In the first half of the lecture, we became familiar with some of the Operating system concepts such as follows:

- Some Operating Systems (E.g. Windows XP, Linux, Solaris, etc.)
- Starting from base-up and building various parts of a system (understanding various stages of OS development such as “Write Only”, “Adding I/O – RDY & VAL”, “OS Linked with Program”, “OS separate from the program”, “OS Call Table”, “Disk Drive Interface”, “OS + Disk R/W”)
- Understanding the advantages and disadvantages of each stage of the system development.

Topics to be covered in the course of the syllabus:

- Hardware Models
- Program Interface (OS API)
- Programs
- Processes
- Threads and Synchronization
- Memory Management
- Virtual Memory
- I/O Device Interfaces
- Virtual Machines
- Disk and Storage
- File Systems
- Buffering, Caching and Replacement
- SMP and NUMA
- Program Loading
- Security and Authentication

Running multiple programs at once:

Minimum requirements: Hardware Support (E.g. more I/O devices needed)

Hence, we add a few serial ports, namely, Port 1, Port2, Port3 and Port 4:

Now, the system contains 4 serial ports which can send/receive using RDY, IN, OUT through input device such as a Keyboard. The network of computers looks as follows:
The Single-User system memory model can be described as follows:

**SETUP 1:**

In Setup 1, we assume that there is a separate memory space reserved for Hardware, Operating System, and Program Section. The program section consists of Program code, Strings, malloc. The Stack decreases from Top to Bottom while malloc increases from bottom to top as shown in the figure above. You can think of this as a subroutine foo() being called from main program.

**SETUP 2: Multi-User System (multiple processes)**
Here, we have four different processes with separate stacks for each as shown in the figure. We find that whenever each program segment needs to use the memory space reserved for OS or hardware, it enters the space and locks it.

Eg. GETKEY works as follows:

```
GETKEY:
if PORT1.RDY
    continue P1
if PORT2.RDY
    continue P2
if PORT3.RDY
    continue P3
if PORT4.RDY
    continue P4
```

Locating Programs:

1. Separately Compile for each program segment
2. Load-time fix-up
3. Hardware → Relative Addressing

In Relative Addressing, as seen above, the subroutine that is used in various programs’ main function is placed anywhere in memory and a reference is used for effective address computation.

E.g. JMP <here+100>

Use of Segment / Base Registers

Here, we can see four registers, namely R1, R2, R3, R4, Stack Pointer SP, Program Counter PC, and Segment Register. The Segment register points to the base of the current segment being addressed. For E.g., if the base points to zero, all the registers get added to the value in the Segment register.
Process- Context:

- Saved Stack-Pointer
- Load Offset
- Port Number
- Running?

Thus, for the Setup (II), we need several Stack Pointers to run programs at the same time.

Q: Do we need a Program counter (PC) for such a setup?

A: No, there is no need of PC, since whenever a subroutine is entered, a separate stack pointer for that particular process calling the subroutine, is stored and the returned accordingly. Hence, as seen in the figure above, when we are executing P3 and a GETKEY is invoked for P1, the stack S3 is stored which is then returned to, after P1 completes its execution.

PROBLEM with this setup: This is like an early MACHINTOSH PC, provided we add some interrupt handler for an input device such as a mouse. So, there is simultaneous i/o read/write with process execution. We cannot switch between processes unless the current process calls GETKEY().

SOLUTION: Adding an INTERRUPT HANDLER.

Purpose of the Interrupt Handler:

- Find the cause of the interrupt
- Turn it off
- <maybe do something>
- Return

Thus, we can now run Multi-programs. Hence, the topics covered in this pre-emptive multi-tasking system contain:

- Program Memory Map
- Program relocation in memory
- Context Switching
- Process Switching
- Interrupts