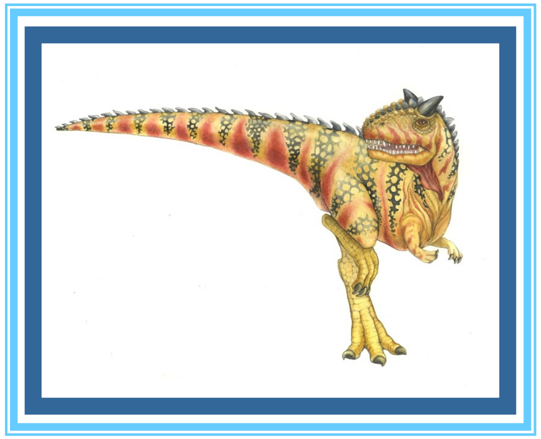
Chapter 7: Synchronization Examples



Classical Problems of Synchronization

- We've looked at a simple example for using synchronization
- Now we're going to use semaphores to look at more interesting examples
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

Problem: There is a set of resource buffers shared by producer and consumer threads

Producer inserts resources into the buffer set

 Output, disk blocks, memory pages, processes, etc.

Consumer removes resources from the buffer set

 Whatever is generated by the producer

Producer and consumer execute at different rates

 No serialization of one behind the other

 Tasks are independent

 The buffer set allows each to run without explicit handoff

- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** *(binary semaphore*): mutex to shared set of buffers
- Semaphore **`full`***(counting semaphore*): count of full buffers
- Semaphore **`empty`***(counting semaphore*): count of empty buffers

Bounded Buffer Problem (Cont.)

- The structure of the producer process

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

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
Do {
    wait(full);   // wait for a full buffer
    wait(mutex);  // lock buffer list
       ...
    /* remove an item from buffer to next_consumed */
       ...
    signal(mutex); // unlock buffer list
    signal(empty); // notify an empty buffer
       ...
    /* 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 – An object is shared among several threads. Some threads only read the object, others only write it
  - We can allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** : exclusive writing or reading
  - Semaphore **`mutex`** : control access to readcount
  - Integer **`read_count`** : number of threads reading object

Readers-Writers Problem (Cont.)

- The structure of a writer process

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

Readers-Writers Problem (Cont.)

- The structure of a reader process

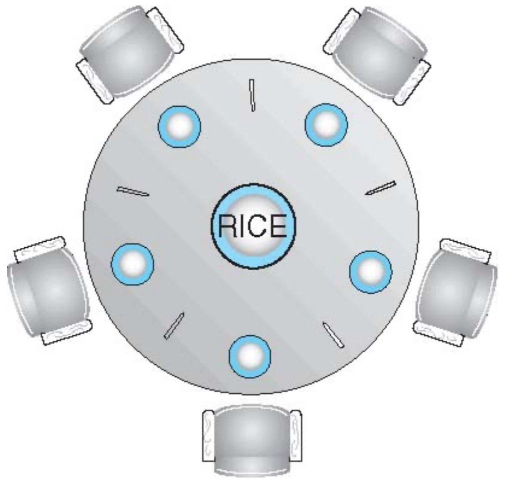
```
do {
        wait(mutex);                    // lock readcount
        read_count++;                   // add one more reader
        if (read_count == 1)
        wait(rw_mutex);                 // synch w/ writers
    signal(mutex);                      // unlock readcount
          ...
        /* reading is performed */
          ...
    wait(mutex);                        // lock readcount
        read count--;                   // one less reader
        if (read_count == 0)
    signal(rw_mutex);                   // resume
    signal(mutex);                      // unlock readcount
} while (true);
```

Readers-Writers Problem Variations

Several variations of how readers and writers are considered – all involve some form of priorities

- ***First*** variation – no reader kept waiting unless writer has permission to use shared object
- ***Second*** variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

Dining-Philosophers Problem



- Philosophers spend their lives alternating thinking and eating
- Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done

- Imagine that we have 5 philosophers who need to eat lunch. Additionally, we have one table that can simultaneously accommodate no more than two people.
- Our task is to feed all the philosophers. None of them should go hungry, and none of them should "block" each other when trying to sit down at the table (we must avoid deadlock).

Implementation

```java
public class Main {

    public static void main(String[] args) {

        Semaphore sem = new Semaphore(2);
        new Philosopher(sem, "Socrates").start();
        new Philosopher(sem,"Plato").start();
        new Philosopher(sem,"Aristotle").start();
        new Philosopher(sem, "Thales").start();
        new Philosopher(sem, "Pythagoras").start();
    }
}
```

```java
class Philosopher extends Thread {

    private Semaphore sem;

    // Did the philosopher eat?
    private boolean full = false;

    private String name;

    Philosopher(Semaphore sem, String name)
        this.sem=sem;
        this.name=name;
    }

```

Output:

```
Socrates takes a seat at the table
Plato takes a seat at the table
Socrates has eaten! He leaves the table
Plato has eaten! He leaves the table
Aristotle takes a seat at the table
Pythagoras takes a seat at the table
Aristotle has eaten! He leaves the table
Pythagoras has eaten! He leaves the table
Thales takes a seat at the table
Thales has eaten! He leaves the table
```

We created a semaphore whose counter is set to 2 to satisfy the condition: only two philosophers can eat at the same time. That is, only two threads can run at the same time.

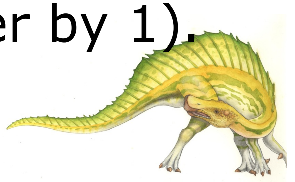
Implementation (con't)

```java
    public void run()
    {
        try
        {
            // If the philosopher has not eaten
            if (!full) {
                // Ask the semaphore for permission to run
                sem.acquire();
                System.out.println(name + " takes a seat at the table");

                // The philosopher eats
                sleep(300);
                full = true;

                System.out.println(name + " has eaten! He leaves the table");
                sem.release();

                // The philosopher leaves, making room for others
                sleep(300);
            }
        }
        catch(InterruptedException e) {
            System.out.println("Something went wrong!");
        }
    }
}
```

The **acquire()** and **release()** methods of the Semaphore class control its access counter.

The acquire() method asks the semaphore for access to the resource. If the counter is >0, then access is granted and the counter is reduced by 1.

The release() method "releases" the previously granted access, returning it to the counter (increases the semaphore's access counter by 1).



Synchronization Examples

- Solaris
- Windows
- Linux
- Pthreads

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses **adaptive mutexes** for efficiency when protecting data from short code segments
  - Starts as a standard semaphore spin-lock
  - If lock held, and by a thread running on another CPU, spins
  - If lock held by 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
- Uses **turnstiles** to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
  - Turnstiles are per-lock-holding-thread, not per-object
- Priority-inheritance per-turnstile gives the running thread the highest of the priorities of the threads in its turnstile

Windows Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
  - Spinlocking-thread will never be preempted
- Also provides **dispatcher objects** user-land which may act mutexes, semaphores, events, and timers
  - **Events**
    - An event acts much like a condition variable
  - Timers notify one or more thread when time expired
  - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (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
- Linux provides:
  - Semaphores
  - atomic integers
  - spinlocks
  - reader-writer versions of both
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variable
- Non-portable extensions include:
  - read-write locks
  - spinlocks

Alternative Approaches

- Transactional Memory
- OpenMP
- Functional Programming Languages

Transactional Memory

- A **memory transaction** is a sequence of read-write operations to memory that are performed atomically.
- If all operations in a transaction are completed, the memory transaction is committed. Otherwise, the operations must be aborted and rolled back.

```
void update()
 {
        /* read/write memory */
  }
```

An advantage of using transactional memory is that
- The atomicity is guaranteed by the transactional memory system.
- There is no possibility of deadlocks.
- The transactional memory system can identify which statements in atomic blocks can be executed concurrently.

OpenMP

- OpenMP is a set of compiler directives and API that support parallel progamming.
- The advantage of OpenMP (and similar tools) is that thread creation and management are handled by the OpenMP library and are not the responsibility of application developers.

```
void update(int value)
 {
       #pragma omp critical
       {
              count += value
       }
  }
```

The code contained within the **`#pragma omp critical`** directive is treated as a critical section and performed atomically.

However, a disadvantage is that application developers must still identify possible race conditions and adequately protect shared data using the compiler directive. Deadlock is still possible when two or more critical sections are identified.

Functional Programming Languages

- Functional programming languages offer a different paradigm than procedural languages in that they do not maintain state.
- Variables are treated as immutable and cannot change state once they have been assigned a value.
- There is increasing interest in functional languages such as Erlang and Scala for their approach in handling data races.
- The Erlang language has gained significant attention because of its support for concurrency and the ease with which it can be used to develop applications that run on parallel systems.
- Scala is a functional language that is also object-oriented. In fact, much of the syntax of Scala is similar to the popular object-oriented languages Java and C#.

End of Chapter 7