# Illustrate the working principles of semaphore in critical section

# Critical-Section Handling in OS

- Two approaches depending on if kernel is pre emptive or non pre-emptive
- Pre-emptive – allows pre emption of process when running in kernel mode
- Non-preemptive – runs until exits kernel mode, blocks, or voluntarily yields CPU

➢ Essentially free of race conditions in kernel mode

# 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

# Synchronization Hardware

- 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

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

- Mutual Exclusion with swap function

```
do {
   Key = true;
   While (key==true)
   Swap (&lock,&key);
//Critical section
Lock = false;
)while = true;
```

# Mutex Locks

- Lock State & Operations
- Locks have two states

  Held : Someone in the critical section

  Not held: Nobody in the critical section
- Two Operation :

  Acquire : Mark lock as held or wait until release

  Release : Mark lock as un held

# Mutex Locks

- Previous 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

# 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 usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider $P_1$ and $P_2$ that require $S_1$ to happen before $S_2$

  Create a semaphore "`synch`" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

- Can implement a counting semaphore $S$ as a binary semaphore

# Semaphore Implementation

```
do
{
Wait (mutex);
//Critical section//
Signal(mutex);
//remainder section
}
While (true)
```

# Semaphore Implementation

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

# Semaphore implementation

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

signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```

# Implementation

- Semaphore operations now defined as

*wait*(*S*):

```
S.value--;
if (S.value <= 0) {
        add this process to S.L;
        block;
}
```

*signal*(*S*):

```
S.value++;
if (S.value <= 0) {
        remove a process P from S.L;
        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.