Operating Systems and Multicore Programming (1DT089)

Introduction to Operating Systems (Chapter 1)

Wednesday January 22

Uppsala University 2013

karl.marklund@it.uu.se
Chapter 1: Introduction

Chapter objectives:

★ To provide a grand tour of the major components of operating systems.

★ To describe the basic organization of computer systems.
From transistor to logical gates
How can transistors be used to construct logical gates?
An *inverter* - Logical *NOT*

- **$V_{in}$** and **$V_{out}$**
- **Collector**
- **Base**
- **Emitter**
- **$+V_{CC}$**

**Truth Table:**

<table>
<thead>
<tr>
<th>$V_{in}$</th>
<th>$V_{out}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>low</td>
<td>high</td>
</tr>
<tr>
<td>high</td>
<td>low</td>
</tr>
</tbody>
</table>

**Logic Diagram:**

- **A**
- **X**

<table>
<thead>
<tr>
<th>A</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Logical **NOR**

![NOR Circuit Diagram]

<table>
<thead>
<tr>
<th>$V_1$</th>
<th>$V_2$</th>
<th>$V_{out}$</th>
<th>A or B</th>
<th>$a \text{ nor } b = \text{ not } (A \text{ or } B)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Low $\rightarrow$ 0  High $\rightarrow$ 1
Logical **NAND**

### NAND Truth Table

<table>
<thead>
<tr>
<th>$V_1$</th>
<th>$V_2$</th>
<th>$V_{out}$</th>
<th>A and B</th>
<th>A nand B = not (A and B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Low $\rightarrow$ 0  High $\rightarrow$ 1
Ripple adder
Using a few logical gates, we can construct circuits capable of adding binary numbers.
The von Neuman Model

Central Processing Unit

Input devices

Output devices

Control unit

Arithmetic logic unit

Memory

External storage

ALU

A → B → D

F → R
A quick recap of computer architecture
Operating System

Controls the hardware and coordinates its use among the various application programs for the various users.
User view and system view

What is an operating system?

User view

System view
## 1.1.1) User View

<table>
<thead>
<tr>
<th>Computer system</th>
<th>Operating System Design Concerns</th>
</tr>
</thead>
</table>
| Single user, one computer (PC). | Designed for one user to monopolize its resources.  
| | The goal is to maximize the work (or play) that the single user is performing.  
| | Designed mostly for *ease of use*.  
| | Some attention paid to performance and none paid to resource utilization. |
| Terminals connected to a mainframe  
Users share the resources and may exchange information. | Designed to maximize resource utilization:  
| | Assure that all available CPU time, memory, and I/O are used efficiently.  
| | Assure that no individual user takes more than her fair share. |
| Workstations connected to networks of other workstations and servers.  
The users have dedicated resources at their disposal, but they also share resources such as networking and servers (file storage, computation, printing). | Designed to compromise between individual usability and resource utilization. |
| Handheld computers (and smartphones).  
Usually standalone units for individual use. | Designed mostly for individual usability, but performance per unit of battery life is important as well. |
1.1.2) System View

From the computer’s point of view, the operating system is the program most intimately involved with the hardware.

The operating system can be seen as a resource allocator:

- CPU time
- Memory space
- File storage space
- I/O devices
- etc...

Facing numerous and possibly conflicting requests for resources, the operating system must decide how to allocate them to specific programs and users.

An operating system "Controls the hardware and coordinates its use among the various application programs for the various user."

An operating systems is a control program, a program that manages the execution of user programs to prevent errors and improper use of the computer.
What is the definition of an operating system?

The common functions of controlling and allocating resources are brought together into one piece of software: the operating system.

The operating system is the program running at all times on the computer, this program is usually called the **kernel**.

The text book used in the course
1.2.1) A typical modern computer system
To ensure orderly access to the shared memory, a memory controller is provided whose function is to synchronize the access to the memory.

The CPU and the device controllers can execute in parallel, competing for memory cycles.
In computer architecture, the combination of the CPU and main memory (i.e. memory that the CPU can read and write to directly, with individual instructions) is considered the "brain" of a computer.

Any transfer of information between the "brain" and other parts of the system is considered IO.
Operating System

Controls the hardware and coordinates its use among the various application programs for the various users.

Bootstrap Program

Kept on chip (ROM or EEPROM), aka firmware.

Small program executed on power up or reboot.

Initializes all aspects of the system, from CPU register to device controllers to memory content.

Locates and loads the kernel into memory for execution.

Kernel

The part of the operating system that is running at all times.

On boot, starts executing the first process such as *init*.

Waits for some event to occur...

Computer Hardware

CPU

Disk Controller

USB Controller

Graphics Adapter

Memory Controller

Memory
How will the kernel know when some event has occurred?

What kind of events?
Presses a key on the keyboard.

**CPU** executing instructions,

While executing, the CPU "listens" for interrupt signals.

Generates an **interrupt** signal

When an interrupt is received, the CPU immediately stops what it is doing and transfers execution to a fixed location in memory.

The CPU resumes execution of the interrupted instruction

---

**Service routine for all interrupts**

Examine the interrupt and jump to interrupt specific service routine.

---

**Service routine for keyboard interrupt**

Some action is taken handling the interrupt.

When done, control is transferred back to the interrupted instruction
Interrupt handling must be fast ...

Only a (small) number of predefined interrupts ...
CPU executing instructions,

While executing, the CPU "listens" for interrupt signals.

Use an interrupt vector table in memory to look up interrupt specific service routine.

The CPU resumes execution of the interrupted instruction.

Generates **interrupt #1**

Presses a key on the keyboard.

<table>
<thead>
<tr>
<th>Index</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>N-1</td>
<td></td>
</tr>
</tbody>
</table>

**Service routine for interrupt # 1**

A common technique is to use a interrupt vector table.
This makes is easy and fast to dispatch to the correct service routine (interrupt handler).

Interrupt vectors are used by both Windows and Unix.

User
A single program cannot in general keep either the CPU or the I/O devices busy at all times.

Is this something to be worried about? If so, what can be done about it?

Solution: let the OS keep several jobs (executing programs) in memory at the same time and switch between them.
1.4) Multiprogramming

The OS keeps several Jobs in memory at the same time.

- Job 1
  Ready to Run

- Job 2
  Ready to Run

- Job 3
  Ready to Run

- Job 4
  Ready to Run
1.4) Multiprogramming

The OS keeps several Jobs in memory at the same time.

One Job is chosen for execution and the status is changed to executing.

- **Job 1**
  - Status: Executing

- **Job 2**
  - Status: Ready to Run

- **Job 3**
  - Status: Ready to Run

- **Job 4**
  - Status: Ready to Run
1.4) Multiprogramming

The OS keeps several Jobs in memory at the same time.

One Job is chosen for execution and the status is changed to executing.

Eventually, the job may have to wait for some task, such as an I/O operation.

The status of the executing job is changed to **waiting**.
1.4) Multiprogramming

The OS keeps several Jobs in memory at the same time.

One Job is chosen for execution and the status is changed to executing.

Eventually, the job may have to wait for some task, such as an I/O operation.

The status of the executing job is changed to waiting.
1.4) Multiprogramming

The OS keeps several Jobs in memory at the same time.

One Job is chosen for execution and the status is changed to executing.

Eventually, the job may have to wait for some task, such as an I/O operation.

The status of the executing job is changed to waiting.

Instead of being idle waiting for the task to complete, the OS simply switches to another job.
1.4) Multiprogramming

The OS keeps several Jobs in memory at the same time.

One Job is chosen for execution and the status is changed to executing.

Eventually, the job may have to wait for some task, such as an I/O operation.

The status of the executing job is changed to waiting.

Instead of being idle waiting for the task to complete, the OS simply switches to another job.

Eventually the task job 1 is waiting for will complete and job 1 will change status from waiting to **ready to run**.
Nice, multiprogramming keeps the CPU quite busy despite individual jobs waiting for I/O etc.

This seems good if we want to maximize the CPU utilization. But ....

... each job might execute quite some time before any other job gets a spin on the CPU ...

Problem and solution?
We must ensure that the operating system maintains control over the CPU.

We cannot allow a user program to get stuck in an infinite loop or to fail to call system services and never return control to the operating system.

Solution?

To accomplish this goal, we can use a **timer**. The timer can be set to interrupt the computer after a specified period.
A variable timer is generally implemented by a fixed-rate clock and a counter.

Every time the clock ticks, the counter is decremented. When the counter reaches zero, an interrupt occurs.
1.4) Multitasking (timesharing)

A logical extension of multiprogramming

**CPU**

- **Job 1**
  - Ready to Run

- **Job 2**
  - Ready to Run

- **Job 3**
  - Ready to Run

- **Job 4**
  - Ready to Run
1.4) Multitasking (timesharing)

A logical extension of multiprogramming

One Job is chosen for execution and the status is changed to executing.
1.4) Multitasking (timesharing)  
A logical extension of multiprogramming

One Job is chosen for execution and the status is changed to executing.

A timer is used to enforce interrupts to be sent to the CPU at regular intervals.

When a timer interrupt occurs, the status of the executing job is changed to **Ready to Run**.
1.4) Multitasking (timesharing)

A logical extension of multiprogramming

One Job is chosen for execution and the status is changed to executing.

A timer is used to enforce interrupts to be sent to the CPU at regular intervals.

When a timer interrupt occurs, the status of the executing job is changed to **Ready to Run**.
1.4) Multitasking (timesharing)

A logical extension of multiprogramming

One Job is chosen for execution and the status is changed to executing.

A timer is used to enforce interrupts to be sent to the CPU at regular intervals.

When a timer interrupt occurs, the status of the executing job is changed to Ready to Run.
1.4) Multitasking (timesharing)

A logical extension of multiprogramming.

One job is chosen for execution and the status is changed to executing.

A timer is used to enforce interrupts to be sent to the CPU at regular intervals.

When a timer interrupt occurs, the status of the executing job is changed to Ready to Run.

A new job is selected to utilize the CPU.

The switching between jobs is so fast that the user can interact with each program as if they were all executing in parallel "at the same time".

An executing job can still be taken off the processor by other interrupts (for example I/O) and the status changed to waiting.
Definition

A program loaded into memory and executing (or waiting) is called a **process**. When a process executes, it typically executes for only a short time before it either finishes or needs to perform I/O (waiting).

Definition

If several **jobs** are ready to be brought into memory, and if there is not enough room for all of them, then the OS must choose among them. Making this decision is called **job scheduling**.

Definition

If several **jobs** are ready to **run** at the same time, the OS must choose among them. Making this decision is called **CPU scheduling**.
What if the memory is too small to fit all processes?

What if a process is too large to fit in memory?
**Definition**

Processes can be swapped in and out of main memory to disk storage, this is called **swapping**.

**Definitions**

To allow execution of a process that is not completely in memory, **virtual memory** can be used. This allows users to run programs that are larger than the actual physical memory.

Virtual memory abstracts main memory into a large, uniform array of storage, separating **logical memory** as viewed by the user from **physical memory**.
Key-presses on a keyboard might happen at any time. Even if a program is run multiple times with the same input data, the timing of the key presses will most likely vary.

Read- and Write requests to disk is similar to key presses. The disk controller is external to the program and the timing of a disk operation might vary even if the same program is executed several times.

*Interrupts* are external and asynchronous.
overflow, division by zero and bad data address are examples of internal errors in a program.

For the same input, the same errors will occur at the same place every time.

Exceptions are internal and synchronous.

Another name for exception is trap. A trap (or exception) is a software generated interrupt.

User programs can request service from the OS by generating a special system call trap.
1.5.1) Dual-Mode Operation

Needs to protect the OS from user programs. Also need to protects users from each other.

The approach taken by most computer systems is to provide hardware support that make it possible to differentiate among various modes of execution. At the very least we need two separate modes.

User Process

user process executing \(\rightarrow\) performs a \textit{system call} \(\rightarrow\) return from system call

Kernel (OS)

execute the requested \textit{system call}

User Mode

mode bit = 1

Kernel Mode

mode bit = 0

When a user application request a service from the operating system (via a system call), it must transition from user to kernel mode to fulfill the request.
1.5.1) Dual-Mode Operation

Needs to protect the OS from user programs. Also need to protect users from each other.

The hardware only allows privileged instructions to be executed in kernel mode. If an attempt is made to execute a privileged instruction in user mode, this is illegal and the hardware traps into the OS to deal with this.

How does the mode bit help us with this?

The machine instructions that may cause harm are treated as *privileged instructions*.

The hardware only allows privileged instructions to be executed in kernel mode. If an attempt is made to execute a privileged instruction in user mode, this is illegal and the hardware traps into the OS to deal with this.

As for now, here is a small example set of privileged instructions:

- The instruction to switch to kernel mode.
- I/O control instructions.
- Timer management instructions.
- Interrupt management instructions.
1.6) Program vs. Process

A *program* is a passive entity - a collection of binary machine instructions on disk (or cd, usb-stick, etc...).

A *process* is:

- an active entity.
- the unit of work in a system.
- a program in execution.

A process need resources:

- CPU time
- State (to save the content of all registers on a context switch)
- memory.
- files and I/O devices.

A *single threaded process* uses one program counter specifying the next instruction to be executed.

A *multi threaded process* uses multiple program counters, each pointing to the next instruction to execute for a given thread.
A **process** is the unit of work in a system. Such a system consists of a collection of processes:

- some of which are operating-system processes.
- and, the rest of which are user processes.

All these processes can potentially execute concurrently - by multiplexing on a single CPU for example.

The operating system is responsible for:

- **Scheduling** processes and threads on the CPU(s).
- **Creating** and **deleting** both user and system processes.
- **Suspending** and **resuming** processes.
- Providing mechanisms for **process synchronization**.
- Providing mechanisms for **process communication**.
1.6) Process management - preview of chapter 3 - 6

PCB = Process Control Block
The main memory is generally the only large storage device that the CPU is able to address and access directly.

For the CPU to process data from disk, data must first be transferred to main memory by CPU-generated I/O calls.

In the same way, instructions must be in memory for the CPU to execute them.

The operating system is responsible for the following activities in connection with memory management:

★ Keeping track of which parts of memory are currently being used and by whom.
★ Deciding which processes (or parts thereof) and data to move into and out of memory.
★ Allocating and deallocating memory space as needed.