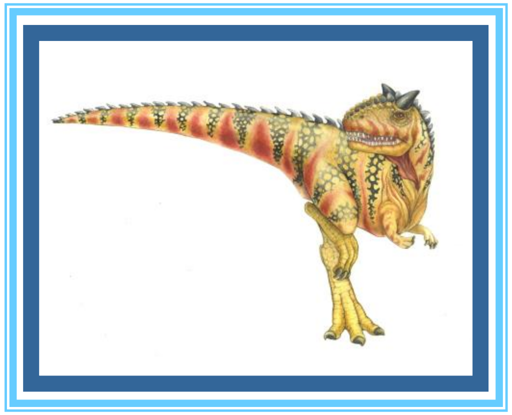
Chapter 5: Process Synchronization



Cooperating Process

- A **cooperating process is one that can affect or be affected by other processes** executing in the system.
- Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages.
- CPU scheduler switches rapidly between processes to provide concurrent execution. This means that one process may only partially complete execution before another process is scheduled

Bounded Buffer (Producer-Consumer Problem)

- In computing, the **producer–consumer problem** (also known as the **bounded-buffer problem**) is a classic example of a multi-process synchronization 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 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.

- The solution for the producer is to either go to sleep or discard data if the buffer is full.
- The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again.
- In the same way, the consumer can go to sleep if it finds the buffer empty.
- The next time the producer puts data into the buffer, it wakes up the sleeping consumer.
- The solution can be reached by means of inter-process communication, typically using semaphores. An inadequate solution could result in a deadlock where both processes are waiting to be awakened.

Inadequate Implementation

- To solve the problem, some programmer might come up with a solution shown below.
- In the solution two library routines are used, sleep and wakeup.
- When sleep is called, the caller is blocked until another process wakes it up by using the wakeup routine.
- The global variable itemCount holds the number of items in the buffer.

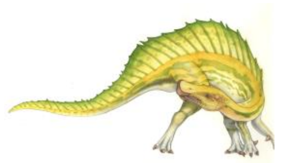
```
int itemCount = 0;

procedure producer()
{
    while (true)
    {
        item = produceItem();

        if (itemCount == BUFFER_SIZE)
        {
            sleep();
        }

        putItemIntoBuffer(item);
        itemCount = itemCount + 1;

        if (itemCount == 1)
        {
            wakeup(consumer);
        }
    }
}
```

```
procedure consumer()
{
    while (true)
    {

        if (itemCount == 0)
        {
            sleep();
        }

        item = removeItemFromBuffer();
        itemCount = itemCount - 1;

        if (itemCount == BUFFER_SIZE - 1)
        {
            wakeup(producer);
        }

        consumeItem(item);
    }
}
```

- The problem with this solution is that it contains a race condition that can lead to a deadlock. Consider the following scenario:
- The consumer has just read the variable itemCount, noticed it's zero and is just about to move inside the if block.
- Just before calling sleep, the consumer is interrupted and the producer is resumed.
- The producer creates an item, puts it into the buffer, and increases itemCount.
- Because the buffer was empty prior to the last addition, the producer tries to wake up the consumer.
- Unfortunately the consumer wasn't yet sleeping, and the wakeup call is lost. When the consumer resumes, it goes to sleep and will never be awakened again. This is because the consumer is only awakened by the producer when itemCount is equal to 1.
- The producer will loop until the buffer is full, after which it will also go to sleep.
- Since both processes will sleep forever, we have run into a deadlock.

Critical Section Problem

- Consider system of ***n*** processes {p_0, p_1, … 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**



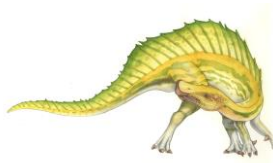
Critical Section

- General structure of process ***P_i***

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

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted



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 $\mathbf{P_i}$ is ready!

Example Algorithm (two processes P1 and P2)

- Process P1 wants to enter into its critical section:

```
flag[1]=true
 turn= 2
if (turn==2 && flag[2]==true)
        {
        P2 enters its critical section
        flag[2]=false
        }
P1 waits
```

Example Algorithm (two processes P1 and P2)

- Process P2 wants to enter into its critical section:

```
flag[2]=true
 turn= 1
if (turn==1 && flag[1]==true)
          {
          P1 enters its critical section
          flag[1]=false
          }
P2 waits
```