

# SYSTEMES, MICROPROCESSEURS, ARCHITECTURE 1

- x86 (x86-64)
- ELF
- Compil toolchain
- Mémoire
- Assembleur

2 syntaxes en x86 Intel et gcc → celle utilisée par Gabi

Réf : \* Intel Software Dev. Manual (SDM)  
I Basic Archi II Ref instruct III \$

↳ Chap 3 et 5

exec ↳ résumé jeu d'instruct'

très utile, bien de l'avoir sous la main

\* Manuel de l'assembleur : GNU AS (gas manual) 'info as'

## Registers

RIP (l'adresse du pgm)

RSP, RBP (stack)

RDI, RSI  
RAX, RBX, RCX, RDX } quelques  
r8 → r15 }

eflags

Z, N, O, C (condit, comparaisons)

extend eax en rax: cdq



ds RAX  
if (a == 0)  
return 1;  
else  
return 0;

jmp %rax, \$0  
jne tota  
mov \$1, %rax  
ret  
mov \$0, %rax  
ret

jmp %rax, \$0  
jne tota  
mov \$1, %rax  
ret

instruct de branchemt (commence soit par J pour jump)

ici instruct taille variable → + rapide car condamne - de bande passante  
le + long est souvent c'est d'aller chercher des instruct°

taille instruction :	b	byte	1
	w	word	2
	l	long	4
	q		8

label : donner un nom à une adresse

Condition sur 2 instr.

Privilégiez un seul ret de 1 fab.

# ELF

Format d'executable.

Partie 2 : Biblios dynamiques.  
(shared object)  
ET\_DYN .so

Stocken des données de manière organisée.  
Fichiers objets, Core Dump, executable.  
(relocatable objects, o) ET\_CORE  
ET\_REL

À Static lib ! - ELF. Ce sont des archives (AR)

## Compilat\* Toolchain

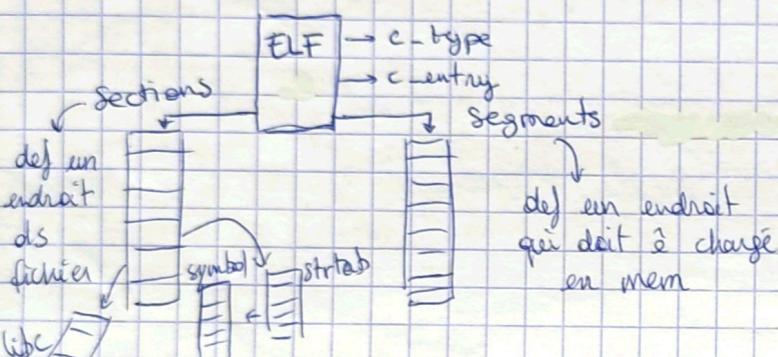
1. CPP → 2. CC → 3. As → 4. ld

↳ i  
(ou c)

↳ o

As → si ref / adresses non connues → vide ds.o (avec des 0)  
ld → patch les troués ds les o

→ lireelf.h



Pour manip ELF : nm · objdump · readelf · glibs · objcopy · ld / gas ; xxd,  
↳ binutils

Toolchain :

- binutils
- gcc
- libc
  - header
  - libc.so
  - ld.so (interpréteur → linkeur de libc.)

Métadonnées pour le linker : tue en haut de l'assembleur

:=  
mnémonique opérande

• file file info pour debug & error (uniquement à titre informatif)  
• global name ds la table de symbole → name devient global

• par défaut en C externe/global DS un ELF par défaut local/static

• type name, type ds table de symbole, def type de name  
type par défaut @object (table: @function)

valeur de retour d'une fonction : RAX

paramètres de fonction : Stack (car seule zone mémoire non globale)  
mais aussi par registre (64 → plein de reg.)

threads: Partagent les données mais font des choses différentes.  
Si param globaux, va ds les 2 en même temps ⇒ n'importe quoi

# SYSTÈMES, MICROPROCESSEURS, ARCHITECTURE 2

En 32 bits, stock param sur stack en ordre inverse.

Ainsi | param1  
param2  
:  
param n

↳ utile pour fonction variadiques  
(ex. printf)

## Calling conventions

ex: registres, ret ds RAX  
pile et ret ds RAX

⇒ Pour pouvoir communiquer entre ≠ pgms.

Paramètres: %rdi, %rsi, %rdx, %rcx, %r8, %r9

Ret value: %rax

Si +6 paramètres : sur la stack (→ fct + lentes)

Pour les floats → différent: %xmm0, %xmm1, ..., %xmm5

À l'origine rdi et rsi → stock addresses mem (spéc. sur string)  
dest ↴ src

leaq .LC0(%rip), %rdi → met la str passée en param de rdi

.rodata → toutes les trucs en READ ONLY

.section .text .Section → indique où on va le code.

. : adresse courante.

(%reg) → déréférencement

\* reg : appel de la fct des reg  
↳ ptr sur fct.

movzb : z ext° signe à 1 byte l : stock ds 2 opérande

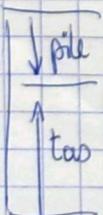
-n(%reg) déréférencement de reg -n

Architecture moderne : risk → jeu d'instruction simplifié : + d'opti  
⇒ Auj. majorité des assembleur génère par machine qui optimise  
x86 : vieux, avant compilateur optimisant

haut de la pile : rsp

pile fct à l'envers : addr hautes : bas de la pile

" basses : haut de " "



push et pop → pr manipuler la pile

call et ret → ressemble à push et pop?  
↳ utilisent la pile ⇒ call : push sur la pile  
ret : pop de la pile

Appel de fct : si push sur la pile, doit pop pour revenir à l'état précédent.

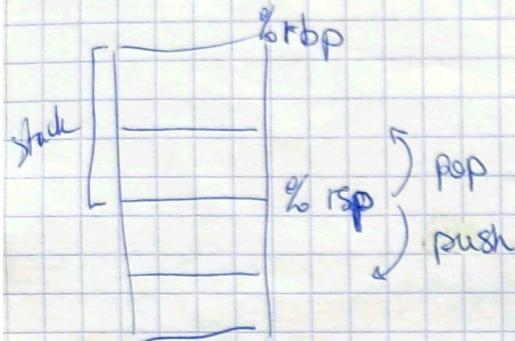
→ doit stocker taille

libérer tt l'espace d'un coup :

créer stack frame (espace qu'on se donne)

adresse bas de stack frame = rbp.

si rsp < rbp ⇒ on libère la stack frame  
et on revient à l'état précédent.



pop : incrémente rsp

push : décrémente rsp.

Début de fct : tjs la même chose et fin n'est pas la même chose

pushq %rbp

movq %rsp, %rbp

prologue

popq %rbp

ret

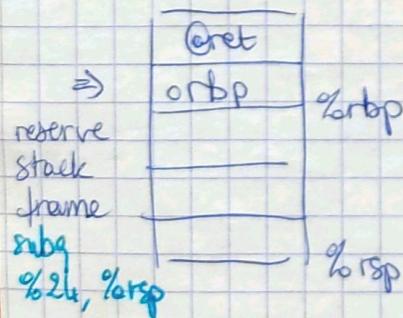
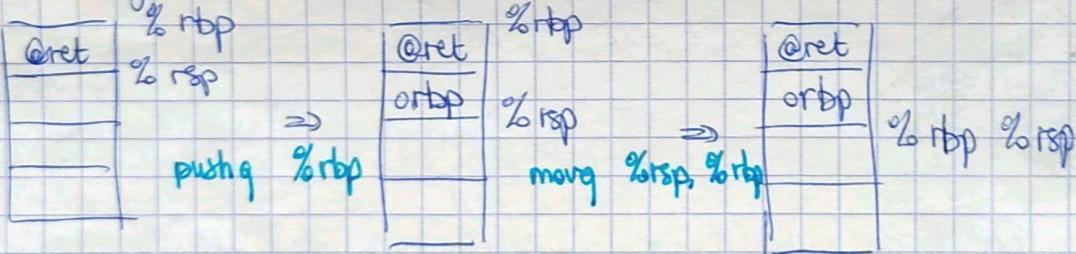
épilogue

creer stack frame

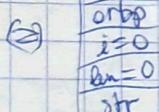
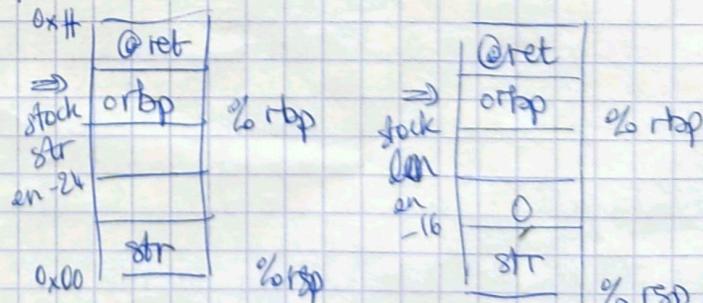
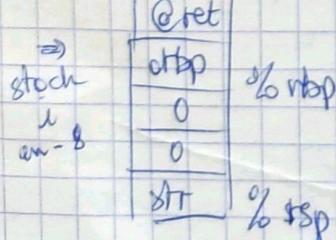
reviens à état initial

→ set bas de stack frame

Call my8tlen



rsp peut changer donc on regarde par rapport à rbp qui ne bouge pas



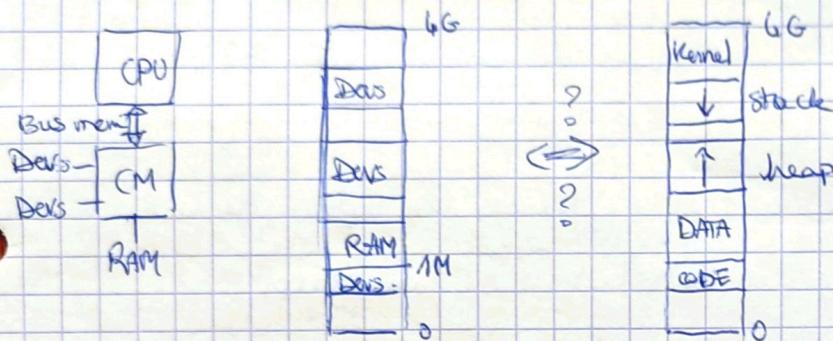
# SYSTEMES MICROPROCESSEURS, ARCHITECTURE 3

Fournir un env sain au pgm.

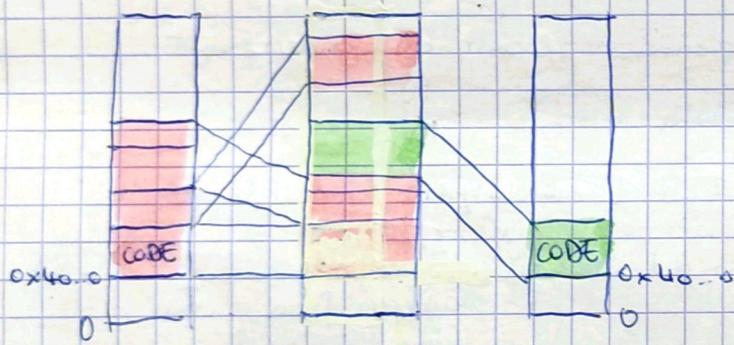
★ Espace d'adressage: rep. un range d'adresses dispo ou non.  
début - addr. alloués ou non - fin.

ex: esp. addr. mem. 0 zones 4K  $\rightarrow$  4G zone = page.

Pages : sur x86 la plupart du temps 4K.  
Disk bloc (512, 2K, 4K)  
HD CD HD records



P 2 pgm:



Plus droits d'accès:

- Read
- Write
- Exec
- System/User

Si  $\text{addr}^{32b}$  @ 4b AS 0  $\rightarrow 2^{32} = 4G$   
Page 4K 0  $\rightarrow 2^{12} = 4096$



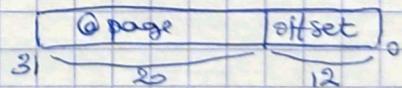
Pour faire correspondre addr virtuelles et physiques avec permissions:

```
struct {
    u32 physique_addr
    u12 flags
} pages[220]
```

Flags: présent, R, W, X, S

(2b.) { 32b = 6 bytes.  
(12b)

avec indices de tableau : n° de page

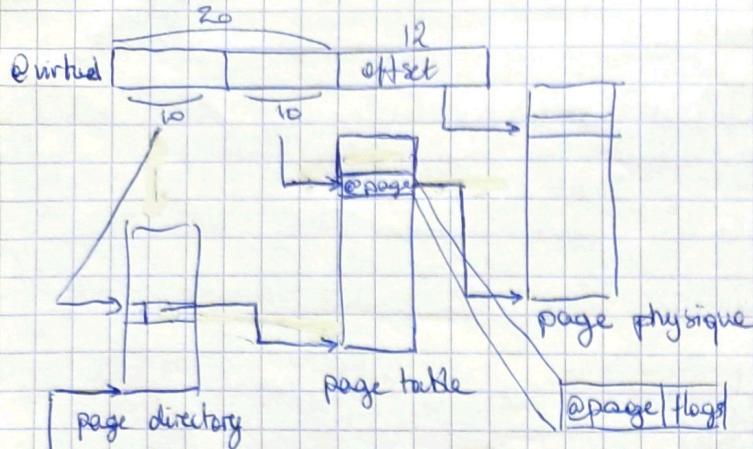


$\Rightarrow$  Bof Bof, allouer 4M contigus à chaque fois pr le pgm  
c'est moyen.

Appel à chaque instruc<sup>t</sup> + si instruc<sup>t</sup>  
faut un appel à la memoire  
 $\Rightarrow$  On veut un O(1), O(n) c'est déjà trop.

$\Rightarrow$  On va utiliser un BTee (arbre gen de recherche, nœud contient  
tableau de offset)  
design pr créer des index (base du données, filesystem)

Arbre : BTee de hauteur 2, nœud de 4K, 1 entrée = 40, 1K entrées



Plus besoin d'être contigüe.

+ seulement besoin de mapper les pages nécessaires.

20 bit : se balader dans BTee.

10 bit fort : page dir (niv. 1)  
10 bit faible : page table (niv. 2)

$\Rightarrow$  sys de paginat<sup>o</sup> x86 32 bits

[CR3] reg de contrôle : root du BTee

$\Rightarrow$  Pour passer d'un pgm à un autre : changer la valeur de CR3.

temps constant mais 3 déréf  $\Rightarrow$  pas suff : comment accélérer ?

cache : zone mem rapide, stock des trucs accédés récemment  
struct donnée + efficace (tab, tab de hash)  
petite taille, facile d'accès.  
Si plein : on enlève les données accédées récemment

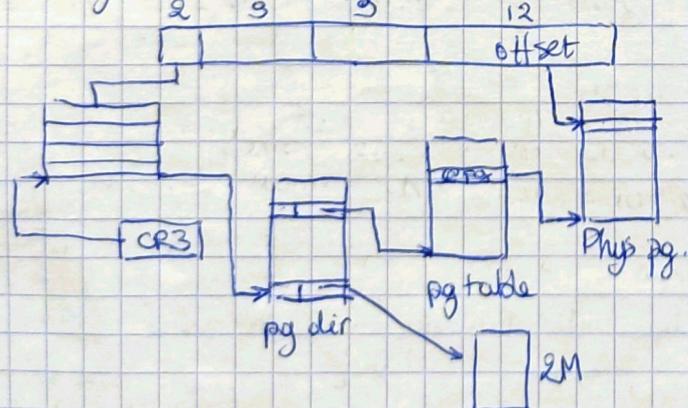
Vider cache : algo LRU (Least Recently Used)

$\Rightarrow$  Devant structure de BTee : Cache TLB (Translation Lookaside Buffer)

Qd lib mem  $\rightarrow$  sortir du cache  
Qd change flag  $\rightarrow$  update cache

Qd change CR3  $\rightarrow$  vide cache (flush)  
sauf si flag global (kernel).

Si + grosse adresse, - d'entrées :



ds pg dir : flag sur pg table ou pg.

en 64b : + de niveau, @+gde.

Note : la structure s'appelle PAE

Résumé : @phys @virt : map les 2. Structure compliquée pour le faire.

erreurs : si taper des pas mapper, si taper des mauvais flag (ou autres)  
 $\hookrightarrow$  rapporter l'erreur. Exceptions (may asynchrone de renvoyer l'erreur)  
les renvoyer un signal :  $\hookrightarrow$  changer de bout de code

Except<sup>o</sup>: PAGEFAULT  $\rightarrow$  accès zone non mappée  
#partial  $\rightarrow$  pas les bons droits (flag permissions)

1 pgm qui fait rien  $\sim$  40 pagefaults, ls : + 1000 pagefault 0.0  
 $\rightarrow$  Kernel map rien jusqu'à ce que ça pagefault  $\rightarrow$  map à ce moment là : map le + tard possible pgm de l'ordre

# SYSTEMES, MICROPROCESSEURS, ARCHITECTURE 4

mmap munmap mprotect  
(modif permission) mremap (linux only) agrandir map  
madvise (trucs chebus ??)

allocateurs mem : malloc free calloc realloc.

calloc : malloc(AxB)+ memset(0) (check overflow AxB)

v.1] realloc : malloc + memcpy + free (de manière débile, à changer après)

Best: allouer et libérer en temps constant ( $O(1)$ )

Advice: liste doublement chaînée  $\rightarrow$  libération en temps constant

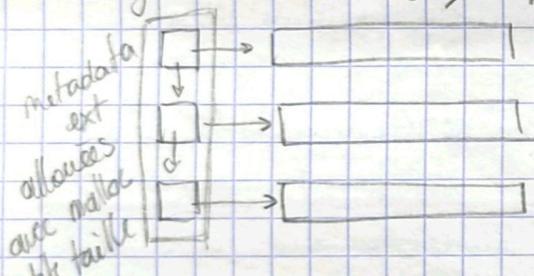


Bloc 75s à la fin de la page  
pr 1 cache  
pr free, lib à 0  $\rightarrow$  metadata  
table  $\Rightarrow$  odd freelist (ex:  $0x1234 \rightarrow 0x1000$ )  
 $\Rightarrow$  marche pr petites allocs ( $< 1$  page)

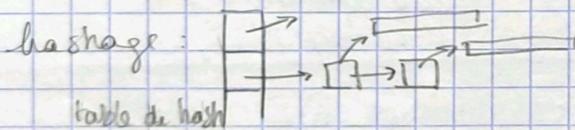
liste de cache

realloc + gd < bloc  $\rightarrow$  ok. realloc + petit  $\rightarrow$  overflow  
realloc + gd > bloc  $\rightarrow$  changer de cache

Pr gdes allocs ( $\geq 1$  page): metadata ext.  $\rightarrow$  allouer avec malloc petite taille



pr free : si @ alignée : gd  
sinon : petit  
(mais linéaire)  
 $\rightarrow$  search tree, hashage



hash : modulo (bits faibles)  
 $\hookrightarrow$  Internet: hasher un ptr  
 $\rightarrow$  temps constant amorti

pb init: allocations qui s'appellent l'un l'autre, en allouant tout par petits bouts pour avoir tout

malloc gdes tailles  $\rightarrow$  mremaps (uniquement sur linux)

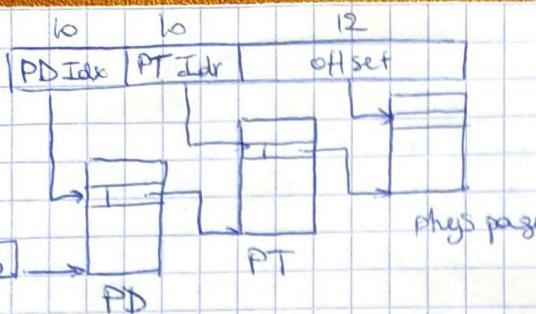
Pour debug: LD Preload ds le debugger, pas avant

A printf utilise malloc au debutt  $\rightarrow$  pas de debug au printf T.T

Par contre algo  $\rightarrow$  pourri niveau thread safety

best sur petit pgm  
utiliser l'trace, refaire bouts de  
code utiles pr + petits bouts  
relancer avec un petit OS (???)  $\rightarrow$  en fait non

$\Rightarrow$  mutex  
petit lock cache grand tout lock  
 $\Rightarrow$  variables thread locale  
un alloc par thread?  
 $\rightarrow$  mais struct globale  
avec petit code (mutex)



Des pd et pt :

@page	Flags
- present	- Global
- R/W (read / write)	
- S/U (System / User)	
- NX (64 b / PTE)	
- AVL (available)	
- Dirty → (ù que accès en écriture)	
- Access → (CPU écrit dedans)	
- Cache → (1 si pagewalk)	(Quelles zones ont été lues/utilisées par le pagewalk)

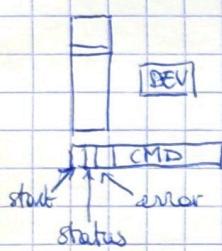
→ synchronisation du cache avec la zone

Politiques de cache (cache de CPU)

- Noe
- write-through (lecture ds le cache écriture ds mem principale)
- write-back (lecture & écriture ds le cache)

+ lent en écriture mais synchro si pls cache.

On veut un cache la plupart du temps SAUF pour les devices



Pas (sinon pour récup le DE bit de status c'est pas évident...)

(Dirty & Access) track quels fichiers sont utilisés ou pas.  
utile pour la synchro

Migrat° de VM : Clean dirty - Copy all - Recup only dirty → peu de data à copy  
→ Pause - longue - Pause - Copy dirty - Relance VM  
⇒ Très peu de perte de service.

- On demande Paging : alléger la pression mem en ne donnant la mêm tâche qu'à sollicitée.
- Swap
- mapping de fichier
- Shared memory
- CoW (copy on write (ex: fork))

Note: le kernel a une stack par address space

Note: user = pgm. set des humains

→ PAGE FAULT : exception (erreur du CPU C interruption)

jump au bout de code d'interruption et rentre à l'endroit qui a fait l'err

Stack [RIP (Instruction ptr), CS (segur sur le code, info dessus (Kernel/User?))]

ret sur et les 3 [RSP, SS]

[eflags] (modif qd exception) (error) (instruction)

enracode (type d'erreur) → Pst, U/S, R/W, I/D, ... (System) (data)

+ registre CR2 : fault address (l'adresse qui a merde)

Pour chaque tâche : struct task\_struct {

    struct mm\_struct \*mm; → ptr car possible que plusieurs thread partagent m même address space.

    struct mm\_struct {

        pgd  
        struct vm-area \*vmas;     list ordonnée de tous les mappings.

    };

Pour chaque tâche, bas de la stack : struct task\_struct

# SYSTÈME, MICROPROCESSEURS, ARCHITECTURE 5

Si pagefault parce que vraiment pas possible d'y accéder  
→ SEGV envoyé (sauf si RIP Kernel → oups du panic)

Oups : kill le thread Kernel et passe à un autre  
Panic : Arrête tout

Pas possible de revenir à l'instr prec. → instr taille variable

Sinon : !P → W

↳ map (perm, get-free-pg());

!P → R

↳ map (R0, zero-pfn)

W sur zone R0

↳ map (RW, copy(pg))

zero-pfn : page avec que des zéros, tjs la même

même  
code  
avec  
des  
particularités

shared memory ≈ file mapping

(Création d'un file cible sur la memShared)

Swap : plus de RAM, il faut en récup!  
→ piquer de la RAM à quelqu'un d'autre, garder ses données qu'il n'a pas utilisées.  
on prend les pages les moins récemment utilisées.  
on crée des shared par bcp de monde.  
possible de swap sur soi-même  
par contre : swap = lent!

Si !swap ou plus de RAM et plus de swap : on kill un truc.  
→oom killer : kill gros trucs qui sont pas lancés en fait  
(la plupart du temps, kill browser)  
de qui demande bcp de RAM, qui fait bcp ...

Compter le nbre de write mem :

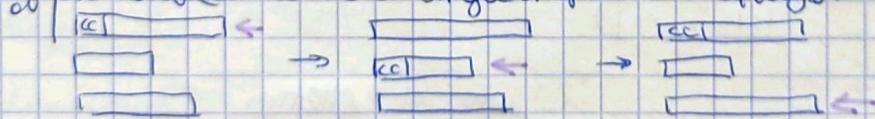
tjs R0. → Pagefault.

chaque write : R0 → RW ; Instr ; RW → R0.

Pr gr, breakpoint : int3 = 0xcc (interruption 3 (exception))

on met int3 au debut de l'instruction.

ou l'on demande de single step → efflags.



par contre : marche pas en multithread

Paril pr track des trucs : breakpoint sur des pages (PROT\_NONE)

Posix

↳ process ) Task  
↳ thread ) Task

FAS

- cwd → chdir()
- rest → chroot()

Task

- context
- memaire (AS)
- FDTable
- State
- 
- 

Metadata

PID	UID	EUID
TID	GID	EGID

## Fork

À si ya un fork, il y a un wait. (parent va lâcher au free, open w/close)

int execve (char \*filename, char \*\*argv, char \*\*\*envp)

ex: \$ls toto  
 ↳ chemin complet du pgm, dernier:0  
 ↳ environnement (var d'env.)  
 → execve("/bin/ls", ["ls", "toto", 0], environ)

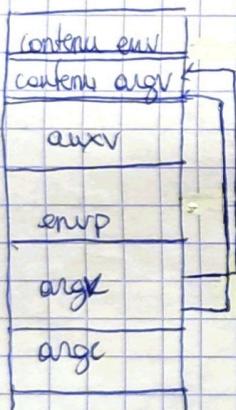
Parfois, envie d'isoler le pgm → envp = NULL.

Pour passer les m° var d'env → envp = environ.

var d'env passée dans fork, fils héritent de l'env du père.

recup env: getenv modif env: setenv

Passer les param au pgm:  
 qd execve, nouvelle stack



auxv : vecteur auxiliaire.

autres param

utile princ. pour ld.so (interpréteur de binaire, charge lib dyn.)

info aussi pr libc

afficher contenu auxv:

LD\_SHOW\_AUXV = 1

execvp ("ls", ["ls", "toto", 0])

↳ call à execve mais pas d'env et ... pas besoin de path complet

\$ VAR=a ls toto → changer env avec execvp.

↳ setenv ("VAR=a"), execvp ("ls", ["ls", "toto", 0]);

fd = socket()  
 bind(fd, addr)  
 listen(fd)  
 while(client = accept(fd))  
 read/write (client)  
 close(client)

→ 1 client à la fois  
 → si client dis rien  
 on attend  
 ⇒ NUL

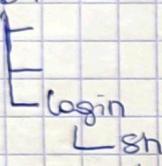
fd = socket()  
 bind(fd, addr)  
 listen(fd)  
 while(client = accept(fd))  
 fork()  
 read/write (client) avec des watchs ...  
 close(client)

# Système, microprocesseurs, Architecture 6

- select()
  - ↳ premiers qd les fd qui sont watched sont prêts à faire opér.
  - pas ouf qd bcp de fd, un pr chaque act<sup>e</sup> (read/write),  
reparcourir tableau à chaque fois pr savoir lequel est bon.
  - BREF : inutilisable.
- poll (struct pollfd \*)
  - ↳ fd, read/write/whatever, outjet
  - aussi obligé de parcourir en boucle, mais + utilisable que select  
watch linéaire → performance pas ouf qd bcp de fd.
- LINUX : epoll\_\*
- BSD : kqueue

## Signaux

★ PID1



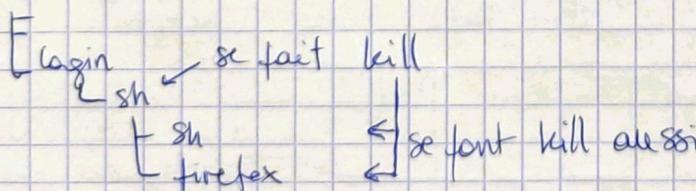
wait() renvoie un true et repasse la main  
sinon signal : SIGCHLD au père

int wait(&status, flag) attend en général et récup l'mort  
int waitpid(pid, &status, flag) attend un fils en particulier

```

void sigchld_handler(int)
{
    // sauve errno
    for(;;)
        {
            rc = wait(0, WNOHANG);
            if (rc < 0 && errno == NOCHILD)
                return;
        }
    // restaurer errno
}
  
```

★ PID1



exit explicitement

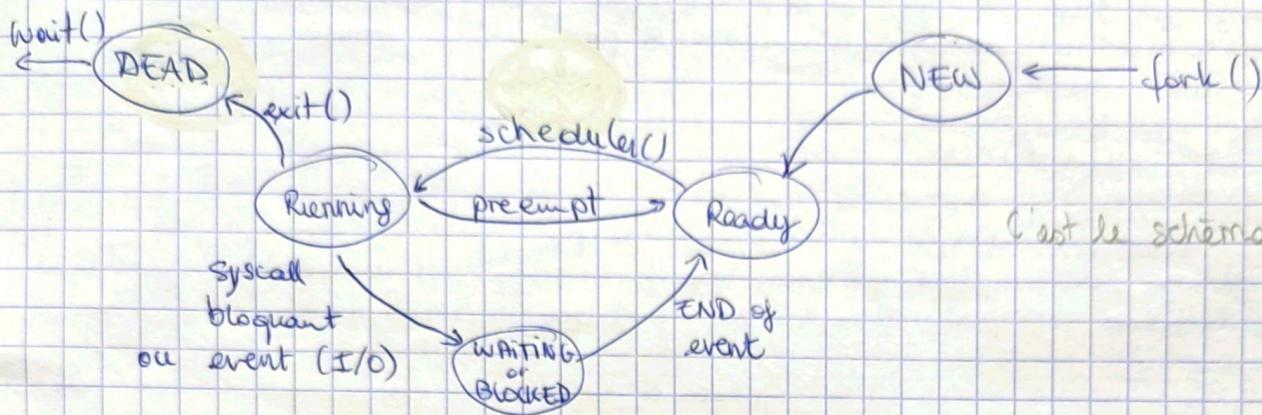
↳ pas kill, deviennent orphelins  
↳ se font rattachés à init  
⇒ deviennent daemon



## Gérer les tâches

Scheduler : ordre des tâches

⚠ Algo de scheduling : utiliser par plein de tâches ≠



C'est le schéma utiles !

Scheduling FIFO (sans preempt)

Scheduling Round-Robin - Quantum (timer)

Shell : attend bcp → très peu de temps en running  
(wait, fork, sleep)

mais on veut qu'il soit schedule asap.

→ il choisit par HS les progs interachg qui vont bcp.

CPU bound + I/O bound CPU Schedule - svt

I/O bound : - CPU + schedule

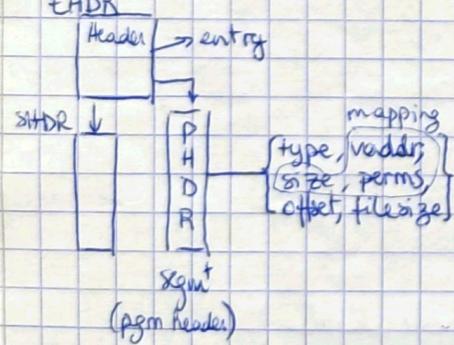
2 schedule 1s ≠ 1 schedule 2s

↳ flush tps & cache - prend du temps.

→ un prog met du tps à être efficace qd scheduled

waiting time : tps passé en READY

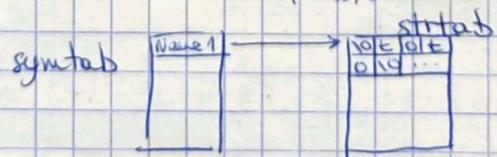
**ELF** (man elf & /usr/include/elf.h et .../link.h)



type: LOAD (et d'autres)

ex: 1° o int toto = 12;  
2° o int foo(){  
  return toto; }  
3° o int main(){  
  return foo(); }

toto: long 12  
foo: moveq toto,%eax  
ret  
main: call foo  
ret



si symbole substr suffixe d'un autre symbole, peut pointer sur substr.

DS 3° o [ ] [ @ ]

mais on a pas l'adresse  
autre symtab foo pr dire  
qu'on en aura besoin au link

Qd -g de debug : sects avec [fichier, ligne nb de code]  
[ang des fcts, info types] [call-frame] → DWARF

# SYSTEME, MICROPROCESSEUR, ARCHITECTURE 7

## LIB DYN

→ Fix un bug ds la lib sans avoir à recompiler les progs qui l'utilise

→ remplace des libs par d'autres qui font un peu  $\neq^{\text{at}}$

ex de remplacement: OpenGL, blas, opak

Pas besoin de H ds la lib, pas envie de H patch pour call d lib.

Pas envie non plus de patch à chaque call.

→ patch au 1<sup>o</sup> call puis grande l'entrée du patch.

ex: int main() {  
 puts("hi man");  
}

on ne peut pas partager.

⇒ pour pouvoir partager

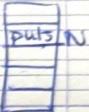
main:

mov "hi man", %rdi  
call puts@plt

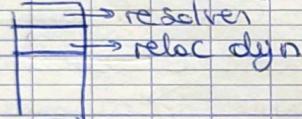
le but de patch  
de dyn lib.

puts@plt:

jmp puts ← seul endroit  
où on a besoin  
de patch.  
GOT



resolver dyn



⇒ GOT



plt0:

push GOT[1]  
jmp \*GOT[0]

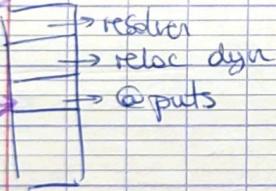
puts@plt:

jmp \*GOT[N]  
push \$N  
jmp plt0

main:

mov "hi man", %rdi  
call puts@plt  
ret

avec GOT



ex: plt0

push GOT[1]  
jmp \*GOT[0]

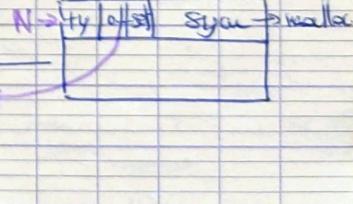
malloc@plt:

jmp \*GOT[N]  
push \$N  
jmp plt0

main:

mov \$1b, %rdi  
call malloc@plt  
ret

GOT



LD-PRELOAD: hijacke les libs : place les libs au debut

→ ld.so trouvera cette lib en cherchant le symbole.

on peut mettre plus de libs : LD-PRELOAD = liba:libb:...:libn

GOT: Global Offset Table

## Syscalls

man 2 syscall → détails sur ts les syscalls et équivalent.

Assemblage & Syscall. Pr passer le n° du syscall → RAX  
#include <asm/unistd.h> → contient ts les n° de syscall

NR\_Close 3

Param : RDI RSI RDX R10 RS R9 - Tlè que fait sauf R10 (RXX pr fact)

return : RAX errno : si erreur, ret -ERRNO → pas de syscall à ret neg.

syscall invalide RCX et R11 pr stocker là où (pb avec par ex mmap, etc)

ce n'est pas au lieu de stocker bien la stack ou

Moment du passage user/kernel.

Injecter dé d'adresse de la C : pb → pas toucher aux reg des core solo  
passer des valeurs du C à l'assem.

mot clé : asm(str assemblage); génère .S avec dedans la str assemblage

asm("int3"); interrupt 3 (breakpoint) → "register" pr ts les reg

en général. asm("string avec %"); output : input : dotbber); stock ce qui est utilisé et à redonner

ex : long close (int fd){ → exope du % → 0<sup>ème</sup> true après l'"ouvrir" (seul en input)  
asm("mov %%0, %%rdi"; : "r"(fd)); } "r"(var) met var ds whatever reg.

→ long close (int fd){

int rc;

} asm("syscall")

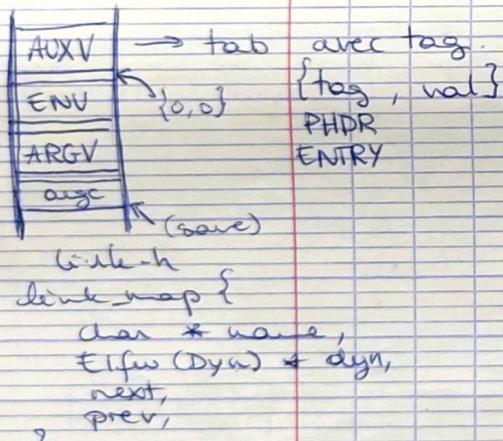
" : "=a"(rc) : "D"(fd), "a"(3) : "rc", "rdi");

Volatile : ne pas redonner le code (asm volatile)

## ld.so

lien dyn / interpréter. → trouve lib dyn et les charge en mem. → readelf -d pr voir

PHDR types : doset(dyld) → INTERP PHDR DYNAMIC LOAD LOAD  
.interp "/lib/ld.so" ← pr addr du mapping → dynamic struct avec {tag, val}



Pour afficher AUXV, nom de lib  
"LD\_SHOW\_AUXV=1"  
ds l'env.

tab de reloc ←

DT\_NEEDED "libc.so.6"  
DT\_SYMTAB  
DT\_STRTAB  
DT\_PLTGOT  
DT\_RELATI  
et d'autres.  
DT\_DEBUG  
struct pr debig  
(genre sp du futur)

d'ee utilise  
l'autre et  
inversent

Grosses interact' entre  
libdl et ld.so  
libc et ld.so  
gdb et ld.so  
(debugger)

os. shage state  
state = ADD  
ret  
ret

si debug  
state = ADD  
int 3  
ret.

Partiel : avec document → faire des jolies fiches

- quatrième cours
- passage C - assemblage

# OPERATING-SYSTEM - 1

strace → trace des syscalls.

- What
- mapping de fichier

Qu'est ce qu'un système d'exploitation?

Netter à disposer aux autres prog. via des abstract°

↳ + simple + générique

Fournir un bain et simple aux prog.

## - Kernel

Code privilégié qui s'occupe du "management" des ressources

→ accès aux RSC. Il restreint l'accès pour le distribuer correctement

RSC : CPU, mem, accès HW

## - Processeur et ordinateur

ISA (Instruction set Archi)

privilégiées (use reg. control)

non priv

→ Registres (zones mem très limitées en taille & ab pr faire calcul (exec instr.))

↳ généraux (calcul, use by everyone)

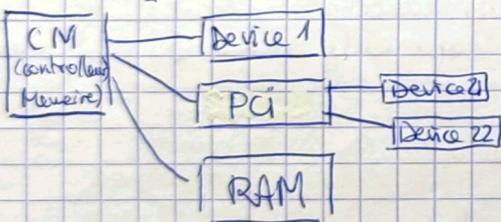
↳ Contrôle (config processeur & machine en elle même)

Bus d'adresse ; Bus de données ; Bus Mem

→ 2 modes

↳ User (non instr° priv)

↳ supervisor (instr° priv.) (a le droit à tout)



Mécanisme in Kernel, policy in userland

## - Syscalls

↳ API Kernel. || API = Application programming interface

↳ ce qui se fait des choses

Contexte switch → sigle état process actuel, mode kernel puis restore état (modif ret. val)

Comment ça marche derrière un ordinateur?

Kernel sur media de stockage. Idem pour rootfs

Code pour lancer le kernel qd ordi démarre

Device

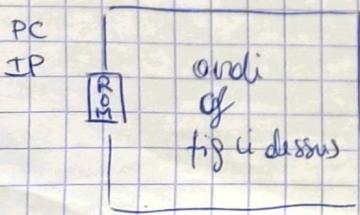
PC (prog counter)

IP (instruction ptr)

↳ EIP / RIP

IP pointe sur code Firmware pr lancer ordi/device (BIOS UEFI)

→ mettre la machine ds état "sain" pour arrivée du Kernel



Pour démonter :

→ Firmware (→ Bootloader) prend un bloc, le met en memo et hante

→ Kernel

init<sup>o</sup> (tinit de journal, charge plein de trucs)

mount (mount (2))

map nouveau disque sur arborescence de systèmes

→ /sbin/init

tinit l'init<sup>o</sup> rc (BSD) sysVinit (Linux) upstart (Ubuntu) systemd (?)  
gros paquet de script shell

multiple run level.

Run level | Target

- 0 poweroff.target
- 1 rescue.target
- 2,3,4 multi-user.target
- 5 graphical.target
- 6 reboot.target

Système de dépendance entre les init

- sleep while
- makefile

Systemd.

/\* NTP : network time protocol \*/ /\* DNS : domain → IP \*/

Bcp de choses qui tournent de base sur une machine

authentification : "je suis login-x" autorisation/permis : "il a le droit de"

## Résumé

Firmware lance kernel qui mount rootfs qui lance /sbin/init

Après ya pleins de trucs à faire et c'est la merde mais c'est pas le mem

## Mapping de fichier

iso = file \$, read oules, img d'un disk

```
for (shell->cuds[i].fd[i], i=0; i<shell->cuds[i].size; i++) {  
    if (fstat(shell->cuds[i].fd[i], &st) == -1) {  
        perror("fstat");  
        exit(1);  
    }  
    if (st.st_size < 0) {  
        perror("st.st_size");  
        exit(1);  
    }  
    if (st.st_size > 0) {  
        shell->iso[i].size = st.st_size;  
        shell->iso[i].fd = shell->cuds[i].fd[i];  
        shell->iso[i].offset = 0;  
    }  
}
```

struct file dir  
inode : fichier ds mem.  
offset

si read, reset offset. mais preadv

## ⇒ mmap (2)

(es)

tjs pas yet  
NULL

```
struct stat stat;  
fstat(fd, &stat)
```

void \* ptr = mmap(NULL, stat.st\_size, PROT\_READ, flags, fd, 0); ] tester error

MAP\_FAILED → ??

et tjs munmap(ptr, stat.st\_size)

char \* sd = (char \*)ptr + 0x8000 + 1; (↑ pas coding style → cast exp)

Si char non null term → ("%.\*s", size, sd) (ds printf)

struct iso\_file \* voldesc \* super → std identifier

open  
mmap  
close  
# whatever over le file  
munmap

struct iso\_file {

    struct iso\_file \* voldesc; //? (on trace ds le genre)