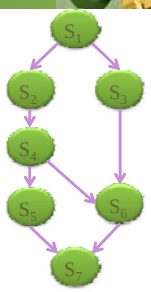
- 如果两个进程需要合作，如何控制他们的步调呢？
  - 同步和互斥的概念？
- 如果两个进程共享读写同一块内存区（同一个变量），如何保证他俩的写操作不冲突呢？

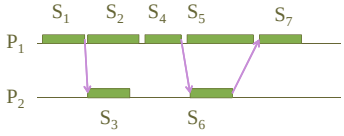
进程的同步与互斥

- 进程间的制约关系
  - 同步关系
    - 直接制约关系
    - 为完成用户任务的伙伴进程间
    - 需要在某些位置上协调其工作次序
    - 等待、传递信息所产生的制约关系
  - 互斥关系
    - 间接制约关系
    - 进程间因相互竞争使用独占型资源

同步问题

- 让两个进程实现如图所示的任务
- 其中P1依次运行S1，S2，S4，S5，S7子任务，进程2依次运行S3, S6子任务
- 则P1和P2之间存在如下同步关系
  - P2 在S3之前等待P1完成S1
  - P2在S6之前等待P1完成S4
  - P1在S7之前等待P2完成S6





互斥问题：银行存取款账户操作

- 进程P1:存款进程
- 进程P2:取款进程
- 两个进程独立、并发执行

```
Parbegin
  P1(amount){
        balance=balance+amount;
  };
  P2(amount) {
        balance=balance-amount;
  };
Parend;
```

```
Parbegin
  P1(amount){
     R1=balance;
     R2=amount;
    R1=R1+R2;
   balance=R1;
  };
```

```
  P2(amount) {
     R1=balance;
     R2=amount;
     R1=R1-R2;
     balance=R1;
  };
Parend;
```

临界资源和临界段问题

- 临界资源（Critical Resource）
  - 一次仅允许一个进程使用（必须互斥使用）的资源
- 临界段（Critical Section）
  - 进程必须互斥执行的程序段，该程序段实施对临界资源的操作
- 临界段问题
  - 若n个进程共享同一临界资源，则每个进程(P1，P2，…，Pn）所执行的程序中均存在关于该临界资源的临界段{CS1，CS2，…，CSn}，这些临界段必须互斥执行
  - 此时，称 这组进程间存在着临界段问题
- 解决问题的关键
  - 进程在执行临界段程序期间，不被其他进程打断

解决临界段问题的硬件方法

- 利用处理机提供的特殊指令实现临界段加锁
- 方法1：关闭中断（单处理机，禁止进程调度）

```
Parbegin
    P1(amount){
        disableInterrupt();
        balance=balance+amount;
        enableInterrupt();
      };
     P2 (amount) {
        disableInterrupt();
        balance=balance-amount;
       enableInterrupt();
      };
Parend;
```

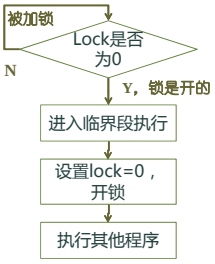
实现临界段问题的硬件方法

- 方法2：“Test_and_Set”指令（多处理机）
  - 该指令功能描述为：
  - Boolean Test_and_Set (boolean &target) {
  - Boolean rv = target;
  - target = true;
  - return rv;
  - }
  - 返回target变量里原来的值，并且把该变量置1

用Test&Set实现加锁

- 设Lock为全局布尔变量，初值为false（0），表示资源可用
- 利用Test&Set指令，实现对临界区的加锁

```
Do{
   while Test_and_Set (Lock) ;
   critical section
   Lock = false;
    non-critical section
} while (1);
```



解决临界段问题的硬件方法

- 方法3：“Swap”指令
  - 该指令功能描述为：
  - Void Swap (boolean &a, boolean &b) {
  - boolean temp = a ;
  - a = b;
  - b = temp;
  - }
  - 交换两个变量的值

用Swap实现加锁

- 设Lock为全局布尔变量，初值为false（0），表示资源可用
- 每个进程设一个局部布尔变量Key
- 利用

```
Do {
   key = true;
   while (key == ture) Swap (Lock, key);
                  //资源不可用，则循环
   critical section
   Lock = false;  //已用完，设置资源可用
   non-critical section
}while (1);
```

用信号量解决临界段问题

- 1965年，荷兰学者Dijkstra
- 信号量
  - s是一整数，初值代表可用资源数，应大于0
  - s>=0时，代表可供并发进程使用的资源实体数；
  - s<0时，表示正在等待使用临界段的进程数；
- PV操作
  - 信号量s 的值只能由P、V操作改变
  - P操作使s减1，V操作使s加1；

PV操作与P V原语

```
P(s):{
    While (s≤0);
    s=s-1;
}
```

```
V(s):{
    s=s+1;
}
```

怎么保证P V操作对s的访问互斥呢？

- 原语（原子操作）
  - 完成某种功能且不被中断执行的操作序列
- P、V原语
  - 互斥操作信号量
  - 通过关闭中断或为信号量加硬锁实现

用关闭中断方法实现P和V

```
P (s) {
    DisableInterrupt();
    while (s<0)
        EnableInterrupt();
    DisableInterrupt();

    s = s - 1;
    EnableInterrupt();
}
V (s) {
    DisableInterrupt();
    s = s +1;
    EnableInterrupt();
}
```

开中断的目的是让CPU在循环等待过程中能够响应中断，进行调度，让其他进程执行释放临界资源

互斥问题解决了，但、、、

```
P (s) {
    DisableInterrupt();
    while (s<0)
        EnableInterrupt();
    DisableInterrupt();

    s = s - 1;
    EnableInterrupt();
}
V (s) {
    DisableInterrupt();
    s = s +1;
    EnableInterrupt();
}
```

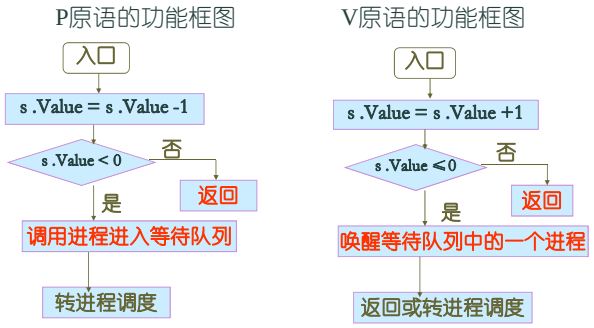
占用CPU，降低系统执行效率

不公平现象：一个进程反复使用临界区时，会造成饥饿

P V原语的实现

- 操作系统实现P V原语时与进程调度相结合，消除忙等待和饥饿现象
- 原则
  - 在P操作循环等待的地方加入放弃处理机/挂入等待队列动作
  - 在V操作时，从等待队列中摘取进程变为就绪态

PV原语的流程



信号量及P V原语的具体实现

❖信号量定义

```
typedef struct{
    int value;
    struct  process *L;一个PCB队列
}semphore;
```

❖P操作

```
void P(semaphore s){
   s.value = s.value  -1;
   If  (s.value<0)  {
        add this process to s.L; 将本进程挂入S.L队列
        block();   重新调度
     }
}
```

信号量及P V原语的具体实现

❖V操作

```
void V(semaphore s){
    s.value = s.value + 1;
    If (s.value <0) {
       remove a process P from s.L; 从S.L队列取一进程
       wakeup(); 唤醒，挂入就绪队列
    }
}
```

用PV原语实现进程互斥

❖假设进程A,B竞争进入临界段

❖信号量S的初值为1

进程A	进程B
P(S); 临界段操作 V(S);	P(S); 临界段操作 V(S);

用P V原语实现互斥

❖用于n个进程的临界段互斥，n进程共享一个信号量mutex，初值为1，任一进程P_i的结构为：

```
Do{
   P(mutex) ;
        critical section
   V(mutex);
    non-critical section
} while (1);
```

用P V原语实现同步

❖有P1、P2 两进程，必须在P1执行完S1语句后，P2才能执行S2。需同步的两进程共享信号量synch，初值为0 ：

```
Semaphore synch;
Synch = 0;
```

P1的程序框架：

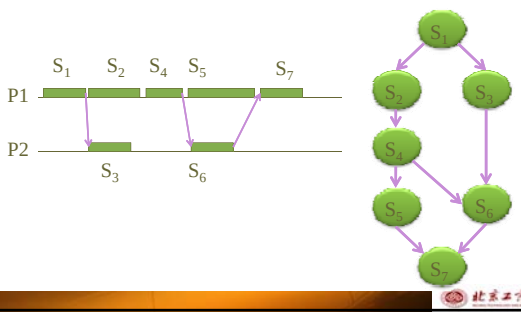
```
....
S1;
V(synch) ;
…
```

P2的程序框架：

```
....
P(synch) ;
S2;
…
```

用P V原语实现进程同步

❖P1依次运行S_1，S_2，S_4，S_5，S_7子任务，进程2依次运行S_3，S_6子任务



用PV原语实现进程同步

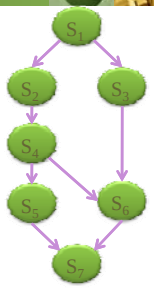
Semaphore s_{13},s_{46},s_{67};
s_{13}= 0; s_{46}= 0; s_{67}= 0;

P1的程序框架:

```
....
s1;
V(s13);
s2;
s4;
V(s46);
s5;
P(s67);
S7;
...
```

P2的程序框架:

```
....
P(s13);
s3;
P(s46);
s6;
V(s67);
...
```



PV原语的优缺点

- 优点:
  - 思想简洁，表达能力强
  - 用P. V操作可解决任何同步互斥问题
- 缺点:
  - 不够安全；P. V操作使用不当会出现死锁；
  - 遇到复杂同步互斥问题时实现复杂

进程同步与互斥举例

- 有限缓冲区问题
- 读写者问题（Readers/Writers问题）
- 哲学家就餐问题

有限缓冲区问题

- 问题描述
  - 设有n个缓冲区，一组生产者进程往缓冲区写数据，一组消费者进程从缓冲区取数据，写取以一个缓冲区为单位
- 说明
  - 将缓冲池看作是共享数据，对缓冲区的操作必须是互斥操作
  - 如果n个缓冲区全满，生产者进程必须等待
  - 如果缓冲区全空，消费者进程必须等待

有限缓冲区问题

- 解：设置以下信号量
  - mutex，初值为1，控制互斥访问缓冲池
  - full，初值为0，表示当前缓冲池中满缓冲区数
  - empty，初值为n，表示当前缓冲池中空缓冲区数
  - 有限缓冲区生产者/消费者进程描述如下：

```
typedef struct{
   ...
 }item; //消息类型
typedef struct{
    struct item inst;
    struct buffer *next;
 }buffer; //缓冲类型
```

```
semaphore full, empty, mutex; //信号量
struct item nextp, nextc; //消息变量
full = 0;
Empty = n;
mutex = 1;
```

有限缓冲区问题

- 生产者进程代码框架

```
do{
   ......
   produce an item in nextp
   ......
   P(empty);
   P(mutex);
   get an empty item;
   copy nextp to an empty item
   add item to full buffer list
   V(full);
   V(mutex);
}while (1);
```

- 消费者进程代码框架

```
do{
   P(full);
   P(mutex);
   get an full item;
   copy data to nextc
   add item to empty buffer list
   V(empty);
   V(mutex);
   ......
   consume the item in nextc
}while (1);
```

两个P操作的顺序不能颠倒，为什么？

读写者问题

- 若存在一共享数据A，那些对它进行读访问者叫读者(Reader)，对它进行写访问者叫做写者(Writer)
- 第一类读写者问题（读者优先）
  - Reader和Writer争夺访问共享数据A时，Reader有较高优先权
  - 如果已存在一个Reader正在访问数据，其它Reader可马上访问
  - 而Writer则需要等待，直到所有Reader全部结束

读写者问题的分析

- 前提
  - 多个进程可同时读数据
  - 任一个进程写数据时，不允许其他进程读或写
  - 当有进程读数据时，不允许任何进程写
- 如果读者来
  - 无读者、写者，新读者可以读
  - 有写者等，但有其它读者正在读，则新读者也可以读
  - 有写者写，新读者等
- 如果写者来
  - 无读者，新写者可以写
  - 有读者，新写者等待
  - 有其它写者，新写者等待

解答

- 定义信号量
  - 读、写者的互斥信号量wrt，初值为1
  - 读者不互斥，但需定义变量readcount，记录读者的个数，以实现与写者的互斥，初值为0
  - 读者对readcount的访问要互斥，定义互斥信号量mutex，初值为1

程序代码

- Reader的一般结构为：
  - P(mutex)；
  - readcount = readcount+1;
  - If (readcount ==1) P(wrt);
  - V(mutex)；
  - reading data
  - P(mutex)；
  - readcount = readcount-1;
  - If (readcount==0) V(wrt);
  - V(mutex);

- Writer的一般结构为：
  - P（wrt）；
  - Writing data
  - V（wrt）；

第二类Reader/Writer问题

- 写者优先
  - Reader和Writer争夺访问共享数据A时，Writer有较高优先权
  - 多个读者可以同时进行读
  - 只允许一个写者写，不允许其他进程读或者写
  - 一旦有写者，则后续读者 必须等待，唤醒时优先考虑写者

第二类读写者问题：解答

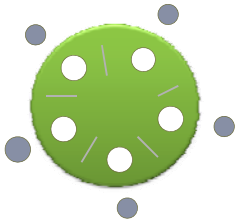
- 增加定义读写者互斥信号量w2，用于实现写者优先，即保证当有写者等待时，读者不能读。初值为1；

```
Reader:
do{
 P(w2);
     P(mutex);
          readcount = readcount+1;
          if (readcount==1) P (w);
     V(mutex);
 v(w2);
 reading  data
 P(mutex);
      readcount = readcount-1;
      if (readcount==0)  V(w);
 V(mutex);
}
```

```
Writer:
 do{
  P(w2);
     P(w);
        writing data
     V(w);
  v(w2);
 }
```

哲学家就餐问题

- 问题描述
  - 五个哲学家五只筷子
  - 哲学家循环做着思考和吃饭的动作
  - 吃饭程序是先取左边筷子，再取右边筷子，再吃饭，再放筷子



哲学家就餐问题：解答

- 为每个筷子设一把锁（信号量，初值为1）每个哲学家是一个进程

```
semaphore chopstick[5]; //信号量
chopstick[4] = {1,1,1,1,1};
第i个哲学家进程的程序：
do{
   P(chopstick[i]);
   P(chopstick[(i+1) mod 5]);
      eating;
   V(chopstick[i]);
   V(chopstick[(i+1) mod 5]);
   thinking;
}while (1);
```

这个解法会死锁，为什么？

哲学家就餐问题

- 为防止死锁发生可采取的措施：
  - 最多允许4个哲学家同时坐在桌子周围
  - 仅当一个哲学家左右两边的筷子都可用时，才允许他拿筷子
  - 给所有哲学家编号，奇数号的哲学家必须首先拿左边的筷子，偶数号的哲学家则反之

哲学家就餐问题——解法1

- 把拿两根筷子的过程原子化
- 定义信号量
  - 每一根筷子一个互斥信号量chopstick [i]，初值均为1
  - 每个人拿两根筷子的过程互斥，定义互斥信号量 mutex，初值为1

最坏情况下，只能有一人就餐

```
do{
   P(mutex);
      P(chopstick[i]);
      P(chopstick[(i+1) mod 5]);
   V(mutext);
   eating;
   V(chopstick[i]);
   V(chopstick[(i+1) mod 5]);
   thinking;
}while (1);
```

能够两人同时就餐

```
do{
  thinking ;
  P(mutex);
  if (stat[i]&stat[i+1])
   {
      P ( chopstick[i] ) ;
      stat[i]=0 ;
      P ( chopstick [(i+1) mod 5] ;
      stat[i+1]=0;
  V ( mutex )
  eating ;
```

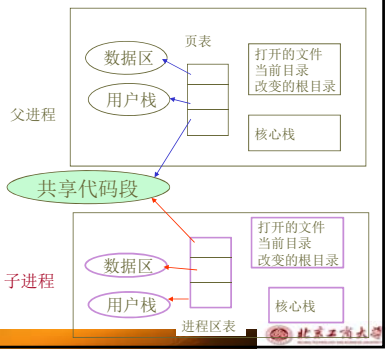
```
    P(mutex);
     V ( chopstick [i] ) ;
     stat[i]=1;
     V ( chopstick [(i+1) mod 5] ) ;
     stat[i+1]=1;
    V(mutex);

  }else   V(mutex);
}
```

进程同步与通信

- 并发执行实现
- 进程的同步与互斥
- 消息传递原理

问题

- 进程有独立的地址空间的含义是什么？
- 如何交换信息？
- 如何交换数据？



消息传递原理

- 进程间的低级通信
  - P.V操作实现 进程间的同步和互斥
- 进程间传递大量信息（高级通信）
  - 共享内存 （shared memory)
    - 相互通讯的进程有共享存储区
    - 进程间可以通过直接读写共享存储区的变量来交互数据
    - 考虑同步与互斥
    - 操作系统提供共享存储区及某些同步互斥工具
  - 消息传递 （message passing)
    - 若进程间无共享空间，则必须通过消息传递通讯，操作系统提供系统调用
      - Send()
      - Receive()

消息传递通信原理

- 消息传递系统调用语句的一般形式
  - 发送：Send &消息 to 目的地标识
  - 接收：Receive &消息 from 源地址标识
- 消息传递方法
  - 直接通讯法
  - 间接通讯法（信箱命名法）

消息传递通信原理

- 直接通讯法
  - 基本思想：进程在发送和接收消息时直接指明接收者或发送者进程ID
  - 缺点：必须指定接收进程ID
  - 举例：UNIX中两进程利用信号通讯

```
Process A
    kill(1040, SIGUSR1);      向1040号进程发送一个SIGUSR1信号

Process B
    Signal(SIGUSR1,func);     //指定收到SIGUSR1信号时，执行func()
```

消息传递通信原理

- 间接通讯法（信箱命名法）
  - 基本思想
    - 系统为每个信箱设一个消息队列
    - 消息发送和接收都指向该消息队列
    - 每个进程可以对消息队列发送并接收/只发送/只接收
  - 特点
    - 必须有一个通讯双方共享的一个逻辑消息队列
    - UNIX的PIPE，FIFO及IPC消息传递机制都属于这种形式
    - 消息发送者以写方式打开信箱
    - 消息接受者以读（或读写）方式打开信箱
  - 优点
    - 很容易建立双向通讯链
    - 只要对信箱说明为读写打开

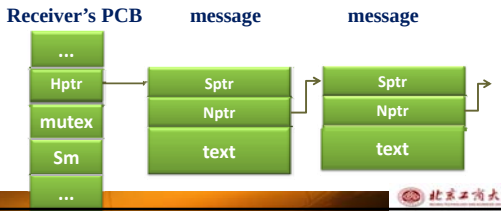
进程通讯示例

- 消息系统的两个基本操作
  - send(A)
    - A指向消息缓冲区
  - Receive(A)
    - A 指向用于接收消息的缓冲区
    - 返回值是消息发送者 pid

- 实现
  - 系统有一空闲缓冲池，每个进程有一个消息缓冲队列
  - 缓冲区用于存放消息、消息发送者pid、和链指针
  - 用pid定位进程PCB表

消息系统

- 实现
  - 每个进程的消息队列存放发送给该进程的消息
  - PCB中设
    - 队列头指针Hptr
    - 互斥信号量mutex（ 初值为1 ）
    - 信号量Sm（初值为0），记录消息队列中的消息数



发送和接收函数

```
Send(&A){
  ……
  new(&p);//从系统缓冲区获得一个p
  p.Sptr = sender'PCB address
  copy the message to p;
  find the receiver' s PCB;
  P(mutex);
  add p to receiver_message_queue;
  V(Sm);
  V(mutex);
  ……
};
```

```
Receive(&A){
  ……
  P(Sm);
  P(mutex);
  move  out a buffer f;
  V(mutex);
  move message from f to receiver;
  dispose(&f); //释放f回系统缓冲区
  ……
};
```

小结

- 进程同步和互斥
  - 信号量
  - P V原语
- 如何用 P V原语实现同步和互斥
  - 理解并发进程之间的逻辑关系
  - 同步：先V后P
  - 互斥：先P后V
- 作业
  - 4.13
  - 制作本章思维导图