15

UNIX PROCESSES ENVIRONMENT

* The main() function:

A C program starts the execution with a function called main() and the prototype is:

```
int main(int argc, char *argv[])
```

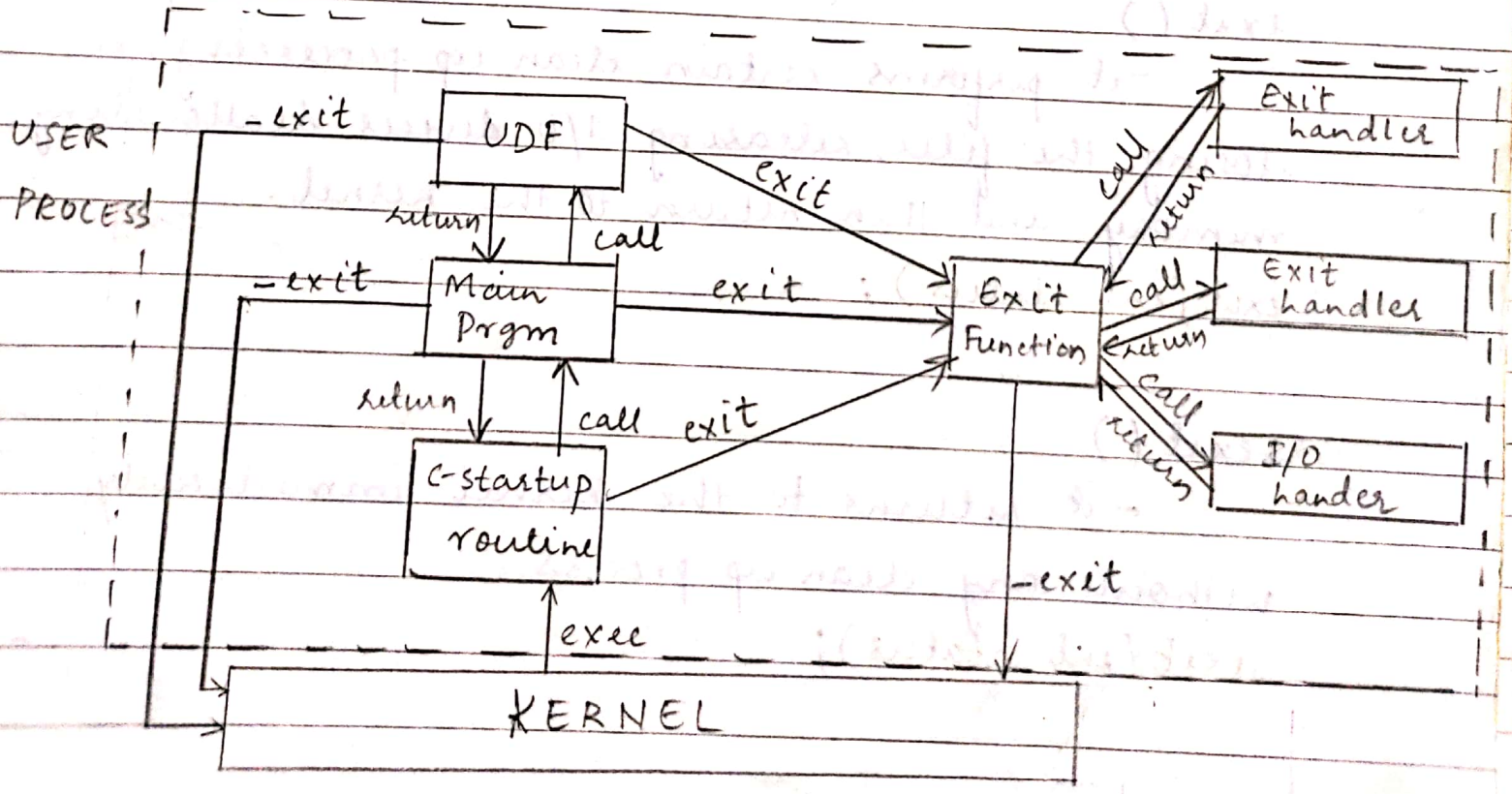
where

argc: no. of command line arguments

argv: is an array of pointers to the command line arguments

When the C program is started by the Kernel a special start-up routine is called before the main. This executable program specifies the start-up routine as the starting address for the program. This start-up routine takes the value from the kernel and sets things to the main() function

* PROCESS TERMINATION:



There are five ways of Process Termination

1. Normal Termination
   - return from main()
   - exit()
   - _exit()

2. Abnormal Termination
   - Calling Abort
   - Terminated by Signals.

* NORMAL TERMINATION

Return from main()

- the return function writes the value to the OS and sets back the control to the kernel, thereby releasing the resources.

exit()

- it performs certain clean-up processing, ie, closing the files, releasing I/O devices, deallocating memory and then return to the kernel.

exit(int status);

_exit()

- it returns to the kernel immediately without any clean-up process.

_exit(int status);

atexit()

- with ANSI-C a process can register upto

32 functions that are automatically called by exit(). These are called as exit handlers and are registered by calling the function atexit().
int atexit (void (*func)(void));

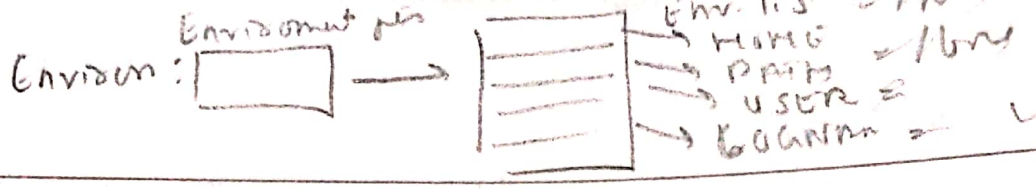
* ABNORMAL TERMINATION

Abort

-it is called when a process is in a panic state. This function terminates the process and in UNIX it causes a 'core' file to be generated. A core file is useful for users to debug an aborted process
void abort(void);

Signals

-Signals are Software interrupts for a program. The KILL signal is used for abnormal termination of a process.
int kill (pid_t pid, int signalno);
The various signals are
SIGTERM (15)
SIGKILL ( )
SIGSTOP ( )
SIGINT ( )



* COMMAND LINE ARGUMENTS

The sample code for knowing the no. of arguements and its location is

```
main(int argc, char *argv[])
{
    int i;
    printf("\n Total no. of arguments are: %d", argc);
    for(i=0; i<argc; i++)
        printf("%s", argv[i]);
}
```

Access to specific environment variable is normally through the getenv or putenv function.

* ENVIRONMENTAL LIST

Each program in POSIX is also passed with a environment list. This list is an array of character pointers. With each pointer containing the address of environment variable followed by NULL terminated C script.

The address of the array of pointers is contained in the global variable environ

Syntax:

```
extern char **environ
```

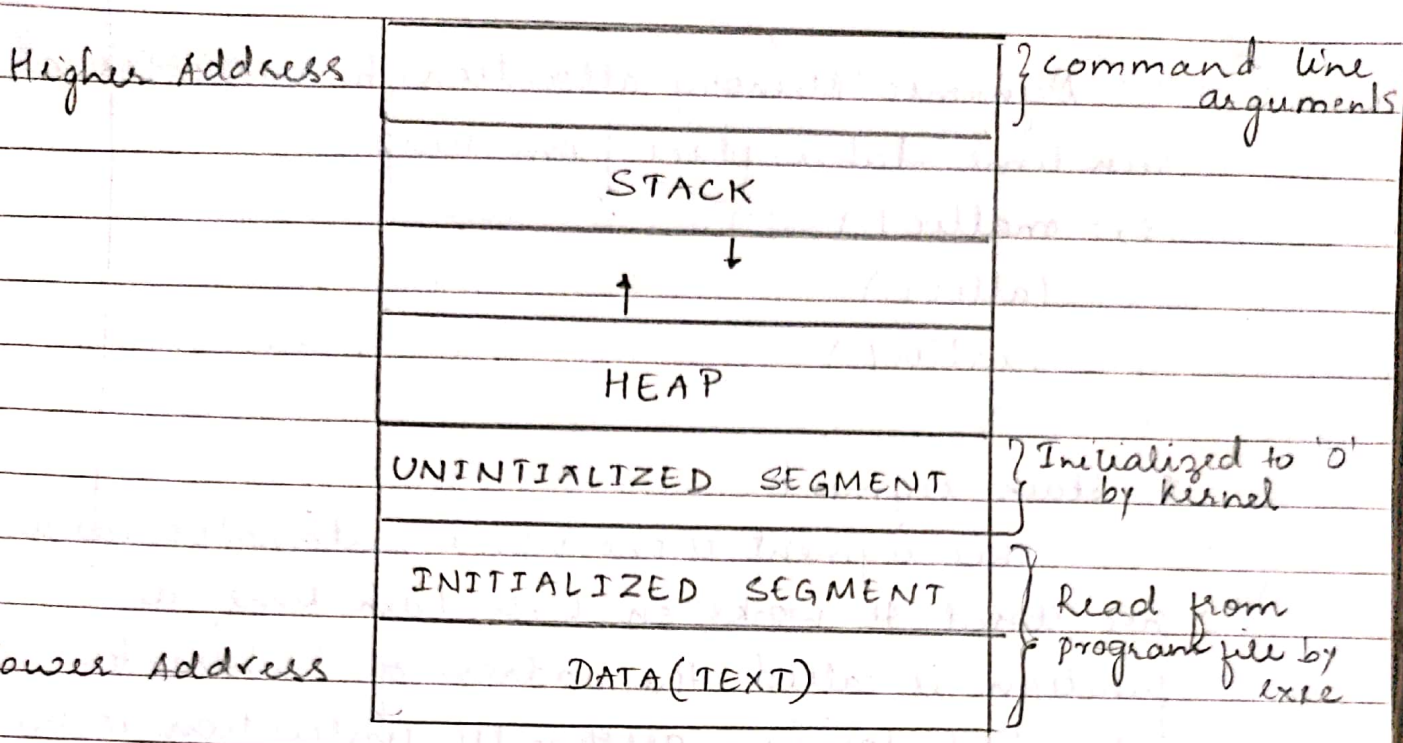
* MEMORY LAYOUT OF A C-PROGRAM

A C-program is composed of the following segments:

1. Text Segment:

This segment consists of m/c instructions that are executed by CPU. Usually the text

int main(int argc, char *argv[], char *envp[])



segment is shareable so that only a single copy needs to be in memory for frequently executed programs like Editor, C-compiler, Shell.

2. Initialized Segment: This is called as the data segment.

This segment contains variables that are specifically initialized.

Ex: int a=3;

3. Uninitialized segment:

This segment is often called as BSS (Block started by symbols). Data in this segment is initialised by kernel to arithmetic zero or NULL pointer before the program starts executing.

Ex: int a[1000];

4. Heap:

Dynamic Memory allocation to variables at run time takes place from Heap.

Ex: malloc()
calloc()
realloc()

5. Stack segment:

This segment is one where automatic variables are stored. It works on LIFO. Each time a function is called, the address of "where to return", performing arithmetic evaluation is saved on Stack.

$ ls -l a.out
$ size a.out

* SHARED LIBRARIES

They remove the common library routines from the executable file by maintaining a single copy of library routine somewhere in the memory so that all processes can reference to the single routine. This reduces the size of the executable file.

* DYNAMIC MEMORY ALLOCATION

There are three functions specified by ANSI C for memory allocation.

1. malloc

It allocates specified no. of bytes of memory from the heap.

2. calloc

It allocates space for specified no. of objects of specified size from the heap.

3. realloc

It changes the size of previously allocated area. When the size increases it may involve moving the previously allocated area somewhere else to provide additional room at the end of memory.

The prototypes are:

```
#include<stdlib.h>

void *malloc(size_t size);
void *calloc(size_t n, size_t size);
void *realloc(void *ptr, size_t size);
void free(void *ptr);
```

Free is used to deallocate the memory.

4. alloca :

This function has the same calling function as malloc. However, instead of allocating memory from heap. The memory is allocated from the stack frame of the current function.
The advantage of this is that, we don't have to free the space, ie, memory is deallocated automatically when the function returns.

* ENVIRONMENT VARIABLES:

These variables control the behaviour of the system. They determine the environment in which the user works. ANSI-C defines a function that can be used to fetch values from the environmental list. The function is

```
#include <stdlib>
char *getenv( const char *name);
```

In addition to fetching the values of the environmental variables sometimes we want to set an environmental variable, the user may want to change the value of the existing variable or add the new variable to the list. This is done by the following functions:

putenv: It takes a string of the form name=value and places in the environment list. If the name already exists its older definition is first removed

setenv: It also sets name=value. If the name already exists then

- If rewrite bit is non-zero then existing definition for name is removed
- If rewrite bit is zero then existing definition is not removed.

unsetenv: It removes all definitions of the name specified.

```
int putenv(const char *str);
int setenv(const char *name, const char *value,
                                    int rewrite);
int unsetenv(const char *name);
```

* setjmp AND longjmp

In C programming we can't GOTO a label defined in another function. To cope up with this problem two new functions have been introduced — setjmp and long-jmp

```
#include<setjmp.h>
int setjmp(jmp_buf loc);
int longjmp(jmp_buf loc, int val);
```

where

loc : is the argument which records the location of setjmp function.

setjmp and longjmp are useful for dealing with ERRORS and INTERRUPTS encountered in low level subroutines of a program.

setjmp saves the stack content in loc for later use by longjmp.

longjmp restores the environment saved by the last call of setjmp.

```
#include<stdio.h>
#include<setjmp.h>
static jmp_buf buf
```

```
int main()
{
  if(!setjmp(buf))
    {
      first(); //when executed setjmp returns 0
    }
  else
    {
      printf("\n main");
    }
  return 0;
}

void first(void)
{
  second();
  printf("\n first"); //never executed
}

void second(void)
{
  printf("second");
  longjmp(buf,1); //jumps back when setjmp was
                  set and returns value 1
}
```

* getrlimit AND setrlimit

Every process has a set of resource limits, some of which can be queried and changed by the getrlimit and setrlimit functions.

```
#include<sys/time.h>
#include<sys/resources.h>

int getrlimit(int resource, struct rlimit *rlptr);
int setrlimit(int resource, const struct rlimit *rlptr);

struct rlimit
{
  rlim_t    rlim_cur;
  rlim_t    rlim_max;
};
```

where

rlim_cur : is the soft limit

rlim_max : is the hard limit value

The various resources are:

RLIMIT_CORE : It specifies maximum size in bytes of a core file

RLIMIT_CPU : It specifies maximum amount of CPU time in seconds

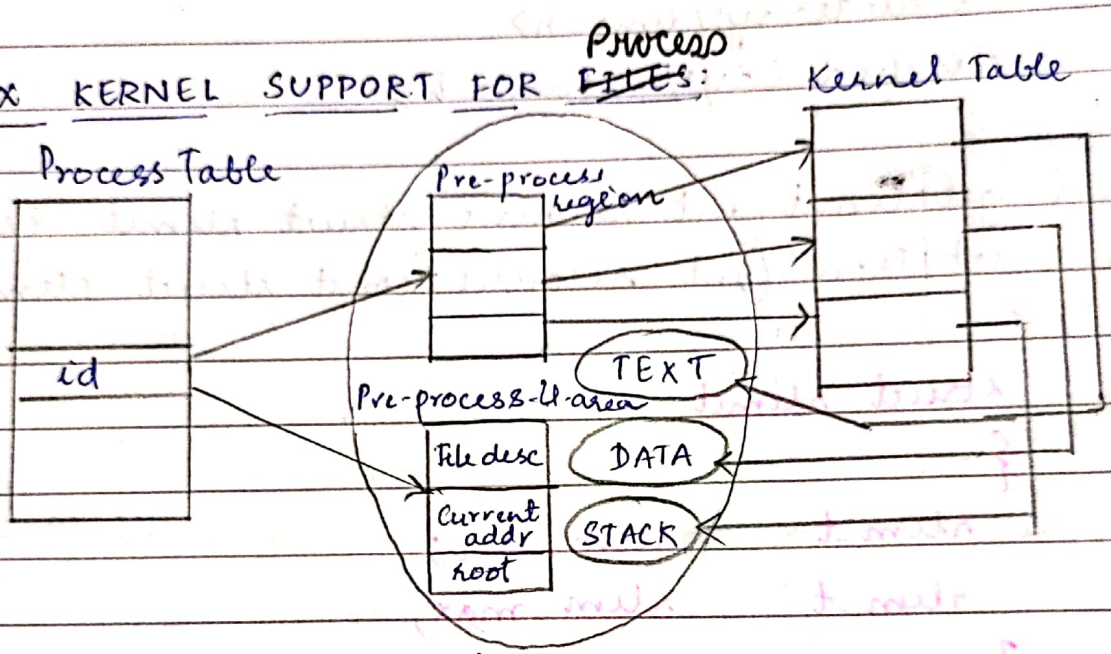
RLIMIT_DATA : It specifie maximum size in bytes of data segment

RLIMIT_FSIZE : It specifies maximum size in bytes of a file

RLIMIT_MEMLOCK : It specifies lock in the memory space

RLIMIT_NOFILE : It specifies maximum no. of opened file in a process

* UNIX KERNEL SUPPORT FOR ~~FILES~~ PROCESS:



A - process

The execution of a process is dependent upon the data structure and the OS. The implementation of it is described in the above fig.

1. UNIX's process consists of text segment, data segment and stack segment.

   Segment is an area of memory that is managed by the system

2. Text segment contains the program text of a process in M/c executable format.
3. A data segment contains static and global variables and their corresponding data
4. A stack segment contains a run time stack.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/resource.h>
#include <sys/time.h>
#include <errno.h>

int main()
{
    struct rlimit limit;
    if (getrlimit(RLIMIT_DATA,
                  &limit) < 0)
        perror(error);
    else {
        printf("%s", "RLIMIT_DATA");

    if (limit.rlim_cur == RLIM_INFINITY)
        printf("infinite");
    else
        printf("%ld", limit.rlim_cur);

    if (lit.rlim_max == RLIM_
        pf("___");
    else pf("%ld", lit.rlim_max);
```

5. UNIX kernel has a process table that keeps track of all active processes

6. Each entry in the process table contains a pointer to text, data, stack segment and U-Area of a process

7. The U-area is an extension of a process table that contains other specific data like file descriptor, current address, root.

DATA STRUCTURE OF PARENT-CHILD PROCESS

refer diag. on Pg 209

1. When a process is created by fork() it contains duplicate copies of data and stack segment of its parent.
2. It also has a file descriptor table that contains references to the same opened files as its parent, such that both share the same file pointer to each opened file.
3. The process is assigned the following attributes when it is inherited by its parent

* RUID : real UID
* RGID : real GID
* EUID : effective UID
* EGID : effective GID
* SID : Session ID
* PGID : Parent Group ID

Following are the attributes that are different for parent and child:

* PID
* PPID
* Pending Signals
* File Locks

03/10

UNIT 5

PROCESS CONTROL

Process Identifier

$ps -ef : process currently executing

UID PID PPID TTY STIME CMD

Each process is uniquely identified by a number called process identifier (PID). The kernel is responsible for management of process. By default two process identifiers are well defined, those are:

pid 0 : It is usually the scheduler often known as Swapper (Bootstrap).

pid 1 : It is usually the init process and is invoked by the kernel at the end of bootstrap procedure.

In addition to these PID's there are other ID's for every process found in

#include <sys/types.h>

#include <unistd.h>

pid_t getpid() ; returns