Chapter 6: Process Synchronization

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
- Less complicated
- 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 S; // initialized to 1
  - wait (S);
    Critical Section
    signal (S);

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the crtical section.
  - Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that 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. Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the 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 (S){
     value--;
     if (value < 0) {
          add this process to waiting queue
          block(); }
}
```

- Implementation of signal:

```
Signal (S){
     value++;
     if (value <= 0) {
          remove a process P from the waiting queue
          wakeup(P); }
}
```

Deadlock and Starvation

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem V1

- Revisit the problem
- Version1: using "count" for syncronization…

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
 }
```

Problem for version 1.0 ?

Bounded-Buffer Problem V2

- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N.

Bounded Buffer Problem V2

- The structure of the producer process

```
do {
     //  produce an item

   wait (empty);

     //  add the item to the  buffer

   signal (full);
 } while (true);
```

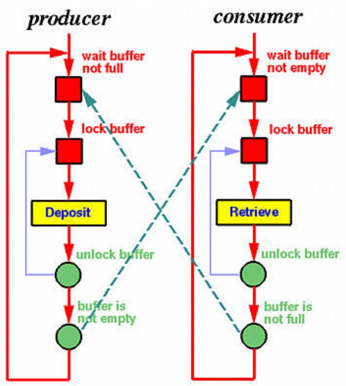
Bounded Buffer Problem V2

- The structure of the consumer process

```
do {
   wait (full);

     //  remove an item from  buffer

   signal (empty);

     //  consume the removed item

} while (true);
```

Problem of V2?

- one producer and one consumer
- Multiple producers and consumers



Bounded-Buffer Problem V3

- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N.

Bounded Buffer Problem (V3)

- The structure of the producer process

```
do {
     //  produce an item

   wait (empty);
   wait (mutex);

     //  add the item to the  buffer

   signal (mutex);
   signal (full);
 } while (true);
```

Bounded Buffer Problem (V3)

- The structure of the consumer process

```
do {
   wait (full);
   wait (mutex);

      // remove an item from buffer

    signal (mutex);
    signal (empty);

       // consume the removed item

} 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 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

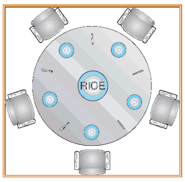
```
do {
     wait (wrt) ;

      //   writing is performed

     signal (wrt) ;
  } while (true)
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
     wait (mutex) ;
     readcount ++ ;
     if (readercount == 1)  wait (wrt) ;
     signal (mutex)

         // reading is performed

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

Dining-Philosophers Problem



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

Dining-Philosophers Problem

- The structure of Philosopher *i*:

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

          // eat

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

         // think

} while (true) ;
```

Problems with Semaphores

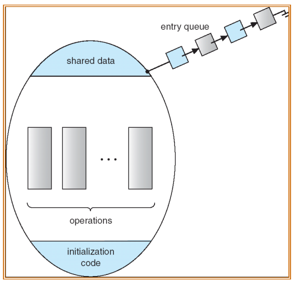
- Correct use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

```
monitor monitor-name
{
   // shared variable declarations
   procedure P1 (…) { …. }
         …

   procedure Pn (…) {……}

    Initialization code ( ….) { … }
         …
   }
}
```

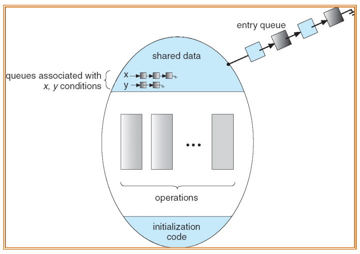
Schematic view of a Monitor



Condition Variables

- condition x, y;
- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended.
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()

Monitor with Condition Variables



Solution to Dining Philosophers

```
monitor DP
 {
    enum { THINKING; HUNGRY, EATING) state [5] ;
    condition self [5];

    void pickup (int i) {
         state[i] = HUNGRY;
         test(i);
         if (state[i] != EATING) self [i].wait;
    }

    void putdown (int i) {
         state[i] = THINKING;
            // test left and right neighbors
          test((i + 4) % 5);
          test((i + 1) % 5);
    }
```

Solution to Dining Philosophers (cont)

```
void test (int i) {
    if ( (state[(i + 4) % 5] != EATING) &&
    (state[i] == HUNGRY) &&
    (state[(i + 1) % 5] != EATING) ) {
        state[i] = EATING ;
          self[i].signal () ;
    }
}

initialization_code() {
    for (int i = 0; i < 5; i++)
    state[i] = THINKING;
}
}
```

Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads
- Java (homework assignment)

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
- Uses condition variables 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 which may act as either mutexes and semaphores
- Dispatcher objects may also provide events
  - An event acts much like a condition variable

Linux Synchronization

- Linux:
  - disables interrupts to implement short critical sections
- Linux provides:
  - semaphores
  - spin locks

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

Exercise 1: