Chapter 6: Process Synchronization

Module 6: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Consumer-producer problem
  - Having an integer count that keeps track of the number of full buffers.
  - Initially, count is set to 0.
  - It is incremented by the producer after it produces a new buffer
  - and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {

        /*  produce an item and put in nextProduced  */
        while (count == BUFFER_SIZE)
                ; // do nothing
         buffer[in] = nextProduced;
         in = (in + 1) % BUFFER_SIZE;
         count++;
}
```

Consumer

```
while (true)  {
        while (count == 0)
              ; // do nothing
        nextConsumed =  buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;

      /*  consume the item in nextConsumed */
}
```

Race Condition

- **count++** could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

- **count--** could be implemented as

```
register2 = count
register2 = register2 - 1
count = register2
```

Race Condition

- This execution interleaving with "count = 5" initially:
  - S0: producer execute register1 = count {register1 = 5}
  - S1: producer execute register1 = register1 + 1 {register1 = 6}
  - S2: consumer execute register2 = count {register2 = 5}
  - S3: consumer execute register2 = register2 - 1 {register2 = 4}
  - S4: producer execute count = register1 {count = 6 }
  - S5: consumer execute count = register2 {count = 4}

Critical Section

- A segment of code in which the process may be changing shared resources.
  - Common variables, a table, a file...
- IMPORTANT:
  - No two processes are executing in their critical sections at the same time

Solution to Critical-Section Problem must Satisfy

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
  - int turn;
  - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready!

Algorithm for Process P_i

```
while (true) {
     flag[i] = TRUE;
     turn = j;
     while ( flag[j] && turn == j)
          ;

          CRITICAL SECTION

     flag[i] = FALSE;

          REMAINDER SECTION

}
```

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

TestAndndSet Instruction

- **Definition:**

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv:
}
```

Solution using TestAndSet

- Shared boolean variable lock., initialized to false.
- Solution:

```
while (true) {
        while ( TestAndSet(&lock) )
                ;    /* do nothing

        //    critical section

        lock = FALSE;

        //        remainder section
}
```

Semaphore

- Semaphore *S* – integer variable
- Two standard operations modify S:
  - wait() and signal()
  - Originally called P() and V()
- Less complicated

Semaphore

- Can only be accessed via two indivisible (atomic) operations
  - wait (S) {
    while S <= 0
    ; // no-op
    S--;
    }
  - signal (S) {
    S++;
    }

Semaphore as General Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion
  - Semaphore mutex=1; // initialized to 1
  - wait (mutex);

    Critical Section

    signal (mutex);

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
  - Disable/Enable Interrupts
  - TestAndSet()
- Busy waiting (spinlock)
  - Little busy waiting if critical section rarely occupied
  - Applications may spend lots of time in critical sections and therefore this is not a good solution.

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue.
  - ```
    struct semaphore {
        int value;
        struct process * waiting_list;
    }
    ```
- Two operations:
  - block – place the process invoking the operation on the appropriate waiting queue.
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue.


    ```

Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of wait:

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

- Implementation of signal:

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

Classical Problems of Synchronization

- Bounded-Buffer Problem (Producer-Consumer Problem)
- Readers and Writers Problem
- Dining-Philosophers Problem

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

```
while (true)  {
     //  produce an item
     wait (empty);
     wait (mutex);
     //  add the item to the  buffer
     signal (mutex);
     signal (full);
}
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
while (true) {
     wait (full);
     wait (mutex);
     //  remove an item from  buffer
     signal (mutex);
     signal (empty);
     //  consume the removed item
}
```

Readers-Writers Problem

- A data set is shared among a number of processes
  - Readers – only read the data set
  - Writers – can both read and write
- Problem
  - 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 mutex initialized to 1.
  - Semaphore wrt initialized to 1.
  - Integer readcount initialized to 0.

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
while (true) {
        wait (wrt) ;

        //    writing is performed

        signal (wrt) ;
}
```

Readers-Writers Problem (Cont.)

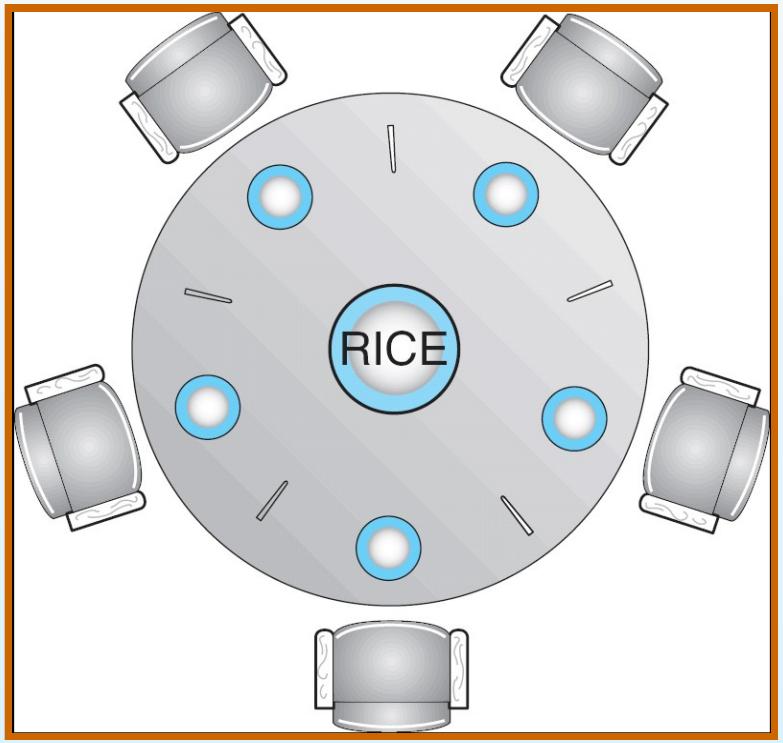
- The structure of a reader process

```
while (true) {
     wait (mutex) ;
     readcount++ ;
     if (readcount == 1)  wait(wrt) ;
     signal (mutex)

     // reading is performed

     wait (mutex) ;
     readcount-- ;
     if (readcount == 0)  signal(wrt) ;
     signal (mutex) ;
}
```

Dining-Philosophers Problem



- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem (Cont.)

- The structure of Philosopher *i*:

```
while (true)  {
   wait( chopstick[i] );
   wait( chopStick[ (i + 1) % 5] );

   //  eat

   signal( chopstick[i] );
   signal(chopstick[ (i + 1) % 5] );

   //  think
}
```

Dining-Philosophers Problem (Cont.)

- Some solutions
  - Allow at most four philosophers to be sitting simultaneously at the table
  - Pick up chopsticks only if both chopsticks are available
  - An odd philosopher picks up left chopstick first;
    An even philosopher picks up right chopstick first.

Monitors

- A high-level abstraction. More convenient and effective.
  - Java, C#
- Only one process may be active within the monitor at a time

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (…) {…}
    …
    procedure Pn (…) {…}
    Initialization code (…) {…}
    …
}
```

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 mutexes for efficiency when protecting data from short code segments
  - Spin or no-spin adaptively
- Uses condition variables, semaphore and 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

Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses spinlocks on multiprocessor systems
- Also provides dispatcher objects
  - Mutexes
  - Semaphores
  - Events
  - Timers

Linux Synchronization

- Uniprocessor
  - Disable/enable preemption
- Multiprocessor
  - spinlocks

- Linux also provides:
  - semaphores

Pthreads Synchronization

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

End of Chapter 6