Synchronization

Hardware solutions

Stefano Quer, Pietro Laface, and Stefano Scanzio
Dipartimento di Automatica e Informatica
Politecnico di Torino
skenz.it/os stefano.scanzio@polito.it

Hardware solutions

- Hardware solutions to the CS problem can be classified as follows:
  - Solutions for systems that **do not allow preemption**
  - Solutions for systems that **allow preemption**
    - Solutions based on **interrupts** management
    - Solutions based on an "extension" of software solutions, or based on
      - Some kind of **lock**
      - Some kind of atomic **instruction**

This aspect is complicated by the presence of multiprocessor or multi-core systems

Systems without preemption

- In a system **without preemption**
  - The P (or T) in execution in the CPU **cannot** be interrupted
  - The control is released from the P (or T) to the kernel only in a **voluntary** way

The CPU **cannot** be subtracted (preempted) from a P (or T), which is in the running state

Systems without preemption

- In **mono-processor** systems **without preemption**
  - The CS problem does not exist, because only a P (or T) can use the only CPU at a certain time, and this P (or T) cannot be interrupted
- However, this situation rarely occurs because
  - Systems are often multi-processor or multicore, and even without preemption the parallelism is effective: i.e., distinct processors or cores can concurrently execute more than one P (or T)
  - Kernels **without preemption** are not secure, have excessive response times, and are not suitable for "real-time"

Systems with preemption

- In a system **with preemption**
  - A running process can be interrupted
  - As a matter of fact, the operating system or the arrive of an **interrupt** changes/preempts the control flow to another process
  - The original process will be terminated later

The CPU can be subtracted from a running P (or T)

Using the interrupt mechanism

- In **mono**-processor system **with preemption**
  - It is possible to solve CS problem with interrupts
    - **Disable** interrupts in the reservation section
    - **Enable** interrupts 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 interrupt mechanism

- In general, disabling interrupts has several disadvantages
  - The procedure is inherently insecure
    - What happens if to a user process is given the right to disable interrupts, and that process has an incorrect behavior?
    - This opportunity can be provided only to kernel level processes (super-user)
  - In multi-processor (multi-core) systems it is necessary to disable the interrupt on all processors
    - The interrupt disabling request must be sent
    - Long processing times are needed
    - System management becomes **non** real-time

Using lock-unlock mechanisms

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

Using lock-unlock mechanisms

- 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 of type char or int (but just one bit / byte is enough) 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: disadvantages

```
char lock = FALSE;
```

TestAndSet must be atomic

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

Busy form of waiting over a spin-lock: consumes CPU cycles while it waits

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

Swap

Receives the pointer to the shared lock and to a local lock variable. The 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==TRUE wait

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

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

Swap: disadvantages

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

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

```
char lock = FALSE;
```

The swap procedure must be atomic

Busy form of waiting over a spin-lock: consumes CPU cycles while it waits

```
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
  - Ensure mutual exclusion
  - Ensure progress, avoiding the deadlock
  - They do **not** ensure the definite waiting for a process, or they do **not** guarantee non-starvation
  - Are symmetric
- To avoid starvation
  - Previous solution must be extended
  - The following solution is derived from TestAndSet
    - It is due to Burns [1978]

Slow T/P never enter the CS because the fast ones keep it busy

Mutual exclusion without starvation

T_i

A reservation vector, with an element per T/P, initialized to FALSE

```
while (TRUE) {
  waiting[i] = TRUE;
  while (waiting[i] && TestAndSet (&lock));
  waiting[i] = FALSE;
  CS
  j = (i+1) % N;
  while ((j!=i) && (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

The T/P in the queue enter the SC because they receive the entering opportunity from the previous one

Mutual exclusion without starvation

T_i

```
while (TRUE) {
  waiting[i] = TRUE;
  while (waiting[i] && TestAndSet (&lock));
  waiting[i] = FALSE;
  CS
  j = (i+1) % N;
  while ((j!=i) && (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 → return TRUE or waiting[i] has been set to FALSE by another T/P

Releasing the SC set lock= FALSE if no T/P is waiting

Otherwise yield the lock to a waiting T/P 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 software/user point of view
  - Symmetric

Conclusions

- Disadvantages of hardware solutions
  - Not easy to implement at the hardware level
    - Need atomic operations on global variables
  - Possible starvation
    - The selection of processes for entering the CS using busy-waiting is arbitrary, and managed by the processes and not by the SO
  - Busy waiting on spin-lock
    - Waste of resources (i.e., CPU cycles) for waiting
      - In practice, busy-waiting is used only for very short waiting

Conclusions

- Priority inversion: a higher priority task is preempted by a lower priority task.
  - 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.
- A possible solution to this problem is to use the priority inheritance protocol
  - A process holding a lock automatically inherits the priority of the process with the higher priority waiting for the same lock