The lecture started with a discussion of Reading Assignment 1. The following table summarizes the differences between MS-DOS and UNIX.

<table>
<thead>
<tr>
<th>UNIX</th>
<th>MS-DOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programmed in C</td>
<td>Programmed in assembly</td>
</tr>
<tr>
<td>Everything is a file</td>
<td>Deals with devices</td>
</tr>
<tr>
<td>Multi-user system</td>
<td>Single user system</td>
</tr>
<tr>
<td>Multitasking</td>
<td>Single process at a time</td>
</tr>
<tr>
<td>Higher level of security provided</td>
<td>Little/No security</td>
</tr>
<tr>
<td>Kernel/shell based</td>
<td>COMMAND.COM, MSDOS.SYS,IO.SYS</td>
</tr>
</tbody>
</table>

Other notable differences discussed were the differences in file systems, file name size and the differing directory structures of the two Operating Systems.

The agenda of the lecture was established to continue with Multitasking and security and begin with processes and threads.

Assembly program

<table>
<thead>
<tr>
<th>0, 1, 2, 3</th>
<th>Startup code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program 1</td>
<td></td>
</tr>
<tr>
<td>SP1</td>
<td>Context for program 1</td>
</tr>
<tr>
<td>Program 1</td>
<td>Stack</td>
</tr>
<tr>
<td>Program 2</td>
<td></td>
</tr>
<tr>
<td>SP2</td>
<td>Context for program 2</td>
</tr>
<tr>
<td></td>
<td>Stack for program 2</td>
</tr>
</tbody>
</table>
The above assembly program has two functions at the bottom as follows:

yield 1:

MOV sp1, sp
MOV bx , sp2
MOV sp, [bx]
RET

This saves SP into the address 'sp1', and loads it from the address 'sp2'.

yield 2:

MOV sp2, sp
MOV bx ,sp1
MOV sp, [bx]
RET

Vice versa

Now, the above functions simply transfer the context of stack pointer for program 1(sp1) to the Stack Pointer(SP). Then the contents of register bx are loaded into the stack pointer for program 2(sp2) and it saves the current stack pointer to register bx so as to remember where that program left off. The RET statement returns the program to the previous state it was in before yield1 was invoked. Yield 2 works in a similar manner for program 2. This carries on till one of the programs executes the HLT(halt) command. The purpose of this program was to demonstrate a simple task switching exercise.

Flaws in this system:
If even 1 of the programs executes the halt instruction the entire system comes to a halt which is least desirable.

Protection – in earlier versions of UNIX a buggy/malicious program did not affect the Operating System(OS) or another users program.

Security is an important aspect of an OS even on the most primitive OS's because it makes the system much more reliable.

To protect the OS data and other programs we need security. The simplest way to do that is to introduce two registers known as the base and bound into our simple OS.

When a program is running we check its address against the bound register pointing to the top and the base register which points to the bottom. This principle is illustrated in figure 1 below.

But this poses another problem we can change the Base and Bound registers. Also this will allow the user to access any location in memory including the OS area. This is a serious flaw and will lead to a total disruption in service. To protect them from such an eventuality we need the concept of user space and supervisor space.
Now, as per our earlier discussion we need to introduce a concept of User and Supervisor space. This is achieved by introducing a single bit register known as $S_i$. This bit will allow us to set appropriate permissions to allow/disallow user to access OS specific areas. This can be illustrated as per figure 2 below.

Figure 1.

Now, as per our earlier discussion we need to introduce a concept of User and Supervisor space. This is achieved by introducing a single bit register known as $S_i$. This bit will allow us to set appropriate permissions to allow/disallow user to access OS specific areas. This can be illustrated as per figure 2 below.
We don't want the programs to be able to modify the base/bound. But this is not practical as you need the OS to have access to the Base/bound register. The only way we can program the machine to hop into the supervisor mode is by executing an interrupt (INT n).

Operation of yield in OS:

What if there is an arbitrary interrupt to put the machine in the supervisor mode such as:

```
INT xxxx
```

This would be a problem as it would allow us to jump to any location in the OS space resulting in a serious lapse in security.
Consider the above system attached to three terminals as shown.

While I,line = get input() (tells us where we got i/p from)
    if running[i]
        set stack, base, bound[i]
        set return value = <line>
        switch to I
    else
        cmd, args = split(line)
        load cmd into process[i]
        running[i] = T
        switch

We need to handle the process (creation, waiting, execution, etc.)

The following is a list of system calls:
getkey
getline
putchar
putline
readblock 1 Disk
writeblock 1
...

We need to create a process so as to separate the command line from the OS.
    Pid = spawn(cmd,arguments)
    wart(pid)
Spawn would create a new process and load its command and arguments.

Capturing the output is taken care of by print routines but these do not print to files (they print on the screen), so you can create another system call to write o/p to file or disk, but this does not provide a robust solution.

We introduce 2 state a) runnable representing processes either running or waiting ready to run and b) I/O wait which indicates that the process is waiting for some I/O.
We can re-write the scheduler as follows:

```
while runnable !empty
    take 1st on list
    switch to it
If I/O available
    read it in
    move waiting process from I/O wait → runnable
```

```
read I/O:
    if !available
        move self to I/O wait
        schedule
    ...
```

A question was raised: How to take care of a running process when I/O does appear on a waiting process?

Solution: In the current scenario we have to essentially till the current process finishes executing before we can move the waiting process out of the I/O wait list.