

## 课堂测试— 限时20分钟以内

1. 请说明什么是用户态与核心态。
2. 进程运行态、阻塞态及就绪态这三个状态之间在理论上有6种可能的相互转换，但其中有两种不可能存在，请问是哪两种并说明原因。
3. 我们在编程作业中用到的函数，`printf`属于C语言库函数，`write`属于操作系统提供的系统调用函数。在没有其它任何信息的情况下，请你判断`getpid()`(获取进程编号)是库函数还是系统调用？说明你的理由。
4. 什么是进程上下文切换？如果让你实现上下文切换功能，你会采用下列编程语言中的哪一种：**Java, C, C++**, 汇编语言。说明你选择该编程语言的原因。

# 临界资源与临界区

考虑网络订票软件:

- ▶ 进程A发现3号车10C座位空闲
- ▶ 此时操作系统调度进程B运行
- ▶ 进程B同样发现该位子的票尚未售出, 于是将该票买给旅客
- ▶ 进程A重新运行后, 再次将3-10C售出

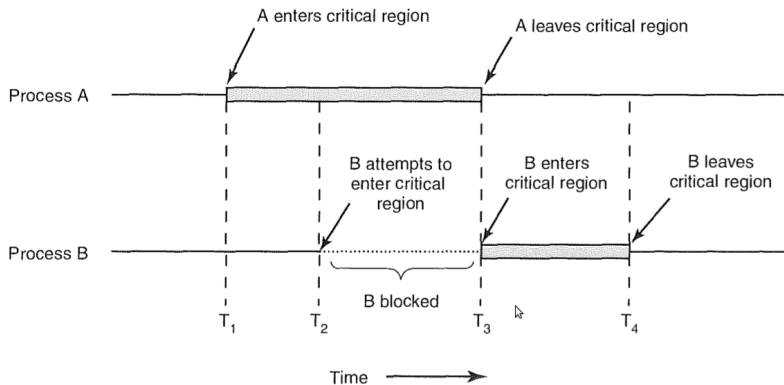


# 对临界资源的互斥访问

理想的互斥方案需要满足4个条件:

1. 两个进程不能同时进入临界区
2. 不能依赖CPU数目或者运行速度
3. 不在临界区的进程，不能妨碍其他进程进入临界区
4. 任一进程需在有限时间内能够进入临界区

# 对临界资源的互斥访问



## 对临界资源的互斥访问: 方法1 – 交替进入临界区

```
while (TRUE) {  
    while (turn != 0)    /* loop */;  
    critical_region();  
    turn = 1;  
    noncritical_region();  
}
```

(a)

```
while (TRUE) {  
    while (turn != 1)    /* loop */;  
    critical_region();  
    turn = 0;  
    noncritical_region();  
}
```

(b)

缺陷: 违反条件3 (设想进程A循环体运行1秒, 进程B运行100秒)  
进程A与B必须锁步(交替)进入临界区

## 对临界资源的互斥访问: 方法2 – 忙等待

- ▶ 需要硬件支持TSL指令
- ▶ 进程进入临界区前, 调用enter\_region
- ▶ 离开临界区时, 调用leave\_region
- ▶ 这是一个正确的解决方法, 但是...

```
enter_region:
    TSL REGISTER, LOCK
    CMP REGISTER, #0
    JNE enter_region
    RET
```

```
leave_region:
    MOVE LOCK, #0
    RET
```

## 对临界资源的互斥访问: 方法2 – 忙等待

两个缺点:

- ▶ 缺点1: 忙等待浪费了CPU时间
- ▶ 缺点2: 优先级反转问题
  - ▶ 进程H优先级高于进程L, 二者同时需要某临界资源
  - ▶ 假设当进程L在临界区时, 进程H可以运行
  - ▶ 结局: 进程L永远无法离开临界区, H永远忙等待

为了克服这些缺点, 增加sleep和wakeup系统调用

# 生产者-消费者问题

```
#define N 100
```

```
int count = 0;
```

```
void producer(void)
```

```
{
```

```
    int item;
```

```
    while (TRUE) {
```

```
        item = produce_item();
```

```
        if (count == N) sleep();
```

```
        insert_item(item);
```

```
        count = count + 1;
```

```
        if (count == 1) wakeup(consumer);
```

```
    }
```

```
}
```

```
void consumer(void)
```

```
{
```

```
    int item;
```

```
    while (TRUE) {
```

```
        if (count == 0) sleep();
```

```
        item = remove_item();
```

```
        count = count - 1;
```

```
        if (count == N - 1) wakeup(producer);
```

```
        consume_item(item);
```

```
    }
```

```
}
```

缺陷: 测试`count==0`成功后, 消费者进程调用`sleep`之前, 调度生产者进程运行...



# 信号量机制(Semaphores)

为了解决唤醒信号丢失的问题，引入信号量，它是一种特殊的整型变量。在信号量上定义两个原子操作：

**down** 如果信号量值大于0，则将其减1然后返回；否则，进程在该信号量上进入睡眠

**up** 如果有进程在该信号量上睡眠，则选择其中一个唤醒；否则，信号量加1

# 用信号量解决生产者-消费者问题

```
#define N 100
```

```
typedef int semaphore;
```

```
semaphore mutex = 1;
```

```
semaphore empty = N;
```

```
semaphore full = 0;
```

```
void producer(void)
```

```
{
```

```
    int item;
```

```
    while (TRUE) {
```

```
        item = produce_item( );
```

```
        down(&empty);
```

```
        down(&mutex);
```

```
        insert_item(item);
```

```
        up(&mutex);
```

```
        up(&full);
```

```
    }
```

```
}
```

```
void consumer(void)
```

```
{
```

```
    int item;
```

```
    while (TRUE) {
```

```
        down(&full);
```

```
        down(&mutex);
```

```
        item = remove_item( );
```

```
        up(&mutex);
```

```
        up(&empty);
```

```
        consume_item(item);
```

```
    }
```

```
}
```

该方案中，信号量**empty**和**full**具有计数和同步功能，而**mutex**仅有互斥功能。

## 专门用来实现互斥的特殊信号量- 互斥锁

互斥锁只有两种状态: **locked (1) / unlocked (0)**

mutex\_lock:

TSL REGISTER,MUTEX

CMP REGISTER,#0

JZE ok

CALL thread\_yield

JMP mutex\_lock

ok: RET

mutex\_unlock:

MOVE MUTEX,#0

RET

## 互斥锁与忙等待的区别

```
mutex_lock:
    TSL REGISTER,MUTEX
    CMP REGISTER,#0
    JZE ok
    CALL thread_yield
    JMP mutex_lock
ok:      RET
```

```
mutex_unlock:
    MOVE MUTEX,#0
    RET
```

```
enter_region:
    TSL REGISTER,LOCK
    CMP REGISTER,#0
    JNE enter_region
    RET
```

```
leave_region:
    MOVE LOCK,#0
    RET
```

后者：不断利用**CPU**指令测试临界资源，直至时间片用光被从**CPU**上撤下来

# 信号量的危险情形— 管程机制的引入

```
#define N 100
```

```
typedef int semaphore;
```

```
semaphore mutex = 1;
```

```
semaphore empty = N;
```

```
semaphore full = 0;
```

```
void producer(void)
```

```
{
```

```
    int item;
```

```
    while (TRUE) {
```

```
        item = produce_item( );
```

```
        down(&empty);
```

```
        down(&mutex);
```

```
        insert_item(item);
```

```
        up(&mutex);
```

```
        up(&full);
```

```
    }
```

```
}
```

```
void consumer(void)
```

```
{
```

```
    int item;
```

```
    while (TRUE) {
```

```
        down(&full);
```

```
        down(&mutex);
```

```
        item = remove_item( );
```

```
        up(&mutex);
```

```
        up(&empty);
```

```
        consume_item(item);
```

```
}
```

```
}
```

**危险:**如果程序员不小心把producer中

的down(empty)和down(mutex)顺序颠倒, 则当缓冲区满时, 会发生什么?

# 信号量的危险情形——管程机制的引入

- ▶ 发生死锁。
- ▶ 因此，最好由编译器自动处理这种容易出错的程序段。——引入管程。
- ▶ 对比：C++中构造函数与析构函数

# 管程:解决生产者-消费者问题

**monitor** *ProducerConsumer*

**condition** *full, empty;*

**integer** *count;*

**procedure** *insert(item: integer);*

**begin**

**if** *count = N* **then wait**(*full*);

*insert\_item(item);*

*count := count + 1;*

**if** *count = 1* **then signal**(*empty*)

**end;**

**function** *remove: integer;*

**begin**

**if** *count = 0* **then wait**(*empty*);

*remove = remove\_item;*

*count := count - 1;*

**if** *count = N - 1* **then signal**(*full*)

**end;**

*count := 0;*

**end monitor;**

**procedure** *producer;*

**begin**

**while true do**

**begin**

*item = produce\_item;*

*ProducerConsumer.insert(item)*

**end**

**end;**

**procedure** *consumer;*

**begin**

**while true do**

**begin**

*item = ProducerConsumer.remove;*

*consume\_item(item)*

**end**

**end;**

**注意概念:** 条件变量empty, full以及wait, signal  
此外, insert与remove之间的互斥由编译器完成

# 管程:解决生产者-消费者问题

- ▶ 管程内程序段之间的互斥（自动）
- ▶ 进程同步问题?: 条件变量及wait, signal实现
  - ▶ wait: 将当前进程阻塞, 并允许其他进程进入管程
  - ▶ signal: 将被相应条件变量阻塞的进程唤醒
- ▶ 上述方法中, signal必须是最后一条指令, 为什么?



# 消息传递机制：解决不同机器上进程间同步问题

```
#define N 100
```

```
void producer(void)
```

```
{
    int item;
    message m;

    while (TRUE) {
        item = produce_item();
        receive(consumer, &m);
        build_message(&m, item);
        send(consumer, &m);
    }
}
```

```
void consumer(void)
```

```
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m);
    while (TRUE) {
        receive(producer, &m);
        item = extract_item(&m);
        send(producer, &m);
        consume_item(item);
    }
}
```