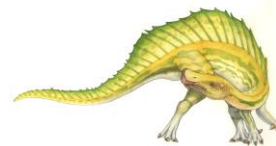




# Synchronization Hardware

---

- 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**
    1. Either test memory word and set value
    2. 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);
```





# TestAndndSet Instruction

---

- Definition:

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



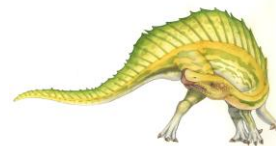


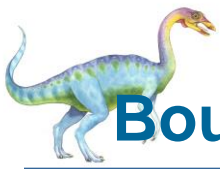
# Solution using TestAndSet

---

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

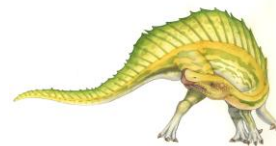
```
do {  
    while ( TestAndSet (&lock ))  
        ; // do nothing  
  
        // critical section  
  
    lock = FALSE;  
  
        // remainder section  
  
} while (TRUE);
```





# 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);
```





# Software Solutions

- Mutex Lock
  - short for mutual exclusion
  - software tool to solve critical section problem
  - acquire () acquires the lock
  - release () releases the lock

```
acquire ()  
{  
    while (!available); /* busy wait */  
    available = false;  
}
```

```
release () {available = true;}
```

```
do{ // solution to critical section  
    acquire ();  
    enter critical section  
    release ();  
    remainder section  
} while (true);
```





# Mutex Lock

---

- Mutex Lock
  - acquire/release are atomic
  - often implemented using one of the hardware mechanisms
  - requires busy waiting
    - ▶ spinlock
      - any other process trying to enter critical section must wait (“spins”)
      - disadvantage: wastes CPU cycles
      - advantage: no context switch





# 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) { // originally P Dutch proberen “to test”`

```
    while S <= 0
```

```
        ; // no-op
```

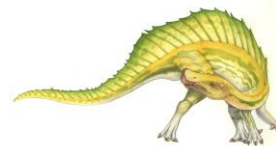
```
        S--;
```

```
    }
```

- `signal (S) { // originally V Dutch verhogen “to increment”`

```
    S++;
```

```
    }
```





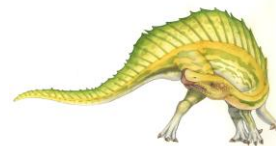


# Semaphore as General Synchronization Tool

---

- **Counting** semaphore – integer value can range over an unrestricted domain
- **Binary** semaphore – integer value can range only between 0 and 1;
  - 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 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

*semaphore data structure in C*

```
typedef struct semaphore
{
    int value;
    struct process *list;
};
```





# 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

