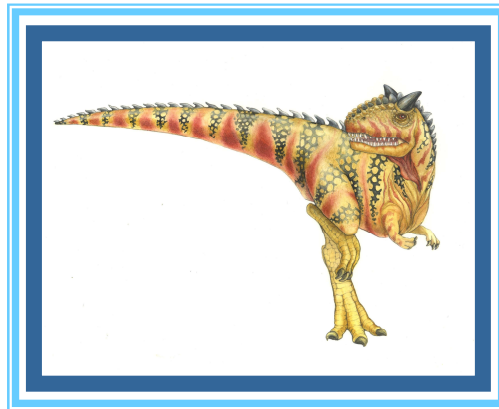
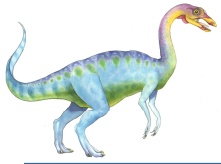


Chapter 5: Process Synchronization

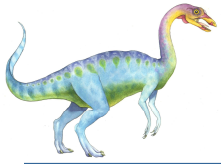




Chapter 5: Process Synchronization

- 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.





Bounded-Buffer – Producer-Consumer Problem

Producer

```
item next_produced;
while (true) {
    /* produce an item in next produced */
    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* buffer is full, do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

Consumer

```
item next_consumed;
while (true) {
    while (in == out)
        ; /* buffer is empty, do
nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next
consumed */
}
```

□ **Solution is correct but can only use BUFFER_SIZE-1 elements.**





Producer - Consumer

Producer

```
while (true) {
    /* produce an item in next produced */
    while (counter == BUFFER_SIZE); //(in + 1) % BUFFER_SIZE == out
        // buffer is full, do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

This solution allows us to use the full buffer size. However, we have a racing problem explained next slide.

Consumer

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

Note: **counter** is a shared variable between the two processes (Producer & Consumer).





Race Condition

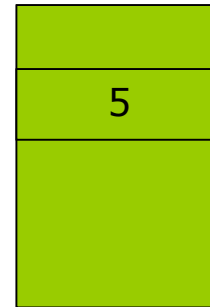
- `counter++` could be implemented as

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

- `counter--` could be implemented as

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

Memory
address
counter



Memory

- Counter is initially "5", then a producer produces an item, and a consumer consumes an item. So, if all goes well, the counter value is supposed to stay 5.
- However, in the scenario below, the counter result is 4 which is wrong.
- Concurrent access to counter caused this problem.

- Consider this execution interleaving with "counter = 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}

- As a result, we have inconsistent values (counter = 6 or 4), that is we have a race between the two processes (who will modify the value first?).
- To avoid race condition, it must execute all three instructions as one unit (no interleaving).

Another scenario if you switch S4, S5, last value of counter is 6 which is also wrong.

To fix this:

- Counter++ with all its steps should be considered a critical section.
- Counter-- with all its steps should be considered a critical section.





Critical Section Problem

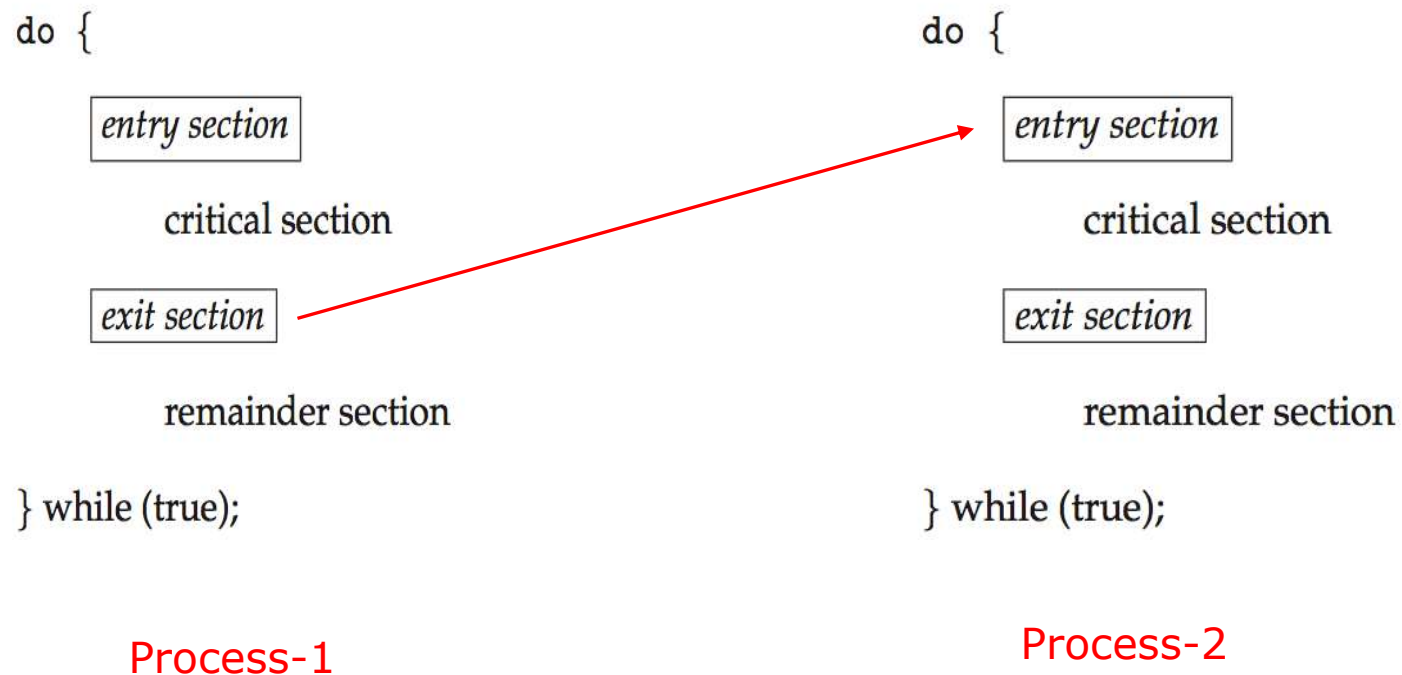
- Consider system of n processes $\{p_0, p_1, \dots, p_{n-1}\}$.
- Each process has **critical section** segment of code:
 - Process may be changing common variables, updating table, writing file, etc.
 - **When one process in critical section, no other may be in its critical section.**
- ***Critical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





Critical Section (CS)

- General structure of process P_i



Two processes, each can have its own task. However, the two of them needed to access a shared variable. So, the part of code that access this variable is considered a **critical section**. Only one process can enter it.





Algorithm for Process P_i

// Process i

```
do {  
    while (turn == j); // if true, busy wait  
        critical section  
    turn = j; // exit section  
    remainder section  
} while (true);
```

// Process j

```
do {  
    while (turn == i); // if true, busy wait  
        critical section  
    turn = i; // exit section  
    remainder section  
} while (true);
```





Solution to Critical-Section Problem

This solution must hold three conditions:

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.**
 - Otherwise, processes may not finish their work.
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.
 - Otherwise, processes might wait forever for entering the critical section and therefore they won't finish their work.
 - 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:

1. **Preemptive** – Allows preemption of process when running in kernel mode.
 - So, process is forced to leave CPU.
2. **Non-preemptive** – Runs until exits kernel mode, blocks, or voluntarily yields CPU.
 - So, process is not forced to leave CPU. So, it leaves if it blocks, or it leaves by choice.
 - Essentially free of **race conditions** in kernel mode. That is, grantees to complete the access of its critical section.





Peterson's Solution

- Good algorithmic description of solving the problem.
- It is a software-based solution for only two processes solution.
- Assume that the **load** and **store** machine-language instructions are **atomic; that is, cannot be interrupted**.
- In the Peterson's solution, two processes share two variables:
 - **int turn;** // integer variable
 - **Boolean flag[2]** // flag is an array of two Boolean elements
- 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**. For example, **flag[i] = true** implies that process P_i is ready!





Algorithm for Process P_i

```
do { //Process i
```

```
    flag[i] = true;
```

```
    turn = j;
```

```
    while (flag[j] && turn == j);
```

If true, P_i waits
If false, P_i enters CS

critical section

```
    flag[i] = false;    //exit section
```

remainder section

```
    } while (true);
```

```
//Process j
```

```
    flag[j] = true;
```

```
    turn = i;
```

```
    while (flag[i] && turn == i); //if true,  $P_j$  waits.
```

//if false, P_j can enter CS.

critical section

```
    flag[j] = false;    //exit section
```







Peterson's Solution

```

do {
    P0
    flag[0] = true;
    turn = 1;
    while (flag[1] && turn == 1);
    //if true, P0 waits
    print("Hello");
    critical section
    flag[0] = false;
    remainder section
} while (true);

do {
    P1
    flag[1] = true;
    turn = 0;
    while (flag[0] && turn == 0);
    //if true, P1 waits
    print("world");
    critical section
    flag[1] = false;
    remainder section
} while (true);
  
```

Diagram showing the execution flow between P₀ and P₁. Blue arrows indicate the sequence of operations and the state of the flags and turn variable.

Example	Turn	flag[0]	Flag[1]	P ₀	P ₁
initialization	0	F	F		
P₀ : flag[0] = true		T			
P₀ :turn = 1	1				
P₁ :flag[1] = true			T		
P₁ :turn = 0	0				
P₀ :while (flag[1] && turn ==1)= T&&F= F				C.S	
P₁ :while (flag[0] && turn==0) = T&&T=T					
P₀ :flag[0] = false		F		R	
P₁ :while (flag[0] && turn==0) F&T=F					C.S
P₀ :flag[0] = true; turn = 1;	1	T			
P₀ :while (flag[1] && turn ==1) T					
P₁ :flag[1] = false;			F		
P₀ :while (flag[1] && turn ==1) F				C.S	

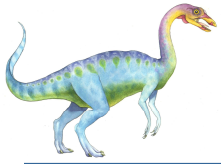


Peterson's Solution (Cont.)

- Provable that the three Critical Section requirements (conditions) 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 (i.e., exist section).
 3. Bounded-waiting requirement is met.

Points 2 and 3 are met because P_i will enter the critical section (progress) after at most one entry by P_j (bounded waiting).



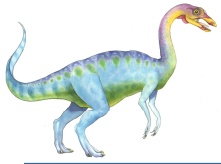


Synchronization Hardware



- 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
 - ▶ can be time consuming, since the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases.
 - ▶ the effect on a system's clock if the clock is kept updated by interrupts.
 - ▶ Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
 - ▶ **Atomic** = **non-interruptible unit**
 - Either test memory word and set value
 - Or swap contents of two memory words





test_and_set Instruction



Definition:

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

1. Executed atomically as one uninterruptable unit.
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);
```





Solution using test_and_set()

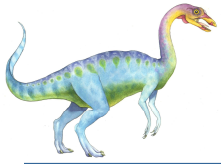
```

boolean test_and_set (boolean *target)
{
    do {
        P0
        while (test_and_set(&lock));
        /* do nothing */
        /* critical section */
        lock = false;
        /* remainder section */
    } while (true);

    do {
        P1
        while (test_and_set(&lock));
        /* do nothing */
        /* critical section */
        lock = false;
        /* remainder section */
    } while (true);

```

	Lock	target	rv	P ₀	P ₁
initialization	F				
P ₀ : while (test_and_set(&lock))		F			
Function P ₀ : rv = *target			F		
Function P ₀ : *target = TRUE	T	T			
Function P ₀ : return rv (test_and_set(&lock)) F				C.S	
P ₁ : while (test_and_set(&lock))	T				
Function P ₁ : rv = *target		T			
Function P ₁ : *target = TRUE		T			
Function P ₁ : return rv:			T		
P ₁ : while (test_and_set(&lock)) T					
P ₀ : lock = false	F	F	F		
Op P ₁ : while (test_and_set(&lock)) F	T				C.S



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.





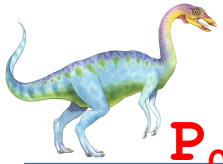
Solution using compare_and_swap



- 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);
```





P₀

```

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

```

P₁

```

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

```

```

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

```

Boolean lock = 0

	Lock	*value	exp	New va	temp	P ₀	P ₁
initialization	0						
P₀ : while (compare_and_swap(&lock, 0, 1) != 0)		0	0	1			
Function P₀ : temp = *value					0		
Function P₀ : if (*value == expected) → yes							
Function P₀ : *value = new_value	1	1					
Function P₀ : return temp; (compare_and_swap(&lock, 0, 1) != 0) 0						C.S	
P₁ : while (compare_and_swap(&lock, 0, 1) != 0)		1	0	1			
Function P₁ : temp = *value					1		
Function P₁ : if (*value == expected) → no then do nothing							
Function P₁ : return temp; : (compare_and_swap(&lock, 0, 1) != 0) 1							
P₀ : lock = 0	0	0	0	1			
Function P₁ : temp = *value,					0		
Function P₁ : if (*value == expected) yes	1	1					
P₁ : return temp; (compare_and_swap(&lock, 0, 1) != 0) 0						C.S	



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);
```





Solution to Critical-Section Problem Using Locks

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

Previous solutions (hardware solutions) are complex.
Next solutions are simpler and are based on acquiring
and releasing locks.





Mutex Locks

- Hardware solutions are complicated and generally inaccessible to application programmers.
- OS designers build software tools to solve critical section problem.
- Simplest is **mutex lock** (between two processes only).
- Protect a critical section by first **acquire()** a lock then **release()** the lock.
 - **Boolean variable** indicating if lock is available or not.
- System calls in OS to **acquire()** and **release()** must be **atomic (cannot be interrupted)**.
 - Usually implemented via hardware atomic instructions.
- **Negative:**
 - But this solution requires **busy waiting**.
 - This lock therefore called a **spinlock**.
- **Positive:**
 - No **context switch (store and load PCBs)** is required when a process must wait on a lock, and a context switch may take **considerable time**.
 - Thus, when locks are expected to be held for **short times**, spinlocks are useful.
 - They are often employed on **multiprocessor** systems where one thread can “**spin**” on one processor while another thread performs its critical section on another processor.

So, many CPU cycles are wasted for the process to keep waiting.

Spinlock: Because the process keeps spinning until it acquires the lock.





acquire() and release()

- ```
acquire() {
 while (!available)
 ; /* busy wait */
 available = false; // To prevent other processes from
 // entering their critical section
}
```
- ```
release() {  
    available = true;  
}
```
- ```
do {
 acquire lock
 critical section
 release lock
 remainder section
} while (true);
```





**P<sub>0</sub>**

```

acquire() {
 while (!available)
 ; /*if true busy wait */
 available = false;
}

```

do {

*acquire lock*

*critical section*

*release lock*

*remainder section*

} while (true);

```

release() {
 available = true;
}

```

**P<sub>1</sub>**

do {

*acquire lock*


*critical section*

*release lock*

*remainder section*

} while (true);

Boolean Available = T

|                                                                         | Available | Effect            | P <sub>0</sub> | P <sub>1</sub>                                                                        |
|-------------------------------------------------------------------------|-----------|-------------------|----------------|---------------------------------------------------------------------------------------|
| initialization                                                          | T         |                   |                |                                                                                       |
| P <sub>0</sub> : acquire lock                                           |           |                   |                |                                                                                       |
| Function acquire() P <sub>0</sub> : while (!available) → while (!T) = F |           | C.S is available  |                |                                                                                       |
| Function acquire() P <sub>0</sub> : available = false                   | F         | Close C.S         |                |                                                                                       |
| P <sub>0</sub> : acquire lock                                           |           |                   | C.S            |                                                                                       |
| P <sub>1</sub> : acquire lock                                           |           |                   |                |                                                                                       |
| Function P <sub>1</sub> : while (!available) → while (!F) = T           |           | Stuck with loop   |                |  |
| P <sub>0</sub> : release lock                                           |           |                   |                |                                                                                       |
| Function release () P <sub>0</sub> : available = true                   | T         |                   |                |                                                                                       |
| Function acquire() P <sub>1</sub> : while (!available) → while (!T) = F |           | C.S is available  |                |                                                                                       |
| Function acquire() P <sub>1</sub> : available = false                   | F         | Close C.S         |                |                                                                                       |
| P <sub>1</sub> : acquire lock                                           |           | Enter to the C.S  |                | C.S                                                                                   |
| P <sub>1</sub> : release lock                                           |           |                   |                |                                                                                       |
| Function release () P <sub>1</sub> : available = true                   | T         | Now C.S available |                |                                                                                       |



# Semaphore

- **Synchronization** tool that provides more sophisticated ways (than **mutex locks**) for processes 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()**
    - **wait()** was originally called **P** (from the Dutch **Proberen**, “to test”);
    - **signal()** was originally called **V** (from **Verhogen**, “to increment”).

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

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

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

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





# Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain (0, 1, 2, ...). It can be used among various numbers of processes.
- **Binary semaphore** – integer value can range only between **0** and **1**.
  - Same as a **mutex lock** (between two processes only).
- 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**.
- In this example, always  $S_1$  is performed before  $S_2$
- That is, we force the execution  $S_1$  first, even if  $P_2$  comes first.

**P1:**

```
 S_1 ; //printf("Hello");
signal(synch);
```

**P2:**

```
wait(synch);
 S_2 ; //printf("PSUT");
```

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

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

Output

```
 S_1 Hello
 S_2 PSUT
```

- 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 since in this case the **busy wait** will be for a long time (wasteful of resources).

To get rid of the problem of busy waiting, instead of making a process "busy wait" for a semaphore, the process blocks and wait in **waiting queue** for the semaphore.



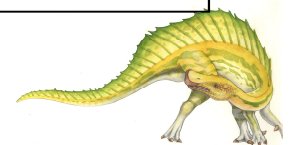


## Semaphore implementation with **no busy waiting**

- With each semaphore there is an associated **waiting queue** (e.g., FIFO).
- 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** (no need for busy waiting).
  - **wakeup** – remove one of processes in the waiting queue and place it in the **ready queue** (ready for scheduling and execution).

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

- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the **number of resources available**.
- In our case, we assume we have one instance is available, this is why the semaphore is **initialized to 1**.







## Implementation with no busy waiting (Cont.)

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

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

Each process that wishes to use a resource performs a **wait()** operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a **signal()** operation (incrementing the count).

Original wait and signal  
with busy waiting

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

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





```
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);
 }
}
```

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

**P<sub>0</sub>**

```
While (true){
 wait(&S)
 critical section
 print(x)
 Signal(&S)
 remainder section
}
```

**P<sub>1</sub>**

```
While (true){
 wait(&S)
 critical section
 print(y)
 Signal(&S)
 remainder section
}
```

| Example of two processes                                | value | Waiting Queue                                               | P <sub>0</sub> | P <sub>1</sub> |
|---------------------------------------------------------|-------|-------------------------------------------------------------|----------------|----------------|
| initialization                                          | 1     |                                                             |                |                |
| P <sub>0</sub> :While (true)                            |       |                                                             |                |                |
| Function wait() P <sub>0</sub> : S->value--;            | 0     |                                                             | C.S            |                |
| P <sub>1</sub> : While (true)                           |       |                                                             |                |                |
| Function P <sub>1</sub> : = S->value--;                 | -1    |                                                             |                |                |
| Function wait() P <sub>1</sub> :if (S->value < 0) → yes |       | Add P1 to Queue and Block                                   |                |                |
| P <sub>0</sub> :Signal (&S) S->value++                  | 0     | P0 is completed                                             |                |                |
| P <sub>0</sub> :if (S->value <= 0)→ yes                 |       | P <sub>1</sub> :remove a process P from S->list; wakeup(P); |                |                |
| critical section                                        |       | Now the C.S is available                                    |                | C.S            |



```
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);
 }
}
```

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

While (true){
 1→ wait(&S)
 2→ critical section
 5→ Signal(&S)
 6→ remainder section
} P0

While (true){
 4→ wait(&S)
 9→ critical section
 Signal(&S)
 remainder section
} P1

While (true){
 3→ wait(&S)
 7→ critical section
 8→ Signal(&S)
 remainder section
} P2
```

| Example of three processes                                              | Value | Waiting Queue                                                                                                      | P <sub>0</sub> | P <sub>1</sub> | P <sub>2</sub> |
|-------------------------------------------------------------------------|-------|--------------------------------------------------------------------------------------------------------------------|----------------|----------------|----------------|
| initialization                                                          | 1     |                                                                                                                    |                |                |                |
| 1→ P <sub>0</sub> :wait(&S)                                             |       |                                                                                                                    |                |                |                |
| 2→ Function wait()                                                      | 0     | P <sub>0</sub> : S->value--,if (S->value < 0)→ no                                                                  | C.S            |                |                |
| 3→: P <sub>2</sub> : wait(&S) : Function P <sub>2</sub> : = S->value--; | -1    |                                                                                                                    |                |                |                |
| Function wait() P <sub>2</sub> :if (S->value < 0) → yes                 |       | Add P <sub>2</sub> to Queue and Block                                                                              |                |                |                |
| 4→ wait(&S): Function P <sub>1</sub> : = S->value--;                    | -2    |                                                                                                                    |                |                |                |
| Function wait() P <sub>1</sub> :if (S->value < 0) → yes                 |       | Add P <sub>1</sub> to Queue and Block                                                                              |                |                |                |
| 5→ P <sub>0</sub> :Signal (&S) S->value++                               | -1    | P <sub>0</sub> is completed                                                                                        |                |                |                |
| P <sub>2</sub> :if (S->value <= 0)→ yes                                 |       | Remove P <sub>2</sub> from queue (P <sub>2</sub> ) and wakeup, P <sub>2</sub> will go to ready queue for execution |                |                |                |
| 7→ P <sub>2</sub> :critical section                                     |       | Now the C.S is available for P <sub>2</sub>                                                                        |                |                | C.S            |
| 8→ P <sub>2</sub> :Signal (&S) S->value++                               | 0     |                                                                                                                    |                |                |                |
| 8→ if (S->value <= 0)→ yes                                              |       | Remove P <sub>1</sub> from queue (P <sub>1</sub> ) and wakeup, P <sub>1</sub> will go to ready queue for execution |                |                |                |
| 9→ P <sub>1</sub> :critical section                                     |       | Now the C.S is available for P <sub>1</sub>                                                                        |                | C.S            |                |



# 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.

|         | $P_0$             | $P_1$             |
|---------|-------------------|-------------------|
| Step 1: | <i>wait(S);</i>   | <i>wait(Q);</i>   |
| Step 2: | <i>wait(Q);</i>   | <i>wait(S);</i>   |
|         | $\vdots$          | $\vdots$          |
| Step 3: | <i>signal(S);</i> | <i>signal(Q);</i> |
| Step 4: | <i>signal(Q)</i>  | <i>signal(S);</i> |





## Deadlock and Starvation (Cont.)

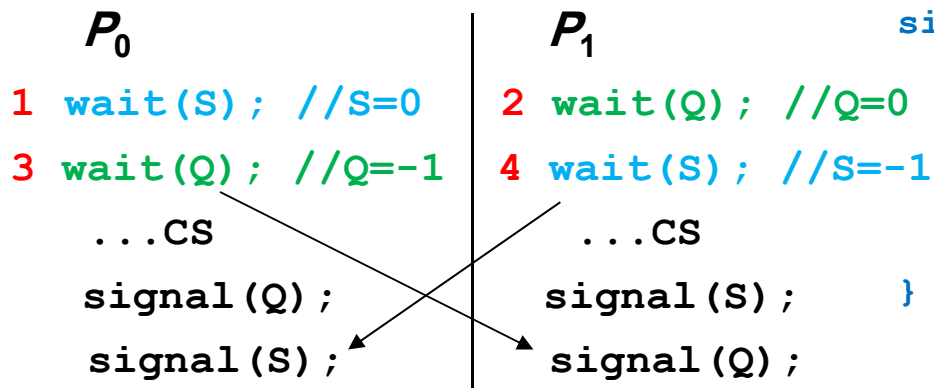
- **Step 1**: Suppose that P0 executes wait(S) and then P1 executes wait(Q).
- **Step 2**: When P0 executes wait(Q), it must wait until P1 executes signal(Q).
- **Step 2**: Also, when P1 executes wait(S), it must wait until P0 executes signal(S).
- **Step 3 & 4**: Since these signal operations cannot be executed, P0 and P1 are deadlocked.
- A set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set.





# 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 for two processes.  $P_0$   $P_1$



```

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

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

```

- Both  $P_0$  and  $P_1$  are waiting.
- Both signal(S) and signal(Q) will not be executed.
- **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**.





# Deadlock and Starvation

- Let  $S$  and  $Q$  be two semaphores initialized to 1 for two processes.

- Example: Starvation  $P_1$

$P_0$

```
1 wait(S); //S=0
 CS print("hello");
 signal(Q); // S=0
```

$P_1$

```
2 wait(S); //S=-1
 CS print("PSUT");
 signal(S);
```

S **P1**

Q

- $P_1$  will starve, since there is no **signal(S)** in  $P_0$ .

```
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);
 }
}
```



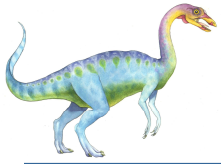


# Priority-Inheritance Protocol

- **As an example**, assume we have three processes (L, M, and H) whose priorities follow the order  $L < M < H$  (So, H has highest priority).
- Assume that **process H** requires **resource R**, which is currently being accessed by process L. Ordinarily, process H would wait for L to finish using resource R. However, now suppose that process M becomes runnable, thereby preempting process L. Indirectly, a process with a lower priority (process M) has affected how long process H must wait for L to relinquish (give up) resource R (**Priority Inversion Problem**).
- According to **priority-inheritance protocol**, all processes that are accessing resources needed by a higher-priority process inherit (get) the higher priority until they are finished with the resources in question. When they are finished, their priorities revert (return) to their original values.
- In the example above, a priority-inheritance protocol would allow process L to temporarily inherit the priority of process H, thereby preventing process M from preempting its execution. When process L had finished using resource R, it would relinquish (give up) its inherited priority from H and assume its original priority. Because resource R would now be available, process H (not M) would run next.





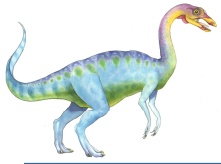


# Classical Problems of Synchronization

---

- Classical problems used to test newly-proposed synchronization schemes:
  1. Bounded-Buffer Problem.
  2. Readers and Writers Problem.
  3. Dining-Philosophers Problem.





# 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

$n=10$



```
do {
 ...
 /* produce an item in next_produced */
 ...
 wait(empty); //initially = n
 wait(mutex); //initially = 1
 ...
 CS /* add next produced to the buffer */
 ...
 signal(mutex);
 signal(full); //initially = 0
} while (true);
```

Full

Empty=0  
Full=n

Empty

Empty=n  
Full=0

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

$n$  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  $n$

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



# Bounded Buffer Problem (Cont.)

- The structure of the **consumer** process

$n=10$

Do {

```
wait(full); //initially = 0
```

```
wait(mutex); //initially = 1
```

...

```
CS /* remove an item from buffer to next_consumed */
```

...

```
signal(mutex);
```

```
signal(empty); //initially = n
```

...

```
/* consume the item in next consumed */
```

...

```
} while (true);
```

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

$n$  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  $n$

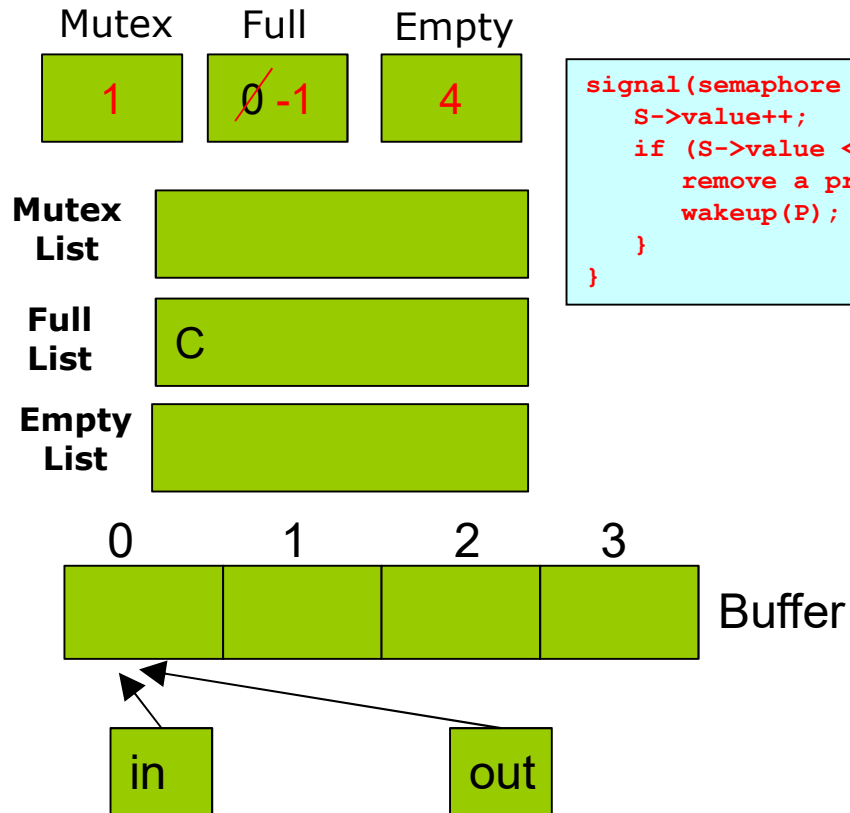
```
signal(semaphore *S) {
 S->value++;
 if (S->value <= 0) {
 remove a process P from S->list;
 wakeup(P);
 }
}
```



# Consume when buffer empty

Assume initially buffer is empty and buffer size is 4.

Also, assume **Consumer (C)** tries to consume an item when buffer is empty.



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

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

```
// Consumer Process
Do {
 wait(full); //initially = 0
 wait(mutex); //initially = 1
 ...
 /* remove an item from buffer
 to next_consumed */
 ...
 signal(mutex);
 signal(empty); //initially = n
 ...
 /* consume the item in next
 consumed */
 ...
} while (true);
```



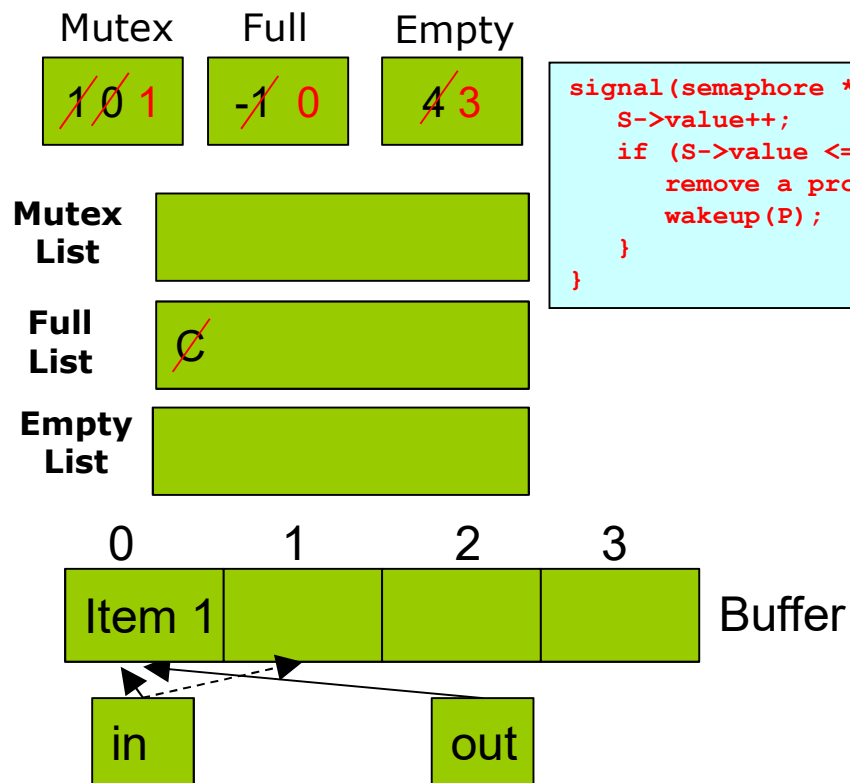
Can consumer consume item?

- No, call to **wait(full)** causes full semaphore to be -1 and process added to full queue.



# Produce one item

Assume **Producer (P)** produces one item.



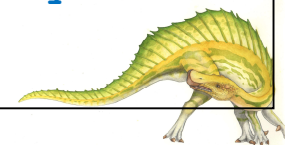
Can producer produce item?

- Call to **wait(empty)** causes empty semaphore to be 3, no waiting.
- Call to **wait(mutex)** causes mutex semaphore to be 0, no wait. Now, add item to buffer.
- **Signal(mutex)** and **signal(full)** which causes C to stop waiting and be removed from full queue.

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

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

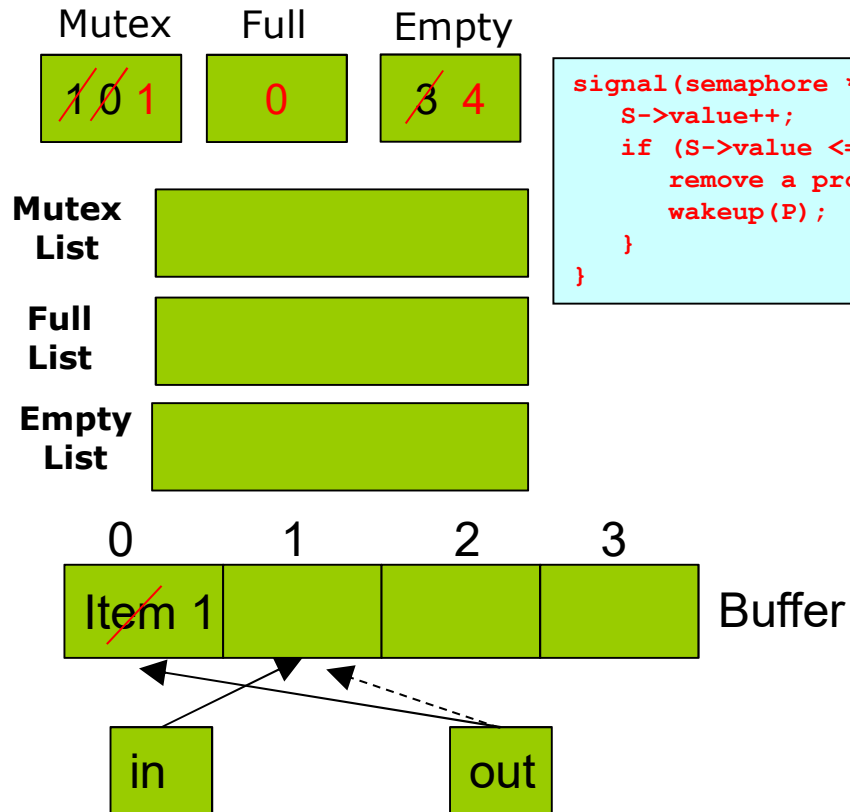
```
// Producer Process
do {
 ...
 /* produce an item in next_produced
*/
 ...
 wait(empty); //initially = n
 wait(mutex); //initially = 1
 ...
 /* add next produced to the
buffer */
 ...
 signal(mutex);
 signal(full); //initially = 0
} while (true);
```





# Consume an item

Assume **Consumer (C)** consumes one item.



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

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

```
// 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);
```

Can consumer consume item?

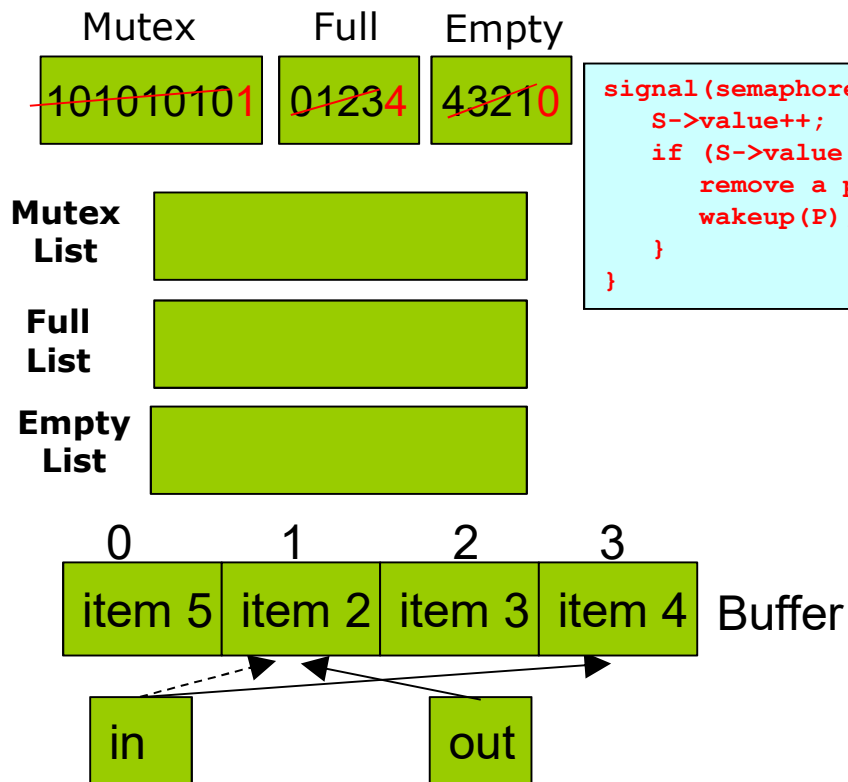
- After consumer resumes its work, it calls **wait(mutex)**, then it consumes an item from buffer.
- Then, call **signal(mutex)**, **signal(empty)**.





# Produce 4 items

Assume **Producer (P)** produces 4 items.



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

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

```
// 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);
```

No consumers now, produce 4 items?

- Everything happens normally. No waiting.
- At the end buffer is full. So,
  - Full semaphore = 4.
  - Empty semaphore = 0.

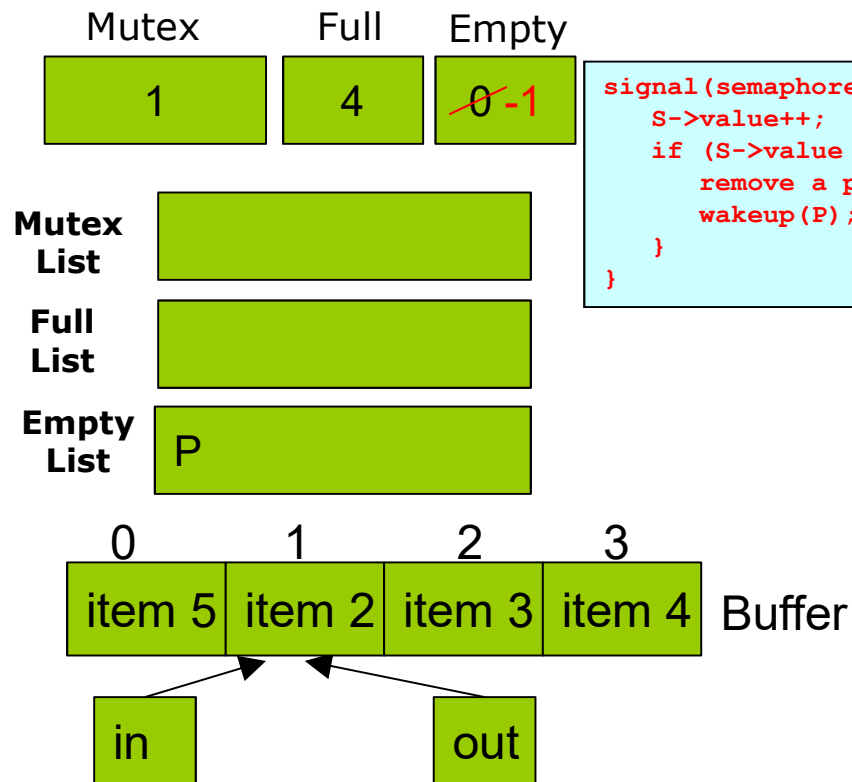






# Produce one item

Assume **Producer (P)** produces one item when buffer is full.



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

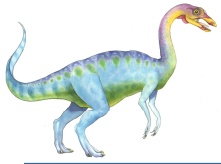
```
wait(semaphore *S) {
 S->value--;
 if (S->value < 0) {
 add this process to S->list;
 block();
 }
}
```

```
// 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);
```

Can producer produce item 6?

- Buffer is full. Call to **wait(empty)** causes producer to wait in empty queue.





# 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**).
- **First variation** – no reader kept waiting unless writer has permission to use shared object (**We will only consider this**).
- **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**.

In the **first variation**. **Writers may starve** because as long as there are concurrent reads that keeps coming, writers will have to wait indefinitely.

In the **second variation**. **Readers may starve** because as long as writers keeps coming, readers will have to wait indefinitely.





# Readers-Writers Problem (Cont.)

- We will consider the **First variation** – no reader kept waiting unless writer has permission to use shared object.

- Shared Data:

- Data set.
- Semaphore **rw\_mutex** initialized to 1.
- Semaphore **mutex** initialized to 1.
- Integer **read\_count** initialized to 0.

This is applied in **databases** because many users are allowed to **read** shared data concurrently. However, only one user can **write** to shared data on database.

The idea is as long as one reader is reading the shared data, then it is OK to let other readers read the data at the same time.

The **mutex semaphore** is used to ensure mutual exclusion when the variable **read\_count** is updated.

The **read\_count variable** keeps track of how many processes are currently reading the object.

The **semaphore rw\_mutex** functions as a mutual exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections.





# Readers-Writers Problem (Cont.)

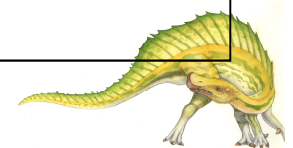
- The structure of a **writer** process:

```
do {
 wait(rw_mutex); //initially 1

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

```
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);
 }
}
```





# Readers-Writers Problem (Cont.)

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

```
do {
 wait(mutex); //initially = 1
 read_count++; //initial = 0

 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);
```

```
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);
 }
}
```

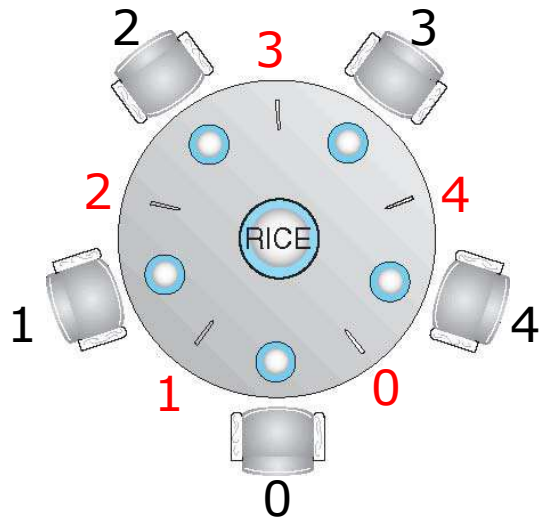
**First reader:** If no reader is reading the shared data and the writer is not currently writing on the shared data, then the reader can read data.

Also, if one reader is currently reading shared data, this means that writer is not currently writing. So, other concurrent readers are allowed to read data immediately.





# Dining-Philosophers Problem



- One chopstick between any two philosophers.
- Chopsticks are picked one at a time.
- Two neighbors cannot both eat at the same time because only one of them can use the chopstick between them.
- How to synchronize philosophers' behavior so that no two neighbors are eating synchronously?

- 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.

5 semaphores. One for each chopstick.





# Dining-Philosophers Problem Algorithm

## ■ The structure of **Philosopher i**

do {

2S-available: 2 chopsticks available

//Wait(2S-available)

Pick a chopstick

wait (chopstick[i]);

wait (chopstick[ (i + 1) % 5] );

Pick another chopstick

//Signal(2S-available)

CS // eat

Put a chopstick

signal (chopstick[i] );

signal (chopstick[ (i + 1) % 5] );

Put another chopstick

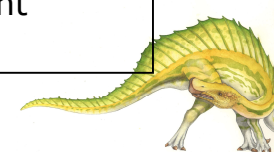
// think

} while (TRUE);

## ■ What is the problem with this algorithm?

- Circular, so that we get the correct index of chopstick.
- Example: For philosopher 4, he needs:
  - Chopstick 4
  - Chopstick  $(4+1)\%5 = 0$

- The problem in this solution is **deadlock**. Consider when each philosopher executes the first wait for chopstick which allows him to pick the chopstick to his left.
- Each philosopher would be waiting for the chopstick to his right
- No one will ever eat.





## Dining-Philosophers Problem Algorithm (Cont.)

---

- **Deadlock handling (No deadlock):**
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the forks 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.
- **Note:** A deadlock-free solution does not necessarily eliminate the possibility of starvation.





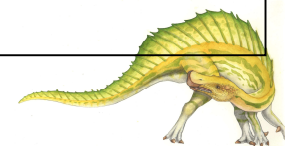


# Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) .... wait (mutex)
  - wait (mutex) ... wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation are possible.

```
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);
 }
}
```





# 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
- 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 (...) { ... }

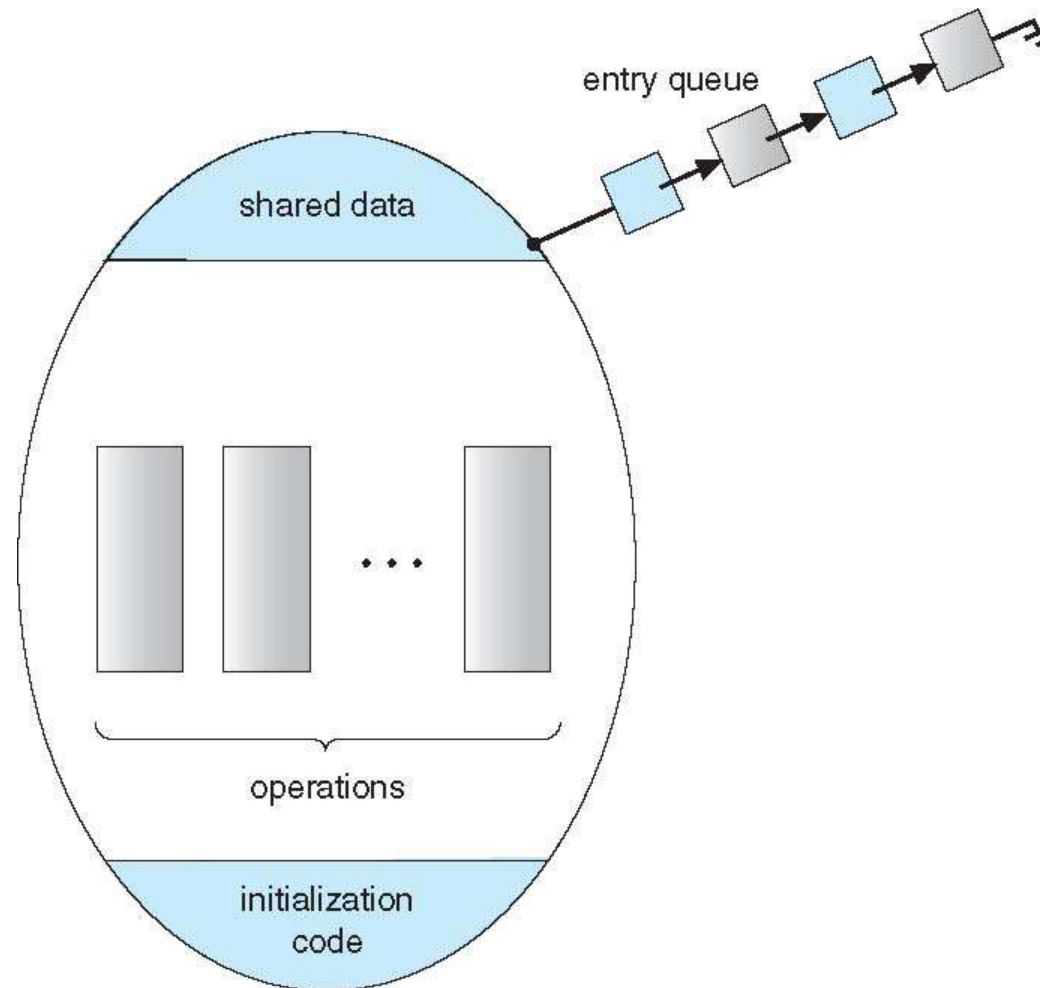
 procedure Pn (...) {.....}

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





# Schematic view of a Monitor





# Condition Variables

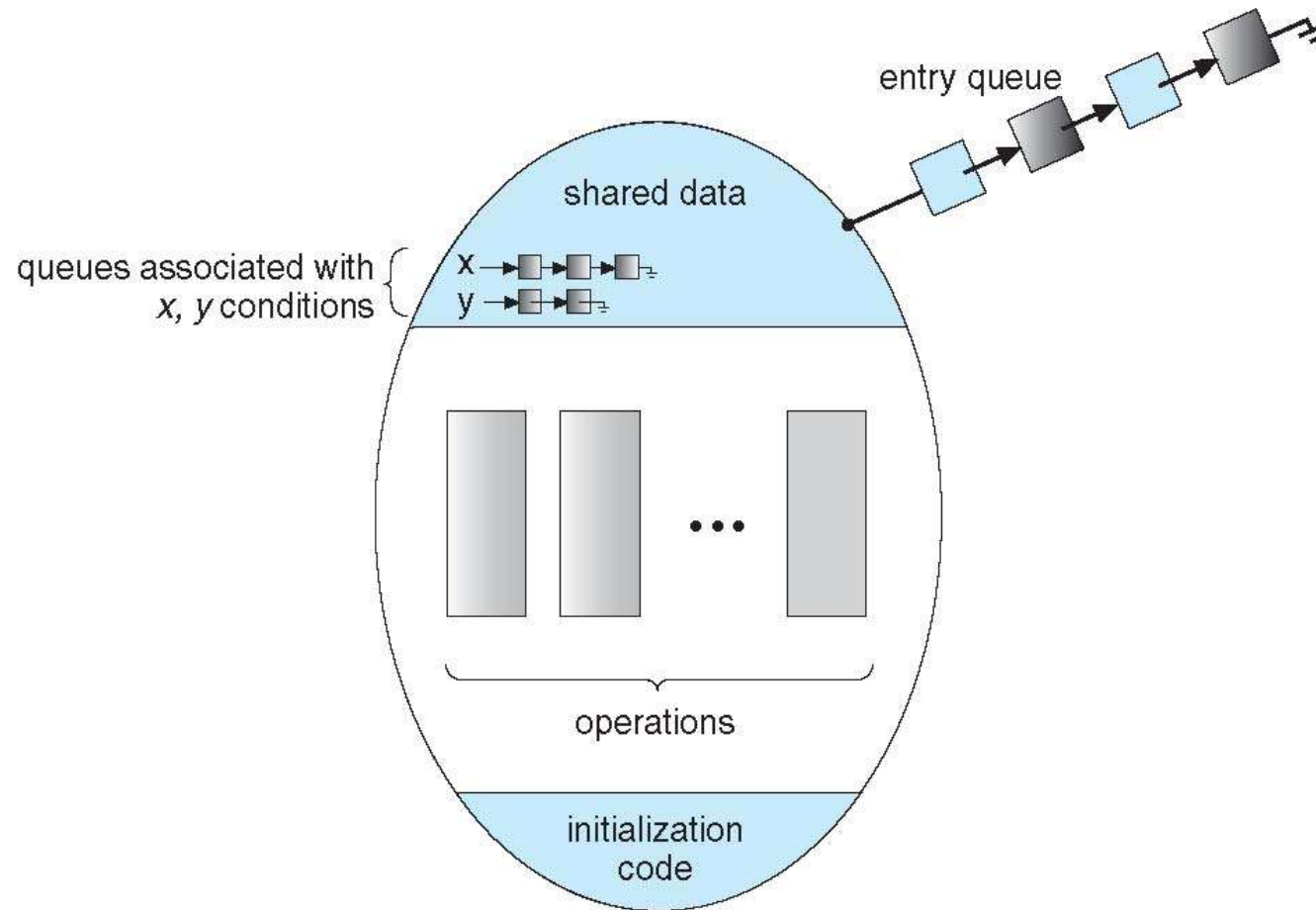


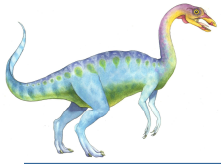
- **condition  $x$ ,  $y$ ;**
- Two operations are allowed on a condition variable:
  - **$x.\text{wait}()$**  – a process that invokes the operation is suspended until  **$x.\text{signal}()$**
  - **$x.\text{signal}()$**  – resumes one of processes (if any) that invoked  **$x.\text{wait}()$** 
    -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





# Monitor with Condition Variables



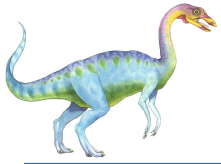


# Condition Variables Choices



- 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.)

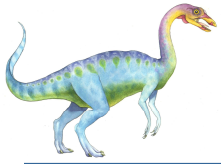


```
void test (int i) {
 if ((state[(i + 4) % 5] != EATING) &&
 (state[i] == HUNGRY) &&
 (state[(i + 1) % 5] != EATING)) {
 state[i] = EATING ;
 self[i].signal () ;
 }
}

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





# Monitor Implementation Using Semaphores



- Variables

```
semaphore mutex; // (initially = 1)
semaphore next; // (initially = 0)
int next_count = 0;
```

- Each procedure  $F$  will be replaced by

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

- Mutual exclusion within a monitor is ensured





# Monitor Implementation – Condition Variables



- For each condition variable  $x$ , we have:

```
semaphore x_sem; // (initially = 0)
int x_count = 0;
```

- The operation  $x.\text{wait}$  can be implemented as:

```
x_count++;
if (next_count > 0)
 signal(next);
else
 signal(mutex);
wait(x_sem);
x_count--;
```





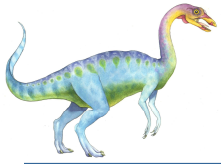
# Monitor Implementation (Cont.)



- The operation `x.signal` can be implemented as:

```
if (x_count > 0) {
 next_count++;
 signal(x_sem);
 wait(next);
 next_count--;
}
```





# Resuming Processes within a Monitor



- If several processes queued on condition  $x$ , and  $x.\text{signal}()$  executed, which should be resumed?
- FCFS frequently not adequate
- **conditional-wait** construct of the form  $x.\text{wait}(c)$ 
  - Where  $c$  is **priority number**
  - Process with lowest number (highest priority) is scheduled next





# Single Resource allocation

- Allocate a single resource among competing processes using priority numbers that specify the maximum time a process plans to use the resource

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

R.release ;
```

- Where R is an instance of type **ResourceAllocator**





# A Monitor to Allocate Single Resource



```
monitor ResourceAllocator
{
 boolean busy;
 condition x;
 void acquire(int time) {
 if (busy)
 x.wait(time);
 busy = TRUE;
 }
 void release() {
 busy = FALSE;
 x.signal();
 }
 initialization code() {
 busy = FALSE;
 }
}
```





# Synchronization Examples

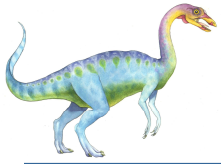
---



- ☐ Solaris
- ☐ Windows
- ☐ Linux
- ☐ Pthreads





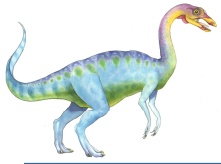


# Solaris Synchronization



- ❑ 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



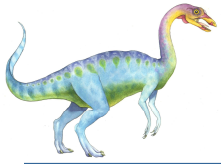


# Windows Synchronization



- 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



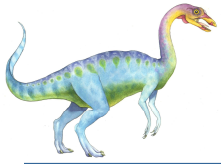


# Pthreads Synchronization



- 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





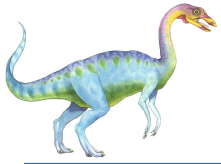
# Transactional Memory

- 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

---

