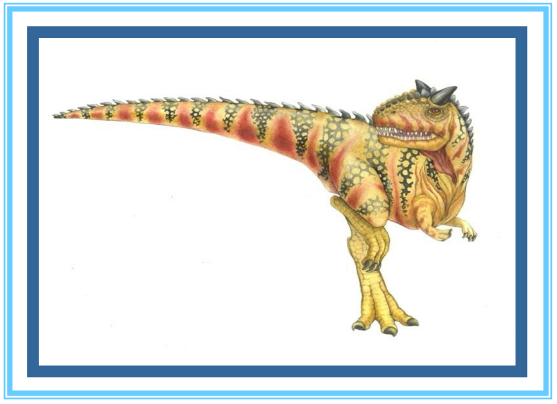
Chapter 7: Synchronization Example



Synchronization Examples

- Classic Problems of Synchronization
  - Bounded-Buffer Problem 有界缓冲区问题
  - Readers and Writers Problem

- Window Synchronization
- POSIX Synchronization

Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- 信号量
  Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value n

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {
       ...
       /* produce an item in next_produced */
       ...
    wait(empty);
    wait(mutex);
       ...
       /* add next produced to the buffer */
       ...
    signal(mutex);
    signal(full);
} while (true);
```

这段代码是一个典型的生产者-消费者模型中的生产者部分，使用信号量来管理对共享缓冲区的访问。以下是对代码的逐行解释：

1. **do { ... } while (true);**: 这是一个无限循环，表示生产者将持续不断地生产项目。
2. */ produce an item in next_produced */**: 这一行是注释，表示生产者在这里生成一个新的项目，并将其存储在变量 `next_produced` 中。
3. **wait(empty);**: 这是一个信号量操作，表示生产者在尝试添加新项目之前，首先检查缓冲区是否有空位。`empty` 信号量表示缓冲区中空位的数量。如果 `empty` 的值为0，生产者将被阻塞，直到有空位可用。
4. **wait(mutex);**: 另一个信号量操作，表示生产者请求对缓冲区的互斥访问。`mutex` 信号量确保在同一时刻只有一个进程可以访问缓冲区，以避免数据竞争。
5. */ add next produced to the buffer */**: 这一行是注释，表示生产者将 `next_produced` 中的项目添加到缓冲区中。
6. **signal(mutex);**: 释放互斥锁，允许其他进程（如消费者）访问缓冲区。
7. **signal(full);**: 这是一个信号量操作，表示生产者在成功添加新项目后，通知消费者缓冲区中有新项目可用。`full` 信号量表示缓冲区中已填充项目的数量。

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
do {
    wait(full);
    wait(mutex);
        ...
        /* remove an item from buffer to next_consumed */
        ...
    signal(mutex);
    signal(empty);
        ...
        /* consume the item in next consumed */
        ...
    } while (true);
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - **Readers** – only read the data; they **do not** perform any updates
  - **Writers** – can both read and write
- Problem – allow multiple readers to read the data set at the same time, but at most only one single writer can access shared data at a time
- Several variations of how readers and writers are treated – involve different priorities.
- The simplest solution, referred to as the first readers-writers problem, requires that no reader be kept waiting unless a writer has already gained access to the shared data
  - Shared data update (by writers) can be delayed
  - This gives readers priority in accessing shared data
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {
    wait(rw_mutex);
        ...
        /* writing is performed */
        ...
    signal(rw_mutex);
} while (true);
```

1. **do { … } while (true);**: 这是一个无限循环，表示写者将持续不断地进行写操作。
2. **wait(rw_mutex);**: 这是一个信号量操作，表示写者请求对共享资源的互斥访问。`rw_mutex` 是一个互斥锁，确保在同一时刻只有一个写者可以访问共享资源，以避免数据竞争。
3. */ writing is performed */**: 这一行是注释，表示在这里进行实际的写操作。写者将对共享资源进行修改。
4. **signal(rw_mutex);**: 释放互斥锁，允许其他进程（如其他写者或读者）访问共享资源。

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
       wait(rw_mutex);
    signal(mutex)
    ...
    /* reading is performed */
    ...
    wait(mutex);
    read_count--;
    if (read_count == 0)
       signal(rw_mutex);
    signal(mutex);
} while (true);
```

Note:

- `rw_mutex` controls the access to shared data (critical section) for writers, and the first reader. The last reader leaving the critical section also has to release this lock
- `mutex` controls the access of readers to the shared variable `count`
- Writers wait on `rw_mutex`, first reader yet gain access to the critical section also waits on `rw_mutex`. All subsequent readers yet gain access wait on `mutex`

1. **do { … } while (true);**
   这是一个无限循环，表示读者将不断尝试读取共享资源。

2. **wait(mutex);**
   通过调用 `wait(mutex)`，读者请求对 `mutex` 的访问，以确保对 `read_count` 的操作是互斥的。

3. **read_count++;**
   读者数量增加，表示有一个新的读者开始读取。

4. **if (read_count == 1) wait(rw_mutex);**
   如果这是第一个读者（即 `read_count` 变为 1），则请求对 `rw_mutex` 的访问。这是为了确保在有读者时，写者不能访问共享资源。

5. **signal(mutex);**
   释放 `mutex`，允许其他读者或写者访问。

6. */ reading is performed /*
   在这里，读者执行实际的读取操作。

7. **wait(mutex);**
   读者完成读取后，再次请求对 `mutex` 的访问，以更新 `read_count`。

8. **read_count--;**
   读者数量减少，表示有一个读者结束了读取。

9. **if (read_count == 0) signal(rw_mutex);**
   如果这是最后一个读者（即 `read_count` 变为 0），则释放 `rw_mutex`，允许写者访问共享资源。

10. **signal(mutex);**
    释放 `mutex`，允许其他读者或写者继续访问。

Readers-Writers Problem Variations

- **First variation** – no reader kept waiting unless a writer has gained access to use shared object. This is simple, but can result in starvation for writers, thus can potentially significantly delay the update of the object.

尽快执行更新

- **Second variation** – once a writer is ready, it needs to perform update asap. In another word, if a writer waits to access the object (this implies that there could be either readers or a writer inside), no new readers may start reading, i.e., they must wait after the writer updates the object

- A solution to either problem may result in starvation

解决方法

- The problem can be solved or at least partially by the kernel providing **reader-writer locks**, in which multiple processes are permitted to concurrently acquire a reader-writer lock in read mode, but only one process can acquire the reader-writer lock for writing (exclusive access). Acquiring a reader–writer lock thus requires specifying the mode of the lock: either read or write access

Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing

自适应互斥锁

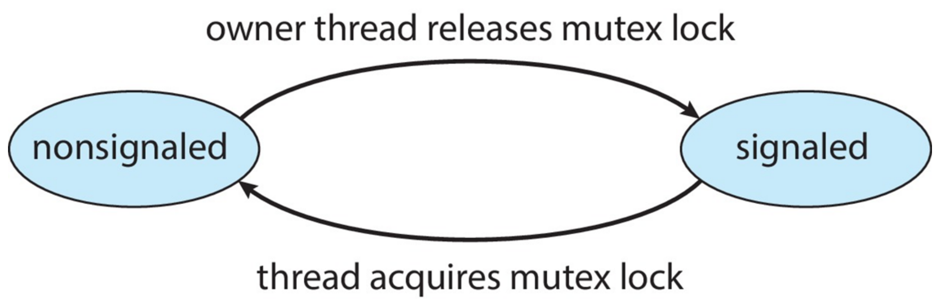
- Uses adaptive mutex for efficiency when protecting data from *short code segments*, usually less than a few hundred (machine-level) instructions
  - Starts as a standard semaphore implemented 作为 as a **spinlock** 旋转锁 in a multiprocessor system
  - If lock held, and by a thread running on another CPU, spins to wait for the lock to become available
  - If lock held by a non-run-state thread, block and sleep waiting for signal of lock being released

- Uses condition variables

- Uses readers-writers locks when longer sections of code need access to data. These are used to protect data that are frequently accessed, but usually in a read-only manner. The readers-writer locks are relatively expensive to implement.

Windows Synchronization

- The kernel uses 中断 掩码 interrupt masks to protect access to global resources in uniprocessor systems
- The kernel uses 自旋锁 spinlocks in multiprocessor systems (to protect short code segments)
  - For efficiency, the kernel ensures that a thread 不会被抢占 will never be preempted while holding a spinlock
- For thread synchronization outside the kernel (user mode), Windows provides dispatcher objects 调度程序对象, threads synchronize according to several different mechanisms 机制, including mutex locks, semaphores, events, and timers
  - Events are similar to condition variables; they may notify a waiting thread when a desired condition occurs
  - Timers are used to notify one or more thread that a specified amount of time has expired
  - Dispatcher objects either signaled-state (object available) or non-signaled state (this means that another thread is holding the object, therefore the thread will block)



Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive kernel 完全抢占式内核
- Linux provides:
  - semaphores 信号量
  - 自旋锁
    Spinlocks – for multiprocessor systems
  - atomic integer, and all math operations using atomic integers performed without interruption
  - reader-writer locks
- On single-CPU system, spinlocks replaced by enabling and disabling kernel preemption

Atomic Variables 原子变量.

- Atomic variables - `atomic_t` is the type for atomic integer

确保在不被中断的情况下执行数学操作

- Consider the variables

```
atomic_t counter;
int value;
```

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 Synchronization

- POSIX API provides
  - mutex locks
  - semaphores
  - condition variables
- Widely used on UNIX, Linux, and MacOS

POSIX Mutex Locks

- Creating and initializing the lock

```
#include <pthread.h>

pthread_mutex_t mutex;

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

- Acquiring and releasing the lock

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

/* critical section */

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

POSIX Condition Variables

- POSIX condition variables are associated with a POSIX mutex lock to provide mutual exclusion: Creating and initializing the condition variable:

互斥

creating

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;
```

initializing

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

POSIX Condition Variables

- Thread waiting for the condition **`a == b`** to become true:

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

pthread_mutex_unlock(&mutex);
```

- **`pthread_cond_wait()`** **`&mutex`** as the second parameter - in addition to putting the calling thread to sleep, releases the lock when putting said caller to sleep. If not, no other thread can acquire the lock and signal it to wake up

POSIX Condition Variables

- Thread signaling another thread waiting on the condition variable:

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

- When signaling (as well as when modifying the condition variable), make sure to have the lock held. This ensures that no race condition is accidentally introduced

- Before returning after being waked up, the **`pthread_cond_wait()`** re-acquires the lock, thus ensuring that any time the waiting thread is running between the lock acquire at the beginning of the wait sequence, and the lock release at the end, it holds the lock.

这段话的意思是，在使用 `pthread_cond_wait()` 函数时，当一个线程被唤醒并准备继续执行之前，它会重新获取锁。这确保了在整个等待过程中，从线程在等待开始时获取锁到等待结束时释放锁的这段时间内，线程始终持有该锁。

简单来说，这个机制保证了线程在等待条件变量时，能够安全地访问共享资源，避免了数据竞争和不一致的问题。

End of Chapter 7