Systems Programm

SP

Robins & Robins. Ch:

Systems Programming is that programming in which system calls are used.

System call is function that invokes kernel.

C is stable language and resembles to hardware.

basic Unix: Systems Programming

communication, concurrency

↓

at a time two

Thread → Flow of execution.

pthread (For multiple threads).

pthread_create (func)

function name is basically jump location.

main is starting or entry point of program.

Ch: 1 Intro

Ch: 2 Program, process Threads

Ch: 3 Processes.

Ch: 4 Unix I/O

5- Files & directories

6 Special Files (Pipes / fifos / Sockets)

M T W T F S

Ch: 7 Proje-
Ch: 8 Signal (event notification) Date:

Standard I/P → keyboard
O/P device → monitor.

When folder is created.

signal (Interrupt specific signal)
Ctrl + C

Ch: 9 Times and timers.

Ch: 12 Threads / IS) IPC using shared memory.

Lecture:

// define A 10.

These are precompiler directives.

main is starting point of program.

return will be returned to the process that created it.

File system is for disk.

File → (After compile, loading) RAM.

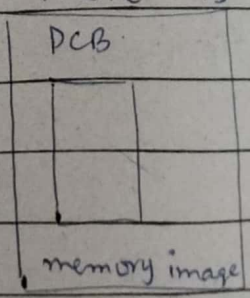
disk

LTS

dispatcher.

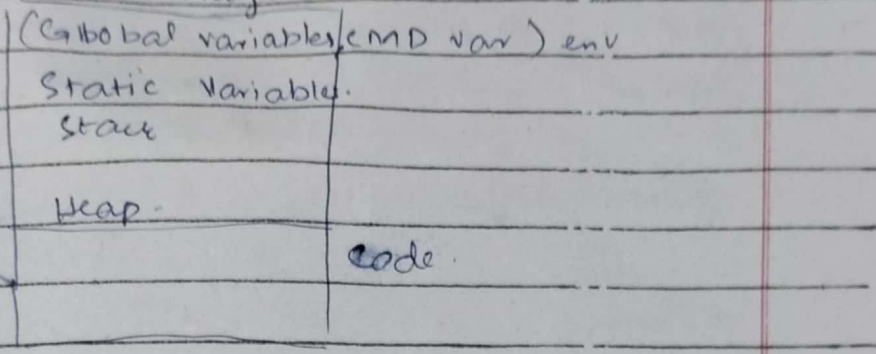
kernel

We see first There is space in RAM.



T W T F S Date:

Process Image.



command line variables get memory allocated until the process exits.

≠ define variables did not use memory.

Static variables.

Static initialize variables Uninitialize variables

Without fun, only memory will be allocated to it. Static variables change their values. When second time function will be called then the previous value will be retained.

When we return value then stack of function destroys.

M T W T F S Date:

Global and static variables didn't destroy.
Global variables get memory at start. and static variables get memory ~~on run time~~.

Memory is allocated to ~~Stat~~ initialized on compile time. and uninitialized gets memory at runtime.

intialized ~~xxx~~ variables will be given memory and that memory will be part of output file.

when we call fun by value then that variable will be pushed to stack.

Activation records: Function arguements and values will be pushed on stack.
When function returns Then this destroys.

Heap → Dynamic variables
Those we get memory at run time.

WTFS Date:

Destroying capability of dynamic variables is with user free delete() -

code section → data and text

Library Function calls:

PCB:

→ PC

→ PID, PPID

→ Statistics.

→ File descriptor table

↳ list of open files -

→ Memory limits.

By default when we start process then 3 files open by default.

default file descriptor

0	STDIN - FILENO	← keyboard
1	STDOUT - FILENO	← Monitor
2	STDERR - FILENO	← Monitor

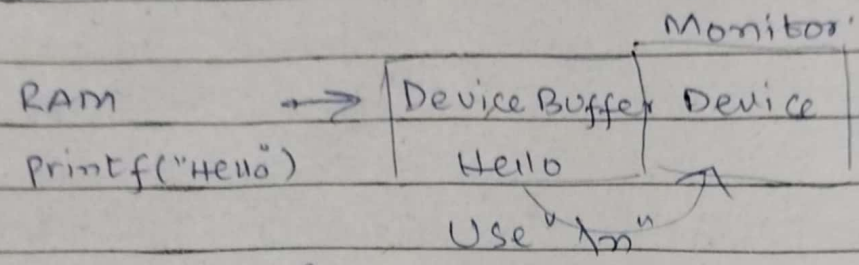
init → login → terminal → Bash → P1.0

Bash ← takes command from user and displays at output. or displays error.

File descriptor table is inherited from Parents.

M T W T F S Date:

Difference Between stdout and stderr :



① New line.

② When we didn't use "\n" then it gets accumulated in buffer until it gets flow.

③ scanf (Input from user.
↳ It will empty out stdout buff.

④ fflush (To manually empty the buffer).

Last ⑤ process terminates then buffer flushes out

When you communicate with any peripheral device then data is first moved to device buffer.

In Linux, peripheral devices are considered as files.

STDIN ... are handlers with the help of which you do function.

M T W T F S stderror. Date:

stderror.

RAM → Device — No buffer

monitor.

we use stderror to print error without any error -

```
main() {
    printf ("Hello 1")
    perror ("Hello 2")
}
```

Hello 2. Hello 1.

This is used when we want to write on specific stdout

fprintf (file, "Hello");

fprintf (stderr, "Hello"); No buffering.

fprintf (stdout, "Hello"); Buffering

wherever we use "\n" then data will be displayed in sequence.

Library Function calls:

Rules:

1) Do error checking before using library system call. Success / fail.

2) if Failed : check reason.

M T W T F S system call. Date:

int close (int fd);

It closes file-

-1 : error.

0 : success-

0	IN
1	OUT
2	ERR
3	fd

set with appropriate no

#include <errno.h> → int errno=4

int main()

{ Printf ("errorNO = %d \n", errno).

int ret = close(4); fd = open("f1");

if (ret == -1) close(fd);

{

Printf ("Errno = %d \n", errno)

Printf ("Error message = %s", strerror(errno)

return -1;

}

Printf("Stderr, error-%s\n"

perror

Printf ("Successfully closed file")

return 0;

}

errno	
0	successful
4	Invalid fd
127	

#include <string.h>

char* Strerror (int errorno);

To print list of all values of errorno.

T W T F S Date:

```
if(ret == -1)
    for(int i=0; i<127; i++){
        printf("Error NO = %d \n", errno)
        printf("Error message = %s",
                strerr(errorno))
fprintf(stderr, "error = %s",
                stderr)
```

(Check first and last errno) Task

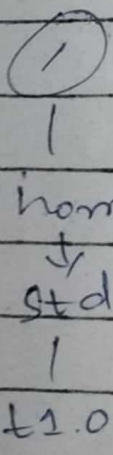
Environment variables:

Parent environment var are inherited in child.

By default envi var are same but we need to run exec call if we want to change environment.

G.V, CLA, Envi var

CLA(argc, argv)..



CWD: /home/std

M T W T F S Date:

List of all environment variables

command: $env.

HOME = /home/student.

PWD = /home/student/OS.

$echo $HOME

$ HOM = /home (Changing value of env var).

In stdio we have variable:

char **environ;

↓ array of strings like

PWD: /home/std

USR:

PATH:

PS2:

NULL.

→OR char *environ[];

```
main()
{ for(i=0; environ[i]!=NULL;i++)
  printf("%s\n", environ[i]);
}
```

List of all environ variables.

which ls → Tells path.

/bin

It searches only in environ path variables.

T W T F S Date:

execlp(command name "ls")

getenv → To get only specific name of environ variable.

input is name of envi va.

/home/student1 "HOME"

```
char *getenv (const char *var);
main() {
  printf("  HOME:%s", getenv("HOME"));
   char *val;
    val = getenv("HOME");
}
```

$HOME = /home

↓ name ↓ value.

Process Termination:

① Normal

② Abnormal.

Normal

① Exit()

② _exit() / _Exit() ← same both

③ return statement of main.

child status return to parent.

Prototypes.

```
int exit(int status);
int _exit(int status);
int _Exit(int status);
```

M T W T F S Date:

at exit → when my program will exit Then it will perform some functions. It's in case of exit() system call.

Already defined.

→ atexit(void *(func));

You can define multiple functions with atexit.

```
main () {
    atexit (fun) ;
    atexit (fun2);
    printf ("In function main in.")
    exit (0);
}
```

```
void fun()
{
    printf ("Process terminating using exit system call");
}
void fun2()
{
    printf ("2nd");
}
```

printing in reverse.

- exit didn't calls atexit functions.

```
main()
{ atexit (fun);
  atexit (fun2);
  int ret = close(u);
  if (ret == -1)
{ perror ("Failed to close");
  exit(-1);
}
  exit (0);
}
```

T W T F S Date:

exit(0) and return 0 are same.

↓ explicit ↓ implicit.