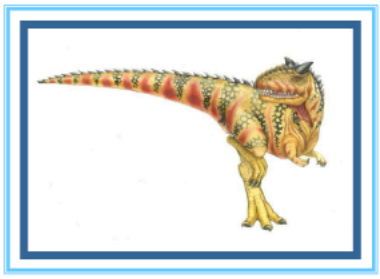
第 7 章：同步
例



同步示例

- 同步的典型问题
  - 有界缓冲区问题 读取器和写入器问题
  -
- 窗口同步 POSIX 同步
-

有界缓冲区问题

- n 个缓冲区，每个缓冲区可以容纳一个项目
- Semaphore mutex 初始化为值 1
- 信号量 full 初始化为值 0
- Semaphore empty 初始化为值 n

有界缓冲区问题（续）

- 生产者流程的结构

```
{
        ... /* 在 next_produced 中生产一个项目 */ ...

    等待（空）;wait
      （互斥锁）;

        ... /* 将 next generated 添加到缓冲区 */ ...

    signal（互斥锁）;signal
      （full） 的

} while （真）;
```

有界缓冲区问题（续）

- 消费者流程的结构

```
{
    wait（full）;wait
    （互斥锁）；

        ... /* 将项目从缓冲区中删除到next_consumed */ ...

    signal（互斥
    锁）;signal（空）；

        ... /* 消耗下个消耗的项目 */ ...

    } while （真）；
```

读写器问题

- 一个数据集在多个并发进程之间共享 读取者 – 只读取数据;它们不执行任何更新
  Writers – 可以读取和写入
- 问题 – 允许多个读取器同时读取数据集，但一次最多只有一个写入器可以访问共享数据
- 对待读者和作者的方式有几种变化 – 涉及不同的优先事项。最简单的解决方案称为第一个读取器-写入器问题，它要求除非写入器已经获得了对共享数据的访问权限，否则没有读取器保持等待
-
  - 共享数据更新（由写入者）可以延迟这使读取器优先访问共享数据
  -
- 共享数据
  - 数据集 信号量 rw_mutex 初始化为 1 信号量互斥量初始化为 1 整数 read_count 初始化为 0
  -
  -
  -

Readers-Writers Problem （续）

- 编写器进程的结构

```
{
    等待 （rw_mutex）；
        执行写入 */ ...

信号（rw_mutex）；} while
（真）；
```

Readers-Writers Problem （续）

读取器进程的结构 do {

```
    wait（互斥
锁）;read_count++;if
  （read_count == 1） wait
  （rw_mutex）；

    信号（互斥锁） ...

    执行 /* 读取 */ ...

    wait（互斥锁）;read_count--;
    如果 （read_count == 0）

    信号（rw_mutex）;signal（互
    斥锁）；
} while （真）；
```

注意：

- rw_mutex 控制写入器和第一个读取者对共享数据（关键部分）的访问。最后一个离开关键部分的读取器也必须释放此锁
- mutex 控制读取器对共享变量计数的访问
- 作家等待 rw_mutex，第一个读者但获得对关键部分的访问权限也等待 rw_mutex。所有后续读取器仍可访问 wait 互斥锁

Readers-Writers 问题变化

- 第一种变体 – 除非写入器已获得使用共享对象的访问权限，否则没有读取器一直等待。这很简单，但可能会导致编写器资源不足，因此可能会显著延迟对象的更新。
- 第二种变体 – 一旦 writer 准备就绪，它就需要尽快执行更新。换句话说，如果写入器等待访问对象（这意味着内部可能有读取器或写入器），则没有新的读取器可以开始读取，即，它们必须在写入器更新对象后等待
- 任何一个问题的解决方案都可能导致饥饿
- 该问题可以通过内核提供读写器锁来解决或至少部分解决，其中允许多个进程在读取模式下同时获取读写器锁，但只有一个进程可以获取读写器锁以进行写入（独占访问）。因此，获取读写器锁需要指定锁的模式：读或写访问

同步示例

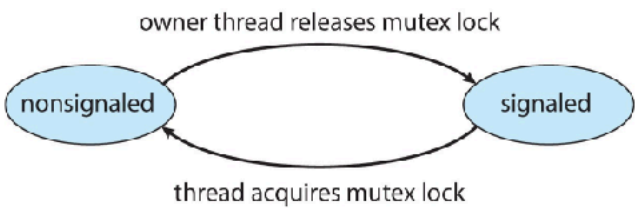
- 索拉里斯
- Windows XP
- Linux的
- Pthread 线程

Solaris 同步

- 实现各种锁以支持多任务、多线程（包括实时线程）和多处理
- 在保护数据免受短代码段（通常少于几百条（机器级）指令）的影响时，使用自适应互斥锁来提高效率
  - 从在多处理器系统中作为旋转锁实现的标准信号量开始，如果持有锁，并且由另一个 CPU 上运行的线程自旋以等待锁变为可用
  - 如果 lock 由非运行状态线程持有，则阻塞并休眠等待释放 lock 的信号
- 使用条件变量
- 当较长的代码段需要访问数据时使用读取器-写入器锁。这些用于保护经常访问但通常以只读方式访问的数据。读取器-写入器锁的实现成本相对较高。

Windows 同步

- 内核使用中断掩码来保护对单处理器系统中全局资源的访问
- 内核在多处理器系统中使用自旋锁（以保护短代码段）
  - 为了提高效率，内核确保在持有自旋锁时永远不会抢占线程
- 对于内核外部的线程同步（用户模式），Windows 提供了调度程序对象，线程根据几种不同的机制进行同步，包括互斥锁、信号量、事件和计时器
  - 事件类似于条件变量;当所需条件出现时，它们可能会通知等待线程
  - 计时器用于通知一个或多个线程指定的时间已过期 Dispatcher 对象为信号状态（对象可用）或非信号状态（这意味着另一个线程正在保存该对象，因此线程将阻塞）



Linux 同步

- Linux的：
  - 在内核版本 2.6 之前，禁用中断以实现简短的关键部分版本 2.6 及更高版本，完全抢占式内核
  -
- Linux 提供：
  - semaphores Spinlocks – 用于多处理器系统原子整数，以及使用原子整数的所有数学运算，不会中断执行
  -
  -
  - 读写器锁
- 在单 CPU 系统上，自旋锁被启用和禁用内核抢占所取代

原子变量

- 原子变量 - atomic_t 是原子整数的类型 考虑 counter atomic_t变量;
-

```
int 值;
```

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

POSIX 同步

- POSIX API 提供
  - 互斥锁 信号量 条件变量
  -
  -
- 广泛用于 UNIX、Linux 和 MacOS

POSIX 互斥锁

- 创建和初始化锁

```
#include <pthread.h>

pthread_mutex_t mutex;

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

- 获取和释放锁

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

/* critical section */

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

POSIX 条件变量

- POSIX 条件变量与 POSIX 互斥锁相关联，以提供互斥：创建和初始化条件变量：

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;

pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&cond_var,NULL);
```

POSIX 条件变量

- 线程等待条件 a == b 变为 true：

```
pthread_mutex_lock(&mutex);
while (a != b)
     pthread_cond_wait(&cond_var, &mutex);

pthread_mutex_unlock(&mutex);
```

- pthread_cond_wait（） &mutex 作为第二个参数 - 除了将调用线程置于睡眠状态外，还会在将所述调用方置于睡眠状态时释放锁。否则，其他线程无法获取锁并发出唤醒信号

POSIX 条件变量

- 线程向另一个线程发出等待条件变量的信号：

```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

- 当发出信号时（以及修改 condition 变量时），请确保持有 lock。这可确保不会意外引入争用条件

- 在唤醒后返回之前，pthread cond wait（） 会重新获取锁，从而确保在等待序列开始时的锁获取和结束时的锁释放之间，每当等待线程运行时，它都会保持锁。

第 7 章结束