上海交通大学
SHANGHAI JIAO TONG UNIVERSITY

SJTU

L5-2. 进程同步实例

宋卓然

上海交通大学计算机系

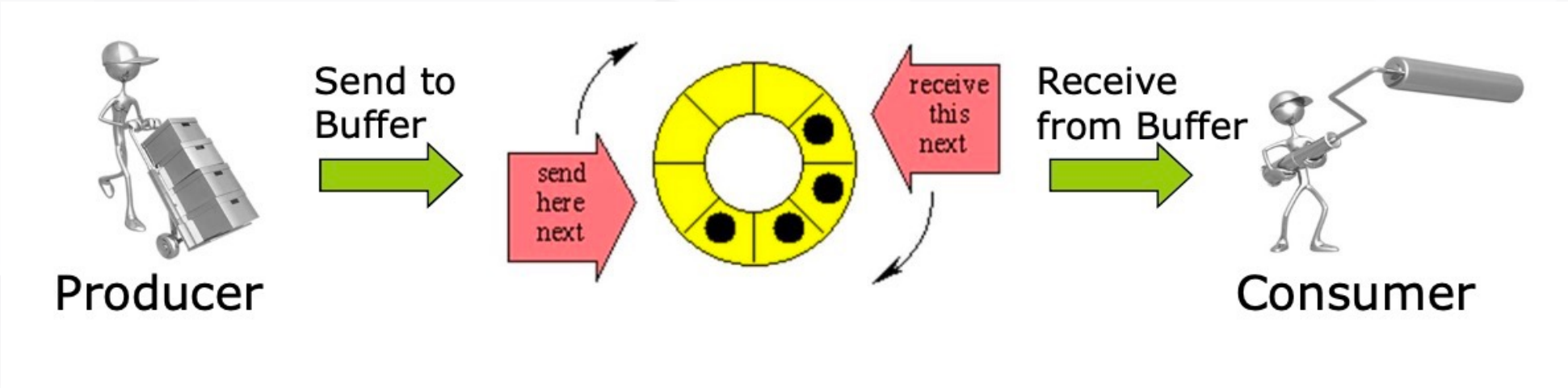
songzhuoran@sjtu.edu.cn

经典同步问题

- 有界缓冲问题
- 读者-作者问题
- 哲学家就餐问题
- Windows同步
- Linux同步
- Pthreads同步

有界缓冲问题

- 有n个缓冲区，不能超过缓冲区访问数据



使用信号量解决有界缓冲问题

- 共享数据结构
  - mutex=1
  - empty=n
  - full=0

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

判断是否写满n个缓冲区

获得锁

生产者

```
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);
```

消费者

读者-作者问题

- 读数据库的进程：读者
  - 不需要修改数据，允许多个读者同时
- 写数据库的进程：作者
  - 读取和修改数据，只允许一个作者
- 当作者在写入数据库同时读者希望访问数据库，产生读者-作者问题
- 要求作者在写入数据库时具有共享数据库独占访问权
- “第一”读者-作者问题
  - 要求读者不应保持等待，除非作者已获得权限使用共享对象
- “第二”读者-作者问题
  - 一旦作者就绪，就会尽快执行

“第一”读者-作者问题的解答方法 （读者优先）

- 共享数据结构
  - wrt=1, readcount=0（有多少个读者），mutex=1
- 一个读者在wrt上等待，n-1个读者在mutex上等待

```
do {
        wait (wrt) ;

        //    writing is performed

        signal (wrt) ;
} while (TRUE);
```

作者

```
do {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)
            wait (wrt) ;
        signal (mutex)

        // reading is performed

        wait (mutex) ;
        readcount  - - ;
        if (readcount  == 0)
            signal (wrt) ;
        signal (mutex) ;
} while (TRUE);
```

确保对readcount互斥保护

确保没有作者在写入

确保对readcount互斥保护

所有读者完成读操作释放锁

读者

“第二”读者-作者问题的解答方法（作者优先）

- 共享数据结构
  - int readcount = 0, writecount = 0;
  - mutexrc = 1, mutexwc = 1, wrt = 1, rd = 1;

作者

```
do {
    wait (mutexwc);
    writecount ++;
    if (writecount == 1)
        wait (rd);
    signal (mutexwc);

    wait (wrt);
    // writing is performed
    signal(wrt);

    wait (mutexwc);
    writecount - -;
    if (writecount == 0)
        signal (rd);
    signal (mutexwc);
} while (TRUE);
```

确保没有读者在读

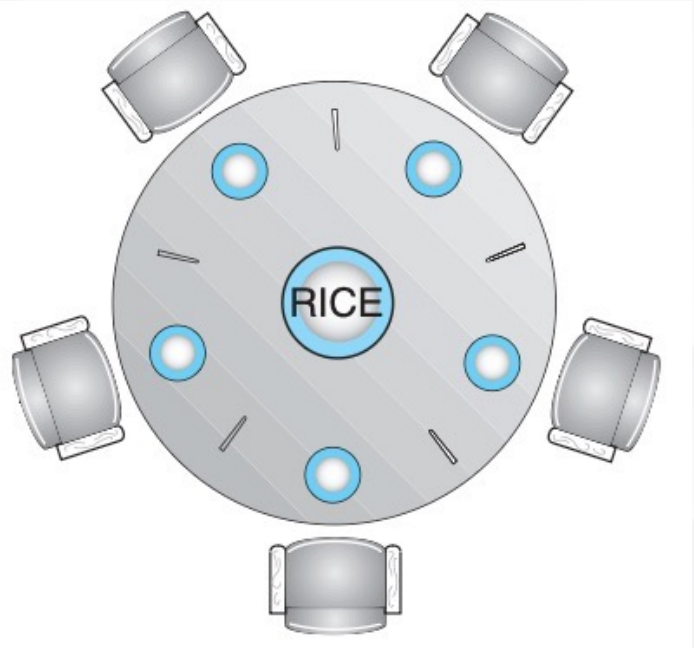
```
do {
    wait (rd);
    wait (mutexrc);
    readcount ++;
    if (readcount == 1)
        wait (wrt);
    signal (mutexrc);
    signal (rd);

    //reading is performed

    wait (mutexrc);
    readcount - -;
    if (readcount == 0)
        signal (wrt);
    signal (mutexrc);
} while (TRUE);
```

增加了rd变量，只有当作者都被处理完，才会进入下面的wait(mutexrc)

读者

哲学家就餐问题

- 桌上有5根筷子，当一个哲学家拿起两根筷子就可以吃饭，但互相不交流
- 经典的同步问题



哲学家就餐问题

- 共享数据结构
  - chopstick[5]，均初始化为1
- 可以确保两个邻居不同时进食，但可能导致死锁
  - 假设5个哲学家同时拿起左边的筷子，此时所有筷子的信号量为0，但当他们试图拿起右边的筷子，都会被推迟

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for awhile */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
} while (true);
```

死锁！

哲学家就餐问题

- 改进一下
  - 5个哲学家同时拿起左边的筷子，同时放下，同时等待

```
do{
        wait(chopstick[i]);
        if(chopstick[(i+1)%5]){
                wait(chopstick[[(i+1)%5]);
                break;
        }
        else{
                signal(chopstick[i]);
                wait some time();
        }
}while(true);
```

死锁！

哲学家就餐问题

- 改进一下
  - 5个哲学家同时拿起左边的筷子，同时放下，随机等待一段时间

```
do{
        wait(chopstick[i]);
        if(chopstick[(i+1)%5]){
                wait(chopstick[[(i+1)%5]);
                break;
        }
        else{
                signal(chopstick[i]);
                wait random time();
        }
} while(true);
```

可行，但不够完美

哲学家就餐问题

- 改进一下
  - 将拿筷子的部分保护起来，利用信号量mutex

```
do{
        wait(mutex);
        wait(chopstick[i]);
        if(chopstick[(i+1)%5]){
                wait(chopstick[[(i+1)%5]);
                break;
        }
        else{
                signal(chopstick[i]);
                wait random time();
        }
        signal(mutex);
} while(true);
```

可行，但只允许一位哲学家进食

哲学家就餐问题

- 改进一下
  - 要么不拿，要么一次性拿两个筷子

S1 哲学家进入饥饿状态

S2 如果左邻居或右邻居正在进餐，则等待；

否则转S3

S3 拿起两个筷子

S4 吃饭

S5 放下左右两个筷子

S6 重新循环

哲学家就餐问题

- 改进一下
  - 要么不拿，要么一次性拿两个筷子

哲学家自己如何解决这个问题

S1 哲学家进入饥饿状态

S2 如果左邻居或右邻居正在进餐，则等待；否则转S3

S3 拿起两个筷子

S4 吃饭

S5 放下左右两个筷子

S6 重新循环

计算机程序如何解决这个问题

S1 哲学家进入饥饿状态

S2 如果左邻居或右邻居正在进餐，则进程进入阻塞态；否则转S3

S3 拿起两个筷子

S4 吃饭

S5 放下左边的筷子，看看左边的邻居是否能进餐（饥饿状态、两个筷子都在），若能则唤醒它

S6 放下右边的筷子，看看右边的邻居是否能进餐，若能则唤醒它

S7 重新循环

哲学家就餐问题

- 共享数据结构
  - int **state**[5] （临界资源）
  - mutex = 1, s[N]=0;

```
void philosopher(int i){
        do{
                think();
                wait_chop(i);
                eat();
                put_chop(i);
        } while(true);
}

void test_take_chopstick(int i){
        if(state[i]== HUNGRY && state[left]!=EATING && state[right]!=EATING{
        state[i]=EATING;   两个叉子到手了
        signal(s[i]);          通知第i人可以吃饭了，s[i]=1
        }
}
```

```
void wait_chop(int i){
        wait(mutex);
        state[i]=HUNGRY;
        test_take_chopstick(i);
        signal(mutex);
        wait(s[i]);
}
```

状态是一个需要被互斥保护的

没有拿到叉子便阻塞自己

```
void put_chop(int i){
        wait(mutex);
        state[i]=THINKING;   把叉子放回去
        test_take_chopstick(left);
        test_take_chopstick(right);
        signal(mutex);
}
```

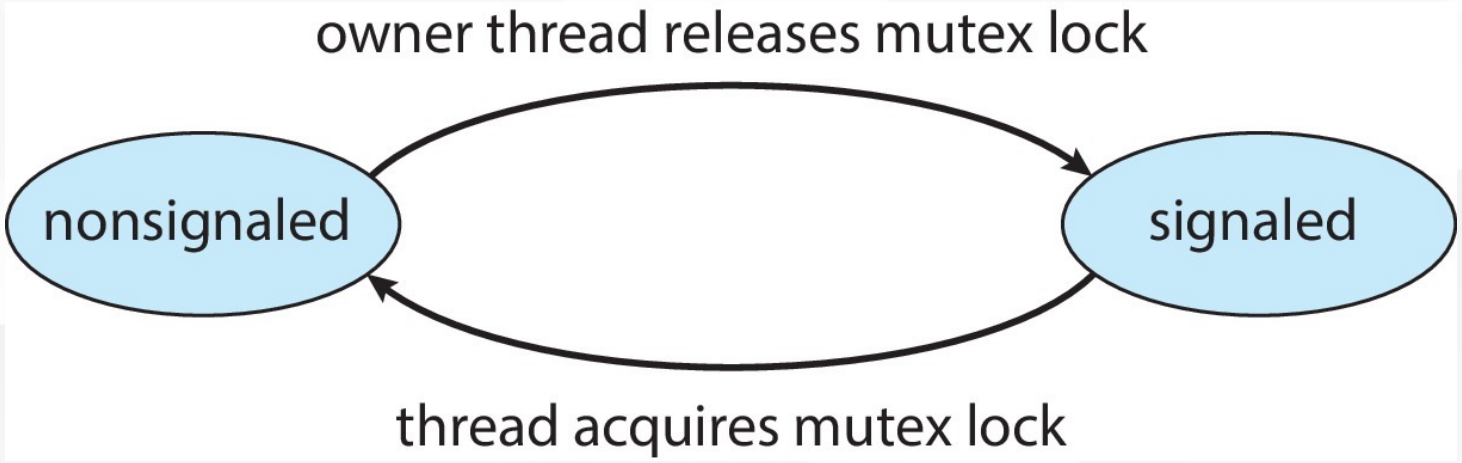
试图唤醒其他人

Windows同步

- 对于多处理器系统，Windows采用自旋锁来保护访问全局资源
  - 内核保证绝不抢占拥有自旋锁的线程
- 针对内核外的线程同步，Windows提供**调度对象**；采用调度对象，有多种不同的线程同步机制，包括互斥锁、信号量、事件、定时器等
  - 事件：类似环境变量（condition variable），当所需变量可用时，通知等待线程
  - 定时器：在一定时间到达时通知一个或多个线程

Windows同步

- 调度对象可以处于触发状态或非触发状态
  - 触发状态（signaled state）：对象可用，线程在获取它时不会被阻塞
  - 非触发状态（nonsignaled state）：对象不可用，线程在获取它时会被阻塞



Linux同步

- Linux2.6版本前，Linux为非抢占内核
- 目前的Linux系统为完全可抢占内核
- Linux系统为进程同步提供：
  - 原子整数
  - 互斥锁
  - 信号量

Linux同步

- 原子整数（atomic integer）
  - 类型为抽象数据类型atomic_t
  - 所有采用原子整数的数学运算在执行时不会中断

```
atomic_t counter;
int value;
```

Atomic Operation	*Effect*
`atomic_set(&counter,5);`	`counter = 5`
`atomic_add(10,&counter);`	`counter = counter + 10`
`atomic_sub(4,&counter);`	`counter = counter - 4`
`atomic_inc(&counter);`	`counter = counter + 1`
`value = atomic_read(&counter);`	`value = 12`

Linux同步

- Linux中的互斥锁
  - 当一个任务在进入临界区前，应调用mutex_lock()
  - 当退出临界区后，调用mutex_unlock()
  - 如果互斥锁不可用，调用mutex_lock的任务会变成睡眠状态，当锁的所有者调用mutex_unlock后，它会被唤醒

Pthreads同步

- Pthreads API只能被用于用户级程序员，不能用于任何特定内核，提供：
  - 互斥锁
  - 条件变量
  - 读写锁
- 互斥锁
  - 采用的数据类型为pthreads_mutex_t
  - 通过pthreads_mutex_init()创建互斥锁

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex,NULL);
```

Pthreads同步

- 互斥锁
  - 通过函数pthread_mutex_lock()或pthread_mutex_unlock()，获取或释放互斥锁
  - 当调用pthread_mutex_lock()时，如果互斥锁不可以，则调用线程阻塞，直到所有者调用pthread_mutex_unlock()

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

Pthreads同步

- 信号量
  - 命名信号量
  - 无名信号量
- 两者的根本区别：命名信号量在文件系统中有实际名称，并**能被多个不相关进程所共享**，而无名信号量只能被同一进程的线程所使用

Pthreads同步

- 命名信号量，创建和初始化

```
#include <semaphore.h>
sem_t *sem;

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

- 其他进程想访问该信号量，通过其名称sem
- 获取和释放该信号量：

```
/* acquire the semaphore */
sem_wait(sem);

/* critical section */

/* release the semaphore */
sem_post(sem);
```

Pthreads同步

- 无名信号量，创建和初始化

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

- 获取和释放该信号量：

```
/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

/* release the semaphore */
sem_post(&sem);
```