进程同步-3

教师：姜博
E-Mail: gongbell@gmail.com

内容提要

- 同步与互斥问题
- 基于忙等待的互斥方法
- 基于信号量的方法
- 基于管程的同步与互斥
- 进程通信的主要方法
- 经典的进程同步与互斥问题

经典进程同步问题

- 生产者－消费者问题(the producer-consumer problem)
- 读者－写者问题(the readers-writers problem)
- 哲学家进餐问题(the dining philosophers problem)

OS和应用开发中，常常遇到类似的问题

生产者消费者问题

- 典型的类似应用场景-事件驱动的程序
- 事件-需要用户响应的事情
  - 用户的按键、磁盘数据返回
  - 网络数据到达、异步的操作完成
- 当事件发生时：
  - 生产者线程创建一个事件对象，放入事件缓冲区
  - 消费者线程（event handlers）从缓冲区取出事件，进行响应处理

生产者消费者问题

生产者	消费者
P(mutex);	P(mutex);
P(empty);	P(full);
one >> buffer	one << buffer
V(full)	V(empty)
V(mutex)	V(mutex)

典型的可能导致死锁的同步错误：
在获取到一个mutex的情况下，等待信号量。

读者－写者问题(the readers-writers problem)

- 问题描述：对共享资源的读写操作，任一时刻“写者”最多只允许一个，而“读者”则允许多个——“读－写”互斥，“写－写”互斥，“读－读”允许
- 实际应用场景，对共享数据结构、数据库、文件的多线程并发访问。

读者-写者问题分析

- 生活中的实例：火车/飞机定票
  - 读者：？
  - 写者：？
- 多个线程/进程共享内存中的对象
  - 有些进程读，有些进程写
  - 同一时刻，只有一个激活的写进程
  - 同一时刻，可以有多个激活的读进程
- 分类互斥问题：
  - 当写线程在临界区中，其他任何线程不能进入
  - 当读线程在临界区中，写线程不能进入，读线程可以

采用信号量机制-信号量定义

读线程在临界区中，写线程不能进入，读线程可以

- 要记录读线程的数目，只有第一个读者要申请锁

int readers = 0 //记录临界区内读者的数目

mutex = Semaphore(1) //保护对readers的访问

roomEmpty = Semaphore(1)

roomEmpty：1 表示临界区没有任何线程（读或写），0表示有线程在临界区

wait：等待条件为真；signal：通知条件为真了。

Writer

```
roomEmpty.wait();
    write
  //critical region
roomEmpty.signal();
```

Reader

保证有写者，读者不能进入
没有写者，开始加锁

```
mutex.wait();
   readers=readers+1;
   if readers == 1 : //第一个读者
      roomEmpty.wait()
mutex.signal();

read //critical region

mutex.wait();
   readers = readers-1;
   if readers == 0:
      roomEmpty.signal();
mutex.signal();
```

Writer

```
roomEmpty.wait();
    write
  //critical region
roomEmpty.signal();
```

Reader

```
mutex.wait();
   readers=readers+1;
   if readers == 1 : //第一个读者
      roomEmpty.wait()
mutex.signal();

read //critical region

mutex.wait();
   readers = readers-1;
   if readers == 0:
      roomEmpty.signal();
mutex.signal();
```

最后一个读者，解锁，让写者能够进入。

采用信号量机制-PV操作的版本

int readers = 0

Semaphore mutex = 1

Semaphore roomEmpty = 1

Writer

```
P(roomEmpty);
    write
//critical region
V(roomEmpty);
```

Reader

```
P(mutex);
   readers=readers+1;
   if readers == 1 : //第一个读者
      P(roomEmpty)
V(mutex);

read //critical region

P(mutex);
   readers = readers-1;
   if readers == 0:
      V(roomEmpty);
V(mutex);
```

读者-写者- 灯开关模式

- 读者（分类互斥）算法的模式：
  - 第一个读线程加锁，最后一个读线程解锁。
- 通常也称为灯开关模式(Lightswitch)
  - 第一个进屋的人开灯（对mutex加锁）
  - 最后一个离开屋的人关灯（对mutex解锁）

读者-写者- 灯开关模式

```
class Lightswitch:
    def __init__(self):
        self.counter = 0
        self.mutex = Semaphore(1)
    def lock(self, semaphore):
        self.mutex.wait()
            self.counter += 1
            if self.counter == 1:
                semaphore.wait()
        self.mutex.signal()
    def unlock(self, semaphore):
        self.mutex.wait()
            self.counter -= 1
            if self.counter == 0:
                semaphore.signal()
        self.mutex.signal()
```

第一个进入者对semaphore加锁

最后一个离开者对semaphore解锁

读者-写者- 使用lightswitch

信号量初始化

```
readLightswitch = Lightswitch()
roomEmpty = Semaphore(1)
```

Reader

```
readLightswitch.lock(roomEmpty)
  read # critical section
readLightswitch.unlock(roomEmpty)
```

Writer 代码不变

```
roomEmpty.wait();
    write  //critical region
roomEmpty.signal();
```

一般“信号量集”机制

- 一般“信号量集”是指同时需要多种资源、每种占用的数目不同、且可分配的资源还存在一个临界值时的信号量处理。
- 一次需要N个某类临界资源时，就要进行N次wait操作——低效且容易死锁
- 基本思想：在AND型信号量集的基础上进行扩充：进程对信号量Si的测试值为ti（用于信号量的判断，即Si >= ti，表示资源数量低于ti时，便不予分配），资源的申请量为di（用于信号量的增减，即Si = Si - di和Si = Si +di)

```
SP(S1, t1, d1, … , Sn, tn, dn)
   if S1>=t1 and … and Sn>=tn then
      for I :=1 to n do
         Si := Si - di;
      endfor
  else
   wait in Si;
  endif
```

```
SV(S1, d1, … ,Sn, dn)
   for I :=1 to n do
         Si := Si + di;
         wake waited process
   endfor
```

- 原语：
- SP(S1, t1, d1; ...; Sn, tn, dn);
- SV(S1, d1; ...; Sn, dn);

特殊情况：

- SP(S, d, d)
  - 表示每次申请**d**个资源，当资源数量少于**d**个时，便不予分配
- SP(S, 1, 1)
  - 表示互斥信号量
- SP(S, 1, 0)
  - 可作为一个可控开关(当**S≥1**时，允许多个进程进入临界区；当**S=0**时禁止任何进程进入临界区)

采用一般“信号量集”机制

- 增加一个限制条件：同时读的“读者”最多RN个
- mx表示“允许写”，初值是1
- L表示“允许读者数目”，初值为RN

Writer

SP(mx, 1, 1; L, RN, 0);
write
SV(mx, 1);

Reader

SP(mx, 1, 0 ; L, 1, 1);
read
SV(L, 1);

L>=RN 允许写
SP(S, 1, 0):可作为一个可控开关：(当S≥1时，允许多个进程进入临界区；当S=0时禁止任何进程进入临界区)

“读者-写者“算法的问题

```
Writer

P(roomEmpty);
   write
//critical region
V(roomEmpty);
```

```
Reader

P(mutex);
   readers=readers+1;
   if readers == 1 : //第一个读者
      P(roomEmpty)
V(mutex);

read //critical region

P(mutex);
   readers = readers-1;
   if readers == 0:
      V(roomEmpty);
V(mutex);
```

该算法是对读者有利，还是对写者有利？

“读者-写者“算法的问题

Reader

```
P(mutex);
   readers=readers+1;
   if readers == 1 : //第一个读者
```

Writer

```
P(
//c
V(
```

写者可能被饿死（Starvation）！
当系统负载很低，可以工作，
当系统负载很高，写者会几乎没机会。

```
P(mutex);
   readers = readers-1;
   if readers == 0:
      V(roomEmpty);
V(mutex);
```

该算法是对读者有利，还是对写者有利？

读者写者算法的特性

- 给定读写序列：r1,w1,w2,r2,r3,w3…
  - 读者优先：r1,r2,r3,w1,w2,w3…
  - 写者优先：r1,w1,w2,w3,r2,r3…
  - 读写公平：r1,w1,w2,r2,r3,w3…
- 如何设计写者优先?
- 如何设计公平读写?

“读者-写者“公平读写算法

信号量初始化

```
readLightswitch = Lightswitch()
roomEmpty = Semaphore(1)
```

Reader

```
readLightswitch.lock(roomEmpty)
  read # critical section
readLightswitch.unlock(roomEmpty)
```

Writer 代码不变

问题：如何扩展当前算法，当一个写者到达，已进入的读者可以结束，但是新的读者无法进入？

“读者-写者“算法的问题

信号量初始化

```
readLightswitch = Lightswitch()
roomEmpty = Semaphore(1)
```

Reader

```
readLightswitch.lock(roomEmpty)
  read # critical section
readLightswitch.unlock(roomEmpty)
```

Writer 代码不变

问题：如何扩展当前算法，当一个写者到达，已进入的读者可以结束，但是新的读者无法进入？

一种低级通信原语：屏障Barriers

- 思考：如何使用PV操作实现Barrier?
  - n = the number of threads
  - count = 0 //到达汇合点的线程数
  - mutex = Semaphore(1) //保护count
  - barrier = Semaphore(0)//线程到达之前都是0或者负值。到达后取正值
  - ...
  - if count == n: barrier.signal() # 唤醒一个线程
  - barrier.wait()
  - barrier.signal() # 一旦线程被唤醒，有责任唤醒下一个线程

Turnstile-闸机

- barrier = Semaphore(0)
- barrier.wait()
- barrier.signal()

- Turnstile：连续两个wait和signal组成
- 它可以关闭以阻止所有线程，也可以让线程轮流通过
  - 初值为0，闸机关闭，任何线程不能进入
  - 当值为1，多个线程可以轮流通过

```
int readers = 0
Semaphore mutex = 1
Semaphore roomEmpty =1
Semaphore turnstile = 1
```

```
Writer

P(turnstile);
    P(roomEmpty);
    write  //critical region
V(turnstile);
V(roomEmpty);
```

```
Reader

P(turnstile)
V(turnstile)
P(mutex);
   readers=readers+1;
   if readers == 1 : //第一个读者
      P(roomEmpty)
V(mutex);

read //critical region

P(mutex);
   readers = readers-1;
   if readers == 0:
      V(roomEmpty);
V(mutex);
```

读者需要在闸机排队

```
int readers = 0
Semaphore mutex = 1
Semaphore roomEmpty =1
Semaphore turnstile = 1
```

```
Writer

P(turnstile);
   P(roomEmpty);
   write  //critical region
V(turnstile);
V(roomEmpty);
```

writer在通过闸机后，必须等待roomEmpty，才会离开闸机，后续reader才能进入

```
Reader

P(turnstile)
V(turnstile)
P(mutex);
   readers=readers+1;
   if readers == 1 : //第一个读者
      P(roomEmpty)
V(mutex);

read //critical region

P(mutex);
   readers = readers-1;
   if readers == 0:
      V(roomEmpty);
V(mutex);
```

读者需要在闸机排队

非饥饿版本的读者写者算法-公平读写

```
readSwitch = Lightswitch()
roomEmpty = Semaphore(1)

turnstile = Semaphore(1)//对写者互斥锁，对读者闸机
Writer
turnstile.wait()
        roomEmpty.wait()
        # critical section for writers
turnstile.signal()
roomEmpty.signal()

Reader
turnstile.wait()
turnstile.signal()
readSwitch.lock(roomEmpty)
        # critical section for readers
readSwitch.unlock(roomEmpty)
```

非饥饿版本的读者写者算法-公平读写

```
readSwitch = Lightswitch()
roomEmpty = Semaphore(1)

turnstile = Semaphore(1)
```

Writer

```
turnstile.wait()
        roomEmpty.wait()
        # critical section for writers
```

调度器会决定闸机外排队的那个进程先被调度

Reader

```
turnstile.wait()
turnstile.signal()
readSwitch.lock(roomEmpty)
        # critical section for readers
readSwitch.unlock(roomEmpty)
```

理发师问题

- Dijkstra首先提出
- 理发店里有一位理发师、一把理发椅和n把供等候理发的顾客坐的椅子；
- 如果没有顾客，理发师便在理发椅上睡觉，当一个顾客到来时，叫醒理发师；
- 如果理发师正在理发时，又有顾客来到，则如果有空椅子可坐，就坐下来等待，否则就离开。

互斥访问资源：排队的顾客数（计数器 waiting）
同步：顾客唤醒理发师、理发师唤醒下一个位等待顾客

理发师问题

```
semaphore customers = 0; //等待理发的顾客
semaphore barbers = 0;  //等待顾客的理发师
int waiting = 0;  //等待的顾客数 (不包含正在理发的顾客)
semaphore mutex = 1;  //互斥访问waiting
```

算法描述

```
信号量：customers=0;barbers=0;mutex=1
整型变量：waiting=0;
假设：CHIRS=10
理发师进程：
begin
   While(true)then
   begin
      P(customers);
      P(mutex);
        waiting=waiting-1;
      V(mutex);
      V(barbers);
      Cut hair();
   end
end
```

```
顾客进程：
begin
   P(mutex);
      If （waiting<CHIRS)
then
         begin
           waiting=waiting+1;
           V(mutex);
           V(customers);
           P(barbers);
             Get_haircut();
         end
    else
         begin
           V(mutex);
         end
end
```

若顾客为0，睡觉

开始排队

等待理发师

准备好剪发

有座位等么？

没座位离开

理发师问题

```
#define CHAIRS 5   //chairs for waiting customers
typedef int semaphore;
semaphore customers = 0; //# of customers waiting service
semaphore barbers = 0;  //# of barbers waiting customers
semaphore mutex = 1;  //for mutual exclusion of waiting
int waiting = 0;  //customer are waiting (not being cut)
```

理发师问题

```
void barber(void) {
    while (TRUE) {
        down(&customers); /* go to sleep if # of customers is 0 */
        down(&mutex); /* acquire access to "waiting' */
        waiting = waiting - 1; /* decrement count of waiting customers */
        up(&mutex); /* release 'waiting' */
        up(&barbers); /* one barber is now ready to cut hair */
        cut_hair(); /* cut hair (outside critical region */
        }
}
```

理发师问题

```
void customer(void) {
      down(&mutex); /* enter critical region */
      if (waiting < CHAIRS) {
      /* if there are no free chairs, leave */
            waiting = waiting + 1; /* increment count of waiting customers */
            up(&mutex); /* release access to 'waiting' */
            up(&customers); /* wake up barber if necessary */
            down(&barbers); /* go to sleep if # of free barbers is 0 */
            get_haircut(); /* be seated and be served */ }
      else {
        up(&mutex); /* shop is full; do not wait */
      }
}
```

“生产者-消费者”扩展问题

- 某银行有n个服务柜台。每个顾客进店后先取一个号，并且等待叫号。当一个柜台人员空闲下来时，就叫下一个号。试设计一个使柜台人员和顾客同步的算法。
- 谁是生产者?
- 谁是消费者?

“生产者-消费者”扩展问题

- 某银行有n个服务柜台。每个顾客进店后先取一个号，并且等待叫号。当一个柜台人员空闲下来时，就叫下一个号。试设计一个使柜台人员和顾客同步的算法。
- 谁是生产者?
- 谁是消费者?

算法描述-基于信号量同步

```
int cstmr_cnt = 0; //下一个要服务的客户
Semaphore s_mutex=1;//服务器进程互斥访问cstmr_cnt
Semaphore next_cstmr = 0; //客户服务器进程同步
```

```
process customer i
Begin

   v(next_cstmr);

  }
end
```

```
process servers i(i=1,...,n)
begin
    while(true){
        p(s_mutex);
        p(next_cstmr);
        cstmr_cnt ++;
        v(s_mutex);
        为持有next_cstmr的客户服务;
    }
end
```

另外一种算法描述-基于忙等同步

```
int cstmr_id = 0;   //当前客户编号
semaphore mutex=1; //对cstmr_id互斥访问
int next_cstmr = 0; //下一个要服务客户编号
semaphore s_mutex=1; //服务器进程互斥访next_cstmr
```

```
process customer i
Begin
  p(mutex);
     cstmr_id ++;
  v(mutex);

  }
end
```

```
process servers i(i=1,...,n)
begin
    while(true){
        p(s_mutex);
        p(mutex);
           if(next_cstmr < cstmr_id)
              next_cstmr ++;
        v(mutex)
        v(s_mutex);
        为next_cstmr号码持有者服务;
    }
end
```

构建水分子(H2O)问题

- Berkeley OS 课程习题.
- Gregory R. Andrews. Concurrent Programming: Principles and Practice. Addison-Wesley, 1991.
- 存在两种线程，一个线程提供氧原子O，一个线程提供氢原子H。为了构建水分子，我们需要使用barrier让线程同步从而构建水分子(H_2O)。
- 当线程通过barrier，需要调用bond（形成化学键），需要保证构建同一个分子的线程调用bond。
  - 当氧原子线程到达barrier，而氢原子线程还没到达，需要等待氢原子。
  - 当1个氢原子到达而没有其他线程到达，需要等待1个氢原子1个氧原子
- 只需要保证成组通过barrier

构建水分子(H2O)问题

- Berkeley OS 课程习题.

如何构建同步原语?

- 存在两种线程，一个线程提供氧原子O，一个线程提供氢原子H。为了构建水分子，我们需要使用barrier让线程同步从而构建水分子(H_2O)。
- 当线程通过barrier，需要调用bond（形成化学键），需要保证构建同一个分子的线程调用bond。
  - 当氧原子线程到达barrier，而氢原子线程还没到达，需要等待氢原子。
  - 当1个氢原子到达而没有其他线程到达，需要等待1个氢原子1个氧原子
- 只需要保证成组通过barrier，不需要

构建水分子(H2O)问题

- 信号量定义

oxygen = 0 //氧原子的计数器
hydrogen = 0 //氢原子的计数器
Semaphore mutex = 1 //保护计数器的mutex
Barrier barrier(3) //3表示需要调用3次wait后barrier才开放
//3个线程调用bond后的同步点，之后允许下一个线程继续
Semaphore oxyQueue = 0 //氧气线程等待的信号量
Semaphore hydroQueue = 0 //氢气线程等待的信号量
//用在信号量上睡眠来模拟队列
P(oxyQueue) 表示加入队列
V(oxyQueue) 表示离开队列

构建水分子(H2O)问题

- 氧气线程

```
P(mutex)
oxygen += 1
if hydrogen >= 2:
    V(hydroQueue)
    V(hydroQueue)
    hydrogen -= 2
    V(oxyQueue)
    oxygen -= 1
else:
    V(mutex)
P(oxyQueue) //入队等待
bond()
barrier.wait()
V(mutex)
```

构建H_2O成功

- 氢气线程

```
P(mutex)
hydrogen += 1
if hydrogen >= 2 and oxygen >= 1:
    V(hydroQueue)
    V(hydroQueue)
    hydrogen -= 2
    V(oxyQueue)
    oxygen -= 1
else:
    V(mutex)
P(hydroQueue) //入队等待
bond()
barrier.wait()
```

构建H_2O成功

不成功释放mutext

同步以生成水分子

构建水分子(H2O)问题

- 氧气线程

```
P(mutex)
oxygen += 1
if hydrogen >= 2:
    V(hydroQueue)
    V(hydroQueue)
    hydrogen -= 2
    V(oxyQueue)
    oxygen -= 1
else:
    V(mutex)
P(oxyQueue)
bond()
barrier.wait()
V(mutex)
```

- 氢气线程

```
P(mutex)
hydrogen += 1
if hydrogen >= 2 and oxygen >= 1:
    V(hydroQueue)
    V(hydroQueue)
    hydrogen -= 2
    V(oxyQueue)
    oxygen -= 1
else:
    V(mutex)
P(hydroQueue)
bond()
barrier.wait()
```

构建H_2O成功，有一个mutex没释放，释放mutex

当三个线程离开barrier时候，最后那个线程拿着mutex还没释放，虽然我们不知道那个线程hold mutex，但是我们一定要释放一次。因为氧气只有一个线程，就放在氧气线程中做了。

```
P(mutex)
oxygen += 1
if hydrogen >= 2:
    V(hydroQueue)
    V(hydroQueue)
    hydrogen -= 2
    V(oxyQueue)
    oxygen -= 1
else:
    V(mutex)
P(oxyQueue)
bond()
barrier.wait()
V(mutex)
```

```
P(mutex)
hydrogen += 1
if hydrogen >= 2 and oxygen >= 1:
    V(hydroQueue)
    V(hydroQueue)
    hydrogen -= 2
    V(oxyQueue)
    oxygen -= 1
else:
    V(mutex)
P(hydroQueue)
bond()
barrier.wait()
```

北京航空航天大学　　计算机学院