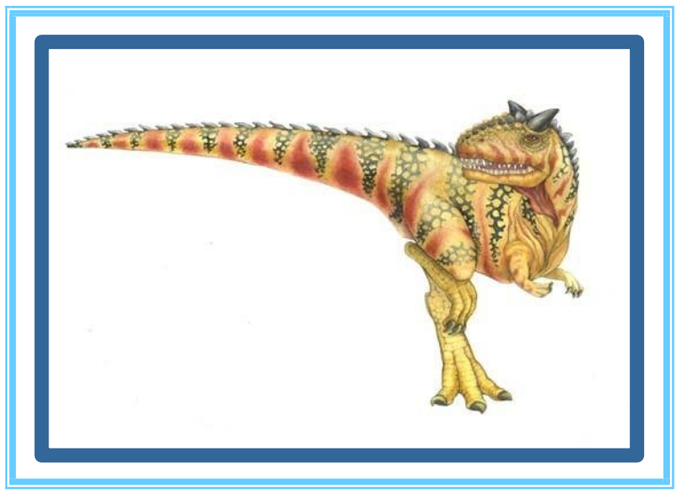
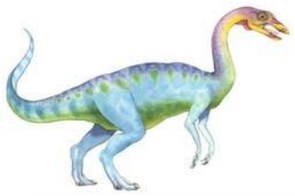
CSE-309
Operating Systems

Mohammad Shariful Islam
Lecturer, Department of CSE
Mobile: 01747612143
Email: sharifulruhan@gmail.com

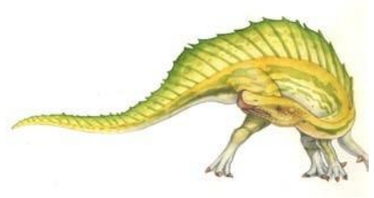
Chapter 6:
Process Synchronization





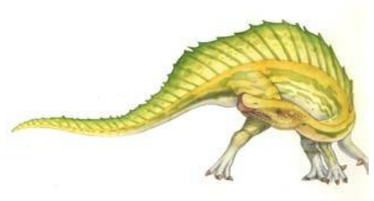
Module 6: Process Synchronization

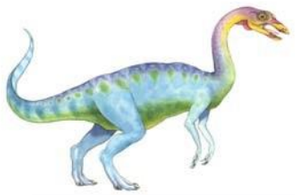
- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Atomic Transactions



Objectives

- 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 introduce the concept of an atomic transaction and describe mechanisms to ensure atomicity



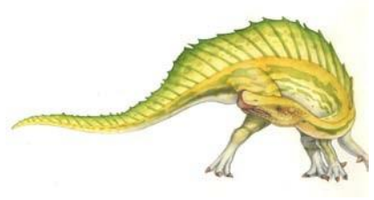


Background

- Concurrent access to shared data may result in data inconsistency

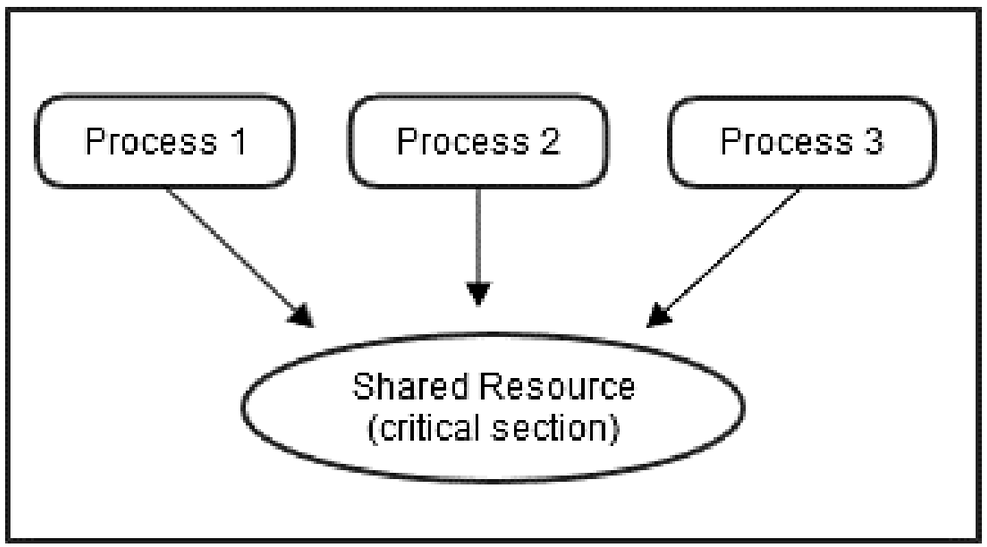
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

- 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 **count** that keeps track of the number of full buffers. Initially, count 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.



Process Synchronization

- When two or more process cooperates with each other, their order of execution must be preserved otherwise there can be conflicts in their execution and inappropriate outputs can be produced.
- A cooperative process is the one which can affect the execution of other process or can be affected by the execution of other process. Such processes need to be synchronized so that their order of execution can be guaranteed.
- The procedure involved in preserving the appropriate order of execution of cooperative processes is known as Process Synchronization. There are various synchronization mechanisms that are used to synchronize the processes.



Critical Section Problem

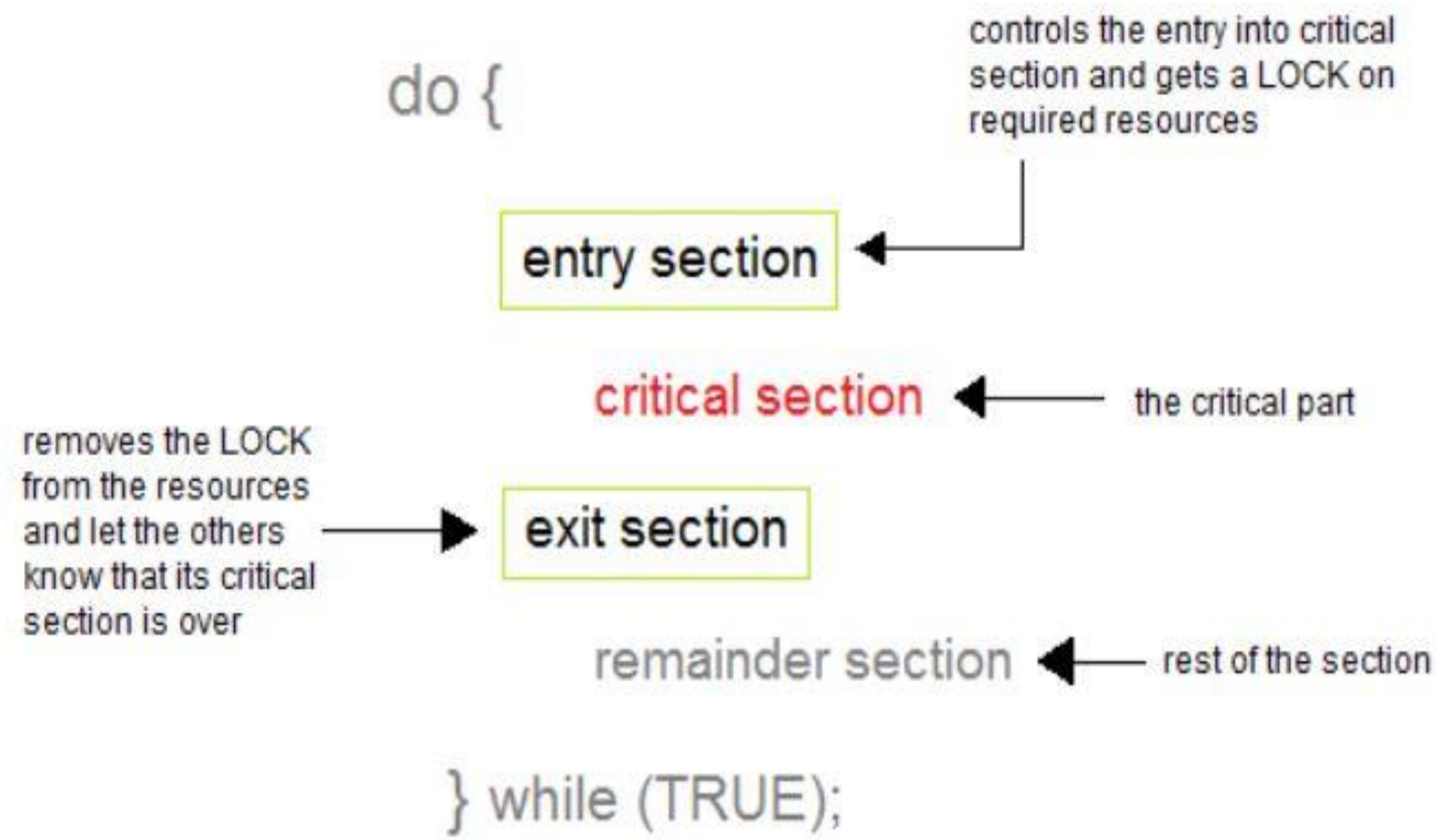
- Informally, a critical section is a code segment that accesses shared variables and has to be executed as an atomic action.
- The critical section problem refers to the problem of how to ensure that at most one process is executing its critical section at a given time.
- Important: Critical sections in different threads are not necessarily the same code segment!

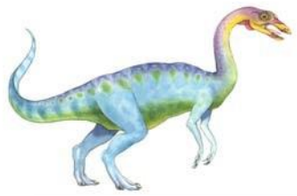
Critical Section Problem

- 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**
- Especially challenging with preemptive kernels.

Critical Section

- General structure of process p_i is

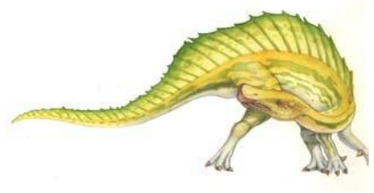


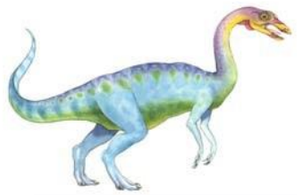


Solution to Critical-Section Problem

Solution to the Critical Section Problem must meet 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 exists some processes that wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision of which will enter its critical section next,
   - if no process is in critical section, can decide quickly who enters
   - only one process can enter the critical section so in practice, others are put on the queue.
3. **Bounded Waiting-** there must exist a bound 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
   - The wait is the time from when a process makes a request to enter its critical section until that request is granted.
   - in practice, once a process enters its critical section, it does not get another turn until a waiting process gets a turn (managed as a queue).

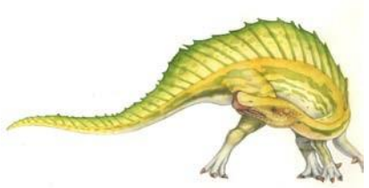


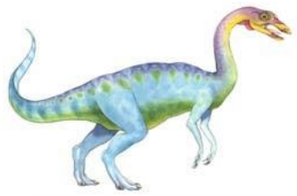


Peterson's Solution

- Peterson's Solution is a classical software based solution to the critical section problem.
- In Peterson's solution, we have two shared variables:
  - boolean flag[i] : Initialized to FALSE, initially no one is interested in entering the critical section
  - int turn : The process whose turn is to enter the critical section.

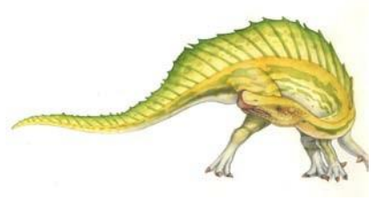
```
do {
        flag[i] = TRUE;
        turn = j;
        while (flag[j] && turn == j);
                critical section
        flag[i] = FALSE;
                remainder section
} while (TRUE);
```

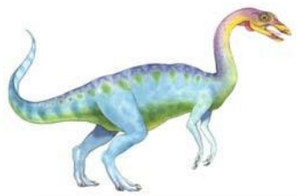




Peterson's Solution

- **Peterson's Solution preserves all three conditions :**
  - Mutual Exclusion is assured as only one process can access the critical section at any time.
  - Progress is also assured, as a process outside the critical section does not blocks other processes from entering the critical section.
  - Bounded Waiting is preserved as every process gets a fair chance.
- **Disadvantages of Peterson's Solution**
  - It involves Busy waiting
  - It is limited to 2 processes.





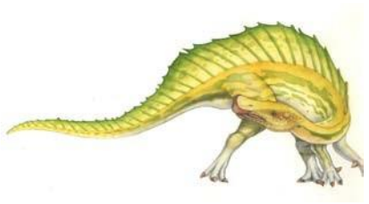
Peterson's Solution

- **Peterson's Solution preserves for two threads**

Flag[i] and intendToEnter[i] are same. Just only two different name but used to do the same function.

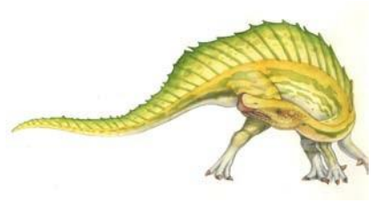
Peterson's algorithm

T0	T1	Comments
(1) context switch		*intendToEnter*[0] is set to true
	(1)	*intendToEnter*[1] is set to true
	(2)	*turn* is set to 0
context switch	(3)	T1 enters the *while* loop
(2)		*turn* is set to 1
(3) context switch		T0 enters the *while* loop
	(3)	T1 exits *while* loop
	(4)	T1 enters the critical section
	(5)	*intendToEnter*[1] is set to false
	(6)	T1 executes the non-critical section
	(1)	*intendToEnter*[1] is set to true
	(2)	*turn* is set to 0
context switch	(3)	T1 enters *while* loop
(3)		T0 exits *while* loop
(4)		T0 enters critical section
(5)		*intendToEnter*[0] is set to false
(6)		T0 executes the non-critical section
context switch		
	(3)	T1 exits the *while* loop
	(4)	T1 enters critical section



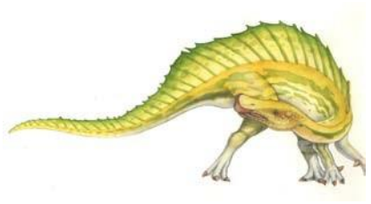
Solution to Critical-section Problem Using Locks

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



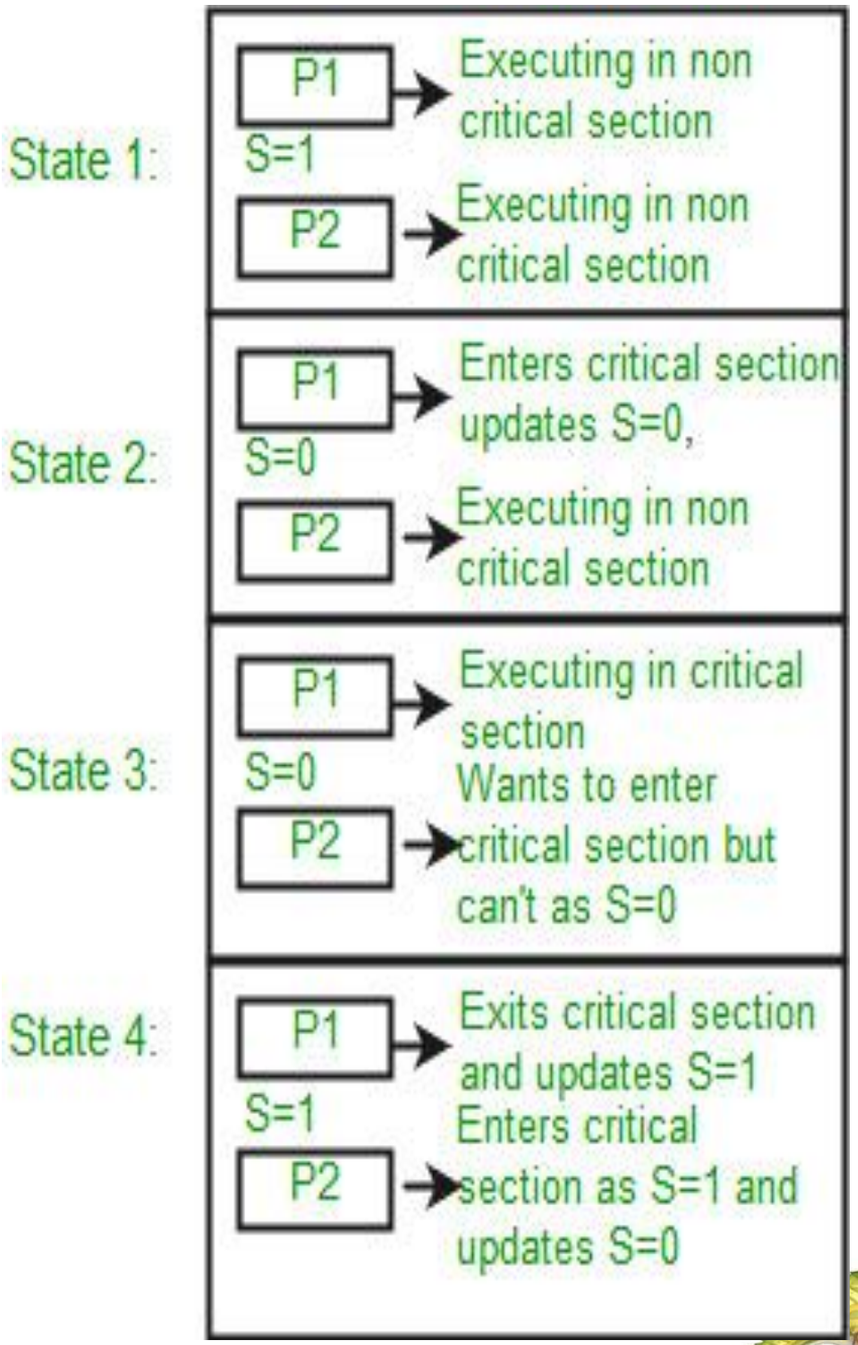
Semaphore

- Semaphore is a simply a variable. This variable is used to solve critical section problem and to achieve process synchronization in the multi processing environment.
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
- Can only be accessed via two indivisible (atomic) operations
  - wait (S) {
    while S <= 0
    ; // no-op
    S--;
    }
    signal (S) { S++;
    }



Semaphore

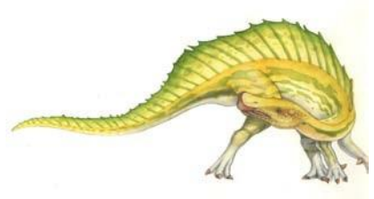
```
P(Semaphore s)
{
    s = s - 1;
    if (s < 0) {
            // add process to queue
            block();
    }
}
V(Semaphore s)
{
    s = s + 1;
    if (s >= 0) {
            // remove process p from queue
            wakeup(p);
    }
}
```



Semaphore as General Synchronization Tool

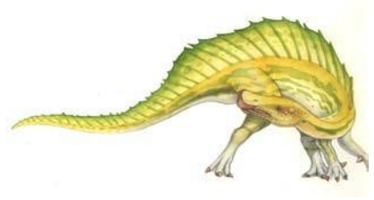
- **Counting** semaphore – integer value can range over an unrestricted domain
- **Binary** semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as **mutex locks**
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

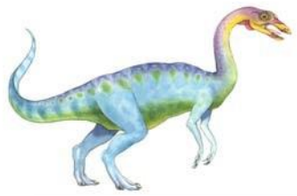
```
// initialized to 1
Semaphore mutex;
do {
    wait (mutex);
    // Critical Section
 signal (mutex);
    // remainder section
} while (TRUE);
```



Classical Problems of Synchronization

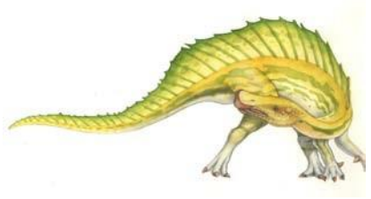
- Classical problems used to test newly-proposed synchronization schemes:
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem
  - Cigarette Smokers problem
  - Sleeping barber problem

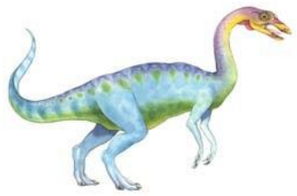




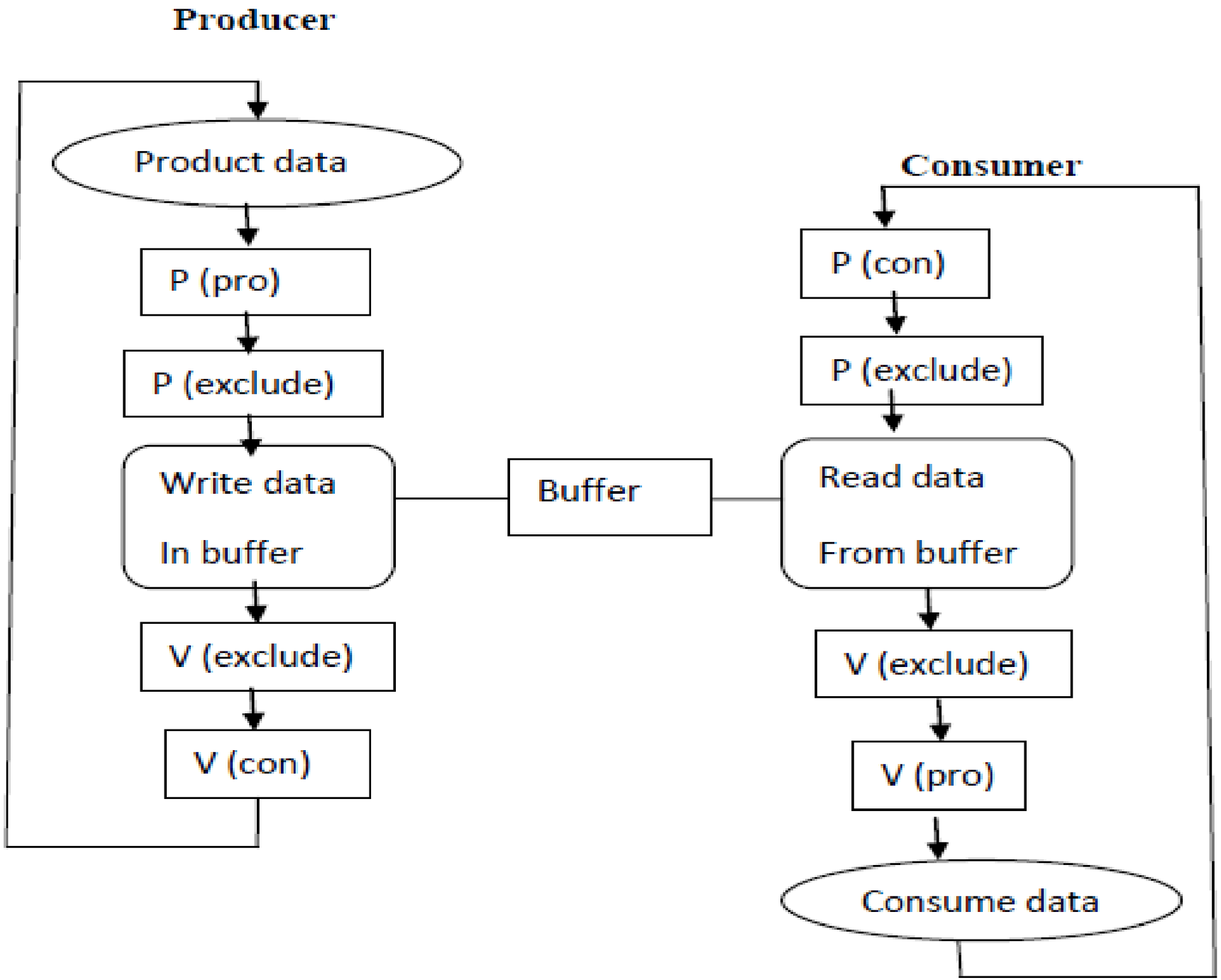
Bounded-Buffer Problem

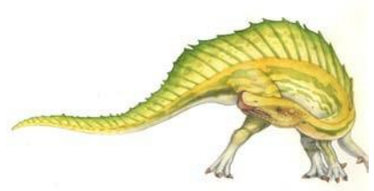
- Also known as producer-consumer problem
- The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer) one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

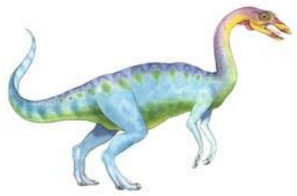




Bounded-Buffer Problem



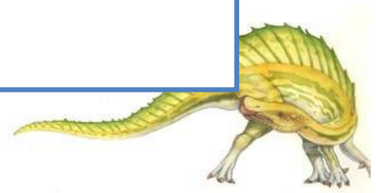


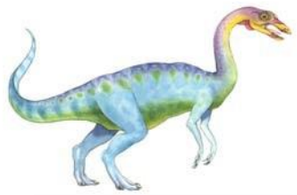


Assignment 2: Bounded-Buffer Problem

- **Implement Producer Consumer Problem in C programming Language.**

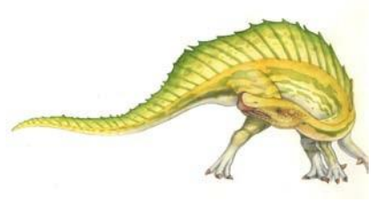
Output: *1.Producer* *2.Consumer* *3.Exit* *Enter your choice:1* *Producer produces the item 1* *Enter your choice:2* *Consumer consumes item 1* *Enter your choice:2* *Buffer is empty!!*	*Enter your choice:1* *Producer produces the item 1* *Enter your choice:1* *Producer produces the item 2* *Enter your choice:1* *Producer produces the item 3* *Enter your choice:1* *Buffer is full!!* *Enter your choice:3*

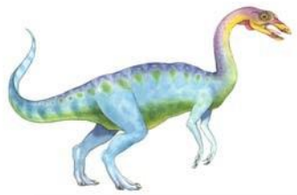




Readers-Writers Problem

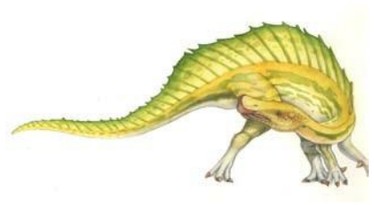
- Consider a situation where we have a file shared between many people.
  - If one of the people tries editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible to him/her.
  - However if some person is reading the file, then others may read it at the same time.
- Precisely in OS we call this situation as the **readers-writers problem**

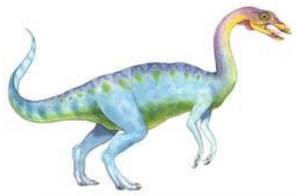




Readers-Writers Problem

- Problem parameters:
  - One set of data is shared among a number of processes
  - Once a writer is ready, it performs its write. Only one writer may write at a time
  - If a process is writing, no other process can read it
  - If at least one reader is reading, no other process can write
  - Readers may not write and only read

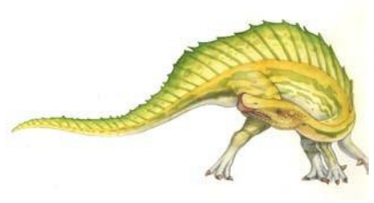


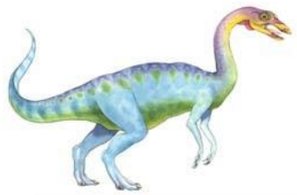


Readers-Writers Problem (Cont.)

- The structure of a writer process

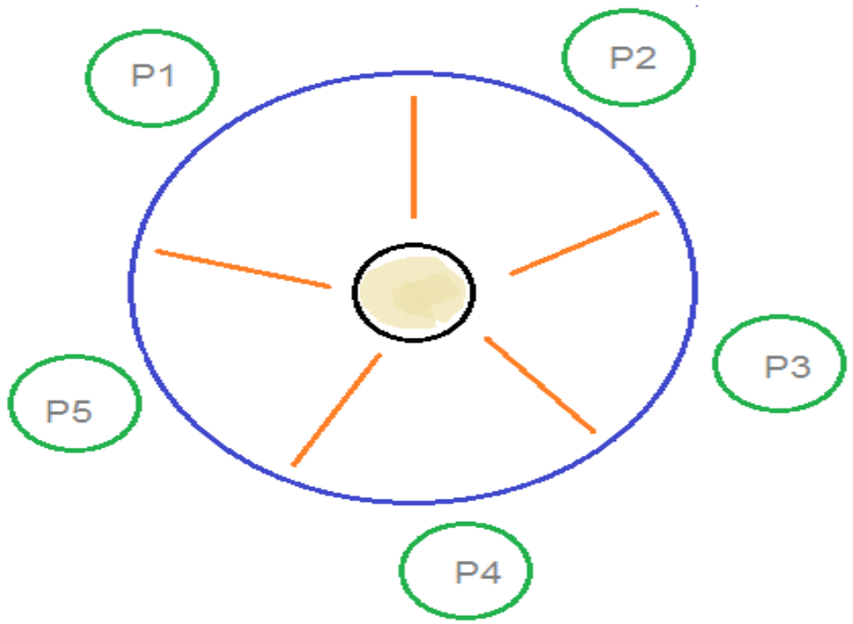
```
do {
        wait (wrt) ;

            //    writing is performed

        signal (wrt) ;
} while (TRUE);
```



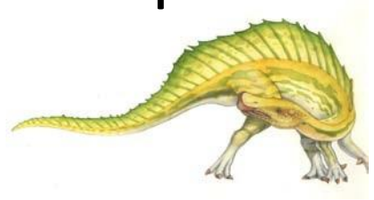


Dining-Philosophers Problem

- Consider there are five philosophers sitting around a circular dining table. The dining table has five chopsticks and a bowl of rice in the middle as shown in the below figure.

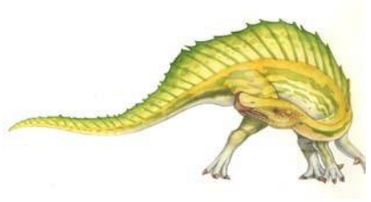


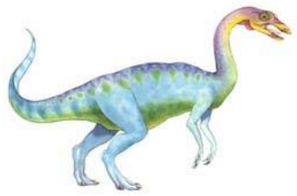
- At any instant, a philosopher is either eating or thinking. When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.



Dining-Philosophers Problem

- From the problem statement, it is clear that a philosopher can think for an indefinite amount of time.
- But when a philosopher starts eating, he has to stop at some point of time.
- The philosopher is in an endless cycle of thinking and eating.
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1





Dining-Philosophers Problem Algorithm

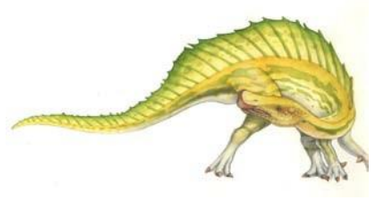
- The structure of Philosopher *i*:

```
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?
- When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick. Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down.
- But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever.

Dining-Philosophers Problem

- **The possible solutions for this are:**
  - A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.
  - Allow only four philosophers to sit at the table. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided.



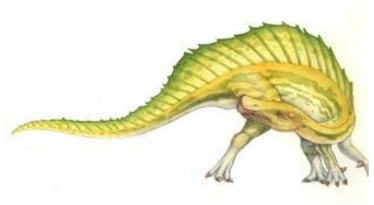
Dining-Philosophers Problem

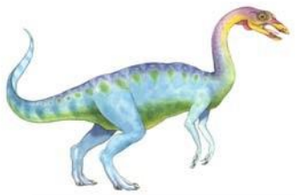
- **Implement Dinning- Philosophers Problem in C programming Language.**

Output

Fork 1 taken by Philosopher 1
Fork 2 taken by Philosopher 2
Fork 3 taken by Philosopher 3
Philosopher 4 is waiting for fork 3

Till now num of philosophers completed dinner are 0
Fork 4 taken by Philosopher 1
Philosopher 2 is waiting for Fork 1
Philosopher 3 is waiting for Fork 2
Philosopher 4 is waiting for fork 3
Till now num of philosophers completed dinner are 0

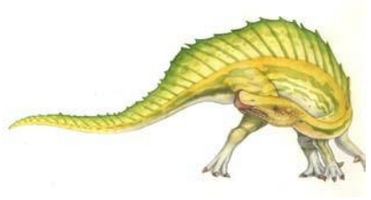


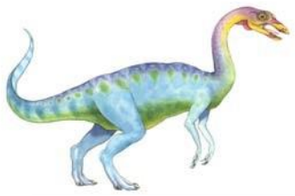


Dining-Philosophers Problem

Philosopher 1 completed his dinner
Philosopher 1 released fork 1 and fork 4
Fork 1 taken by Philosopher 2
Philosopher 3 is waiting for Fork 2
Philosopher 4 is waiting for fork 3
Till now num of philosophers completed dinner are 1

Philosopher 1 completed his dinner
Philosopher 2 completed his dinner
Philosopher 2 released fork 2 and fork 1
Fork 2 taken by Philosopher 3
Philosopher 4 is waiting for fork 3
Till now num of philosophers completed dinner are 2

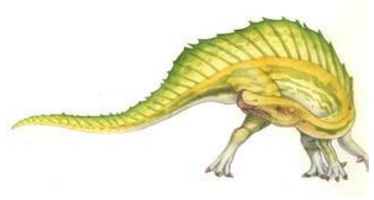


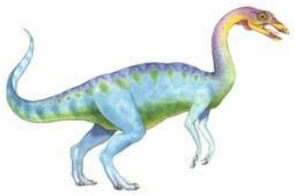


Dining-Philosophers Problem

Philosopher 1 completed his dinner
Philosopher 2 completed his dinner
Philosopher 3 completed his dinner
Philosopher 3 released fork 3 and fork 2
Fork 3 taken by philosopher 4
Till now num of philosophers completed dinner are 3

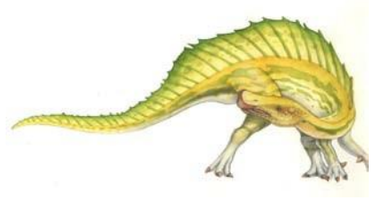
Philosopher 1 completed his dinner
Philosopher 2 completed his dinner
Philosopher 3 completed his dinner
Fork 4 taken by philosopher 4
Till now num of philosophers completed dinner are 3

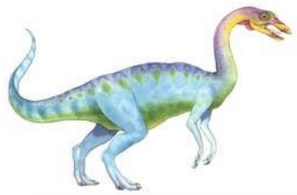




Dining-Philosophers Problem

Philosopher 1 completed his dinner
Philosopher 2 completed his dinner
Philosopher 3 completed his dinner
Philosopher 4 completed his dinner
Philosopher 4 released fork 4 and fork 3
Till now num of philosophers completed dinner are 4

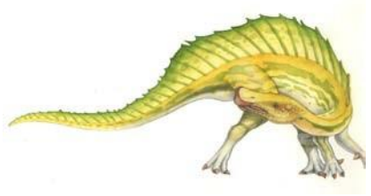


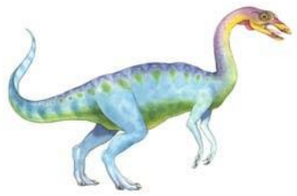


Monitors

- 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

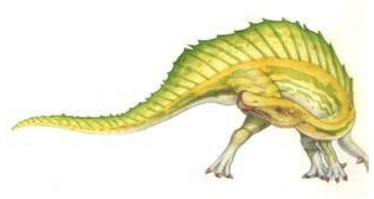
```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (…) { …. }

    procedure Pn (…) {……}

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

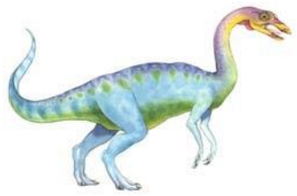




The Cigarette-Smokers Problem

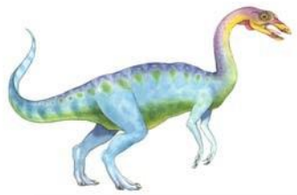
Consider a system with three smoker processes and one agent process. Each smoker continuously rolls a cigarette and then smokes it. But to roll and smoke a cigarette the smoker needs three ingredients: tobacco, paper and matches. One of the smoker processes has paper another has tobacco and the third has matches. The agent has an infinite supply of all three materials. The agent places two of the ingredients on the table. The smoker who has the remaining the ingredient then makes and smokes a cigarette, signaling the agent on completion. The agent then puts out another two of the three ingredients, and the cycle repeats. Write a program to synchronize the agent and the smokers using Java synchronization.





The Sleeping-Barber Problem

A barbershop consists of a waiting room with n chairs and a barber room with one 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. Write a program to coordinate the barber and the customers

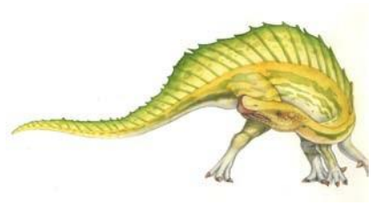


Windows XP Synchronization

- 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
  - spinlocks
  - reader-writer versions of both
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption



End of Chapter 6