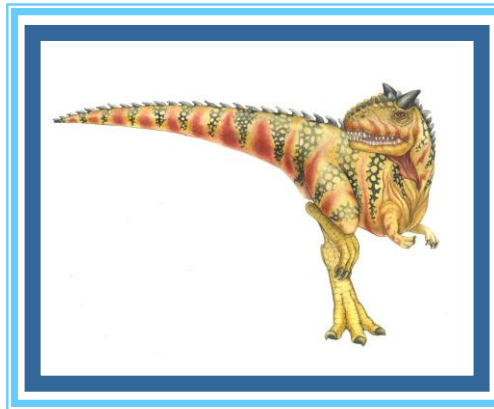


Chapter 6: Process Synchronization





Peterson's Solution

- Good algorithmic description of solving the problem
- **Restriction** - 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_i

```
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
        critical section  
    flag[i] = false;  
        remainder section  
} while (true);
```





Peterson's Solution (Cont.)

□ Provable that the three CS requirement are met:

1. Mutual exclusion is preserved

P_i enters CS only if:

either `flag[j] = false` or `turn = i`

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
 - ▶ **Atomic** = non-interruptible
 - ❑ Either test memory word and set value
 - ❑ Or swap contents of two memory words





Solution to Critical-section Problem Using Locks

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





test_and_set Instruction

Definition:

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

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.





Solution using test_and_set()

- Shared Boolean variable **lock**, initialized to FALSE
- Solution:

```
do {  
    while (test_and_set(&lock))  
        ; /* do nothing */  
        /* critical section */  
    lock = false;  
        /* remainder section */  
} while (true);
```





compare_and_swap Instruction

Definition:

```
int compare_and_swap(int *value, int expected,
int new_value) {
    int temp = *value;

    if (*value == expected)
        *value = new_value;
    return temp;
}
```

1. Executed atomically
2. Returns the original value of passed parameter “value”
3. Set the variable “value” the value of the passed parameter “new_value” but only if “value” == “expected”. That is, the swap takes place only under this condition.



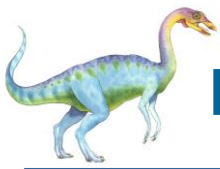


Solution using compare_and_swap

- Shared integer **lock** initialized to 0;
- Solution:

```
do {  
    while (compare_and_swap(&lock, 0,  
1) != 0)  
        ; /* do nothing */  
    /* critical section */  
    lock = 0;  
    /* remainder section */  
} while (true);
```





Bounded-waiting Mutual Exclusion with test_and_set

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&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);
```

Common Data structure

```
boolean waiting[n];
boolean lock;
```

Initialized to FALSE

