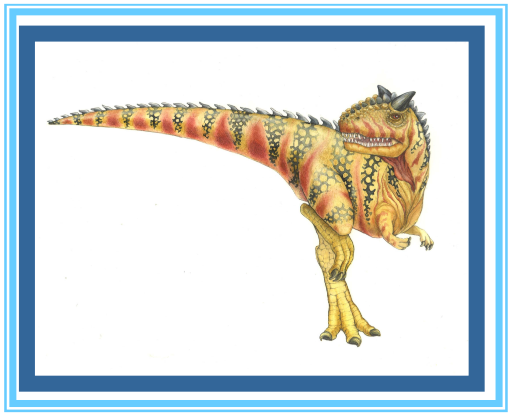
Chapter 6: Process Synchronization



Module 6: Process Synchronization

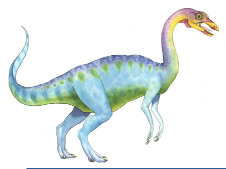
- Background
- The Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Semaphores

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

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes



The Critical-Section Problem

- ***n*** processes
  each has a segment of code called **critical section** in which it may change common var's, updating tables or lists, writing a file, ……..
- The goal **:** if one process is executing in its critical section, no other process to be allowed to enter its critical section
- *Each process must request permission to enter* ***C.S.***
  this is implemented in entry section

The Critical-Section Problem

```
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

- Two process solution
- 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!
- It can not be guaranteed to work properly (machine architecture)

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 Satisfies Requirements?

1. Mutual Exclusion – Only process P_i is executing in its critical section
   P_j can not enter its critical section

2. Progress - If no process is executing in its critical section and there exist a process that wishes to enter its critical section (flag), and turn can have only one value (I or j) then that process will enter the critical section next and cannot be postponed indefinitely

3. Bounded Waiting - Once a process(1) is allowed to enter its **C.S.** it will reset its (flag==FALSE) in exit allowing the process(2) (if ready) to enter its **C.S.** and p(1) can’t enter its **C.S.** until p(2) enters its **C.S.** [because p(1) changed (turn==2)]

Synchronization Hardware

- Any solution to **critical section problem** requires a simple tool or a **lock**
- Modern computers provide special hardware instructions that allow us to test and modify the content of a word atomically i.e. as a one uninterrupted unit
- TestAndSet lock and Semaphores are two examples of H/W tools

TestAndSet Instruction

- Definition:

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

Solution using TestAndSet

- Shared boolean variable lock., initialized to false.
- Solution:

```
do {
        while ( TestAndSet (&lock ))
                  ;   // do nothing

                //    critical section

        lock = FALSE;

                //      remainder section

} while (TRUE);
```

Semaphores

- Previous solution may be complicated to implement
- Semaphore *S* – integer variable accessed only through two standard atomic operations : wait() and signal()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

  -
    ```
    wait (S) {
            while S <= 0
                  ; // no-op
               S--;
     }
    ```

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

Semaphores

- Counting semaphore – integer value can range over an unrestricted domain
  counting semaphore can be used to control access
  to shared resources, initialized to # of resources

- Binary semaphore – integer value can range only between 0 and 1,
  can be simpler to implement
  - Also known as mutex locks

- Provides mutual exclusion

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

Semaphores to Sync processes

Concurrently running processes P_1, P_2

executing statement S_1 first then S_2

int (synch) initialized to 0

$$\left.\begin{array}{l} S_1; \\ \text{signal (synch);} \end{array}\right\} P_1$$

$$\left.\begin{array}{l} \text{wait (synch);} \\ S_2; \end{array}\right\} P_2$$