

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



竞争条件

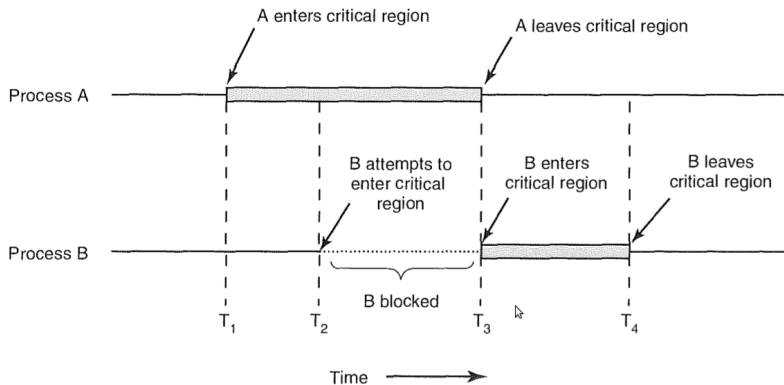
多个进程并发访问和操作同一数据且执行结果与访问的特定顺序有关，称为竞争条件。

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);  
    }  
}
```

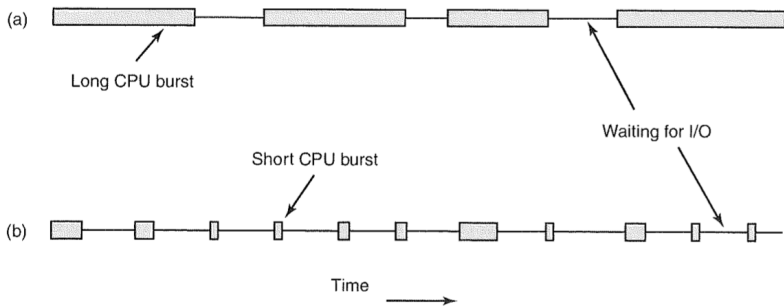
```
void consumer(void)
```

```
{  
    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);  
    }  
}
```

进程（线程）调度

当系统中有多个进程或线程处于就绪态时，操作系统需要从中选择一个放到**CPU**上运行。这就是进程调度问题。实现该任务的部件称作调度器。

进程的典型行为: CPU密集型与IO密集型进程



思考：两种进程举例？

二者关键区别：不是I/O时间长度，而是CPU时间长度

进程的典型行为: CPU密集型与IO密集型进程

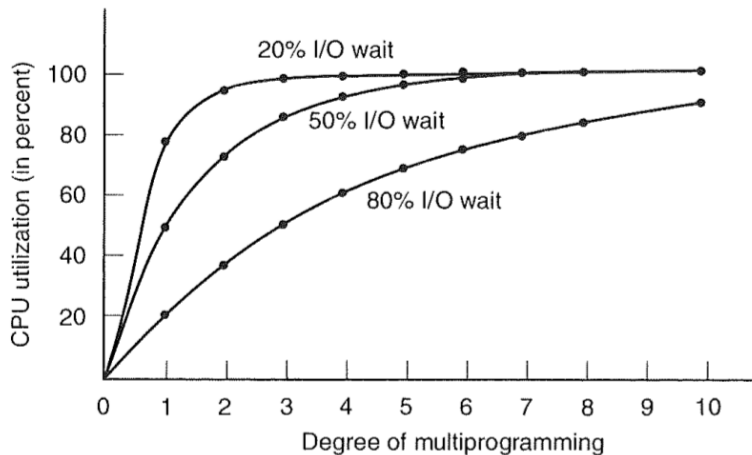
CPU速度增长快于I/O速度增长，因此随着技术发展，进程倾向于越来越I/O密集。后果：I/O密集型进程的调度显得越来越关键。

进程的典型行为: CPU密集型与IO密集型进程

基本想法: 如果某I/O进程处于就绪态, 则应该努力优先让其运行。为什么? (这里有个深刻原因)

思考2: 如何动态识别某进程是CPU密集型还是I/O密集型?

进程的典型行为: CPU密集型与IO密集型进程



进程调度的时机：何时调度？

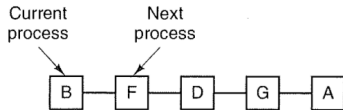
1. 新进程创建时，是继续运行父进程，还是运行新创建的子进程？
2. 当前进程退出时，CPU空闲，此时需从就绪态进程集合中选择一个运行
3. 当前进程阻塞时（I/O或者信号量引起）
4. 当发生I/O中断时，由此I/O信号导致阻塞的进程进入就绪态

抢占式调度与非抢占式调度

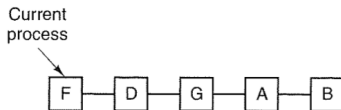
- ▶ 时钟硬件中断信号的频率大约为**50~60Hz**
- ▶ 在**1**个或者**K**个时钟中断信号处，强迫终止当前运行的进程。这类调度称为抢占式调度
- ▶ 非抢占式调度：进程一旦运行，则除非它阻塞或者自愿放弃**CPU**，不剥夺其**CPU**使用权。
- ▶ 抢占式调度用于分时系统；需要**时钟硬件**的支持

时间片轮转调度算法

- ▶ 最简单、最古老、最公平、广泛使用
- ▶ 时间片长度设置:
 - ▶ 时间片不能太短，否则进程切换开销比例太大
 - ▶ 时间片太长，则导致交互式使用时等待时间过久



(a)

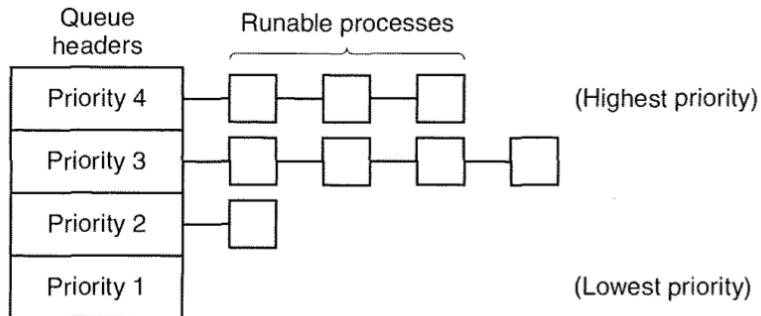


(b)

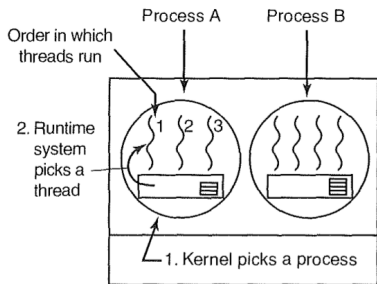
基于优先级的调度算法

- ▶ 时间片轮转算法的假设：所有进程都同等重要
- ▶ 有时，有些进程比较重要（校长、院长、主任、教授、秘书、清洁工、学生）
- ▶ 优先级调度算法：从就绪进程集合中选择优先级最高的进程运行
- ▶ 关键数据结构：优先队列
- ▶ 为避免优先级高的进程独霸CPU，可以动态调整优先级
- ▶ 为照顾I/O密集型进程，优先级可以设置为 $1/f$ ，其中 f 为进程在上一时间片中实际占用CPU的时间比例

优先级调度与时间片轮转相结合的调度算法



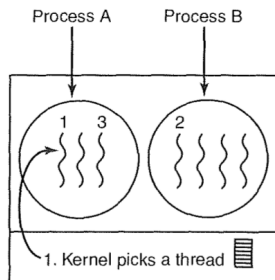
线程调度



Possible: A1, A2, A3, A1, A2, A3

Not possible: A1, B1, A2, B2, A3, B3

(a)



Possible: A1, A2, A3, A1, A2, A3

Also possible: A1, B1, A2, B2, A3, B3

(b)