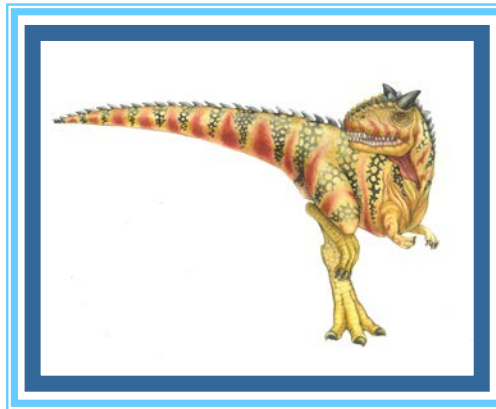


Chapter 5: Process Synchronization





Chapter 5: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Alternative Approaches



Objectives

- ❑ To present the concept of process synchronization.
- ❑ To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- ❑ To present both software and hardware solutions of the critical-section problem
- ❑ To examine several classical process-synchronization problems
- ❑ To explore several tools that are used to solve process synchronization problems



Background

- ❑ Processes can execute concurrently
 - ❑ May be interrupted at any time, partially completing execution
- ❑ Concurrent access to shared data may result in data inconsistency
- ❑ Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- ❑ Illustration of the problem:

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 **counter** that keeps track of the number of full buffers. Initially, **counter** 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 in next produced */  
  
    while (counter == BUFFER_SIZE) ;  
        /* do nothing */  
    buffer[in] = next_produced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```



Consumer

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



Race Condition

- `counter++` could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- `counter--` could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with “count = 5” initially:

S0: producer execute	<code>register1 = counter</code>	{register1 = 5}
S1: producer execute	<code>register1 = register1 + 1</code>	{register1 = 6}
S2: consumer execute	<code>register2 = counter</code>	{register2 = 5}
S3: consumer execute	<code>register2 = register2 - 1</code>	{register2 = 4}
S4: producer execute	<code>counter = register1</code>	{counter = 6}
S5: consumer execute	<code>counter = register2</code>	{counter = 4}

- 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**.



Critical Section Problem

- Consider system of n processes $\{p_0, p_1, \dots, p_{n-1}\}$
- Each process has **critical section** segment of code
 - Process may be changing common variables, updating table, writing file, etc
 - When one process in critical section, no other may be in its critical section
- **Critical section problem** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**



Critical Section

- General structure of process P_i

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



Motivating Example: Too Much Milk

- Two robots are programmed to maintain the milk inventory at a store...
- They are not aware of each other's presence...



Robot: Robot1



Robot: Robot2



Motivating Example: Too Much Milk

Robot1

Robot2

10:00 Look into fridge:

Out of milk





Motivating Example: Too Much Milk

Robot1

10:00 Look into fridge:

Out of milk

10:05 Head for the
warehouse

Robot2





Motivating Example: Too Much Milk

Robot1

10:05 Head for the
warehouse

Robot2

10:10 Look into fridge:
Out of milk





Motivating Example: Too Much Milk

Robot1

Robot2

10:10 Look into fridge:

Out of milk

10:15 Head for the
warehouse





Motivating Example: Too Much Milk

Robot1

10:20 Arrive with milk



Robot2

10:15 Head for the
warehouse





Motivating Example: Too Much Milk

Robot1

10:20 Arrive with milk



Robot2

10:15 Head for the
warehouse





Motivating Example: Too Much Milk

Robot1

10:20 Arrive with milk

10:25 Go party

Robot2





Motivating Example: Too Much Milk

Robot1

10:20 Arrive with milk

10:25 Go party

Robot2

10:30 Arrive with milk: “Uh oh...”





Too Much Milk: Solution 1

- Two properties:
 - Only one robot will go get milk
 - Someone should go get the milk if needed
- Basic idea of solution 1
 - Leave a note (kind of like a lock)
 - Remove the note (kind of like a unlock)
 - Don't go get milk if the note is around (wait)



Too Much Milk: Solution 1

```
if (no milk) {  
    if (no note) {  
        // leave a note;  
        // go get milk;  
        // remove the note;  
    }  
}
```



Too Much Milk: Solution 1

Robot1

```
10:00 if (no milk) {
```

Robot2





Too Much Milk: Solution 1

Robot1

```
10:00 if (no milk) {
```

Robot2

```
10:01 if (no milk) {
```





Too Much Milk: Solution 1

Robot1

```
10:00 if (no milk) {
```

Robot2

```
10:01 if (no milk) {  
10:02     if (no note) {
```





Too Much Milk: Solution 1

Robot1

```
10:00 if (no milk) {  
  
10:03   if (no note) {
```



Robot2

```
10:01 if (no milk) {  
10:02   if (no note) {
```





Too Much Milk: Solution 1

Robot1

```
10:00 if (no milk) {  
  
10:03   if (no note) {  
10:04     // leave a note
```

Robot2

```
10:01 if (no milk) {  
10:02   if (no note) {
```





Too Much Milk: Solution 1

Robot1

```
10:03  if (no note) {  
10:04    // leave a note
```



Robot2

```
10:01  if (no milk) {  
10:02    if (no note) {  
  
10:05    // leave a note
```





Too Much Milk: Solution 1

Robot1

```
10:03  if (no note) {  
10:04      // leave a note  
  
10:06      // go get milk
```

Robot2

```
10:02  if (no note) {  
  
10:05      // leave a note
```





Too Much Milk: Solution 1

Robot1

```
10:03  if (no note) {  
10:04      // leave a note  
  
10:06      // go get milk
```

Robot2

```
10:05      // leave a note  
  
10:07      // go get milk
```





Too Much Milk: Solution 2

- Okay...solution 1 does not work
- The notes are posted too late...
- What if both robots begin by leaving their own notes?



Too Much Milk: Solution 2

```
// leave a note;  
if (no note from the other) {  
    if (no milk) {  
        // go get milk;  
    }  
}  
// remove the note;
```



Too Much Milk: Solution 2

Robot1

10:00 // leave a note

Robot2





Too Much Milk: Solution 2

Robot1

10:00 // leave a note



Robot2

10:01 // leave a note





Too Much Milk: Solution 2

Robot1

10:00 // leave a note

10:02 if (no note from
Robot2) {...}

Robot2

10:01 // leave a note





Too Much Milk: Solution 2

Robot1

10:00 // leave a note

10:02 if (no note from
Robot2) {...}



Robot2

10:01 // leave a note

10:03 if (no note from
Robot1) {...}





Too Much Milk: Solution 2

Robot1

10:00 // leave a note

10:02 if (no note from
Robot2) {...}

10:04 // remove the note

Robot2

10:01 // leave a note

10:03 if (no note from
Robot1) {...}





Too Much Milk: Solution 2

Robot1

10:00 // leave a note

10:02 if (no note from
Robot2) {...}

10:04 // remove the note

Robot2

10:01 // leave a note

10:03 if (no note from
Robot1) {...}





Too Much Milk: Solution 2

Robot1

```
10:02 if (no note from  
      Robot2) {...}  
  
10:04 // remove the note
```



Robot2

```
10:01 // leave a note  
  
10:03 if (no note from  
      Robot1) {...}  
  
10:05 // remove the note
```





Too Much Milk: Solution 2

Robot1

```
10:02 if (no note from  
      Robot2) {...}  
  
10:04 // remove the note
```



Robot2

```
10:01 // leave a note  
  
10:03 if (no note from  
      Robot1) {...}  
  
10:05 // remove the note
```





Too Much Milk: Solution 3

Robot1

```
// leave Robot1's note
while (Robot2's note) { };
if (no milk) {
    // go get milk
}
// remove Robot1's note
```

Robot2

```
// leave Robot2's note
if (no Robot1's note) {
    if (no milk) {
        // go get milk
    }
}
// remove Robot2's note
```



Too Much Milk Solution 3

- How do we verify the correctness of a solution?
- Test arbitrary interleaving of locking and checking locks
 - In this case, leaving notes and checking notes



Robot2 Challenges Robot1: Case 1

Robot1

```
// leave Robot1's note  
while (Robot2's note) { };  
  
if (no milk) {  
    // go get milk  
}  
  
// remove Robot1's note
```

Robot2

```
// leave Robot2's note  
  
if (no Robot1's note) {  
}  
  
// remove Robot2's note
```

Time
↓



Robot2 Challenges Robot1: Case 2

Robot1

```
// leave Robot1's note  
  
while (Robot2's note) { };  
  
if (no milk) {  
    // go get milk  
}  
  
// remove Robot1's note
```

Robot2

```
// leave Robot2's note  
  
if (no Robot1's note) {  
}  
  
// remove Robot2's note
```

Time





Robot2 Challenges Robot1: Case 3

Robot1

```
// leave Robot1's note
```

```
while (Robot2's note) { };
```

```
if (no milk) {  
    // go get milk  
}
```

```
// remove Robot1's note
```

Robot2

```
// leave Robot2's note
```

```
if (no Robot1's note) {  
}
```

```
// remove Robot2's note
```

Time
↓



Robot1 Challenges Robot2: Case 1

Robot1

```
// leave Robot1's note  
while (Robot2's note) { };
```

```
if (no milk) {  
}  
// remove Robot1's note
```

Robot2

```
// leave Robot2's note  
if (no Robot1's note) {
```

```
    if (no milk) {  
        // go get milk  
    }  
}
```

```
// remove Robot2's note
```

Time
↓



Robot1 Challenges Robot2: Case 2

Robot1

```
// leave Robot1's note  
  
while (Robot2's note) { };  
  
if (no milk) {  
    // go get milk  
}  
  
// remove Robot1's note
```

Robot2

```
// leave Robot2's note  
  
if (no Robot1's note) {  
}  
  
// remove Robot2's note
```

Time
↓



Robot1 Challenges Robot2: Case 3

Robot1

```
// leave Robot1's note
while (Robot2's note) { };

if (no milk) {
    // go get milk
}

// remove Robot1's note
```

Robot2

```
// leave Robot2's note

if (no Robot1's note) {
}

// remove Robot2's note
```

Time





Lessons Learned

- Although it works, Solution 3 is ugly
 - Difficult to verify correctness
 - Two threads have different code
 - ▶ Difficult to generalize to N threads
 - While Robot1 is waiting, it consumes CPU time (*busy waiting*)



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 there exist some processes that wish to enter their critical section, then the selection of the processes 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
 - Assume that each process executes at a nonzero speed
 - No assumption concerning **relative speed** of the n processes



Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
 - ▶ Essentially free of race conditions in kernel mode



Algorithm for Process P_i

```
do {  
    while (turn == j);  
        critical section  
    turn = j;  
        remainder section  
} while (true);
```



Peterson's Solution

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the **load** and **store** machine-language 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

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



Peterson's Solution (Cont.)

□ Provable that the three CS requirement are met:

1. Mutual exclusion is preserved

P_i enters CS only if:

either `flag[j] = false` or `turn = i`

2. Progress requirement is satisfied

3. Bounded-waiting requirement is met



From the Previous Slides

- The *too-much-milk* example shows that writing concurrent programs with load and store instructions (i.e., C assignment statements) is tricky
- Because of the way modern computer architectures perform basic machine-language instructions, such as load and store, there are no guarantees that Peterson's solution will work correctly on such architectures.
- Programmers want to use higher-level operations, such as locks
- More elegant with higher-level primitives
lock→acquire();
if (no milk) { // go get milk }
lock→release();



Ways of Implementing Locks

- ◆ All implementations require some level of hardware support

	Locking primitives
High-level atomic operations	Locks, semaphores, monitors, send/receive
Low-level atomic operations	Load/store, interrupt disables, test_and_set



Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
 - Protecting critical regions via locks
- 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-interruptible
 - Either test memory word and set value
 - Or swap contents of two memory words



Solution to Critical-section Problem Using Locks

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



test_and_set Instruction

Definition:

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

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.



Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:

```
do {  
    while (test_and_set(&lock))  
        ; /* do nothing */  
    /* critical section */  
    lock = false;  
    /* remainder section */  
} while (true);
```



compare_and_swap Instruction

Definition:

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

1. Executed atomically
2. Returns the original value of passed parameter “value”
3. Set the variable “value” the value of the passed parameter “new_value” but only if “value” == “expected”. That is, the swap takes place only under this condition.



Solution using compare_and_swap

- Shared integer “lock” initialized to 0;
- Solution:

```
do {  
    while (compare_and_swap(&lock, 0, 1) != 0)  
        ; /* do nothing */  
    /* critical section */  
    lock = 0;  
    /* remainder section */  
} while (true);
```



Bounded-waiting Mutual Exclusion with test_and_set

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);
    waiting[i] = false;
    /* critical section */
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;
    /* remainder section */
} while (true);
```



Mutex Locks

- ❑ Previous solutions are complicated and generally inaccessible to application programmers
- ❑ OS designers build software tools to solve critical section problem
- ❑ Simplest is mutex lock
- ❑ Protect a critical section by first **acquire()** a lock then **release()** the lock
 - ❑ Boolean variable indicating if lock is available or not
- ❑ Calls to **acquire()** and **release()** must be atomic
 - ❑ Usually implemented via hardware atomic instructions
- ❑ But this solution requires **busy waiting**
 - ❑ This lock therefore called a **spinlock**



Lock acquire() and release()

```
□ acquire() {  
    while (!available)  
        ; /* busy wait */  
    available = false;;  
}  
  
□ release() {  
    available = true;  
}  
  
□ do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (true);
```




Implementing Locks with Test&Set(Spinlocks)

0

```
acquire(int *lock) {  
    while(test_and_set(lock) == 1)  
        /* do nothing */;  
}  
release(int *lock) { *lock = 0; }
```

Let me in!!!

```
acquire(houselock);  
Jump_on_the_couch();  
Be_goofy();  
release(houselock);
```

1



```
acquire(houselock);  
Nap_on_couch();  
Release(houselock);
```

1

No, Let *me*
in!!!





Implementing Locks with Test&Set(Spinlocks)

0

```
acquire(int *lock) {  
    while(test_and_set(lock) == 1)  
        /* do nothing */;  
}  
release(int *lock) { *lock = 0; }
```

Let me in!!!

```
acquire(houselock);  
Jump_on_the_couch();  
Be_goofy();  
release(houselock);
```

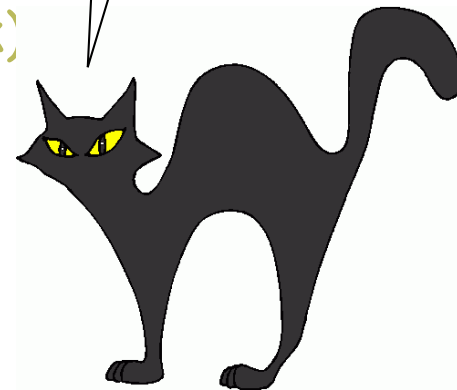
1



```
acquire(houselock);  
Nap_on_couch();  
Release(houselock);
```

1

No, Let me in!!!





Implementing Locks with Test&Set(Spinlocks)

1

```
acquire(int *lock) {  
    while(test_and_set(lock) == 1)  
        /* do nothing */;  
}  
release(int *lock) { *lock = 0; }
```

Yay, couch!!!

```
acquire(houselock);  
Jump_on_the_couch();  
Be_goofy();  
release(houselock);
```

```
acquire(houselock);  
Nap_on_couch();  
Release(houselock);
```

I still want in!

1



Implementing Locks with Test&Set(Spinlocks)

1

```
acquire(int *lock) {  
    while(test_and_set(lock) == 1)  
        /* do nothing */;  
}  
release(int *lock) { *lock = 0; }
```

Oooh, food!

```
acquire(houselock);  
Jump_on_the_couch();  
Be_goofy();  
release(houselock);
```

```
acquire(houselock);  
Nap_on_couch();  
Release(houselock);
```

It's cold here!

1



Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore **S** – integer variable
- Can only be accessed via two indivisible (atomic) operations

- **wait()** and **signal()**

- ▶ Originally called **P()** and **V()**

- Definition of the **wait()** operation

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

- Definition of the **signal()** operation

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



Two Types of Semaphore

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
 - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2
Create a semaphore “**synch**” initialized to 0

P1:

```
 $S_1$ ;  
signal(synch);
```

P2:

```
wait(synch);  
 $S_2$ ;
```

- Can implement a counting semaphore S as a binary semaphore



Semaphore Implementation

- Must guarantee that no two processes can execute the **wait()** and **signal()** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **wait** and **signal** code are placed in the critical section
 - Could now have **busy waiting** in critical section implementation
 - ▶ But implementation code is short
 - ▶ Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution



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



Implementation with no Busy waiting (Cont.)

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

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



Semaphores

- Semaphore count keeps state and reflects the sequence of past operations
 - A negative count reflects the number of processes on the sema wait queue
 - A positive count reflects number of future wait operations that will succeed
- No way to read the count! No way to grab multiple semaphores at the same time! No way to decrement/increment by more than 1!
- All semaphores must be initialized!



Two Uses of Semaphores

1. **Mutual exclusion:** Used to guard critical sections
 - Semaphore has an initial value of 1 (binary semaphore)
 - wait() is called before a critical section
 - signal() is called after the critical section

```
semaphore litter_box = 1;  
wait(litter_box);  
// critical section  
signal(litter_box);
```



Two Uses of Semaphores

1. Mutual exclusion

- Semaphore has an initial value of 1
- `wait()` is called before a critical section
- `signal()` is called after the critical section

```
semaphore litter_box = 1;
```

```
wait(litter_box);
```

```
// critical section
```

```
signal(litter_box);
```



`litter_box = 1`



Two Uses of Semaphores

1. Mutual exclusion

- Semaphore has an initial value of 1
- `wait()` is called before a critical section
- `signal()` is called after the critical section

```
semaphore litter_box = 1;
```

```
wait(litter_box); // purrr...
```

```
// critical section
```

```
signal(litter_box);
```



litter_box = 1 → 0



Two Uses of Semaphores

1. Mutual exclusion

- Semaphore has an initial value of 1
- `wait()` is called before a critical section
- `signal()` is called after the critical section



```
semaphore litter_box = 1;
```

```
wait(litter_box);
```

```
// critical section
```

```
signal(litter_box);
```

litter_box = 0



Two Uses of Semaphores

1. Mutual exclusion

- Semaphore has an initial value of 1
- `wait()` is called before a critical section
- `signal()` is called after the critical section



```
semaphore litter_box = 1;
```

```
wait(litter_box); // meow...
```

```
// critical section
```

```
signal(litter_box);
```

litter_box = 0



Two Uses of Semaphores

1. Mutual exclusion

- Semaphore has an initial value of 1
- `wait()` is called before a critical section
- `signal()` is called after the critical section

```
semaphore litter_box = 1;
```

```
wait(litter_box);
```

```
// critical section
```

```
signal(litter_box);
```

`litter_box = 0 → 1`





Two Uses of Semaphores

- 2. **Condition Synchronization (Scheduling Constraints)**
:used to express general scheduling constraints where threads must wait for some circumstance.

Semaphore usually has an initial value of 0
(usually counting semaphore)

```
semaphore wait_left = 0;
```

```
semaphore wait_right = 0;
```

```
Left_Paw() {  
    slide_left();  
    signal(wait_left);  
    wait(wait_right);  
    slide_right();  
}
```

```
Right_Paw() {  
    wait(wait_left);  
    slide_left();  
    slide_right();  
    signal(wait_right);  
}
```



Two Uses of Semaphores

2. Condition Synchronization

- Semaphore usually has an initial value of 0

```
semaphore wait_left = 0;  
semaphore wait_right = 0;
```

wait_left = 0
wait_right = 0



```
Left_Paw() {  
    slide_left();  
    signal(wait_left);  
    wait(wait_right);  
    slide_right();  
}
```

```
Right_Paw() {  
    wait(wait_left);  
    slide_left();  
    slide_right();  
    signal(wait_right);  
}
```





Two Uses of Semaphores

2. Condition Synchronization

- Semaphore usually has an initial value of 0

```
semaphore wait_left = 0;  
semaphore wait_right = 0;
```

<pre>wait_left = 0 wait_right = 0</pre>

```
Left_Paw() {  
    slide_left();  
    signal(wait_left);  
    wait(wait_right);  
    slide_right();  
}
```

```
Right_Paw() {  
    wait(wait_left);  
    slide_left();  
    slide_right();  
    signal(wait_right);  
}
```





Two Uses of Semaphores

2. Condition Synchronization

- Semaphore usually has an initial value of 0

```
semaphore wait_left = 0;  
semaphore wait_right = 0;
```

<pre>wait_left = 0 wait_right = 0</pre>

```
Left_Paw() {  
    slide_left();  
    signal(wait_left);  
    wait(wait_right);  
    slide_right();  
}
```

```
Right_Paw() {  
    wait(wait_left);  
    slide_left();  
    slide_right();  
    signal(wait_right);  
}
```



wait



Two Uses of Semaphores

2. Condition Synchronization

- Semaphore usually has an initial value of 0

```
semaphore wait_left = 0;  
semaphore wait_right = 0;
```

<pre>wait_left = 0 wait_right = 0</pre>

```
Left_Paw() {  
    slide_left();  
    signal(wait_left);  
    wait(wait_right);  
    slide_right();  
}
```

```
Right_Paw() {  
    wait(wait_left);  
    slide_left();  
    slide_right();  
    signal(wait_right);  
}
```





Two Uses of Semaphores

2. Condition Synchronization

- Semaphore usually has an initial value of 0

```
semaphore wait_left = 0;  
semaphore wait_right = 0;
```

<pre>wait_left = 0 → 1 wait_right = 0</pre>

```
Left_Paw() {  
    slide_left();  
    signal(wait_left);  
    wait(wait_right);  
    slide_right();  
}
```

```
Right_Paw() {  
    wait(wait_left);  
    slide_left();  
    slide_right();  
    signal(wait_right);  
}
```





Two Uses of Semaphores

2. Condition Synchronization

- Semaphore usually has an initial value of 0

```
semaphore wait_left = 0;  
semaphore wait_right = 0;
```

<pre>wait_left = 1 → 0 wait_right = 0</pre>

```
Left_Paw() {  
    slide_left();  
    signal(wait_left);  
    wait(wait_right);  
    slide_right();  
}
```

```
Right_Paw() {  
    wait(wait_left);  
    slide_left();  
    slide_right();  
    signal(wait_right);  
}
```





Two Uses of Semaphores

2. Condition Synchronization

- Semaphore usually has an initial value of 0

```
semaphore wait_left = 0;  
semaphore wait_right = 0;
```

<pre>wait_left = 0 wait_right = 0</pre>

```
Left_Paw() {  
    slide_left();  
    signal(wait_left);  
    wait(wait_right);  
    slide_right();  
}
```

```
Right_Paw() {  
    wait(wait_left);  
    slide_left();  
    slide_right();  
    signal(wait_right);  
}
```





Two Uses of Semaphores

2. Condition Synchronization

- Semaphore usually has an initial value of 0

```
semaphore wait_left = 0;  
semaphore wait_right = 0;
```

wait_left = 0
wait_right = 0

```
Left_Paw() {  
    slide_left();  
    signal(wait_left);  
    wait(wait_right);  
    slide_right();  
}
```

```
Right_Paw() {  
    wait(wait_left);  
    slide_left();  
    slide_right();  
    signal(wait_right);  
}
```

wait





Two Uses of Semaphores

2. Condition Synchronization

- Semaphore usually has an initial value of 0

```
semaphore wait_left = 0;  
semaphore wait_right = 0;
```

<pre>wait_left = 0 wait_right = 0</pre>

```
Left_Paw() {  
    slide_left();  
    signal(wait_left);  
    wait(wait_right);  
    slide_right();  
}
```

```
Right_Paw() {  
    wait(wait_left);  
    slide_left();  
    slide_right();  
    signal(wait_right);  
}
```





Two Uses of Semaphores

2. Condition Synchronization

- Semaphore usually has an initial value of 0

```
semaphore wait_left = 0;  
semaphore wait_right = 0;
```

<pre>wait_left = 0 wait_right = 0 → 1</pre>

```
Left_Paw() {  
    slide_left();  
    signal(wait_left);  
    wait(wait_right);  
    slide_right();  
}
```

```
Right_Paw() {  
    wait(wait_left);  
    slide_left();  
    slide_right();  
    signal(wait_right);  
}
```





Two Uses of Semaphores


2. Condition Synchronization

- Semaphore usually has an initial value of 0

```
semaphore wait_left = 0;  
semaphore wait_right = 0;
```

<pre>wait_left = 0 wait_right = 1 → 0</pre>

```
Left_Paw() {  
    slide_left();  
    signal(wait_left);  
    wait(wait_right);  
    slide_right();  
}  
  
Right_Paw() {  
    wait(wait_left);  
    slide_left();  
    slide_right();  
    signal(wait_right);  
}
```





Two Uses of Semaphores

2. Condition Synchronization

- Semaphore usually has an initial value of 0

```
semaphore wait_left = 0;  
semaphore wait_right = 0;
```

```
wait_left = 0  
wait_right = 0
```

```
Left_Paw() {  
    slide_left();  
    signal(wait_left);  
    wait(wait_right);  
    slide_right();  
}
```

```
Right_Paw() {  
    wait(wait_left);  
    slide_left();  
    slide_right();  
    signal(wait_right);  
}
```





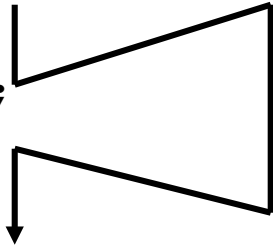
Two Uses of Semaphores

2. Condition Synchronization

- Semaphore usually has an initial value of 0

```
semaphore s1 = 0;
```

```
semaphore s2 = 0;
```

<pre>A() { write(x); signal(s1); wait(s2); read(y); }</pre>		<pre>B() { wait(s1); read(x); write(y); signal(s2); }</pre>
---	--	---



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

P_1
`wait(Q);`
`wait(S);`
`...`
`signal(Q);`
`signal(S);`

- **Starvation** – **indefinite blocking**
 - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
 - Solved via **priority-inheritance protocol**



Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
 - Bounded-Buffer Problem
 - Readers and Writers Problem
 - Dining-Philosophers Problem



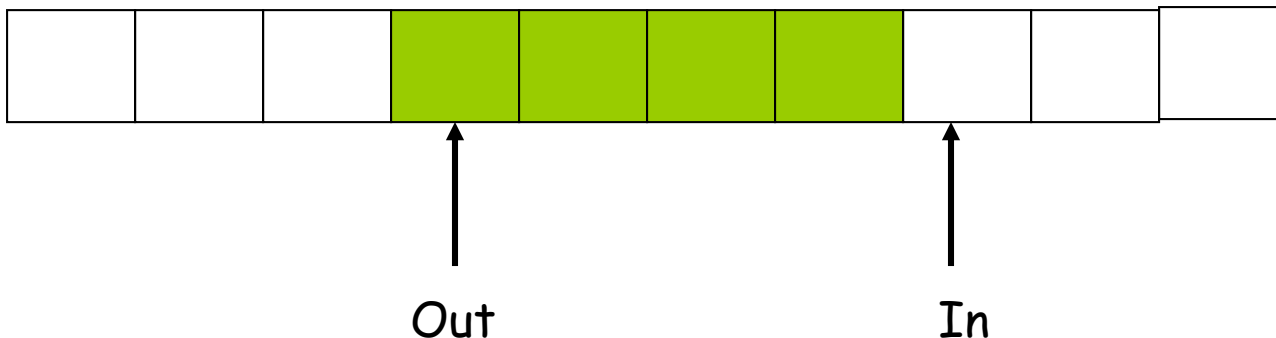
Bounded Buffer

- Bounded buffer:
 - Arises when two or more threads communicate with some threads “producing” data that others “consume”.
 - Example: preprocessor for a compiler “produces” a preprocessed source file that the parser of the compiler “consumes”



Producer-Consumer Problem

- Start by imagining an unbounded (infinite) buffer
- Producer process writes data to buffer
 - Writes to *In* and moves rightwards
- Consumer process reads data from buffer
 - Reads from *Out* and moves rightwards
 - Should not try to consume if there is no data

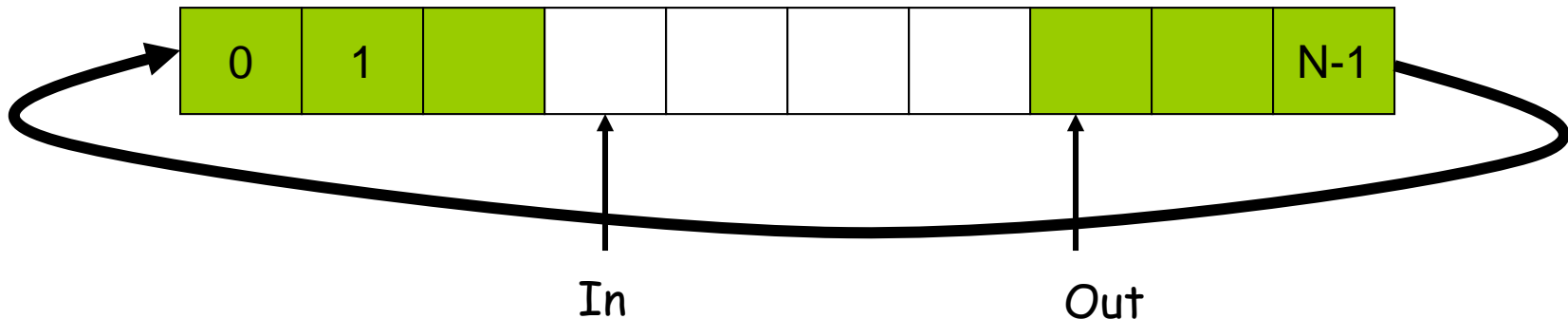


Need an infinite buffer



Producer-Consumer Problem

- Bounded buffer: size 'N'
 - Access entry 0... N-1, then “wrap around” to 0 again
- Producer process writes data to buffer
 - Must not write more than 'N' items more than consumer “ate”
- Consumer process reads data from buffer
 - Should not try to consume if there is no data





Producer-Consumer Problem

- A number of applications:
 - Data in a file you want to print consumed by printer spooler, which produces data consumed by line printer device driver
 - Web server produces data consumed by client's web browser
- Example: so-called “pipe” (|) in Unix
 - > cat file | sort | uniq | more
 - > prog | sort
- Thought questions: where's the bounded buffer?
- How “big” should the buffer be, in an ideal world?



Producer-Consumer Problem

- Solving with semaphores
 - We'll use two kinds of semaphores
 - We'll use *counters* to track how much data is in the buffer
 - ▶ One counter counts as we add data and stops the producer if there are N objects in the buffer
 - ▶ A second counter counts as we remove data and stops a consumer if there are 0 in the buffer
 - Idea: since general semaphores can count for us, we don't need a separate counter variable
- Why do we need a second kind of semaphore?
- We'll also need a mutex semaphore
 - There are multiple Consumers and Producers



Bounded-Buffer Problem

- n buffers, each can hold one item
- Semaphore **mutex** initialized to the value 1
- Semaphore **full** initialized to the value 0
- Semaphore **empty** initialized to the value n



Bounded Buffer Problem (Cont.)

- The structure of the producer process

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



Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
Do {  
    wait(full);  
    wait(mutex);  
    ...  
    /* remove an item from buffer to next_consumed */  
    ...  
    signal(mutex);  
    signal(empty);  
    ...  
    /* consume the item in next consumed */  
    ...  
} while (true);
```




Bounded Buffer Problem (Java)

```
// Producers call this method
public synchronized void insert(E item) {
    while (count == BUFFER_SIZE) {
        try {
            wait();
        }
        catch (InterruptedException e) { }
    }

    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    ++count;

    notify();
}

// Consumers call this method
public synchronized E remove() {
    E item;

    while (count == 0) {
        try {
            wait();
        }
        catch (InterruptedException e) { }
    }

    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    --count;

    notify();

    return 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
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
 - Data set
 - Semaphore `rw_mutex` initialized to 1
 - Semaphore `mutex` initialized to 1
 - Integer `read_count` initialized to 0



Readers-Writers Problem (Cont.)

- The structure of a writer process

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



Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {  
    wait(mutex);  
    read_count++;  
    if (read_count == 1)  
        wait(rw_mutex);  
    signal(mutex);  
  
    ...  
    /* reading is performed */  
    ...  
    wait(mutex);  
    read_count--;  
    if (read_count == 0)  
        signal(rw_mutex);  
    signal(mutex);  
} while (true);
```



Readers/Writers: Scenario 1

R1:

Read ()

R2:

Read ()

W1:

Write ()



Readers/Writers: Scenario 2

R1:

Read ()

R2:

Read ()

W1:

Write ()



Readers/Writers: Scenario 3

R1:

Read ()

R2:

Read ()

W1:

Write ()



Readers/Writers Solution: Discussion

- ❑ The first reader blocks if there is a writer; any other readers who try to enter block on mutex.
- ❑ The last reader to exit signals a waiting writer.
- ❑ When a writer exits, if there is both a reader and writer waiting, which goes next depends on the scheduler.
- ❑ If a writer exits and a reader goes next, then all readers that are waiting will fall through (at least one is waiting on `rw_mutex` and zero or more can be waiting on `mutex`).
- ❑ This Readers/Writers solution favors readers.

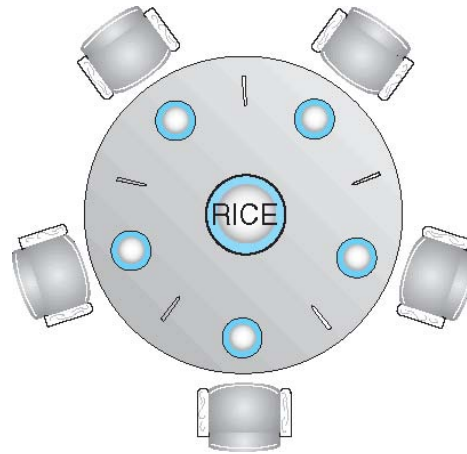


Readers-Writers Problem Variations

- **First** variation – no reader kept waiting unless writer has permission to use shared object
- **Second** variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks



Dining-Philosophers Problem



- ❑ Philosophers spend their lives alternating thinking and eating
- ❑ Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
 - ❑ Need both to eat, then release both when done
- ❑ In the case of 5 philosophers
 - ❑ Shared data
 - ▶ Bowl of rice (data set)
 - ▶ Semaphore **chopstick** [5] initialized to 1



Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

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

- What is the problem with this algorithm?



Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
 - Allow at most 4 philosophers to be sitting simultaneously at the table.
 - Allow a philosopher to pick up the chopsticks only if both are available (picking must be done in a critical section).
 - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.



Real-World Examples

□ Producer-consumer

- Audio/Video player: network and display threads; shared buffer
- Web servers: master thread and slave thread

□ Reader-writer

- Banking system: read account balances versus update

□ Dining Philosophers

- Cooperating processes that need to share limited resources
 - Set of processes that need to lock multiple resources
 - Disk and tape (backup),
 - Travel reservation: hotel, airline, car rental databases



Problems with Semaphores

- Semaphores are a huge step up from the equivalent load/store implementation, but have the following drawbacks.
 - They are essentially shared global variables.
 - There is no linguistic connection between the semaphore and the data to which the semaphore controls access.
 - Access to semaphores can come from anywhere in a program.
 - They serve multiple purposes, (mutual exclusion, scheduling constraints, ...)
 - There is no control or guarantee of proper usage.
 - Deadlock and starvation are possible.

Solution: use a higher level primitive called *monitors*



Monitors

- ❑ A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- ❑ *Abstract data type*, internal variables only accessible by code within the procedure
- ❑ A monitor is similar to a class that ties the data, operations, and in particular, the synchronization operations all together
- ❑ Unlike classes:
 - monitors guarantee mutual exclusion, i.e., only one thread may execute a given monitor's methods at a time.
 - monitors require all data to be private.



Monitors

- ❑ Only one process may be active within the monitor at a time
- ❑ But not powerful enough to model some synchronization schemes

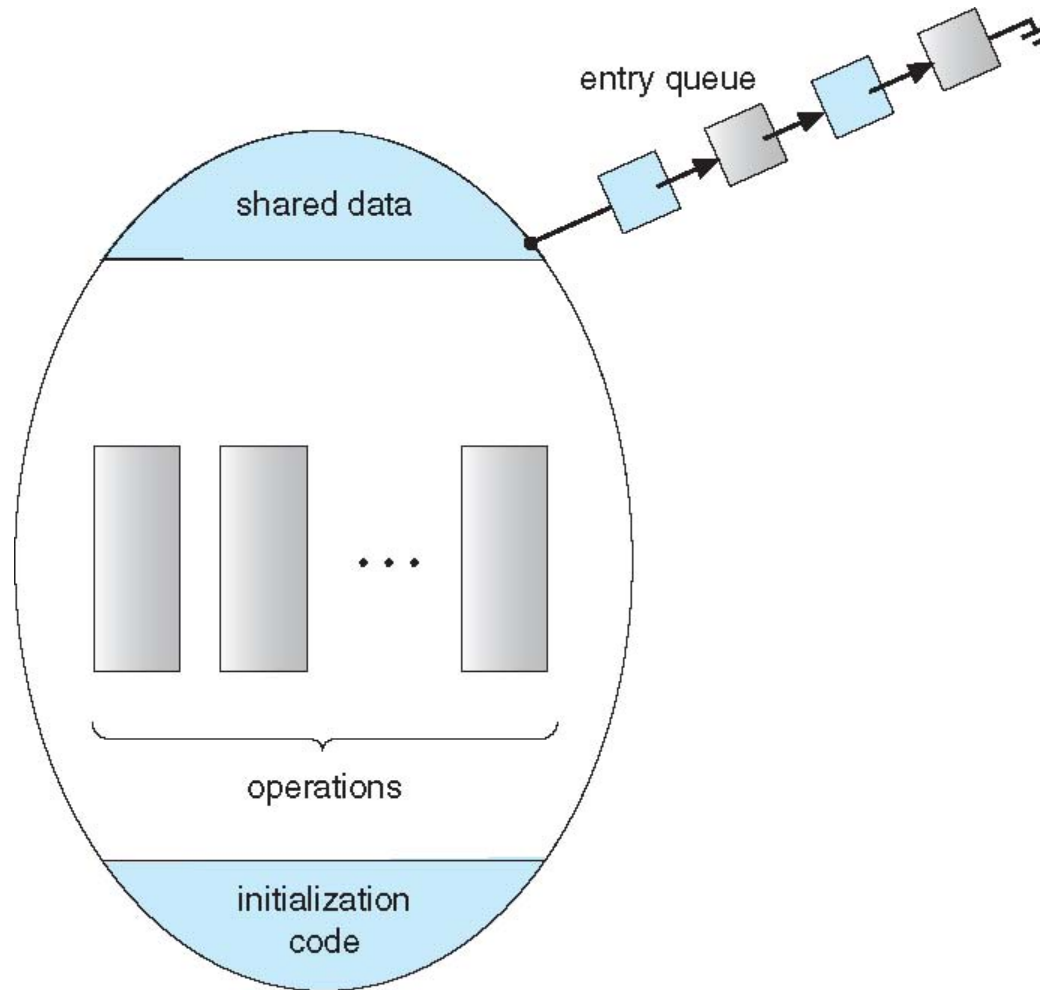
```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { ... }

    procedure Pn (...) {.....}

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




Schematic view of a Monitor



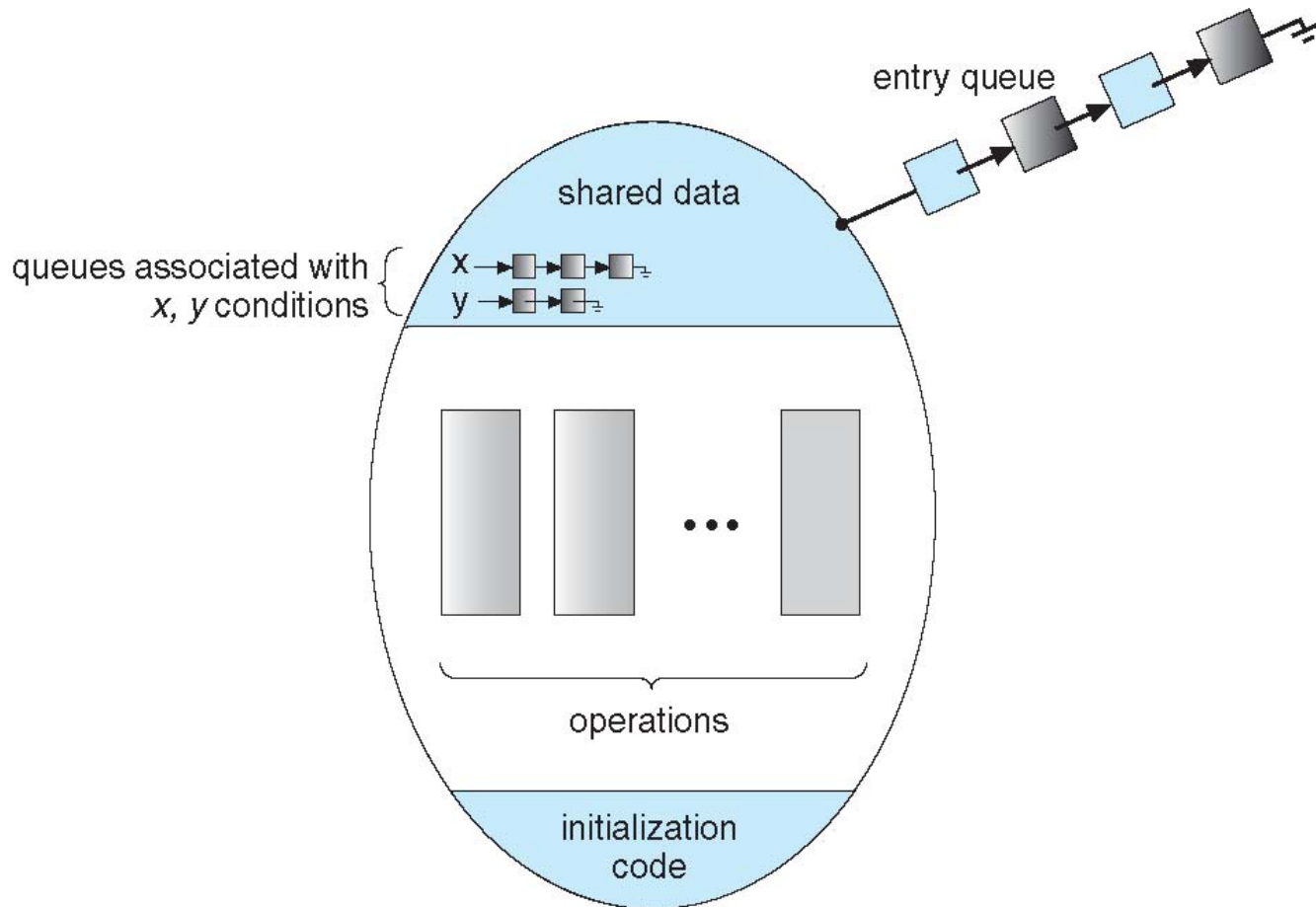


Condition Variables

- **condition** **x**, **y**;
- Two operations are allowed on a condition variable:
 - **x.wait()** – a process that invokes the operation is suspended until **x.signal()**
 - **x.signal()** – resumes one of processes (if any) that invoked **x.wait()**
 - ▶ If no **x.wait()** on the variable, then it has no effect on the variable



Monitor with Condition Variables





Condition Variables Choices

- If process P invokes `x.signal()`, and process Q is suspended in `x.wait()`, what should happen next?
 - Both Q and P cannot execute in parallel. If Q is resumed, then P must wait
- Options include
 - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
 - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
 - Both have pros and cons – language implementer can decide
 - Monitors implemented in Concurrent Pascal compromise
 - ▶ P executing signal immediately leaves the monitor, Q is resumed
 - Implemented in other languages including Mesa, C#, Java



Monitor Solution to Dining Philosophers

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

    void pickup (int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING) self[i].wait;
    }

    void putdown (int i) {
        state[i] = THINKING;
        // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
    }
}
```



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;
}
}
```



Solution to Dining Philosophers (Cont.)

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

`DiningPhilosophers.pickup(i);`

`EAT`

`DiningPhilosophers.putdown(i);`

- **No deadlock**, but starvation is possible



Synchronization Examples

- Solaris
- Windows
- Linux
- Pthreads



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
 - ❑ Starts as a standard semaphore spin-lock
 - ❑ If lock held, and by a thread running on another CPU, spins
 - ❑ If lock held by non-run-state thread, block and sleep waiting for signal of lock being released
- ❑ Uses **condition variables**
- ❑ Uses **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
 - ❑ Turnstiles are per-lock-holding-thread, not per-object
- ❑ Priority-inheritance per-turnstile gives the running thread the highest of the priorities of the threads in its turnstile



Windows Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
 - Spinlocking-thread will never be preempted
- Also provides **dispatcher objects** user-land which may act mutexes, semaphores, events, and timers
 - **Events**
 - ▶ An event acts much like a condition variable
 - Timers notify one or more thread when time expired
 - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)



Linux Synchronization

- ❑ Linux:
 - ❑ Prior to kernel Version 2.6, disables interrupts to implement short critical sections
 - ❑ Version 2.6 and later, fully preemptive
- ❑ Linux provides:
 - ❑ Semaphores
 - ❑ atomic integers
 - ❑ spinlocks
 - ❑ reader-writer versions of both
- ❑ On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption



Pthreads Synchronization

- ❑ Pthreads API is OS-independent
- ❑ It provides:
 - ❑ mutex locks
 - ❑ condition variable
- ❑ Non-portable extensions include:
 - ❑ read-write locks
 - ❑ spinlocks



Alternative Approaches

- Transactional Memory
- OpenMP
- Functional Programming Languages



Transactional Memory

- A **memory transaction** is a sequence of read-write operations to memory that are performed atomically.

```
void update()  
{  
    /* read/write memory */  
}
```



OpenMP

- OpenMP is a set of compiler directives and API that support parallel programming.

```
void update(int value)
{
    #pragma omp critical
    {
        count += value
    }
}
```

The code contained within the `#pragma omp critical` directive is treated as a critical section and performed atomically.



Functional Programming Languages

- Functional programming languages offer a different paradigm than procedural languages in that they do not maintain state.
- Variables are treated as immutable and cannot change state once they have been assigned a value.
- There is increasing interest in functional languages such as Erlang and Scala for their approach in handling data races.

End of Chapter 5

