In the first half of the lecture we covered:

Examples of some operating systems:

- Windows – Vista, XP
- Linux
- Solaris
- OS X
- Symbian (open source for mobile OS)
- Windows Mobile
- Palm OS
- FreeDos
- MS DOS

Why use an operating system?
- Manage resources - (such as memory, input / output from various I/O devices, processes, files, etc).
- Platform for running applications – The OS “manages” and executes application programs. “Manage” could mean how much time each program gets to use the computer’s resources such as processor time and memory.
- Evolving – moving target: The definition of an operating system is evolving as new functionalities and features get added.

One way to describe the operating system is to think of it as an interface between hardware and an application program. From the application program’s point of view, it does not worry about the hardware aspects as these are taken care of by this “interface” program i.e. the OS.

Materials to be covered for the course semester:
- Hardware models
- Programs – what they mean to the O.S.
- Program Interface (OS API)
- Processes
- Threads and Synchronization
- Memory Management
- Virtual Memory
- I/O Device Interfaces
- Virtual Machine (Hardware Virtualization)
- Disks and Storage
Multiprogramming System:
In the first half of the lecture we went through various phases, moving up from a example (OS excluded) designed to print the string “Hello” on screen, towards using subroutines (such as RDY, VAL, etc) inside an OS to print the same. Our elementary OS had simple primitive subroutines that made it easier for the single user to write programs effectively.

Now we modify our example to include the multi-user case which has the following features:
- 4 Terminals: A terminal will be thought of as a video terminal display and a keyboard (for simplicity, assume it to be just like the terminal in first half of the lecture).
- 1 CPU(Central Processing Unit) – comprising of the OS, Main Memory(RAM)
  - This CPU has 4 Ports and visualize each of the terminals plugged into each of these ports. Further assume, that the OS (to which we will add more features as the lecture progresses) is able to recognize the ports effectively. This means that if user 1 is typing program P1 at port 1, the OS recognizes this program as belonging to port 1. This assumption allows not to get into a discussion on drivers for each of the I/O devices connected to the ports.

The following diagram shows the above architecture where we are given just 1 CPU Unit of the computer with 4 different terminals. This architecture can now process more than one program from 4 different terminals and serve across 4 different ports. The constraint, however, is that you have one CPU that will now have to be allocated across these users.
**Program Mapping:**

Before, we get into a discussion of how these programs get managed, it is important to get an understanding of some of the preliminaries that the OS would have to take care of before it loads the program.

Visualize a computer program as made of instructions and data. Further, assume that we are able to separate the program into two sections—a instruction portion and a data portion.

What are some of the other aspects that this program might need?

1. This program might make subroutine calls, may declare temporary variables whose “life”: is that of the time of execution of the program. The data structure that implements this idea is a STACK- a set of contiguous memory locations whose length increases/decreases as the program executes.

2. At run-time, memory may need to be allocated(dynamically allocated), and they would need to be stored somewhere in memory.

The OS is thus faced the challenge of allocating these to a program and it should do it in a structured way so that the same rules can be applied for any program.
For our memory addressing scheme, it has been found that the allocation shown below has many advantages:

Here, we have the hardware programs, the OS, the STACK (which is referenced from top of the closest program above and grows downwards), MALLOC (referenced from bottom of our program and grows upwards), and the program section. The OS decides on these when a program is loaded into memory. Thus, the OS is now able to "structure" this program in the memory and can now manage it more effectively. Program mapping helps!

For instance, if a user wrote a C program that looks like the one below. After being "assembled" it would look like the one on the right(with the CALL functions). The machine code instruction would be loaded to the memory with the mapping as shown below:
Here, the “foo();” inside “main()” function indicates that a call to the function “foo” is made, and if “foo” is 200, then “CALL 200” actually occurs.

Multiple programs from multiple users:
If we have worked out a good strategy for mapping a single program, for multiple programs, it should be simple to do just the same.

The main memory would look like this:
("X" means we are denoting the addresses in hexadecimal. Each address is 16 bit (hence 4 digits). Not shown in the diagram is the growth of MALLOC growing from the top of every program.

More explanation:
Here we have separate spaces in Main Memory allocated for programs, stacks and malloc sections for every user. The hardware programs (like GETCHAR and PUTCHAR etc) and the OS are at the top of the memory separate (note that this separation is helpful: from first part of lecture) from other programs. In order to run multiple programs, the programs for each are put in separate memory areas. Separate stacks and malloc sections are created for each program. The stack for each program lowest end of the space allocated for the program, and the malloc is stored starting from the location where the program just ends. Program mapping again has helped the OS to come up with a structured approach to managing multiple programs.)
However, there is still some complexity left with these programs: Visualize every program in the computer as a set of instructions in assembly language. These instructions are fed into the computer to execute them and produce the desired results. Assume that the CALL takes 2 bytes of memory (2 locations as each location in our elementary machine is 1 byte) and it calls addresses that are 16 bits (2 bytes). The program would like this; I have used a decimal numbering scheme to address the locations from x0000 to x000A

<table>
<thead>
<tr>
<th>Instruction in Main Memory</th>
<th>Instruction Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 CALL</td>
<td>0001 CALL 0009</td>
</tr>
<tr>
<td>02</td>
<td>0004 MOV</td>
</tr>
<tr>
<td>03</td>
<td>0007 CALL 0003</td>
</tr>
<tr>
<td>04</td>
<td></td>
</tr>
<tr>
<td>05</td>
<td></td>
</tr>
<tr>
<td>06</td>
<td></td>
</tr>
<tr>
<td>07</td>
<td></td>
</tr>
<tr>
<td>08</td>
<td></td>
</tr>
<tr>
<td>09</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

In the above case, the instruction set starts at location 01. There is a call made at 01 to the location 09, and at 07 to the location 03. It is possible that this program will get loaded onto a different memory address. For instance, these instruction sets are loaded starting at location 4010. Hence within the instruction set the location to which calls are made are no longer valid! If the program were to make the first call, it would make it to 09 and not 4009 where the call should have actually been made.

Let us look at a couple of ways to fix these issues:

There are 3 approaches to solving the above problem.

1. Separately compile for each program: This solution requires that once the program is loaded onto the main memory, compile it in a way so that the calls are made to correct locations within the program. This is very time consuming but was used in early days of computers.

This is the approach used in homework 1 - compile each program specifically for the address at which it will be loaded into memory.
2. Load time fix up: This requires the fix up of calls made to the memory locations within the instruction set (i.e., the program) during load time. Imagine the program containing additional information in the form of a "header" information that contains address locations that require fix up. Assume that the OS is able to understand the header and doing the needful to fix them up. The load time fix up can be described by the following example:

Think of a simple fix up that the OS could do to this program: Here the fix up is made by incrementing the address value for each location by 4000, since 4000 is the starting location allotted for the program. Hence, within the instruction set, the value for each location and calls made to them are incremented by 4000.

But note the OS will have to do this computation at load time which is the cost of this implementation. Many program systems such as shared libraries (i.e. programs containing tons of functions) use this methodology. The OS knows exactly where to find each function in the memory. So, of any program makes a call, the OS can resolve the call effectively. Thus, many programs can use the same subroutines.
3. Hardware: If we assumed that all programs were written in such a way that they always used “relative addressing”. The first address would be x0000 and every other address in the program were addressed relative to the beginning of this program. This is very useful because we again use the same methodology as the previous section to simply “add” the loading location address.

We could also make this faster by implementing the same logic in hardware fix up the memory references. The hardware implementation requires the use of a segment register that “adds” through hardware to the addressees in the program. This speeds up the computation but the cost is the use of an extra register.

Consider the following example wherein we have 4 registers, a Stack Pointer and a Program Counter. Segment Register is added as a new feature which effectively performs our fix up of memory locations or references for any program loaded onto the memory.

The Stack Pointer is saved inside the current stack, and it points to the location (inside main memory or hard disk) being addressed. When a program is loaded in memory, it can be stored at any location. Hence the reference pointers to calls made within this program needs to fixed up. Unlike “load time fix up”, here the segment register takes care of this fix up. It does not change the reference values of any “Call” function used within the program, but adjusts those values upon executing the program. For instance:

```
0001    CALL 0003
0002    MOV
0003    PUSH
```
Here, CALL 0003 means PUSH.

Now after, loading it on memory, it assumes different location:

```
0009    CALL 0003
0010    MOV
0011    PUSH
```

The segment register sees that CALL 0003 should now be CALL 0011, by adding 0009 while executing the particular instruction. Hence it doesn’t alter any values inside the program.

The advantage of using segment register is that it performs faster than “Load time fix up” method.

Thus, through “program mapping”, we have been able to structure our programs and have also been able to fix internal issues (line numbers) for these programs.

Disadvantage of our architecture:
The major disadvantage of the architecture described for running multiple program or processes is that it doesn’t switch effectively between processes nor has the ability to recognize inputs from another terminal while it is executing a program for any given terminal. To be precise, it cannot come out of a program suddenly and take inputs from another program.

Example of GETKEY:

```
If Port 1 RDY
    Continue Program 1
If Port 2 RDY
    Continue Program 2
If Port 3 RDY
    Continue Program 3
If Port 4 RDY
    Continue Program 4
```

Consider a situation where the OS is executing Program 1 sent from terminal 1, and suddenly a user across terminal 2 presses keys on keyboard, the request from terminal 2 does not get executed unless the Program 1 comes to its GETKEY line. In other words the user across terminal 2 doesn’t see any keys or character appearing on screen. This is because the GETKEY at this time is not called by OS and hence it cannot detect input of keys from the user. The OS cannot shift from Program 1 to program 2. Thus, an “impolite” program 1 could take charge of the entire system by doing a lot of heavy computation and keeping the other programs unable to use the system. This architecture
is biased towards a “few” who know this and we need to come up with a strategy that is “fair”.

**Context Switching:**
Let us assume that we are able to switch between programs “fairly”. Suppose we do switch from a program 1 that was executing to program 2. If program 2 gets executed, and we have to return to program 1, wouldn’t be efficient if we could resume the program execution from the same point where we stopped its execution. This brings us to the idea of context switching.

If we could some of the variables associated with the “context” of this program, then we can reload the variables to resume execution. The question is: what can those variables be:

**Process Context:**

The Process State or context refers to the variables that define the “State” of the process. Before switching, the state of the process is saved.

To handle a process context, the following should be taken care of:
- Save Stack Pointer
- Save Process State
- Save Port Number
- Whether program is running?
- Save Segment Pointer.

There is no need to save Program Counter as it gets saved automatically on Stack Pointer. Below is an example of a scheduling algorithm that allows various program to execute.

Assume that program 3 was executing and then the GETCHAR detected an input at port 2. Further assume that the preemptive schedule has to respond to a keyboard input preemptively from a user who is not using the CPU(it stops what it’s doing and responds to the program that initiated the input). Assume that process 1 and 4 are not doing anything.
When a switching occurs, the OS saves the process context of the current process (program 3) and switches to the next process (program 2). It will then use the program context saved for 3 to resume execution when process 3 resumes.

**Interrupt Handler:**

In order to be able to switch effectively between processes, an interrupt handler is used. The idea is that when something happens to the system, the OS should be able to figure out what happened and do something about it. An interrupt handler can be software or hardware interrupt.
Let us look at two examples:

Example of a hardware interrupt: Think of a button attached to the CPU through a wire and given to one of the users. The button works in the following manner: If this user presses the button, the CPU suspends whatever it is doing and responds (say by loading his program for execution) to the user who pressed the button. Thus, we have come up with one way of preventing a program taking control of this system (though this user now can take control! If we gave 4 buttons to the 4 users, then we have a better system)

Example of a software interrupt: The GETKEY can be used effectively now that a timer interrupt runs, lets say every $1/100^{th}$ of a second to accept inputs from keyboard across terminals. Whenever an interrupt occurs (e.g. a mouse or a keyboard interrupt), the OS switches between programs (how and what exactly happens is not our focus here. The key idea is that we have prevented control).

Recap: The interrupt handler is a set of programs that are part of the operating system. Being a special program, it is designed to handler various interrupts such as mouse interrupt, keyboard interrupt, etc. It detects each of these interrupts and sends signals to the operating system. The OS then decides what to do with each of the interrupts.

An interrupt handler:

\begin{itemize}
  \item Finds the cause and if required turns the process off
  \item <maybe do something>
  \item Return
\end{itemize}

Mouse interrupt – It occurs when a user moves a mouse or clicks a mouse button. Every time a mouse interrupt occurs, the OS:

\begin{itemize}
  \item updates the mouse position
  \item updates the cursor position.
\end{itemize}

Materials covered in this lecture:

\begin{itemize}
  \item Program Memory Map
  \item Program Relocation in Memory
  \item Context Switching
  \item Process State (Context).
  \item Interrupts.
\end{itemize}