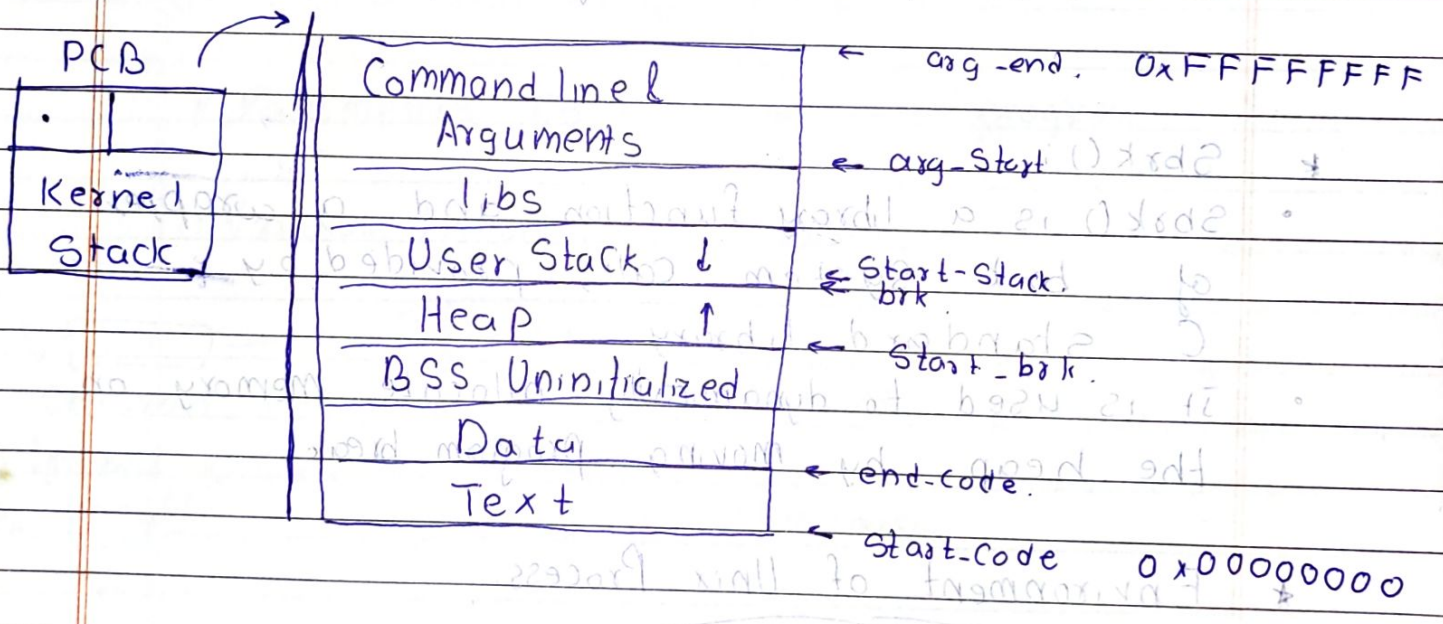
Page :
Date : / /

★ mm_struct and memory layout of a process.

- In Linux, each process has an associated mm_struct which contains all the information about the process's virtual memory layout.
- It is defined in the kernel source (include/linux/mm_types.h) and is crucial for managing memory, performing context switches and handling system calls like brk(), mmap() etc.



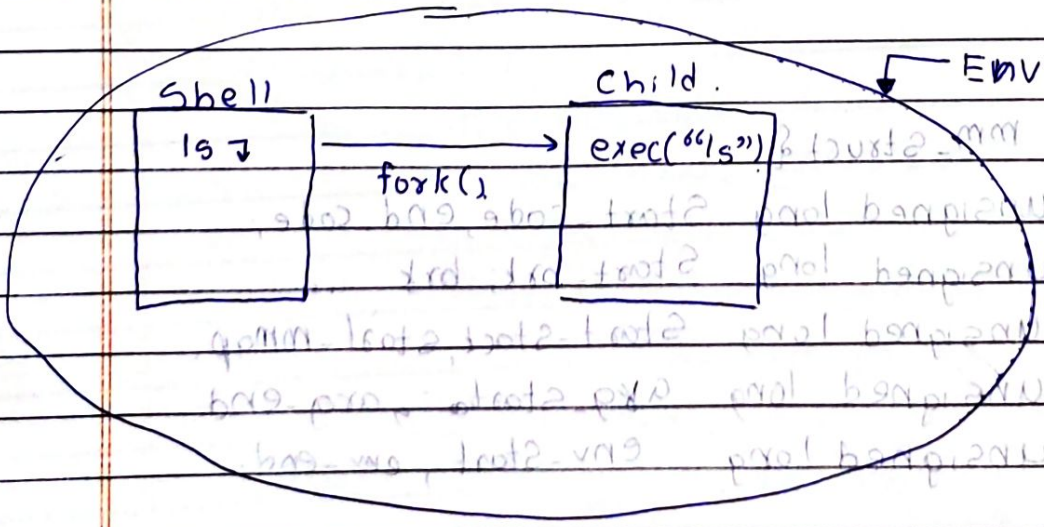

```
struct mm_struct {
    unsigned long start_code, end_code;
    unsigned long start_brk; brk
    unsigned long start_stack, start_mmap,
    unsigned long arg_start, arg_end
    unsigned long env_start, env_end.
};
```

System Call Used to terminate Process. _exit()

```
Start_code    // Start of Code Segment
end_code      // End of Code Segment.
Start_brk     // Start of heap
brk           // Current end of heap
Start_stack   // Start of User stack
arg_start     // Arguments.
arg_end
env_start     // environment
env_end
Start_mmap    // Base of mmap area
```

* Sbrk():
  - Sbrk() is a library function and a wrapper of brk system call, provided by C standard library.
  - It is used to dynamically allocate memory on the heap by moving program break.

* Environment of Unix Process.



Shell (bash

→ fork() → Child Process
↳ exec ("ls")
↳ New Process Image:
- argv = ["ls"]
- envp = inherited from Shell
- memory layout reset

→ wait()

* Parameters Required to run program.

1. Variable known to the Shell.

environ
global Variable

PATH	
HOME	/home/RIK.
USER	
SHELL.	/bin/bash.
.....	

These are environment variables that are known to the shell

2. getenv()
It is a function in C used to retrieve the value of an env variable.

3. putenv()
The putenv() function is used to set or modify an env variable in the current process.

Who creates → init() first process
Shell.

4. export Command.
It is used to mark env variables for export to child process.
ex. export MY_VAR=42.
./my_program

* Set Command:
Displaying Shell variables and functions.

* Signals:
- Signals are software interrupts used for asynchronous notifications
- They inform processes of the occurrence of asynchronous events.
- Signals are a form of inter-process communication
- Processes may send each other Signals with kill system call, or the kernel may send Signals internally.

* Handling Signals.
There are three cases for handling Signals: the process exits on receipt of the Signal, it ignores the Signal, or it executes a particular (user) function on receipt of the Signal

The kill() fun is used to send signals to processes.

int kill (pid_t pid, int sig);

- The default action is to call exit in kernel mode, but a process can specify special action to take on receipt of certain signals with the signal system call.

Syntax:

oldfunction = signal (signum, function);

where signum is the signal number the process is specifying the action for, function is the address of the (user) function the process wants to invoke on receipt of the signal.

Old function was the value of function in the most recent call to signal for signum.

When handling a signal the kernel determines the signal type and turns off the appropriate signal bit in the process table entry, set when the process received the signal.

```
int main(){
    signal (SIGINT, myfn);
    for (i=0; i<5000; i++){
        printf ("I am in loop\n");
        sleep(2);
    }
}
void myfn(){
    printf ("Control C was pressed");
}
```

Output: I am in loop
" " " "
" " " "
^C
Ctrl+C was pressed
I am in loop
" " " "
" " " "

Ctrl C → defaul action is to terminate
But the program doesn't terminate on (Ctrl + C because we catch SIGINT and handle it.

How to find out Signals available in System
kill -l

return 0 in C return control to whom?
→ to OS more Specifically to parent process ex Shell like bash if you run program on terminal

* Standard & Common Signals.

1. SIGALRM - Sent when a timer set by alarm() expires. Used to implement time outs
2. SIGQUIT - Sent when user presses Ctrl+\ ^\ (Ctrl backslash). and also dumps core (for debugging).

3 SIGABRT - Sent by abort() to cause abnormal termination, often for assertions or fatal errors. (terminate process prematurely

4 SIGURG : Indicates urgent data is available on Socket. Used in networking (TCP/IP)

5 SIGPIPE : Sent when a process writes to a pipe with no reader. Prevents broken pipe issues.

6 SIGCHLD (4.3 BSD) : Sent to a parent when a child
SIGCLD process terminates or Stops
(SysV) Used in wait() handling.

7 SIGBUS : Indicates bus error, like unaligned memory access or invalid physical address.

8 SIGSYS : Sent when a process calls an invalid or restricted syscall.

9. SIGCONT : Continue a stopped process.

10. SIGEMT : Emulation Trap.

11 SIGFPE : Floating Point Exception

12. SIGHUP : Hangup - terminal disconnected.

13 SIGINT : Interrupt from keyboard (Ctrl+C)

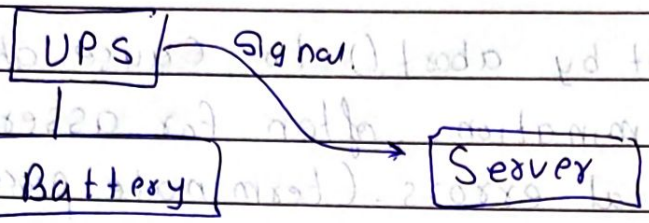
14. SIGIO : Some events occur in IO.

15 SIGKILL : kill the process immediately - Cannot be caught or ignored.

16 SIGPOLL : From pollable device (Pollable event)

17 SIGPROF - Profiling timer expired set by Setting.

18. SIGPWR - Power failure or battery low.



~~19. SIGQUIT~~

19 SIGSEGV - Segment Violated (Segmentation fault)

20 SIGSTOP - Stop the process immediately - Can't be caught or ignored.

21. SIGTERM - Termination request.

22. SIGTRAP - Trap - Used by debuggers.

23 SIGSTP - Terminal stop (Ctrl+z) ^Z

24. SIGTTIN - Background process tried to read from terminal.

25 SIGTTOUT - Background process tried to write to terminal.

26 SIGUSR1, USR2 } User defined Signals.

27. SIGVTALRM - Virtual timer expired.

28 SIGWINCH - Window size changed.

29. SIGXCPU - Sent when process exceeds its CPU time limit

30. SIGXFSZ - Sent when process tries to write more data to a file that its file size limit

Page :
Date : / /

```
#include <stdio.h>
#include <signal.h>
int main() {
    printf("One\n");
    signal(SIGALRM, alarm_catch); // define handler and
    alarm(10);                    // Set the time
    pause(); // Pause program until a signal is received
    printf("two\n");
}
alarm_catch() {
    return;
}
```

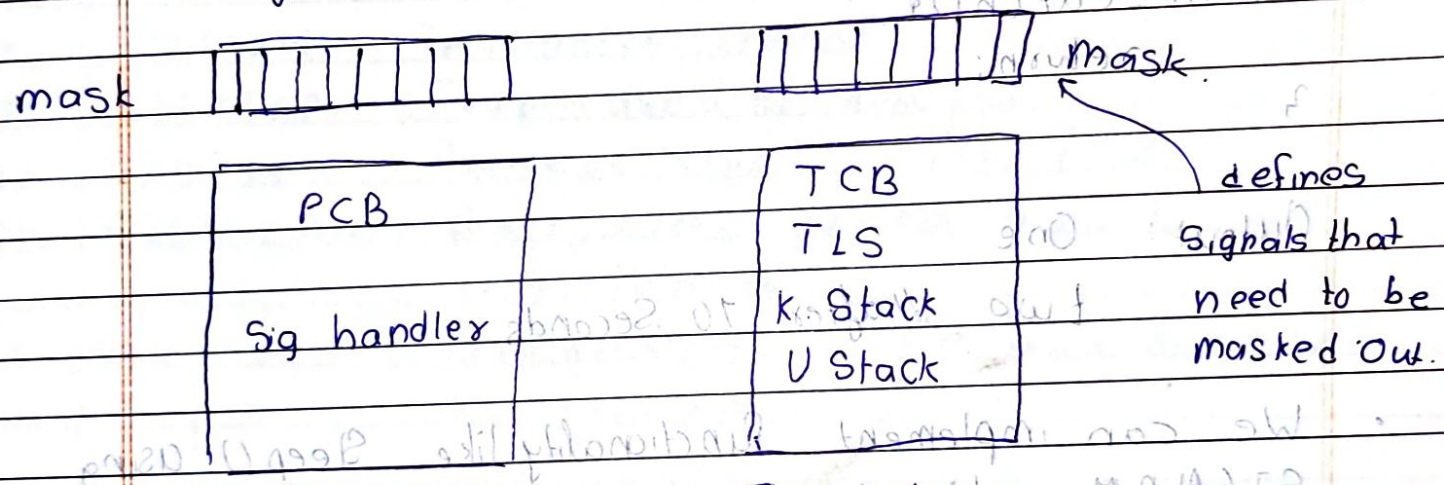
Output: One
two // after 10 seconds.

- We can implement functionality like sleep() using SIGALRM alarm().

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
void segfault(int dummy) {
    printf("Segment fault occurred");
    exit(1);
}
int main() {
    int *p=0
    signal(SIGSEGV, segfault); //Signal handler
    *p=17;
    return 0;
}
```

Output Segfault Occured.

P is pointing to address 0. (Null ptr)
We are trying to store 17 at memory location of P - Which is address 0x0, not allowed.
The address does not belong to this process
∴ Sigfaut occurs.



Signal ⇒ Shared by both process and thread.

```
#include<signal.h>
int Sigaddset(sigset_t *set, int signo);
 Sigdelset(          );
 Sigemptyset(        );
 Sigfillset(         );
 Sigismember(const sigset_t *set, int no);
 pthread_Sigmask();
 int sigprocmask(int how, sigset_t* newmask
                 Sigset_t* oldmask);
```

* Signal Set : It is a data structure used to represent a collection of Signals.

1. Sigaddset : Adds a specific Signal to the Set
2. Sigemptyset : Initializes set to an empty Set.
3. Sigdelset : Removes a signal from the set,
4. Sigismember : Tests if a Signum is in Set.
5. Sigfillset : Fills set with all Signals
6.

- pthread_sigmask : Changes signal mask of the calling thread.

- Sigprocmask : Similar to Sigmask but for Single threaded programs.

- Sigaction() : It is used to define a custom action that a process should take when a specific signal is received.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

Static void sig_quit(int signo){
    printf("Caught SIGQUIT\n");
    signal(SIGQUIT, SIG_D
    // ↑ restore defaut behavior for Sigquit.
    // Next time it's received, the prog will terminate
```

```
int main(){
    sigset_t newmask, oldmask, pendmask;
    signal(SIGQUIT, sig_quit); // set handler.

    sigemptyset(&newmask); // Initialize empty set
    sigaddset(&newmask, SIGQUIT); // Add to set.

    sigprocmask(SIG_BLOCK, &newmask, &oldmask); //Block SIGQU
    printf("SIGQUIT blocked for 10 seconds.
    Try pressing Ctrl+\ now.\n");
    sleep(10);

    sigpending(&pendmask); //Check pending signals
    if(sigismember(&pendmask, SIGQUIT)){
        printf("SIGQUIT is pending\n");
    }

    sigprocmask(SIG_SETMASK, &oldmask, NULL); //Unblock SIGQUIT
    printf("SIGQUIT unblocked\n");
    sleep(10);
    exit(0);
}
```

- We register a custom signal handler for SIGQUIT
- We prepare a signal set containing only SIGQUIT.
- we Block SIGQUIT for 10 seconds.
  If we press Ctrl+\ now, the signal is not delivered, but is marked as pending.
- Then we check for any pending signal.
- If we find SIGQUIT in pending we print SIGQUIT is pending.

- Then we unblock the signals. If the signal was pending, it's now delivered, and the handler sig_quit() is executed.
- Inside the handler, it resets SIGQUIT back to default behavior (ie, the next time it's received, the program will terminate).

Output: SIGQUIT blocked for 10 seconds. Try pressing Ctrl+\ now.
^\ // Pressed by user
Caught SIGQUIT // After 10 sec.
SIGQUIT unblocked
^\ // default behaviour
— end of program

* Process Tracing:
- Process tracing lets one process (like a debugger) observe and control the execution of another process.
- Typical things a tracing process can do:
  - Inspect or change registers and memory
  - Intercept system calls
  - Stop / resume the traced process.
  - Inject breakpoints

- A debugger process, such as Sdb, spawns a process to be traced and controls its execution with the ptrace system call.

- A process (tracer) can observe and control another process (tracee).
- It is used for
  - Debugging.
  - Security auditing
  - Sandboxing
  - Reverse engineering.

* Working
  1. Target process calls ptrace (PTRACE_TRACEME)
  2. Parent process calls wait()
     → waits for child to stop at next signal or exec()
  3. Parent now can use ptrace() system call to
     - read/write into child's memory (PEEKDATA, POKEDATA)
     - read/write registers (GETREGS, SETREGS)
     - Stop/resume execution (SINGLESTEP, CONT)
     - Intercept System call.
  4. Each time the tracee hits a syscall, signal, or trap, the tracer gets notified.

* ptrace()
- Ptrace is a System call that allows one process (usually a debugger) to observe and control execution of another process.
- It is used by debuggers like gdb.
- ptrace Stands for process trace.

```
#include <stdio.h>
#include <sys/user.h>
#include <sys/ptrace.h>
int main(){
    int pid = fork()
    if (pid == 0){
        ptrace(PTRACE_TRACEME, 0, 0, 0);
        execl("/bin/ls", "ls", 0);
    }
    while (1){
        int status;
        struct user_regs_struct regs;
        waitpid(pid, &status, 0);
        if (WIFEXITED(status)){
            printf("Child exited.\n");
            break;
        }
        if(WIFSTOPPED(status)){
            printf("Child has been Stopped.\n");

        ptrace(PTRACE_GETREGS, pid, 0, &regs);
        printf("eip=%lx, esp=%lx", regs.eip, regs.esp);
        sleep(1);
        ptrace(PTRACE_SINGLESTEP, pid, 0, 0);
        }
    }
}
```

Here, the child process calls PTRACEME and executes /bin/ls where Child is stopped at every Sys call and notified to parent.

Parent process waits for child using waitpid.

- Then we trace out execution of ls.
- We prnig different contentents of registers. (extended Instruction ptr, Exlend Stack pointer
- When child runs 1 instruction it Stops and send signal to parent SIGTRAP
- Every PTRACE_SINGLESTEP that generates a SIGTRAP.

- Ptrace Single Step tells child to execute one instruction and stop and send signal to parent.
- Child process is the process we need to debug.

* Syntax:

long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void * data);

request: The operation to perform
pid: The process Id of the target process
addr: Memory address, depending on the req.
data: Data to write or a pointer to receive data, Rep depending on the request

- PTRACE_TRACEME:
  Called by a child process to indicate that it wants to be traced by its parent. The parent can then use other ptrace requests on it. Usually followed by exec() in child

- PTRACE_ATTACH:
  Attach to an already running process (target process is stopped). Similar to what debuggers do when the attach to a PID. Once you are done you can call CONT or DETACH to let it continue.

- PTRACE_DETACH
  Detach from a process that was previously attached. Can optionally continue the process

- PTRACE_CONT
  Continue execution of a stopped process. Optionally, a signal can be sent to the process.

- PTRACE_KILL
  Sent SIGKILL to the traced process. It kills the traced process (terminates).

- PTRACE_SINGLESTEP
  Resumes execution for one instruction and then stop (like a breakpoint step).

* Memory Access.

- PTRACE_PEEKTEXT
  Read a word from the code Segment (instruction Space) (~~stack, heap, global var~~)
  • Used when you need to read machine instructions.

- PTRACE_PEEKDATA
  Read a word (usually 4 or 8 bytes) from the targets memory.
  (stack, heap, global var)
  • Used when you want to inspect variables, buffers, or runtime memory.

- PTRACEPOKETEXT
  Write a word to the code Segment

- PTRACE_POKEDATA
  Write a word into target's memory (data Segment),
  Used to modify variables etc.

- PTRACE_PEEKUSER
  Read a word from the user Area (registers) etc. (debug info.)

- PTRACE_POKEUSER
  Write a word to the user area.

* Advanced.

- PTRACE_SYSCALL.
  Stop the trace at entry and exit of system calls.
  Used to trace Systemcalls.
  Used to intercept every system call - both when call is entered and when it exits.

* Registers and CPU State.

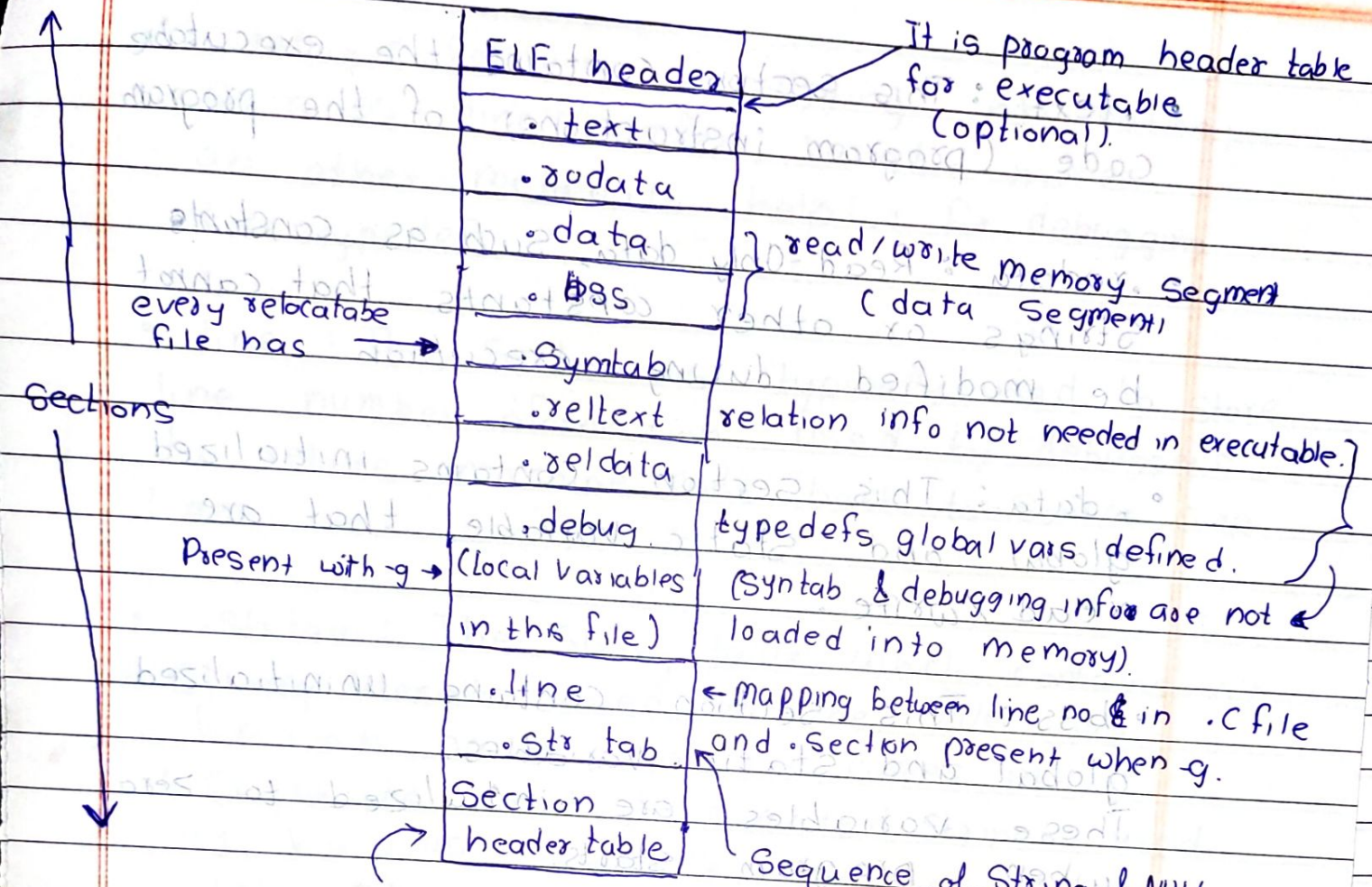
- PTRACE_GETREGS
  Get all general purpose registers. (RAX, RSP, RIP).
  fetch all reg state
- PTRACE_SETREGS.
  Set general purpose registers.
  Used to modify the CPU register value of tracee.
- PTRACE_GETFPREGS.
  Get floating point registers.
  Allows you to read floating point registers of tracee.
- PTRACE_GETFPXREGS.
  Get extended floating point registers of tracee.
  ex. SIMD

There are 16 general purpose registers.

- PTRACE_SETFPREGS.
  - Set floating point registers.
  - Allows you to modify floating point regs of tracee.
- PTRACE_SETFPXREGS.
  - Set Extended floating point registers.
  - It is used to modify the extended floating point registers of a traced process.
  - ex. SSE, AVX.

* What is User area.
  User area typically refers to the portion of memory allocated to a user process in OS.

* Typical ELF Relocatable Object file.
  An ELF (Executable and Linkable Format) relocatable object file consists of several important sections and headers that define how the file is organize
  - An ELF file is a common standard file format used for executables, object code, shared libraries, and core dumps in Unix-like operating systems.
  - It is created by compiler (.o / .obj elf) assembler, linker, debugger etc.



1. ELF Header :

- It contains metadata about the file, such as the architecture, the entry point address, the program header table location, the Section header table location.
- It helps the loader understand how to load and execute the file.

2. Section Headers

These describe various sections of the ELF file.

- .text : This section contains the executable code (program instructions) of the program

- .rodata : Read-Only data, such as constants strings or other constants that cannot be modified during execution

- .data : This section contains initialized global and static variable that are read/write.

- .bss : This section contains uninitialized global and static variables. These variables are initialized to zero when program starts.

- .symtab : It stands for Symbol table, It is sometimes used for debugging and contains symbol information for the compiler or debugger.

- .reltext : It contains relocation entries for the .text section. Used to adjust code references during linking
  - Stores relocation info not needed in executable.

- .reldata : It contains relocation entries for the .data section

- .debug : Contains debugging information, such as line numbers, variable names an other metadata helpful for debugging like typedefs.
- .line : This section is typically used to store line number information used by debuggers to relate machine code back to source code lines.
- .Strtab : The String table, which contains all String literals, used in the file. like function names, variable names etc.
  - It stores null terminated strings used by various sections such as .symbab, .debug etc. section names in section header.

3. Section Header Table:
This is a table of sections that describes each section's characteristics, such as its type, size and where it's located in the file.
- Each section header describes a specific section.

* Types of ELF (Object Files)
1. Relocatable Object Files:
   - File extension is .o or .obj
   They are produced by compilers and.

are used to create executables or shared libraries after being linked.
- They contain machine code, but cannot be executed directly because they are incomplete. (They are intermediate output of compilation process)
- have Section headers.

2. Executable ELF
- File extension usually no extension often just file name like a.out or custom extension like .elf .exe
- They are the final output produced after linking all object files.
- They have program headers.

3. Shared Object file :
- File extension is .so.
- They are dynamically loaded libraries that can be shared among multiple processes.
- It has both Section headers and program headers.
- It contains code and data that can be shared by different programs.

4. Core dump file
- When program crashes the data is stored in core dump file where we can check what went wrong.
- Primarily used for debugging.