Chapters 6 and 7: Process Synchronization Tools and Examples

1

Chapter Topics

- Background
- The Critical-Section Problem
- Peterson’s Solution
- Hardware instructions for synchronization
- Semaphores
- Classic Problems of Synchronization
- Monitors

2

Concurrent Processes

- Processes are concurrent if they exist at the same time.
- Independent processes cannot affect or be affected by the other processes executed in the system.
  - The state of an independent process is not dependent on other processes, and its execution is deterministic.
- Cooperating processes can affect or be affected by other processes in the system as they share data.
  - Concurrent access to shared data may result in data inconsistency (e.g., race condition on the shared variable *count* in the Producer-Consumer problem).
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes on the shared data.

Producer-Consumer Problem

- A producer process produces data items that are consumed by a consumer process.
- To allow the producer and consumer processes to run concurrently, a (logical ring) buffer of items in memory can be defined and shared by the producer and consumer processes.
- The producer and consumer must be synchronized:
  - The consumer has to wait if the buffer is empty.
  - The producer has to wait if the buffer is full.
- We can use a counter that keeps track of the number of items in the buffer. Initially, the counter is set to 0. It is incremented by the producer after it places a new item in the buffer and is decremented by the consumer after it consumes an item in the buffer.

Critical Section (CS)

- When a process is accessing shared data, the process is said to be in its critical section.
- A critical section is a segment of a code where statements are manipulating the shared data.
- Requirement: When one process is executing its critical section, no other process is to be allowed to execute its critical section.
  - This requirement is called *mutual exclusion*.

Requirements for a Solution to the Critical-Section Problem

1. **Mutual Exclusion**: If one process is executing its critical section, then other processes should not be allowed to execute their critical sections.
2. **Progress**: If no process is executing its critical section and there exist some processes that wish to enter their critical section, then the selection of a process 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.
   - A process could be starved if bounded waiting requirement is not satisfied.

General Structure of a Typical Cooperating Process

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

- Entry section is the section of code requesting the permission to enter its critical section.
- Exit section is executed to indicate that the process has left its critical section.
- Cooperating processes may share some variables to synchronize their actions.

Different Solutions to the Critical-Section Problem

- There are four different solutions we can use to implement the entry and exit sections of cooperating processes (for a given CS problem).

1. Software solution
2. h/w instruction
3. Semaphores
4. Monitor

Incorrect S/W Solution1 (for two processes)

- Two cooperating processes are denoted by P_i and P_j , or by P_0 and P_1.
- Shared variables:
  - **int turn**;
    initially **turn = i or j** (when P_i and P_j are two cooperating processes)
  - **turn = i** ⇒ process P_i can enter its critical section.

- Process P_i

```
while (true)  {
        while (turn != i) ; // busy waiting until turn== i
                critical section
        turn = j;
                remainder section
}
```

- Satisfies mutual exclusion, but not the progress requirement because it requires strict alternation of processes in the execution of the critical section.

Incorrect S/W Solution2 (for two processes)

- Shared variables:
  - **boolean flag[2]**;
    initially **flag[0] = flag[1] = false.**
  - **flag[i] = true** ⇒ P_i is ready to enter its critical section.

- Process P_i

```
while (true)  {
        flag[i] = true;
        while (flag[ j]);   // busy waiting until flag[j]==false
                critical section
        flag[i] = false;
                remainder section
}
```

- Satisfies mutual exclusion, but not the progress requirement: If Pi and Pj start their entry sections almost at the same time, such that Pi sets flag[i]=true and Pj sets flag[j]=true, then both processes will loop forever in their while statements.

Peterson's Solution (for two processes)

- A correct solution for two processes.
- The two processes share:

  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 each process is ready to enter the critical section: flag[i]==true implies that process P_i is ready.

11

Peterson's Solution (cont'd)

- Process P_i :

```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[ j] && turn == j); // busy waiting
            critical section
    flag[i] = false;
            remainder section
}
```

- Suppose that Pi is in its while statement, whereas Pj is in its CS. Once Pj exits its CS, it will reset flag[j] to false, and Pi can escape its while statement. Even if Pj restarts its entry section before Pi escapes its while statement, Pj will set turn to i so Pj cannot escape its own while statement because flag[i]==true and turn== i.

12

How to Check a Software Solution

- To check if a s/w solution is correct or not we can consider the following scenarios:

1. Two or more processes start their entry sections almost at the same time: Can only of them enter its CS?
2. One process just finishes it CS while some other processes are waiting in their entry sections: Can only one of the waiting processes enter its CS?
3. After a process finishes it CS and exit section, is it possible the process restarts its entry section and enters its CS again and again, while other processes are still waiting in their entry sections?

13

Solution to the Critical-Section Problem Using a Lock

- A binary lock (aka *mutex lock*) is a variable that can have only two states: *locked state* and *unlocked state*.
- When a process acquires a lock, the lock's state is changed from unlocked to locked, and other processes cannot acquire the lock until it is released (by the process that is holding the lock).
- Shared variable:

  boolean lock=false;
- Structure of each cooperating process:

```
while (true) {
    acquire lock
      critical section
    release lock
      remainder section
}
```

14

Two Main Issues Related to a Lock

- How to implement locking/unlocking operations (i.e., acquire/release operations on a lock), so that they are executed in atomic manner, and how to satisfy the bounded waiting requirement?
  - Locking/Unlocking can be implemented using:
    1. a h/w instruction or
    2. a semaphore
  - Bounded waiting requirement can be satisfied by
    1. using a waiting queue or
    2. when a process P_i finished its CS, it will scan the states of other processes in a cyclic order — P_{i+1}, P_{i+2}, …, P_{n-2}, P_{n-1}, P_o, P_1, P_2, …, P_{i-1} — and allow the first waiting process (in this order) to enter its CS.
       - Thus, any process waiting to enter its critical section will do so within n-1 turns when there are n cooperating processes.

15

Synchronization Using Hardware Instructions

- Modern machines provide special atomic (i.e., non-interruptable or nondivisible) hardware instructions:
  - test_and_set() reads the content of a memory word and also modifies it.
  - swap() exchanges the contents of two memory words.
  - compare_and_swap() reads the content of a memory word and then modifies it only if the word has certain value.
- Each *atomic instruction* is executed to the end without being interrupted (or interleaved with other instructions).
  - In other words, **context switching cannot happen in the middle of a hardware instruction.**
  - If two or more h/w instructions are issued by processes (on the same variable(s)), then they will be executed one at a time.

16

test_and_set() h/w Instruction

- Definition:

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

17

Solution Using test_and_set()

- Shared variable:
  boolean lock=false;
- Process P_i:

```
while (true) {
    while ( test_and_set (&lock )) ;   /* busy waiting */

            /* critical section */

    lock = false;

            /* remainder section */
}
```

- Note: mutual exclusion and progress requirements are satisfied, but not the bounded waiting requirement.

18

swap() h/w Instruction

- Definition:

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

19

Solution Using swap()

- Shared variable:

  boolean lock=false;
- Each process has a local boolean variable key.
- Process P_i

```
while (true)  {
        boolean key = true;
        while ( key == true) swap (&lock, &key ); /* busy
waiting */

                /* critical section */

        lock = false;

                /* remainder section */
}
```

Note: mutual exclusion and progress requirements are satisfied, but not the bounded waiting requirement.

20

compare_and_swap() h/w Instruction

- Definition:

```
int compare_and_swap (int *value, int expected,
      int new_value)
      {
            int temp = *value;
            if (*value == expected)
                *value = new_value;
            return temp;
       }
```

21

Solution Using compare_and_swap()

- Shared variable:
  int lock=0;
- Process P_i

```
    while (true)  {
        while (compare_and_swap(&lock, 0, 1) ==1); /*  busy
waiting */

                    /* critical section */

               lock = 0;

                    /* remainder section */
      }
```

- Note: mutual exclusion and progress requirements are satisfied, but not the bounded waiting requirement.

22

Bounded-waiting Solution with test_and_set()

- Shared variables:

```
boolean waiting[n];  /* initially false */
boolean lock = false;
```

- Process P_i

```
while (true)  {
        waiting[i] = true;
        boolean key = true; /* in order to execute test_and_set(&lock) in the first
  iteration */
        while (waiting[i] && key)
                key = test_and_set(&lock);  /*  busy waiting */
        waiting[i] = false;

                /* critical section */
        j = (i + 1) % n; /* to find the 1st waiting process in a cyclic order */
        while ((j != i) && !waiting[ j])
                j = (j + 1) % n;
        if (j == i)
                lock = false; // no other process is currently waiting
        else
                waiting[ j] = false;  // to let  Pj enter its critical section
                /* remainder section */
  }
```


Semaphore

- Original definition of a semaphore:

  A semaphore *S* is an integer variable that, apart from initialization, can be accessed only via two atomic (i.e., indivisible) operations: wait(S) and signal(S), which were originally called P() and V() operations, respectively.
- Definitions of wait(S) and signal(S) on an original semaphore S:

```
semaphore S;

   wait (S) {
       while (S <= 0); // no-op (i.e., busy waiting until S > 0)
       S--;
   }

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

Semaphore (cont'd)

- A binary semaphore is a semaphore whose value could be only 0 or 1.
  – Typically used to provide the mutual exclusion.

Shared variable:

semaphore mutex =1; // initial value of mutex is set to 1

Process P_i:

```
while (true) {
    wait(mutex);
        critical section
    signal(mutex);
        remainder section
}
```

```
wait (mutex) {
    while (mutex <= 0);
        // busy waiting
    mutex--;
}

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

- A counting semaphore (aka general semaphore) is a semaphore whose value could be any integer value (typically within a range).
  – Used to count the number of available resources (that are identical).
  – If a counting semaphore is used to provide the mutual exclusion, it should be initialized to 1.

25

Semaphore Implementation with No Busy Waiting

- The wait() operation on the original semaphore (i.e., just an integer value) has the busy waiting problem:

```
wait (mutex) {
    while (mutex <= 0);  // busy waiting
    mutex--;
}
```

  – When a process is in its critical section, any other process that tries to enter its critical section must check the value of **mutex** semaphore repeatedly until it becomes positive.
  – Busy waiting wastes CPU cycles.
- Solution: semaphore with an associated waiting queue for processes:

```
typedef struct {
    int value;
    struct process *list;  // a list of processes
} semaphore;
```

26

Semaphore Implementation with No Busy Waiting (cont'd)

- Each semaphore has an integer value and a list of processes (i.e., a queue in which processes can wait).
- If the integer value of a semaphore is not positive when a process executes a wait() operation on it, so that the wait() operation cannot be finished, the process must wait on the semaphore; that means, the process is added to the list of processes associated with the semaphore and blocked (i.e., suspended).
- A signal() operation increments the integer value of the semaphore by 1, then removes one process from the list of waiting processes (if at least one waiting process exists in the list) and awakens that process.
- Here we assume two simple operations (i.e., system calls) are available:
  - block() suspends the process that invokes it.
  - wakeup(P) resumes the execution of a blocked process P.

27

Semaphore Operations with No Busy Waiting

- Definition of wait() on a general semaphore with a queue:

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

- Definition of signal() on a general semaphore with a queue:

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

28

Semaphore Operations with No Busy Waiting (cont'd)

- These wait() and signal() operations are for a general semaphore with an associated waiting queue. However, a general semaphore can be used for mutual exclusion by setting its initial value to 1.
- In waiting(semaphore S), if S.value-- is to be performed later (i.e., after a waiting process is waken up), the waken-up process needs exclusive access to S again. S.value-- should be executed to complete wait(S), so executing it first is okay.
  - If S.value is negative after S.value--, there are |S.value| processes currently blocked on S.
- In signal(semaphore S), if S.value is positive after S.value++, there is no process blocked on S. Otherwise, |S.value| +1 processes are currently blocked on S.

29

Critical Section Aspect of Semaphores

- wait() and signal() operations on a semaphore must be executed atomically, such that no two processes can access the same semaphore and change its value at the same time.
  - On each semaphore, only one wait() or signal() operation should be allowed to access and change the value of the semaphore at a time, no matter how many processes invoke their wait() and signal() operations almost at the same time, in order to avoid the race condition.
- In the uniprocessor system, we can inhibit (i.e., disable) the interrupt during the time a wait() or a signal() operation is executed, so that context switching will not happen. We can also use test_and_set() or compare_and_swap() to implement wait() and signal(), so that only one process can access a semaphore at any time.
- In a shared-memory multiprocessor system, if a h/w instruction, such as test_and_set() or compare_and_swap(), is not supported, we can employ any of the correct software solutions (for *N* processes) proposed for the critical section problem, such as:
  - Eisenberg and McGuire's Algorithm
  - Bakery Algorithm

30

Implementation of wait(S) and signal(S) Using test_and_set()

- Shared variable:
  boolean lock=false; // a lock for semaphore S
- Implementation of wait(semaphore S):

```
while (test_and_set(&lock)); /* busy waiting to acquire the
                                 lock on S */
S.value--;
if (S.value < 0)
        {  add this process to S.list;
           lock = false;
           block();
         }
else lock = false;
```

31

Implementation of wait(S) and signal(S) Using test_and_set() (cont'd)

- Implementation of signal(semaphore S):

```
while (test_and_set(&lock)); /* busy waiting to acquire
                                 the lock on S */
S.value++;
if (S.value <= 0)
        { lock = false;
          remove a process P from S.list;
          wakeup(P);
        }
else lock = false;
```

32

Use of a Semaphore for Counting Purpose

- An example synchronization requirement: There are 5 identical resources shared by a number of processes: Each process uses one of the resources at a time, and each resource should be used by only one process at a time.
- Shared variable:

```
semaphore S = 5; /* total number of identical
                    resources is 5 */
```

- Process P_i:

```
while (true)  {
     wait (S);
        use one of the available resources
     signal (S);
        remainder section
}
```

33

Semaphore as a General Synchronization Tool

- An example synchronization requirement:

  Execute a statement *B* in process P_j only after a statement *A* is executed in process P_i.

  – Shared variable

  semaphore flag=**0**;

  – Codes:

process P_i :	process P_j :
⋮	⋮
A	wait(flag);
signal(flag) ;	*B*

34

Exercise Question: Simulating a Counting Semaphore **S** by Using an Integer Variable and Two Binary Semaphores

- Shared variables:

```
binary-semaphore S1, S2;
int S;   /* S works like the integer value of a
            general semaphore */
```

- Initialization:

```
S1 = 1;  /* to provide exclusive access (i.e., a lock)
            to S */
S2 = 0;  /* to block a process when S is negative
            (after S--) */
S = initial_value_of_S;
```

35

Simulating a Counting Semaphore **S** Using Two Binary Semaphores (cont'd)

- Implementation of wait(int **S**):

```
wait(S1); /* to access S exclusively*/
S--;
if (S < 0) {  /* some other process is in its CS, or there is no
                 resource available for this process */
        signal(S1); /* to release S */
        wait(S2);  /* to block the process on S2 */
      }
else signal(S1); /* to release S */
```

- Implementation of signal(int **S**):

```
wait(S1); /* to access S exclusively*/
S++;
if (S <= 0)  { /* at least one process is blocked on S2 */
        signal(S1); /* to release S */
        signal(S2); /* to wake up a process currently blocked
on S2 */
      }
else signal(S1); /* to release S */
```

36

Different Uses of a Semaphore

1. To provide mutual exclusion, by initializing the semaphore to 1.
2. For counting purpose, by initializing the semaphore to a certain value): Only one process can increment or decrement the value of the semaphore by executing wait() or signal() on the semaphore, respectively.
3. To provide a waiting queue for processes (i.e., to block processes until waken up one by one), by initializing the semaphore to 0.
4. To satisfy a problem-specific synchronization requirement between processes.

Classical Problems of Synchronization

- Bounded-Buffer Problem (aka Producer-Consumer Problem)
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded Buffer Problem (aka Producer-Consumer Problem)

- A producer process produces data items that are consumed by a consumer process.
- To allow the producer and consumer processes to run concurrently, a (logical ring) buffer of items in memory can be defined and shared by the producer and consumer processes.
- The producer and consumer must be synchronized:
  - The consumer has to wait if the buffer is empty.
  - The producer has to wait if the buffer is full.
- We can use a counter that keeps track of the number of items in the buffer.
  - Initially, the counter is set to 0.
  - It is incremented by the producer after it places a new item in the buffer and is decremented by the consumer after it consumes an item in the buffer.

39

Producer-Consumer Problem (cont'd)

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0; // points to the next free buffer element
int out = 0; // points to the first full buffer element
int count = 0; // counts the number of data items in
  the buffer
```

40

Producer

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

41

Consumer

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

42

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}
- Note: This race condition is called a lost-update problem (in DB), meaning that the effect of an update operation is lost.

43

A Semaphore Solution for the Bounded-Buffer Problem

- A logical ring buffer has *N* buffer elements, each of which can hold one item.
- A (binary) semaphore *mutex* provides exclusive access to the buffer for a producer or a consumer process, and its value is initialized 1.
- A counting semaphore *full* counts the number of full buffer elements (i.e., the number of buffer elements filled with unconsumed items) and its value is initialized 0.
- A counting semaphore *empty* counts the number of empty buffer elements (i.e., the number of buffer elements not filled with unconsumed items) and its value is initialized to *N*.
- Counting semaphores *full* and *empty* are incremented/decremented by 1 exclusively via wait()/signal() operation.
- int in, out; // pointers to buffer elements and initialized to 0.
  - mutex also provides exclusive access to *in* and *out* pointers, which is required when there are multiple producers and consumers.

44

Bounded Buffer Problem (cont'd)

- The structure of the Producer process:

```
while (true)  {

   /* produce an item in next_produced */

   wait (empty); /* to wait until there is an empty buffer element */
   wait (mutex); /* to access buffer[in] and in exclusively */

   /* add the item in next_produced to buffer[in] */

   in = (in +1) % N ;
   signal (mutex); /* to release buffer[in] and in */

   signal (full); /* to increment the number of filled buffer
                         elements */
}
```

45

Bounded Buffer Problem (cont'd)

- The structure of the Consumer process:

```
while (true)  {

     wait (full); /* to wait until there is a filled buffer element */
     wait (mutex); /* to access buffer[out] and out exclusively */

     /* remove an item from buffer[out] to next_consumed */

     out = (out +1) % N;
     signal (mutex); /* to release buffer[out] and out */

     signal (empty); /* to increment the number of empty buffer
                           elements */

     /* consume the item in next_consumed */
  }
```

46

Bounded Buffer Problem (cont'd)

- This implementation allows multiple producers and multiple consumers, because not only each buffer element and counter but also each pointer (i.e., *in* and *out*) is accessed exclusively by each producer and consumer.
- If we use an integer variable *count*, instead of two counting semaphores (*full* and *empty*), in order to count the number of data items in the buffer, then we need two semaphores — one for exclusive access to *count* and another for providing a waiting queue for the consumer (in case count==0) and the producer (in case count==*N*).
  - This similar to the two binary semaphores used for simulating a counting semaphore.
  - So, there is no advantage of using an integer variable *count*.

47

Readers-Writers Problem

- A data set is shared among a number of concurrent processes of two types:
  - Readers only read the data set: they do not perform any update (i.e., write operation) on the data set.
  - Writers can perform both read and write operations on the data set.
- Constraint: Multiple readers are allowed to read the data set at the same time, because each reader does not change the data set; but each writer must access the shared data set exclusively (to avoid a race condition).
  - A reader should wait until there is no active writer.
    - If we have an active write and an active reader at the same time, we may have the incorrect-summary problem.
  - When there is one or more active readers, an incoming reader can join the active reader(s).
  - A writer should wait until there is no active writer and no active reader.
    - If we have more one active writer, we may have the lost-update problem.
    - Thus, a waiting writer could starve if there is an active reader and new readers arrive continuously (and become active).

48

Readers-Writers Problem (cont'd)

- Shared data and variables:
  - Data set.
  - Semaphore *rw_mutex* is used to ensure the data set is accessed only by a writer (by blocking incoming writers and readers) or by one or multiple readers (by blocking incoming writers), and its value is initialized to 1.
  - Integer *read_count* counts the number of (active and waiting) readers, and is initialized to 0.
    - read_count is incremented by each incoming reader, and is decremented by each outgoing reader.
  - Semaphore *mutex* is used to ensure exclusive access to read_count for each reader, and its value is initialized to 1.

Readers-Writers Problem (cont'd)

- The structure of a **Writer process**:

```
while (true)  {

        wait (rw_mutex) ; /* to wait if there is an active writer
                                     or active reader(s) */
         /* perform writing on the data set */

         signal (rw_mutex) ; /* to wake up a waiting writer
                                     or an waiting reader (if exists) */
 }
```

- A writer doesn't need to check if read_count is 0 because rw_mutex is set to 1 only when there is no active/waiting reader (i.e., read_count == 0) as well as no active writer.

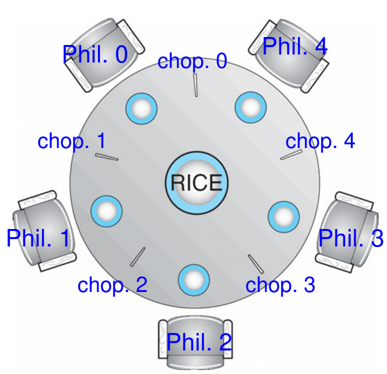
Readers-Writers Problem (cont'd)

- The structure of a **Reader process:**

```
while (true)  {
    wait (mutex) ; // to access read_count exclusively
    read_count++ ;
    if (read_count == 1) wait (rw_mutex) ; /* to wait if there
                                    is an active writer */
    signal (mutex) ; // to release read_count

      /* perform reading on the data set */

    wait (mutex) ; // to access read_count exclusively
    read_count-- ;
    if (read_count == 0) signal (rw_mutex) ; /* to wake up
                              a waiting writer (if exists) */
    signal (mutex) ; // to release read_count
}
```

51

Dining-Philosophers Problem



- Constraints: Each philosopher needs two chopsticks to eat, but can pick up one chopstick at a time.
- A simple solution is to represent (the availability of) each chopstick by a semaphore:
  - to grab a chopstick, execute wait() on it; and to release it, execute signal() on it.

52

Dining-Philosophers Problem (cont'd)

- Shared variable: (for 5 Philosophers)

  semaphore chopstick [5]; /* and each chopstick[i] is initialized to 1 */

- The structure of Philosopher i:

```
while (true) {
     wait ( chopstick[i] ); /* to grab the left chopstick */
     wait ( chopstick[(i + 1) % 5] ); /* to grab the right chopstick */

          /* eat for a while*/

     signal ( chopstick[i] ); /* to release the left chopstick */
     signal ( chopstick[(i + 1) % 5] ); /* to release the right
                                              chopstick */
          /* think for a while */
 }
```

53

Dining-Philosophers Problem (cont'd)

- A deadlock occurs if all five philosophers become hungry and each grabs his/her left chopstick.
- Possible remedies:
  1. Allow at most four philosophers (out of five) to be sitting at the same time.
  2. Allow a philosopher to pick up a his/her chopstick only if both chopsticks are available (i.e., both left and right neighbors are not eating).
     - Checking if both chopsticks are available should be implemented as a critical section so that his/her neighbors cannot check either one of those two chopsticks at the same time. It is like checking two door locks at the same time by each person.
  3. Asymmetric solution: Every odd-numbered philosopher picks up his/her left chopstick then his/her right chopstick, whereas every even-numbered philosopher picks up his/her right chopstick then his/her left chopstick.

54

Deadlock and Starvation

- Deadlock: two or more processes are waiting permanently for an event that can be caused by one of those waiting processes.
- Let S and Q be two shared semaphores initialized to 1.

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

- Starvation: indefinite waiting of some process.
  - For example, a process may not be waken up for a long time from the waiting queue of a semaphore in which it is blocked.
  - Associating a FIFO queue with each semaphore can avoid this starvation because the waiting processes would be waken up in the order they were placed in the queue.

55

Problems with Semaphores

- Incorrect use of semaphore operations by a process as follows:
  - signal (mutex);
    CS
    wait (mutex);
  - wait (mutex);
    CS
    wait (mutex);
  - Omitting wait (mutex) or signal (mutex), or both.

56

Monitor

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization.
- A monitor type is an Abstract Data Type (ADT) which encapsulates private data with public methods to operate on that data.
- The monitor ensures mutual exclusion by itself: only one process at a time can be active within the monitor; i.e., executing a function (or a procedure) defined within the monitor.

57

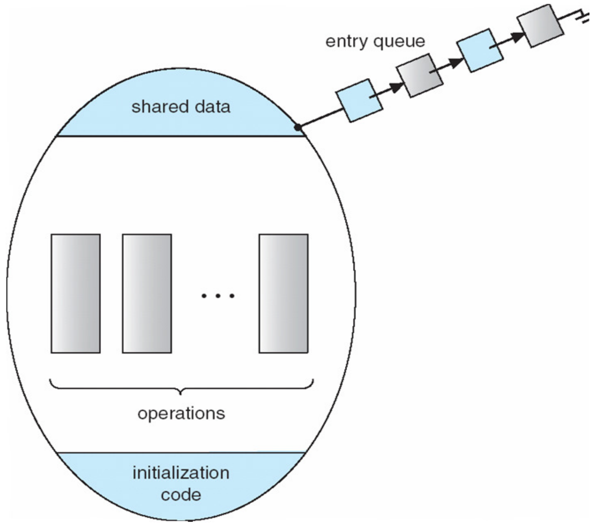
Monitor (cont'd)

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

  function P1 (…) { … }
        .
         .
         .
  function Pn (…) { … }

  Initialization code () { … }
    /* initialize shared variables */
}
```

- Shared variables of cooperating processes are declared inside the monitor and initialized.
- Entry section and exit section of each cooperating process are implemented as functions (or procedures) inside a monitor.

58

Schematic View of a Monitor



59

Condition Variables

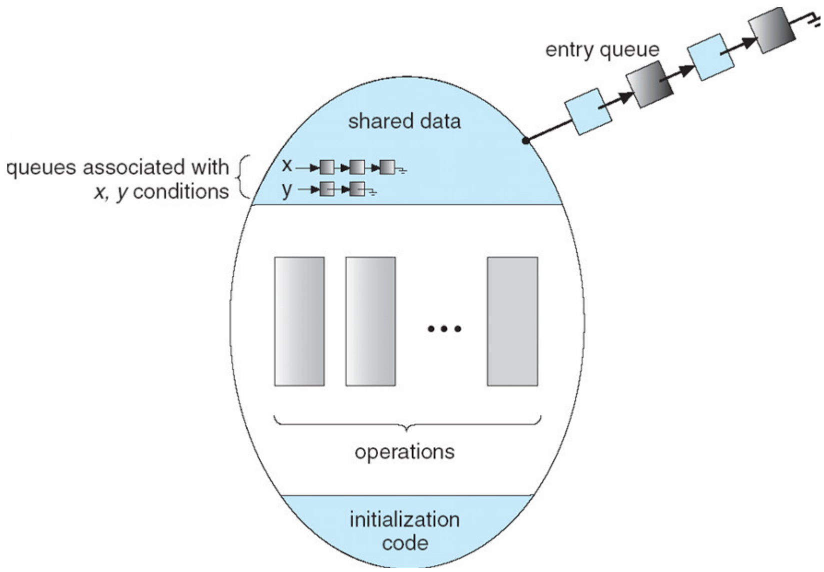
- For problem-specific synchronization, we can define variables of type *condition* inside the monitor. For example,

  condition x, y;

- The only operations that can be invoked on each condition variable, say x, are x.wait() and x.signal():
  - x.wait(): a process that invokes this operation is suspended until another process invokes x.signal().
  - x.signal(): resumes exactly one suspended process (if any) that invoked x.wait(). It has no effect if no process is suspended on x.

60

Monitor with Condition Variables



A Monitor to Allocate a Single Resource to a Process

```
monitor ResourceAllocator1
{
   boolean busy;
   condition x;

   void acquire() {
          if (busy) x.wait();
          busy = TRUE;
   }
   void release() {
          busy = FALSE;
          x.signal();
   }
```

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

Note: Each process invokes the operations acquire() and release() in the following sequence:

```
ra.acquire();
  // access the resource;
ra.release();
```

where ra is an instance of the monitor ResourceAllocator1.

Monitor Implementation of the Bounded-Buffer problem

```
monitor bounded-buffer
{
    item  buffer[n];
    int  in, out, count;
    condition  x, y;

    void append (item nextp) {

        if (count==n) x.wait();
        buffer[in] = nextp;
        in = (in + 1) % n;
        count ++;
        y.signal();
    }
```

- Note: append(item nextp) is invoked by the producer; and readout() is invoked by the consumer.

```
item readout () {

    if (count==0) y.wait();
    item nextc = buffer[out];
    out = (out+1) % n;
    count--;
    x.signal();
    return nextc;

}

 initialization_code() {
        in = 0;
        out = 0;
        count = 0;
 }
}
```

53

Solution to Dining Philosophers

```
monitor DiningPhilosophers
 {
  enum {THINKING, HUNGRY, EATING} state[5] ;
  condition self[5];

  void pickup (int i) {
       state[i] = HUNGRY;
       test(i);  /* to test if Philosopher i can eat*/
       if (state[i] != EATING) self[i].wait();
  }

  void putdown (int i) {
       state[i] = THINKING;
        test((i + 4) % 5);  /* to test if left-side neighbor is hungry and
  can eat */
        test((i + 1) % 5);  /* to test if right-side neighbor is hungry and
  can eat */
  }
```

64

Solution to Dining Philosophers (cont'd)

```
void test (int i) {
      if ( (state[(i + 4) % 5] != EATING) &&
          (state[i] == HUNGRY) &&
          (state[(i + 1) % 5] != EATING) ) {
               state[i] = EATING ;
               self[i].signal() ; /* to wake up Philosopher i by a
                                    neighbor who just finished eating */
      }
 }

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

65

Solution to Dining Philosophers (cont'd)

- Each philosopher i invokes the operations pickup() and putdown() in the following sequence, where dp is an instance of the monitor DiningPhilosophers:

  ```
  dp.pickup (int i);
      eat;
  dp.putdown (int i);
  ```

- A hungry philosopher i can set the variable state[i]=EATING only if his/her two neighbors are not eating.
- Philosopher i can suspend himself/herself (by executing self[i].wait()) when he/she is hungry but unable to obtain the two needed chopsticks.
- test(int i) is invoked by Philosopher i when he/she is hungry, and by his/her neighbor who just finished eating.
  - In the latter case, if Philosopher i is hungry and he/she can eat now (as his/her another neighbor is not eating), then he/she is woken up via self[i].signal().

66

Monitor Implementation Using Semaphores

- For each monitor, a semaphore mutex (initialized to 1) is provided automatically so that at most one cooperating process can be active inside the monitor.
  - wait(mutex) must be executed automatically by each process before entering the monitor (i.e., invoking a function within the monitor).
  - signal(mutex) must be executed automatically by a process (to be suspended inside the monitor or leaving the monitor) to allow another process to enter the monitor.

67

Monitor Implementation Using Semaphores

- Suppose that there is a process *Q* suspended on a condition variable x when the x.signal() operation is invoked by a process *P* that is active in the monitor.
  - As the suspended process *Q* is allowed to resume its execution, the signaling process *P* must wait: this scheme is called ***signal-and-wait***.
    - If P is allowed to continue its execution (which is called ***signal-and-continue***), both *P* and *Q* would be active simultaneously within the monitor, which is not allowed by the definition of monitor and may violate the mutual exclusion if *P* and *Q* try to access the same shared variable or data.
  - When the ***signal-and-wait*** scheme is used, since a signaling process must wait, an additional semaphore ***next*** (initialized to 0) is provided automatically inside the monitor, on which every signaling process would be suspended.
    - *P* is suspended on *next* by executing wait(*next*) as a part of x.signal().
    - *P* is suspended on *next* until either *Q* leaves the monitor or *Q* starts waiting for another condition (e.g., by executing y.wait()).
    - An integer variable next-count is provided automatically to count the number of processes suspended on *next*.

68

Monitor Implementation Using Semaphores (cont'd)

- Variables

```
semaphore mutex;  // initially 1
semaphore next;   // initially 0
int next-count = 0;
```

- Each external call of a function *F* within a monitor will be automatically replaced by:

```
wait(mutex);

body of F;

if (next_count > 0)
    signal(next); /* to wake up a process suspended on next */
else
    signal(mutex); /* to wake up a process waiting on mutex
                      (if any exists) */
```

- Thus, a process waiting (i.e., suspended) on the semaphore *next* (inside the monitor) has a higher priority than the processes waiting on the semaphore *mutex* (outside the monitor).

69

Monitor Implementation Using Semaphores (cont'd)

- Each condition variable x is implemented using:

```
semaphore  x_sem; // initialized to 0
int  x_count = 0;
```

x-count stores the number of processes waiting on x_sem.

- The operation x.wait() can be implemented as:

```
x_count++;
if (next_count > 0)
    signal(next); /* to wake up a process suspended on next
                     (if any exists) */
else
    signal(mutex); /* to wake up a process suspended on mutex
                      (if any exists) */
wait(x_sem); /* to be suspended on x_sem */
x_count--;
```

- We don't need mutual exclusion on x_count and next_count because only one process can execute x.wait at a time (as the process is the only one being active inside the monitor).

70

Monitor Implementation Using Semaphores (cont'd)

- The operation x.signal() can be implemented as:

```
if (x_count > 0) {
      next_count++;
      signal(x_sem); /* to wake up a process suspended on x_sem */
      wait(next); /* to be suspended on next */
      next_count--;
 }
```

- next_count++ is executed before signal(x_sem); otherwise, both the waken-up process and the signaling process need to be active at the same time (inside the monitor).

71

Conditional-wait operation x.wait(*C*)

- Resumption order of suspended processes on a condition variable:

  If several processes are suspended on a condition variable x, and an x.signal() operation is executed by some other process, then how can we select a suspended process based on some priority values of the processes, rather than FIFO?
- *Conditional-wait operation* x.wait(*C*) can be used, where *C* is an integer expression that is evaluated when the Conditional-wait operation is executed on x.
- The value of *C*, which is called a priority number, is then stored with the id of the process that is suspended on x.
- When x.signal() is executed, the process with the smallest associated priority number would be waken up.
- This Conditional-wait operation is to resume a process based on some priority of the processes.

72

Example of Using the Conditional-wait Operation

- A single resource allocation based on the (estimated) usage time:
  - Each process, when requesting a resource, specifies the time it plans to use the resource.
  - When the resource becomes available, it will be allocated to the process (among the waiting processes) that has requested the minimum usage time.

73

A Monitor to Allocate a Single Resource

```
monitor ResourceAllocator2
{
   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;
   }
}
```

Note: Each process invokes the operations acquire(int time) and release() in the following sequence:

```
      ra.acquire(int time);
        access the resource;
      ra.release();
```

where ra is an instance of the monitor ResourceAllocator2.

74