

CS 3305

Process Synchronization

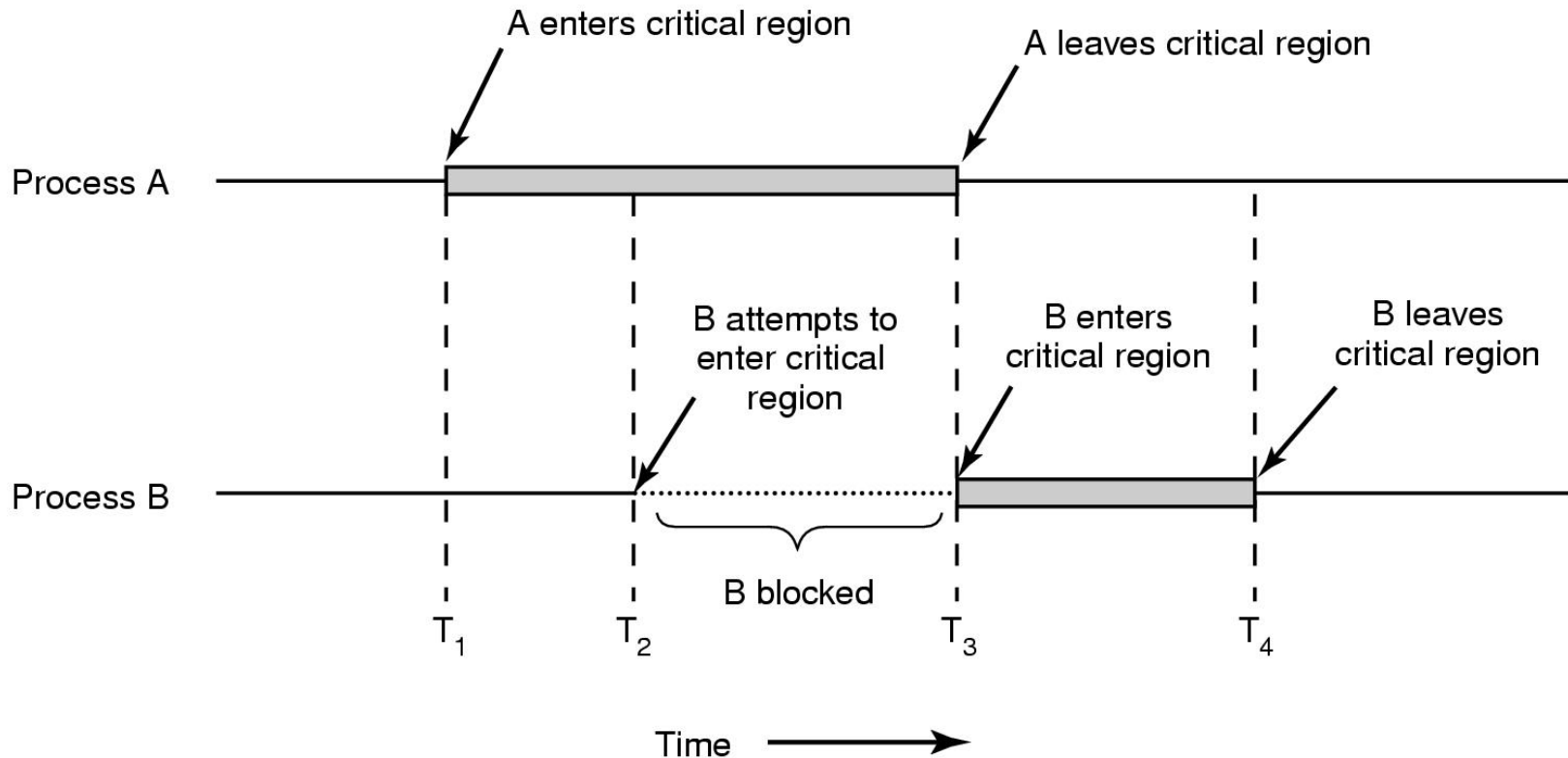
Lecture 12

Oct 23, 2019

Process Synchronization

- ❑ Race Condition
- ❑ Critical Section
- ❑ Mutual Exclusion
- ❑ Peterson's Solution
- ❑ Disabling Interrupts
- ❑ Test and Lock Instruction (TSL)
- ❑ Semaphores
- ❑ Deadlock



Mutual Exclusion in Critical Sections



General structure for Mutual Exclusion

```
{  
    entry section (set FLAG to 1)  
        critical section  
    exit section (set FLAG to 0)  
  
    remainder section  
}
```

Race Condition

```
bank_example (account, amount_to_withdraw)
{
     FLAG = 1
    1.  balance = get_balance(account);
    2.  if (balance >= amount_to_withdraw)
        withdraw_authorized();
    else
        withdraw_request_denied();
    3.  balance = balance - amount_to_withdraw;
     FLAG = 0
}
```

What happens if both processes request that \$600 be withdrawn?

Peterson's Solution

- ❑ Solution developed in 1981
- ❑ Considered revolutionary at the time
- ❑ Restricted to **two processes**

Peterson's Solution

- Two variables are shared
 - `int turn;`
 - `int flag[2]`
- The `flag` array is used to indicate if a process is ready to enter the critical section. `flag[i] = true` implies that process P_i is ready!

Peterson's Solution

flag [0] = false; flag[1] = false; turn;

For process i:

{

 flag[i] = true;

 turn = j;

 while (flag[j] && turn == j);

 critical section

 flag[i] = false;

 remainder section

}

Only way P_i enters critical section is when $\text{flag}[j] == \text{FALSE}$

Peterson's Solution

flag [0] = false; flag[1] = false;

Process 0

```
{
    flag[0] = true;
    turn = 1;
    while (flag[1] && turn == 1);
        critical section
    flag[0] = false;
    remainder section
}
```

Process 1

```
{
    flag[1] = true;
    turn = 0;
    while (flag[0] && turn == 0);
        critical section
    flag[1] = false;
    remainder section
}
```

(a) Mutual Exclusion (b) Progress (c) Bounded Wait

Possible OS Mechanisms

- ❑ Today the assumption is that the OS will provide some sort of support for mutual exclusion
 - ❑ Hardware solutions e.g., Disable interrupts
 - ❑ Software solutions e.g., locks, semaphores

Mutual Exclusion via Disabling Interrupts

- ❑ Process disables all interrupts before entering its critical region
- ❑ Enables all interrupts just before leaving its critical region
- ❑ CPU is switched from one process to another only via clock / interrupts
- ❑ So disabling interrupts guarantees that there will be no process switch

Mutual Exclusion via Disabling Interrupts

How would this look like for the bank account example?

```
bank_example (account, amount_to_withdraw)
{
  disable(interrupts);
  1.   balance = get_balance(account);

  2.   if (balance => amount_to_withdraw)
        withdraw_authorized();
      else
        withdraw_request_denied();

  3.   balance = balance - amount;
  enable(interrupts);
}
```

Mutual Exclusion via Disabling Interrupts

- ❑ Disadvantage:

- ❑ Gives the power to control interrupts to user (what if a user turns off the interrupts and never turns them on again?)
- ❑ Does not work in the case of multiple CPUs. Only the CPU that executes the disable instruction is effected.

Test and Set Lock Instruction (TSL)

- ❑ Many computers have the following type of instruction: `TSL REGISTER, LOCK`
 - ❑ Reads `LOCK` (initial value of 0) into register `REGISTER`
 - ❑ Stores a nonzero value at the memory location `LOCK`

General structure for Mutual Exclusion

```
{  
    entry section (set FLAG to 1)  
        critical section  
    exit section (set FLAG to 0)  
  
    remainder section  
}
```

Using the TSL Instruction

enter_region


TSL REGISTER, LOCK

CMP REGISTER, #0

JNE enter_region

Critical Section

The TSL instruction
copies the value of
LOCK to REGISTER;
and places non-zero
value to LOCK



Using the TSL Instruction

enter_region

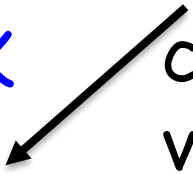
TSL REGISTER, LOCK

CMP REGISTER, #0

JNE enter_region

Critical Section

The CMP command checks to see if the value of REG is 0



Using the TSL Instruction

enter_region

TSL REGISTER, LOCK

CMP REGISTER, #0

JNE enter_region

Critical Section

- If REG value != 0
 - JNE causes control to go to the start of enter_region
- If REG value is zero then enter CS

Using the TSL Instruction

leave_region

MOVE LOCK, #0

← Move value of 0 to
LOCK

Using the TSL Operation

- ❑ Before entering its critical region, a process calls `enter_region`
- ❑ What if `LOCK` is 1?
 - ❑ Busy wait until lock is 0
- ❑ When leaving the critical section, a process calls `leave_region`

Using the TSL Operation

- ❑ Assume two processes: P_0 and P_1
- ❑ LOCK is initialized to zero
- ❑ Assume that P_0 wants to enter the critical section
- ❑ It executes the TSL instruction.
 - ❑ The register value is 0 which reflects the value of LOCK
 - ❑ LOCK is set to 1

Using the TSL Operation

- ❑ Now P_1 wants to enter the critical section; It executes the TSL instruction
 - ❑ The register value is 1 which reflects the value of LOCK
 - ❑ P_1 cannot enter the critical section
 - ❑ It repeats the TSL instruction and comparison operation until it can get into the critical section
- ❑ P_0 is done with the critical section
 - ❑ LOCK becomes 0
- ❑ The next time P_1 executes the TSL instruction and comparison operation it finds that the register value (which reflects LOCK) is zero. It can now enter the critical section.

Mars Pathfinder

- ❑ Considered flawless until July 4th, 1997 landing on the Martian surface
- ❑ After a few days the system kept resetting causing losses of data
- ❑ Press called it a “software glitch”
- ❑ What was the problem?

Mars Pathfinder

- ❑ Scheduling Problem
 - ❑ Scheduling problem due to a faulty algorithm that gave lower priority process an exclusive access to critical region than the higher priority processes

2003 Northeast Blackout

- ❑ Widespread power outage
 - ❑ Northeastern and Midwestern US
 - ❑ Ontario
- ❑ Cause
 - ❑ Software bug which triggered a race condition in the control software