Operating Systems

Karim Lounis

Queen’s University, Canada

Winter’19, Jan-Apr 2019
Process Synchronization

Overview

Introduce the critical section problem and process synchronization.

Introduce software and hardware solutions for critical-section problem.

Examine some well-known classical process synchronization problems.

Deadlocks and Deadlock recovery techniques.
What is synchronization? and Why do we care about?
Motivation example. Let $X$ be a shared variable between process 1 and 2:

Process 1
(A) $X := 1$;
(B) ...
(C) print($X$);

Process 2
(D) $X := 2$;
(E) ...
(F) $X := 3$;

What is the value of $X$ that is printed by process 1 when the following program is executing?:

CoBegin
    Process 1;
    Process 2;
CoEnd
Motivation example. Let $X$ be a shared variable between process 1 and 2:

Process 1

(A) $X := 1$;
(B) ...;
(C) print($X$);

Process 2

(D) $X := 2$;
(E) ...;
(F) $X := 3$;

The value of $X$ printed by **process 1** can be: 1, 2, or 3. But which value is most likely?

- Depends on the amount of code in (B) and (E).
- Depends on which process started execution first.
- Depends on the hardware, each process is running on one CPU core.
- ...
Motivation example. Let $X$ be a shared variable between process 1 and 2:

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A) $X := 1;$</td>
<td>(D) $X := 2;$</td>
</tr>
<tr>
<td>(B) . . .;</td>
<td>(E) . . .;</td>
</tr>
<tr>
<td>(C) print($X$);</td>
<td>(F) $X := 3;$</td>
</tr>
</tbody>
</table>

The value of $X$ printed by process 1 can be: 1, 2, or 3. But which value is most likely?

- Depends on the amount of code in (B) and (E).
- Depends on which process started execution first.
- Depends on the hardware, each process is running on one CPU core.
- . . .

$\implies$ The correct value of $X$ will be directed by the logic of your program.
**Another Motivation example.** Let \( i \) be a shared variable between two Java threads 1 & 2, where \( i \) is initially set to 5:

\[
\begin{align*}
\text{Thread 1} & \quad \text{Thread 2} \\
\text{i = i + 1;} & \quad \text{i = i - 1;} 
\end{align*}
\]

What is the value of \( i \) when the following program terminates?:

```
Thread Thread_1 = new Thread();
Thread Thread_2 = new Thread();
Thread_1.start();
Thread_2.start();
```
**Another Motivation example.** Let $i$ be a shared variable between two Java threads 1 & 2, where $i$ is initially set to 5:

- **Thread 1**
  
  $i = i + 1;$

- **Thread 2**
  
  $i = i - 1;$

The value of $i$ can be: 4, 5, or 6. But which value is most likely?

- Depends on which thread is started first.
- Depends on which thread has the higher priority.
- Depends on the hardware, each thread is running on one CPU core.
Another Motivation example. Let $i$ be a shared variable between two Java threads 1 & 2, where $i$ is initially set to 5:

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i = i + 1;$</td>
<td>$i = i - 1;$</td>
</tr>
</tbody>
</table>

The value of $i$ can be: 4, 5, or 6. But which value is most likely?

- Depends on which thread is started first.
- Depends on which thread has the higher priority.
- Depends on the hardware, each thread is running on one CPU core.

$\implies$ Again: The correct value of $i$ will be directed by the logic of your program.
Process Synchronization

Example: Program logic semantic and shared data corruption:

```
Producer()
{
    BUFFER_SIZE = n;
    counter, in = 0;
    While (true)
    {
        /*Produce an item in X*/
        While (counter == BUFFER_SIZE)
        ; /* do nothing*/
        Buffer[in] = X;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
    }
}

Consumer()
{
    BUFFER_SIZE = n;
    counter, out = 0;
    While (true)
    {
        /*concume the item in Y*/
        While (counter == 0)
        ; /* do nothing*/
        Y = Buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
    }
}
```
Let us write the two statements "counter++;" & "counter--;" in machine language, where Ax is the accumulator register & [0xB4FF55EF] is the address of the variable counter in the main memory.

For "counter++;" :

\[
\begin{align*}
I_0: & \quad \text{Mov Ax, [0xB4FF55EF]} \\
I_1: & \quad \text{Inc Ax} \\
I_2: & \quad \text{Mov [0xB4FF55EF], Ax}
\end{align*}
\]

For "counter--;" :

\[
\begin{align*}
I_3: & \quad \text{Mov Ax, [0xB4FF55EF]} \\
I_4: & \quad \text{Dec Ax} \\
I_5: & \quad \text{Mov [0xB4FF55EF], Ax}
\end{align*}
\]
Process Synchronization

When the two processes are concurrently executed, their lower-level instructions are interleaved in some order e.g., \([\text{counter}(t) = n]\)

\[\begin{align*}
&l_0: \ P_p \text{ executes } \text{Mov} \ Ax, \ [0xB4FF55EF] \quad \{Ax = n\} \\
&l_1: \ P_p \text{ executes } \text{Inc} \ Ax \quad \{Ax = n + 1\} \\
&l_2: \ P_c \text{ executes } \text{Mov} \ Ax, \ [0xB4FF55EF] \quad \{Ax = n\} \\
&l_3: \ P_c \text{ executes } \text{Dec} \ Ax \quad \{Ax = n - 1\} \\
&l_4: \ P_p \text{ executes } \text{Mov} \ [0xB4FF55EF], \ Ax \quad \{\text{counter} = n + 1\} \\
&l_5: \ P_c \text{ executes } \text{Mov} \ [0xB4FF55EF], \ Ax \quad \{\text{counter} = n - 1\}
\end{align*}\]

The final state is \(\text{counter} = n - 1\) (or \(\text{counter} = n + 1\) if we switch \(l_4 \& l_5\)) which is an incorrect state, since the correct value for counter should be \(n\) if the producer then the consumer (or the reverse) are executed separately.
Process Synchronization

When the two processes are concurrently executed, their lower-level instructions are interleaved in some order e.g., \( \text{counter}(t) = n \)

\[
\begin{align*}
I_0: & \quad P_p \text{ executes } \text{Mov} \ Ax, [0xB4FF55EF] \quad \{ Ax = n \} \\
I_1: & \quad P_p \text{ executes } \text{Inc} \ Ax \quad \{ Ax = n + 1 \} \\
I_2: & \quad P_c \text{ executes } \text{Mov} \ Ax, [0xB4FF55EF] \quad \{ Ax = n \} \\
I_3: & \quad P_c \text{ executes } \text{Dec} \ Ax \quad \{ Ax = n - 1 \} \\
I_4: & \quad P_p \text{ executes } \text{Mov} \ [0xB4FF55EF], Ax \quad \{ \text{counter} = n + 1 \} \\
I_5: & \quad P_c \text{ executes } \text{Mov} \ [0xB4FF55EF], Ax \quad \{ \text{counter} = n - 1 \}
\end{align*}
\]

The final state is \( \text{counter} = n - 1 \) (or \( \text{counter} = n + 1 \) if we switch \( I_4 \) & \( I_5 \)) which is an incorrect state, since the correct value for counter should be \( n \) if the producer then the consumer (or the reverse) are executed separately.

Problem of shared data corruption (compromising data coherency).
Problem of shared data corruption is due to:

- Processes can execute in (fake) concurrency i.e., any process can be interrupted at any point in its instruction flow (e.g., quantum expires or I/O operation), and the CPU is assigned to another process.

- Processes can execute in (real concurrency) parallel (e.g., two instruction flows of two different processes are simultaneously executing on separate processing cores).

- These concurrent or parallel processes are sharing some data.

The concurrent or parallel execution of multiple processes may result in the corruption of shared data by among several processes.
A multiprocessing system may reach an **incorrect state** if it allows the manipulation of shared data concurrently by multiple processes.

**race condition**

Race condition is when: (1) Multiple processes are being executed concurrently (2) These processes share at least a common variable and (3) The outcome of the execution depends on the order in which the processes modified the shared variable.
Process Synchronization

A multiprocessing system may reach an **incorrect state** if it allows the manipulation of shared data concurrently by multiple processes.

**race condition**

Race condition is when: (1) Multiple processes are being executed concurrently (2) These processes share at least a common variable and (3) The outcome of the execution depends on the order in which the processes modified the shared variable.

**Synchronization**

A way to ensure that only one process at a time can manipulate a shared variable.
Process Synchronization

A multiprocessing system may reach an incorrect state if it allows the manipulation of shared data concurrently by multiple processes.

race condition

Race condition is when: (1) Multiple processes are being executed concurrently (2) These processes share at least a common variable and (3) The outcome of the execution depends on the order in which the processes modified the shared variable.

Synchronization

A way to ensure that only one process at a time can manipulate a shared variable.

How can we solve such problem? Can we use boolean?
Synchronization with booleans Let us consider the previous example of two threads, and let us use a boolean variable free initially set to true:

Thread 1
While(!free);
free=false;
i = i + 1;
free=true;

Thread 2
While(!free);
free=false;
i = i - 1;
free=true;
Synchronization with booleans Let us consider the previous example of two threads, and let us use a boolean variable free initially set to true:

**Thread 1**
- While(!free);
- free=false;
- i = i + 1;
- free=true;

**Thread 2**
- While(!free);
- free=false;
- i = i - 1;
- free=true;

This solution often works but fails sometimes (same time execution).
Synchronization with booleans Let us consider the previous example of two threads, and let us use two twisted boolean variable `free1` and `free2` both initially set to true:

**Thread 1**

1. `free1=false;`
2. `While(!free2);`
3. `i = i + 1;`
4. `free1=true;`

**Thread 2**

1. `free2=false;`
2. `While(!free1);`
3. `i = i - 1;`
4. `free2=true;`
Synchronization with booleans Let us consider the previous example of two threads, and let us use two twisted boolean variable free1 and free2 both initially set to true:

**Thread 1**

free1=false;

While(!free2);

i = i + 1;

free1=true;

**Thread 2**

free2=false;

While(!free1);

i = i - 1;

free2=true;

This solution often works but fails sometimes (same time execution). This time a deadlock may take place.
Process Synchronization

Definition

**Process Synchronization**: It is a multiprocessing concept that aims to manage and control the execution and the access to shared resources between multiple processes or threads.

- Important when the order of execution matters.
- Allows transparent process communication.
- Preserve data integrity (coherency).
- Mechanisms such as: Mutex, Semaphores, Monitors, Peterson’s solution, and Messages can be used for process synchronization.
End.