Synchronization

Hardware solutions

Stefano Quer and Pietro Laface
Dipartimento di Automatica e Informatica
Politecnico di Torino

Using the interrupt mechanism

- Use
  - **Disable** interrupt in the reservation section
  - **Enable** interrupt in the release section
    - Used only inside the kernel, and for short sections
    - In multi-processor (multi-core) the interrupts must be disabled on all processors

Enabling and disabling interrupts are privileged instructions

```
while (TRUE) {
  disable interrupt
  CS
  enable interrupt
  non critical section
}
```

Using the lock mechanism

- An alternative strategy is to simplify the software solutions, using locking mechanisms supported by the hardware, by means of
- A **lock** can be uses to protect a CS
  - The lock value allows or prohibits access to the CS
- An **indivisible instruction** executed in a single "memory cycle", which
  - Cannot be interrupted
  - Allows testing and simultaneous setting of a shared variable

Using the lock mechanism

- Two main atomic lock instructions exist
  - **Test-And-Set**
    - **Sets to one** and **returns the previous value** of a shared lock variable
    - Executed in a **single indivisible cycle**
  - **Swap**
    - Swaps the content of two variables, one of which is a shared lock
    - Executed in a **single indivisible cycle**

Test-And-Set

Receives, the pointer to the shared lock.
The lock is initialized to FALSE

```
char TestAndSet (char *lock) {
  char val;
  val = *lock;
  *lock = TRUE;
  return val;
}
```

Sets the lock to TRUE,
i.e., locks the CS

Returns the previous value
of the lock

Using Test-And-Set instruction

```
char lock = FALSE;
```

Shared lock variable

```
char TestAndSet (char *lock) {
  char val;
  val = *lock;
  *lock = TRUE;    // Set new lock
  return val;      // Return old lock
}
```

Reservation code:
Test and Set

If lock==TRUE
the CS is busy,
thus waits

```
while (TRUE) {
   while (TestAndSet (&lock));     // lock
   CS
   lock = FALSE;                   // unlock
   Non critical section
}
```

If lock==FALSE
Set lock=TRUE and enter CS

Test-And-Set instruction

```
char lock = FALSE;
```

TestAndSet is atomic

```
char TestAndSet (char *lock) {
  char val;
  val = *lock;
  *lock = TRUE;    // Set new lock
  return val;      // Return old lock
}
```

Busy form of waiting

```
while (TRUE) {
   while (TestAndSet (&lock));     // lock
   SC
   lock = FALSE;                   // unlock
   sezione non critica
}
```

Swap

Receives, the pointer to the shared lock and to a local lock variable.
Il shared lock initialized to FALSE

```
void swap (char *v1, char *v2) {
  char = *tmp;

  *tmp = *v1;
  *v1 = *v2;
  *v2 = *tmp;
  return;
}
```

Performs the **atomic** exchange

Using swap

```
void swap (char *v1, char *v2) {
  char = *tmp;

  *tmp = *v1;
  *v1 = *v2;
  *v2 = *tmp;
  return;
}
```

```
char lock = FALSE;
```

Shared lock variable

swap is atomic

Setting key=TRUE reserve the CS

If lock==FALSE the CS is free, set key=FALSE, lock =TRUE, and enter the CS

If lock==TRUE wait

```
while (TRUE) {
  key = TRUE;
  while (key==TRUE)
    swap (&lock, &key); // Lock
  CS
  lock = FALSE;          // Unlock
  non critical section
}
```

Using swap

```
void swap (char *v1, char *v2) {
  char = *tmp;

  *tmp = *v1;
  *v1 = *v2;
  *v2 = *tmp;
  return;
}
```

```
char lock = FALSE;
```

Busy form of waiting

```
while (TRUE) {
  key = TRUE;
  while (key==TRUE)
    swap (&lock, &key); // Lock
  CS
  lock = FALSE;          // Unlock
  non critical section
}
```

Mutual exclusion without starvation

- The previous techniques
  - Are symmetric
  - Ensure mutual exclusion
  - Ensure progress, avoiding the deadlock
  - Cannot ensure starvation
- To avoid starvation
  - Extend the previous solution
  - The solution illustrated uses TestAndSet
    - It is due to Burns [1978]

Mutual exclusion without starvation

A reservation vector, with an element per thread, initialized to FALSE

T_i

```
while (TRUE) {
  waiting[i] = TRUE;
  key = TRUE;
  while (waiting[i] && key)
    key = TestAndSet (&lock);
  waiting[i] = FALSE;
  CS
  j = (i+1) % N;
  while ((j!=i) and (waiting[j]==FALSE))
    j = (j+1) % N;
  if (j==i)
    lock = FALSE;
  else
    waiting[j] = FALSE;
  non critical section
}
```

Single shared lock initialized to FALSE

Mutual exclusion without starvation

T_i

```
while (TRUE) {
  waiting[i] = TRUE;
  key = TRUE;
  while (waiting[i] && key)
    key = TestAndSet (&lock);
  waiting[i] = FALSE;
  CS
  j = (i+1) % N;
  while ((j!=i) and (waiting[j]==FALSE))
    j = (j+1) % N;
  if (j==i)
    lock = FALSE;
  else
    waiting[j] = FALSE;
  non critical section
}
```

Enter the CS if it is free lock=FALSE → key=FALSE or waiting[i] has been set to FALSE by another thread

Releasing the SC set lock= FALSE if no thread is waiting

Otherwise yield the lock to a waiting thread by setting waiting[j]=FALSE

Conclusions

- Advantages of hardware solutions
  - Can be used in multi-processor environments
  - Easily extensible to N threads
  - Easy to use from the user point of view
  - Symmetric

Conclusions

- Disadvantages of hardware solutions
  - Busy form of waiting with spin-lock
  - Possible starvation
  - Priority inversion: a higher priority task is preempted by a lower priority one.
    - Consider two threads H and L, of high and low priority, respectively, accessing a resource in mutual exclusion.
    - L is in its CS, H is blocked outside until L exits its CS.
    - If a third thread M of medium priority becomes ready, it preempts L, thus L does not leave its CS promptly, causing H, the highest priority process, to remain blocked.