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
- Atomic Transactions

Background

- Concurrent access to shared data may result in **data inconsistency**
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- 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.

Producer

```
while (true) {

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

Consumer

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

                        /*  consume the item in nextConsumed
 }
```

Problem with concurrent execution

- Although both the producer and consumer routines are correct separately, they may not function correctly when executed **concurrently**.
- Suppose that the value of the variable counter is currently 5 and that the producer and consumer processes execute the statements "counter++" and "counter--" concurrently.
- Following the execution of these two statements, the value of the variable counter may be 4, 5, or 6!
- The only correct result, though, is counter 5, which is generated correctly if the producer and consumer execute separately.

Race Condition

- counter++ could be implemented in machine language as

  register1 = counter
  register1 = register1 + 1
  counter = register1

- count-- could be implemented as

  register2 = counter
  register2 = register2 - 1
  counter = register2

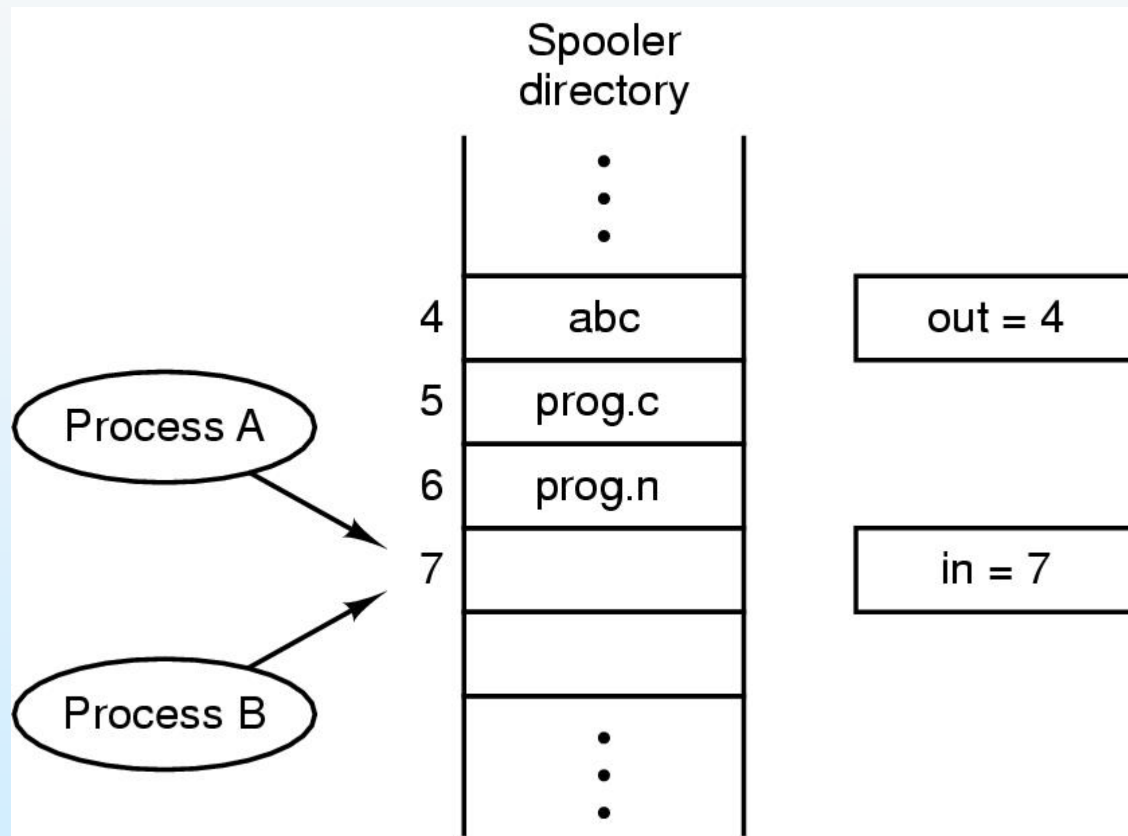
- Consider this execution interleaving with "count = 5" initially:

  T0: producer execute register1 = counter {register1 = 5}
  T1: producer execute register1 = register1 + 1 {register1 = 6}
  T2: consumer execute register2 = counter {register2 = 5} !!!
  T3: consumer execute register2 = register2 - 1 {register2 = 4}
  T4: producer execute counter = register1 {counter = 6 }
  T5: consumer execute counter = register2 {counter = 4}

Race Condition

- Notice that we have arrived at the incorrect state "counter 4", indicating that four buffers are full, when, in fact, **five buffers are full**. If we reversed the order of the statements at T_4 and T_5, we would arrive at the incorrect state "counter == 6".
- We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter **concurrently**.
- A situation like this, 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 above, we need to ensure that only **one process at a time** can be manipulating the variable counter. To make such a guarantee, we require that the processes be synchronized in some way.
- Because of the importance of this issue, a major portion of OS research is concerned with process **synchronization** and **coordination**.

Race between processes



Two processes want to access shared memory at same time

Critical-Section Problem

- Consider a system consisting of *n* processes {P_0, P_1, ... , P_{n-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.
- The important feature of the system is that, when one process is executing in its **critical section**, no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time.
- 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 process structure

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

Figure 6.1 General structure of a typical process P_i.

Preemptive and nonpremptive Kernels

- Two general approaches are used to handle critical sections in operating systems:
  - A **preemptive kernel** allows a process to be preempted while it is running in kernel mode.
  - 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.
- Obviously, a **nonpreemptive kernel** is essentially free from **race conditions** on kernel data structures, as only one process is active in the kernel at a time.
- Windows XP and Windows 2000 are nonpreemptive kernels, as is the traditional UNIX kernel. Prior to Linux 2.6, the Linux kernel was nonpreemptive as well. However, with the release of the 2.6 kernel, Linux changed to the preemptive model. Several commercial versions of UNIX are preemptive, including Solaris and IRIX

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 some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on 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.

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

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); ///// LOOP

            CRITICAL SECTION

        flag[i] = FALSE;

             REMAINDER SECTION

 }
```

Mutual exclusion using critical regions

A enters critical region

A leaves critical region

Process A

Process B

B attempts to enter critical region

B enters critical region

B leaves critical region

B blocked

T_1 T_2 T_3 T_4

Time →

Locking

```
do {
     acquire lock

          critical section

     release lock

          remainder section

} while (TRUE);
```

Figure 6.3 Solution to the critical-section problem using locks.

Race conditions are prevented by requiring that critical regions be protected by locks. That is, a process must acquire a lock before entering a critical section; it releases the lock when it exits the critical 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

TestAndSet Instruction

Definition:

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

- The important characteristic is that this instruction is executed atomically.
- Thus, if two TestAndSet() instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.
- If the machine supports the TestAndSet() instruction, then we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false.

Solution using TestAndSet

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

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

                //    critical section

        lock = FALSE;

                //      remainder section

}
```

Swap Instruction

- Definition:

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

Solution using Swap

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

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

                //    critical section

         lock = FALSE;

                //      remainder section

 }
```

Semaphore

- The various hardware-based solutions to the critical-section problem (using the TestAndSet() and Swap() instructions) are complicated for application programmers to use.
- To overcome this difficulty, we can use a synchronization tool called a semaphore.
- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal().
- The wait() operation was originally termed P (from the Dutch *proberen,* "to test"); signal() was originally called V (from *verhagen,* "to increment").

Semaphore

- 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) {

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

  - signal (S) {

```
signal (S) {
   S++;
}
```

Semaphore

- All the modifications to the integer value of the semaphore in the wait() and signal() operations must be executed indivisibly.
- That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.
- In addition, in the case of wait(S), the testing of the integer value of S (S <= 0), and its possible modification (S--), must also be executed without interruption.

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 in many OS as mutex locks
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion
- Can solve also Synchronization problems.

Mutual exclusion with semaphores

```
do {
   waiting(mutex);

      // critical section

   signal(mutex);

      // remainder section
}while (TRUE);
```

Figure 6.9 Mutual-exclusion implementation with semaphores.

Producer-Consumer with Semaphores

```
#define N 100                           /* number of slots in the buffer */
typedef int semaphore;                  /* semaphores are a special kind of int */
semaphore mutex = 1;                    /* controls access to critical region */
semaphore empty = N;                    /* counts empty buffer slots */
semaphore full = 0;                     /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                     /* TRUE is the constant 1 */
          item = produce_item( );       /* generate something to put in buffer */
          down(&empty);                 /* decrement empty count */
          down(&mutex);                 /* enter critical region */
          insert_item(item);            /* put new item in buffer */
          up(&mutex);                   /* leave critical region */
          up(&full);                    /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                     /* infinite loop */
          down(&full);                  /* decrement full count */
          down(&mutex);                 /* enter critical region */
          item = remove_item( );        /* take item from buffer */
          up(&mutex);                   /* leave critical region */
          up(&empty);                   /* increment count of empty slots */
          consume_item(item);           /* do something with the item */
     }
}
```

Spinlocks

- The main disadvantage of the semaphore definition given here is that it requires busy waiting.
- While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code.
- This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes.
- Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a spinlock because the process "spins" while waiting for the lock.
- Spinlocks do have an advantage in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time.
- Thus, when locks are expected to be held for short times, spinlocks are useful; they are often employed on multiprocessor systems where one thread can "spin" on one processor while another thread performs its critical section on another processor.)

No busy waiting

- To overcome the need for busy waiting, we can modify the definition of the wait() and signal() semaphore operations.
- When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait.
  - However, 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.

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.

Sleep and wakeup

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count − 1;                    /* decrement count of items in buffer */
          if (count == N − 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

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);  }
}
```

Deadlocks

- 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 only by one of the waiting processes.
- The event in question is the execution of a signal() operation. When such a state is reached, these processes are said to be deadlocked.
- We say that a set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are *resource acquisition and release.*
- Another problem related to deadlocks is indefinite blocking, or starvation, a situation in which processes wait indefinitely within the semaphore.

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

- We present here a number of synchronization problems as examples of a large class of concurrency-control problems.
- These problems are used for testing nearly every newly proposed synchronization scheme.
- Problems
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- *N* buffers, each can hold one item
- The empty and full semaphores count the number of empty and full buffers.
- Semaphore mutex initialized to the value 1
  - The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is 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 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

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

            //    writing is performed

        signal (wrt) ;
}
```

Silberschatz, Galvin and Gagne ©2005

Readers-Writers Problem (Cont.)

- The structure of a reader process

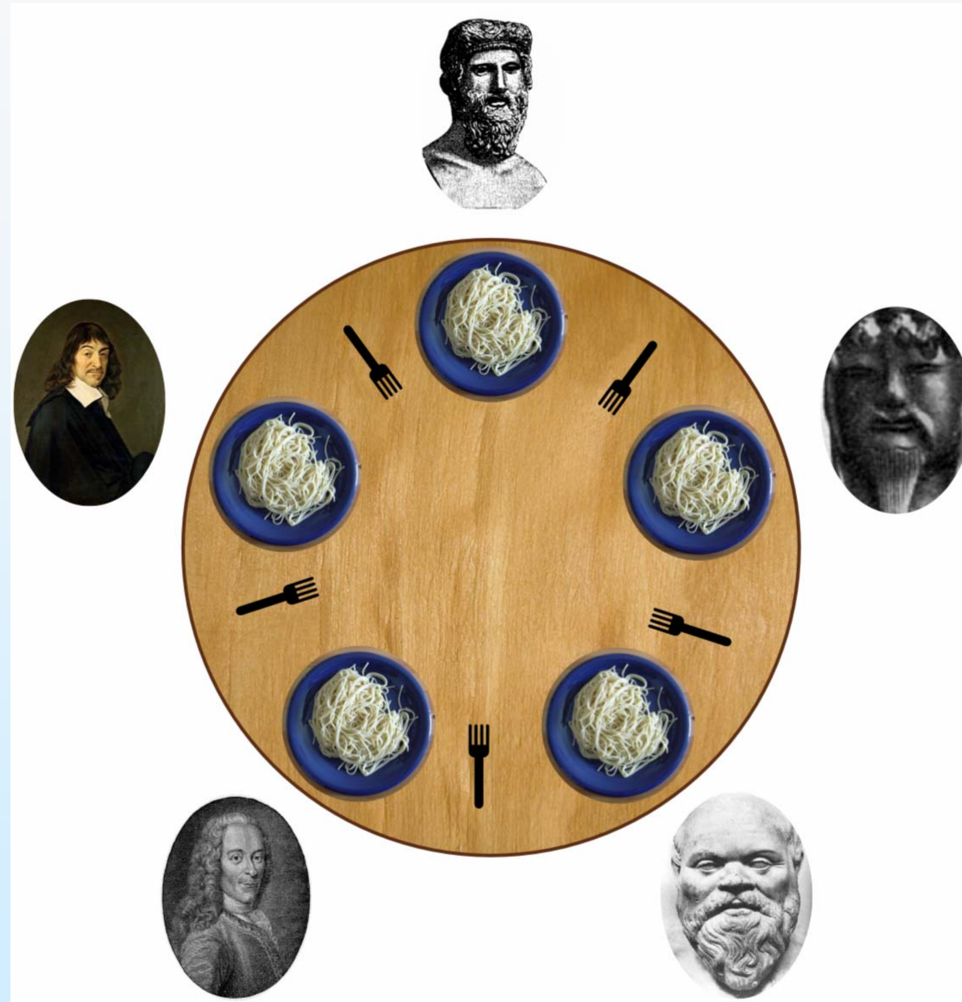
```
while (true) {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)  wait (wrt) ;
        signal (mutex)

                // reading is performed

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

Dining-Philosophers Problem

- Philosophers eat or think
- Eating needs 2 forks
- Pick one fork at a time
- How to prevent deadlock



Plato
Confucius
Socrates
Voltaire
Descartes

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

Silberschatz, Galvin and Gagne ©2005

A non solution for Dining-Philosophers

```
#define N 5                                /* number of philosophers */

void philosopher(int i)                    /* i: philosopher number, from 0 to 4 */
{
     while (TRUE) {
          think( );                        /* philosopher is thinking */
          take_fork(i);                    /* take left fork */
          take_fork((i+1) % N);            /* take right fork; % is modulo operator */
          eat( );                          /* yum-yum, spaghetti */
          put_fork(i);                     /* put left fork back on the table */
          put_fork((i+1) % N);             /* put right fork back on the table */
     }
}
```

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

}
```

Problems with semaphores 1

- Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect
  - These errors happen only if some particular execution sequences take place and these sequences do not always occur
- Each process must execute wait (mutex) before entering the critical section and signal (mutex) afterward.
  - If this sequence is not observed, two processes may be in their critical sections simultaneously.
  - These difficulties will arise even if a *single* process is not well behaved.
  - This situation may be caused by an honest programming error or an uncooperative programmer.

Problems with semaphors 2

- 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);
```

- In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement.
- This error may be discovered only if several processes are simultaneously active in their critical sections. Note that this situation may not always be reproducible.

Problems with semaphors 3

- 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.

Problems with semaphors 4

- 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.

- These examples illustrate that various types of errors can be generated easily when programmers use semaphores incorrectly to solve the critical-section problem.
- Similar problems may arise in the other synchronization models.
- To deal with such errors, researchers have developed high-level language constructs.
- One fundamental high-level synchronization construct is the **monitor** type.

Monitors

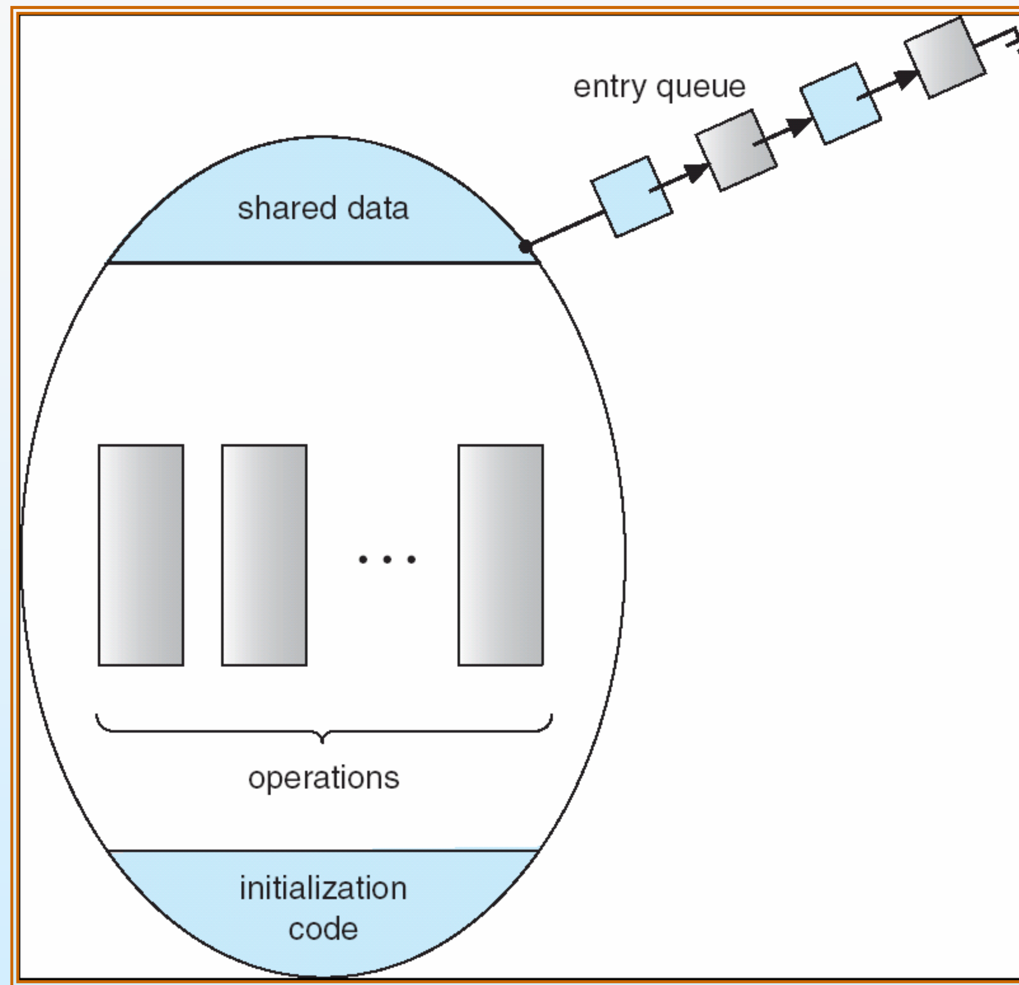
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

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

   procedure Pn (…) {……}

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

Schematic view of a Monitor



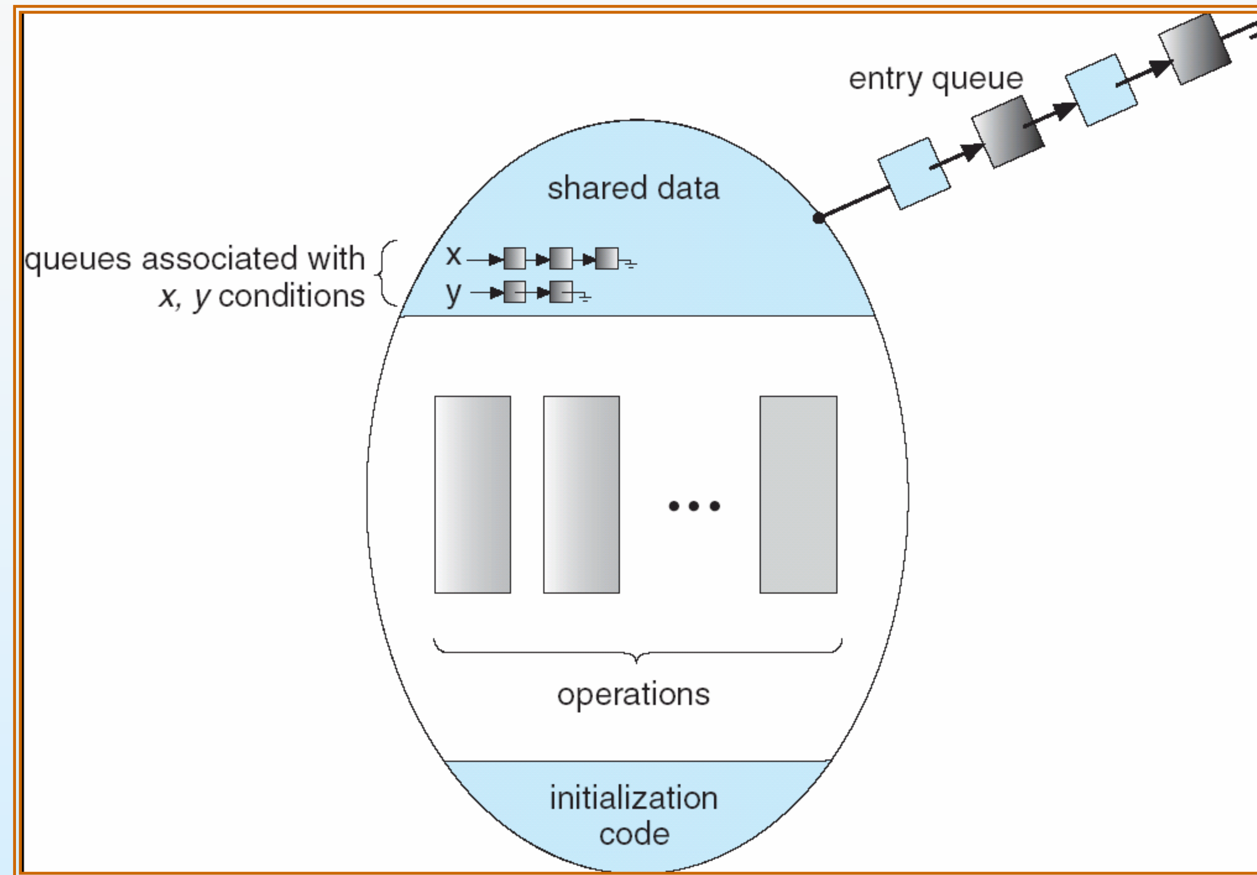
The monitor construct ensures that only one process at a time can be active within the monitor. Consequently, the programmer does not need to code this synchronization constraint explicitly

Condition Variables

- However, the monitor construct, as defined so far, is not sufficiently powerful for modeling some synchronization schemes.
- For this purpose, we need to define additional synchronization mechanisms. These mechanisms are provided by the ***condition*** construct.

- condition x, y;
- The only operations that can be invoked on a condition variable are wait() and signal().
  - 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



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.

Deadlock-free 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;
}
}
```

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+l) % 5J ! = eating).

It is easy to show that this solution ensures that no two neighbors are eating simultaneously and that no deadlocks will occur.
We note, however, that it is possible for a philosopher to starve to death, ☺.

Solution to Dining Philosophers (cont)

- Each philosopher *i* invokes the operations pickup() and putdown() in the following sequence:

  dp.pickup (i)

  EAT

  dp.putdown (i)

Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads

- The synchronization methods available in these differing systems vary in subtle and significant ways.

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

Silberschatz, Galvin and Gagne ©2005

Pthreads Synchronization

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

Atomic Transactions

- System Model
- Log-based Recovery
- Checkpoints
- Concurrent Atomic Transactions

System Model

- Assures that operations happen as a single logical unit of work, in its entirety, or not at all
- Related to field of database systems
- Challenge is assuring atomicity despite computer system failures
- Transaction - collection of instructions or operations that performs a single logical function
  - Here we are concerned with changes to stable storage – disk
  - Transaction is series of read and write operations
  - Terminated by commit (transaction successful) or abort (transaction failed) operation
  - Aborted transaction must be rolled back to undo any changes it performed

Silberschatz, Galvin and Gagne ©2005

Types of Storage Media

- Volatile storage – information stored here does not survive system crashes
  - Example: main memory, cache
- Nonvolatile storage – Information usually survives crashes
  - Example: disk and tape
- Stable storage – Information never lost
  - Not actually possible, so approximated via replication or RAID to devices with independent failure modes

Goal is to assure transaction atomicity where failures cause loss of information on volatile storage

Log-Based Recovery

- Record to stable storage information about all modifications by a transaction
- Most common is write-ahead logging
  - Log on stable storage, each log record describes single transaction write operation, including
    - Transaction name
    - Data item name
    - Old value
    - New value
  - <T_i starts> written to log when transaction T_i starts
  - <T_i commits> written when T_i commits
- Log entry must reach stable storage before operation on data occurs

Log-Based Recovery Algorithm

- Using the log, system can handle any volatile memory errors
  - Undo(T_i) restores value of all data updated by T_i
  - Redo(T_i) sets values of all data in transaction T_i to new values
- Undo(T_i) and redo(T_i) must be idempotent
  - Multiple executions must have the same result as one execution
- If system fails, restore state of all updated data via log
  - If log contains <T_i starts> without <T_i commits>, undo(T_i)
  - If log contains <T_i starts> and <T_i commits>, redo(T_i)

Checkpoints

- When a system failure occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone.
- In principle, we need to search the entire log to make these determinations. There are two major drawbacks to this approach:
  - The searching process is time consuming.
  - Most of the transactions that, according to our algorithm, need to be redone have already actually updated the data that the log says they need to modify.
    - Although redoing the data modifications will cause no harm (due to idempotency), it will nevertheless cause recovery to take longer.

Checkpoints

- Checkpoints shorten log and recovery time.
- To reduce the overhead, we introduce the concept of checkpoints. During execution, the system maintains the write-ahead log. In addition, the system periodically performs checkpoints that require the following sequence of actions to take place:
    1. Output all log records currently in volatile storage to stable storage
    2. Output all modified data from volatile to stable storage
    3. Output a log record <checkpoint> to the log on stable storage
- Once transaction T_i has been identified, the redo and undo operations need be applied only to transaction T_i and all transactions T_j that started executing after transaction T_i.

- Readings
  - Silberschatz, Chapter 6, except 6.7.3 and 6.7.4;
  -

End of Chapter 6