بسم الله نبدأ... رب اشرح لي صدري ويسر لي أمري

System Programming Concepts

3.1 System Calls

it is a Controll entry point to the kernel allow process to request kernel to perform some action on process behalf

ممكن نقول انها وسيلة لل Process انها تخلي ال kernel تعملها actions معينة بناءً على طلبها لأنه ال Process بتكون في ال user space اللي ملوش أكسس على أي حاجة إلا بواسطة ال kernel

ممكن بال System Call اعمل Creat لـ Process جديدة أو مثلاً اعمل pipe علشان ال processes تقدر تعمل Communicate مع بعض وهكذا

قبل ما ندخل في ال System Call لازم نعرف شوية حاجات

- ال System call بتغير حالة ال Processor من user لـ kernel mode علشان نقدر نكسس الميموري اللي ال kernel بيحميها.

- عدد ال System Calls في distribution ثابت موجود في file هنعرفه بعدين

- كل Call System ليها عدد من ال arguments محتاجاهم

/ /

ال call system بتبان كأنها C function عاديه خالص وبسيطه لكن اغوص أعقد من كدة. فى شوية خطوات لازم تتعمل :

- لازم يحصل invoking لـ Wrapper function وهنعرف معناها

Wrapper function :-

هى function جيه بتنادى نفسها

```
Void print_greetings (Char *name){
    Printf ("Hello %s \n", name); }

int main () {
    Print_greetings ("User"); }
```

هنا ال Print_greeting هى مش بتعمل حاجه غير انها تنادى Printf لاصليه وبس بتاخد argument والى اقدر اعمله من غير Wrapper function حاجه لكن الموضوع له فوايد زى Abstraction - Modularity - Extensibility

وكنا حاجه عمات وزارة ال call للSystem انا مش بنده كطول أنا بنده ال Wrapper الى موجوده فى GNU lib C وهى الى بنده زى ماكنت بعمل فى ال write مثلا مش بندهها هى لكن بنده ال Wrapper بتاعتها

- ال Wrapper هياخد كل ال arguments بتاعت ال system call
اللي محتاجاها عشان تباصيها لل system call trap handler routine
وال routine دة هيستنى بـ أن ال kernel تلاقيهم في Registers
معينة وتبدأ تتعامل معاها

- ال Wrapper هياخد ال system call handler ويحطه في
register معين

- ال Wrapper هتنفذ instruction معينة لل HW بتاعي تخلي
ال CPU يروح من ال User mode لل kernel mode
فمثلا ال X86 - 32 مثلاً هتنفذ instruction اسمها int 0x80
بتفرق من Architecture للتانية وممكن اجيبها بالنسبة لل Arch بتاعي
من man Sys Call

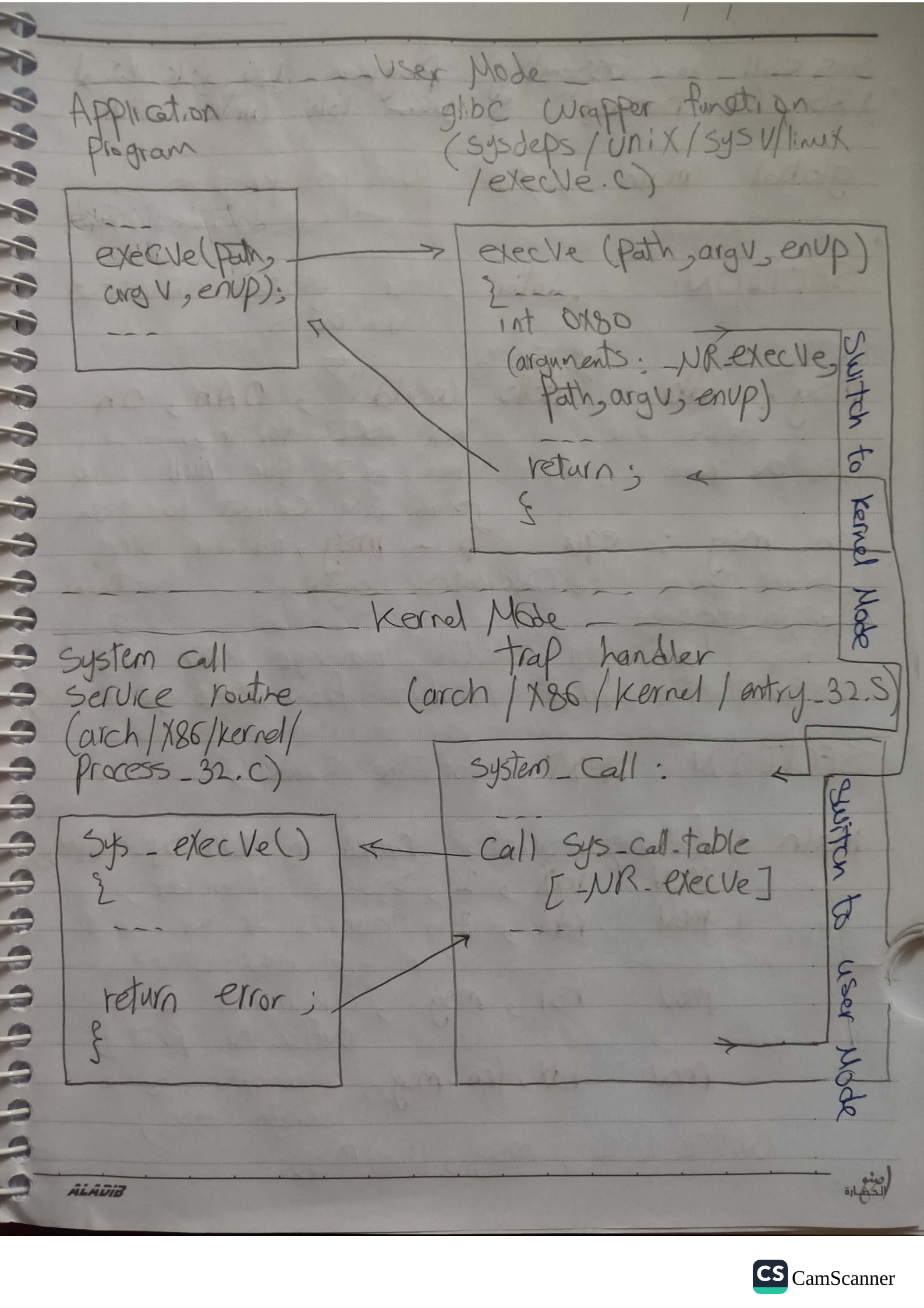
- بعد ما انفذ ال 0x80 int دة ايه هيحصل؟
System call routine to handle trap
ال handler دة هيعمل ايه؟
① هيروح يحط ال registers القيم بتاعتها في ال Stack لل kernel
② هيشوف هل ال System call ال number بتاعها موجود و valid
③ بناء على رقم ال Sys Call هيروح ينفذ ال routine بتاعها
ولو هي بتاخد arguments هيشوف هم valid ولا و بعد ما ينفذ
ال routine وهيعمل transfer لل data بين ال user memory
وال kernel memory بعدها يرجعلك Value بعد ما تنفذها
④ هيرجع ال Values من ال kernel stack، ومنهم ال return
بتاعت ال Sys Call

⑤ هيرجع من ال Wrapper ودة معناه اني هرجع لل User mode
تاني

الموضوع شبه اي interrupt بيتعمل بعمل ال push ال Context
في ال Stack عشان متضيعش وارجعلها تاني، بعد كده بروح
أتأكد من رقم ال Sys call دي اصلا هل هي موجودة ولالا
ولو ليها arguments هيروح يتأكد هي Valid ولالا
لو كله تمام هيروح ينفذ ال handler بتاع ال SysCall
دي بناءً على رقمها اللي هو نفسه شبه ال interrupt بيكون في
IVT فيه عناوين ال ISR بتاعت كل interrupt وال
لما interrupt معين برقم معين بروح ينفذ ال ISR بتاعتها
وبعد ما اخلص هاخد ال return بتاع ال Sys call دي
وارجع ارجع ال Context switching من ال Stack ومعاها
ال return الموضوع شبه ال SW interrupt

- لو حصل اي error ال Wrapper هيعمل set
ال global Var اللي اسمه errno

وفي ال Linux عموما لو رجع ليا Value موجبه ابقى تمام
لو سالبة يبقى حصل error


User Mode
Application program
glibc wrapper function
(sysdeps/unix/sysv/linux/execve.c)
execve(path, argv, envp);
execve (path, argv, envp)
int 0x80
(arguments: __NR_execve, path, argv, envp)
return;
Switch to kernel Mode
Kernel Mode
trap handler
(arch/x86/kernel/entry_32.S)
System call service routine
(arch/x86/kernel/process_32.c)
system_call:
call sys_call_table [__NR_execve]
Switch to user Mode
sys_execve()
return error;

لو مثلا عايزين نعمل sys call بتنفذ assembly Program
"X86_64 based" يعمل إيه write الـ

global main ; entry Point of Program ;
ويكون المسمى اللي اختاره

SECTION .data ; for global initialized Varibles ;

msg: db "Hello world", 0Ah, 0h
- db → define byte
- msg → Varible name
- "Hello world" → my string
- 0Ah → (hex) ASCII for \n
- 0h → (hex) ASCII for null

Len_msg : equ $ - msg
- Len_msg → Varible stores length message
- equ → Calculates length
- $ → indicating current location
- msg → indicating string location

; This line will calculate massage length as it subtract current location from message location;

SECTION .text ; for the code;

```
main : ;following instruction based on man syscall page;
    mov rax , 1 ; sys call number.
    mov rdi , 1 ; argument 1 for write
                  File discribtor;
    mov rsi , msg ; argument 2
                  buffer to Print;
    mov rdx, len_msg ; argument 3
                  length msg;
write ( 1 , msg , len_msg )
```

/ /

Sys Call ; instruction for SW interrupt ;

```
mov r15, rax ; return Value ;
mov rax, 60 ; exit sys Call number ;
mov rdi, r15 ; return Value is the
              first argument of exit ;
sys Call ; exit ( write_len )
```

```
nasm -f elf 64 sysCall 64.nasm
ld sysCall64.o --entry main -o my64 sysCall
```

library functions :-

هي ال functions اللي موجودة في ال C واللي ممكن تعمل Call لـ System Call زي ال printf بتنادي ال write

أو ممكن متعملش كادي زي string manipulation

وفي الاصل library functions هي layred على systemCall

زي ال fopen() هي بتستخدم open()

معمولة ليه ؟ حاجة اعمل عملها أكثر لـ user friendly

الصعم هو ال glibc

Handling Errors from system calls and library functions

دايما ال function سواء كانت system call او library function
بترجع Value تقولك نجحت ولا، ولازم دايما ا check
عليها عشان لو فشلت اهندل ده مطلوب error message

ممكن الواحد يقول مش هعمل check على ال return Value
عشان اوفر شوية وقت في الكتابه لكن ده هيخليها اضيع وقت
كبير جدا في ال debugging لو حصل fail ف لازم دايما نتعامل

وفي شوية system calls عمرها ما هيحصلها fail
زي ال getpid() وال exit().

Handling System call errors :-

عشان اعرف اهندل ال errors لازم اقرأ ال man page بتاعت
كل system call واعرف هي ممكن ترجع ايه في كل حالة
ممكن اعمل ال message error وممكن اعمل set
لل errno ودة global var موجود في <errno.h>
ودة فيه Value لكل error ممكن يحصل وترجعه ال syscall
وممكن ا check عليه بعدين واستخدمه، وكله في ال man

طب لو محصلش ال error هتكون ال errno بصفر
وفي ال SUS هي بتقول & هخليها برقم موجب ما عاليا
كل مرة لازم اشوف حصل error ولا، ومعظم functions بترجع
-1 لو محصلش error ف نقرأ ال man page
افضل بعض
في function اسمها perror بتطبع ال error
وفي strerror الفرق بينهم

Perror ( );

هنا هنطبع مثلاً ال string اللي اختارة وبعدة ال error
اللي موجود من ال msg ودة argument في ال error
او تصلحه هو نفسه لوحدة.

Void perror (const char * msg);

Strerror ( );

هنا دي بتاخد ال errnum int ودة بـ indicate
لـ error message معينة وترجع pointer to char
او string ال error message دي

Char * strerror (int errnum);

ولو مثلاً ال errnum دة غلط هنطبع unknown error name
وساعات ممكن ترجع NULL

Handling error from library functions

- ممكن بعض ال library functions تتصرف زي ال system calls
بالظبط من الموضوع دة

وممكن ترجع 1- او تعمل set ال errno

وممكن متستخدمش ال errno خالص ودي اكيد مش هنعمل check
على errno او استخدم perror او strerror

نقرأ ال man page هنا برضو

/ /

Command line options and arguments

فى حاجة ممكن اديها لـ اى Command اسمها options
ودى اما تبدأ بـ (-) بعدها حرف
أو (--) بعدها string
ودى بنستخدم library fun اسمها getopt()
عشان اعملها Parsing

getopt() , man 3 getopt()

prototype :

```
int getopt(int argc, char * const argv[]
        , const char * optstring);
```

دى هتاخد الـ argc والـ argv بتوع الـ main
وهتاخد كمان String عبارة عن الـ options
وهتبدأ تعمل Parsing لو الـ element من الـ argv
بيبدأ بـ "-" او "--" هتعرف أن دة option
وباقى التفاصيل فى الـ man Page.