同步与互斥

基本概念

- 1.临界资源
  - 两种资源共享方式
    - 互斥共享方式 —— 系统中的某些资源，虽然可以提供给多个进程使用，但【一个时间段内只允许一个进程访问该资源】
    - 同时共享方式 —— 系统中的某些资源，允许一个时间段内由多个进程"同时"对它们进行访问（宏观）（微观并发）
  - 一个时间段内只允许一个进程使用的资源（摄像头、打印机、缓冲区）
  - **对临界资源的访问，必须互斥地进行**
    - 如何实现临界区互斥
      - 1软件实现方法
      - 2硬件实现方法
      - 3采用信号量机制
    - 如果不使用任何机制实现临界区互斥访问→打印机同时打印两个进程的消息→出错
  - **进程在操作系统【内核】程序临界区中【不能】进行进程调度与切换** —— 普通临界区可以
  - 临界资源的访问过程

```
do {
    entry section;        //进入区
    critical section;     //临界区
    exit section;         //退出区
    remainder section;    //剩余区
} while(true)
```

    - 负责检查是否可进入临界区，若可进入，则应设置正在访问临界资源的标志（可理解为"上锁"），以阻止其他进程同时进入临界区
    - 访问临界资源的那段代码
    - 负责解除正在访问临界资源的标志（可理解为"解锁"）
    - 做其他处理
    - **临界区** —— 是进程中访问临界资源的代码段（临界段）
    - 进入区和退出区 —— 是负责实现互斥的代码段
    - 课后题常出现
- 2.同步
  - **亦称直接制约关系** —— 是指为完成某种任务而建立的两个或多个进程，这些进程因为需要在某些位置上协调它们的工作次序而等待、传递信息所产生的制约关系
  - 什么是进程同步
    - **进程具有异步性** —— 异步性是指各并发执行的进程以各自独立的、不可预知的速度向前推进
    - 如何解决这种异步问题（满足次序上的需求），就是进程同步所讨论的内容
    - 操作系统要提供【进程同步机制】来解决异步问题
- 3.互斥
  - **亦称间接制约关系** —— 进程互斥指当一个进程访问某临界资源时，另一个想要访问该临界资源的进程必须等待。当前访问临界资源的进程访问结束，释放该资源之后，另一个进程才能去访问临界资源。
  - 为了实现对临界资源的互斥访问要遵循以下原则：
    - 1. 空闲让进 —— 临界区空闲时，可以允许一个请求进入临界区的进程立即进入临界区
    - 2. 忙则等待 —— 当已有进程进入临界区时，其他试图进入临界区的进程必须等待
    - 3. 有限等待 —— 对请求访问的进程，应保证能在有限时间内进入临界区（保证不会饥饿）
    - 4. 让权等待 —— 当进程不能进入临界区时，应立即释放处理机，防止进程忙等待
    - 2020
      - 问必须遵守的是 —— 没有标准答案
      - 让权等待优先级最低 —— **好多算法都没有实现让权等待**

互斥锁

- 概念
  - **用于进程互斥的访问临界区**
  - 解决临界区最简单的工具
  - 函数 acquire（）获得锁，而函数 release（）释放锁
- 一个进程在进入临界区时应获得锁;在退出临界区时释放锁。
- 每个互斥锁有一个布尔变量available,表示锁是否可用 —— 进程共享变量
- acquire()或release()的执行必须是【原子操作】，因此互斥锁通常采用硬件机制来实现
- 自旋锁 —— 忙等 不满足让权等待
  - 需要连续循环忙等的互斥锁，都可称为自旋锁
  - TSL指令、swap指令、单标志法
- 特性：
  - **· 缺点：需忙等，进程时间片用完才下处理机，违反【让权等待】**
  - · 优点：等待期间不用切换进程上下文，多处理器系统中，若上锁的时间短，则等待代价很低
  - **· 常用于多处理器系统** —— 一个核忙等，其他核照常工作，并快速释放临界区
  - · 不太适用于单处理机系统
    - 忙等的过程中不可能解锁
    - 必须等忙等的A进程时间片用完后→上锁的进程B上处理机解锁后→时间片又轮回A时才能进入临界区
    - 意思是，在一个时间片内只允许一个进程运行

```
acquire()
   while(!available)
      ;                       //忙等待
   available = false;         //获得锁
}
release(){
   available = true;          //释放锁
}
```

信号量

- 复习
  - 进程互斥（互斥访问临界区）
    - 四种软件实现方式
    - 三种硬件实现方式
    - 都无法实现让权等待
  - 信号量机制可以解决上述方式中的缺点 —— 它不仅可以实现进程互斥，还能实现进程同步
    - 记录型信号量 —— 引入阻塞机制，满足让权等待
    - 整型信号量不满足让权等待
- 概念
  - **信号量可用来解决互斥与同步问题，它【只能】被两个标准的原语wait(S)和signal(S)访问，也可记为"P操作"和"V操作"。**（PV操作是原语）
    - 原语通常可由硬件来实现，原语功能的不被中断执行特性在单处理机上可由软件通过屏蔽中断方法实现
    - **原语是一种特殊的程序段**
      - 其执行只能一气呵成，不可被中断
      - **原语是由关中断/开中断指令实现的（硬件）** —— 用开/关中断指令包住的程序段
    - 因此可用原语来解决软件互斥方案中有些操作无法一气呵成而导致的问题
  - 信号量其实就是一个变量，可以用一个信号量来表示系统中某种资源的数量
    - 整型信号量
    - 记录型信号量
  - **一对原语：** —— wait(S) 原语和 signal(S) 原语，可以把原语理解为我们自己写的函数，函数名分别为 wait和signal，括号里的【信号量 S】其实就是函数调用时传入的一个参数。
- **1.整型信号量**
  - 用一个整数型的变量作为信号量，用来表示系统中某种资源的数量
  - 与普通整数变量的区别： —— 对信号量的操作只有三种
    - 初始化
    - P操作
    - V操作
  - P=wait|V=signal
  - **wait操作** —— 相当于检查和上锁一气呵成 —— 联系软件实现方式的双标志先检查法
  - 问题 —— **不满足"让权等待"原则，会发生"忙等"**

```
int S = 1;  //初始化整型信号量s，表示当前系统中可用的打印机资源数

void wait (int S) {   //wait 原语，相当于"进入区"
  while (S <= 0);     //如果资源数不够，就一直循环等待
  S=S-1;              //如果资源数够，则占用一个资源
}

void signal (int S) { //signal 原语，相当于"退出区"
  S=S+1;         //使用完资源后，在退出区释放资源
}
```

  - "检查"和"上锁"一气呵成，避免了并发、异步导致的问题
  - 存在的问题：不满足"让权等待"原则，会发生"忙等"
  - 要会默写
- **2.记录型信号量（默认）**

```
/*记录型信号量的定义*/
typedef struct {
  int value;            //剩余资源数
  struct process *L;    //等待队列
} semaphore;
```

  - 进程链表L,用于链接所有等待该资源的进程
  - 即用记录型数据结构表示的信号量 —— 不"忙等"
  - 对信号量 S 的一次 P 操作
    - **请求（申请）一个单位的该类资源** —— 运行→阻塞态

```
/*某进程需要使用资源时，通过 wait 原语申请*/
void wait (semaphore S) {
  S.value--;
  if (S.value < 0 ) {
    block (S.L);
  }
}
```

      - 如果剩余资源数不够，使用block原语使进程从运行态进入阻塞态，并把挂到信号量 S 的等待队列（即阻塞队列）中
    - 因此需要执行 S.value--，表示资源数减1，减1后当S.value < 0 时表示该类资源已分配完毕，因此进程应调用
    - **【 block 原语进行自我阻塞】（当前运行的进程从运行态——>阻塞态），主动放弃处理机，并插入该类资源的等待队列 S.L 中** —— 遵循了让权等待
  - 对信号量 S 的一次 V 操作
    - **释放一个单位的该类资源** —— 阻塞→就绪态 —— 有细节

```
/*进程使用完资源后，通过 signal 原语释放*/
void signal (semaphore S) {
  s.value++;
  if (S.value <= 0) {
    wakeup(S.L);
  }
}
```

      - 释放资源后，若还有别的进程在等待这种资源，则使用 wakeup 原语唤醒等待队列中的一个进程，该进程从阻塞态变为就绪态
    - 因此需要执行 S.value++，表示资源数加1，若加1后仍是 S.value <= 0，表示依然有进程在等待该类资源，因此应调用
    - 【 wakeup 原语】唤醒等待队列中的第一个进程（被唤醒进程从阻塞态——>就绪态）
  - **总的来说，调用PV操作前，若s.value≥0则阻塞队列无等待该资源的进程** —— 执行V操作前若s.value=-1（表示有一个进程在等待该资源），执行V后s.value=0（此进程被唤醒） —— 注
  - 考试中出现 P(S)、V(S) 的操作，除非特别说明，否则默认 S 为记录型信号量
- **3.利用信号量实现同步**
  - **进程同步**
    - 要让各并发进程按要求有序地（我们想要的顺序）推进
    - 让本来异步并发的进程相互配合，有序推进
  - 步骤
    - 1. 分析什么地方需要实现"同步关系"，即必须保证"一前一后"执行的两个操作（或两句代码）
    - **2. 设置同步信号量 S，初始为 0**
    - **3. 在"前操作"之后执行 V(S)**
    - **4. 在"后操作"之前执行 P(S)**
  - 前V后P —— 信号量初值为0

```
semaphore S=0;  //初始化
P1(){              P2(){
  代码1;             P(S);
  代码2;             代码4;
  V(S);   释放资源    代码5;
  代码3;             代码6;
}                  }
```

- **5.利用信号量实现前驱关系**（多个进程之间的同步）
  - 每一对前驱关系都是一个进程同步问题
  - 1. 要为每一对前驱关系各设置一个同步信号量
  - 2. 在"前操作"之后对相应的同步信号量执行 V操作
  - 3. 在"后操作"之前对相应的同步信号量执行 P 操作
  - 相反
  - 前V后P

- **4.利用信号量实现进程互斥**

```
/*信号量机制实现互斥*/
semaphore mutex=1;  //初始化信号量

P1(){
  ...
  P(mutex);       //使用临界资源前需要加锁
  临界区代码段...
  V(mutex);       //使用临界资源后需要解锁
  ...
}
```

  - 信号量mutex初值为1
  - 在进入区P(mutex)申请资源
  - 在退出区V(mutex)释放资源

```
/*记录型信号量的定义*/
typedef struct {
  int value;           //剩余资源数
  struct process *L;   //等待队列
} semaphore;
```

  - semaphore
  - 注意
    - 对不同的临界资源需要设置不同的互斥信号量
    - P、V操作必须成对出现
- 6.分析进程同步和互斥问题的方法步骤
  - (1)关系分析 —— 找出问题中的进程数，并分析它们之间的同步和互斥关系
  - (2)整理思路 —— 确定P操作、V操作的大致顺序
  - (3)设置信号量 —— 设置需要的信号量，确定初值

管程

- 1.管程的定义
  - 概念
    - 信号量机制存在的问题： —— 编写程序困难、易出错。需要再关注复杂的PV操作
    - 进程同步工具——管程
      - 也是用来实现进程的互斥和同步的
      - **管程是由编程语言支持的进程同步机制** —— 2016
    - **管程的特性保证了进程互斥** —— 无须程序员自己实现互斥，从而降低了死锁发生的可能性
    - 提供了条件变量，灵活地实现进程同步
    - 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
  - **组成** —— 图

```
monitor Demo{ //①定义一个名称为"Demo"的管程
  //②定义共享数据结构，对应系统中的某种共享资源
  共享数据结构 S;
  //④对共享数据结构初始化的语句
  init_code(){
     S=5;          //初始资源数等于 5
  }
  //③过程 1：申请一个资源
  take_away(){
     对共享数据结构 x 的一系列处理;
     S--;          //可用资源数-1
     ...
  }
  //③过程 2：归还一个资源
  give_back(){
     对共享数据结构 x 的一系列处理;
     S++;          //可用资源数+1
     ...
  }
}
```

    - 1. 管程有一个名字。
      - **monitor Demo ... end monitor（名称: Demo）**
      - **管程是语法范围，无法创建和撤销**
    - 2. 局部于管程的【共享数据结构】说明 —— 相当于共享数据
    - 3. 对局部于管程的共享数据设置【初始值】的语句
    - 4. 对该数据结构进行操作的一组【过程】
      - "过程"其实就是"函数"
      - 相当于访问共享数据的入口
  - **基本特征**
    - 1. 局部于管程的数据只能被局部于管程的过程所访问
    - **2. 一个进程只有通过调用管程内的过程才能进入管程访问共享数据;**
    - **3. 每次仅允许一个进程在管程内执行某个内部过程** —— **注意：这种互斥特性是由编译器负责实现的**
  - 封装思想
- **2.条件变量**
  - 可在管程中设置条件变量及等待唤醒操作以解决同步问题
  - 将阻塞原因定义为条件变量condition
    - 一个进程被阻塞的原因可以有多个，因此在管程中设置了多个条件变量
    - 每个条件变量保存了一个等待队列，用于记录因该条件变量而阻塞的所有进程
    - condition x;
  - 对条件变量只能进行两种操作，即wait和signal
    - x.wait
      - 当x对应的条件不满足时，正在调用管程的进程调用x.wait将自己插入x条件的等待队列，并释放管程
      - 此时其他进程可以使用该管程
    - x.signal —— x对应的条件发生了变化，则调用x.signal,唤醒一个因x条件而阻塞的进程
  - 解决的问题
    - **同步** —— 可在管程中设置条件变量及等待/唤醒操作以解决同步问题 —— 可以让一个进程或线程在条件变量上等待 可以通过唤醒操作将等待在条件变量上的进程或线程唤醒。
    - **互斥** —— 各进程必须互斥的访问管程的特性是由编译器实现的

```
monitor Demo{
   共享数据结构 S;
   condition x;                //定义一个条件变量 x
   init_code(){ ... }
   take_away(){
      if(S<=0) x.wait();       //资源不够，在条件变量 x 上阻塞等待
      资源足够，分配资源，做一系列相应处理;
   }
   give_back(){
      归还资源，做一系列相应处理;
      if(有进程在等待) x.signal;  //唤醒一个阻塞进程
   }
}
```

- 条件变量和信号量的比较 —— 真
  - 相似点： —— 条件变量的wait/signal操作类似于信号量的P/V操作，可以实现进程的阻塞/唤醒
  - 不同点： —— **条件变量是【没有值】的，仅实现了"排队等待"功能；而信号量是【有值】的，信号量的值反映了剩余资源数，而在管程中，剩余资源数用【共享数据结构】记录** —— int S