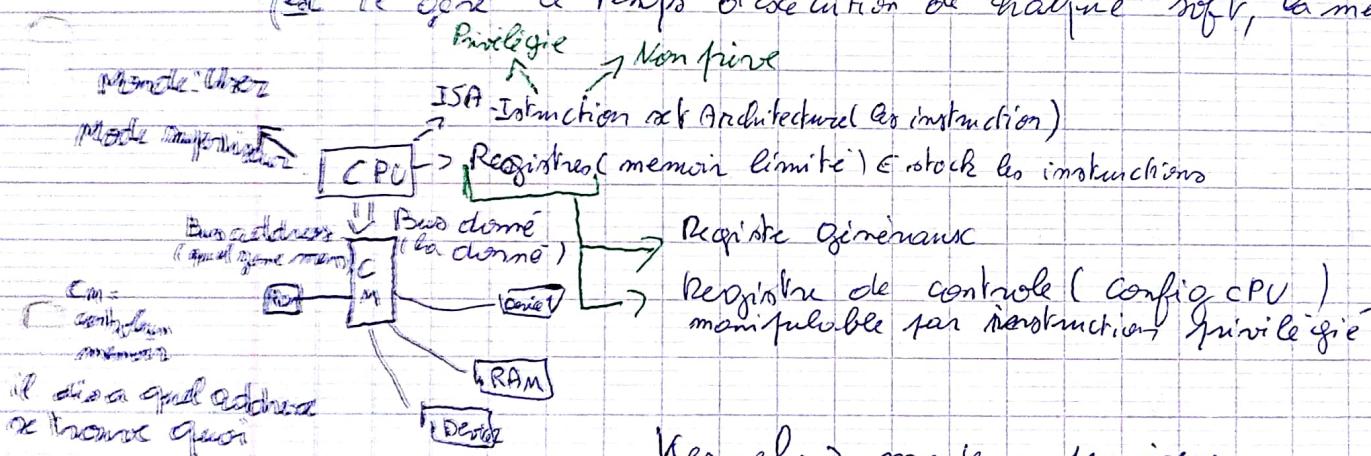


OS : met en place des ressources aux programmes, manipule des abstractions par les ressources direct.
Fournit un environnement simple à manipuler pour les programmes.

Kernel bout de code privilégié qui manage les ressources, il expose les abstractions.
ressource restante au kernel qui les redistribut.
(sic il gère le temps d'exécution de chaque soft, la mem...)



Kernel → mode superviseur

Autre programmes → mode user

Kernel → le moins de choses possible devant pour pas faire de la merde.

Mécanisme → Kernel / comment on s'en sort → user

Syscall → API du kernel

API ; application programming interface (protocole ^{win} pour échanger pour récupérer de la donnée etc..)

Context switch → sauve l'état, faire le syscall et tout redéclencher.

Kernel sur disk donc code pour lancer kernel
Device simple pour tout lancer.

PC → Programme Counter souvent c'est le processeur dans ses instructions

IP → instruction pointeur

Dans toute mémoire ROM on met le code de démarrage ROM (Read only mem) et dans c'est un Firmware on trouve (Firmware) dans quasi tout les device GPU, HDD... → BIOS, uEFI il initialise le contrôleur mémoire et charge l'OS.

Firmware → Bootloader → Kernel → Rootfs monté → /sbin/init → Login

Linux fait avec init SysVinit, ou systemd → redhat

Ces init sont souvent des script shell.

run level : différents niveau pour initialiser différents trucs
SysV exécute les dépendances par ordre alphabétique

NTP Network Time protocole → récupérer le temps.

DNS : premier élément à une IP

Login Authentication : Je sais que

Authorisation/permission à quoi tu fais faire ça.

Taille block 2 * 1024 → CD

read (fd, buf, pg)

Struct file { → struct metadata sur ce fichier b. tailles... }
Struct inode à lire

off -> offset

mapping → je veux sa et le différencier l'autre si des fois quand on veut.

mmap : void *ptr = (mmap(NULL, , PROT_READ, MAP_PRIVATE, fd, 0))

stat → avoir taif fichier
stat.size

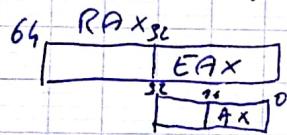
Assembleur

- Intel software Developer Manual
 - I → user chap 5, 3
 - II → ref
- GNU AS (gao manual)

} ressources

Registers :

- RIP (adresse next instruction)
- RSP (pointeur haut de la pile) et RBP
- RDI, RSI
- RAX, RBX, RCX, RDX] on fait ce qu'on veut
- R8 → R15
- flags (flags (Z, N, O, C) pour condition)



Portionne de certains registre
étant directement accessible par
des sous registre

ex: %rax, \$0

Jne toto

mov \$1, %rax

ret

toto: mov \$0, %rax

ret

if (a == 0)
 return 1; label

else

 return 0;

Les opérations d'instruction ont des tailles variables.
permet d'avoir des instructions les plus utilisées
occupant moins de place donc code plus rapide
et moins de déréf mémoire.

En mode immédiat le chiffre est sur 6 bytes

taille instruction:

b:1 ; w:2 ; l:4 ; q:8

cmpl %rax, \$0

je toto

mov \$1, %rax

toto:

ret

| if(a == 0)

| return 1;

| else

| return 0;

ELF → structure de données exécutables

| shared object (.so)

| Relocatable object (.o)

| core dump

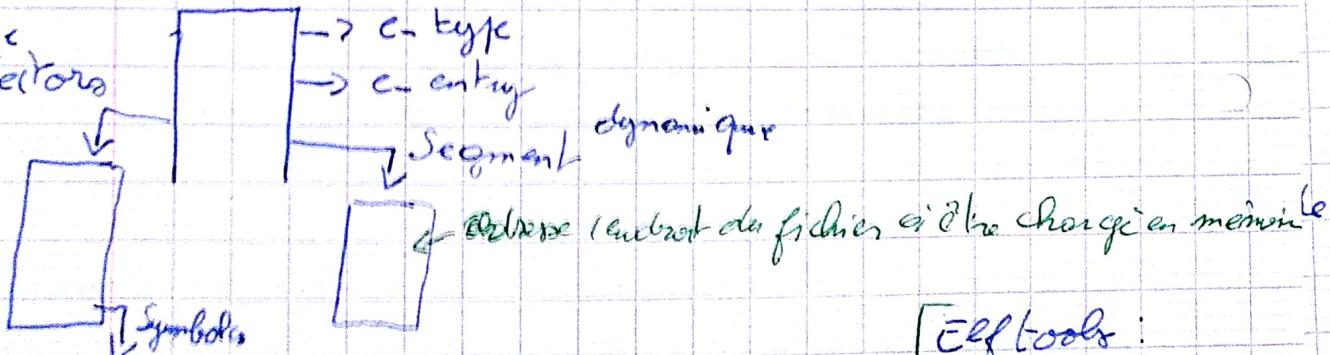
Static lib : ELF → (c'est une archive de .o)

Compilation: Processeur, compile en assembleur, assembleur transform en .o, link

.o → du code avec des trous pour les adresses

header ELF ou début du fichier

Statics
Sectors



Toolchain

- Binutils
- = objdump
- = ld
- = linker
- = libelf
- = elfedit
- = ldd

Binutils

Elf tools:

- nm
- dddump
- readelf
- gdb
- lbt / gas

GFLAGS - setjmp → sauver toutes les étapes

• file } directive } dans
• * } } table de symboles.

• * : → label

• file "test.c" → démonte symbole pour savoir quel c file le code appartient.

Paramètre dans fonction → registre ou pile car local il ne faut pas qu'ils soient global.

Paramètre sont empilé en sens inverse (droite à gauche) → utile pour fonction variadique.

calling convention:

→ valeur retour dans RAX ~~RDX~~.

paramètres:

int → rdi, rsi, rdx, rcx, r8, r9 } n + de 6
float → r8f, r9f, r10f } paramètre on met
return value in rax } dans la stack

float → xmm0, xmm1, xmm2 ... 6

• section directive pour out put data

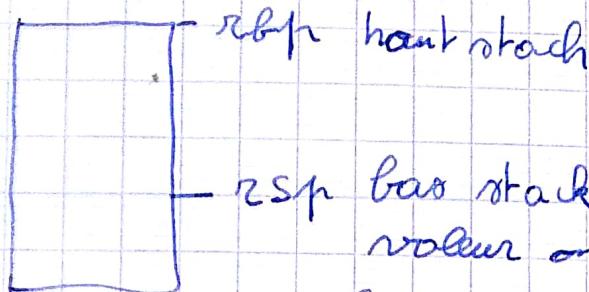
• nodata section qui rappelle rodata

• text = .section .text → où on il y a le code

() → déréférence

Push pop → pour la pile

Stack frame on crée un stack pour la fonction definie RBP
finit RSP



RSP bas stack Si on ajoute valeur on descend rsp
se agrandi par en bas.

Leave → inverse de prolog:
mov %rbp, rbp
pop %rbp

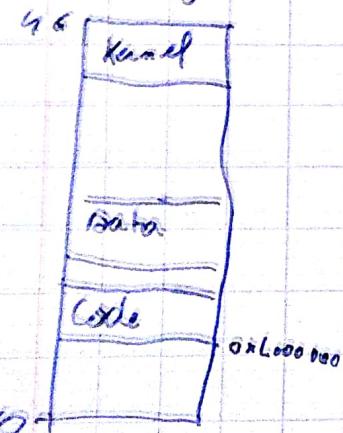
Espace d'adressage: range d'adresses disponible

0 → 4GB

divisé zones 4KB

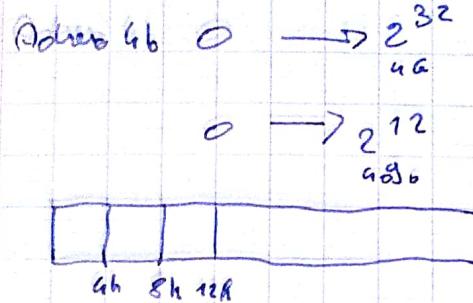
* Ces zones sont des pages ($\times 86$ 4K)
→ block (512 K, 2 K, 4 K
HD CD)

Non Privé

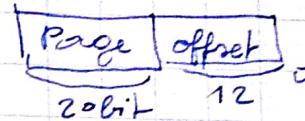


Mémoire direct : Read Write, exec,
superUser, user

Mémoire virtuelle répartie en
environnement de mémoire
contigüe.



$2^{10} \rightarrow 1024 = 1K$
 $2^{20} \rightarrow 1M$
 $2^{30} \rightarrow 1G$



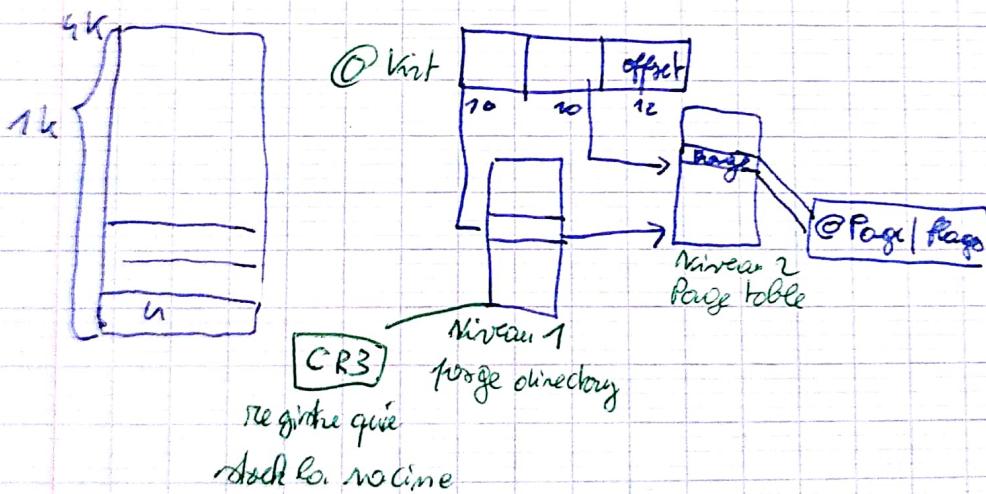
Struct {

420 physique addr
412 flags (present, R,W,X, System)

}

Pages [2^20]

Arbre de hauteur 2 B-tree avec des nœuds de 64K



LRU → algo de cache

Le cache TLB stock les adresses fréquentes de la page table. TLB flit à chaque fois qu'on change d'adresse space.

Il contient des pages globales qui changent pas par le Kernel qui est mapé dans tous les programmes.

Exception: un moyen asynchrone utilisé par le processeur pour faire remonter une erreur.

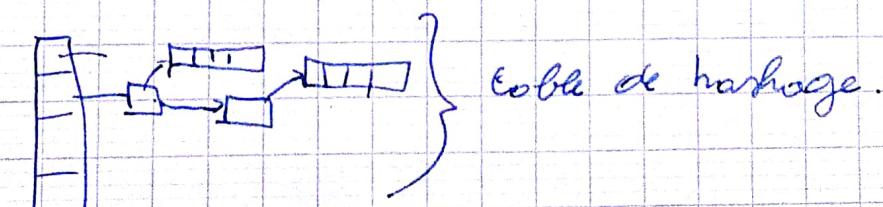
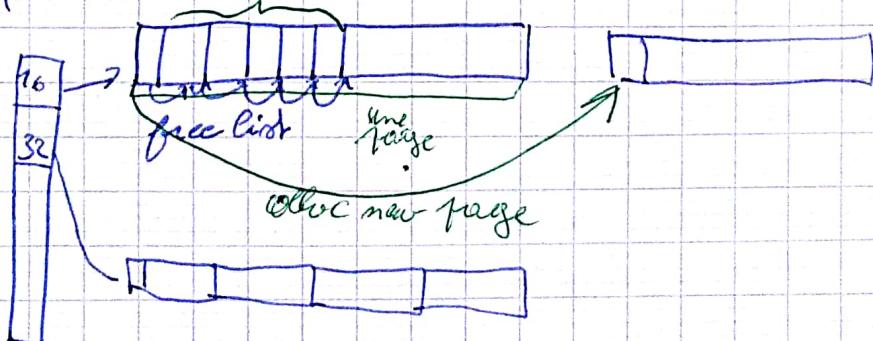
Portiel // Page Fault → erreur si problème dans l'adressage
→ accès sur une zone non mapé
→ Si on a pas les bons droits.

Le Kernel fournit des abstractions pour utiliser le mapping:

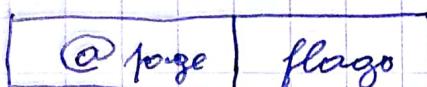
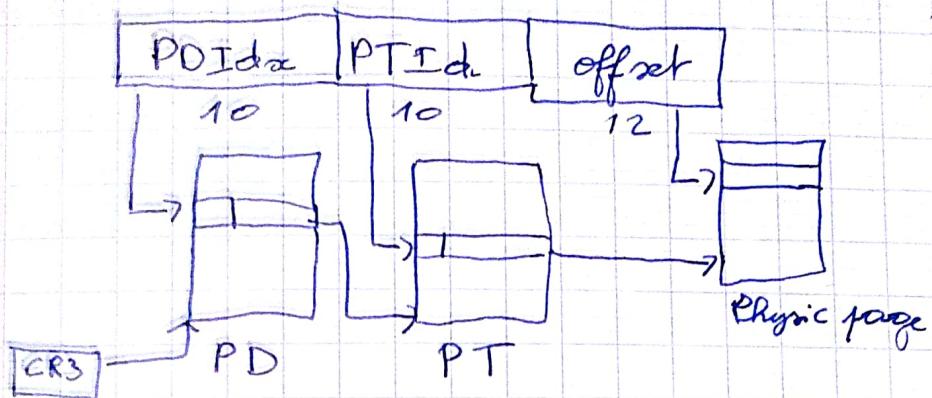
mmmap }
munmap } overflow
 | malloc (A - B)
 | memset (0)

redoc : | malloc
 | memory
 | free

maintenir la liste de blocs libre ~~plus~~ que les blocs en taille.



Toujours garder 2 blocs avec de faire a new page.



P kernel met à 0, CPU met à 1

- R / N } Dirty - 1 si on a écrit

- S / V } Access → a-t-on le droit pour
avoir accès à cette page ?

- NX (PAE) - Cache

- AVL

Cache :

- NoPC

- write through

- write back

→ pas de乐观主义 en cache

politique de cache

- On demand Paging : Donner la ram au dernier moment on affiche l'erreur pour donner
- Swap : on déplace sur le disk.
- File mapping
- Shared memory
- CoW (copy on write)

Page Fault : → zone non mapée
→ pas permission d'accès.

Page fault

Stack :

RIP (pointeur de stack)

CS : si RIP pointe code ou informations

RSP :

SS

[effacement]

error code (Present, User/System, Read/Write, Instruction/Block)

Register

CR2 (fault-addr)

Non Present \rightarrow X

\hookrightarrow map (perm, getfree - 18(1))

Non Present \rightarrow R \rightarrow lire qq ch non mappé et donne

\hookrightarrow map (RO, zero, pfr) toujours la même page avec des zeros en read only

Non zone RO

\hookrightarrow map(MR, copy (tg)) on donne une
page initialisée

OOM killer : tuer les programmes n'ont plus de
Out of memory killer
RAM.

Ces tue ces processus qui sont pas en root et
qui demandent beaucoup de mémoire.

Process

Process } Task (même chose pour le kernel.
Thread }

Tache :

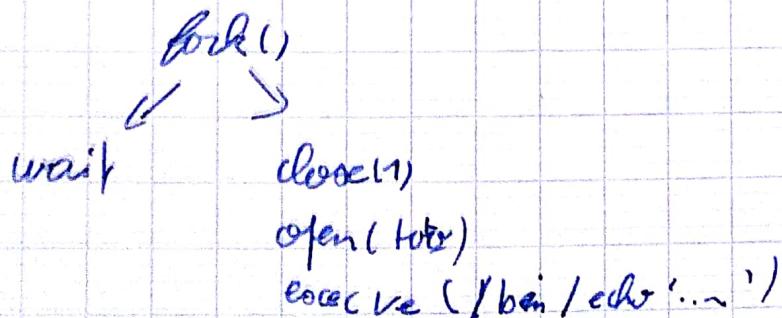
- Copie d'adressage
 - Context (register)
 - FDTTable file descriptor open
 - Signal Handlers Table swoh & qui en fait si la réception d'un signal.
 - File system info
 - Cwd (current working dir) → chdir()
 - root → chroot()
 - rlimit : rlimCPU, limit nb fork, mem virtuel
- fork() → new process
done()

Meta Data:

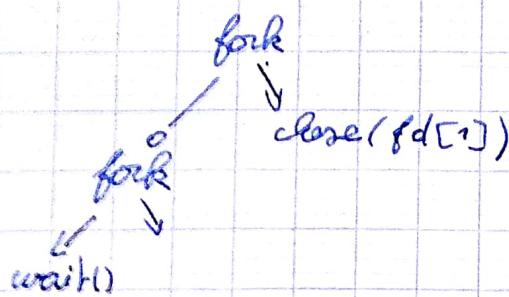
Process ID
PID UID EUID
TID GID EGID
Thread ID

File Descriptor open assign le plus petit disponible

\$ /bin/echo foo > tot.o



\$ /bin/echo foo | cat



int execve (char *filename, char ***argsv, char ***envp

execve ("bin/ls", ["ls", "toto", 0], environ);

env
!
ARGSV
environ
argsv
argsc

Changer une variable d'environnement pour un seul programme :

\$ VAR=a ls toto
setenv ("VAR=a")
execvp ("ls", ["ls", toto, 0])

Poll (struct pollfd *) savoir dans une list de fd des quels sont trés.

epoll -> crée une queue d'évenement

Quand on fork on wait!

int wait (pid_t status, sig) → wait le prochain fils qui meurt
waitpid(pid, ..., ...)

void sigHandler (int)

{

Save errno

fork();

re = wait(0, WNOHANG)

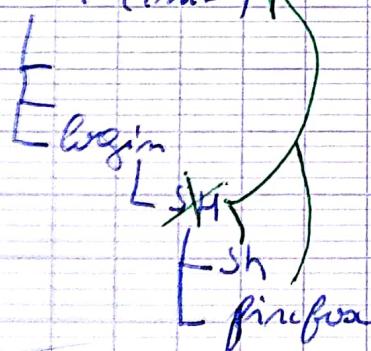
if (re == -1 & errno == NO CHILD)

return;

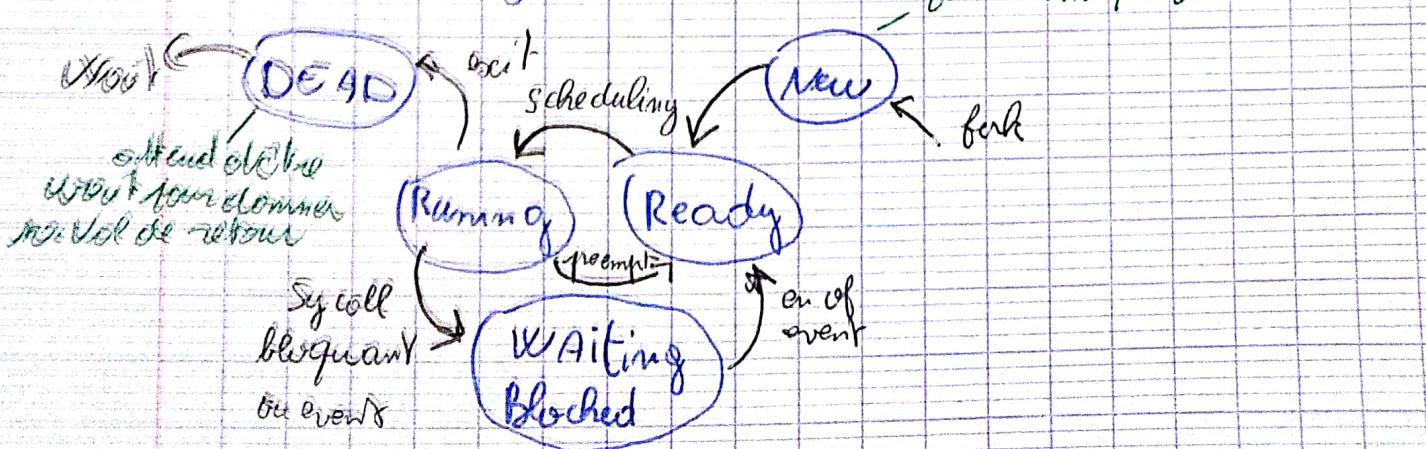
}

revert errno

Pid1 (init)



Scheduling:



Alg scheduling.

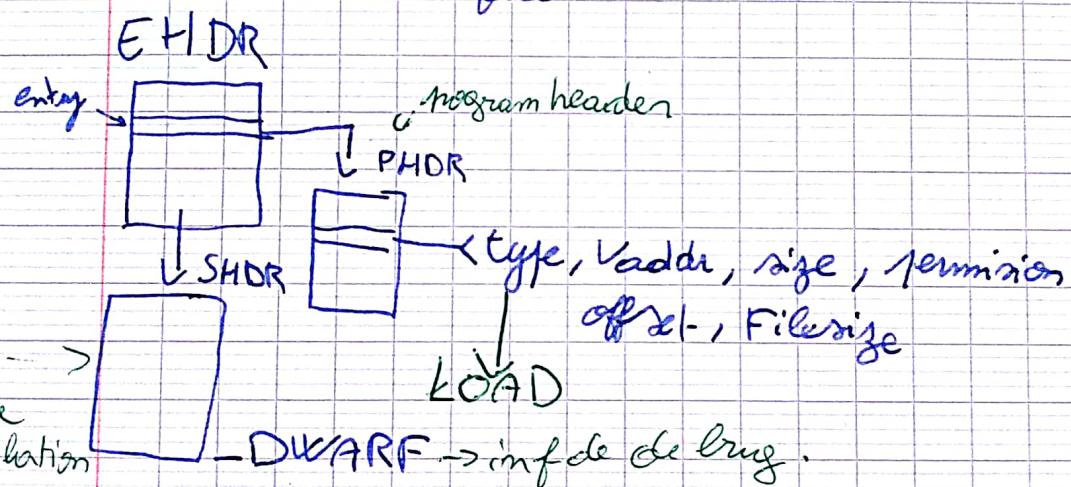
FIFO → algo sans préemption

on met nouveau program ready dans la queue
on run on le remet à la fin de la queue.

Round-Robin

Quantum de temps

ELF file



int toto = 12

Toto . Long 12

int foo () {
 return toto;
}

foo:
 mov toto,%eax
 ret

int main () {
 return foo();
}

main:
 call foo
 ret

• Les différents segments:

- text → le code
- data → variable
- .rodata
- .bss
- .symtab
- .strtab
- rel.text → relocation pour le link à celle adresse

- gl link address de l'igne de code

int de type

on vont la variables dans la pile

Lib dynamique:

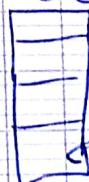
main:

movs "Hello" %rdi
call puts @ plt

puts @ plt:

jump * GOT[N]

GOT



pointe aux puts
de la lib.

plt:

push GOT[1]

Jump * GOT[0]

puts @ plt:

jump * GOT[N]

Push N

Jump plt0

main:

movs "Hello" %rdi
call puts @ plt



Syscall:

#include <asm/unistd.h> contient tous les numéros de syscall.

Paramètres: RDI, RSI, RDX, R10, R8, R9

Valeur retour: RAX

Erreur: on met -ERRNO dans RAX.

Prace(Pid, Req, void*, void*) → API debugger

Les syscall renvoient des valeurs positives si ça s'est bien passé.

Syscall → RCX, R11 → pour stocker d'où il a été appellé il ne le met pas sur la stack mais dans ces registres.

injecter de l'assemblage dans le C: arm("btx")
asm("int3") → instruction assembleur pour faire un break point.

Donner des paramètres:

asm("mov %0, %1%rdi": : "r"(fd));

long close(int fd){

asm("mov %0, %1%rdi": : "r"(fd));

Variable 0 obligé de double % pour dire c'est pas une variable.

int rc = 3
asm("syscall": : "a"(rc) : "D"(fd): rax, r11)

(input)

Autre racineci
 int rc = 3
`asm(" syscall ":"=& a"(rc) : "D"(fd) : "rax", "rdi");`
asm volatile (...)

↳ pour dire on veut le mettre la
 sans intervention et modifier la compilation.

Ld.so

PHDR

↳ INTERP → .interp "/lib/ld.so"

PHDR

DYNAMIC → dynamic reld off - d pour les officher

LOAD

LOAD

↳{targ, val}

DT_NEEDED → renvoie vers les
 lib dynamique

DT_SYMTAB

DT_STRTAB

DT_PLTREL } tab de reloc

DT_RELAT

DT_DEBUG = 0



on remonte pour recuperer l'auxv

AUXV

↳ PHDR
 ↳ ENTRY

Pour afficher l'Auxv

on met LD_SHOW_AUXV=1
 dans le programme

Si ld.so voit DT_Debug il le fait pointer
 sur la struct r_debug.

Street & debug {

link-map

& bkh

& state → { constant, ADD, DEL }