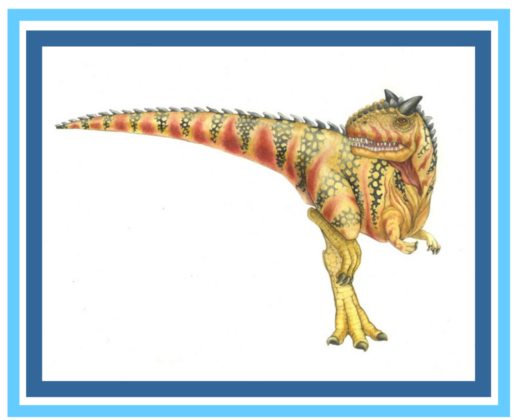
Chapter 6: Synchronization Tools



Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization

Background

- **Concurrent access to shared data** may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Classic problem – **Producer & Consumer**
  - Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by 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.
  - Examples:
    - Data from bar-code reader consumed by device driver
    - Data in a file you want to print consumed by printer spooler, which produces data consumed by line printer device driver
    - Web server produces data consumed by client’s web browser

Producer   Consumer

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

```
while (true)
 {
    while (count == 0)
    ; // do nothing
    nextConsumed =  buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
    /*consume the item in nextConsumed
 }
```

Are these code correct ???

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

- Consider 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 Problem

- Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc.
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

```
while (true) {

        entry section

            critical section

        exit section

            remainder section

}
```

Solution to Critical-Section Problem

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
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the N processes

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 – TestAndSet ()
  - Or swap contents of two memory words

TestAndSet Instruction

- Definition:

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

If *target = TRUE - > TestAndSet = TRUE; *target = TRUE
If *target = FALSE - > TestAndSet = FALSE; *target = TRUE

Solution Using test_and_set()

- Shared boolean variable **`lock`**, initialized to **`false`**
- Solution:

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

                /* critical section */

    lock = false;
                /* remainder section */
 } while (true);
```

- Does it solve the critical-section problem?

Lock = False;

Process P1

Process P2

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

  /* critical section */

     lock = false;
   /* remainder section */
       } while (true);
```

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

  /* critical section */

     lock = false;
   /* remainder section */
       } while (true);
```

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

  /* critical section */

     lock = false;
   /* remainder section */
       } while (true);
```

If *lock = TRUE - > TestAndSet = TRUE; *lock = TRUE
If *lock = FALSE - > TestAndSet = FALSE; *lock = TRUE

Bounded Waiting with TestAndSet

- Shared boolean variable **lock, initialized to FALSE**.
- Solution for *i* th process (total n number of processes) :

```
do { waiting[i] = True;
   while ( waiting[i] &&TestAndSet (&lock ))
           ;   /* do nothing
     waiting[i] = False;
     //    critical section
     j = ( i+1 ) % n ;
     while ( (j!=i) && !waiting[j] )
           j = (j+1) % n;

     if ( j == i)
         lock = FALSE;
     else
         waiting[j] = False;

     //      remainder section
```

	i		j						

Boolean array Waiting[ ]
With size of total number of processes

Swap Instruction

- **Definition**:

```
void Swap (boolean *a, boolean *b)
 {
     boolean temp = *a;
     *a = *b;
     *b = temp:
 }
```

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key.
- Solution:

```
do {
    key = TRUE;
     while ( key == TRUE)
          Swap (&lock, &key );
     //    critical section
      lock = FALSE;
     //      remainder section
   } while ( TRUE);
```

Semaphore

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Synchronization tool that is easier to use than TestAndSet()
- 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; 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 critical section.
  - Could now have **busy waiting in critical section implementation - Spinlock**
    - 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);  }
}
```

Liveness

- Processes may have to wait indefinitely while trying to acquire a synchronization tool such as a mutex lock or semaphore.
- Waiting indefinitely violates the progress and bounded-waiting criteria discussed at the beginning of this chapter.
- **Liveness** refers to a set of properties that a system must satisfy to ensure processes make progress.
- Indefinite waiting is an example of a liveness failure.

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

```
P0                    P1
   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.
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

Classical Problems of Synchronization

- Bounded-Buffer 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.

- 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);
```

- 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.
- Example: making an airline reservation
  - When you browse to look at flight schedules the web site is acting as a reader on your behalf
  - When you reserve a seat, the web site has to write into the database to make the reservation
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1.
  - Semaphore wrt initialized to 1.
  - Integer readcount initialized to 0.

Readers-Writers Problem (Cont.)

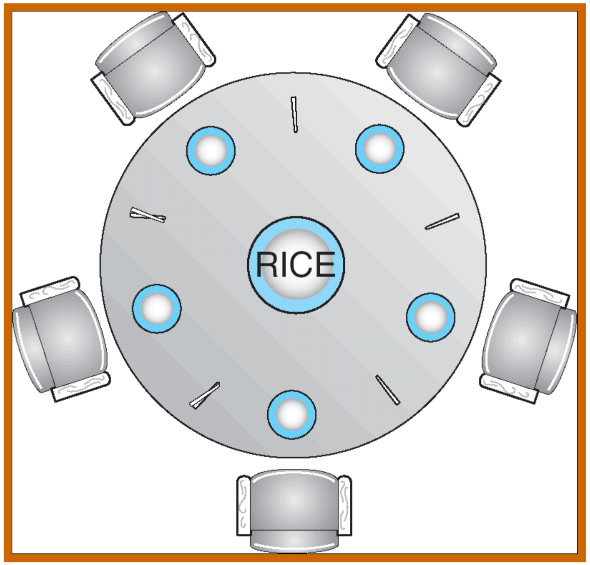
- The structure of a writer process

```
do {
      wait (wrt) ;

        //    writing is performed

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

- The structure of a reader process

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

            // reading is performed

      wait (mutex) ;
      readcount  - - ;
      if (readcount  == 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 (Cont.)

- 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!
  - Could cause deadlock

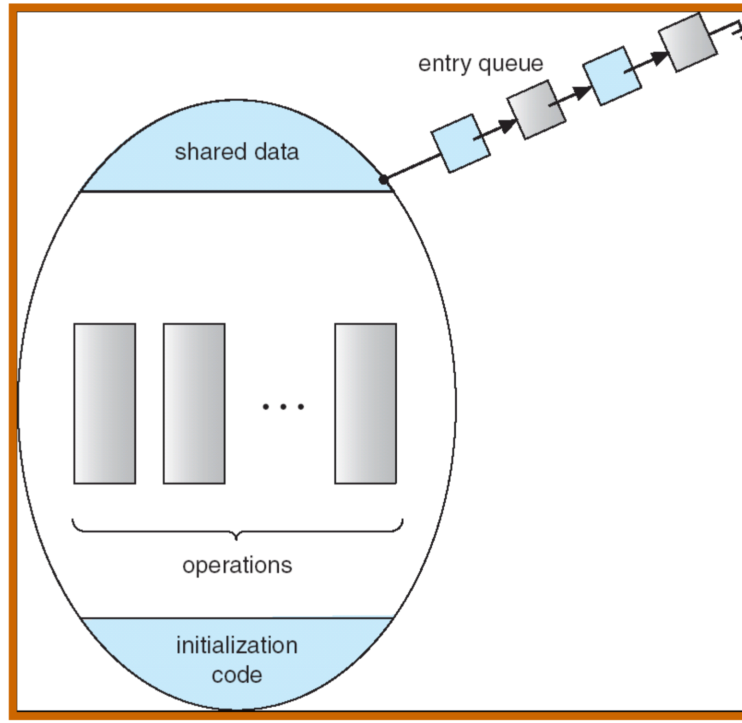
Problems with Semaphores

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

Monitors

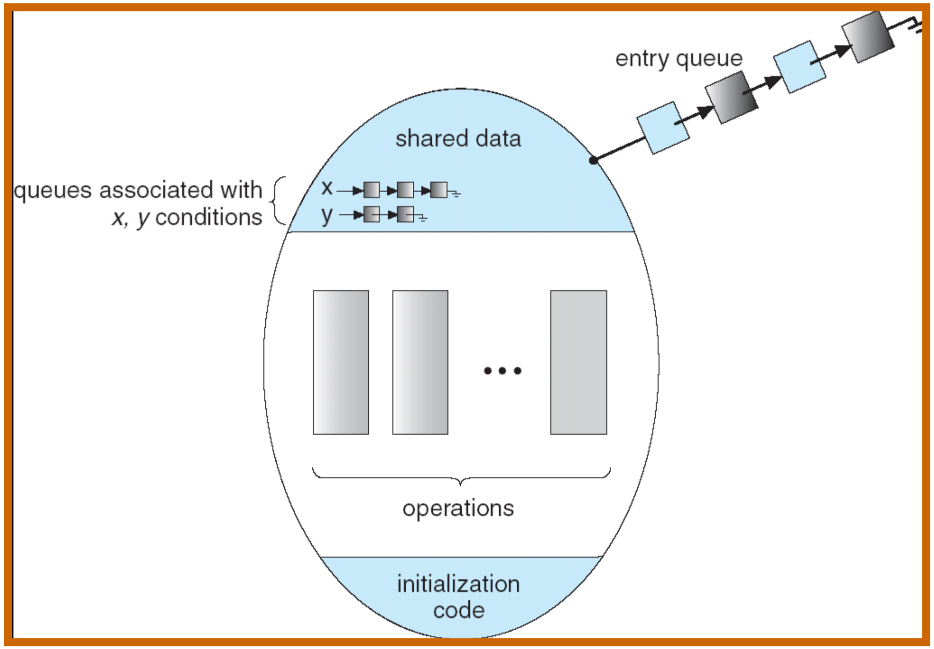
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- **Only one process may be active within the monitor at a time**

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

    procedure Pn (…) {……}

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



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 ()



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);
      }
      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
- Android

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

Android Synchronization

- Disables interrupts to implement short critical sections
- For example, write checkpoint

  - Start with mutex_lock()

```
int write_checkpoint(struct f2fs_sb_info *sbi, struct cp_control
{
    struct f2fs_checkpoint *ckpt = F2FS_CKPT(sbi);
    unsigned long long ckpt_ver;
    int err = 0;

    mutex_lock(&sbi->cp_mutex);
```

  - Finish with mutex_unlock()

```
     trace_f2fs_write_checkpoint(sbi->sb, cpc->reason, "finish checkpo
 out:
     mutex_unlock(&sbi->cp_mutex);
     return err;
 } ? end write_checkpoint ?
```

End of Chapter 6