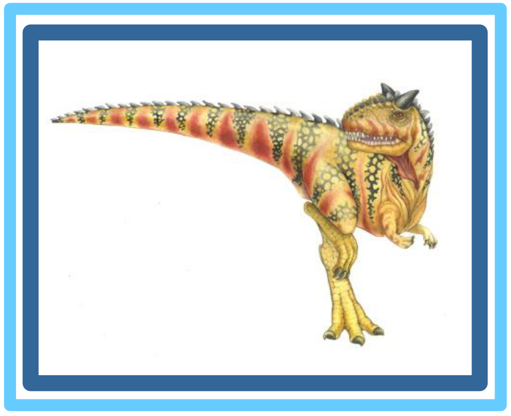
Chapter 5: CPU Scheduling



Multiple-Processor Scheduling

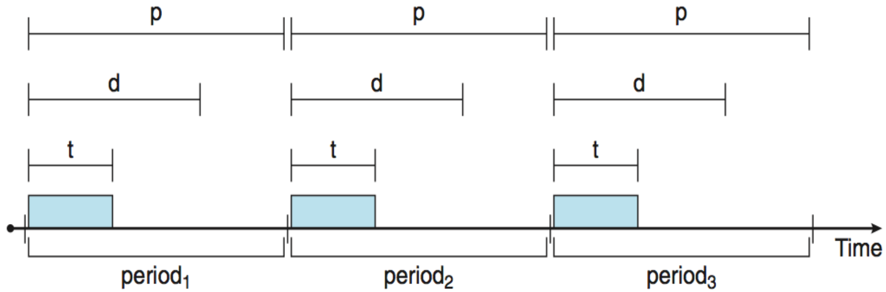
- Multiple processors?
- CPU scheduling more complex when multiple CPUs are available
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing** (**SMP**) – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
  - Currently, most common

Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
  - **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
  - **Pull migration** – idle processors pulls waiting task from busy processor

Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at constant intervals
  - Has processing time *t*, deadline *d*, period *p*
  - $0 \leq t \leq d \leq p$
  - **Rate** of periodic task is $1/p$



Rate Montonic Scheduling

- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority
- Example with three processors

Process	Capacity	Period
P1	3	20
P2	2	5
P3	2	10

Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:

  the earlier the deadline, the higher the priority;
  the later the deadline, the lower the priority

Process	Capacity	Period	deadline
P1	3	20	7
P2	2	5	4
P3	2	10	8

Activity

Apply Rate Monotonic Scheduling

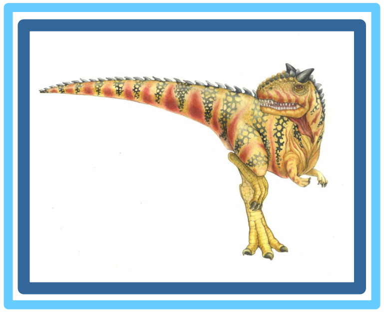
Process	Capacity	Period
P1	5	15
P2	4	5
P3	4	3

Process	Capacity	Period
P1	1	15
P2	3	5
P3	1	3

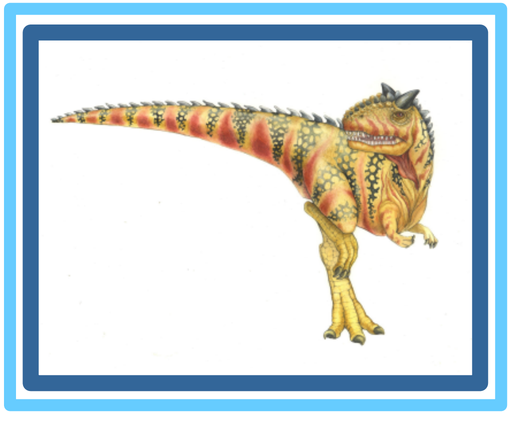
EDF:

Process	Capacity	Period	Deadline
P1	1	4	4
P2	2	6	6
P3	3	8	8

End of Chapter 5



Chapter 6: Synchronization Tools



Chapter 6: Synchronization Tools

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores

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

Cooperating process - Processes that can affect or get affected by other processes executing in the system.

Cooperating processes can be:

- Directly sharing a logical address space(both code and data)
- Allowed to share data only through files or messages

Problem: Concurrent access to shared data may result in data inconsistency

Solution: Process synchronization

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Eg: Producer consumer problem
- Solution : with the use of shared memory
- Buffer: the region in the shared memory that both producer and consumer use
- The producer produce one item while the consumer is consuming another item
- The producer and consumer should be synchronized

Background

Buffer: the region in the shared memory that both producer and consumer use

Two types of buffer:

- Unbounded — No limit on the size of the buffer. The producer can always produce items, consumer may need to wait for new items
- Bounded — Fixed buffer size. Consumer must wait when buffer is empty. Producer must wait when the buffer is full

Illustration of the problem:
Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers. We can do so by having an integer **`counter`** that keeps track of the number of full buffers. Initially, **`counter`** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {
        /* produce an item in next produced */

        while (counter == BUFFER_SIZE) ;
                /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Consumer

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

Race Condition

`counter++` could be implemented as

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

`counter--` could be implemented as

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

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

S0: producer execute `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}

When several processes access and manipulate the data concurrently and the outcome of the execution depends on the particular order in which the access takes place — Race Condition