Operating Systems
COMP2006

Process Synchronization
Lecture 3

Process Synchronization

References:

Silberschatz, Galvin, and Gagne, *Operating System Concepts*, Chapter 5.

Topics:

- Concurrent processes.
- The critical section problem.
- Solutions for critical-section problems using semaphores, etc.
- Classical problems of synchronization.
- Monitors

Background

- A *cooperating* process can affect or be affected by the other processes executing in the system.
  - an *independent* process cannot affect or be affected by other processes.
- Cooperating processes may:
  - Share a logical address space (i.e., both code and data)
  - Share data through files
- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

Example: The producer/consumer problem

Bounded buffers (Circular list).

// in = out when the buffer is empty.

in: next free buffer;

out: first full buffer;

Shared data

Type *item* = … ;

```
var     buffer: array[0..n-1] of item;
        nextp, nextc: item;
        in, out: 0 .. n-1;
        counter: 0 .. n;
        in, out, counter  = 0;
```

Producer process

```
repeat
    produce an item in nextp;
        …
    /* busy waiting */
    while counter = n do no-op;

    buffer[in] = nextp;
    in = (in+1) mod n;

    counter := counter + 1;
until false;
```

Consumer process

```
repeat
        /* busy waiting */
        while counter = 0 do no-op;

        nextc = buffer[out];
        out = (out+1) mod n;
        counter = counter –1;
        consume item in nextc;
        …
until false;
```

Example (cont.)

- Each of the following statements must be executed *atomically* → one uninterruptible unit.

  counter = *counter* + 1; and *counter* = *counter* - 1;
- *Atomic* operation means an operation that *completes entirely without interruption*.

Example: Assume initially *counter* = 5;

producer executes *counter* = *counter* +1; and

consumer executes *counter* –1;

- The correct result for *counter* should be 5
  - if the producer and consumer executes separately (not concurrently); or
  - If each of the instruction is executed atomically
  - Otherwise, *counter* can be 4, 5, or 6.
- Statements *counter* = *counter* + 1 and *counter* = *counter* - 1 may be implemented in machine language as:

  $Register_1 = counter;$

  $Register_1 = Register_1 + 1;$

  $counter = Register_1;$

  $Register_1 = counter;$

  $Register_1 = Register_1 - 1;$

  $counter = Register_1;$

Example (cont.)

One example of interleaving of machine instruction:

T_0: *producer* **execute**	$register_1 = counter$	$\{register_1 = 5\}$
T_1: *producer* **execute**	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2: *consumer* **execute**	$register_2 = counter$	$\{register_2 = 5\}$
T_3: *consumer* **execute**	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4: *producer* **execute**	$counter = register_1$	$\{counter = 6\}$
T_5: *consumer* **execute**	$counter = register_2$	$\{counter = 4\}$

→ We get *counter* = 4 (incorrect) because we allowed both processes to manipulate *counter* concurrently.

Race condition

- *Race condition* is a situation where several processes access and manipulate the same data concurrently.
  - The outcome of the execution depends on particular order in which the access takes place.
- In order to prevent race condition on *counter*, we need to ensure that only one process at a time can be manipulating *counter*
  - We need some form of *process synchronization*.

The critical section problem

- n processes $\{P_0, P_1, \ldots, P_{n\text{-}1}\}$ all competing to use some shared data.
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- The *critical section problem* is to make sure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

Structure of process P_i

Repeat

Entry section.

Critical section.

Exit section.

Remainder section.

Until *false;*

Solution to the critical-section problem

Must satisfy these three requirements:

1. **Mutual Exclusion:** If a process is in the critical section then no other processes can be executing in their critical sections

   or No processes may be simultaneously inside their critical sections.

2. **Progress:** If no process is in its critical section and there exists some processes that wish to enter their critical sections, then only processes not in their remainder section can participate in the decision as to which one enters its critical section next (cannot postpone indefinitely)

   or No process running outside its critical section may block other processes from entering their critical sections.

3. **Bounded Waiting:** There must exist a bound on the number of times that other processes can enter their critical section after a process has made a request to enter its critical section and before that request is granted

   or No process should have to wait forever to enter its critical section.

Assumption:

- Each process executes at non-zero speed.
- No assumption concerning *relative* speed of the *n* processes or the number of CPUs

Solution (cont.)

Simplest solution:

Each process *disables* all interrupts just *after entering* its critical section and *re-enables* them just *before leaving* it.

- Not wise, because enabling/disabling interrupt is a privileged instruction.

Solution in kernel mode:

- **Preemptive kernel:** a process can be preempted while running in the kernel mode
  - Otherwise, the kernel is **nonpreemptive kernel**: allows the process to run until it exits kernel mode, blocks, or voluntarily yields CPU.
- Nonpreemptive kernel is free from race conditions on kernel data structure
  - However preemptive kernel is more responsive and suitable for real time system.

Solution for two processes, P_0 and P_1
Software-based

ALGORITHM 1

var *turn*: (0 .. 1); // Initially turn = 0; *turn* = *i* means P_i can enter its CS

Process P_i
repeat
  while $turn \neq i$ **do** *no-op*;
    Critical Section
  $turn = j$;
    Remainder Section
until *false*;

Process P_j
repeat
  while $turn \neq j$ **do** *no-op*;
    Critical Section
  $turn = i$;
    Remainder Section
until *false*;

- Satisfies mutual exclusion, but not progress requirement.
  - If $turn = 0$, P_1 cannot enter its CS even though P_0 is in its RS.
  - Taking turn is not good when one process is slower than other.
- *Busy waiting*: continuously testing a variable waiting for some value to appear
  - not good since it wastes CPU time

Solution for two processes (cont.)

Software-based

ALGORITHM 2

// Initially *flag*[0] = *flag*[1] = *false*; *flag[i]* = *true* means P_i wants to enter its CS

var *flag*: **array** [0 .. 1] **of** *boolean;*

Process P_i

```
repeat
    flag[i] = true;
    while flag[j] do no-op;
        Critical Section;
    flag[i] = false;
        Remainder Section;
until false;
```

Process P_j

```
repeat
    flag[j] = true;
    while flag[i] do no-op;
        Critical Section;
    flag[j] = false;
        Remainder Section;
until false;
```

- Satisfy mutual exclusion, but violates the progress requirement:

  T_0: P_0 sets flag[0] = true.

  T_1: P_1 sets flag[1] = true.

  → P_0 and P_1 are looping in their respective **while.**

Solution for two processes (cont.)

Software-based

// **Peterson's solution**: Combine shared variables of Algorithms 1 and 2

Process P_i

```
repeat
    flag[i] = true;
    turn = j;
    while (flag[j] and turn = j) do no-op;
          critical section
    flag[i] = false;
          remainder section
until false;
```

Process P_j

```
repeat
    flag[j] = true;
    turn = i;
    while (flag[i] and turn =i) do no-op;
          critical section
    flag[j] = false;
          remainder section
until false;
```

- Solves the critical-section problem for two processes.
  - It meets all the three requirements
- Proof: need to show that:
  - Mutual exclusion is preserved.
  - The progress requirement is satisfied.
  - The bounded waiting time requirement is met.
- For detailed proof, read the textbook.

Bakery Algorithm

- The solution to the critical section problem for n processes by Leslie Lamport
- Before entering its critical section, each process receives a number.
  - The holder of the smallest number enters the critical section.
  - If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first.
- Notation (**ticket#**, process **id#**)
  - $(a, b) < (c, d)$ if $a < c$ or if $a = c$ and $b < d$.
  - $max\ (a_0, \ldots, a_{n-1})$ is a number k such that $k \geq a_i$ for $i = 0, \ldots, n-1$.
- shared data
  - **var** *choosing*: **array**[0..*n*-1] of *boolean*;
  - *number*: **array** [0 .. *n*-1] of *integer*;
- data structures are initialised to *false* and 0, respectively

Bakery Algorithm (cont.)

Process P_i

```
repeat
    choosing[i] = true;
    number[i] = max (number[0], number[1], …, number[n-1]) +1;
    choosing[i] = false;
    for j = 0 to n-1 do
    begin
        while choosing[j] do no-op;
        while number[j] ≠ 0 and (number[j], j) < (number[i], i) do no-op;
    end;
            critical section
    number[i] = 0;
            remainder section
until false
```

Synchronization Hardware

- There is no guarantee that the software-based solution will work correctly in all computer architectures.
- The simple solution to critical section problem: disable interrupt while a shared variable is being modified.
  - This solution is not feasible in multiprocessor. Why?
- Use special hardware instructions such as *Test-and-Set* and *Swap*.
  - *Test-and-set* or *Swap* is an atomic instruction: it can not be interrupted until it completes its execution

// Test and set the content of a word *atomically*

function *Test-and-Set* (**var** *boolean*: *target*)
begin
    Test-and-Set = *target*;
    target = *true*;
end;

// Swapping instruction is done *atomically*

procedure *Swap* (**var** *boolean*: *a*, *b*)
var *boolean*: *temp*;
begin
    temp = *a*;
    a = *b*;
    b = *temp*;
end;

How to use them?

Mutual Exclusion with *Test-and-Set*

var *boolean*: *lock*; *lock* is a shared variable, initially set to *false*.

Repeat // *Process* P_i
    while *Test-and-Set* (*lock*) **do** *no-op*;
        Critical Section
    lock = *false*;
        Remainder Section
until *false*;

Repeat // *Process* P_j
    while *Test-and-Set* (*lock*) **do** *no-op*;
        Critical Section
    lock = *false*;
        Remainder Section
until *false*;

Mutual Exclusion with *Swap*

Repeat // *Process* P_i
    key = *true*;
    repeat
        Swap (*lock*, *key*);
    until *key* = *false*;
        Critical section
    lock = *false*;
        Remainder section
until *false*;

Repeat // *Process* P_j
    key = *true*;
    repeat
        Swap (*lock*, *key*);
    until *key* = *false*;
        Critical section
    lock = *false*;
        Remainder section
until *false*;

- Both do not satisfy the bounded waiting requirement.

Correct solution with Test-and-set

Shared data: **var** *waiting*: **array**[0..*n*-1] **of** *boolean*; *lock*: *boolean*; //All initialized to *false*

Process P_i

```
var j: 0..n-1; key: boolean;
repeat
    waiting[i] = true;
    key = true;
    while waiting [i] and key do          // enter CS if either waiting[i] or key is false
        key = Test-and-Set (lock);        // key is false if lock is false
    waiting[i] = false;
        Critical Section
    j = i+1 mod n;
    while (j ≠ i) and not waiting[j] do   // check if any P_j is waiting for CS
        j = j+1 mod n
    if j = i then
        lock = false;                     // no other process is waiting for CS
    else
        waiting[j] = false;               // P_j is waiting, let it enter CS next

        Remainder Section
until false;
```

Proof: Read textbook.

Mutex locks

- The hardware solutions are complicated and inaccessible to application programmers
  - Use software tool, called mutual exclusion (mutex)
- Mutex: acquire a lock before *entering* a critical section, and release the lock on *exit*
  - Calls to *acquire* () or *release* () must be atomic
  - Implement the functions with the hardware solution (e.g., test-and-set)
  - Busy waiting (also called *spinlock*) wastes CPU, but does not need context switch → it is good with short wait.
    - Implemented on multiprocessor systems

Repeat

acquire lock.

Critical section.

release lock

Remainder section.

Until *false;*

```
acquire () {
   while (!available); // busy wait
   available = false
}

release () {
   available = true
}
```

Semaphores

- Semaphore S is an *integer variable* that can only be accessed via two indivisible (atomic) operations.

  wait (S): **while** $S \leq 0$ **do** *no-op*; // busy wait
  $S = S - 1;$

  signal (S): $S = S + 1;$

- *wait* (S) and *signal* (S) are respectively equivalent to P (S) and V (S)
  - *P*roberen means to test; *V*erhogen means to increment
- Can be used to solve the *n*-process critical section problem.
- Note, the definition of *wait* (S) and *signal* (S) uses *busy waiting*.
  - Also called *spinlock*; that is, the process *spins* waiting for the *lock* to be false
  - Disadvantage? Advantage?

Semaphores (cont.)

Shared variables: **var** *mutex*: *semaphore*;

/* initially *mutex* = 1 */

<u>Process P_i:</u>

Repeat

wait (*mutex*);

Critical Section

signal (*mutex*);

Remainder Section

until *false*;

<u>Process P_j:</u>

Repeat

wait (*mutex*);

Critical Section

signal (*mutex*);

Remainder Section

until *false*;

<u>Process P_k:</u>

Repeat

wait (*mutex*);

Critical Section

signal (*mutex*);

Remainder Section

until *false*;

Semaphore Implementation

// Define a semaphore as a record

type *semaphore* = **record**
    value: *integer*;
    list: **list of** *process*;
end;

***wait* (*S*):** *S.value* = *S.value* - 1;
    if *S.value* < 0 **then**
    begin
        add process to *S.list*;
        block ();
    end;

***signal* (*S*):** *S.value* = *S.value* + 1;
    if *S.value* ≤ 0 **then**
    begin
        remove a process *P* from *S.list*;
        wakeup (*P*);
    end;

- *block* () suspends the process that invokes it
  - The process is **put into a waiting queue** associated with the semaphore; i.e., *S.list*
  - The process state is switched from **running** into **waiting** state.
  - CPU scheduler selects another process from **ready queue**.
- *wakeup* (*P*) resumes the execution of a blocked process *P* in *S.list*
  - *P* is placed in the **ready queue**.
  - The process state is switched from **waiting** into **ready** state.
- Each *wait* and *signal* must be executed atomically.
  - In a uni-processor, inhibit interrupts during *wait* and *signal*.
    - OK because it is done by the system
  - In a multi-processor, use other method such as spinlock.

Semaphore as General Synchronization Tool

- Consider two processes that require P_j to execute code *B* only after code *A* has been executed by P_i.
- Use semaphore *flag* initialized to 0.

Code:

P_i	P_j
.	.
.	.
A	*wait* (*flag*)
signal (*flag*)	*B*

- *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.

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);*

- Other problem: Starvation – *indefinite blocking*.
  - A process may never be removed from the semaphore queue in which it is suspended.
  - What happens if *S.list* is implemented as a last-in first out (LIFO)?

Priority Inversion

- **Scenario 1:** Consider a shared kernel data *X* that is being accessed by a process *A*. What happens if a higher priority process *B* wants to access *X*?
  - Kernel data is usually protected by a lock, and thus *B* must wait for *A* to finish with the data
- **Scenario 2:** Consider a shared resource *R* and three processes, *A*, *B* and *C*, where *C* and *A* have the highest and lowest priority respectively.
  - Assume *R* is being used by *A*, and *C* wants to use *R*
    - *C* must wait until *A* finishes with *R*
  - What happens if *B* preempts *A*? (*B* has higher priority than *A*, but lower than *C*)
    - Process *C* must wait for process *B* that has a lower priority before it can access resource *R*.
    - This is called *priority inversion* problem
    - Priority inversion occurs only in a system with more than two priorities.

Solution: *priority inheritance protocol* – all processes that are accessing resources needed by a higher priority process inherits the higher priority until they finish using the resources

- Process *A* would temporarily have the priority of process *C*, and thus *B* cannot preempt *A*
- When *A* finish using *R*, it goes back to its own priority and *C* will run because it has a higher priority than *B*

Bounded-Buffer Problem

Shared data

```
type item = …
var buffer = …
    mutex, full, empty: semaphore;
    nextp, nextc: item;
    full = 0; empty = n; mutex = 1;
```

Producer process

```
repeat
    ...
    produce an item in nextp
    …
    wait (empty);
    wait (mutex);
    ...
    add nextp to buffer
    ...
    signal (mutex);
    signal (full);
    …
until false;
```

Consumer process

```
repeat
    wait (full);
    wait (mutex);
    ...
    remove item from buffer to nextc
    …
    signal (mutex);
    signal (empty);
    …
    consume item in nextc
    …
until false;
```

Readers-writers problem

- Synchronisation problem involving reading and writing shared data objects.
  - Data can be accessed by more than one reader at the same time
  - However, when the data is being accessed by a writer, no other writer or readers can access it simultaneously.
- Assume no reader will be kept waiting unless a writer has already obtained permission to use shared object.
  - reader has higher priority.
  - An alternative problem: writer has higher priority than reader.
  - Solutions to either problem may cause starvation

Readers-writers problem (cont.)

Shared data

```
var mutex, wrt: semaphore;
    readcount: integer;
    readcount := 0; /* number of processes currently reading object */
    mutex := 1; /* ensure mutual exclusion for readcount */
    wrt := 1 /* for writers critical section and used by the first and last readers */
```

<u>Reader process</u>

```
repeat

    P(mutex);
        readcount = readcount + 1;
        if readcount = 1 then
            P(wrt);
    V(mutex);
        ...
        reading is performed;
        …
    P(mutex);
        readcount = readcount - 1;
        if readcount = 0 then
            V(wrt);
    V(mutex);

until false;
```

<u>Writer process</u>

```
repeat

    P(wrt);
        …
        writing is performed
        …
    V(wrt);

until false;
```

Readers-writers problem

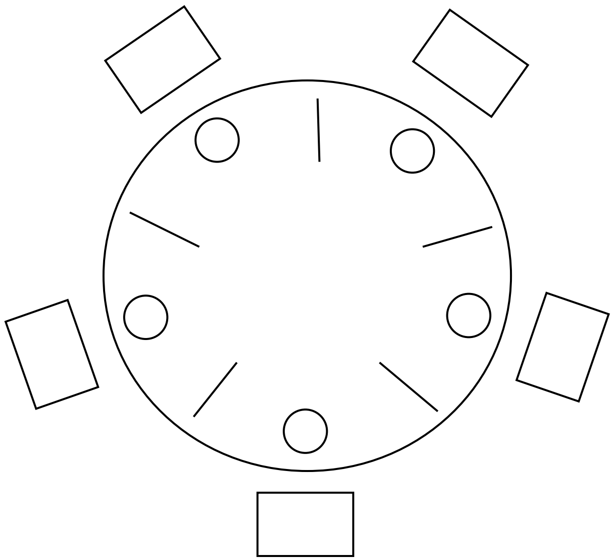
- Variable *mutex* is used to ensure mutual exclusion on variable *readcount*
- If writer is in its critical section and *n* readers are waiting, then:
  - one reader process is queued on *wrt*, and
  - *n*-1 reader processes are queued on *mutex*.
- Reader-writer problem and its solution has been used to provide reader-writer locks on some systems, e.g., Linux.
  - When a process wants to only read shared data, it gets the lock as reader.
    - More than one processes can read at the same time
  - When a process wants to write shared data, it gets the lock as writer
    - Only one process at a time can get the write lock.

Dining-philosophers problem by Dijkstra

- Five philosophers sit around a circular table
  - Each philosopher has a bowl of spaghetti
  - There are five *chopsticks*, each is placed between each pair of bowls
  - Each philosopher needs *two chopsticks* to eat the spaghetti
- Each philosopher can be either *thinking* or *eating*
  - When *thinking*, the philosopher doesn't interact with the others
  - From time to time each philosopher gets *hungry*
    - When getting *hungry*, a philosopher tries to *grab* two chopsticks close to her (i.e., one on her left and the other on her right) from the table
      - A philosopher can pick up one chopstick at a time
      - A philosopher is not allowed to seize chopsticks from others' hands! Not polite!
  - A philosopher can eat once she has two chopsticks
    - Once finish eating, she *releases* both chopsticks, and start thinking again

Problem: synchronize the philosophers so that each can think and eat with no deadlock and no philosopher can be starved to death!

Dining-philosophers problem (cont.)



Shared data
var *chopstick*: **array**[0..4] **of** *semaphore*;   // initially = 1
Philosopher *i*
repeat
P(*chopstick*[*i*]);   // pick left chopstick
P(*chopstick*[(*i*+1) **mod** 5]); // pick right chopstick
...
eat   // you have two chopsticks
...
V(*chopstick*[*i*]);   // release left chopstick
V(*chopstick*[(*i*+1) **mod** 5]); // release right chopstick
...
think   // you have no chopsticks
...
until *false*;

In the solution: No two neighbours can eat simultaneously; may have a deadlock (if all five grab left chopsticks).

Other Solutions:

- Allow at most four philosophers to sit at the table.
- Pickup chopsticks only if both are available; picking both chopsticks is a critical section
- *Odd* philosopher picks left chopstick first; *even* philosopher grabs right chopstick first.

Semaphore Limitation

- Incorrect use of semaphore can still result in timing errors→ hard to detect.
- Omitting *P*, or *V*, or both may create deadlock or no mutual exclusion.

- **Example 1:** several processes may execute in their critical sections simultaneously:

  V(mutex);

  …

  critical section

  …

  P(mutex);

- **Example 2:** deadlock may occur:

  P(mutex);

  …

  critical section

  …

  P(mutex);

Monitors

- The Monitor is a high-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.
- A monitor type contains a set of **defined functions** and a set of **shared variables** that can be accessed only by one of the functions
  - The monitor ensures that only one process at a time is active in the monitor

type *monitor-name* = **monitor**

variable declarations

procedure entry P_1 (…);

begin … **end**;

procedure entry P_2 (…);

begin … **end**;

procedure entry P_n (…);

begin … **end**;

begin

initialisation code

end.

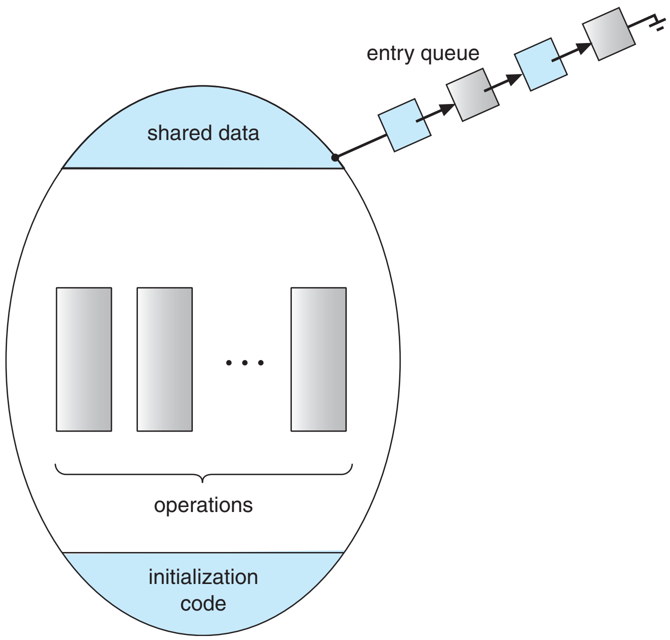


Fig. 5.16 (textbook)

Monitors (cont.)

- To allow a process to wait within the monitor, a condition variable must be declared as:

  var *x*, *y*: *condition*

- Condition variable can only be used with the operations *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 → different from semaphore.

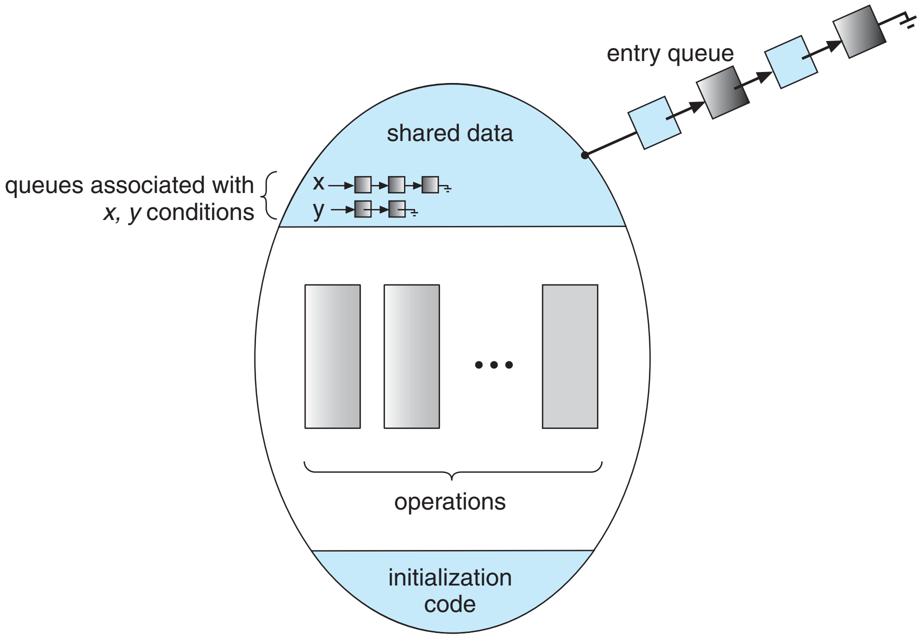


Fig. 5.17 (textbook)

Monitors (cont.)

- What happens if there is a suspended process Q waiting for condition x when $x.signal()$ is called by process P?
  - Monitor must ensure that only one of the two processes can be active at a time.
    - If Q is activated, P must wait
- There are two possibilities:
  - Signal and wait: P either waits until Q leaves the monitor or waits for another condition
  - Signal and continue: Q either waits until P leaves the monitor or waits for another condition

Dining Philosophers example

```
type dining-philosophers = monitor
    var state: array [0 .. 4] of (thinking, hungry, eating);
    var self: array [0 .. 4] of condition;

procedure entry pickup (i: 0 . .4);
begin
    state[i] = hungry;
    test (i);
    if state[i] ≠ eating then self[i].wait;
end;

procedure entry putdown (i:0..4);
begin
    state[i] = thinking;
    test (i+4 mod 5);
    test (i+1 mod 5);
end;

// initialization
begin
    for i = 0 to 4 do
        state[i] = thinking;
end.
```

```
procedure test (k: 0 .. 4);
begin
    //two neighbors are not eating
    if state[k+4 mod 5 ] ≠ eating
          and state [k] = hungry
          and state [k+1 mod 5] ≠ eating  then
    begin //pick up chopsticks if both are available
          state[k] = eating;
          self[k].signal;
    end;
end;
```

- Philosopher *i* must invoke:

  dp.pickup(*i*);

  …

  dp.putdown(*i*);

- Still possibility of starvation.

Semaphores for Monitor

- Use a *mutex* variable (initialized to one) for each monitor
  - A process executes *wait* (*mutex*) before entering the monitor
  - Use *signal* (*mutex*) when exiting the monitor
  - See Section 5.8.3 for details!