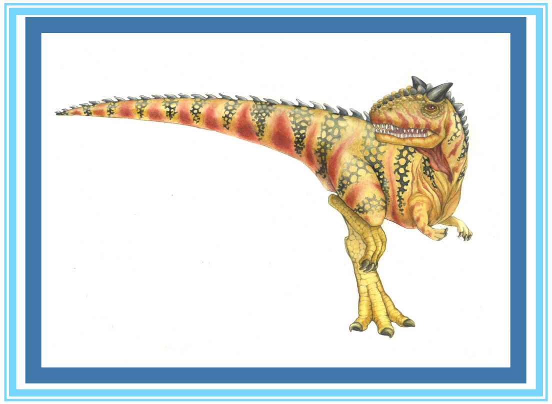
Chapter 6: Process Synchronization



Critical Section Problem

- General structure of process p_i is

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

Figure 6.1 General structure of a typical process P_i.

Requirements of Critical-Section Prob.

1. **Mutual Exclusion** - If process P_i is 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 some processes wish to enter their critical section, then the selection of the next process cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes enter critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the n processes

1st: Use lock

```
shared int locked = false;
do {
        while (locked == true);
        locked = true;
        critical section
        locked = false;
        remainder section
} while (true);
```

- Fails to meet
- Solution: Allow only one process to

2^{nd}: Take turns

```
shared int turn = 0;
do {
        while (turn != me);
        critical section
        turn = !me;
        remainder section
} while (true);
```

- Fails to meet
- Solution: Check if the other process

3rd : Check intention

```
shared int flag[2];
do {
        flag[me] = true;
        while (flag[ !me ] == true);
        critical section
        flag[me] = false;
        remainder section
} while (true);
```

- Fails to meet
- Solution: check both

Peterson's Solution

```
shared int turn, flag[2];
do {
        flag[me] = true;
        turn = ! me;
        while (flag[! me] && turn == ! me);
        critical section
        flag[me] = false;
        remainder section
} while (true);
```

- Provable that

1. Mutual exclusion:
2. Progress:
3. Bounded-waiting:

Peterson's Solution

Process 0:

```
shared int turn, flag[2];
do {
        flag[me] = true;
        turn = ! me;
        while (flag[! me] && turn == ! me);
        critical section
        flag[me] = false;
        remainder section
} while (true);
```

Process 1:

```
shared int turn, flag[2];
do {
        flag[me] = true;
        turn = ! me;
        while (flag[! me] && turn == ! me);
        critical section
        flag[me] = false;
        remainder section
} while (true);
```

Lessons

- Need a locking mechanism

    acquire lock

        critical section

    release lock

- Peterson’s algorithm still needs atomic access to shared variables
- Problem about shared variable comes from
    - the interruptible gap between get value & set value operations

        register ← <memory>

        register = <new value>

        <memory> ← register

    - Make these operations not *interruptible*, but HOW?

Disabling interrupts

- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable

Atomic instruction

```
shared int locked = false;
do {
    while (locked == true);
    locked = true;
    critical section
    locked = false;
    remainder section
} while (true);
```

Remove gap between TEST and SET!!

```
while( TestSet( &locked ) );
```

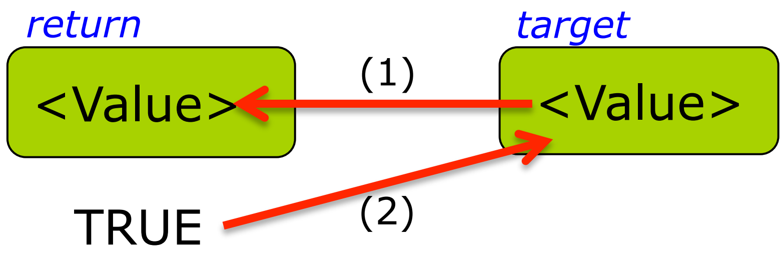
Returns the current value and set TRUE if FALSE

TestAndSet Instruction

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

TestAndSet Instruction-Better

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



Solution using TestAndSet

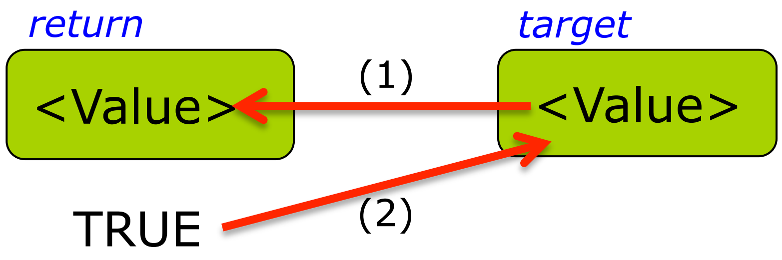
- Shared boolean variable lock, initialized to FALSE

```
do {
        while ( TestAndSet (&lock ));

                critical section

        lock = FALSE;

                remainder section

} while (TRUE);
```

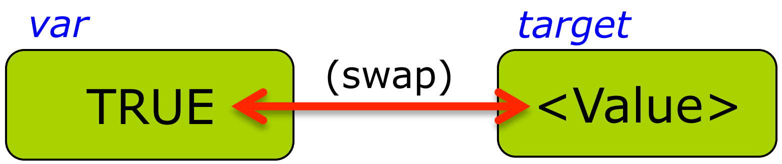
MX	
Prog.	
BW	

Another way of doing it

TestAndSet()



Swap()



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

```
do {
        key = TRUE;
        while ( key == TRUE)
                Swap (&lock, &key );
         //    critical section
         lock = FALSE;

         //      remainder section
 } while (TRUE);
```

Bounded-waiting Mutual Exclusion with TestandSet()

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
                key = TestAndSet(&lock);
        waiting[i] = FALSE;
                // critical section
        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
                j = (j + 1) % n;
        if (j == i)
                lock = FALSE;
        else
                waiting[j] = 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 mutex;    //  initialized to 1
do {
    wait (mutex);
        // Critical Section
    signal (mutex);
          // remainder section
} while (TRUE);
```

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(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

- Implementation of signal:

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                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
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**