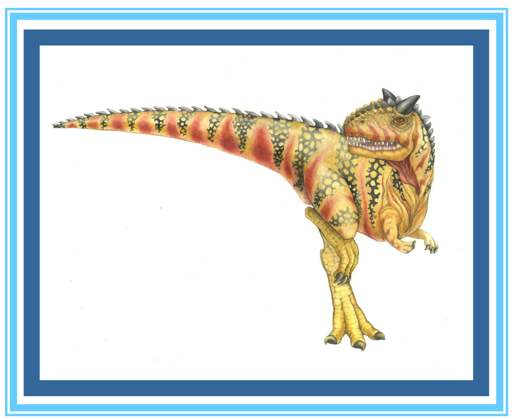
Chapter 5: Process Synchronization

Lecture # 13



Chapter 5: Process Synchronization

- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores

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 $P_{0\ and}$ P_1

P_0

```
while(1)
   {
    flag[0] = true;
    turn = 1;
        while (turn==1 &&
       flag[1]==T);
        critical section
    flag[0] = false;
        remainder section
    }
```

P_1

```
while(1)
   {
    flag[1] = true;
    turn = 0;
        while(turn==0 &&
      flag[0]==T);
        critical section
    flag[1] = false;
        remainder section
    }
```

Turn = 0 Or 1	
P_0	P_1
Flag= F	Flag= F

1: Mutual Exclusion: let,P0 in C.S and P1 try to enter C.S.
When P0 in C.S. flag[0] = T , flag[1] = F , turn = 1
the while condition of P0 code is false so P0 enters in C.S.
Meanwhile, if P1 try to enter it will do flag[1] =T , turn =0
so the while condition of P1 is true hence P1 is blocked.

Algorithm for Process P_0 and P_1

P_0

```
while(1)
   {
    flag[0] = true;
    turn = 1;
       while (turn==1 &&
       flag[1]==T);
       critical section
    flag[0] = false;
       remainder section
    }
```

P_1

```
while(1)
   {
    flag[1] = true;
    turn = 0;
       while(turn==0 &&
      flag[0]==T);
       critical section
    flag[1] = false;
       remainder section
    }
```

Turn = 0 Or 1

P_0	P_1
Flag= F	Flag= F

2 Progress: P0 and P1 both want or ready to enter C.S simultaneously then the last value in turn determines which process will enter C.S. let Last value in turn = 1 then P1 access the C.S but as it left C.S. it make flag[1] = F that allows P0 to enter C.S. Hence, no deadlock and stick alternations.

Algorithm for Process P_0 and P_1

P_0

```
while(1)
   {
    flag[0] = true;
    turn = 1;
       while (turn==1 &&
       flag[1]==T);
       critical section
    flag[0] = false;
       remainder section
    }
```

P_1

```
while(1)
   {
    flag[1] = true;
    turn = 0;
       while(turn==0 &&
      flag[0]==T);
       critical section
    flag[1] = false;
       remainder section
    }
```

Turn = 0 Or 1

P_0	P_1
Flag= F	Flag= F

3: Bounded wait: P0 enters C.S. and immediately it exist it again
Try to enter. But if P1 wants to enter mean flag[1] = T then P0 must wait
Until P1 once enters and exit C.S.

Peterson’s Solution (Cont.)

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved

     P_0 enters CS only if:

     either **`flag[1] = false`** or **`turn = 0`**
  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 below 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

Solution to Critical-section Problem Using Locks

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

Mutex Locks

- Hardware 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);
  ```

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

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

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

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

Semaphore

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

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

```
do {
Entry section
//critical section
Exit section
//remainder section
 } while (true);
```

S = 1