Chapter 6: Process Synchronization

Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE 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_i

```
do {
      flag[i] = TRUE;
      turn = j;
      while ( flag[j] && turn == j);

            CRITICAL SECTION

      flag[i] = FALSE;

             REMAINDER SECTION

   } while (TRUE);
```

Synchronization Hardware

- Many systems provide hardware support for critical section code
- 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
    - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

TestAndndSet 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);
```

Swap Instruction

- Definition:

```
void Swap (boolean *a, boolean *b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp:
}
```

Solution using Swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key.
- Solution:

```
do {
    key = TRUE;
    while ( key == TRUE)
        Swap (&lock, &key );

        //   critical section

    lock = FALSE;

        //   remainder section

} while ( TRUE);
```

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

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

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

```
– Semaphore S;   // initialized to 1
– wait (S);
      Critical Section
  signal (S);
```

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the crtical section.
  - Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - block – place the process invoking the operation on the appropriate waiting queue.
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue.

Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of wait:

```
wait (S){
      value--;
      if (value < 0) {
              add this process to waiting queue
              block();  }
 }
```

- Implementation of signal:

```
Signal (S){
        value++;
         if (value <= 0) {
                remove a process P from the waiting queue
                 wakeup(P);  }
}
```

Deadlock and Starvation

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.