CS 3305A

CPU Scheduling - Multiprocessor

Lecture 11

Oct 21  2019
Multiple-Processor Scheduling

- So far, we've only dealt with a single processor

- CPU scheduling more complex when multiple CPUs are involved
Multiple-Processor Scheduling

- Asymmetric multiprocessing (master)
- There is one processor that makes the decisions for
  - Scheduling, I/O processing, system activities
  - Other processor(s) execute only user code.
- This is a simple approach due to master-slave model / centralized command model
- Master CPU: Load sharing
Multiple-Processor Scheduling

- Symmetric Multiprocessing (SMP)

- Here, each processor is self-scheduling.

- Share a common ready queue or each processor may have its own private queue of ready processes.

- Most modern operating systems support SMP including Windows XP, Solaris, Linux, and Mac OS X.
CS 3305A

Process Synchronization - I

Lecture 11

Oct 21 2019
Process Synchronization

- Race Condition
- Critical Section
- Mutual Exclusion
- Peterson’s Solution
- Disabling Interrupts
- Test and Lock Instruction (TSL)
- Semaphores
- Deadlock
Race Condition Example (1)

- Assume a spooler directory array (in shared memory) has a number of slots
  - Numbered 0, 1, 2...
  - Each slot has a file name
- Two other variables:
  - In points to the first empty slot where a new filename can be entered.
  - Out points to the first non-empty slot, from where the spooler will read a filename and print the corresponding file.
Race Condition Example (1)

- Slots 1, 2, 3 are empty indicating the files in those slots have printed
- Each process has a local variable `next_free_slot` representing an empty slot
- Assume both process A and B want to print files.
  - Each wants to enter the filename into the first empty slot in spooler directory
Race Condition Example (1)

- Process A reads IN and stores the value 7 in its local variable, `next_free_slot`
- Process A’s time quanta expires
- Process B is picked as the next process to run
Race Condition Example (1)

- Process B reads IN and stores the value 7 in its local variable, `next_free_slot`
- Process B writes a filename to slot 7
Race Condition Example (1)

- **Process B** then updates the **In** variable to 8
- **Process A** now gets control of the CPU
Race Condition Example (1)

- Process A writes its filename to slot 7 which erases process B's filename.
- Process B does not get its file printed.
Race Condition (2)

- Application: Withdraw money from a bank account
- Two requests for withdrawal from the same account comes to a bank from two different ATM machines
- A thread for each request is created
- Assume a balance of $1000
Race Condition (2)

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

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

  3. balance = balance - amount_to_withdraw;
}

What happens if both processes request that $600 be withdrawn?
**Race Condition (2)**

<table>
<thead>
<tr>
<th>Process / Thread 1</th>
<th>Process / Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Read balance: $1000</td>
<td></td>
</tr>
<tr>
<td>2. Withdraw authorized for $600 (now actual balance is $400)</td>
<td></td>
</tr>
<tr>
<td><strong>CPU switches to process 2</strong></td>
<td>3. Read Balance $1000</td>
</tr>
<tr>
<td></td>
<td>4. Withdraw authorized for $600 <em>(this is unreal!!)</em></td>
</tr>
<tr>
<td>5. Update balance $1000-$600 = $400</td>
<td><strong>CPU switches to process 1</strong></td>
</tr>
<tr>
<td><strong>CPU switches to process 2</strong></td>
<td>6. Update balance $400-$600 = $-200</td>
</tr>
</tbody>
</table>
Critical Sections and Mutual Exclusion

- A critical section is any piece of code that accesses shared data
  - Printer example: In, Out variables are shared
  - Bank account: Balance is shared

- Mutual exclusion ensures that only one thread/process accesses the critical section at a time i.e., No two processes simultaneously in critical section!
Mutual Exclusion in Critical Sections

Process A

A enters critical region

A leaves critical region

Process B

B attempts to enter critical region

B blocked

B enters critical region

B leaves critical region

Time
General structure for Mutual Exclusion

Do {
    entry section
    critical section
    exit section
    remainder section
} while(1)