Operating Systems

Karim Lounis

Queen’s University, Canada

Winter’19, Jan-Apr 2019
Semaphores

Definition

Semaphore: Allows two or more processes or threads to communicate in order to be executed in a correct way (order).

- Implemented by an integer variable which modification can only be performed via two primitives:
  1. acquire() (or P() or wait()): Tests whether the semaphore S is greater or equal to 1 and decrements its value if so, else the process spins around.
  2. release() (or V() or signal()): Increments the value of S.

```c
acquire()
{
    while(S <= 0);
    S = S - 1;
}
```

```c
release()
{
    S = S + 1;
}
```
Semaphores

Two main types of semaphores:
- Binary semaphore are used when the semaphore value ranges only between 0 and 1 (i.e., Mutex lock).
- Counting semaphore are used when the semaphore value ranges over an unrestricted domain (i.e., $s \in \mathbb{Z}$).

Two styles of use:
- Mutual exclusion style, in which case $s_{init} = n \mid n \geq 1$.
- Processes waiting style, in which case $s_{init} = 0$.

Different implementations exist, with busy waiting and without busy waiting.

A process can block itself with a system primitive block() and waken another process $P$ with a primitive wakeup($P$).

Atomic execution of acquire() and release() is possible by: inhibiting interrupts in single-processor systems, and by disabling interrupts in all processors in a multiprocessor system.
Example (Waiting style) Let Thread 1 and Thread 2 be two Java threads such that T1 should be executed after the termination of T2:

Thread 1

Code 1;

Thread 2

Code 2;

Such that:

Thread Thread_1 = new Thread();

Thread Thread_2 = new Thread();

Thread_1.start();

Thread_2.start();
Example (Waiting style) Let T1 and T2 be two Java threads such that T1 should be executed after the termination of T2.

Thread 1

acquire(S);

Code 1;

Thread 2

Code 2;

release(S)

We use a semaphore S initialized to 0:

Semaphore S = 0;
Semaphores

Example (counting semaphore) Consider a petrol station with 6 petrol pumps, and a flow of vehicles is getting into the station to pump gas/diesel:

```
Vehicle_i

Begin

    acquire(pump);
    Pump_gas() || Pump_diesel();
    release(pump);

End
```

What is the initial value of the semaphore `pump`?
Semaphores

Example (counting semaphore) Consider a petrol station with 6 petrol pumps, and a flow of vehicles is getting into the station to pump gas/diesel:

Vehicle\_i

Begin

acquire(pump);

Pump\_gas() || Pump\_diesel();

release(pump);

End

The semaphore pump should be initialized to 6.
To eliminate the busy waiting, semaphores are often implemented as follow:

1. An integer (denoted $s$) that holds the number of instances of a given resource available for processes to use simultaneously.

2. A queue (denoted $q$) where blocked processes wait for the availability of a resource (critical section).

The semaphore can only be modified by two atomic primitives:

1. `acquire()`: tells the currently running process or thread to claim the control of a resource or make it wait if that resource is not available.

2. `release()`: Allows a process or thread to inform other processes or threads that it is done with that resource and wakes up a blocked process or thread in the queue if there is any.
Semaphores

- **acquire()** decrements the value of $s$ then: if $s < 0$ the “current” process is blocked and placed in the queue $q$. Else it goes on.

- **release()** increments the value of $s$ then: if $s \leq 0$ the “blocked” process in the head of the queue is dequeued and resumed.

```c
acquire(s) {
    s = s - 1;
    if(s<0) {
        Block & place P in q;
    }
}
```

```c
release(s) {
    s = s + 1;
    if(s\leq0) {
        Wake up P from q;
    }
}
```
Binary Semaphores

A binary semaphore is a semaphore which value can be 0 or 1.

- **acquire(s)** nullifies the value of s and place all other processes to the waiting queue q (semaphore queue of PCBs).
- **release(s)** makes the critical section available by setting the value of s to one or releasing a “blocked” process from the queue.

```plaintext
acquire(s)                    release(s)
{
    if(s==1) s = 0;
    If(q is ∅) s = 1;
    Else
        Block & place P in q;
        Wake up P from q;
}
```

- Each process, once dequeued and resumed will start executing its critical section.
**Example.** Let $i$ be a shared variable between two Java threads 1 & 2, where $i$ is initially set to 13, and $s$ is a binary semaphore initialized to 1:

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>acquire(s);</code></td>
<td><code>acquire(s);</code></td>
</tr>
<tr>
<td>$i = i + 10;$</td>
<td>$i = i - 10;$</td>
</tr>
<tr>
<td><code>release(s);</code></td>
<td><code>release(s);</code></td>
</tr>
</tbody>
</table>

This time the value of $i$ is deterministic and it is equal to 13.
History of software solutions to the critical section problem

- In 1962, the critical section problem was posed.
- In 1965, there was Dekker’s Algo for two processes.
- In 1966, There was the first correct solution for n-processes.
- In 1968, THE OS by Dijkstra was the first OS that uses semaphores.
- In 1972, Eisenberg & McGuire’s algo, a correct solution for n-processes.
- In 1974, Lamport’s Bakery algo, a correct solution for n-processes.
- In 1981, Peterson’s algo for two processes.
Peterson’s Algorithm

Definition

Peterson’s Algo is a solution for the critical section problem involving two processes that alternate their CS execution. It was proposed in 1981.

do {
    flag[i]=true;
    turn = 1 - i;
    while (flag[1-i] && turn==1-i); Structure of process $P_i$
    CS
    flag[i]=false;
    ...
} while(true);
Recall. The Critical Section Problem is all about designing a protocol that processes can apply to cooperate. Such protocol must satisfy the following three requirements:

- **Mutual exclusion**: At most, one process is executing its critical section at a time.
- **Progress**: A process must leave the critical section once it is done with it, and must notify other processes about that.
- **Bounded waiting**: A process must not wait indefinitely in the queue to access its critical section.

Paterson’s algorithms fulfills these three requirements.
Recall. The Critical Section Problem is all about designing a protocol that processes can apply to cooperate. Such protocol must satisfy the following three requirements:

- **Mutual exclusion**: At most, one process is executing its critical section at a time.
- **Progress**: A process must leave the critical section once it is done with it, and must notify other processes about that.
- **Bounded waiting**: A process must not wait indefinitely in the queue to access its critical section.

Paterson’s algorithms fulfills these three requirements.

But has busy waiting
Hardware solutions for CS

**Easy but not efficient:**

Certain OSs running on single-processor systems disable interrupts when a shared variable is being manipulated.

*e.g.*, Early versions of UNIX did that.

**Modern solution:**

Certain computers provide a machine-code instruction that performs indivisible read-write operation on some memory location.

*e.g.*, 8086 has LOCK instruction prefix to lock access to memory when executing XCHG instruction.
Hardware solutions for CS

Consider the following 8086 assembly code to be executed in a single-processor system where [0xF457] is the address of a semaphore variable:

```
Mov Al, 0
Label: XCHG [0XF457], Al
Test Al, Al
Jz Label
... CS ...
Mov [0xF457], 1
```

It works fine because: processor cannot switch from one process to another in the middle of an instruction.
Hardware solutions for CS

Consider the following 8086 assembly code to be executed in a single-processor system where [0xF457] is the address of a semaphore variable:

```
Mov Al, 0
Label: XCHG [0xF457], Al
    Test Al, Al
    Jz Label
... CS ...
Mov [0xF457], 1
```

But fails in a multiprocessor system
Hardware solutions for CS

Consider the following 8086 assembly code to be executed in a single-processor system where [0xF457] is the address of a semaphore variable:

```
  Mov Al, 0
Label:  LOCK XCHG [0XF457], Al
  Test Al, Al
  Jz Label
  ... CS ...
  Mov [0xF457], 1
```

Will work now on multiprocessor system. By using LOCK the processor will have exclusive use of the bus till XCHG is completely executed (two cycles)
Hardware solutions for CS

To understand how these instructions work, we use an abstract form called test\_and\_set (an instruction executed atomically & in mutual exclusion):

```plaintext
boolean test\_and\_set(boolean *target)
{
    boolean rv = *target;
    *target = true;
    return rv;
}
```

The program structure will be:

```plaintext
boolean lock=false;
while(1){ while(test\_and\_set(&lock)); CS; lock=false; ...; }
```
End.