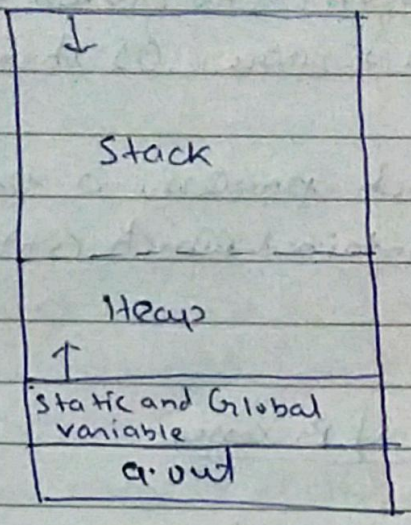
Process management

Date: __/__/__

when we want to execute a program, the OS will create a process related to the program

Stack is needed to function and recursive calls. It also contains local variables and parameters passed to the functions.
Heap is used in case of dynamic need of space.

The executable code cannot take any space beyond the process boundary.



Process id

Program counter: next instruction to be executed.

Process state

Priority :- OS process will have higher priority over user process.

General purpose registers: - when the current process is preempted during the execution then the values in the GPRS should be saved otherwise new process will replace the current GPRS values with new values. In that case, when the old process will resume its value ~~will~~ GPRS values will be lost. GPRS are commonly used by each process.

List of open devices

Protection :- no process should occupy other process's space. OS should ensure this.

For each process, a process control block is maintained which contains above variables.

States of Processes

(i) New :- The program going to be executed is called a process.

(ii) Ready :- when we create a process, it is ready to run. So, it is in ready state.
when many process are ready to run then it is called multiprogramming.
multiprogramming is two types : with preemption (multitasking) and without preemption.

(iii) Run :- In a single CPU, only one process will run at a time.

(iv) wait :- During running if the process need I/O access then the process is brought into wait state. Foom

Page No:
Date: / /

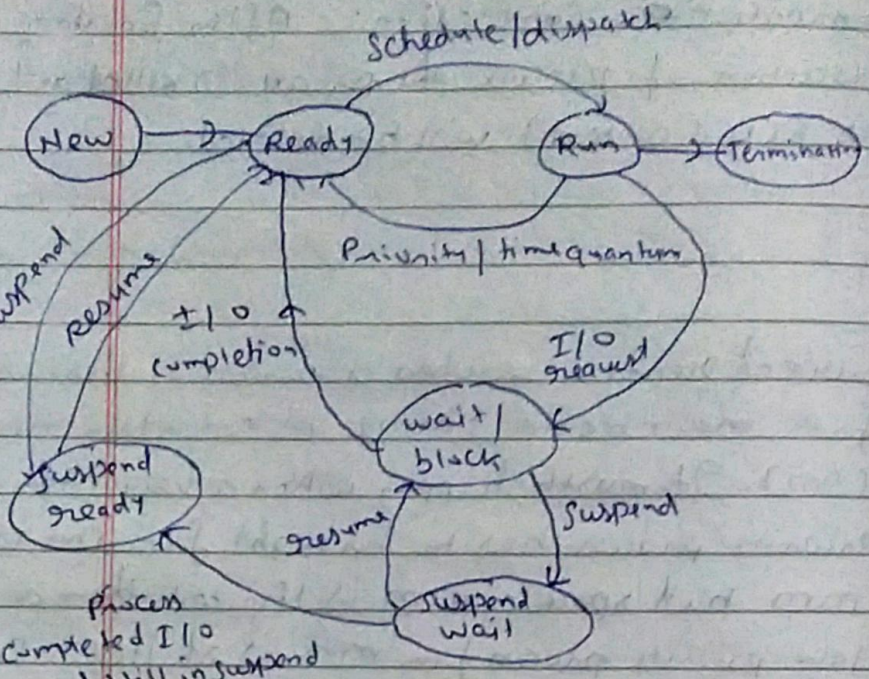
(v) termination on completion:- After finishing execution of process, the process is killed and its PCB (content) will be deleted.

(vi)

(vi) Suspend ready:- when a process is thrown from main memory (mm) to secondary memory (sm). It mostly happen when a very high priority process has to brought from sm to mm but space in mm is limited then a low priority process from mm is shifted to sm by the scheduler.

(vi) Suspend wait or suspend block:- Similar to suspend ready, in place of a process in ready state, a process in wait state is thrown to the sm. In this case, if the process finishes its wait in the sm then it can move to suspend ready so that when the process is brought back to mm, it kept in ready state and not in wait state.

Processes are created, scheduled, executed & deleted



Long term Scheduler:- It decides how many Processes should be created or made. It also checks that correct balance of CPU and I/O bound Processes in the ready queue. It may take time in decision but it should take efficient decisions.

Short term Scheduler:- It decides which Process to be shifted to run state. It should take the decision quickly. e.g. if time quantum is 10ms then it has to decide within 10ms that which Process should be executing next otherwise CPU will sit idle.

Middle term Scheduler:- It decides which Process to be suspend.

Degree of multiProgramming:- maximum no. of Process that can be present at ready state.

Long term scheduler decides degree of multiProgramming

After choosing a process to be shifted from ready to run state, the short term scheduler calls dispatcher which whose job is to bring the chosen job from ready to run state and shift the running process to ready in case of Priority scheduling. This is called content switching, i.e. changing the status of the CPU by replacing an old process by a new process. This the delay in content switching should be as minimum as possible.

swapping :- moving a process from mm to sm.

LT is related to overall Performance.
MT scheduler is related to swapping
ST - - - - content switching.

Consider a OS system with 'N' CPU & 'm' processes

Ready : - min - 0
max - m

running - min - 0
max - N

block - min - 0
max - m

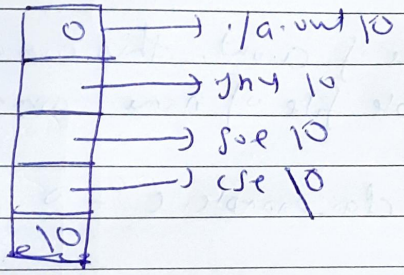
Command line arguments

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

$ ./a.out jnv sce cse

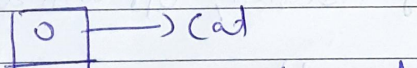
∴ argc = 6

char *argv[]



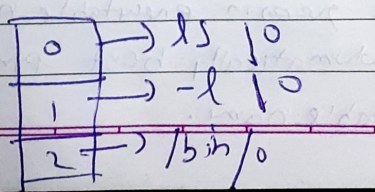
$ cat /etc/passwd

∴ argc = 2, argv



$ ls -l /bin

argc = 3, argv



Here, ./a.out or cat or ls are also part of argv bcoz we can create the executable file of a program with different names e.g.

```
int main (int argc, char *argv[])
{
    if (Strcmp (argv[0], "./command1") == 0)
        Printf ("command1 Printed");
    else
        Printf ("command2 Printed");
}
```

o/p gcc cla_example.c -o command1

Now in place of a.out, this will create an executable file of name command1

Now, gcc cla_example.c -o command2

now, we have 2 executable files, command1 and command2 for the same program. But, depending upon which file is being executed, o/p will differ as per the p/m.

$ ./command1

o/p command1 Printed

$ ./command2

o/p command2 Printed

so this is the reason executable name is also passed as argv[0] automatically bcoz p/m o/p can may depend on the executable name.

M T W T F S S
Page No.:
Date:
YOUVA

=> environment variables

$ env

$ set | grep -i histsize
HISTSIZE = 1000

$ HISTSIZE = 250

$ set | grep -i histsize
HISTSIZE = 250

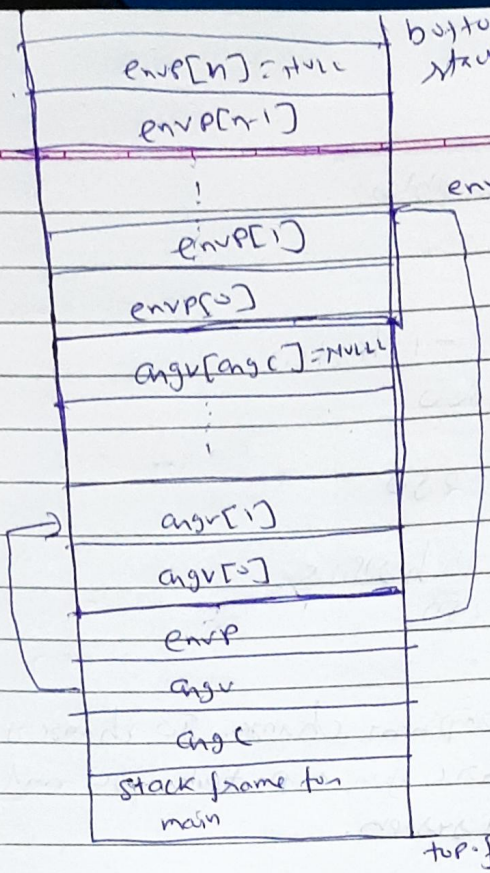
This change is temporary change. To change it permanently, change in .bashrc for a particular user and /etc/profile for all user of the system.

$ man environ

extern char **environ
int main()
{
printf("env variables are\n");
for(int i=0; environ[i] != NULL; i++)
printf("environ[%d]: %s\n", i, environ[i]);
}

* gt will print all environment variables as env command printed.

putenv & setenv functions can be used to change the value of env. variable
* getenv can be used to access a particular env. variable.



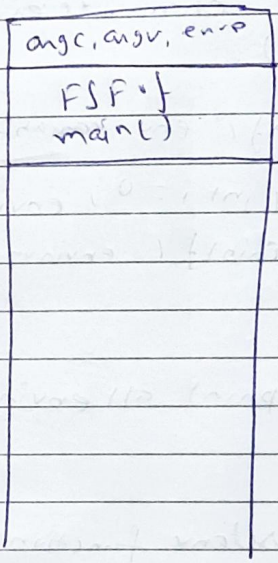
Layout of process stack

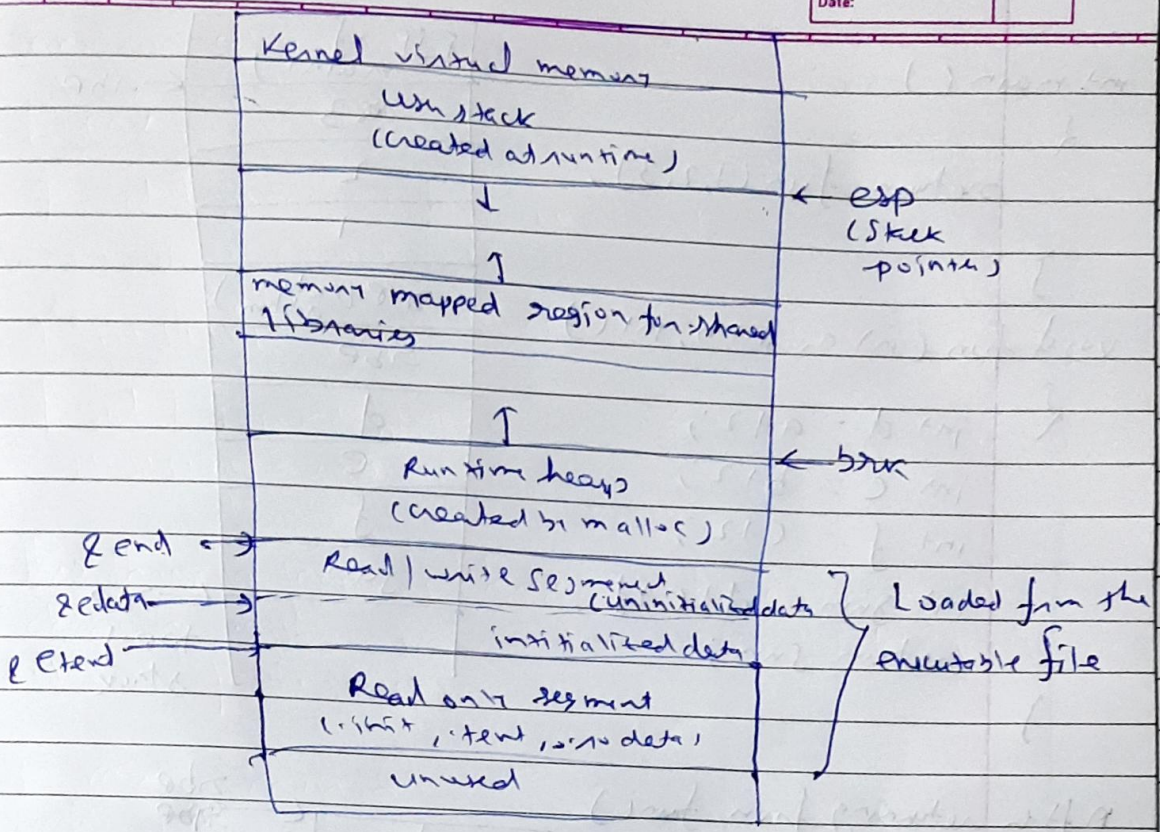
⇒ Function stack frame of a function contains

(a) local variables of the function
(b) argument passed to the function
(c) return address
(d) base pointer

%rsp & %rbp are base pointers.
%rbp points to beginning of FSF of the function.

%rsp points to top of the stack.

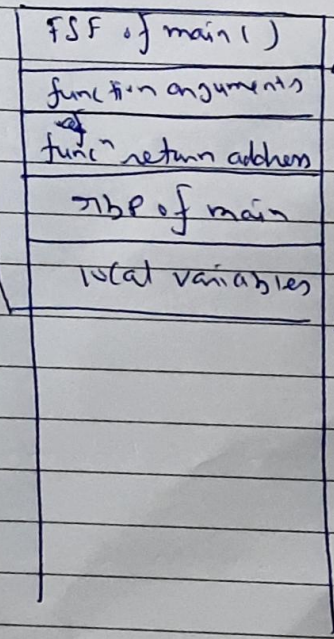




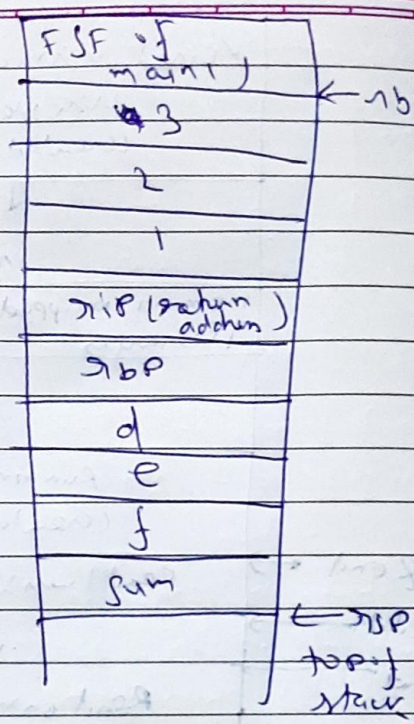
Let main() function is calling fun() function

-) arguments are pushed in stack in reverse order

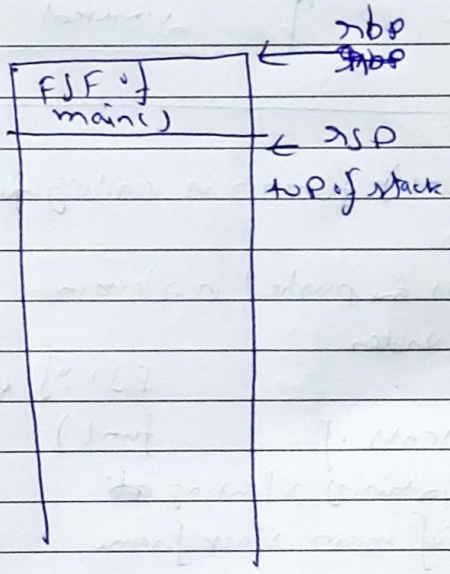
-) the contents of rbp containing starting addres of main stack frame is saved on stack for later use and rbp is moved to where rsp is pointing to create new stack frame pointer of function fun()



```
int main()
{
    return fun(1,2,3);
}
void fun(int a, int b, int c)
{
    int d = a+2;
    int e = b+2;
    int f = c+2;
    int sum = d+e+f;
    return sum;
}
```

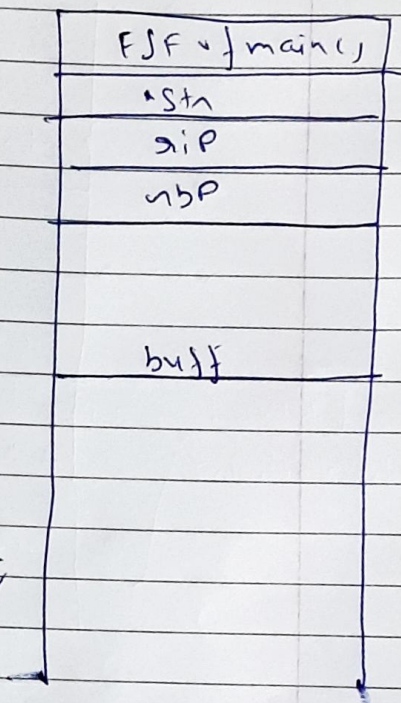


After returning from fun()



Buffer overflow - It is a bug in a program which occurs when more data is written to a block of memory than it can handle.

```
int main (int argc, char *argv[])
{
    display (argv[1]);
}
void display (char *str)
{
    char buff[10];
    strcpy (buff, str);
    printf ("data is : %s\n", buff);
}
```



Processes

$ ps -f

kernel keeps track of pages occupied by different process

In the top of process (address space), 'kernel virtual memory' is there.

The kernel address space is divided into 2 parts, 1st contain kernel code, its data structures which are same for all processes whereas part 2 contains process specific data structures which are different for every process.
e.g. state of a process, resources occupied by the process, etc.

$ uname -a

$ cd /usr/src/linux-headers-5.4.0-90-generic/include/linux

$ less fs.h

:/struct file \r
to check the structure of a file.

$ less sched.h (to see structure related to process management, which is PCB)

:/struct task_struct \r

explore the different variables declared in this structure.

Q) $ cd ... check the files related to a process in /proc directory.

=> getpid() and getppid() can be used to get the pid of a process and its parent process.

=> In a shell, the PID of the shell can be get through the environment variable $$ and its parent by $PPID

=> Swapper or sched process scheduler is a system process having PID as 0. No program file for swapper in /proc directory

=> Init or Systemd has a PID 1

=> Page daemon or Kthreadd has PID 2.
It supports the paging of virtual memory system.

=) fork() system call allows one process the parent to create a new process, the child

=). It is a system call which is called once but return twice; once in the parent and once in the child. To the parent it returns PID of child & to the child it returns 0. After the call returns, both parent & child process continues their execution concurrently from the next line of code.

=) The child process is a clone of the parent & obtains copies of the ~~parent~~ ~~parent's~~ parent's stack, data, heap, text segments.

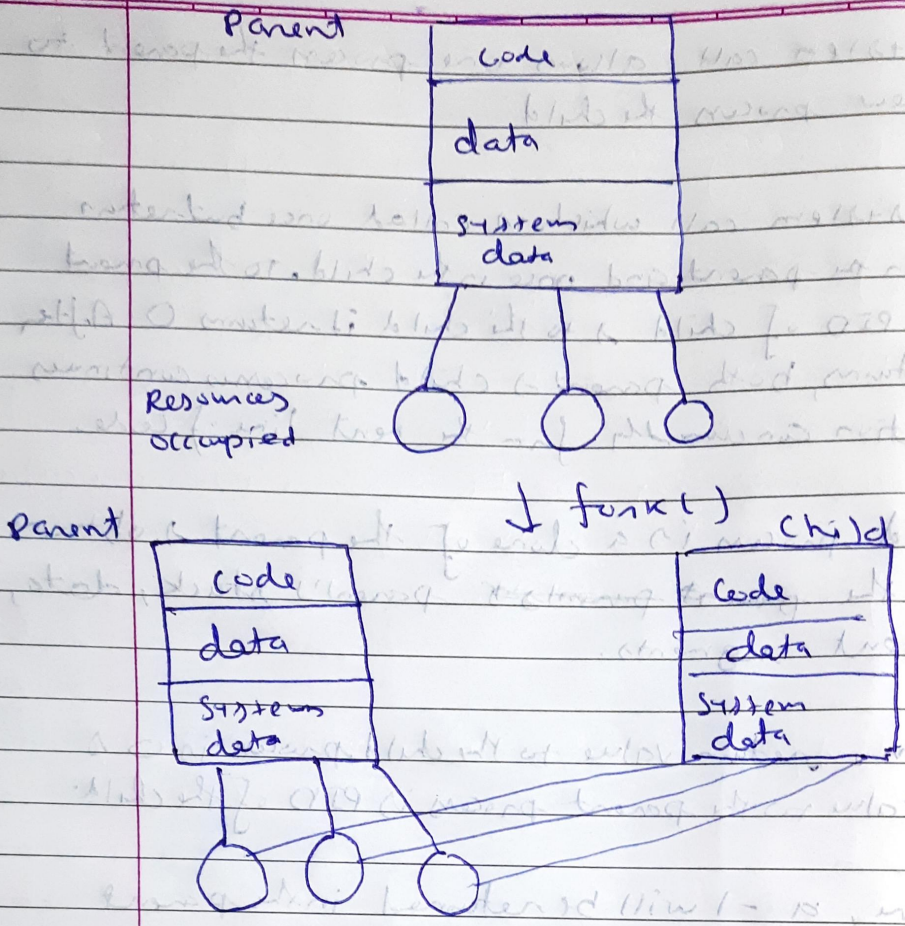
→ On success, return value to the child process is 0 & return value to the parent process is PID of the child.

→ On failure, a -1 will be returned in the parent context, no child process created ~~is~~

→ After a fork(), it is indeterminate which process, parent or child will execute. In most of the cases, parent will execute but parent's state is already active in the CPU & its memory mgmt. info is already cached.

→ fork1.c - simple fork program

→ fork2.c - ~~two~~ child & parent runs concurrently two o.s can be seen by PS - a command.


Parent
code
data
system data
Resources occupied
fork()
Parent
code
data
system data
Child
code
data
system data

2

Orphan Process

→ If a parent has terminated before reaping its children and the child process is still running, then these children are called orphans.

→ They are adopted by init or systemd process to see whether the process has adopted by init or not

$ PS -l (orphan-c)

Here check the PPID of a.out

now,

$ ls -l /Proc/PPid/exe

It will give the name of the process having process id as PPid.

Zombie Process

→ processes which have terminated but their parents have not collected their exit status & has not reaped them are called zombies or defunct. So, a parent must reap its child.

→ When a process terminates but still is holding system resources like PCB & various tables maintained by OS. It is half-alive & half dead but it is holding resources like memory but it is never scheduled on the CPU.

→ Zombies can't kill by a signal, not even with the SIGKILL. The only way to remove them from the system is to kill their parent, at which time they become orphan & adopted by init or systemd.

for an zombie.c, child will become zombie. To check, go to another terminal & use command PS -a. It will show 'defunct' process.

$ kill PID of zombie (not killed)

$ kill -9 PID of zombie (not killed)

$ kill -9 pid of parent of zombie

- now zombie process's parent will be init.

wait() System call

Parent process can monitor the execution or termination of its child process through wait() system call.

pid_t wait (int *status)

-) the process that calls the wait() system call, gets blocked till any one of its child terminates

-) the child process returns its termination status using the exit() system call & that integer value is received by the parent inside the status argument

-) on success, the wait() system call returns PID of the terminated child & in case of errors, returns -1.

-) If a process wants to for termination of all its children, then while(wait(null)>0);

-) A process can end in four ways

(a) Success/failure (b) Killed by a signal (c) stopped by a signal (d) continued by a signal

Exec functions

→ A process may overwrite itself with another executable image. When a process calls one of the six exec() functions, it is completely replaced by the new program & the new program starts executing its main function.

→ There are five library funcs of exec family & all are layered on top of execve() system call. Each of these functions provides a different interface to the same functionality.

→ There is no return after a successful exec call. The exec() functions return only if an error has occured. The return value is -1 & errno is set to indicate the error.

Exec functions

→ A process may overwrite itself with another executable image. When a process calls one of the six exec() functions, it is completely replaced by the new program & the new program starts executing its main function.

→ There are five library funcs of exec family & all are layered on top of execve() system call. Each of these functions provides a different interface to the same functionality.

→ There is no return after a successful exec call. The exec() functions return only if an error has occured. The return value is -1 & errno is set to indicate the error.

int execl(const char *pathname, const char *arg0, ..., (char *)0);
int execlp(const char *filename, const char *arg0, ..., (char *)0);

int execle(const char *pathname, const char *arg0, ..., (char *)0,
char *const envp[]);

→ The 1st argument to this family of exec() calls, is the name of the executable, which on success will overwrite the address space of the calling process with a new p/m from the 2° storage.

→ The 'l' after the exec means that command line arguments to the new p/m will be passed as a comma separated list of strings with a '\0' character at the end.

→ The 'p' stands for path. It means that the p/m specified as the 1st argument should be searched in all directories listed in the PATH variable. However, using absolute path

to p/m is more secure than relying on PATH variable, which can be more easily altered by malicious users.

→ The 'e' stands for environment. It means that after the command line arguments, the p/m should pass an array of pointers to null terminated strings, specifying the new environment of the p/m to be executed. Otherwise the caller environment will be used.

execv family

int execv (const char * pathname, char *const argv[]);

int execvp (const char * filename, char* const argv[]);

int execvpe (const char * pathname, char* const argv[],
char* const envp[]);

here 'v' means the list of argument will not be passed as comma separated list but as an array of pointers.