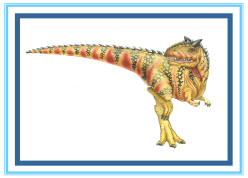
Chapter 7: Synchronization Example



Synchronization Examples

- Classic Problems of Synchronization
  - Bounded-Buffer Problem
  - Readers and Writers Problem
- Window Synchronization
- POSIX Synchronization

Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- 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);
```

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 set; 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
  - Only one single writer can access shared data at a time
- Several variations of how readers and writers are treated – all 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
- 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);
```


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`

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.
- **Second variation** – once a writer is ready, it needs to perform update asap. In another word, if a writer starts waiting to access the object (this implies that there are readers reading at the moment), no new readers may start reading, i.e., they must wait after the writer updates the object. Of course, those reader(s) inside will finish reading the object while the writer waits outside
- A solution to either problem may result in starvation
- The problem can be solved or at least partially by 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

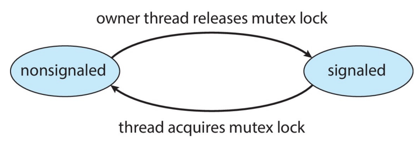
- Solaris
- Windows XP
- Linux
- Pthreads

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses adaptive mutex for efficiency when protecting data from *short code segments*, usually less than a few hundred instructions
  - Starts as a standard semaphore implemented as a spinlock 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 interrupt masks to protect access to global resources in uniprocessor systems
- The kernel uses spinlocks in multiprocessor systems (to protect short code segments)
  - For efficiency, the kernel ensures that a thread will never be preempted while holding a spinlock
- For thread synchronization outside the kernel (user mode), Windows provides dispatcher objects, threads synchronize according to several different mechanisms, 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
- Linux provides:
  - semaphores
  - 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

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

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;

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);
```

- Thread signaling another thread waiting on the condition variable:

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


End of Chapter 7