Notes

> Raphaël Oberdisse RT2APP

Rappel de l'assembleur vu en première année :
UC qui pilote l'ALU.

On avait fait de l'assembleur basique sur de l'ARM (téléphone mobile / RPI)
Nos ordinateurs actuels : x86/64 (x86 >> 8086 à l'époque)

On écrit dans la RAM :
Les variables
Les données
Le programme, écrit sous forme directe d'instruction ("tiens, une addition")
Unité de Controle = lit le code et configure le processeur pour faire le calcul / l'opération demandée.

Un registre spécial qui s'appelle le registre d'instructions : "prochaine instruction à tel adresse", avec un compteur.
registre nommé FLAGS : chaque bit est un petit drapeau (ex : un drapeau qui indique une division par 0), l'ALU met a jour l'état de la dernière opération en levant x ou y drapeau.

tout les registres sont sur 64 bits

intel : 18 registres généraux/principaux + une 60aine de registre spécialisés.

Quand on voulait sauter a une autre partie du code, on avait une fonction qui s'appelait CALL/JUMP (en fonction de la version de l'assembleur utilisé), en prenant une adresse et en la stockant dans le registre d'instruction.

Plus rapide de passer par les registres
top horloge : en une seconde
accès au registre : une seconde et demi
accéder à la ram : environ 3 minutes (et encore, grâce aux caches L2 L3)

grande famille intel : intel + amd
histoire :
8086 et 186, dans un univers de 16 bits : on adresse 2^{16-1} (un bon gros méga de RAM)
ensuite rupture : on passe au 286 : proco en 32 bits. grosse inovation : on rajoute un composant entre le processeur et la RAM : le MMU
elle traduit les adresses pour que le programme se sente "seul", les process sont isolés dans leur mémoire virtuelle.
particularité du 286 : introduire le MMU. Avant ça, les processus n'étaient pas isolés et pouvaient écrire sur la mémoire d'un autre.

386,386, les séries K6 K7, les pentiums (fameuse erreur calcul 4ème digit)
puis apparait AMD à la même époque.
puis on passe au 64 bits.

amd : AMD64 | intel : ilanium (flop de l'année)
suite au flop, intel à racheté l'architecture amd très cher.

quand on boot : **real mode** (le bios vient de nous donner la main)
les bios actuels : 16 bits.
puis il avance, il passe en 32 bits et donne la main au bootloader.
Le bios bascule en **protected mode** : la MMU s'active, on est en 32 bits.
Grub démarre à ce moment là.
le bios a déjà fait le café, on sait ou sont les disques durs (avec le MasterBootRecord MBR).

Puis on démarre le kernel : on passe en 64 bits, le **long mode**
la première application se démarre : PID 1 (systemd généralement)

Maintenant, UEFI :
toute la détection du matériel est unifiée.
ça boot à peu près de la même maniere : c'est plus le bios classique qui démarre, c'est l'uefi (qui possède un bout constructeur sur la carte mère) qui démarre. (ce qu'on appelle avec un abus de language le bios UEFI). Puis il vérifie aussi les disques mais plus de MBR, on passe au GPT. cherche une partition <32mo en VFAT (FAT32).
windows manager : bootloader comme grub.

au niveau du disque dur :
une zone au départ : MBR (analysé en héxa en RT1)
dedans, il y a une entête qui décrit globalement le disque (tant de cylindre, de secteurs etc)
puis 4 entrées qui correspondent aux partitions
et ensuite : une zone ou l'on pouvait écrire un programme : zone du grub (grub-legacy) dans les 540 et quelques Ko de places disponibles.

Puis arrivent les 4 partitions (pas plus, les partitions primaires).
Mais on peut en avoir plus : les partitions étendues : on va créer en tête d'une des partitions un mini MBR (eibr) qui va définir des sous partitions.

encore le choix entre bios et UEFI (voix moderne)
disque uefi : on a une partition MBR qui dit "il y a une grosse partition" (qui correspond a tout le disque) et dans cette grosse partition, on a une entête "nouvelle" MBR.
Et après l'entête GPT, on retrouve les partitions qu'on veut, aussi grosse qu'on veut.
quand on est en GPT : une partition va stocker le bootloader. (et ne pas se retrouver prisonnier du MBR)

autre différence : au niveau de l'exécution :
ancien bootlaoder : 16 bits, donc pas de couleur.
nouveau bootloader : 32 ou 64 bits donc des couleurs / images / souris présente au démarrage.

intel : faux CISC
parce que vrai CISC
code machine : microcode qui transformer CISC en RISC.

En deuxième année :
apprendre à bien se servir d'un debugger : car apprendre a programmer = apprendre à voir comment s'éxecute un programme
deuxio : pour la cybersécurité, savoir comment fonctionne la machine.
CVE : alerte sur une vulnérabilité, écrit en C (publié par quelqu'un)
python bien plus compliqué que le C (en terme de définition)
voir comment est traduit le C en assembleur et vice versa

paravirtualisation : le système est conscient qu'il est virtualisé, dans une VM
virtualisation forte : on virtualise tout le système, tout le matériel.

commandes :
gcc -o exectuable fichier.c
size executable

text : instruction
data : varaible globale et intialisée
bss : variable globable non intialisée

en rajoutant int K=12; le bss passe de 8 à 4, et le data passe de 600 a 604. (int = 32 bits soit 4 octets).
effet du include : il se reserve une page sans même l'occuper entièrement. donc rajouter int K; le bss n'augmente pas.

ELF >> Executable Linux Format/Files.
loader (dans le kernel) lit l'ELF et décide de combien allouer a chaque zone (.bss, .text,)
puis il prend la première ligne du code, le met dans le registre d'instruction et le programme se lance.
mmu permet au programme d'accéder à la même zone du kernel (des économies !)
segfault : segment mémoire qui n'est pas mappé
MMU respecte électroniquement des DROITS : RWX
éviter le sigterm (kill -9).

la zone de .text est généralement mappée en rx. il ne peut pas se modifier de lui même.
la zone de.data est en rw
la zone de .bss est en rw
depuis les années 2000, on a retiré les droits d'executable à la pile.

backup du tableau de la MMU.
processus zombie le processus père n'a pas conscience que le processus fils est terminé.

objdump
(rajouter un -g au gcc) >> intêret principal du -g : on a en commentaire le C.
-S : source
| less
utiliser le / pour rechercher

GDB

boucle for :

```c
#include <stdio.h>

int K;
int i;
int main() {
    for(i=0;i<10;i++) {
        printf("Hello\n");
    }
    return 0;

    }
}
```

gdb ./executable

run : éxecute le programme
placer un breakpoint =

- b main (s'arrête à la fonction main)
- b main.c:14 (s'arrête à la ligne 14)
- delete breakpoints
- info breakpoints
  catch syscall 1

pour activer/désactiver un breakpoint : enable disable breakpointnumber

pour passer au pas suivant, "n" (next)
pour reprendre l'exécution ou il en était : "c" (continue)
s : montre les détails et rentre dans la fonction. (step)

print i >> afficher les variables (la variable i)
print &i >> afficher l'adresse de i
display i >> à chaque fois que le debugger va se mettre en pause, il va remonter i (sans qu'on le demande)
delete display "numéro du display"
delete display = delete tout les displays
bt = backtrack (on a mis en pause le soucis)
on voit que coucou est appelé par main.
catch syscall x = mettre un breakpoint a un syscall nommé

```c
#include <stdio.h>

void coucou(int j) {
    printf("Coucou %d \n", j);
}

int main() {
    int i;
    for(i=0;i<10;i++) {
        printf("Hello\n");
        coucou(i);
    }
    return 0;
}
```

frame 0 : dans la fonction coucou
frame 1 : on change de réferentiel pour aller dans le main et vérifier la valeur de i (par exemple)

---

installer gef pour gdb
https://gef.readthedocs.io/en/master/

---

plusieurs registres :

vmmap : permet d'afficher l'état du MMU.

Cours de vendredi :
processeur : composé d'une ALU, piloté par l'UC. autour de l'ALU se trouve des registres généraux, ainsi qu'un registre d'instructions.
un process tourne en ram virtuelle, traduit par la MMU dans la vraie RAM/
les zones mémoires sont mappés par le MMU.
l'étât mmu est backup, on passe au prochain process
le processus ne se rend pas compte qu'il est arrêté

deux types d'OS :
os pré-emptif et non pré-emptif
os pré-emptif : le process sait qu'il n'est pas seul au monde et qu'il doit rend la main à l'OS : OS des années 80 (n'existe plus)

os non pré-emptif : le process est isolé.

Le CPU possède deux modes de fonctionnement : quand il démarre, le processus refait l'histoire.
il commence en realmode (16bits sans MMU), passe en protected mode (32bits avec MMU) et ensuite en long mode (64bits avec MMU).
Il existe aussi des sous modes :

- Ring 0 : droit à toutes les instructions de l'assembleur (du processeur) : kernel : kernel space
- Ring 1 :
- Ring 2 :
- Ring 3 : limité, certaines instructions sont interdites. (ex : l'accès au matériel, la configuration de la MMU). Droits réduits (safe mode) : les processus : user space

On va alterner entre les rings
notion d'interrumption ou IRQ : interrumption matérielle.
qui va déclencher un comportement un peu spécial : l'instruction va se mettre en pause, et déclenche la fonction IRQn (IRQ 1 si le1 est précisé, etc), une fois qu'il a fini son IRQn, il reprend son instruction en pause.
existe 64 IRQ sur un PC.
exemple du dd : il va récupérer des données sur le dd, qui prend du temps. lors que dd répond, IRQ se déclenche, l'IRQ du dd se déclenche et met en pause l'instruction atm.

intêret de l'IRQ : un certain nombre et chacune associée a un certaine action
quand je déclenche une IRQ : automatiquement le processeur repasse en ring 0

au tout départ :
on a le boot, qui se lance. puis on exécute notre kernel qui lui s'éxecute dans le ring 0
le kernel va programmer un timer : 1ms.
ensuite il charge en mémoire le process 1
à ce moment là, le kernel va faire une bascule en ring 3 et le process s'éxecute (avant c'était le kernel en ring 0)
après le timer 1ms : il va déclencher une IRQ, on repasse en ring 0
et donc le kernel reprend son execution, décharger le process de la RAM
une fois le déchargement du process N, il charge le process N+1 et rebascule en ring 3.
et c'est le process n+1 qui prend la suite.

on change d'application a une fréquence de 1KHz (fréquence de changement de process)

= ordonnencement : passage d'un process à l'autre.

macos, linux et windows ne se servent pas des rings 1&2

le process ne fait pour l'instant pas d'accès matériel :
à chaque fois que le process a besoin d'un service : fonction particulière **syscall**
syscall va interrompre mon programme : déclencher un IRQ particulier, passe en ring 0, et reviens enn ring 3
sans arrêt des interruptions, pour n'importe quoi (ouvrir un document, recevoir un paquet réseau, etc)
processus d'interruption peuvent être interrompu par un autre processus d'interruption

syscall : rax, rbx, rcx, rdx, rex, 6, 7 8
syscall qui existe pour les entrées sorties stdin stdout.
syscall qui existe pour ouvrir un fichier
tout les syscall ont un numéro, il y a une table des syscalls

syscall = instruction

outils sous linux : strace
ex : strace /bin/ls

rbx = 1 : file descriptor : le plus connu, 1 = stdout
syscall write

TD Vendredi
différent noyau : android, Mac osX, windows, linux freebsd, netbsd

histoire :
à l'époque 1970, AT&T, bosse sur multix.
trois ingénieurs bossent dessus : brian kerninghan, denis Ritchie, Kenthompson
et créeent unix pour troller multix.
il trouve un pdp11
trouve 15 jours pour coder unix en assembleur.
unix devient plus populaire que multix.
l'inventeur du pipe, du bash, du shebang,
en 1984 : unix5 (grande particularité : ils inventent le C pour éviter de faire de l'assembleur, et écrivent donc Unix5 en C).
université de Berkley : recrute toute l'équipe, leur donne un budget illimité ; le début de la version Berkley d'Unix.
l'université de Berkley fait payer la licence Unix.
particularité : une lience = valable a vie, sur autant de poste que l'on veut.

un étudiant (linux torvald) : va faire un clone d'Unix pour l'adapter au PC (moins puissant à l'époque) et c'est la création de Linux.
le clonage des outils s'apelle : GNU.

Richard Stallamn : il veut que tout soit gratuit (vu comme un extrême gauchiste aux USA), licence, GNU, tout soit gratuit.

Unix va finalement devenir libre (gratuit), et deux kernels en naissent : FreeBSD & NetBSD (qui sont deux kernels différents, même si très ressemblant)

les outils de GNU sont complètement compilable sur freeBSD & netBSD car code open source en C.

atm : très très récemment, linux a encore divergé : systemd.
systemd = linux non compatible : ne marche que sous linux.
problème de systemd : ça doit marcher sur mon portable (mais les serveurs ne sont pas des portables, donc pas forcément compatible pour de multiples raisons)

compatible norme posix

https://filippo.io/linux-syscall-table/
https://syscalls64.paolostivanin.com/

driver : une ou plusieurs fonctions écritent en C dans notre kernel qui répond aux IRQ

/dev/nodes

un certain nombre de syscall classique :
étape quand on ouvre un fichier :

- open()
- read() ou write()
- un chattr (changer les droits)
- close()

exec

^^^ tout ça sont des syscalls

mmap : memory map : je prend le fichier et je le recopie en RAM.

un process ne peut pas directement executer quelque chose

fork() >> consiste a se cloner moi même
unsahre() >> annuler tout les memory map qui me concerne

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
```

```c
int main() {
	pid_t A;
	A=fork();
	if(A==0)
		printf("Pouet\n");
	else
		printf("Meuh\n");

	return 0;
}
~
```

```
spleenftw in ~/Documents/test | 11:02:00 | | took 15s ➜ gcc -o seb seb.c
spleenftw in ~/Documents/test | 11:02:01 | ➜ ./seb
Meuh
Pouet
```

dans le fils, fork renvoie 0, donc print d'abord Meuh.
dans le père, le numéro du PID est renvoyé, et donc print ensuite Pouet.

fork : je crée un process (tout est indépendant)
thread : je suis dans le même process, je crée juste un fil dans le même process. (les variables globales sont partagées)

```
spleenftw in ~/Documents/test | 11:20:46 | ➜ strace ./seb
execve("./seb", ["./seb"], 0x7ffef4f3b680 /* 43 vars */) = 0
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffc50d4a5e0) = -1 EINVAL (Argument
invalide)
brk(NULL)                               = 0xd0f000
brk(0xd0fdc0)                           = 0xd0fdc0
arch_prctl(ARCH_SET_FS, 0xd0f3c0)       = 0
set_tid_address(0xd0f690)               = 132621
set_robust_list(0xd0f6a0, 24)           = 0
uname({sysname="Linux", nodename="pop-os", ...}) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024,
rlim_max=RLIM64_INFINITY}) = 0
readlink("/proc/self/exe", "/home/spleenftw/Documents/test/s"..., 4096) = 34
getrandom("\xd9\x0f\xaf\x87\x3d\x58\xa2\xd7", 8, GRND_NONBLOCK) = 8
brk(0xd30dc0)                           = 0xd30dc0
brk(0xd31000)                           = 0xd31000
mprotect(0x4c0000, 16384, PROT_READ)    = 0
clone(child_stack=NULL,
flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
child_tidptr=0xd0f690) = 132622
```

```
newfstatat(1, "", {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0x2), ...},
AT_EMPTY_PATH) = 0
Pouet
write(1, "Meuh\n", 5Meuh
)                   = 5
exit_group(0)                           = ?
+++ exited with 0 +++
```

On voit donc bien le fork avec le clone()
on ne voit pas le syscall du Pouet parce qu'on suit le père et non le fils
(et c'est pour ça qu'on voit bien le syscall du Meuh)

on peut suivre le fils avec un -f

```
spleenftw in ~/Documents/test | 11:24:59 | ➜ strace -f ./seb
execve("./seb", ["./seb"], 0x7ffc1e4b3cd8 /* 43 vars */) = 0
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffcbb24fc00) = -1 EINVAL (Argument
invalide)
brk(NULL)                               = 0x248d000
brk(0x248ddc0)                          = 0x248ddc0
arch_prctl(ARCH_SET_FS, 0x248d3c0)      = 0
set_tid_address(0x248d690)              = 136922
set_robust_list(0x248d6a0, 24)          = 0
uname({sysname="Linux", nodename="pop-os", ...}) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024,
rlim_max=RLIM64_INFINITY}) = 0
readlink("/proc/self/exe", "/home/spleenftw/Documents/test/s"..., 4096) = 34
getrandom("\x70\xca\x65\x92\x98\x76\xcb\x67", 8, GRND_NONBLOCK) = 8
brk(0x24aedc0)                          = 0x24aedc0
brk(0x24af000)                          = 0x24af000
mprotect(0x4c0000, 16384, PROT_READ)    = 0
clone(child_stack=NULL,
flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLDstrace: Process 136923
attached
, child_tidptr=0x248d690) = 136923
[pid 136923] set_robust_list(0x248d6a0, 24 <unfinished ...>
[pid 136922] newfstatat(1, "",  <unfinished ...>
[pid 136923] <... set_robust_list resumed>) = 0
[pid 136922] <... newfstatat resumed>{st_mode=S_IFCHR|0620,
st_rdev=makedev(0x88, 0), ...}, AT_EMPTY_PATH) = 0
[pid 136922] write(1, "Meuh\n", 5Meuh
 <unfinished ...>
[pid 136923] newfstatat(1, "",  <unfinished ...>
[pid 136922] <... write resumed>)       = 5
```

```
[pid 136923] <... newfstatat resumed>{st_mode=S_IFCHR|0620,
st_rdev=makedev(0x88, 0), ...}, AT_EMPTY_PATH) = 0
[pid 136922] exit_group(0 <unfinished ...>
[pid 136923] write(1, "Pouet\n", 6 <unfinished ...>
Pouet
[pid 136922] <... exit_group resumed>)  = ?
[pid 136923] <... write resumed>)       = 6
[pid 136923] exit_group(0)              = ?
[pid 136922] +++ exited with 0 +++
+++ exited with 0 +++
```

On voit bien le write() du Pouet du fils

TP du lundi 16/05 :

Comment marche le printf :

On prend un programme simple qui affiche Hello World.

on utilise divers outils : objdump -S, strace, gdb avec gef.
on cherche le syscall qui s'occupe du printf : write.
On place un breakpoint au syscall avec "catch syscall write" / "catch syscall 1"
et on voit :

```
[#0] 0x447a07 → write()
[#1] 0x40f5ad → _IO_new_file_write()
[#2] 0x410750 → _IO_new_do_write()
[#3] 0x411233 → _IO_new_file_overflow()
[#4] 0x40c2a2 → puts()
[#5] 0x4017cc → main()
```

Si j'affiche une variable au lieu d'hello world :

```
[#0] 0x44fa17 → write()
[#1] 0x41569d → _IO_new_file_write()
[#2] 0x416840 → _IO_new_do_write()
[#3] 0x415ea7 → _IO_new_file_xsputn()
[#4] 0x40ecdf → __vfprintf_internal()
[#5] 0x40b62c → printf()
[#6] 0x4017a1 → main()
```

quand on appelle prinft :
printf va regarder la chaine qu'il doit affiche :
il va appeler puts avec un syscall s'il connait la chaine.

si on doit créer une nouvelle chaine, il doit passer par une autre fonction qui utilise le format, lis les variables, remplis le tableau : vprintf.

```c
#include <stdio.h>

void fct(int B) {
    printf("Valeur : %d\n", B);
}

int main() {
    int  A;
    A = 42;
    fct(A);
}
```

```
[#0] 0x44fa37 → write()
[#1] 0x4156bd → _IO_new_file_write()
[#2] 0x416860 → _IO_new_do_write()
[#3] 0x415ec7 → _IO_new_file_xsputn()
[#4] 0x40ecff → __vfprintf_internal()
[#5] 0x40b64c → printf()
[#6] 0x40179d → fct(B=0x2a)
[#7] 0x4017bd → main()
```

rsp & rbp : délimiteur du frame courant.

ABI : Application Binary Interface
sous linux : ABI passe par les registres en priorité. (si on dépasse les 8 arguments, le reste passe par la pile)
sous windows : tout le monde passe par la pile.

```
00000000004017a0 <main>:

int main() {
  4017a0:       f3 0f 1e fa             endbr64
  4017a4:       55                      push   %rbp
  4017a5:       48 89 e5                mov    %rsp,%rbp
  4017a8:       48 83 ec 10             sub    $0x10,%rsp
        int  A;
        A = 42;
  4017ac:       c7 45 fc 2a 00 00 00    movl   $0x2a,-0x4(%rbp)
        fct(A);
  4017b3:       8b 45 fc                mov    -0x4(%rbp),%eax
  4017b6:       89 c7                   mov    %eax,%edi
```

```
  4017b8:       e8 b8 ff ff ff          call   401775 <fct>
  4017bd:       b8 00 00 00 00          mov    $0x0,%eax
}
  4017c2:       c9                      leave
  4017c3:       c3                      ret
  4017c4:       66 2e 0f 1f 84 00 00    cs nopw 0x0(%rax,%rax,1)
  4017cb:       00 00 00
  4017ce:       66 90                   xchg   %ax,%ax
```

On voit donc bien dans notre main() :
Il push le base pointer et définit ses limites (rbp & rsp).
vu que la pile augmente en "descendant", on y soustrait 16 bits (sub 0x10).

```
        fct(A);
  4017b3:       8b 45 fc                mov    -0x4(%rbp),%eax
  4017b6:       89 c7                   mov    %eax,%edi
  4017b8:       e8 b8 ff ff ff          call   401775 <fct>
  4017bd:       b8 00 00 00 00          mov    $0x0,%eax
```

On y voit la fonction qui y envoit les 3 arguments et qui appelle la fonction.