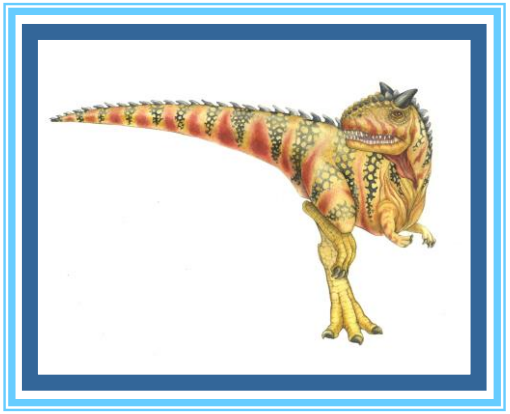
Module 3

Chapter 5: Process Synchronization



Course Objective

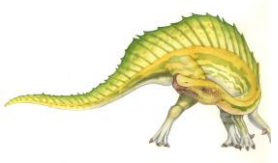
- CO3

Explain process synchronization in Operating Systems and illustrate process synchronization mechanisms using

  - Mutex Locks
  - Semaphores
  - Monitors

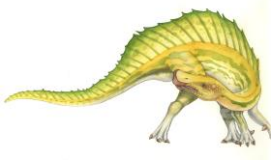
- CO4

Explain any one method for detection, prevention, avoidance and recovery for managing deadlocks in Operating Systems.



Chapter 5: Process Synchronization

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



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

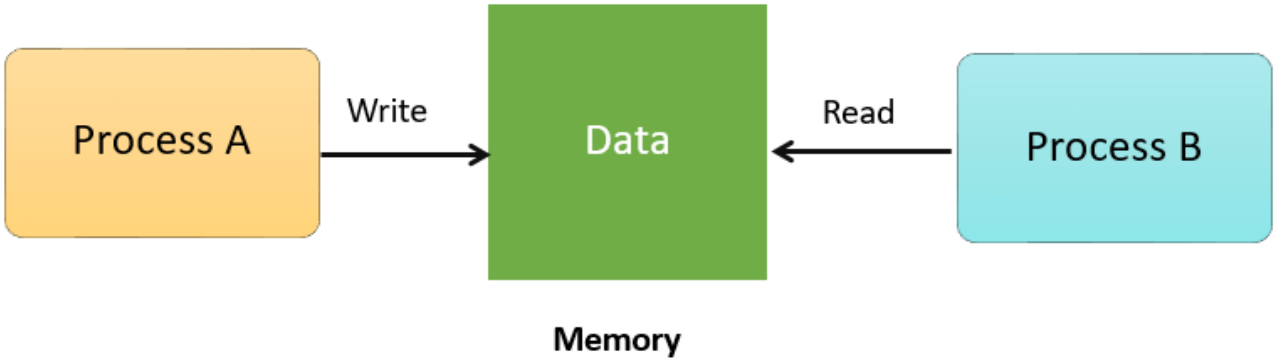
Background

- **Process Synchronization** is a technique which is used to coordinate the process that use shared Data.
- There are two types of Processes in an Operating Systems:-
- **Independent Process –** The process that does not affect or is affected by the other process while its execution then the process is called Independent Process.
  - Example :The process that does not share any shared variable, database, files, etc.
- **Cooperating Process –** The process that affect or is affected by the other process while execution, is called a Cooperating Process.
  - Example: The process that share file, variable, database

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
  - It is specially needed in a multi-process system when multiple processes are running together, and more than one processes try to gain access to the same shared resource or data at the same time.
- Concurrent access to shared data may result in data inconsistency
- So the change made by one process not necessarily reflected when other processes accessed the same shared data.
- To avoid this type of inconsistency of data, the processes need to be synchronized with each other.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Background



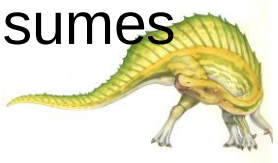
- Illustration of the problem:

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.

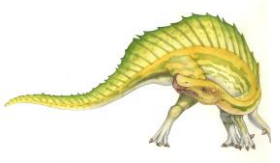
Initially, **`counter`** is set to 0.

It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.



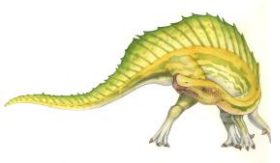
Producer

```
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
```

Race Condition

- Consider this execution interleaving with "count = 5" initially:

  S0: producer execute `register1 = counter`
  {register1 = 5}

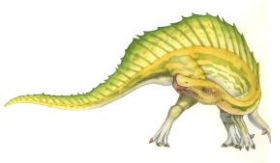
  S1: producer execute `register1 = register1 + 1`
  {register1 = 6}

  S2: consumer execute `register2 = counter`
  {register2 = 5}

  S3: consumer execute `register2 = register2 - 1`
  {register2 = 4}

  S4: producer execute `counter = register1`
  {counter = 6 }

  S5: consumer execute `counter = register2`
  {counter = 4}



Race Condition

- Consider this execution interleaving with "count = 5" initially:

  S0: producer execute `register1 = counter`
  {register1 = 5}

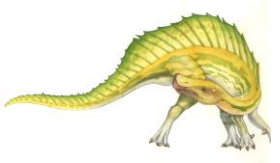
  S1: producer execute `register1 = register1 + 1`
  {register1 = 6}

  S2: consumer execute `register2 = counter`
  {register2 = 5}

  S3: consumer execute `register2 = register2 - 1`
  {register2 = 4}

  S4: consumer execute `counter = register2`
  {counter = 4}

  S5:producer execute `counter = register1`
  {counter = 6 }



Race Condition

- Consider this execution interleaving with "count = 5" initially:

  S0: producer execute `register1 = counter`
  {register1 = 5}

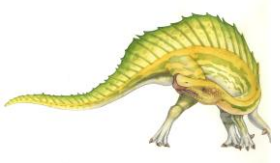
  S1: producer execute `register1 = register1 + 1`
  {register1 = 6}

  S2:producer execute `counter = register1`
  {counter = 6 }

  S3: consumer execute `register2 = counter`
  {register2 = 6}

  S4: consumer execute `register2 = register2 - 1`
  {register2 = 5}

  S5: consumer execute `counter = register2`
  {counter = 5}



Race Condition

- It is the **condition** where several **processes** tries to access the resources and modify the shared data concurrently and outcome of the **process** depends on the particular order of execution that leads to data inconsistency, this **condition** is called **Race Condition**.

Types of Race Condition

- Race conditions can occur when two or more threads read and write the same variable according to one of these two patterns:
- Read-modify-write
- Check-then-act
- The read-modify-write pattern means, that two or more threads first read a given variable, then modify its value and write it back to the variable.
- For this to cause a problem, the new value must depend one way or another on the previous value.
- The problem that can occur is, if two threads read the value (into CPU registers) then modify the value (in the CPU registers) and then write the values back.

Check-then-act

- The check-then-act pattern means, that two or more threads check a given condition, for instance if a Map contains a given value, and then go on to act based on that information, e.g. taking the value from the Map.
- The problem may occur if two threads check the Map for a given value at the same time - see that the value is present - and then both threads try to take (remove) that value.
- However, only one of the threads can actually take the value.
- The other thread will get a null value back.
- This could also happen if a Queue was used instead of a Map.

Critical Section Problem

- Consider system of ***n*** processes {$p_0, p_1, \ldots p_{n-1}$}
- 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**

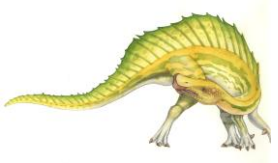
Critical Section

- General structure of process P_i

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

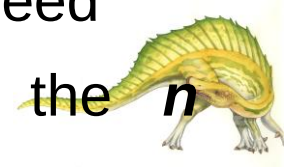
Algorithm for Process P_i

```
do {
    while (turn == j);

        critical section

    turn = j;

        remainder section

} while (true);
```



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 only those processes that are not executing in their remainder section can participate in deciding which will enter its CS and this 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

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

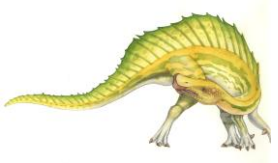
```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn = = j);

        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

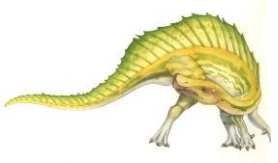


Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions 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);
```

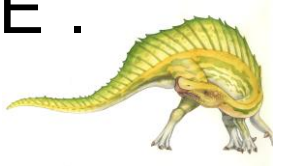


test_and_set Instruction

Definition:

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

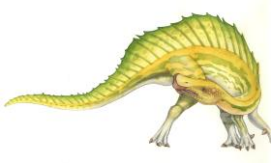
1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.



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:

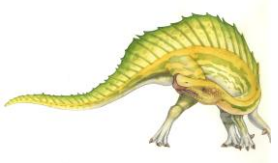
```
    int compare_and_swap(int *value, int
expected, int new_value) {
        int temp = *value;
        if (*value == expected)
           *value = new_value;
        return temp;
    }
```

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:

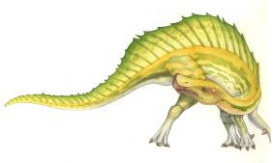
```
do {
   while(compare_and_swap(&lock,0,1)!= 0)
          ; /* do nothing */
      /* critical section */
    lock = 0;
      /* remainder section */
  } while (true);
```



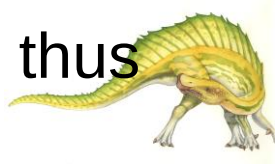
Bounded-waiting Mutual Exclusion with test_and_set

```
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);
```

- The data structures boolean waiting[n]; boolean lock; are initialized to false.
- Process *Pi* can enter its critical section only if either waiting[i] == false or key == false.
- The value of key can become false only if the test and set() is executed.
- The first process to execute the test and set() will find key == false; all others must wait.
- The variable waiting[i] can become false only if another process leaves its critical section; only one waiting[i] is set to false, maintaining the mutual-exclusion requirement.



- To prove that the progress requirement
- A process exiting the critical section either sets lock to false or sets waiting[j] to false.
- Both allow a process that is waiting to enter its critical section to proceed.
- To prove that the bounded-waiting requirement
- When a process leaves its critical section, it scans the array waiting in the cyclic ordering ($i + 1, i + 2, ..., n - 1, 0, ..., i - 1$).
- It designates the first process in this ordering that is in the entry section (waiting[j] == true) as the next one to enter the critical section.
- Any process waiting to enter its critical section will thus do so within $n - 1$ turns.



Mutex Locks

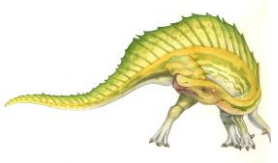
- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - Boolean variable indicating 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**

acquire() and release()

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

```
release() {
 available = true;
     }
```

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



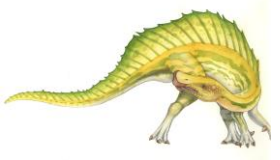
- Spinlocks do have an advantage, in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time.
- Thus, when locks are expected to be held for short times, spinlocks are useful.
- They are often employed on multiprocessor systems where one thread can “spin” on one processor while another thread performs its critical section on another processor.

Semaphores

- semaphore = a synchronization primitive
  - higher level than locks
  - invented by Dijkstra in 1968, as part of the THE os
- A semaphore is:
  - a variable that is manipulated atomically through two operations, signal and wait
  - wait(semaphore): decrement, block until semaphore is open
    - also called P(), after Dutch word for test, also called down()
  - signal(semaphore): increment, allow another to enter
    - also called V(), after Dutch word for increment, also called up()

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()`** **`proberen`** and **`V()`** **`verhogen`**



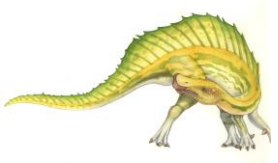
Semaphore

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

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

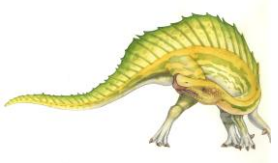
- Definition of the **`signal()` operation**

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



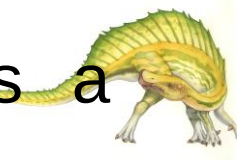
Types of Semaphore

- Counting semaphore (aka counted semaphore)
- **Counting semaphore** – integer value can range over an unrestricted domain
  - represents a resources with many units available
  - allows threads/process to enter as long as more units are available
  - counter is initialized to N
    - N = number of units available



Types of Semaphore

- **Binary semaphore** (aka mutex semaphore)
  - guarantees mutually exclusive access to resource
  - only one thread/process allowed entry at a time
  - counter is initialized to 1
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Semaphore Usage
  - Can solve various synchronization problems
- Can implement a counting semaphore ***S*** as a binary semaphore



Semaphore Usage

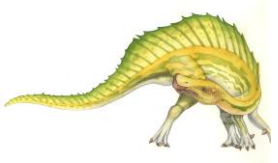
- Consider P_1 and P_2 that require S_1 to happen before S_2

Create a semaphore "**synch**" initialized to 0

```
P1:                          P2 :

S1;                          wait(synch);
signal(synch);               S2;
```

Semaphores

- Before entering critical section
  - **`Wait(s)`**
    - Receive signal via semaphore **`s`**
    - “down” on the semaphore - Also: **`P`** – proberen
- After finishing critical section
  - **`Signal(s)`**
    - Transmit signal via semaphore **`s`**
    - “up” on the semaphore - Also: **`V`** – verhogen
- Implementation requirements
  - **`Signal`** and **`Wait`** must be atomic



Semaphore Implementation

- Requirement
  - No two processes can execute **`wait()`** and **`signal()`** on the same semaphore at the same time!
- Critical section
  - **`wait()`** and **`signal()`** code
  - Now have busy waiting in critical section implementation
    - + Implementation code is short
    - + Little busy waiting if critical section rarely occupied
    - – Bad for applications may spend lots of time in critical sections

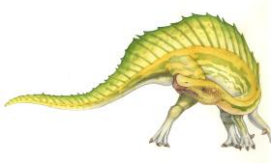
Semaphore Implementation with no Busy waiting

- 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);
    }
}
```

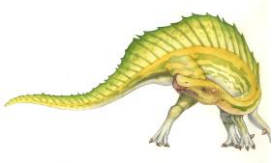


Blocking in Semaphores

- Each semaphore has an associated queue of processes/threads
  - when wait() is called by a thread,
    - if semaphore is “available”, thread continues
    - if semaphore is “unavailable”, thread blocks, waits on queue
  - signal() opens the semaphore
    - if thread(s) are waiting on a queue, one thread is unblocked
    - if no threads are on the queue, the signal is remembered for next time a wait() is called
- In other words, semaphore has history
  - this history is a counter
  - if counter falls below 0 (after decrement), then the semaphore is closed
    - wait decrements counter
    - signal increments counter

Mutual Exclusion Using Semaphores

```
semaphore s = 1;
Pi {
  while(1)     {
     Wait(s);
     /* Critical Section */
     Signal(s);
     /* remainder */
  }
}
```



Semaphores vs. Test_and_Set

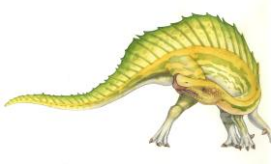
Semaphore

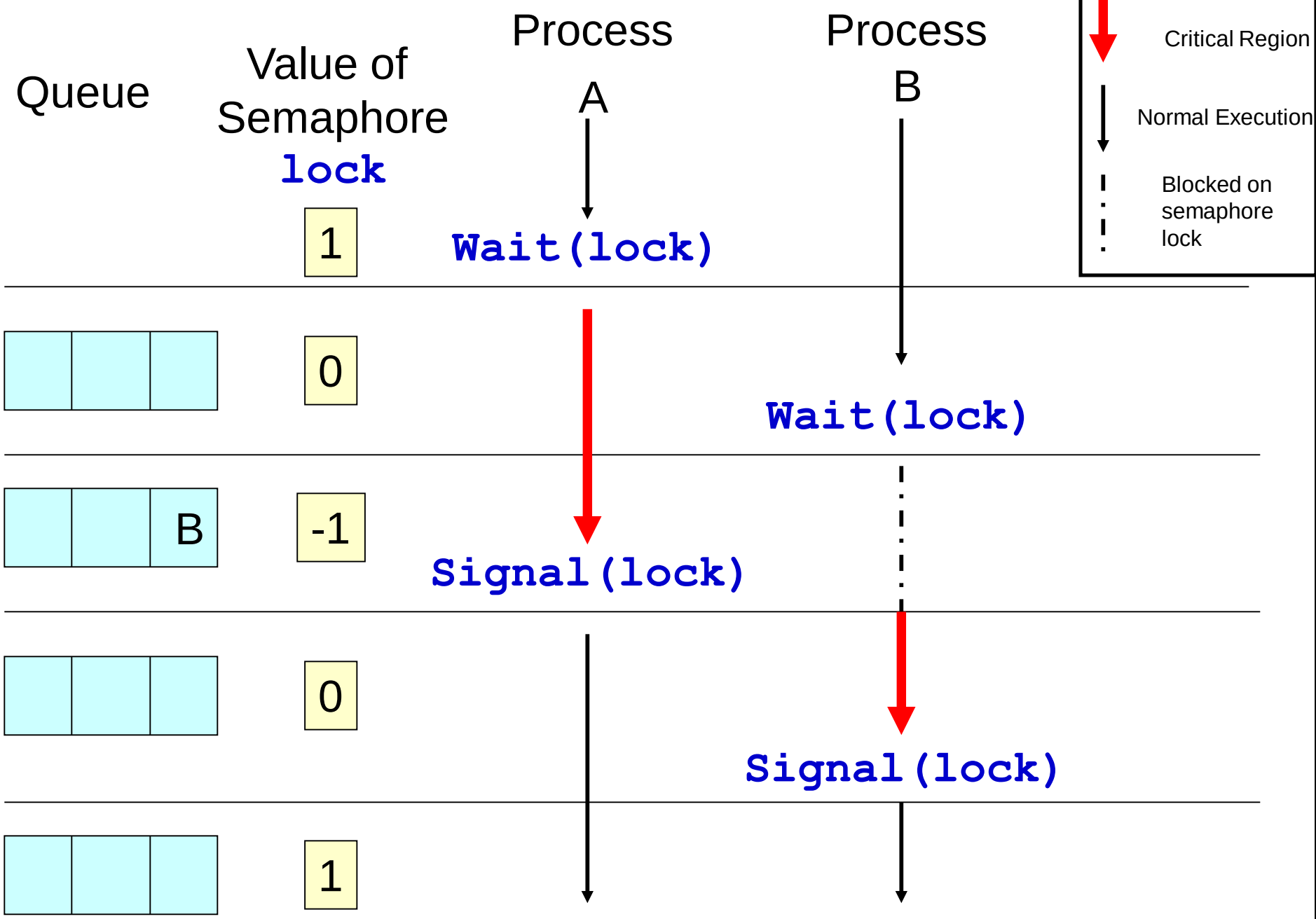
```
semaphore s = 1;
Pi {
   while(1) {
      Wait(s);
      /* CS */
      Signal(s);
      /* remainder */
   }
}
```

Test_and_Set

```
lock = 0;
Pi {
   while(1) {
   while(Test_And_Set(lock))
   ;
      /* CS */
      lock =0;
      /* remainder */
   }
}
```

- Avoid busy waiting by suspending
  - Block if **`s == False`**
  - Wakeup on signal (**`s = True`**)




Queue
Value of Semaphore
lock
Process A
Process B
Critical Region
Normal Execution
Blocked on semaphore lock
1
Wait(lock)
0
Wait(lock)
B
-1
Signal(lock)
0
Signal(lock)
1

Semaphore Example 1

```
semaphore s = 2;
P_i {
  while(1)     {
     Wait(s);
     /* CS */
     Signal(s);
     /* remainder */
  }
}
```

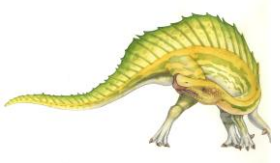
- What happens?
- When might this be desirable?

Semaphore Example 1

```
semaphore s = 2;
Pi {
   while(1)      {
      Wait(s);
      /* CS */
      Signal(s);
      /* remainder */
   }
}
```

s = 2 ~~1~~ ~~0~~ -1

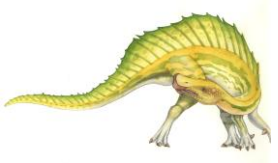
- What happens?
- When might this be desirable?



Semaphore Example 1

```
semaphore s = 2;
Pi {
   while(1)  {
        Wait(s);
        /* CS */
        Signal(s);
        /* remainder */
   }
}
```

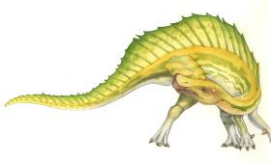
- What happens?
  - Allows up to 2 processes to enter CS
- When might this be desirable?
  - Need up to 2 processes inside CS
    - e.g., limit number of processes reading a var
  - Be careful not to violate mutual exclusion inside CS!



Semaphore Example 2

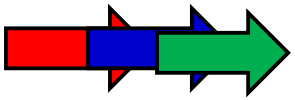
```
semaphore s = 0;
Pi {
   while(1)  {
        Wait(s);
        /* CS */
        Signal(s);
        /* remainder */
   }
}
```

- What happens?
- When might this be desirable?



Semaphore Example 2

```
semaphore s = 0;
Pi {
   while(1)  {
      Wait(s);
      /* CS */
      Signal(s);
      /* remainder */
   }
}
```



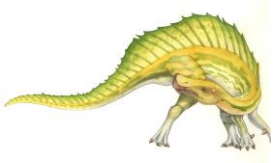
- What happens?
- When might this be desirable?

s = 0 ~~-1~~ ~~-2~~ -3

Semaphore Example 2

```
semaphore s = 0;
Pi {
   while(1)  {
        Wait(s);
        /* CS */
        Signal(s);
        /* remainder */
   }
}
```

- What happens?
  - No one can enter CS! Ever!
- When might this be desirable?
  - Never!



Semaphore Example 3

```
semaphore s = 0;
P1 {
   /* do some stuff */
   Wait(s);
   /* do some more stuff */
}
```

```
semaphore s; /* shared */
P2 {
   /* do some stuff */
   Signal(s);
   /* do some more stuff */
}
```

- What happens?

- When might this be desirable?

Semaphore Example 3

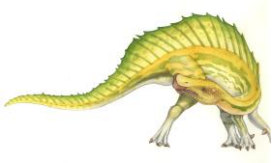
```
semaphore s = 0;
P1 {
   /* do some stuff */
   Wait(s);
   /* do some more stuff */
}
```

```
semaphore s; /* shared */
P2 {
   /* do some stuff */
   Signal(s);
   /* do some more stuff */
}
```

- What happens?

s = ~~1~~ ~~0~~ 1

- When might this be desirable?



Semaphore Example 3

```
semaphore s = 0;
P1 {
   /* do some stuff */
   Wait(s);
   /* do some more stuff */
}
```

```
semaphore s; /* shared */
P2 {
   /* do some stuff */
   Signal(s);
   /* do some more stuff */
}
```

- What happens?
  - P1 waits until P2 signals
  - if P2 signals first, P1 does not wait
- When might this be desirable?
  - Having a process/thread wait for another process/thread

Semaphore Example 4

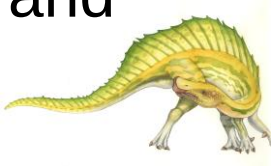
Process 1 executes:

```
while(1) {
    Wait(S);
    a;
    Signal(Q);
}
```

Process 2 executes:

```
while(1) {
    Wait(Q);
    b;
    Signal(S);
}
```

- Two processes
  - Two semaphores: S and Q
  - Protect two critical variables ‘a’ and ‘b’.
- What happens in the pseudocode if Semaphores S and Q are initialized to 1 (or 0)?



Semaphore Example 4

Process 1 executes:

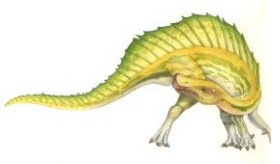
```
while(1) {
    Wait(S);
    a;
    Signal(Q);
}
```

S = 0 ✗ -1

Q = 0 ✗ -1

Process 2 executes:

```
while(1) {
    Wait(Q);
    b;
    Signal(S);
}
```



Semaphore Example 4

Process 1 executes:

```
while(1) {
    Wait(S);
    a;
    Signal(Q);
}
```

S = ~~1~~ ~~0~~ ~~1~~ 0

Q = ~~1~~ ~~0~~ ~~1~~ 0

Process 2 executes:

```
while(1) {
    Wait(Q);
    b;
    Signal(S);
}
```

Semaphore Example 4

Process 1 executes:

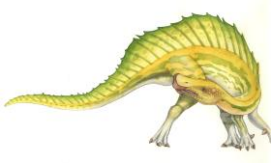
```
while(1) {
    Wait(S);
    a;
    Signal(Q);
}
```

Process 2 executes:

```
while(1) {
    Wait(Q);
    b;
    Signal(S);
}
```

S = 1 0 ~~-1~~ 0 1

Q = 1 0 ~~1~~ 0



Be careful!

Deadlock or Violation of Mutual Exclusion?

1
```
semSignal(s);
critical_section();
semWait(s);
```

2
```
semWait(s);
critical_section();
```

3
```
critical_section();
semSignal(s);
```

4
```
semWait(s);
critical_section();
semWait(s);
```

5
```
semWait(s);
semWait(s);
critical_section();
semSignal(s);
semSignal(s);
```

Deadlock and Starvation

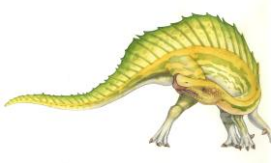
- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let ***S*** and ***Q*** be two semaphores initialized to 1

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`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**

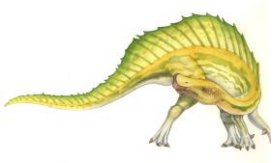
Classical Problems of Synchronization

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



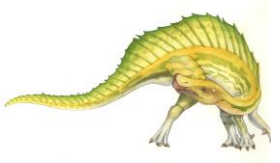
Bounded-Buffer Problem

- ***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
- "Full" keeps track of number of items in the buffer at any given time
- "Empty" keeps track of number of unoccupied slots



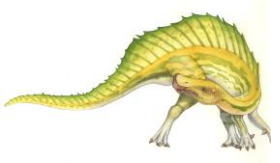
Bounded Buffer Problem - Producer

```
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- Consumer

```
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);
```



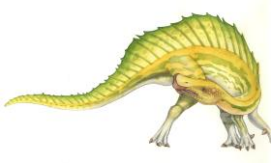
Readers-Writers Problem

- A data set is shared among a num 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.)

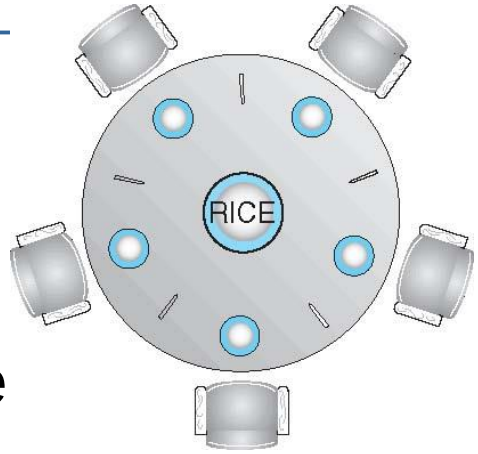
- The structure of a reader process

```
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);
```

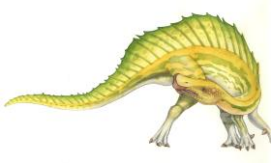
Readers-Writers Problem Variations

- ***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

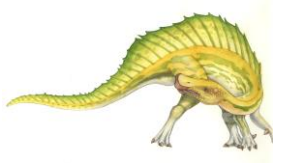
- The structure of Philosopher *i*:

```
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 forks only if both are available (picking must be done in a critical section.
  - Use an asymmetric solution
    - 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.

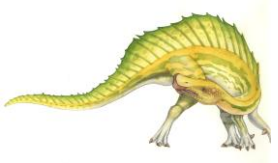


Problems with Semaphores

- Incorrect use of semaphore operations:

Process i	Process j	Process k	Process m
P(S)	V(S)	P(S)	P(S)
CS	CS	CS	if(something or other)
P(S)	V(S)		return;
			CS
			V(S)

- Deadlock and starvation are possible.



- Semaphores are very “low-level” primitives
- Users could easily make small errors
- Similar to programming in assembly language
- Small error brings system to grinding halt
- Very difficult to debug
- Also, we seem to be using them in two ways
- For mutual exclusion, the “real” abstraction is a critical section
- But the bounded buffer example illustrates something different, where threads “communicate” using semaphores
- Simplification: Provide concurrency support in compiler Monitors

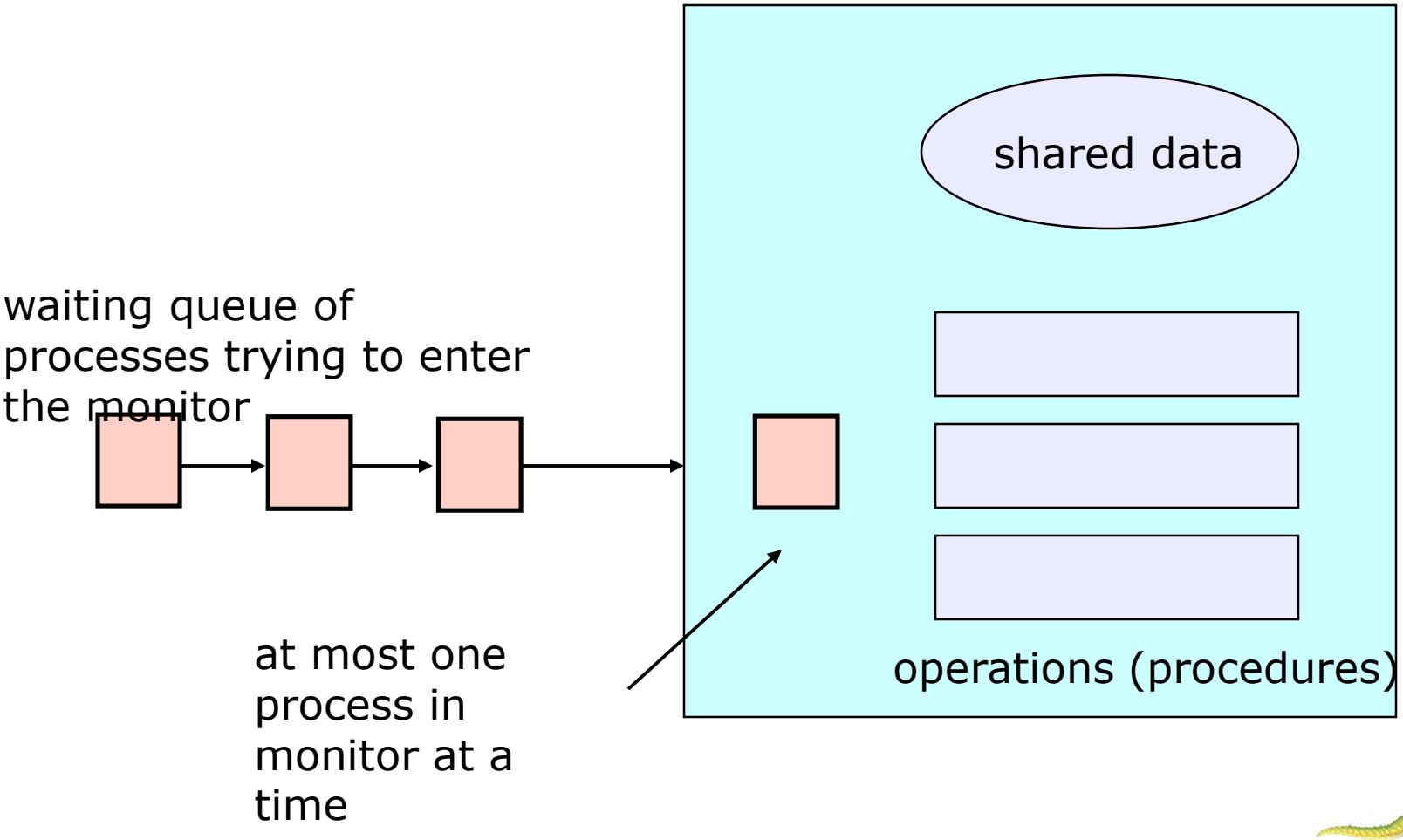
Problems with Semaphores

- They can be used to solve any of the traditional synchronization problems, but:
  - semaphores are essentially shared global variables
    - can be accessed from anywhere (bad software engineering)
  - there is no connection between the semaphore and the data being controlled by it
  - used for both critical sections (mutual exclusion) and for coordination (scheduling)
  - no control over their use, no guarantee of proper usage
- Thus, they are prone to bugs
  - another (better?) approach: use programming language support

Monitors

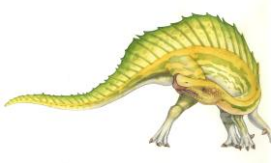
- A programming language construct that supports controlled access to shared data
  - synchronization code added by compiler, enforced at runtime
- Monitor is a software module that encapsulates:
  - shared data structures
  - procedures that operate on the shared data
  - synchronization between concurrent processes that invoke those procedures
- Monitor protects the data from unstructured access
  - guarantees only access data through procedures, hence in legitimate ways

A monitor


shared data
waiting queue of processes trying to enter the monitor
at most one process in monitor at a time
operations (procedures)

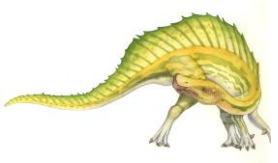
Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- It is the collection of condition variables and procedures combined together in a special kind of module or a package.
- The processes running outside the monitor can't access the internal variable of the monitor but can call procedures of the monitor.
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time

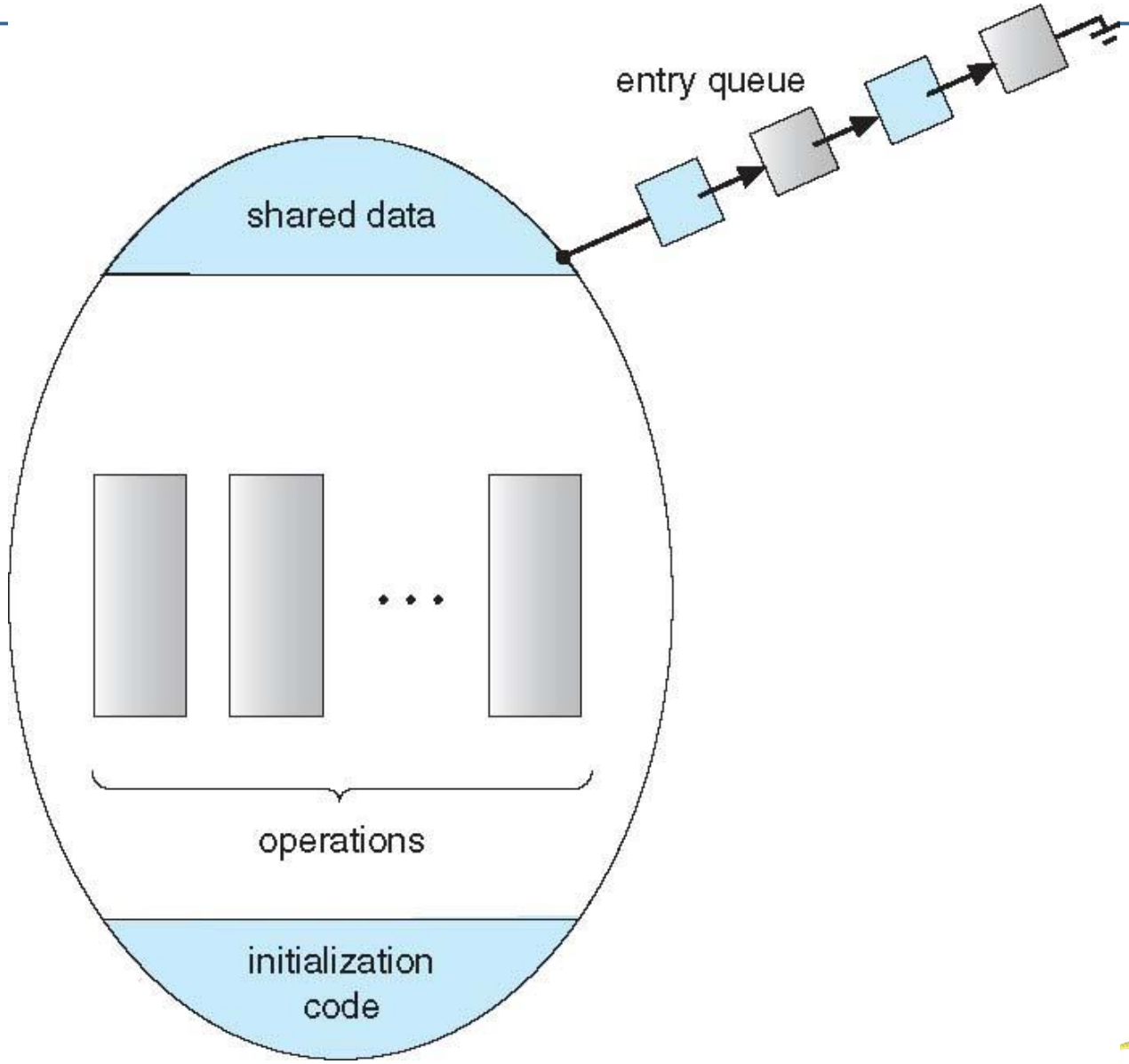


Monitors

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

 procedure Pn (…) {……}

    Initialization code (…) { … }
 }
```



Schematic view of a Monitor



Components of Monitor

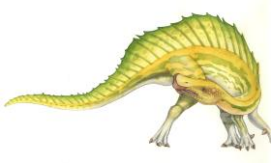
- There are four main components of the monitor:
- **Initialization: -** Initialization comprises the code, and when the monitors are created, we use this code exactly once.
- **Private Data:** - Private data is another component of the monitor. It comprises all the private data, and the private data contains private procedures that can only be used within the monitor. So, outside the monitor, private data is not visible.
- **Monitor Procedure:** - Monitors Procedures are those procedures that can be called from outside the monitor.
- **Monitor Entry Queue: -** Monitor entry queue is another essential component of the monitor that includes all the threads, which are called procedures.

Monitor facilities

- Mutual exclusion
  - only one process can be executing inside at any time
    - thus, synchronization implicitly associated with monitor
  - if a second process tries to enter a monitor procedure, it blocks until the first has left the monitor
    - more restrictive than semaphores!
    - but easier to use most of the time

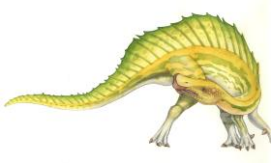
Monitor facilities

- Once inside, a process may discover it can't continue, and may wish to sleep
  - or, allow some other waiting process to continue
  - condition variables provided within monitor
    - processes can wait or signal others to continue
    - condition variable can only be accessed from inside monitor



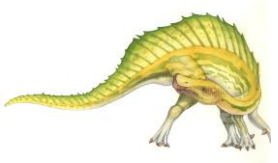
Condition Variables

- Three operations on condition variables
  - wait(c)
    - release monitor lock, so somebody else can get in
    - wait for somebody else to signal condition
    - thus, condition variables have wait queues
  - signal(c)
    - wake up at most one waiting process/thread
    - if no waiting processes, signal is lost
    - this is different than semaphores: no history!
  - broadcast(c)
    - wake up all waiting processes/threads



Condition Variables

- A place to wait; sometimes called a rendezvous point
- `condition x, y;`
  - **Wait operation:** x.wait() : Process performing wait operation on any condition variable are suspended.
  - The suspended processes are placed in block queue of that condition variable.
  - **Note:** Each condition variable has its unique block queue.
  - Wait(condition): release monitor lock, put process to sleep. When process wakes up again, re-acquire monitor lock immediately



Condition Variables

Signal operation

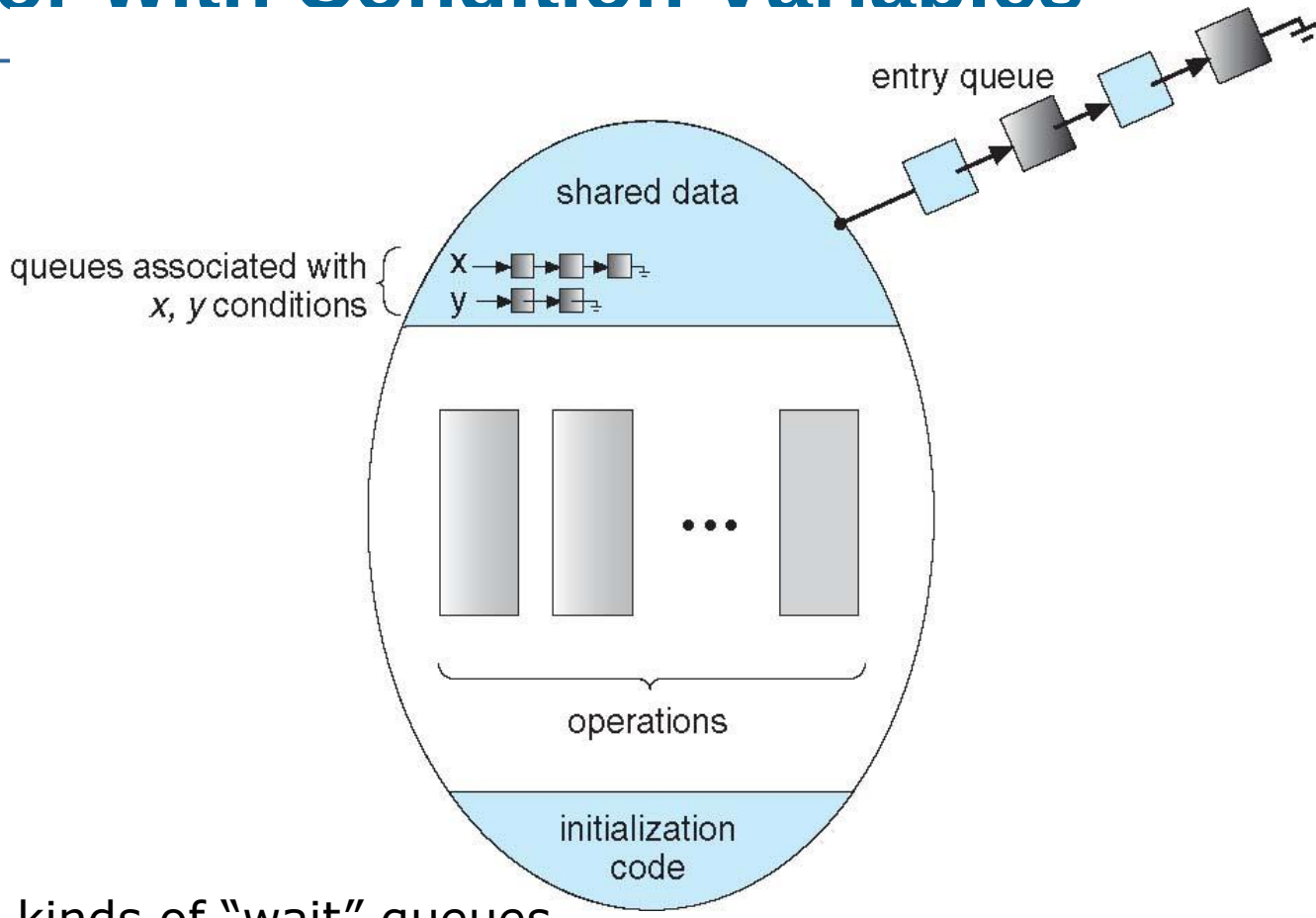
- x.signal(): When a process performs signal operation on condition variable, one of the blocked processes is given chance.

```
If (x block queue empty)
   // Ignore signal
else
   // Resume a process from block queue.
```

If no **`x.wait()`** on the variable, then it has no effect on the variable

Signal(condition): wake up one process waiting on the condition variable (FIFO). If nobody waiting, do nothing (no history).

Monitor with Condition Variables



Monitors have two kinds of "wait" queues

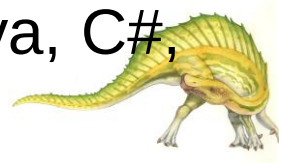
- Entry to the monitor: has a queue of threads waiting to obtain mutual exclusion so they can enter

- Condition variables: each condition variable has a queue of threads waiting on the associated condition

Monitor EventTracker

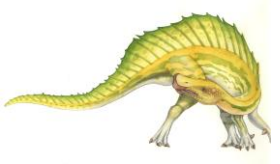
```
{    int numburgers = 0;
     condition hungrycustomer;

     void customerenter() {
          if (numburgers == 0)
               hungrycustomer.wait();
          numburgers -= 1 }

     void produceburger() {
          ++numburgers;
          hungrycustomer.signal(); }
}
```

- **Advantages of Monitor:**
  - Monitors have the advantage of making parallel programming easier and less error prone than using techniques such as semaphore.
- **Disadvantages of Monitor:**
  - Monitors have to be implemented as part of the programming language .
  - The compiler must generate code for them. This gives the compiler the additional burden of having to know what operating system facilities are available to control access to critical sections in concurrent processes.
- Some languages that do support monitors are Java, C#, Visual Basic, Ada and concurrent Euclid.



Bounded Buffer using Monitors

```
Monitor bounded_buffer {
  buffer resources[N];
  condition not_full, not_empty;

  procedure add_entry(resource x) {
    while(array "resources" is full)
      wait(not_full);
    add "x" to array "resources"
    signal(not_empty);
  }
  procedure get_entry(resource *x) {
    while (array "resources" is empty)
      wait(not_empty);
    *x = get resource from array "resources"
    signal(not_full);
  }
```



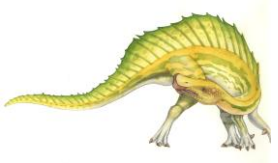
- There are several different variations on the wait/signal mechanism.
- They vary in terms of who gets the monitor lock after a signal.
- If process P invokes **`x.signal()`**, and process Q is suspended in **`x.wait()`**, what should happen next?
  - Both Q and P cannot execute in parallel.
  - If Q is resumed, then P must wait

Two Kinds of Monitors

- Mesa monitors: signal(c) means
  - waiter is made ready, but the signaller continues
    - waiter runs when signaller leaves monitor (or waits)
    - condition is not necessarily true when waiter runs again
  - signaller need not restore invariant until it leaves the monitor
  - being woken up is only a hint that something has changed
    - must recheck conditional case

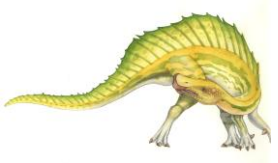
“Mesa semantics” (signal-and-continue):

- On signal, signaller keeps monitor lock.
- Awakened process waits for monitor lock with no special priority (a new process could get in before it).
- This means that the thing you were waiting for could have come and gone: must check again and be prepared to sleep again if someone else took it.
- Signal and continue – Q waits until P either leaves the monitor or it waits for another condition



Two Kinds of Monitors

- Hoare monitors: signal(c) means
  - run waiter immediately
  - signaller blocks immediately
    - condition guaranteed to hold when waiter runs
    - but, signaller must restore monitor invariants before signalling!



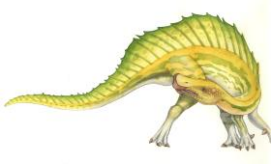
Condition Variables Choices

“Hoare semantics” (signal-and-wait).

- P waits until Q 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

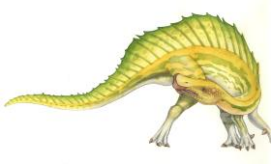
Monitor Solution to Dining Philosophers

```
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; }
}
```

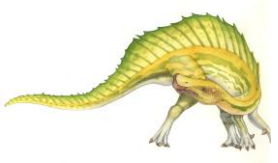


Solution to Dining Philosophers (Cont.)

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

```
DiningPhilosophers.pickup(i);

        EAT

DiningPhilosophers.putdown(i);
```

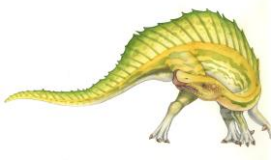
- No deadlock, but starvation is possible



Monitors	Semaphore
We can use condition variables only in the monitors.	In semaphore, we can use condition variables anywhere in the program, but we cannot use conditions variables in a semaphore.
In monitors, wait always block the caller.	In semaphore, wait does not always block the caller.
The monitors are comprised of the shared variables and the procedures which operate the shared variable.	The semaphore S value means the number of shared resources that are present in the system.
Condition variables are present in the monitor.	Condition variables are not present in the semaphore.

Condition Variables & Semaphores

- Condition Variables != semaphores
- Access to monitor is controlled by a lock
- Wait: blocks thread and gives up the monitor lock
  - To call wait, thread has to be in monitor, hence the lock
  - Semaphore P() blocks thread only if value less than 0
- Signal: causes waiting thread to wake up
  - If there is no waiting thread, the signal is lost
  - V() increments value, so future threads need not wait on P()
  - Condition variables have no history!



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

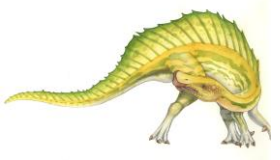
Monitor Implementation – Condition Variables

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

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

- The operation x.wait can be implemented as:

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



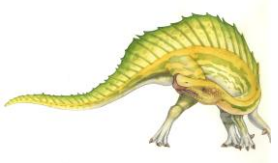
Monitor Implementation (Cont.)

- 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.signal() executed, which should be resumed?
- FCFS frequently not adequate
- **conditional-wait** construct of the form x.wait(c)
  - Where c is **priority number**
  - Process with lowest number (highest priority) is scheduled next

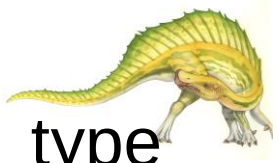


Single Resource allocation

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

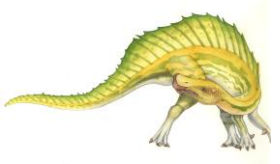
```
R.acquire(t);
    ...
    access the resource;
    ...

R.release;
```

- Where R is an instance of type ResourceAllocator



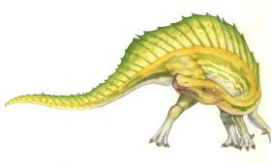
A Monitor to Allocate Single Resource

```
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;  }
 }
```

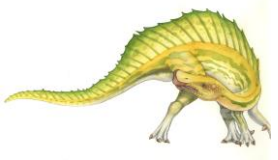


The Sleeping Barber Problem

- Another classical IPC problem takes place in a barber shop.
- A barbershop consists of a waiting room with n chairs, and the barber room containing the barber chair.
- If there are no customers to be served, the barber goes to sleep.
- If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop.
- If the barber is busy, but chairs are available, then the customer sits in one of the free chairs.
- If the barber is asleep, the customer wakes up the barber..



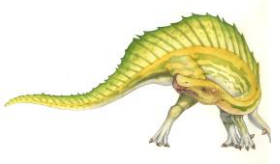
- Uses three semaphores, customers (counts waiting customers), barbers (the number of barbers who are idle), and mutex (mutual exclusion).
- Also need a variable, waiting, which also counts the waiting customers.
- The reason for having waiting is that there is no way to read the current value of a semaphore.
- In this solution, a customer entering the shop has to count the number of waiting customers.
- If it is less than the number of chairs, he stays; otherwise, he leaves.



```
#define CHAIRS 5 /* number of chairs for waiting customers */

typedef int semaphore;


semaphore customers = 0; /* number of waiting customers */


semaphore barbers = 0; /* number of barbers waiting for
customers */

semaphore mutex = 1; /* for mutual exclusion */


int waiting = 0; /* customers are waiting not being haircut */
```

```
void Barber(void)
{
 while (TRUE)
 {
 down(customers); /* go to sleep if number of customers is 0 */
 down(mutex); /* acquire access to ‘waiting’ */
 waiting = waiting – 1; /* decrement count of waiting customers */
 up(barbers); /* one barber is now ready to cut hair */
 up(mutex); /* release ‘waiting’ */
 cut_hair(); /* cut hair, non-CS */
 }
}
```

```
void customer(void) {
 down(mutex); /* enter CS */
 if (waiting < CHAIRS)
 {
 waiting = waiting + 1; /* increment count of waiting customers */
 up(customers); /* wake up barber if necessary */
 up(mutex); /* release access to 'waiting' */
 down(barbers); /* wait if no free barbers */
 get_haircut(); /* non-CS */
 }
 else
 {   up(mutex); /* shop is full, do not wait */  }
}
```



We also need a semaphore cutting which ensures that the barber won't cut another customer's hair before the previous customer leaves.

```
semaphore cutting = 0;
```

```
cut_hair()
{
  waiting(cutting);
 }
```

```
get_haircut()
{
  get hair cut for some time;
  signal(cutting);
}
```

- When the barber shows up for work in the morning, he executes the procedure barber, causing him to block on the semaphore customers because it is initially 0.
- The barber then goes to sleep.
- He stays asleep until the first customer shows up.
- When a customer arrives, he executes customer, starting by acquiring mutex to enter a critical region.
- If another customer enters shortly thereafter, the second one will no be able to do anything until the first one has released mutex.

- The customer then checks to see if the number of waiting customers is less than the number of chairs.
- If not, he releases *mutex* and leaves without a haircut.
- If there is an available chair, the customer increments the integer variable, *waiting*.
- Then he does an *up* on the semaphore customers, thus waking up the barber.
- At this point, the customer and the barber are both awake.
- When the customer releases *mutex*, the barber grabs it, does some housekeeping, and begins the haircut.