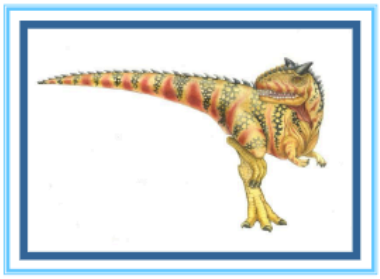
第 6 章：同步工具



第 6 章：同步工具

- 背景 关键部分问题 同步硬件 互斥锁 信号量

目标

- 描述临界区问题并说明争用条件 使用比较和交换运算以及原子变量描述临界区问题的硬件解决方案
- 演示如何使用互斥锁、信号量和条件变量来解决临界区问题

背景

- 进程并发执行
  - 由于各种原因，进程可能随时中断，部分完成执行。
- 对任何共享数据的并发访问都可能导致数据不一致保持数据一致性需要操作系统机制来确保协作进程的有序执行

问题示例

- 考虑 Producer-Consumer 问题：整数计数器用于跟踪占用的缓冲区数量。
  - 最初，counter 设置为 0 每次由 producer 在生成 item 并放置在缓冲区中后递增
  - 每次使用者在消耗缓冲区中的项后，它都会递减。

生产者-消费者问题

制作人

```
while （true） { /* 在下一个生产中生成一个项目 */

        while （counter == BUFFER_SIZE） ;/* 什么都不做 */

        buffer[in] = next_produced;in = （in + 1） %
        BUFFER_SIZE;计数器++;

}
```

消费者

```
while （true） { while （counter == 0）

        ;/* 什么都不做 */ 下次消耗 = buffer[out];输出 =
        （输出 + 1） % BUFFER_SIZE;计数器—;

/* 在下次消耗时消耗项目 */ }
```

争用条件

- counter++ 可以实现为

```
register1 = 计数器 register1 =
register1 + 1 counter = register1
```

- counter-- 可以实现为

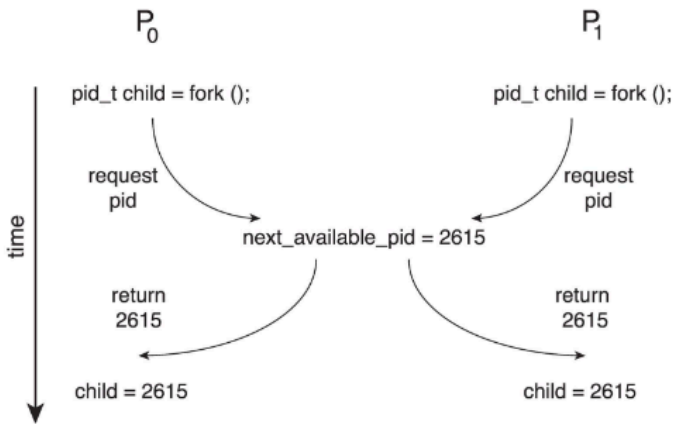
```
register2 = 计数器 register2 =
register2 - 1 计数器 = register2
```

- 最初考虑此执行与 “count = 5” 交错：

S0：生产者执行寄存器 1 = 计数器	{寄存器 1 = 5}
S1：生产者执行 register1 = register1 + 1	{寄存器 1 = 6}
S2：使用者执行 register2 = counter	{寄存器 2 = 5}
S3：使用者执行 register2 = register2 - 1	{寄存器 2 = 4}
S4：生产者执行计数器 = register1	{计数器 = 6 }
S5：使用者执行计数器 = register2	{计数器 = 4}

争用条件

- 进程 P0 和 P1 正在使用 fork（） 系统调用对内核变量 next_available_pid 的争用条件创建子进程，该变量表示下一个可用的进程标识符 （pid）



- 除非存在互斥，否则可以将相同的 pid 分配给两个不同的进程！

关键部分问题

- 争用条件是一种不良情况，其中多个进程同时访问或/和操作共享数据，执行结果取决于访问或执行的特定顺序 - 结果取决于程序的执行时间。如果运气不好（即在执行过程中不合时宜地发生上下文切换），结果将变得不确定
- 考虑一个具有 n 个进程 {P0， P1， ...Pn-1} 进程具有代码的 Critical Section 段（可以是短的或长的），在此期间
-
  - 进程或线程可能正在更改共享变量、更新表、写入文件等。我们需要确保当一个进程位于关键部分时，其他进程都不能位于其关键部分。在某种程度上，互斥和关键部分意味着同样的事情
  -
  -
- 关键部分问题是设计一个协议来解决这个问题
  - 具体来说，每个进程在进入 entry 部分的关键部分之前都必须请求权限，可以在 critical 部分之后使用 exit 部分，然后是 Remainder 部分

关键部分

- 进程 pi 的一般结构为

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

临界截面问题的解决方案

1. 互斥 - 如果进程 Pi 在其关键部分执行，则其他进程都无法执行
   正在执行其关键部分

2. 进度 - 如果其关键部分没有进程正在执行，并且存在一些进程
   希望进入其关键部分的进程，则不能无限期推迟选择接下来将进入关键部分的进程 - 选择一个进入的进程

3. 有限等待 - 其他进程的次数必须存在限制
   允许在进程发出进入其关键部分的请求之后以及在该请求被批准之前进入其关键部分 - 任何等待进程

- 假设每个进程都以非零速度执行，并且没有关于每个单独进程的相对速度的假设

内核中的 Critical-Section 问题

- 内核代码 - 操作系统运行的代码受多种可能的争用条件的影响
  - 维护所有打开文件列表的内核数据结构可以由多个内核进程更新，即两个进程同时打开文件
  - 其他内核数据结构，例如维护内存分配、进程列表、中断处理等的数据结构。
- 两种通用方法用于处理操作系统中的关键部分，具体取决于内核是抢占式还是非抢占式
  - Preemptive – 允许在内核模式下运行时抢占进程，并非不受竞争条件的限制，并且在 SMP 架构中变得越来越困难。
  - Non-preemptive – 一直运行直到退出内核模式、阻止或自愿产生 CPU。这在内核模式下基本上没有竞争条件，可能在单处理器系统中使用

同步工具

- 许多系统为实现关键部分代码提供硬件支持。在单处理器系统上 – 它可以简单地禁用中断，当前正在运行的代码将在不被抢占或中断的情况下执行。但这在多处理器系统上通常效率低下
- 操作系统为关键部分代码提供硬件和高级 API 支持

程序	共享程序
硬件	加载/存储、禁用中断、测试&设置、比较&交换
高级 API	锁， 信号量

同步硬件

- Modern OS 提供特殊的原子硬件指令
  - 原子 = 不可中断 这确保了原子指令的执行不会中断，

  因此，不会发生争用条件 用于更复杂同步机制的构建块
- 有两个常用的原子硬件指令，可用于构建更复杂的同步工具
  - 测试内存字并设置值 – Test_and_Set（） 交换两个内存字的内容 – Compare_and_Swap（）
  -

test_and_set 说明

- 定义：

```
布尔值 test_and_set （boolean *target）
 {
        布尔值 rv = *target;*target = 真;返
        回 RV：


 }
```

使用 test_and_set（） 的解决方案

- 共享布尔变量锁，初始化为 FALSE
- 溶液：

```
{
    while （test_and_set（&lock））
    ;/* 什么都不做 */ /* 关键部分 */
    lock = false;


/* 剩余部分 */ } while （true）；
```

compare_and_swap 说明

- 定义：

```
int compare_and_swap（int *value， int expected， int new_value） {
     int temp = *值;if （*value == 预期）

     *值 = new_value;返回温度;

}
```

使用 compare_and_swap 的解决方案

- 共享布尔变量锁初始化为 FALSE;每个进程都有一个本地布尔变量键

- 溶液：

```
{
     while （比较和交换（&lock， 0， 1） ! = 0）

     ;/* 什么都不做 */ /* 关键部分 */
     lock = 0;


/* 剩余部分 */ } while （true）;
```

有限等待互斥与 test_and_set

```
{
    等待[i] = true;键 = 真;

    while （等待[i] & &键）
    键 = test_and_set（&lock）;等待[i] = false;

    /* 临界截面 */ j = （i + 1） % n;while （（j ! =
    i） & & ! waiting[j]）

    j = （j + 1） % n;如果 （j ==
    i）

    锁 = false;/* 没有人在等待，所以松开锁 */ 否则

    等待[j] = false;/* 解除进程阻塞 j */ /* 剩余部分 */

} while （真）;
```

草图校样

- 互斥：仅当等待[i]==false 或 key==false 时，Pi 才会进入其关键部分。仅当执行 test_and_set（） 时，key 的值才能变为 false。只有第一个执行 test_and_set（） 的进程才会发现 key==false;其他人都必须等待。仅当另一个进程离开其关键部分时，变量 waiting[i] 才能变为 false;只有一个 waiting[i] 设置为 false，从而保持互斥要求。

- 进度：由于进程存在其关键部分，因此要么将 lock 设置为 false，要么将 waiting[j] 设置为 false。两者都允许等待进入其关键部分的进程继续进行。

- 有界等待：当进程离开其关键部分时，它会按循环顺序扫描等待的数组 {i+1， i+2， ...，n-1,0,1,...i-1} 的它将此顺序中位于入口部分 （waiting[j]==true） 中的第一个进程指定为下一个进入关键部分的进程。因此，任何等待进入其关键部分的进程都将在 n-1 回合内完成。

原子变量

- 通常，诸如 compare-and-swap 之类的指令用作其他（更复杂的）同步工具的构建块。
- 一个工具是原子变量，它提供基本数据类型（如整数和布尔值）的原子（不间断）更新。
- 例如，对原子变量 sequence 的 increment（） 操作可确保 sequence 不间断地递增 - increment（&sequence）；
  - increment（） 函数可以按如下方式实现：

```
void increment（atomic_int *v） {

    int temp;

    {
    温度 = *v;}

while （temp ! = （compare_and_swap（v, temp, temp+1））；
}
```

互斥锁

- 操作系统构建了许多软件工具来解决关键部分问题大多数操作系统使用的最简单工具是互斥锁
- 要使用它访问关键区域，首先 acquire（） 一个锁，然后 release（） 它
  - 指示锁是否可用的布尔变量
- 对 acquire（） 和 release（） 的调用必须是原子的（不可中断的）通常通过硬件原子指令实现
- 但这个解决方案需要忙碌的等待。因此，此锁称为自旋锁
  - 由于繁忙等待，自旋锁会浪费 CPU 周期，但它有一个明显的优势，即当进程必须等待锁时不需要上下文切换，并且上下文切换可能需要相当长的时间。因此，当预期短期持有锁时，自旋锁很有用。自旋锁通常用于多处理器系统，其中一个线程可以在一个处理器上“自旋”，而另一个线程在另一个处理器上执行其关键部分

acquire（） 和 release（）

```
acquire（） {
     while （！available）
     ;/* 忙等待 */ 可用 = false;;

} release（） {

可用 = 真;}
```

基于锁的理念保护关键部分的解决方案

- 操作是原子的（不可中断的）——最多一个线程一次获取一个锁在进入访问共享数据的关键部分之前锁定
-
- 访问共享数据后离开关键区域时解锁
- 如果锁定则等待 - 所有同步都涉及忙等待，如果等待很长时间，则应“sleep” 或 “block”

```
{
     获取锁
     关键部分释放锁定

remainder 部分 } while （true）;
```

信号

- 信号量 S – 非负整数变量，可以被认为是 Dijkstra 在 1960 年代后期首次定义的广义锁。它的行为与互斥锁类似，但它具有更复杂的用法 - 原始 UNIX 中使用的主要同步原语
- 两个标准操作修改 S：wait（） 和 signal（）
  - 最初称为 P（） 和 V（），其中 P（） 代表 “proberen” （测试），V（） 在荷兰语中代表 “verhogen” （递增）
- 信号量操作必须以原子方式执行，这保证了没有多个进程可以同时在同一个信号量上执行 wait（） 和 signal（） 操作——序列化
- 信号量只能通过这两个原子操作来访问，但初始化除外

```
等待 （S） {
       while （S <= 0）
       ;忙等待 S--;

} 信号 （S） {
S++;

}
```

信号量使用

- Counting semaphore （计数信号量） – 整数值的范围可以覆盖不受限制的域
  - 计数信号量可用于控制对由有限数量的实例组成的给定资源集的访问;semaphore 值初始化为可用资源的数量
- 二进制信号量 – 整数值的范围只能介于 0 和 1 之间
  - 这可以像互斥锁一样，也可以以不同的方式使用
- 这也可以用来解决各种同步问题 考虑 P1 和 P2 共享一个公共信号量同步，初始化为 0;它确保 P1 进程在 P2 进程执行 S2 之前执行 S1
-

```
P1：
    S1;signal（同步）；

P2：
    wait（同步）；
```

第 2 页;

Semaphore Implementation with no Busy 等待

- 每个信号量都与一个等待队列相关联
- 等待队列中的每个条目都有两个数据项：
  - 值 （整数类型 ） 指向队列上下一条记录的指针
  -
- 两个操作：
  - block – 将调用操作的进程放在相应的等待队列中唤醒 – 删除等待队列中的一个进程，并将其放在就绪队列中
  -
- 信号量值可能会变为负值，而根据忙等待信号量的经典定义，该值永远不能为负值。
- 如果信号量值为负数，则其大小是当前等待信号量的进程数。

无忙等待的信号量实现 （续）

```
typedef struct{
    int 值;结构体进程 *list;
} 信号量;wait（信号量 *S） {
    S->值--;if （S->value < 0） {
        将此进程添加到 S->list;块（）;
} } signal（semaphore *S） {
    S->值++;if （S->值 <= 0） {
        从 S >列表中删除进程 P;觉醒 （P）;
} }
```

注意到这一点

- 递增和递减是在检查信号量值之前完成的，这与 busy 等待实现不同 block（） 操作会暂停调用它的进程。
-
- wakeup（P） 操作恢复暂停进程 P 的执行。

死锁和饥饿

- Deadlock – 两个或多个进程无限期地等待只能由其中一个等待进程引起的事件（将在第 8 章中检查）
- 设 S 和 Q 是初始化为 1 的两个信号量

P0 系列	P1
wait（S）;	等待 （Q）;
等待 （Q）;	wait（S）;
…	…
信号 （S）;	信号 （Q）;
信号 （Q）;	信号 （S）;

- 考虑 P0 是否执行 wait（S） 和 P1 执行 wait（Q）。当 P0 执行 wait（Q） 时，它必须等待 P1 执行 signal（Q），但是，P1 正在等待，直到 P0 执行 signal（S）。由于这些 signal（） 操作永远不会被执行，因此 P0 和 P1 被死锁。这非常难以调试
- 饥饿 – 无限期阻塞 进程永远不能从信号量队列中删除，在该队列中，进程被挂起。例如，如果我们使用 LIFO（后进先出）顺序或基于某些优先级从与信号量关联的队列中删除进程。

第 6 章结束