



操作系统

Operating Systems

并发进程的正确性

■ 独立进程

- ▶ 不和其他进程共享资源或状态
- ▶ **确定性**⇒ 输入状态决定结果
- ▶ **可重现**⇒ 能够重现起始条件
- ▶ 调度顺序不重要

■ 并发进程

- ▶ 在多个进程间有资源共享
- ▶ 不确定性
- ▶ 不可重现

■ 并发进程的正确性

- ▶ 执行过程是不确定性和不可重现的
- ▶ 程序错误可能是间歇性发生的

进程并发执行的好处

- 进程需要与计算机中的其他进程和设备进行协作
- 好处1：共享资源
- 好处2：加速
 - ▶ 多个用户可以使用同一台计算机
- 好处3：模块化
 - ▶ 程序可以分解成许多模块，每个模块可以由不同的处理器并行执行
 - ▶ 以编译为例，gcc会调用cpp, cc1, cc2, as, ld
 - ▶ 使系统易于复用和扩展

并发创建新进程时的标识分配

- 程序可以调用函数fork()来创建一个新的进程
 - ▣ 操作系统需要分配一个新的并且唯一的进程ID
 - ▣ 在内核中，这个系统调用会运行

```
new_pid = next_pid++
```

- ▣ 翻译成机器指令

```
LOAD next_pid Reg1  
STORE Reg1 new_pid  
INC Reg1  
STORE Reg1 next_pid
```

- 两个进程并发执行时的预期结果(假定next_pid=100)
 - ▣ 一个进程得到的ID应该是100
 - ▣ 另一个进程的ID应该是101
 - ▣ next_pid应该增加到102

新进程分配标识中的可能错误

进程A

```
LOAD next_pid Reg1  
STORE Reg1 new_pid
```

```
INC Reg1  
STORE Reg1 next_pid
```

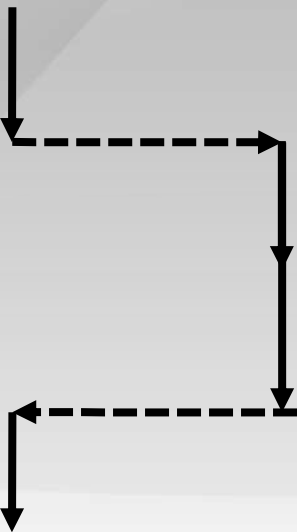
new_pid=200

进程B

```
LOAD next_pid Reg1  
STORE Reg1 new_pid  
INC Reg1  
STORE Reg1 next_kpid
```

new_pid=200

next_pid=100



原子操作 (Atomic Operation)

- 原子操作是指一次不存在任何中断或失败的操作
 - ▣ 要么操作成功完成
 - ▣ 或者操作没有执行
 - ▣ 不会出现部分执行的状态
- 操作系统需要利用同步机制在并发执行的同时, 保证一些操作是原子操作



操作系统

Operating Systems



操作系统

Operating Systems

现实生活中的同步问题

■ 操作系统和现实生活的问题类比

- ▣ 利用现实生活问题帮助理解操作系统同步问题
- ▣ 同时注意，计算机与人的差异

■ 例如: 家庭采购协调

时 间	A	B
3:00	查看冰箱，没有面包了	
3:05	离开家去商店	
3:10	到达商店	查看冰箱，没有面包了
3:15	购买面包	离开家去商店
3:20	到家，把面包放进冰箱	到达商店
3:25		购买面包
3:30		到家，把面包放进冰箱

家庭采购协调问题分析

■ 如何保证家庭采购协调的成功和高效

▣ 有人去买

需要采购时，有人去买面包

▣ 最多只有一个人去买面包

■ 可能的解决方法

▣ 在冰箱上设置一个**锁和钥匙**（lock&key）

▣ 去买面包之前锁住冰箱并且拿走钥匙

■ 加锁导致的新问题

▣ 冰箱中还有其他食品时，别人无法取到

方案一

- 使用**便签**来避免购买太多面包
 - ▣ 购买之前留下一张便签
 - ▣ 买完后移除该便签
 - ▣ 别人看到便签时，就不去购买面包

```
if (nobread) {  
    if (noNote) {  
        leave Note;  
        buy bread;  
        remove Note;  
    }  
}
```

- 有效吗？

方案一分析

■ 偶尔会购买太多面包

- ▣ 检查面包和便签后帖便签前，有其他人检查面包和便签

■ 解决方案只是间歇性地失败 进程B

if (noblread) {
if (noNote) {

- ▣ 问题难以调试
- ▣ 必须考虑调度器所做的事情

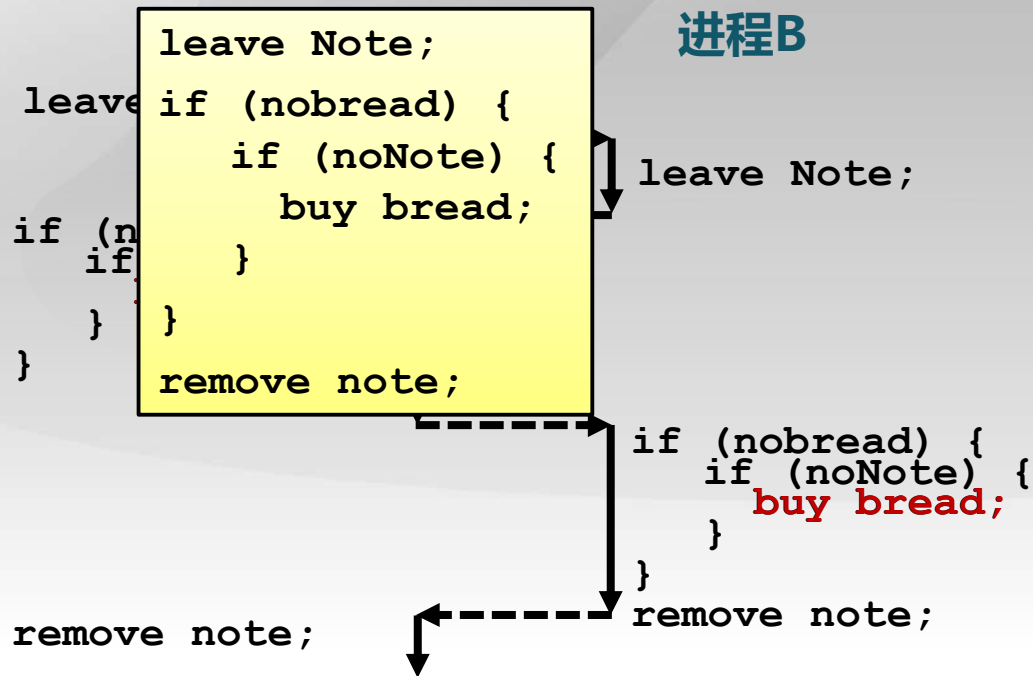
```
leave Note;  
buy bread;  
remove Note;  
}
```

```
if (noblread) {  
if (noNote) {
```

```
leave Note;  
buy bread;  
remove Note;  
}
```

方案二

■ 先留便签，后检查面包和便签



■ 会发生什么?

■ 不会有人买面包

方案三

■ 会便生增加标记，以区别不同人的便签

▶ 可能导致没有人去买面包

▶ 每个人都将认为另外一个去买面包

进程A

进程B

```
leave note_1;
if (no note_2) {
  if (no bread) {
    buy bread;
  }
  buy bread;
}
remove note_1;
```

```
leave note_2;
if (no note_1) {
  if (no bread) {
    buy bread;
  }
  if (no bread) {
    buy bread;
  }
  remove note_2;
```

remove note_1;

remove note_2;

方案四

■ 两个人采用不同的处理流程

进程A

```
leave note_1;  
while(note_2) {  
    do nothing;  
}  
if(no bread) {  
    buy bread;  
}  
remove note_1;
```

如果没有便
签2,那么A可
以去买面包,
否则等待B离
开

进程B

```
leave note_2;  
if(no note_1) {  
    if(no bread) {  
        buy bread;  
    }  
}  
remove note_2;
```

如果没有便
签1,那么B可
以去买面包,
否则B离开并
且再试一次

■ 现在有效吗?

▣ 枚举所有可能后, 可以确认它是有效的

■ 这种解决方案你满足吗?

方案四分析

- 它有效，但太复杂
 - ▣ 很难验证它的有效性
- A和B的代码不同
 - ▣ 每个进程的代码也会略有不同
 - ▣ 如果进程更多，怎么办？
- 当A在等待时，它不能做其他事
 - ▣ 忙等待 (busy-waiting)
- 有更好的方法吗？

方案五

■ 利用两个原子操作实现一个锁(lock)

- ▣ Lock.Acquire()

- ▣ 在锁被释放前一直等待，然后获得锁

- ▣ 解锁时唤醒等待的进程，并且同时发现锁被释放了，那么只有一个能够获得锁

■ 基于原子Release方案

```
breadlock.Acquire(); 进入临界区
```

```
if (nobread) {
```

```
    buy bread;
```

临界区

```
}
```

```
breadlock.Release(); 退出临界区
```

进程的交互关系：相互感知程度

相互感知的程度	交互关系	进程间的影响
相互不感知（完全不了解其它进程的存在）	独立	一个进程的操作对其他进程的结果无影响
间接感知（双方都与第三方交互，如共享资源）	通过共享进行协作	一个进程的结果依赖于共享资源的状态
直接感知（双方直接交互，如通信）	通过通信进行协作	一个进程的结果依赖于从其他进程获得的信息

- 互斥 (mutual exclusion)
- 死锁 (deadlock)
- 饥饿 (starvation)

进程的交互关系：相互感知程度

相互感知的程度	交互关系	进程间的影响
相互不感知（完全不了解其它进程的存在）	独立	一个进程的操作对其他进程的结果无影响
间接感知（双方都与第三方交互，如共享资源）	通过共享进行协作	一个进程的结果依赖于共享资源的状态
直接感知（双方直接交互，如通信）	通过通信进行协作	一个进程的结果依赖于从其他进程获得的信息

- 互斥（mutual exclusion）
 - ▶ 一个进程占用资源，其它进程不能使用
- 死锁（deadlock）
- 饥饿（starvation）

进程的交互关系：相互感知程度

相互感知的程度	交互关系	进程间的影响
相互不感知（完全不了解其它进程的存在）	独立	一个进程的操作对其他进程的结果无影响
间接感知（双方都与第三方交互，如共享资源）	通过共享进行协作	一个进程的结果依赖于共享资源的状态
直接感知（双方直接交互，如通信）	通过通信进行协作	一个进程的结果依赖于从其他进程获得的信息

- 互斥 (mutual exclusion)
- 死锁 (deadlock)
 - ▶ 多个进程各占用部分资源，形成循环等待
- 饥饿 (starvation)

进程的交互关系：相互感知程度

相互感知的程度	交互关系	进程间的影响
相互不感知（完全不了解其它进程的存在）	独立	一个进程的操作对其他进程的结果无影响
间接感知（双方都与第三方交互，如共享资源）	通过共享进行协作	一个进程的结果依赖于共享资源的状态
直接感知（双方直接交互，如通信）	通过通信进行协作	一个进程的结果依赖于从其他进程获得的信息

- 互斥 (mutual exclusion)

- 死锁 (deadlock)

- 饥饿 (starvation)

- 其他进程可能轮流占用资源，一个进程一直得不到资源



操作系统

Operating Systems



操作系统

Operating Systems

临界区(Critical Section)

```
entry section
critical section
exit section
remainder section
```

- 临界区(critical section)
- 进程访问临界资源的一段需要互斥执行的代码
- 检查可否进入临界区的一段代码
- 进程进入临界区时设置标志“临界区”标志
- 代码中的其余部分

临界区的访问规则

- 空闲则入
- 忙则等待
- 有限等待
- 让权等待 (可选)

临界区的访问规则

- 空闲则入
 - ▣ 没有进程在临界区时，任何进程可进入
- 忙则等待
- 有限等待
- 让权等待（可选）

临界区的访问规则

- 空闲则入
- 忙则等待
 - ▣ 有进程在临界区时，其他进程均不能进入临界区
- 有限等待
- 让权等待（可选）

临界区的访问规则

- 空闲则入
- 忙则等待
- 有限等待
 - ▣ 等待进入临界区的进程不能无限期等待
- 让权等待（可选）

临界区的访问规则

- 空闲则入
- 忙则等待
- 有限等待
- 让权等待（可选）
 - ▣ 不能进入临界区的进程，应释放CPU（如转换到阻塞状态）

临界区的实现方法

- 禁用中断
- 软件方法
- 更高级的抽象方法

- 不同的临界区实现机制的比较
 - ▣ 性能：并发级别

提纲

- 背景
- 现实生活中的同步问题
- 临界区
- 方法1: 禁用硬件中断
- 方法2: 基于软件的解决方法
- 方法3: 更高级的抽象方法

方法1：禁用硬件中断

- 没有中断，没有上下文切换，因此没有并发
 - ▣ 硬件将中断处理延迟到中断被启用之后
 - ▣ 现代计算机体系结构都提供指令来实现禁用中断

```
local_irq_save(unsigned long flags);  
critical section  
local_irq_restore(unsigned long flags);
```

- 进入临界区
 - ▣ 禁止所有中断，并保存标志
- 离开临界区
 - ▣ 使能所有中断，并恢复标志

缺点

- 禁用中断后，进程无法被停止
 - ▣ 整个系统都会为此停下来
 - ▣ 可能导致其他进程处于饥饿状态
- 临界区可能很长
 - ▣ 无法确定响应中断所需的时间（可能存在硬件影响）
- 要小心使用



操作系统

Operating Systems



操作系统

Operating Systems

基于软件的同步解决方法

两个线程，T0和T1

线程Ti的代码

```
do {  
    enter section 进入区  
        critical section  
    exit section 退出区  
        reminder section  
} while (1);
```

线程可通过共享一些共有变量来同步它们的行为

第一次尝试

■ 共享变量

```
int turn = 0;  
turn == i // 表示允许进入临界区的线程
```

■ 线程Ti的代码

```
do {  
    while (turn != i) ;  
    critical section  
    turn = j;  
    reminder section  
} while (1);
```

■ 满足“忙则等待”，但是有时不满足“空闲则入”

- Ti不在临界区，Tj想要继续运行，但是必须等待Ti进入过临界区后

第二次尝试

■ 共享变量

```
int flag[2];  
flag[0] = flag[1] = 0;  
flag[i] == 1 //表示线程Ti是否在临界区
```

■ 线程Ti的代码

```
do {  
    while (flag[j] == 1) ;  
    flag[i] = 1;  
    critical section  
    flag[i] = 0;  
    remainder section  
} while (1);
```

■ 不满足“忙则等待”

第三次尝试

■ 共享变量

```
int flag[2];  
flag[0] = flag[1] = 0;  
flag[i] == 1 // 表示线程Ti想要进入临界区
```

■ 线程Ti的代码

```
do {  
    flag[i] = 1;  
    while (flag[j] == 1) ;  
    critical section  
    flag[i] = 0;  
    remainder section  
} while (1);
```

■ 满足“忙则等待”，但是不满足“空闲则入”

Peterson算法

- 满足线程 T_i 和 T_j 之间互斥的经典的基于软件的解决方法（1981年）

- 共享变量

```
int turn; //表示该谁进入临界区  
boolean flag[]; //表示进程是否准备好进入临界区
```

- 进入区代码

```
flag[i] = true;  
turn = j;  
while (flag[j] && turn == j)
```

- 退出区代码

```
flag[i] = false;
```


Peterson算法实现

线程Ti 的代码

```
do {  
    flag[i] = true;  
    turn = j;  
    while ( flag[j] && turn == j);  
  
    CRITICAL SECTION  
  
    flag[i] = false;  
  
    REMAINDER SECTION  
  
} while (true);
```

Dekkers算法

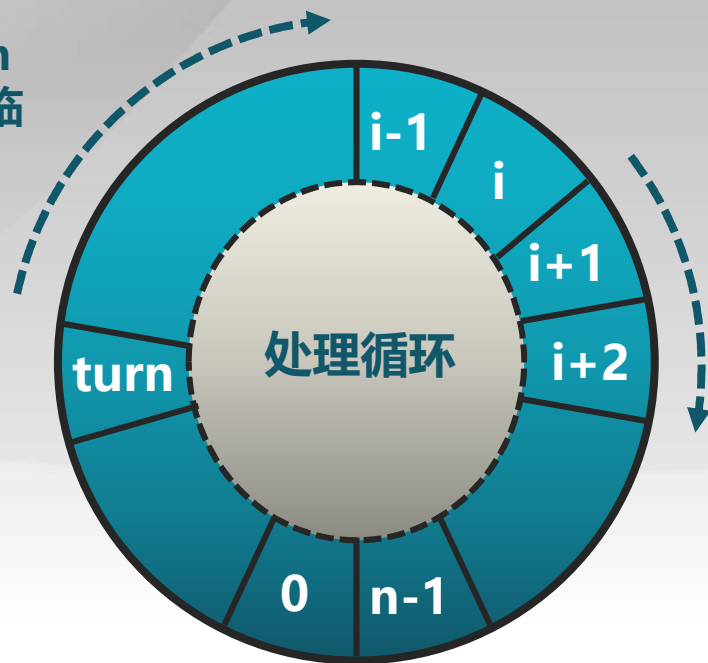
线程Ti 的代码

```
flag[0] := false; flag[1] := false; turn := 0; // or 1

do {
    flag[i] = true;
    while flag[j] == true {
        if turn ≠ i {
            flag[i] := false
            while turn ≠ i { }
            flag[i] := true
        }
    }
    CRITICAL SECTION
    turn := j
    flag[i] = false;
    EMAINDER SECTION
} while (true);
```

N线程的软件方法 (Eisenberg和McGuire)

线程 T_i 要等待从turn到 $i-1$ 的线程都退出临界区后访问临界区



线程 T_i 退出时, 把turn改成下一个请求线程

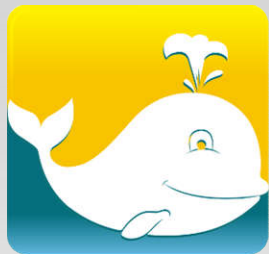
基于软件的解决方法的分析

- 复杂

- ▣ 需要两个进程间的共享数据项

- 需要忙等待

- ▣ 浪费CPU时间



操作系统

Operating Systems



操作系统

Operating Systems

方法3：更高级的抽象方法

- 硬件提供了一些同步原语
 - ▣ 中断禁用，原子操作指令等
- 操作系统提供更高级的编程抽象来简化进程同步
 - ▣ 例如：锁、信号量
 - ▣ 用硬件原语来构建

锁(lock)

- 锁是一个抽象的数据结构
 - ▣ 一个二进制变量（锁定/解锁）
 - ▣ Lock::Acquire()
 - ▣ 锁被释放前一直等待，然后得到锁
 - ▣ 释放锁Release唤醒任何等待的进程
- 使用锁来控制临界区访问

```
lock_next_pid->Acquire();  
new_pid = next_pid++;  
lock_next_pid->Release();
```


原子操作指令

■ 现代CPU体系结构都提供一些特殊的原子操作指令

■ 测试和置位 (Test-and-Set) 指令

▣ 从内存单元中读取值

▣ 测试该值是否为1 (然后返回真或假)

▣ 内存单元值设置为1

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = true;
    return rv;
}
```

原子操作指令

- 现代CPU体系结构都提供一些特殊的原子操作指令
- 测试和置位 (Test-and-Set) 指令
- 交换指令 (exchange)
 - ▣ 交换内存中的两个值

```
void Exchange (boolean *a, boolean *b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

使用TS指令实现自旋锁(spinlock)

```
class Lock {  
    int value = 0;  
}
```

```
Lock::Acquire() {  
    while (test-and-set(value))  
        ; //spin  
}
```

```
Lock::Release() {  
    value = 0;  
}
```

如果锁被释放，那么TS指令读取0并将值设置为1

- ▣ 锁被设置为忙并且需要等待完成

如果锁处于忙状态，那么TS指令读取1并将值设置为1

- ▣ 不改变锁的状态并且需要循环

- ▣ 线程在等待的时候消耗CPU时间

无忙等待锁

忙等待

```
Lock::Acquire() {  
    while (test-and-set(value))  
        ; //spin  
}  
Lock::Release() {  
    value = 0;  
}
```

如何使用交换指令来实现？

无忙等待

```
class Lock {  
    int value = 0;  
    WaitQueue q;  
}  
Lock::Acquire() {  
    while (test-and-set(value)) {  
        add this TCB to wait queue q;  
        schedule();  
    }  
}  
Lock::Release() {  
    value = 0;  
    remove one thread t from q;  
    wakeup(t);  
}
```

原子操作指令锁的特征

■ 优点

- ▣ 适用于单处理器或者共享主存的多处理器中任意数量的进程同步
- ▣ 简单并且容易证明
- ▣ 支持多临界区

■ 缺点

- ▣ 忙等待消耗处理器时间
- ▣ 可能导致饥饿
进程离开临界区时有多个等待进程的情况
- ▣ 死锁
拥有临界区的低优先级进程
请求访问临界区的高优先级进程获得处理器并等待临界区

同步方法总结

- 锁是一种高级的同步抽象方法
 - ▣ 互斥可以使用锁来实现
 - ▣ 需要硬件支持
- 常用的三种同步实现方法
 - ▣ 禁用中断（仅限于单处理器）
 - ▣ 软件方法（复杂）
 - ▣ 原子操作指令（单处理器或多处理器均可）



操作系统

Operating Systems