Operating Systems

Karim Lounis

Queen’s University, Canada

Winter’19, Jan-Apr 2019
Critical section problem
Critical Section

Definition

**Critical Section:** It is a part of a program code in which the program requests to use shared resources on which the access is mutually exclusive.

Any program that uses a shared resource is generally organized into three to five parts:

```c
Program()
{
[ Remainder section ]
Entry section
  Critical section
Exit section
[ Remainder section ]
}
```

- In the entry section, the process places a request to the OS to grant it access to the critical section.
- In the exit section, the process informs the OS of leaving the critical section to wake up any waiting processes.
Critical Section

Let Thread 1 and Thread 2 be two Java Threads:

**Thread 1**
- \( u = v + 50; \)
- \( i = i + 15; \)
- \( w = v \% 20; \)
- \( v = i / 10; \)

**Thread 2**
- \( b = p - 51; \)
- \( i = i - 15; \)
- \( z = p \times 90; \)
- \( p = z \% 10; \)

Identify the critical sections for each thread.
Let Thread 1 and Thread be two Java Threads:

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(u = v + 50)</td>
<td>(b = p - 51)</td>
</tr>
<tr>
<td>(i = i + 15)</td>
<td>(i = i - 15)</td>
</tr>
<tr>
<td>(w = v % 20)</td>
<td>(z = p * 90)</td>
</tr>
<tr>
<td>(v = i / 10)</td>
<td>(p = z % 10)</td>
</tr>
</tbody>
</table>
Critical Section

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.

Note that:

- A process may be interrupted during its critical section.
- There is a queue of PCBs.
Synchronization mechanisms
Mutex Locks

Definition

**Mutex locks**: Mutex for *Mutual exclusion*, is the simplest software-based solution for the critical section problem.

- A process must acquire the lock before entering its critical section.
- Two **atomic primitives** are used, the `acquire()` primitive (entry section) and the `release()` primitive (exit section).
- It uses a boolean variable *available* to determine whether the lock is available or not.

```
acquire()       release()
{              }
While(!available);
available=false;  available=true;
{              }
```

Program() {
```
acquire();
CS;
release();
```
Mutex Locks

- The use of Mutex create **busy waiting** (continuous looping).
- Each process acquiring a non available lock spins while waiting.
- No context switch is required when a process must wait on a lock.
- Used in multiprocessor systems where one process can “spin” on one CPU while another performs its critical section on another CPU.
- The primitives acquire() and release() are sometimes called lock() and unlock() respectively.

```c
acquire()            release()          Program() {
{                {                      acquire();
  While(!available);  CS;
  available=false;    available=true;  release();
}                }
```

Karim Lounis (Queen’s University, Canada)
**Mutex Locks**

**Example.** Let \( i \) be a shared variable between two Java threads 1 & 2, where \( i \) is initially set to 5, and \( M \) is a mutex lock initialized to 1:

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

What will be the value of \( i \) when the following program executes:

```java
Thread Thread_1 = new Thread();
Thread Thread_2 = new Thread();
Thread_1.start();
Thread_2.start();
```
Example. Let $i$ be a shared variable between two Java threads 1 & 2, where $i$ is initially set to 5, and $M$ is a mutex lock initialized to 1:

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

This time the value of $i$ is deterministic and it is equal to 5.
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 than 1 and decrements its value, else the process spins around.
  2. `release()` (or `V()` or `signal()`): Increments the value of $S$.

```java
acquire()
{
    while ($S \leq 0$);
    $S = S - 1$;
}
```

```java
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.
Semaphores

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 $T_1$ and $T_2$ be two Java threads such that $T_1$ should be executed after the termination of $T_2$.

**Thread 1**

acquire(S);

Code 1;

**Thread 2**

Code 2;

release(S)

We use a semaphore $S$ initialized to 0:

Semaphore $S = 0$;
End.