MONITORS

- Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in **timing errors**.
- That are difficult to detect, since these errors happen only if some particular execution sequences take place and these sequences do not always occur.
- 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.
- **Example1**:

  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.
- **Example2:**

  Suppose that a process replaces signal (mutex) with wait (mutex).

  wait(mutex);
  critical section

wait(mutex);

In this case, a deadlock will occur.

- **Example3:**
  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.
- To deal with such errors, researchers have developed a high-level language construct called **Monitors**.

Usage

- An ***abstract data type-*** **or ADT**- encapsulates private data with public methods to operate on that data.
- **A *monitor type* is an ADT which presents a set of programmer-defined operations that are provided mutual exclusion within the monitor.**
- **The monitor type also contains the declaration of variables whose values define the state of an instance of that type, along with the bodies of procedures or functions that operate on those variables.**
- The representation of a monitor type cannot be used directly by the various processes.
- A procedure defined within a monitor can access only those variables declared locally within the monitor and its formal parameters.

- The local variables of a monitor can be accessed by only the local procedures.

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

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

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

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

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

Figure 5.15 Syntax of a monitor.

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

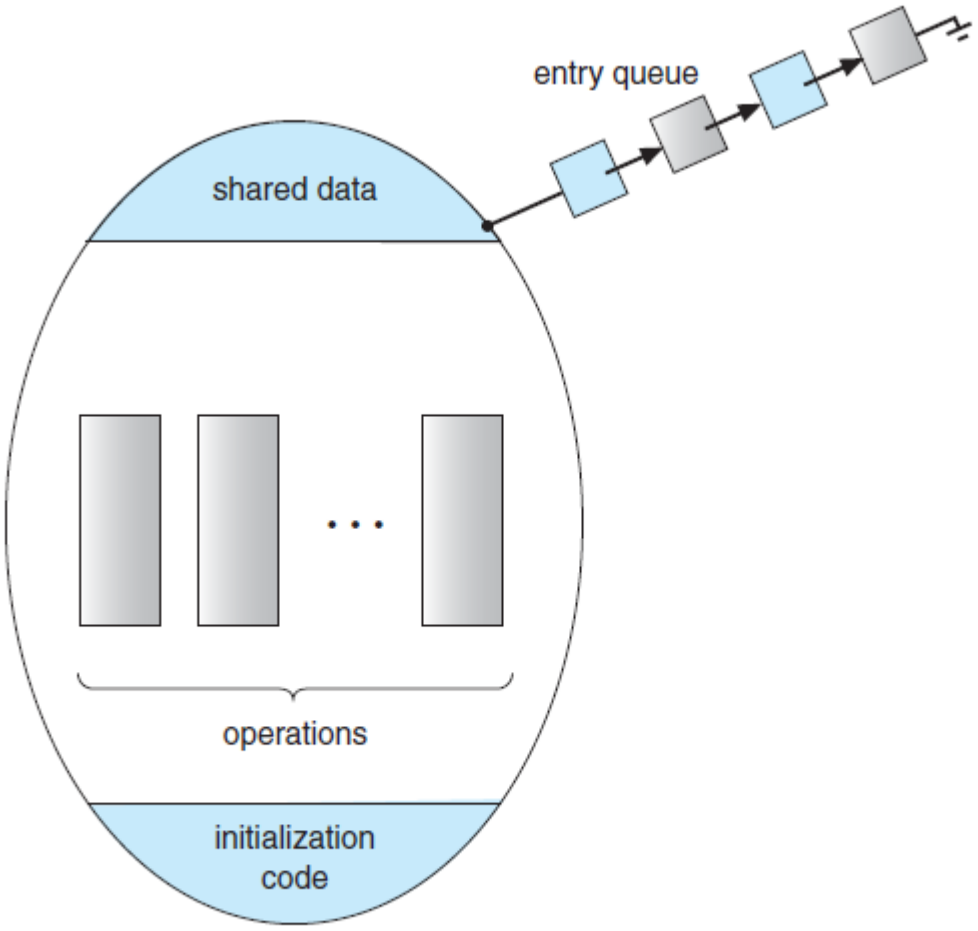

Figure 5.16 Schematic view of a monitor.

- We can define **additional synchronization mechanisms using the condition construct.**
- A programmer who needs to write a synchronization scheme can define 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
- This is a contrast with the signal() operation associated with semaphores, which always affects the state of the semaphore.

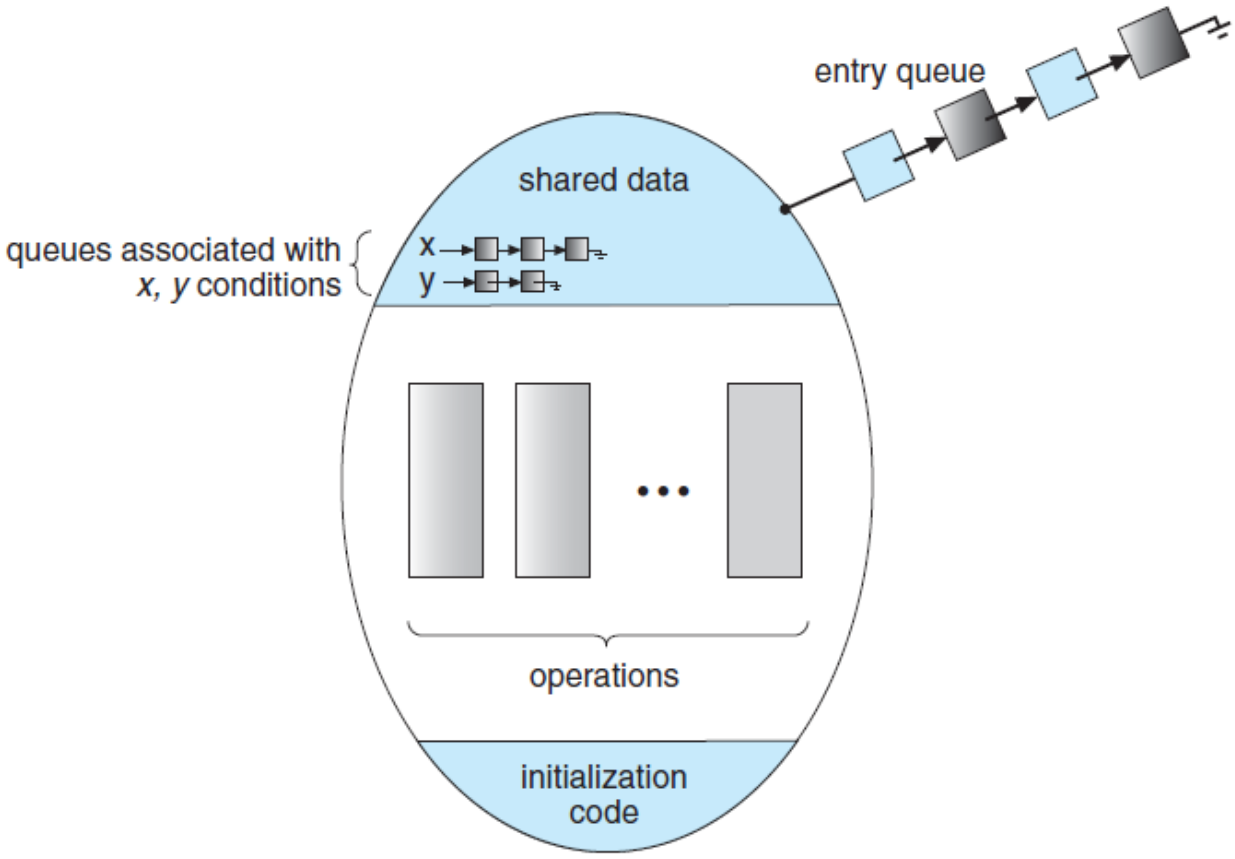

Figure 5.17 Monitor with condition variables.

- Suppose that, when the x. signal () operation is invoked by a process P, there exists a suspended process Q associated with condition x.
- Two possibilities exist:

1. **Signal and wait.** *P* either waits until *Q* leaves the monitor or waits for another condition.
2. **Signal and continue.** *Q* either waits until *P* leaves the monitor or waits for another condition.

- Usually we follow ***Signal and wait***.

Example for Monitor

- Consider the ResourceAllocator monitor, which controls the allocation of a single resource among competing processes.
- Each process, when requesting an allocation of this resource, specifies the maximum time it plans to use the resource.

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

where R is an instance of type ResourceAllocator.

```
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;
   }
}
```

Figure 5.19 A monitor to allocate a single resource.

Implementing 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.
- It follows signal and wait scheme. Since a signaling process must wait until the resumed process 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**.
- Each external procedure F is replaced by:

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

- Mutual exclusion within a monitor is thus 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 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--;
}
```

Resuming Processes within a Monitor

- If several processes are suspended on condition x, and an x. signal() operation is executed by some process, then we should determine which of the suspended processes should be resumed next.
- One simple solution is to use an **FCFS (First Come First Serve)** ordering, so that the process that has been waiting the longest is resumed first.
- In many circumstances, such a simple scheduling scheme is not adequate. For this purpose, the **conditional-wait** construct can be used.
- It has the form **x.wait(c);** 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.
- Eg: Consider the ***ResourceAllocator monitor***
- **The monitor allocates the resource to the process that has the shortest time-allocation request.**

CLASSICAL PROBLEMS OF SYNCHRONIZATION

1. Producer – Consumer Problem
   (Bounded Buffer Problem)
2. Readers – Writers Problem
3. Dining Philosophers Problem

PRODUCER – CONSUMER PROBLEM

- **A producer process produces an item that is consumed by a consumer process.**
- Example, a compiler may produce assembly code, which is consumed by an assembler. The assembler, in turn, produces object modules, which are consumed by the loader.
- **Producer and Consumer processes work concurrently in a cooperating manner**
- This problem is the best example for process synchronization
- **Consumer should consume only those items that Producer has already produced**
- One **solution** to the producer-consumer problem **uses shared memory**.
- To allow producer and consumer processes to run concurrently, we must have **a common buffer of items that can be filled by the producer and emptied by the consumer. (Shared buffer)**
- This buffer will reside in a region of memory that is shared by the producer and consumer processes.
- A producer can produce one item while the consumer is consuming another item.
- **The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.**

- Two types of buffers can be used.
  1. Unbounded Buffer
  2. Bounded Buffer
- The unbounded buffer places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items.
- **The bounded buffer assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.**
- **We focus more on bounded buffer. So the problem is also called bounded buffer problem**
- The following variables reside in a region of memory shared by the producer and consumer processes:

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
}item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- The shared **buffer is implemented as a circular array with two logical pointers: in and out.**
- The variable **in** points to the next free position in the buffer; // used by Producer

- **out** points to the first full position in the buffer. // used by Consumer
- The buffer is **empty when in== out;**
- The buffer is full when ((in+ 1)% BUFFER_SIZE) == out.
- The producer process has a local variable **nextProduced** in which the new item to be produced is stored.
- The consumer process has a local variable **nextConsumed** in which the item to be consumed is stored.
- This scheme allows at most BUFFER_SIZE - 1 items in the buffer at the same time.

```
item nextProduced;

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

Figure 3.14 The producer process.

```
item nextConsumed;

while (true) {
     while (in == out)
        ; // do nothing

     nextConsumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     /* consume the item in nextConsumed */
}
```

Figure 3.15 The consumer process.

- Since common resource is used, critical session problem arises and we can solve it by using semaphores.
- The **mutex** semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. (It is a binary semaphore)
- **The empty and full semaphores count the number of empty and full buffers. (They are counting semaphores)**
- **The semaphore empty is initialized to the value BUFFERSIZE; the semaphore full is initialized to the value 0.**

```
do {
    . . .
   // produce an item in nextp
    . . .
   wait(empty);
   wait(mutex);
    . . .
   // add nextp to buffer
    . . .
   signal(mutex);
   signal(full);
} while (TRUE);
```

Figure 6.10 The structure of the producer process.

```
do {
   wait(full);
   wait(mutex);
      . . .
   // remove an item from buffer to nextc
      . . .
   signal(mutex);
   signal(empty);
      . . .
   // consume the item in nextc
      . . .
} while (TRUE);
```

Figure 6.11 The structure of the consumer process.

THE READERS-WRITERS PROBLEM

- Suppose that a database is to be shared among several concurrent processes.
- Some of these processes may want only to read the database **(Reader),** whereas others may want to update the database **(Writer)**
- If two readers access the shared data simultaneously, no conflicts.
- If a writer and some other process (either a reader or a writer) access the database simultaneously, conflicts may arise.
- Ensure that the writers have exclusive access to the shared database while writing to the database.

- This synchronization problem is referred to as the *readers-writers problem.*
- The readers-writers problem has several variations.
- **The *first* readers-writers problem**, requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object.
- In other words, no reader should wait for other readers to finish simply because a writer is waiting.
- **The *second* readers-writers problem** requires that, once a writer is ready, that writer performs its write as soon as possible.
- In other words, if a writer is waiting to access the object, no new readers may start reading.
- **A solution to either problem may result in starvation.**
- In the first case, writers may starve; in the second case, readers may starve.
- In the **solution to the first readers-writers problem**, the reader processes share the following data structures:

semaphore mutex, wrt;

int readcount;

- The semaphores **mutex** and **wrt** are initialized to 1; **readcount** is initialized to 0.
- The semaphore wrt is common to both reader and writer processes.
- The mutex semaphore is used to ensure mutual exclusion when the variable readcount is updated.

- The readcount variable keeps track of how many processes are currently reading the object.
- The semaphore wrt functions as a mutual-exclusion semaphore for the writers.
- It is also used by the first or last reader that enters or exits the critical section.
- If a writer is in the critical section and *n* readers are waiting, then one reader is queued on wrt, and *n*- 1 readers are queued on mutex.
- When a writer executes signal (wrt), we may resume the execution of either all the waiting readers or a single waiting writer. The selection is made by the scheduler.

```
do {
   wait(wrt);
      . . .
   // writing is performed
      . . .
   signal(wrt);
} while (TRUE);
```

Figure 6.12 The structure of a writer 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);
```

Figure 6.13 The structure of a reader process.

- The readers-writers problem and its solutions have been generalized to provide **reader-writer locks**
- Acquiring a reader-writer lock requires specifying the **mode of the lock either *read* or *write*** access.
- When a process wishes only to read shared data, it requests the reader-writer lock in read mode; a process wishing to modify the shared data must request the lock in write mode.
- Multiple processes are permitted to concurrently acquire a reader-writer lock in read mode, but only one process may acquire the lock for writing, as exclusive access is required for writers.
- Reader-writer locks are most useful in the following situations:

- In applications **where it is easy to identify which processes only read shared data and which processes only write shared data.**
- In applications that have **more readers than writers**. This is because reader-writer locks generally require more overhead to establish than semaphores. The increased concurrency of allowing multiple readers compensates for the overhead involved in setting up the reader-writer lock.

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**
- When a philosopher thinks, she does not interact with her colleagues.
- From time to time, 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).

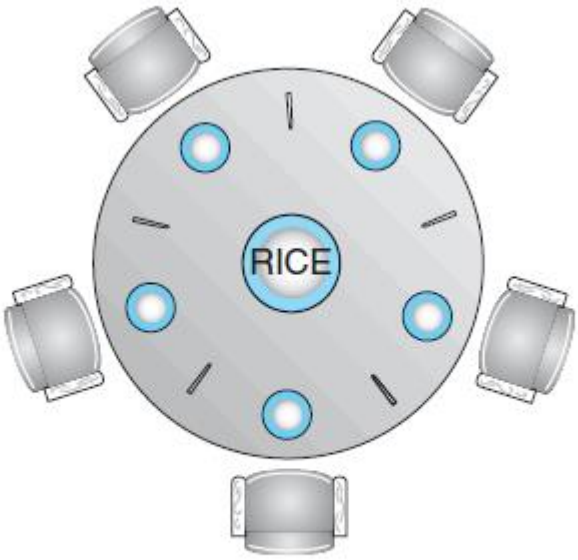

Figure 5.13 The situation of the dining philosophers.

- A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor.
- When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks.
- When she is finished eating, she puts down both of her chopsticks and starts thinking again.
- **This problem is an example of a large class of concurrency-control problems.**
- **It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.**
- 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* can be written as:

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

Figure 6.15 The structure of philosopher *i*.

- Although this solution guarantees that no two neighbors are eating simultaneously, it must be rejected because **it could create a deadlock**.
- Suppose that all five philosophers become hungry simultaneously and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.
- Several **possible remedies to the deadlock problem** are:

- 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 (to do this, she must pick them up in a critical section).
- Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick

- Any satisfactory solution to the dining-philosophers problem must eliminate the possibility that one of the philosophers will starve to death.
- **We can use monitors to solve the dining-philosophers problem that ensures freedom from deadlocks.**
- A deadlock-free solution does not necessarily eliminate the possibility of starvation.

Dining-Philosophers Solution Using Monitors

- This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.
- Three states for a philosopher.
  1. Thinking
  2. Hungry
  3. Eating

  which can be represented using an enum array

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).**
- We also need to declare **condition self[5];** in which philosopher i can delay herself when she is hungry but is unable to obtain the chopsticks she needs.
- The distribution of the chopsticks is controlled by the monitor
- Each philosopher, before starting to eat, must invoke the operation **pickup()**. This act may result in the suspension of the philosopher process.
- After the successful completion of this operation, the philosopher may eat.
- Following this, the philosopher invokes the **putdown()** operation.
- Thus, philosopher *i* must invoke the operations in the following sequence:

DiningPhilosophers.pickup(i);

//eat

DiningPhilosophers.putdown(i);

```
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((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;
    }
}
```

Figure 6.19 A monitor solution to the dining-philosopher problem.

- This solution ensures that no two neighbors are eating simultaneously and that no deadlocks will occur.

- However, that it is possible for a philosopher to starve to death.