Process Synchronization

5.1 Background

Since processes frequently needs to communicate with other processes therefore, there is a need for a well- structured communication, without using interrupts, among processes.

A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition.**

To guard against the race condition ensure only one process at a time can be manipulating the variable or data. To make such a guarantee processes need to be synchronized in some way

5.2 The Critical-Section Problem

Consider a system consisting of n processes {P0, P1, ..., Pn−1}. Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on. when one process is executing in its critical section, no other process is allowed to execute in its critical section. The critical-section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section. The general structure of a typical process Pi is shown in Figure

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

General structure of a typical process P_i.

A solution to the critical-section problem must satisfy the following three requirements:

1. **Mutual exclusion**. If process *Pi* 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 some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

3. Bounded waiting. There exists a bound, or limit, 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.

Two general approaches are used to handle critical sections in operating systems:

- **Preemptive kernels:** A preemptive kernel allows a process to be preempted while it is running in kernel mode.
- **Nonpreemptive kernels**.. A nonpreemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

5.3 Peterson's Solution

A classic software-based solution to the critical-section problem known as **Peterson's solution**. It addresses the requirements of mutual exclusion, progress, and bounded waiting.

- It Is 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 Pi is ready.
- The structure of process *Pi* in Peterson's solution:

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

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

- It proves that
    1. Mutual exclusion is preserved
    2. Progress requirement is satisfied
    3. Bounded-waiting requirement is met

5.4 Synchronization Hardware

Software-based solutions such as Peterson's are not guaranteed to work on modern computer architectures. Simple hardware instructions can be used effectively in solving the critical-section problem. These solutions are based on the **locking** —that is, protecting critical regions through the use of locks.

```
do {
    acquire lock
          critical section
    release lock
          remainder section
} while (TRUE);
```

Solution to Critical Section problem using locks

- 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(Swap( )).
- The definition of the test and set() instruction

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

- Using test and set() instruction, mutual exclusion can be implemented by declaring a boolean variable lock, initialized to false. The structure of process *Pi* is shown in Figure:

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

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

Mutual-exclusion implementation with test and set().

- Using Swap() instruction, mutual exclusion can be provided as : A global Boolean variable lock is declared and is initialized to *false* and each process has a local Boolean variable *key*.

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

The definition of the Swap() instruction

```
do {
       key = TRUE;
       while ( key == TRUE)
              Swap (&lock, &key );

                //    critical section

        lock = FALSE;

               //     remainder section

 } while (TRUE);
```

Mutual exclusion implementation with Swap() instruction

- Test and Set() instruction & Swap() Instruction do not satisfy the bounded-waiting requirement.

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   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 */
} while (true);
```

. Bounded-waiting mutual exclusion with test and set()

5.5 Semaphores

The hardware-based solutions to the critical-section problem are complicated as well as generally inaccessible to application programmers. So operating-systems designers build software tools to solve the critical-section problem, and this synchronization tool called as Semaphore.

- Semaphore *S is an* integer variable
- Two standard operations modify S: wait() and signal()
  Originally called P() and V()
- Can only be accessed via two indivisible (atomic) operations

```
• wait (S) {
     while S <= 0
              ; // no-op
       S--;
  }
• signal (S) {
    S++;
  }
```

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time.

Usage:

Semaphore classified into:

- Counting semaphore: Value can range over an unrestricted domain.

- Binary semaphore(Mutex locks): Value can range only between from 0 & 1. It provides mutual exclusion.

```
Semaphore mutex;    //  initialized to 1
do {
    wait (mutex);
       // Critical Section
    signal (mutex);
         // remainder section
} while (TRUE);
```

- Consider 2 concurrently running processes:

  ```
  S1;
  signal(synch);
  ```

  In process *P*1, and the statements

  ```
  wait(synch);
  S2;
  ```

  Because synch is initialized to 0, *P*2 will execute *S*2 only after *P*1 has invoked signal(synch), which is after statement *S*1 has been executed.

Implementation:

The disadvantage of the semaphore is **busy waiting** i.e While a process is in critical section, any other process that tries to enter its critical section must loop continuously in the entry code. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a **spin lock** because the process spins while waiting for the lock.

Solution for Busy Waiting problem:

Modify the definition of the wait() and signal()operations as follows: When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait. Rather than engaging in busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation. The process is restarted by a wakeup() operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue.

To implement semaphores under this definition, define a semaphore as follows:

```
typedef struct
  { int value;
 struct process *list;
} semaphore;
```

Each semaphore has an integer value and a list of processes list. When a process must wait on a semaphore, it is added to the list of processes. A signal() operation removes one process from the list of waiting processes and awakens that process. Now, the wait() semaphore operation can be defined as:

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

and the signal() semaphore operation can be defined as

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

The block() operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P.

5.6.3 Deadlocks and Starvation

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes, these processes are said to be deadlocked.

Consider below example: a system consisting of two processes, *P*0 and *P*1, each accessing two semaphores, S and Q, set to the value 1:

```
     P0              P1

wait(S);         wait(Q);
wait(Q);         wait(S);
    .                .
    .                .
    .                .
signal(S);       signal(Q);
signal(Q);       signal(S);
```

Suppose that *P*0 executes wait(S) and then *P*1 executes wait(Q).When *P*0 executes wait(Q), it must wait until *P*1 executes signal(Q). Similarly, when *P*1 executes wait(S), it must wait until *P*0 executes signal(S). Since these signal() operations cannot be executed, *P*0 and *P*1 are deadlocked.

Another problem related to deadlocks is **indefinite blocking** or **starvation**.

5.7 Classic Problems of Synchronization

5.7.1 The 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

Code for producer is given below:

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

Code for consumer is given below:

```
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);
```

5.7.2 The 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
- Several variations of how readers and writers are treated – all involve priorities.
  - ✓ *First* variation – no reader kept waiting unless writer has permission to use shared object
  - ✓ *Second* variation- Once writer is ready, it performs asap.
- Shared Data
  - ✓ Data set
  - ✓ Semaphore mutex initialized to 1
  - ✓ Semaphore wrt initialized to 1
  - ✓ Integer readcount initialized to 0

The structure of writer process:

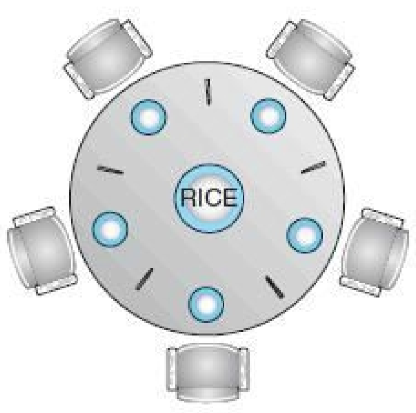
```
do {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
} while (true);
```

The structure of 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);
```

5.7.3 The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks.



A philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again.

It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

Solution: One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore. She releases her chopsticks by executing the signal() operation on the appropriate semaphores. Thus, the shared data are

```
semaphore chopstick[5];
```

where all the elements of chopstick are initialized to 1. The structure of philosopher *i* is shown in Figure

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for awhile */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
} while (true);
```

Several possible remedies to the deadlock problem are replaced by:

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available.
- Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even numbered philosopher picks up her right chopstick and then her left chopstick.

5.8 Monitors

Incorrect use of semaphore operations:

- Suppose that a process interchanges the order in which the wait() and signal() operations on the semaphore mutex are executed, resulting in the following execution:

```
signal(mutex);
   ...
critical section
   ...
wait(mutex);
```

- Suppose that a process replaces signal(mutex) with wait(mutex). That is, it executes

```
wait(mutex);
   ...
critical section
   ...
wait(mutex);
```

In this case, a deadlock will occur.

- Suppose that a process omits the wait(mutex), or the signal(mutex), or both. In this case, either mutual exclusion is violated or a deadlock will occur.

Solution:

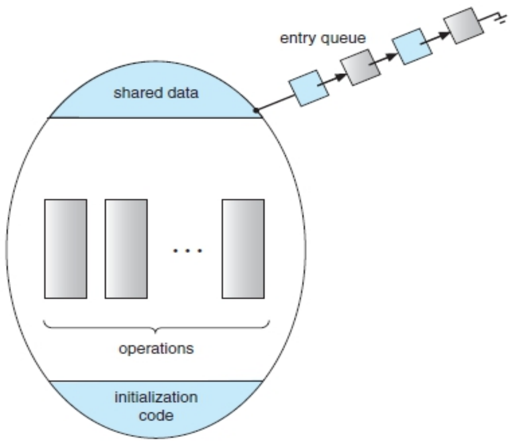
Monitor: An **abstract data type**—or **ADT**—encapsulates data with a set of functions to operate on that data that are independent of any specific implementation of the ADT.

A ***monitor type*** is an ADT that includes a set of programmer defined operations that are provided with mutual exclusion within the monitor. The monitor type also declares the variables whose values define the state of an instance of that type, along with the bodies of functions that operate on those variables. The monitor construct ensures that only one process at a time is active within the monitor.

The syntax of a monitor type is shown in Figure:

```
monitor monitor name
{
   /* shared variable declarations */

   function P1 ( . . . ) {
      . . .
   }

   function P2 ( . . . ) {
      . . .
   }

         .
         .
         .
   function Pn ( . . . ) {
      . . .
   }

   initialization_code ( . . . ) {
      . . .
   }
}
```

Schematic view of a monitor:



To have a powerful Synchronization schemes a *condition* construct is added to the Monitor. So synchronization scheme can be defined with one or more variables of type *condition.*

```
condition x, y;
```

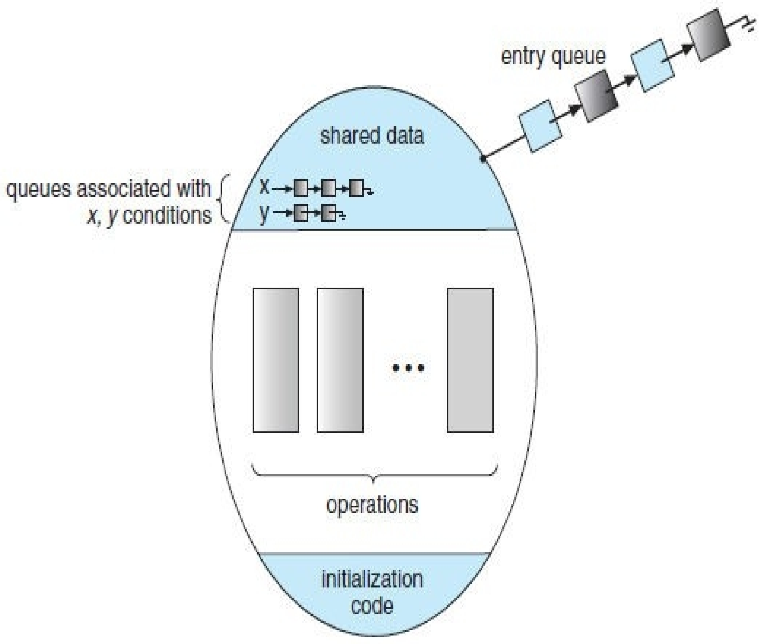
The only operations that can be invoked on a condition variable are wait() and signal(). The operation

```
x.wait();
```

means that the process invoking this operation is suspended until another process invokes

```
x.signal();
```

The x.signal() operation resumes exactly one suspended process. If no process is suspended, then the signal() operation has no effect; that is, the state of x is the same as if the operation had never been executed. Contrast this operation with the signal() operation associated with semaphores, which always affects the state of the semaphore.



5.8.2 Dining-Philosophers Solution Using Monitors

A deadlock-free solution to the dining-philosophers problem using monitor concepts. This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.

Consider following data structure:

```
enum {THINKING, HUNGRY, EATING} state[5];
```

Philosopher *i* can set the variable state[i] = EATING only if her two neighbors are not eating: (state[(i+4) % 5] != EATING) and(state[(i+1) % 5] != EATING).

And also declare:

Condition self[5];

This allows philosopher *i* to delay herself when she is hungry but is unable to obtain the chopsticks she needs.

A monitor solution to the dining-philosopher problem:

```
monitor DiningPhilosophers
  {
     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;
       }
 }
```

5.8.3 Implementing a Monitor Using Semaphores

For each monitor, a semaphore mutex (initialized to 1) is provided. A process must execute wait(mutex) before entering the monitor and must execute signal(mutex) after leaving the monitor.

Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore, next, is introduced, initialized to 0. The signaling processes can use next to suspend themselves. An integer variable next_count is also provided to count the number of processes suspended on next. Thus, each external function F is replaced by

```
wait(mutex);
    ...
  body of F
    ...
if (next_count > 0)
  signal(next);
else
  signal(mutex);
```

Mutual exclusion within a monitor is ensured.

For each condition x, we introduce a semaphore x sem and an integer variable x count, both initialized to 0. The operation x.wait() can now be implemented as

```
x_count++;
if (next_count > 0)
        signal(next);
else
        signal(mutex);
wait(x_sem);
x_count--;
```

The operation x.signal() can be implemented as

```
if (x_count > 0) {
  next_count++;
  signal(x_sem);
  wait(next);
  next_count--;
}
```

5.8.4 Resuming Processes within a Monitor

If several processes are suspended on condition x, and an x.signal() operation is executed by some process, then to determine which of the suspended processes should be resumed next, one simple solution is to use a first-come, first-served (FCFS) ordering, so that the process that has been waiting the longest is resumed first. For this purpose, the **conditional-wait** construct can be used. This construct has the form

```
x.wait(c);
```

where c is an integer expression that is evaluated when the wait() operation is executed. The value of c, which is called a **priority number**, is then stored with the name of the process that is suspended. When x.signal() is executed, the process with the smallest priority number is resumed next.

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;

   void acquire(int time) {
      if (busy)
         x.wait(time);
      busy = true;
   }

   void release() {
      busy = false;
      x.signal();
   }

   initialization_code() {
      busy = false;
   }
}
```

The ResourceAllocator monitor shown in the above Figure, which controls the allocation of a single resource among competing processes.

A process that needs to access the resource in question must observe the following sequence:

```
R.acquire(t);
...
access the resource;
...
R.release();
```

where R is an instance of type ResourceAllocator.

The monitor concept cannot guarantee that the preceding access sequence will be observed. In particular, the following problems can occur:

- A process might access a resource without first gaining access permission to the resource.
- A process might never release a resource once it has been granted access to the resource.
- A process might attempt to release a resource that it never requested.
- A process might request the same resource twice (without first releasing the resource).