CMP325
Operating Systems
Lecture 03

Program vs Process

- **Muhammad Arif Butt, PhD**

Note:
Some slides and/or pictures are adapted from course text book and Lecture slides of
- Dr Syed Mansoor Sarwar
- Dr Kubiatowicz
- Dr P. Bhat
- Dr Hank Levy
- Dr Indranil Gupta

For practical implementation of operating system concepts discussed in these slides, students are advised to watch and practice following video lectures:
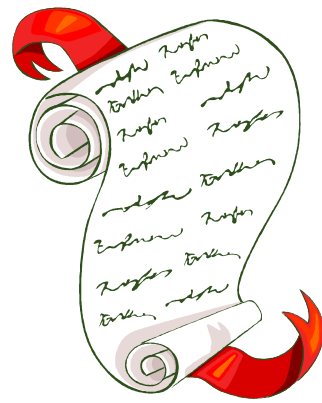**OS with Linux:**
https://www.youtube.com/playlist?list=PL7B2bn3G_wfBuJ_WtHADcXC44piWLRzr8
System Programming:
https://www.youtube.com/playlist?list=PL7B2bn3G_wfC-mRpG7cxJMnGWdPAQTViW
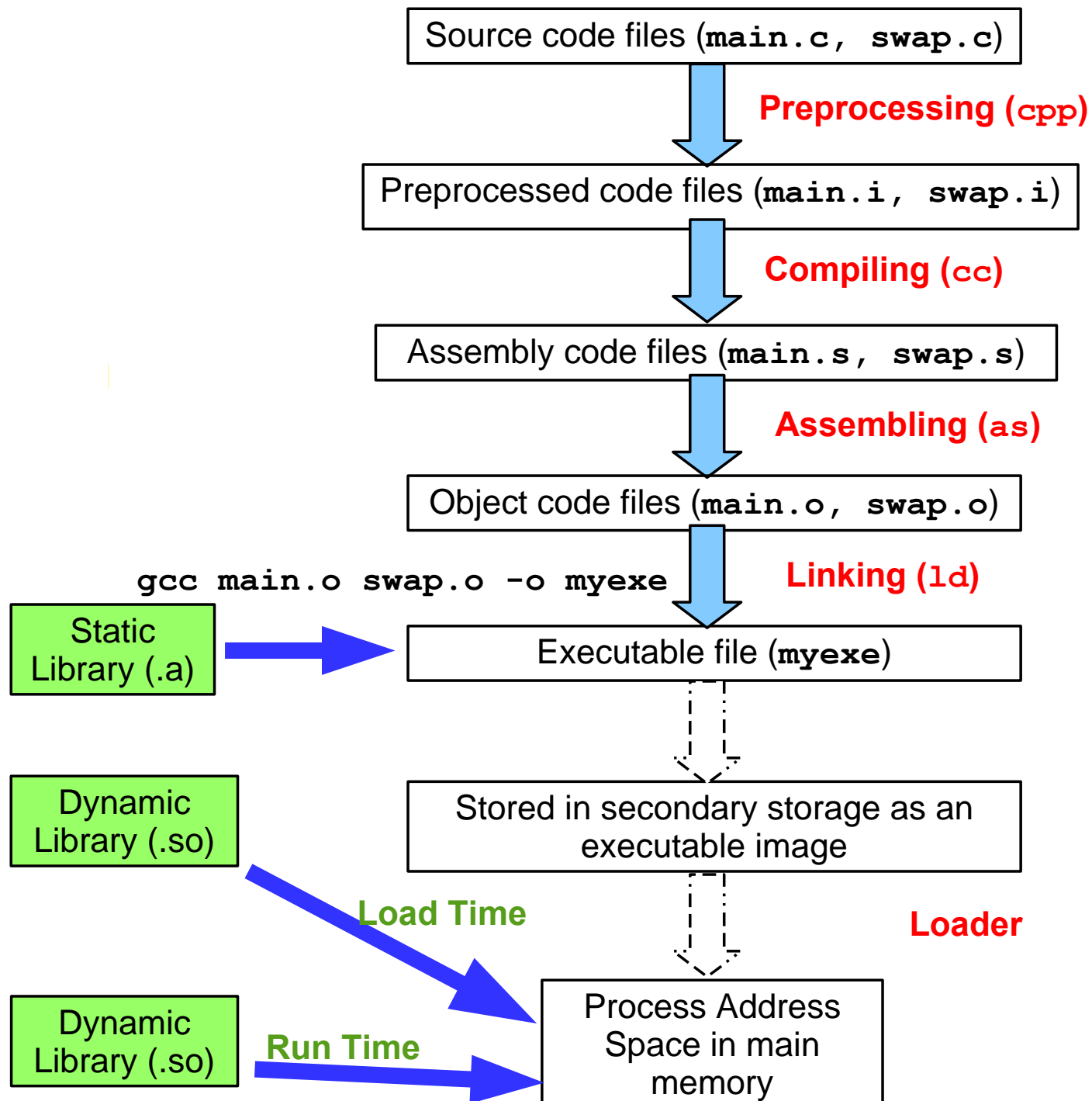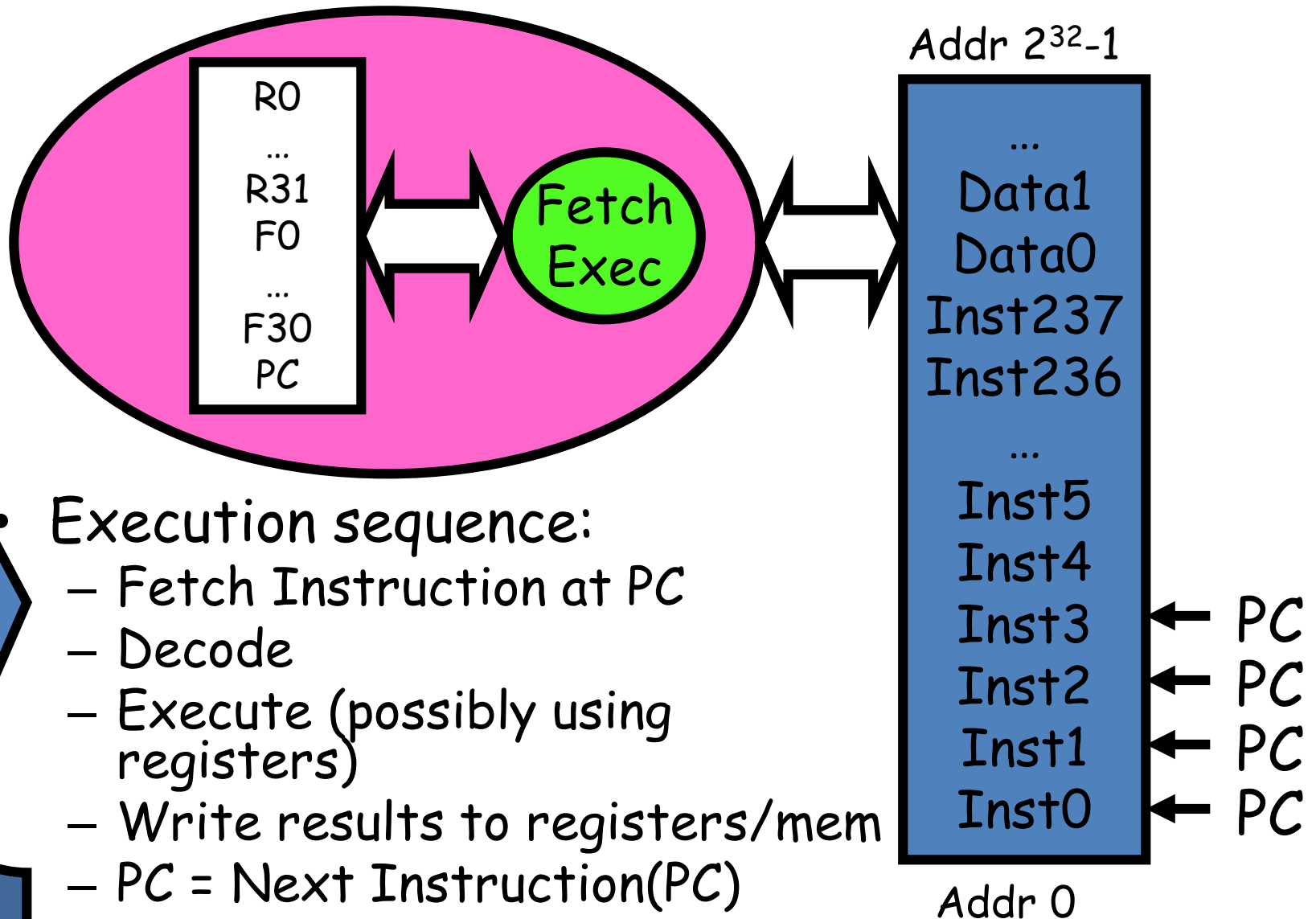
Today's Agenda

- Review of previous lecture

- Users, programs and processes

- Program on disk and its components

- Viewing contents of program file

- Process in memory and its components

- Stack, Heap and PCB

- Command line arguments and environment var

- Viewing contents of a running process

Users, Programs and Processes

- Users have accounts on the system

- Users write programs, then execute programs
  - Different users may execute same program
  - One user may execute many instances of the same program

- A program by itself is not a Process:
  - Program is a passive entity like the contents of a file stored on a disk
  - Process is an active entity, in which the Program Counter specifies the next instruction to be executed

- **A process is**
  - A program in execution
  - An instance of a program running on a computer
  - An entity that can be assigned to and executed on a processor

- The UNIX system creates a process every time you run an **external command**, and the process is removed from the system when the command finishes its execution

Source code files (`main.c, swap.c`)

Preprocessing (`cpp`)

Preprocessed code files (`main.i, swap.i`)

Compiling (`cc`)

Assembly code files (`main.s, swap.s`)

Assembling (`as`)

Object code files (`main.o, swap.o`)

`gcc main.o swap.o -o myexe`

Linking (`ld`)

Static Library (.a)

Executable file (`myexe`)

Dynamic Library (.so)

Stored in secondary storage as an executable image

Load Time

Loader

Dynamic Library (.so)

Run Time

Process Address Space in main memory

4

What happens during Program execution?

Addr $2^{32}-1$

R0
…
R31
F0
…
F30
PC

Fetch Exec

…
Data1
Data0
Inst237
Inst236
…
Inst5
Inst4
Inst3   ← PC
Inst2   ← PC
Inst1   ← PC
Inst0   ← PC

Addr 0

- Execution sequence:
  – Fetch Instruction at PC
  – Decode
  – Execute (possibly using registers)
  – Write results to registers/mem
  – PC = Next Instruction(PC)
  – Repeat

Mapping of a C Program into Process Address Space


```c
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main(int argc, char *argv[])
{
    int *values;
    int i;

    values = (int *)malloc(sizeof(int)*5);

    for(i = 0; i < 5; i++)
        values[i] = i;

    return 0;
}
```
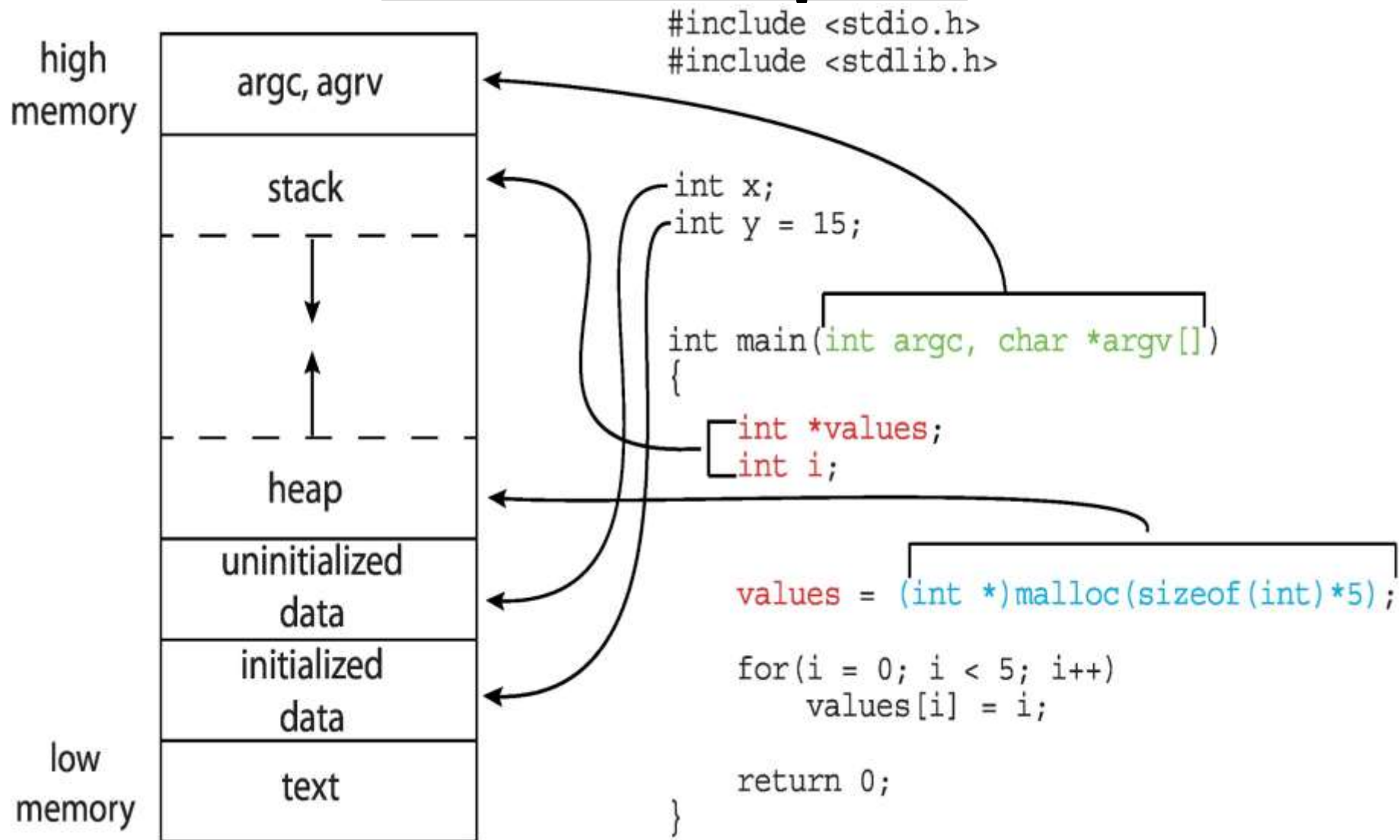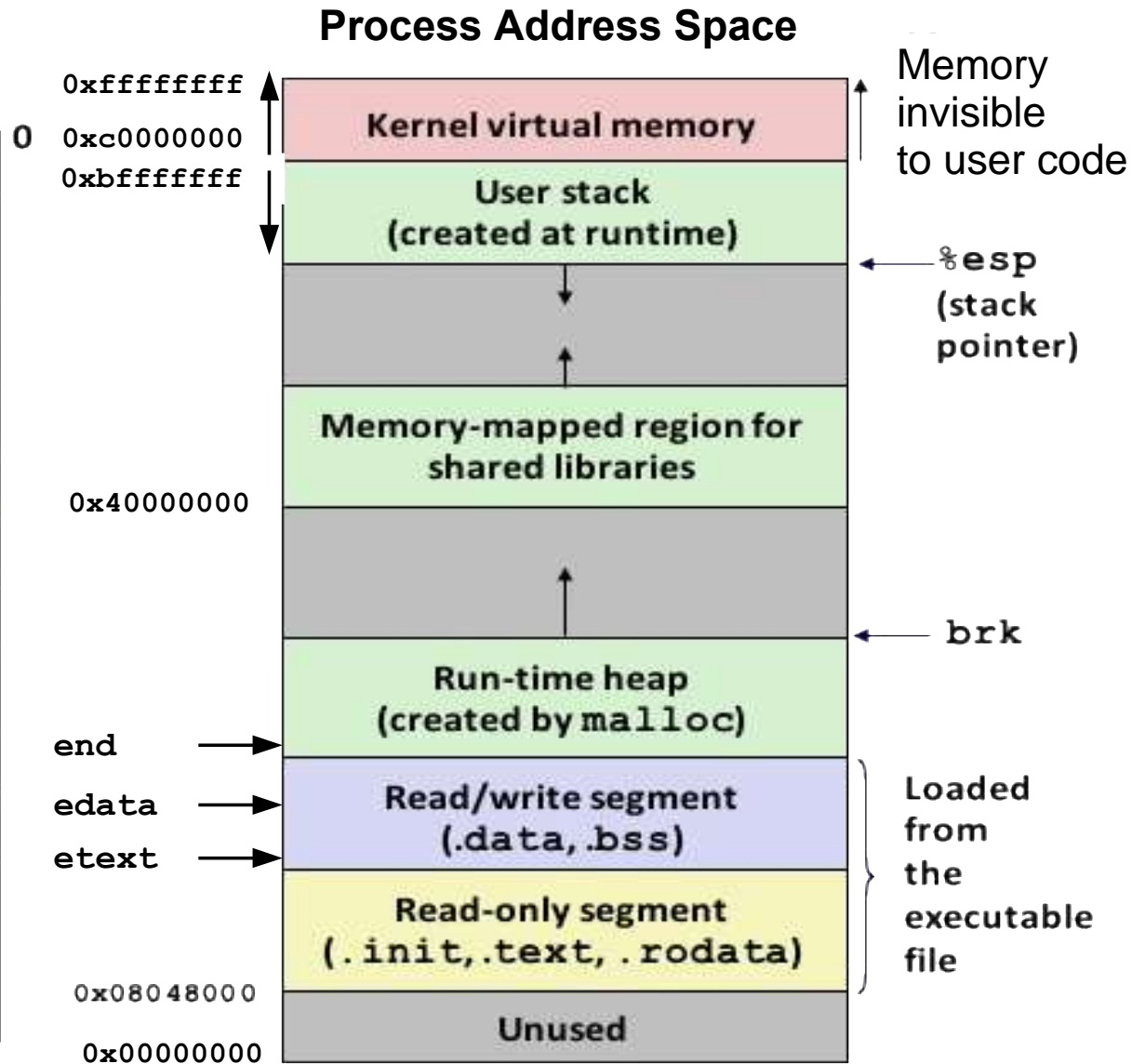
A Program on Disk

- A program is a file containing a range of information that describes how to construct a process, when a program file is loaded into memory

- Object file formats vary from system to system, like a.out, COFF, ELF and PE. Today almost all of the UNICES use Executable and Linking Format

- ELF not only simplifies the task of making shared libraries, but also enhances dynamic loading of modules at run time. An executable file using the ELF format consist of ELF Header, Program Header Table and Section Header Table

- The files that are represented in this format are:
  - Relocatable file objects (.o)
  - Normal executable files (a.out)
  - Shared object files (.so)
  - Core files

Program File vs Process

Process Address Space

Executable Object File

Executable Object File
ELF header
Program header table (required for executables)
.init section
.text section
.rodata section
.data section
.bss section
.symtab
.debug
.line
.strtab
Section header table (required for relocatables)

0

Memory invisible to user code

```
0xffffffff ┌──────────────────────┐  ↑
           │  Kernel virtual memory│
0xc0000000 ├──────────────────────┤  ↕
0xbfffffff │     User stack       │
           │  (created at runtime)│        ← %esp
           │           ↓          │         (stack pointer)
           │           ↑          │
           │  Memory-mapped region for
           │    shared libraries  │
0x40000000 ├──────────────────────┤
           │           ↑          │        ← brk
           │    Run-time heap     │
           │  (created by malloc) │
end   →    ├──────────────────────┤
edata →    │  Read/write segment  │  Loaded
           │   (.data, .bss)      │  from
etext →    ├──────────────────────┤  the
           │  Read-only segment   │  executable
           │ (.init, .text, .rodata) │ file
0x08048000 ├──────────────────────┤
           │       Unused         │
0x00000000 └──────────────────────┘
```

Program Loading is a process of copying a program from disk to main memory in order to make it a process

One program may be used to construct many processes or many processes may be running the same program

8

Components of a Program

1. **.init section**: contains different start up routines, which are executed before calling the main() function
2. **.text section**: contains machine code and is read-only
3. **.rodata section**: contains read only data like strinng literals and constants
4. **.data section**: contains initialized global and static variables
5. **.bss section**: contains uninitialized global variables. Named after IBM704 assembly instruction Block Storage Start. Since this section occupies no actual space in the object file, so some people call it Better Save Space
6. **.symtab section**: contains the symbol table which contains
    *   Name, type and relative address of global variables
    *   Name and relative address of functions defined
    *   Name of external functions like printf()
7. **.debug section**: contains debugging information if compiled with –g option

View Contents of Program File

You can use the size command to view the size of various sections of a program file

`$ size /bin/sh`

The most common command that is used to view and understand the contents of a program file on disk is `readelf`

`$ readelf –a prog1.o` (display all info)

`$ readelf –h prog1.o` (display ELF header)

`$ readelf –a prog1.o` (display names of diff sections)

We can also use `objdump` command to display information from object files

`$ objdump –h prog1.o` (display section headers from obj file)

`$ objdump –d prog1.o` (show disassembly of .text section)

`$ objdump –D prog1.o` (show disassembly of all sections)

`$ objdump –d –M intel prog1.o`

Components of a Process

From the program file on disk, only the .init, .text, .rodata, .data, and .bss sections are copied to the process address space. The rest of the process address space is created at run time

1. **Heap:** Used to allow a process to request for allocation/ de-allocation of memory at run time, grows towards higher addresses

2. **Stack:** Used to store the function stack frames or activation records, which are pushed on the stack when a function is called and popped from the stack when a function returns. A FSF on x86 contains function arguments, function return address, base pointer and local variables. So the top of the stack contains command line arguments and environment variables passed to the main() function. Then the stack grows down, towards lower addresses (on x86)

3. **Process Control Block** At the top, almost 1/4th of the process address space is reserved for code of the kernel and data structures related to itself and also for every process. One important data structure that kernel maintains for every process is called PCB

Process Control Block

PCB is the most important and central data structure in an Operating System maintained by kernel. It contains information associated with a specific process that is used by OS to control that process. It contains information as shown:

1. Process identification:
- PID & PPID
- UID & GID
- EUID & EGID
- Saved SUID& SGID
- File System UID & GID
- Supplementary GIDs

3. Process Control Information:
- Scheduling info
- Privileges info
- Memory management info
- Resource ownership and utilization
- IPC

2. Process State Information:
- User Visible Registers
- Control and Status Registers (flags)
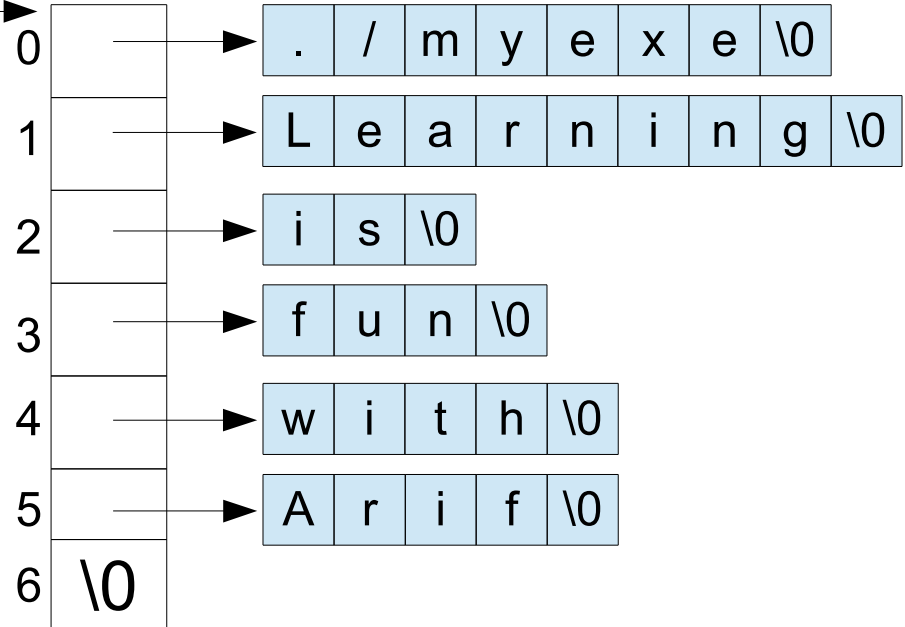
Command Line Arguments

```
int main(int argc, char *argv[]){
    printf("No of arguments passed are: %d\n",argc);
    printf("Parameters are:\n");
    for(int i = 0; argv[i] != NULL ; i++)
        printf("argv[%d]:%s \n", i, argv[i]);
    return 0;
}
```

$ ./myexe Learning is fun with Arif

int argc

6

char * argv[]

0	.	/	m	y	e	x	e	\0
1	L	e	a	r	n	i	n	g	\0
2	i	s	\0
3	f	u	n	\0
4	w	i	t	h	\0
5	A	r	i	f	\0
6	\0

Use of Command Line Arguments

`$ cat /etc/passwd`

int argc

```
2
```

char * argv[]

0	→ `c` `a` `t` `\0`
1	→ `/` `e` `t` `c` `/` `p` `a` `s` `s` `w` `d` `\0`
2	\0

`$ ls -l /bin`

int argc

```
3
```

char * argv[]

0	→ `l` `s` `\0`
1	→ `-` `l` `\0`
2	→ `/` `b` `i` `n` `\0`
3	\0

Environment Variables

```
int extern char **environ;
int main(){
    printf("\n Environment variable passed are:\n");
    for (int i = 0; environ[i] != NULL ; i++)
        printf("environ[%d]:%s\n", i, environ[i]);
    return 0;
}
```
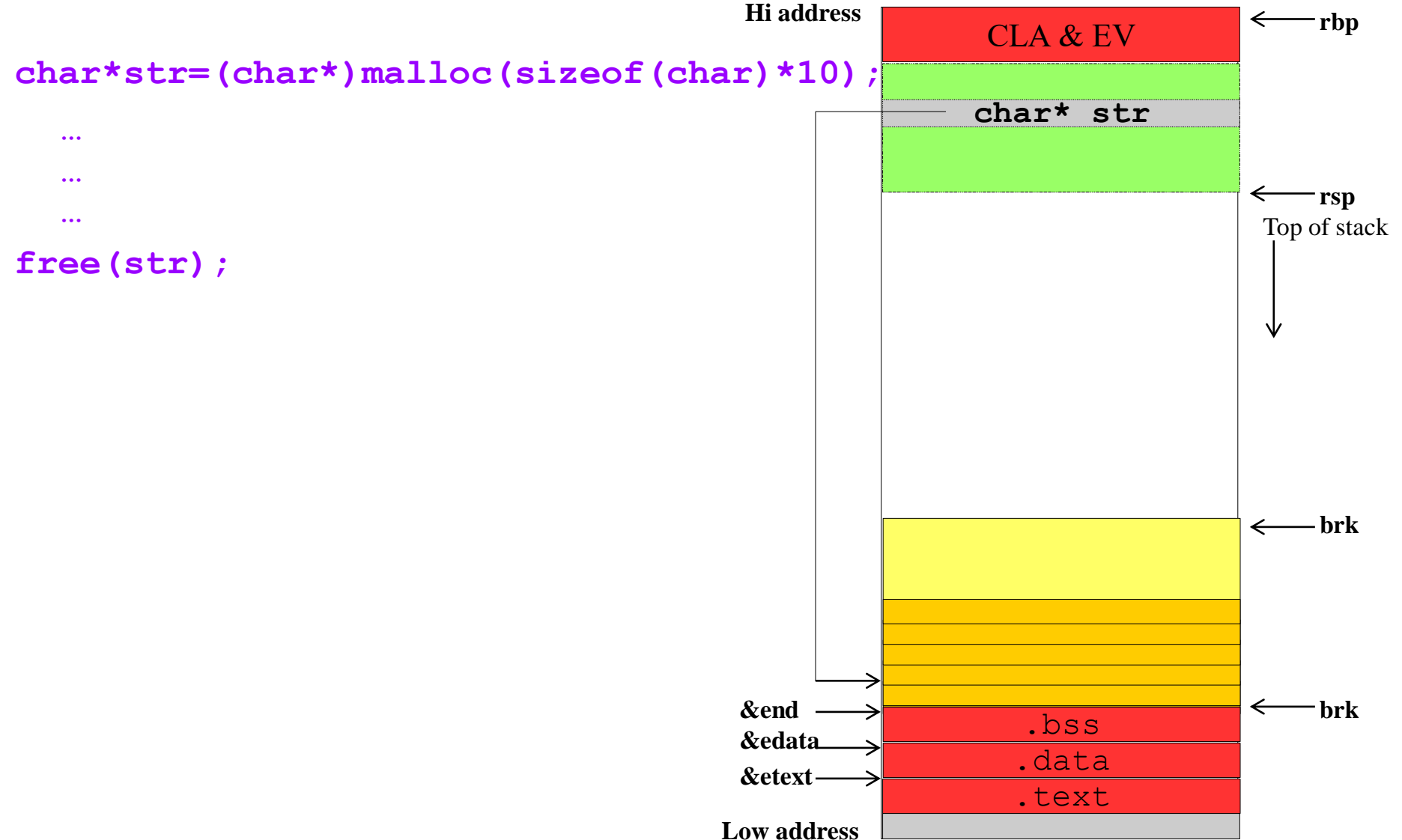
$ ./myexe

char ** environ

0	LANG=en_US.UTF-8'\0'
1	USERNAME=arif'\0'
2	PWD=/home/arif/spvl/08'\0'
3	HOME=/home/arif'\0'
4	SHELL=/bin/bash'\0'
5	PATH=/usr/local/bin:/usr/bin:/bin:'\0'
6	HISTFILE=/home/arif/.bash_history'\0'
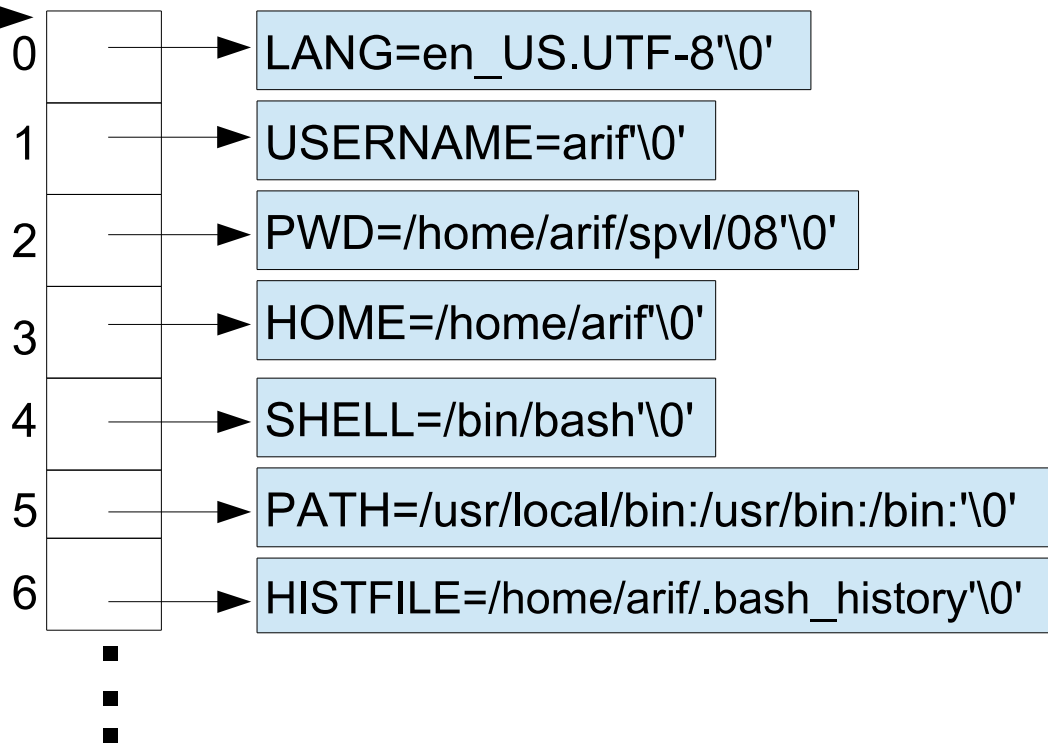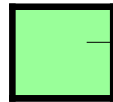
Requesting Memory on Process Heap

- In a process address space the heap starts just above the data segment. Unlike stack, heap grows up towards the higher addresses

- A programmer normally use heap when he/she does not know how much memory (variables, arrays) will be required at run time

- Some of you might have used the function `alloca()` to allocate a specific amount of memory in the caller's stack, but the limitation of this method is that it do not survive the return statement of the function in which it is created, i.e., it is freed automatically when a function returns. So its scope is limited to the FSF of the function in which it is created

- Every programming language provide functions for allocation and deallocation memory on heap. In case of C, we can use the `malloc()` family of functions to allocate and free memory
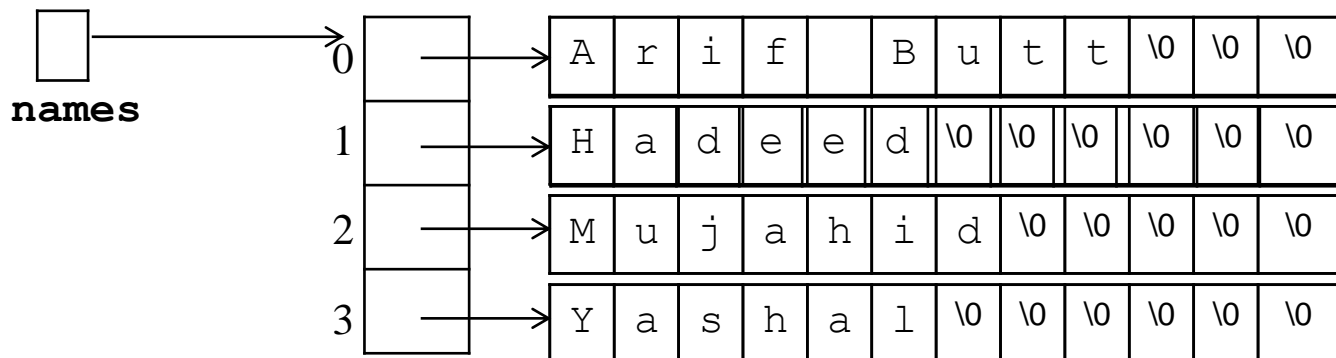
Illustration: 1D Array on Heap

```c
char*str=(char*)malloc(sizeof(char)*10);
   …
   …
   …
free(str);
```

Hi address

CLA & EV ← **rbp**

char* str

← **rsp**
Top of stack

← **brk**

&end →
&edata →
&etext →

.bss ← **brk**
.data
.text

Low address

Illustration: 2D Array on Heap

```
int i, rows = 4, cols = 12;
char ** names = (char**)malloc(sizeof(char*) * rows);
for(i = 0; i < rows; i++)
    names[i] = (char*)malloc(sizeof(char) * cols);
…
…
…
for(i = 0; i < rows; i++)
    free(names[i]);
free(names);
```

View Contents of Running Processes

In order to provide easier access to information related to the OS kernel and running processes, many modern UNIX implementation provides /proc/ virtual file system. The /proc file system is called virtual because the files and sub-directories it contains does not reside on disk. Instead the kernel creates them on the fly as processes are created and terminated. It answers the questions like:

- How many processes are running on the system and who owns them?
- Which program file the process is executing?
- Which is the present working directory of a process?
- What command line arguments are passed to a process?
- What environment variables are passed to a process?
- What all files the process has open?
- What is the status of a running process?

For further information students must visit the man page of /proc directory

Linux Commands to View Processes

- Process Status (ps) command is the most commonly used command to display the information about running processes

`$ ps [-axul]`

- Top is another very famous command that is used to monitor the CPU usage in real time. By default it refreshes its o/p every three seconds. Displays a summary area having about six lines of info and a per process information area having about 12 columns. It is an interactive command, and some of the important command keys are:

  - n: Prompts for number of processes to display
  - u: Prompts user name, whose processes to be displayed
  - s: Prompts number of seconds to refresh the display
  - r: Prompts PID and nice value to be changed
  - k: Prompts PID and signal to be sent to a process

For further information students must visit the man page of ps and top

SUMMARY

We're done for now, but Todo's for you after this lecture…
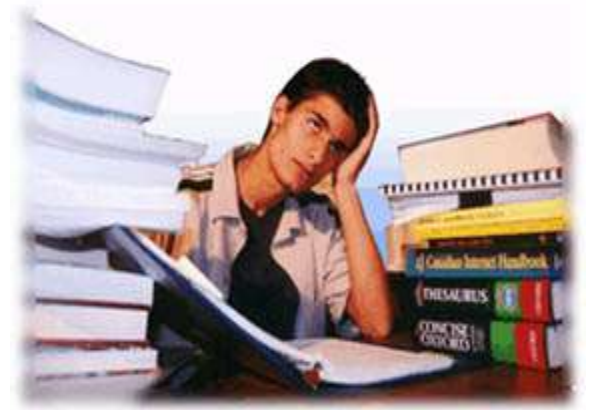
- Go through the slides and Book Sections: **3.1 to 3.3**
- Go through Unix The Text Book Chapter 13
- Type, compile, execute, and understand the programs on the slides and also the programs discussed in class
- Start making your hands dirty before you get your First Programming Assignment

If you have problems visit me in counseling hours. . . .