

100

- Définitions

- Définitions
- Les différents type d'analyse
- Les différents univers
- Ce que couvre ce cours

Définitions

Définition

To search for and eliminate malfunctioning elements or errors in.

Les différents type d'analyse

- Activer, lire et comprendre les messages de log.

Les différents type d'analyse

- Activer, lire et comprendre les messages de log.
- Tracer un processus.

Les différents type d'analyse

- Activer, lire et comprendre les messages de log.
- Tracer un processus.
- Récupérer la pile d'exécution d'un processus.

Les différents type d'analyse

- Activer, lire et comprendre les messages de log.
- Tracer un processus.
- Récupérer la pile d'exécution d'un processus.
- Analyser la mémoire d'un processus.

Les différents univers

Définition

Au sein du système d'exploitation, on peut distinguer trois univers différents :

- L'espace utilisateur : tout ce qui est lancé par un utilisateur.
- L'espace noyau : l'ensemble des composants internes du système d'exploitation (les drivers, le noyau, les modules...).
- Le matériel : les périphériques, la mémoire, le CPU...

Les différents univers

UserSpace

Terminal / Window Manager

Systemd / Init

Code utilisateur

Librairie GNU C (libc)

Ce que couvre ce cours

- Appréhender la construction d'un fichier binaire et comprendre les mécanismes en jeux lors de son exécution.

Introduction

Et si on commençait doucement ?

Avant tout, nous avons besoin de comprendre un minimum d'instructions Assembleur x86 !

Introduction

Et si on commençait doucement ?



Introduction

Et si on commençait doucement ?

Les registres

```
%rax, %rbx, %rcx, %rdx, %rdi, %rsi, %r[8-15]
```

- **%rax** : Accumulateur, sert à effectuer des calculs arithmétiques ou à envoyer un paramètre à une interruption.
- **%rbx** : Registre auxiliaire de base, sert à effectuer des calculs arithmétiques ou bien des calculs sur les adresses.
- **%rcx** Registre auxiliaire (compteur), sert généralement comme compteur dans des boucles.
- **%rdx** Registre auxiliaire de données, sert à stocker des données destinées à des fonctions.
- **%rdi** Registre contenant un index de destination : utilisée comme adresse source pour les copies de données
- **%rsi** Registre contenant un index de source : utilisé comme adresse source pour les copies de données
- **%r[8-15]** Registres complémentaires

rax (64bits)		
	eax (32bits)	
		ax (16 bits)
	ah (8bits)	al (8bits)

Introduction

Et si on commençait doucement ?

Les instructions

mov

mov *src*, *dest*

- copie la source (*src*) vers la destination (*dest*)
- l'instruction peut-être suffixée par (*q,l,w,s,b*) et correspond au nombre de bits à copier (ex : *l* - long (4 octets))

syscall

syscall

- Instruction d'exécution d'une interruption logicielle.

Les instructions

Des sources au fichier binaire

Use the source Luke.

```
$ vim add.c
```

Le code source - langage C

```
void main(void){
    exit(2);
}
```

```
$ vim add.s
```

Le code source - Assembleur x86_64

```
.text
.globl _start

_start:
    mov $60, %rax
    mov $2, %rdi
    syscall
```

Plus d'informations sur l'interface avec les appels systèmes et Linux en assembleur :

https://en.wikibooks.org/wiki/X86_Assembly/Interfacing_with_Linux

Les instructions

Des sources au fichier binaire

Use the source Luke.

```
$ vim add.c
```

Le code source - langage C

```
void main(void){  
    exit(2);  
}
```

```
$ vim add.s
```

Le code source - Assembleur x86_64

```
.text  
.globl _start  
  
_start:  
    mov $60, %rax  
    mov $2, %rdi  
    syscall
```

- main : fonction principale du programme
- exit : appel système

Plus d'informations sur l'interface avec les appels systèmes et Linux en assembleur :

https://en.wikibooks.org/wiki/X86_Assembly/Interfacing_with_Linux

Les instructions

Des sources au fichier binaire

Use the source Luke.

```
$ vim add.c
```

Le code source - langage C

```
void main(void){  
    exit(2);  
}
```

```
$ vim add.s
```

Le code source - Assembleur x86_64

```
.text  
.globl _start  
  
_start:  
    mov $60, %rax  
    mov $2, %rdi  
    syscall
```

- mov \$60, rax : numéro de l'appel système
- mov \$2, rdi : paramètre de l'appel système
- syscall : appel système

Plus d'informations sur l'interface avec les appels systèmes et Linux en assembleur :

https://en.wikibooks.org/wiki/X86_Assembly/Interfacing_with_Linux

Les instructions

Des sources au fichier binaire

Use the source Luke.

```
$ vim add.c
```

Le code source - langage C

```
void main(void){  
    exit(2);  
}
```

```
$ vim add.s
```

Le code source - Assembleur x86_32

```
.text  
.globl _start  
  
_start:  
    movl $1, %eax  
    movl $2, %ebx  
    int $0x80
```

- `mov $1, eax` : numéro de l'appel système
- `mov $2, ebx` : paramètre de l'appel système
- `int $0x80` : appel système

Plus d'informations sur l'interface avec les appels systèmes et Linux en assembleur :

https://en.wikibooks.org/wiki/X86_Assembly/Interfacing_with_Linux

```
$ as add.s -o add.o
```

A set of small navigation icons typically found in Beamer presentations, including symbols for back, forward, search, and other slide controls.

Le linker se charge de transformer l'objet binaire obtenu en code exécutable.

```
$ ld add.o -o add
```

L'exécutable - Linker (ld)

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.note.gnu.property	00000030	0000000000400120	0000000000400120	00000120	2**3
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
1	.text	00000010	0000000000401000	0000000000401000	00001000	2**0
	CONTENTS, ALLOC, LOAD, READONLY, CODE					

Contents of section .note.gnu.property:

```
400120 04000000 20000000 05000000 474e5500 .....GNU.
400130 010001c0 04000000 01000000 00000000 .....
400140 020001c0 04000000 01000000 00000000 .....
```

Contents of section .text:

401000 48c7c03c 00000048 c7c70200 00000f05 H..<...H.....

Disassembly of section .text:

```
0000000000401000 <.text>:
```

```
401000: 48 c7 c0 3c 00 00 00    mov     $0x3c,%rax
401007: 48 c7 c7 02 00 00 00    mov     $0x2,%rdi
40100e: 0f 05                   syscall
```

Les instructions

Des sources au fichier binaire

Définition

Une instruction est simplement un ensemble d'octets transmis au processeur.

Définition

Une instruction est simplement un ensemble d'octets transmis au processeur.

Instruction x86_64 (big endian)

Sections:

Contents of section .note.gnu.property:

Contents of section .text:

Disassembly of section .text:

```
00000000000401000 <.text>:
```

```
401000: 48 c7 c0 3c 00 00 00    mov     $0x3c,%rax
401007: 48 c7 c7 02 00 00 00    mov     $0x2,%rdi
40100e: 0f 05                   syscall
```

Les instructions

Des sources au fichier binaire

Définition

Une instruction est simplement un ensemble d'octets transmis au processeur.

Instructions ARM (little endian)

```
add.arm:      file format elf32-littlearm
```

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	00000010	00008054	00008054	00000054	2**2
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
1	.ARM.attributes	00000014	00000000	00000000	00000064	2**0
	CONTENTS, READONLY					

Contents of section .text:

```
8054 0200a0e3 0170a0e3 04002de5 000000ef .....p....-.....
```

Contents of section .ARM.attributes:

```
0000 41130000 00616561 62690001 09000000 A....aeabi.....
0010 06010801          ....
```

Disassembly of section .text:

```
00008054 <_start>:
```

```
8054:      e3a00002      mov     r0, #2
8058:      e3a07001      mov     r7, #1
805c:      e52d0004      push    [r0] ; (str r0, [sp, #-4]!)
8060:      ef000000      svc     0x00000000
```

Les instructions

Des sources au fichier binaire

Définition

Une instruction est simplement un ensemble d'octets transmis au processeur.

Instructions ARM (little endian)

Les processeurs ARM récents sont également en big endian, mais il n'est pas improbable d'être face à cette notation d'instructions inversées.

Les instructions

Des sources au fichier binaire

Lecture du fichier binaire

- `$ objdump -dsh add.o`
- `$ xxd add.o`

Les instructions

Des sources au fichier binaire

ELF¹⁰¹ a Linux executable walk-through ANGE ALBERTINI CORKAM.COM

DISSECTED FILE

3. Where the linker is, it's compiled all the way to the end of the file.

HEADER¹⁰¹ ENTRY POINT

SECTIONS

HEADER¹⁰¹ ENTRY POINT

ELF HEADER

PROGRAM HEADER TABLE

CODE

DATA

SECTIONS NAMES

SECTION HEADER TABLE

1. The ELF header is the first thing you see when you open an ELF file. It contains information about the file's architecture, version, and other metadata.

2. The program header table is a table of pointers to the segments of the file. Segments are the units of data that are loaded into memory when the file is executed.

3. The section header table is a table of pointers to the sections of the file. Sections are the units of data that are linked together to form the final executable.

LOADING PROCESS

1. HEADER
THE ELF HEADER IS PARSED
THE PROGRAM HEADER IS PARSED
(SECTIONS ARE NOT USED)

2. MAPPING
THE FILE IS MAPPED IN MEMORY
ACCORDING TO ITS SEGMENTS

3. EXECUTION
ENTRY IS CALLED
SYSCALLS ARE ACCESSED VIA:
- SYSCALL NUMBER IN THE RAX REGISTER
- CALLING INSTRUCTION SYSCALL

TRIVIA

THE ELF WAS FIRST SPECIFIED BY U.S. "C" AND "UI"
FOR UNIX SYSTEM V, IN 1989

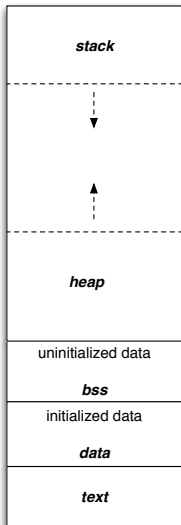
THE ELF IS USED, AMONG OTHERS, IN:
- LINUX, ANDROID, BSD, SOLARIS, BEOS
- PSP, PLAYSTATION 2-4, DREAMCAST, GAMECUBE, Wii
- VARIOUS OSes MADE BY SAMSUNG, ERICSSON, NOKIA
- MICROCONTROLLERS FROM ATMEL, TEXAS INSTRUMENTS

VERSION 1.01 2015/12/04

<https://imgur.com/a/JE0bt>

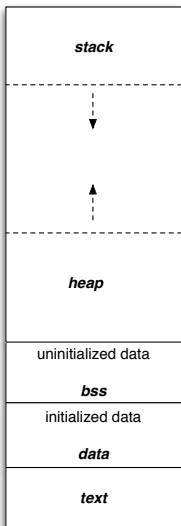
L'exécution

Mise en place en mémoire



L'exécution

Mise en place en mémoire

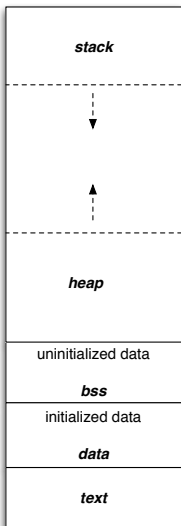


text (fixé à la compilation)

Appelée aussi *code section* : ensemble du code exécutable.

L'exécution

Mise en place en mémoire



data (fixé à la compilation)

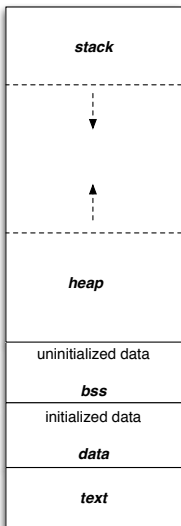
Variables globales initialisées du code.

text (fixé à la compilation)

Appelée aussi *code section* : ensemble du code exécutable.

L'exécution

Mise en place en mémoire



bss (fixé à la compilation)

Block Started by Symbol : Variables globales non initialisées du code.

data (fixé à la compilation)

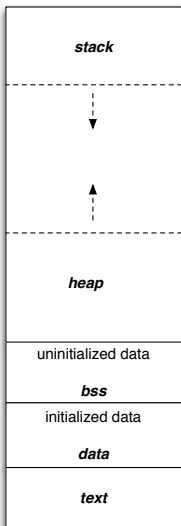
Variables globales initialisées du code.

text (fixé à la compilation)

Appelée aussi *code section* : ensemble du code exécutable.

L'exécution

Mise en place en mémoire



heap (allouée à l'exécution)

Le *tas* : section mémoire utilisée pour allouer les variables dynamiques.

bss (fixé à la compilation)

Block Started by Symbol : Variables globales non initialisées du code.

data (fixé à la compilation)

Variables globales initialisées du code.

text (fixé à la compilation)

Appelée aussi *code section* : ensemble du code exécutable.

L'exécution

Le déplacement dans la pile

À partir de ce moment nous avons un exécutable avec son code qui est monté en mémoire à l'exécution.

C'est sur la pile que va se jouer l'exécution des différentes fonctions ainsi que le passage de leurs paramètres et de leurs valeurs de retour.

Les registres

```
%rbp
```

Base Pointer : Adresse de base d'appel de la fonction

```
%rsp
```

Stack Pointer : Adresse courante de la pile (*stack*)

```
%rip
```

Instruction Pointer : Adresse de la prochaine instruction

Appel de la fonction test, %rip est poussé sur la stack et %rip prend la nouvelle valeur de la fonction test

Code C	
→	void main(void){ test(); }
	void test(void){ return in_test(); }
	void in_test(void){ return; }

L'exécution

L'exécution

Exemple d'exécution dans la stack (pile)

Entrée dans la fonction test, et même principe que pour la fonction main

%rsp	Stack
0xf0	0xff (%rbp)
0xe0	0x09 (main)
→0xd0	0xe0 (%rbp)
0xc0	
0xb0	
0xa0	

%rip	Assembleur - section text
0x01	push %rbp
0x02	mov %rsp,%rbp
0x05	callq 0x12 <test>
0x09	mov \$0x0,%eax
0x0d	retq
→ 0x12	push %rbp
0x13	mov %rsp,%rbp
0x1f	callq 0x26 <in_test>
0x24	pop %rbp
0x25	retq
0x26	push %rbp
0x27	mov %rsp,%rbp
0x2a	nop
0x2b	pop %rbp
0x2c	retq

Code C	
→	void main(void){ test(); }
→	void test(void){ return in_test(); }
	void in_test(void){ return; }

Entrée dans la fonction test, et même principe que pour la fonction main

Code C	
→	void main(void){ test(); }
→	void test(void){ return in_test(); }
	void in_test(void){ return; }

Appel de la fonction `in_test`, même manipulation sur `%rip`

Code C	
→	void main(void){ test(); }
→	void test(void){ return in_test(); }
	void in_test(void){ return ; }

L'exécution

L'exécution

Exemple d'exécution dans la stack (pile)

Entrée dans la fonction in_test

%rsp	Stack
0xf0	0xff (%rbp)
0xe0	0x09 (main)
0xd0	0xe0 (%rbp)
0xc0	0x24 (test)
0xb0	0xc0 (%rbp)
0xa0	

%rip	Assembleur - section text
0x01	push %rbp
0x02	mov %rsp,%rbp
0x05	callq 0x12 <test>
0x09	mov \$0x0,%eax
0x0d	retq
0x12	push %rbp
0x13	mov %rsp,%rbp
0x1f	callq 0x26 <in_test>
0x24	pop %rbp
0x25	retq
0x26	push %rbp
0x27	mov %rsp,%rbp
→ 0x2a	nop
0x2b	pop %rbp
0x2c	retq

Code C	
→	void main(void){ test(); }
→	void test(void){ return in_test(); }
→	void in_test(void){ return; }

L'exécution

Exemple d'exécution dans la stack (pile)

Retour de la fonction, %rbp est restauré et retq restaure %rip

%rsp	Stack
0xf0	0xff (%rbp)
0xe0	0x09 (main)
0xd0	0xe0 (%rbp)
0xc0	0x24 (test)
0xb0	
0xa0	

%rip	Assembleur - section text
0x01	push %rbp
0x02	mov %rsp,%rbp
0x05	callq 0x12 <test>
0x09	mov \$0x0,%eax
0x0d	retq
0x12	push %rbp
0x13	mov %rsp,%rbp
0x1f	callq 0x26 <in_test>
0x24	pop %rbp
0x25	retq
0x26	push %rbp
0x27	mov %rsp,%rbp
0x2a	nop
→0x2b	pop %rbp
0x2c	retq

Code C	
→	void main(void){ test(); }
→	void test(void){ return in_test(); }
→	void in_test(void){ return; }

L'exécution

L'exécution

Exemple d'exécution dans la stack (pile)

Retour de la fonction, %rbp est restauré et retq restaure %rip

%rsp	Stack
0xf0	0xff (%rbp)
0xe0	0x09 (main)
0xd0	
0xc0	
0xb0	
0xa0	

%rip	Assembleur - section text
0x01	push %rbp
0x02	mov %rsp,%rbp
0x05	callq 0x12 <test>
0x09	mov \$0x0,%eax
0x0d	retq
0x12	push %rbp
0x13	mov %rsp,%rbp
0x1f	callq 0x26 <in_test>
→0x24	pop %rbp
0x25	retq
0x26	push %rbp
0x27	mov %rsp,%rbp
0x2a	nop
0x2b	pop %rbp
0x2c	retq

Code C	
→	void main(void){ test(); }
→	void test(void){ return in_test(); }
	void in_test(void){ return; }

%rsp	Stack	%rip	Assembleur - section text	Code C
0xf0	0xff (%rbp)	0x01	push %rbp	void main(void){
0xe0		0x02	mov %rsp,%rbp	test();
0xd0		0x05	callq 0x12 <test>	→ }
0xc0		→0x09	mov \$0x0,%eax	
0xb0		0x0d	retq	
0xa0		0x12	push %rbp	void test(void){
		0x13	mov %rsp,%rbp	return in_test();
		0x1f	callq 0x26 <in_test>	}
		0x24	pop %rbp	void in_test(void){
		0x25	retq	return;
		0x26	push %rbp	}
		0x27	mov %rsp,%rbp	
		0x2a	nop	
		0x2b	pop %rbp	
		0x2c	retq	

L'exécution

Exemple d'exécution dans la stack (pile)

Retour de la fonction, %rbp est restauré et retq restaure %rip

%rsp	Stack
0xf0	0xff (%rbp)
0xe0	
0xd0	
0xc0	
0xb0	
0xa0	

%rip	Assembleur - section text
0x01	push %rbp
0x02	mov %rsp,%rbp
0x05	callq 0x12 <test>
0x09	mov \$0x0,%eax
→0x0d	retq
0x12	push %rbp
0x13	mov %rsp,%rbp
0x1f	callq 0x26 <in_test>
0x24	pop %rbp
0x25	retq
0x26	push %rbp
0x27	mov %rsp,%rbp
0x2a	nop
0x2b	pop %rbp
0x2c	retq

Code C	
	void main(void){ test(); → }
	void test(void){ return in_test(); }
	void in_test(void){ return ; }

Sommaire

3 Processus utilisateur

- ps, top...
- strace, ltrace
- Les sources
- gdb

Les deux outils `ps` et `top` permettent de consulter la structure mémoire d'un processus (`task_struct`).

ps, top...

On consulte la consommation CPU et mémoire de chaque processus, ainsi que l'état dans lequel il se trouve (en cours d'exécution, arrêté, bloqué, zombie...).

- ps
 - permet d'observer un ou plusieurs processus à un instant T .
 - système de filtrage par utilisateur, PID ...
- top
 - Visualisation en continue des processus les plus actifs du système.
 - Consultation unifiée de l'uptime de la machine et de l'utilisation de processeurs.

Ces deux outils permettent de consulter la santé du système d'exploitation.

Exemple de sortie avec la commande ps

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	2	0.0	0.0	0	0	?	S	Oct19	0:00	[kthreadd]
root	3	0.0	0.0	0	0	?	S	Oct19	0:05	\ [ksoftirqd/0]
root	5	0.0	0.0	0	0	?	S<	Oct19	0:00	\ [kworker/0:0H]
root	7	0.0	0.0	0	0	?	S	Oct19	4:58	\ [rcu_sched]
root	8	0.0	0.0	0	0	?	S	Oct19	0:00	\ [rcu_bh]
root	9	0.0	0.0	0	0	?	S	Oct19	0:01	\ [migration/0]
root	10	0.0	0.0	0	0	?	S	Oct19	0:00	\ [watchdog/0]
root	11	0.0	0.0	0	0	?	S	Oct19	0:00	\ [watchdog/1]
root	12	0.0	0.0	0	0	?	S	Oct19	0:01	\ [migration/1]
root	13	0.0	0.0	0	0	?	S	Oct19	0:02	\ [ksoftirqd/1]
root	15	0.0	0.0	0	0	?	S<	Oct19	0:00	\ [kworker/1:0H]
root	16	0.0	0.0	0	0	?	S	Oct19	0:00	\ [watchdog/2]
root	17	0.0	0.0	0	0	?	S	Oct19	0:01	\ [migration/2]
root	18	0.0	0.0	0	0	?	S	Oct19	0:02	\ [ksoftirqd/2]

Les différents champs (*ps aux f*)

- PID : Id du processus
- %CPU : Pourcentage d'occupation CPU
- %MEM : Pourcentage d'occupation Mémoire
- VSZ : Mémoire virtuellement utilisable
- RSS : Mémoire réellement utilisée
- TTY : Terminal attaché au processus
- STAT : Status du processus
- START : Date de début du processus
- TIME : Temps système consommé
- COMMAND : Ligne de commande du processus

Exemple de sortie avec la commande ps

ps aux f										
USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	2	0.0	0.0	0	0	?	S	Oct19	0:00	[kthreadd]
root	3	0.0	0.0	0	0	?	S	Oct19	0:05	\ [ksoftirqd/0]
root	5	0.0	0.0	0	0	?	S<	Oct19	0:00	\ [kworker/0:0H]
root	7	0.0	0.0	0	0	?	S	Oct19	4:58	\ [rcu_sched]
root	8	0.0	0.0	0	0	?	S	Oct19	0:00	\ [rcu_bh]
root	9	0.0	0.0	0	0	?	S	Oct19	0:01	\ [migration/0]
root	10	0.0	0.0	0	0	?	S	Oct19	0:00	\ [watchdog/0]
root	11	0.0	0.0	0	0	?	S	Oct19	0:00	\ [watchdog/1]
root	12	0.0	0.0	0	0	?	S	Oct19	0:01	\ [migration/1]
root	13	0.0	0.0	0	0	?	S	Oct19	0:02	\ [ksoftirqd/1]
root	15	0.0	0.0	0	0	?	S<	Oct19	0:00	\ [kworker/1:0H]
root	16	0.0	0.0	0	0	?	S	Oct19	0:00	\ [watchdog/2]
root	17	0.0	0.0	0	0	?	S	Oct19	0:01	\ [migration/2]
root	18	0.0	0.0	0	0	?	S	Oct19	0:02	\ [ksoftirqd/2]

Ensemble des processus noyaux

Exemple de sortie avec la commande ps

◀ ◻ ▶ ◀ ◻ ▶ ◀ ≡ ▶ ◀ ≡ ▶ ≡ ≡ ↺ 🔍 ↻

Exemple de sortie avec la commande top

[top](#)

```
top - 23:37:03 up 6 days, 1:54, 9 users, load average: 0,29, 0,35, 0,26
Tâches: 260 total, 1 en cours, 258 en veille, 0 arrêté, 1 zombie
%Cpu0 : 1,7 ut, 0,7 sy, 0,0 ni, 97,7 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
top - 23:37:29 up 6 days, 1:54, 9 users, load average: 0.27, 0.33, 0.26
Tasks: 260 total, 1 running, 258 sleeping, 0 stopped, 1 zombie
%Cpu0 : 3.4 us, 1.1 sy, 0.0 ni, 95.5 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu1 : 0.0 us, 0.0 sy, 0.0 ni, 100.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu2 : 1.1 us, 0.0 sy, 0.0 ni, 98.9 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu3 : 0.0 us, 1.1 sy, 0.0 ni, 98.9 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 5831144 total, 410448 free, 3567680 used, 1853016 buff/cache
KiB Swap: 6273020 total, 6043320 free, 229700 used, 1592284 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1231	root	20	0	375884	86940	35752	S	4.5	1.5	56:16.89	Xorg
22000	rachel	20	0	2905836	538740	154532	S	2.2	9.2	4:02.72	firefox
2345	mat	20	0	299152	42664	7084	S	1.1	0.7	82:23.29	awesome
5597	mat	20	0	39028	3432	2796	R	1.1	0.1	0:00.01	top
22103	rachel	20	0	2593304	761204	138880	S	1.1	13.1	6:30.29	Web Content
1	root	20	0	119904	4060	2388	S	0.0	0.1	0:03.34	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.04	kthreadd
3	root	20	0	0	0	0	S	0.0	0.0	0:05.30	ksoftirqd/0
5	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	kworker/0:0H
7	root	20	0	0	0	0	S	0.0	0.0	5:00.94	rcu sched

Les premières lignes donnent une vue synthétique de l'utilisation de la machine.

Sous le capot

Chacune de ces commandes va parcourir */proc*, un pseudo filesystem dans lequel on peut trouver les informations de l'ensemble des processus de la machine.

procfs

```
$ cat /proc/self/stat
15933 (cat) R 10118 15933 10118 34820 15933 4194304 88 0 0 0 0 0 0 20 0 1 0 30568230 7647232 193 3121741824
→ 4194304 4240236 140732349587456 140732349586808 140640986640944 0 128 0 0 17 0 0 6340112 6341364
→ 36347904 140732349592700 140732349592720 140732349592720 140732349595631 0
```

D'autres informations sont également disponibles dans *procfs* et sont lus par d'autres outils.

procfs

\$ ls /proc/self/										
attr	clear_refs	cpuset	fd	limits	mem	net	oom_score	projid_map		
↪ sessionid	stat	task								
autogroup	cmdline	cwd	fdinfo	loginuid	mountinfo	ns	oom_score_adj	root		
↪ setgroups	statm	timers								
auxv	comm	environ	gid_map	map_files	mounts	numa_maps	pagemap	sched	smaps	
↪ status	uid_map									
cgroup	coredump_filter	exe	io	maps	mountstats	oom_adj	personality	schedstat	stack	
↪ syscall	wchan									

strace

L'outil *strace* permet de voir l'ensemble des appels systèmes effectués par un processus. Les appels systèmes sont également appelés syscall et correspondent aux fonctions exposées par le noyau vers l'espace utilisateur.

Exemple d'utilisation de strace

```
# strace -e open,getdents ls
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/libpcre.so.3", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/libld.so.2", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/libpthread.so.0", O_RDONLY|O_CLOEXEC) = 3
open("/proc/filesystems", O_RDONLY) = 3
open("/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
open(".", O_RDONLY|O_NONBLOCK|O_DIRECTORY|O_CLOEXEC) = 3
getdents(3, /* 2 entries */, 32768) = 48
getdents(3, /* 0 entries */, 32768) = 0
+++ exited with 0 +++
```

strace est utilisé pour comprendre comment un processus en espace utilisateur inter-agit avec le noyau via les appels systèmes.

strace

L'outil *strace* permet de voir l'ensemble des appels systèmes effectués par un processus. Les appels systèmes sont également appelés syscall et correspondent aux fonctions exposées par le noyau vers l'espace utilisateur.

Exemple d'utilisation de strace

```
# strace -e open,getdents ls
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/libpcr.so.3", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/libdl.so.2", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/libpthread.so.0", O_RDONLY|O_CLOEXEC) = 3
open("/proc/filesystems", O_RDONLY) = 3
open("/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
open(".", O_RDONLY|O_NONBLOCK|O_DIRECTORY|O_CLOEXEC) = 3
getdents(3, /* 2 entries */, 32768) = 48
getdents(3, /* 0 entries */, 32768) = 0
+++ exited with 0 +++
```

strace est utilisé pour comprendre comment un processus en espace utilisateur inter-agit avec le noyau via les appels systèmes.

syscall - int 0x80


```

22:42:17.942028 execve("/bin/ls", ["ls"], [/ * 50 vars */]) = 0
22:42:17.942537 brk(NULL) = 0xb60000
22:42:17.942644 access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
22:42:17.942774 mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f978836f000
22:42:17.942834 access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
22:42:17.942883 open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
22:42:17.942928 fstat(3, {st_mode=S_IFREG|0644, st_size=251122, ...}) = 0
22:42:17.942969 mmap(NULL, 251122, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f9788331000
22:42:17.943007 close(3) = 0
22:42:17.943045 access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
22:42:17.943096 open("/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
22:42:17.943136 read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\0\3\0\0\0\1\0\0\0\260Z\0\0\0\0\0\0...", 832) = 832
22:42:17.943177 fstat(3, {st_mode=S_IFREG|0644, st_size=130224, ...}) = 0
22:42:17.943215 mmap(NULL, 2234080, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f9787f2a000
22:42:17.943255 mprotect(0x7f9787f49000, 2093056, PROT_NONE) = 0
22:42:17.943297 mmap(0x7f9788148000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1e000) = 0x7f9788148000
22:42:17.943346 mmap(0x7f978814a000, 5856, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f978814a000
22:42:17.943390 close(3) = 0
22:42:17.943429 access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
22:42:17.943473 open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
22:42:17.943513 read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\0\3\0\0\0\1\0\0\0\0P\2\0\0\0\0\0...", 832) = 832
22:42:17.943550 fstat(3, {st_mode=S_IFREG|0755, st_size=1868984, ...}) = 0
22:42:17.943588 mmap(NULL, 3971488, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f9787b60000
22:42:17.943657 mprotect(0x7f9787d20000, 2097152, PROT_NONE) = 0
22:42:17.943707 mmap(0x7f9787f20000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1c0000) = 0x7f9787f20000

```

Voir le nombre d'appel utilisés : -c

% time	seconds	uscs/call	calls	errors	syscall
0.00	0.000000	0	7		read
0.00	0.000000	0	1		write
0.00	0.000000	0	8		open
0.00	0.000000	0	10		close
0.00	0.000000	0	9		fstat
0.00	0.000000	0	18		mmap
0.00	0.000000	0	12		mprotect
0.00	0.000000	0	1		munmap
0.00	0.000000	0	3		brk
0.00	0.000000	0	2		rt_sigaction
0.00	0.000000	0	1		rt_sigprocmask
0.00	0.000000	0	2		ioctl
0.00	0.000000	0	7	7	access
0.00	0.000000	0	1		execve
0.00	0.000000	0	2		getdents
0.00	0.000000	0	1		getrlimit
0.00	0.000000	0	2	2	statfs
0.00	0.000000	0	1		arch_prctl
0.00	0.000000	0	1		set_tid_address
0.00	0.000000	0	1		set_robust_list
100.00	0.000000		90	9	total

Itrace

L'outil *ltrace* permet de voir l'ensemble des appels aux bibliothèques dynamiques d'un programme. Les informations fournies ici correspondent aux différentes fonctions appelées dans les bibliothèques du système. Les appels systèmes ne sont pas visibles.

```
ltrace -e opendir -e readdir ls
# ltrace -e opendir -e readdir ls
ls->opendir(".")           = 0xf6bca0
ls->readdir(0xf6bca0)       = 0xf6bcd0
ls->readdir(0xf6bca0)       = 0xf6bce8
ls->readdir(0xf6bca0)       = 0
+++ exited (status 0) +++
```

ltrace est utilisé pour comprendre le comportement d'un processus utilisateur en dehors des appels système.

Les options

ltrace possède les mêmes options que *strace* :

- -e : selection des fonctions à observer
- -tt : temps passé dans chaque fonction
- -c : nombre d'appel utilisé pour chaque fonction
- -p : PID sur lequel s'attacher

ltrace

L'outil *ltrace* permet de voir l'ensemble des appels aux bibliothèques dynamiques d'un programme. Les informations fournies ici correspondent aux différentes fonctions appelées dans les bibliothèques du système. ~~Les appels systèmes ne sont pas visibles.~~

```

# ltrace -e opendir -e readdir ls
ls->opendir(".")                = 0xf6bca0
ls->readdir(0xf6bca0)            = 0xf6bcd0
ls->readdir(0xf6bca0)            = 0xf6bce8
ls->readdir(0xf6bca0)            = 0
+++ exited (status 0) +++

```

ltrace est utilisé pour comprendre le comportement d'un processus utilisateur en dehors des appels système.

Les options

ltrace possède les mêmes options que *strace* :

- -e : sélection des fonctions à observer
- -tt : temps passé dans chaque fonction
- -c : nombre d'appel utilisé pour chaque fonction
- -p : PID sur lequel s'attacher
- -S : affiche également les appels systèmes

Les sources

Open source is good for me. I will fully embrace it.
Open source is good for me. I will fully embrace it.
Open source is good for me. I will fully embrace it.
Open source is good for me. I will fully embrace it.
Open source is good for me. I will fully embrace it.
Open source is good for me. I will fully embr



Les sources

Le point fondamental permettant de debugger efficacement est d'avoir accès aux sources. Qu'il s'agisse du noyau ou d'un autre programme, les sources facilitent grandement la compréhension d'un problème.

Pour avoir un environnement de debugging complet, il faut :

- La version du logiciel à debugger.
- Les sources correspondantes.
- Les symboles de debug : fournis par la distribution ou à compiler soi-même.


```
Find this C symbol:
Find this global definition: task_struct
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
Find assignments to this symbol:
```

gdb est l'outil par excellence de debug. Il permet d'inspecter le comportement et le contenu d'un exécutable.

- Permet de lancer l'exécution d'un programme pas à pas (breakpoints).
- Visualise les différentes variables des fonctions en cours.
- Affiche la pile d'exécution.
- Affiche les registres.
- Peut aussi modifier le comportement du programme (en modifiant des variables par exemple).

Il existe 2 modes de lancement pour *gdb*

Mode classique au lancement de l'exécutable

```
gdb <exécutable> [corefile]
```

Mode attachement à un processus en cours

```
gdb -p <PID>
```

Les commandes principales

- `break <ligne>` : place un point d'arrêt dans le code.

Exemple d'utilisation de gdb

```
(gdb) break opendir
Breakpoint 1 at 0x7ffff78b3140: file ../sysdeps/posix/opendir.c, line 181.
(gdb) run /
Starting program: /bin/ls /
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, _{opendir (name=0x625cc0 "/" at ../sysdeps/posix/opendir.c:182
182 ../sysdeps/posix/opendir.c: Aucun fichier ou dossier de ce type.
(gdb) where
#0 _{opendir (name=0x625cc0 "/" at ../sysdeps/posix/opendir.c:182
↳
#1 0x000000000403849 in ?? (
#2 0x00007ffff780b830 in _{libc_start_main (main=0x402a00, argc=2,
    argv=0x7fffffff118, init=<optimized out>, fini=<optimized out>,
    rtdl_fini=<optimized out>, stack_end=0x7fffffff108)
    at ../csu/libc-start.c:291
#3 0x0000000004049c9 in ?? (}}}}
↳
```


Les commandes principales

- `break <ligne>` : place un point d'arrêt dans le code.
- `run <args>` : lance l'exécutable avec les arguments spécifiés.

Exemple d'utilisation de gdb

```
(gdb) break opendir
Breakpoint 1 at 0x7ffff78b3140: file ../sysdeps/posix/opendir.c, line 181.
(gdb) run /
Starting program: /bin/ls /
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, _{opendir (name=0x625cc0 "/" at ../sysdeps/posix/opendir.c:182
182 ../sysdeps/posix/opendir.c: Aucun fichier ou dossier de ce type.
(gdb) where
#0 _{opendir (name=0x625cc0 "/" at ../sysdeps/posix/opendir.c:182
↳
#1 0x000000000403849 in ?? (
#2 0x00007ffff780b830 in _{libc_start_main (main=0x402a00, argc=2,
    argv=0x7fffffff118, init=<optimized out>, fini=<optimized out>,
    rtdl_fini=<optimized out>, stack_end=0x7fffffff108)
    at ../csu/libc-start.c:291
#3 0x0000000004049c9 in ?? (}}}}
```

Les commandes principales

- `break <ligne>` : place un point d'arrêt dans le code.
- `run <args>` : lance l'exécutable avec les arguments spécifiés.
- `where` : affiche la pile d'appel d'exécution.

Exemple d'utilisation de gdb

```
(gdb) break opendir
Breakpoint 1 at 0x7ffff78b3140: file ../sysdeps/posix/opendir.c, line 181.
(gdb) run /
Starting program: /bin/ls /
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, _{opendir (name=0x625cc0 "/" at ../sysdeps/posix/opendir.c:182
182 ../sysdeps/posix/opendir.c: Aucun fichier ou dossier de ce type.
(gdb) where
#0 _{opendir (name=0x625cc0 "/" at ../sysdeps/posix/opendir.c:182
↳
#1 0x000000000403849 in ?? (
#2 0x00007ffff780b830 in _{libc_start_main (main=0x402a00, argc=2,
    argv=0x7fffffff118, init=<optimized out>, fini=<optimized out>,
    rtdl_fini=<optimized out>, stack_end=0x7fffffff108)
    at ../csu/libc-start.c:291
#3 0x0000000004049c9 in ?? (}}}}
```

Les commandes principales

- `break <ligne>` : place un point d'arrêt dans le code.
- `run <args>` : lance l'exécutable avec les arguments spécifiés.
- `where` : affiche la pile d'appel d'exécution.
- `continue` : continue l'exécution précédemment interrompue.

Exemple d'utilisation de gdb

```
(gdb) break opendir
```

```
Breakpoint 1 at 0x7ffff78b3140: file ../sysdeps/posix/opendir.c, line 181.
```

```
(gdb) run /
```

```
Starting program: /bin/ls /
```

```
[Thread debugging using libthread_db enabled]
```

```
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
```

```
Breakpoint 1, _{opendir (name=0x625cc0 `"/') at ../sysdeps/posix/opendir.c:182
```

```
182 ../sysdeps/posix/opendir.c: Aucun fichier ou dossier de ce type.
```

(gdb) where

```
#0  _fopendir (name=0x625cc0 ``/'') at ../sysdeps/posix/opendir.c:182
```

```
#1 0x00000000000403849 in ?? (
```

```
#2 0x00007ffff780b830 in _libc_start_main (main=0x402a00, argc=2,
      argv=0x7fffffff118, init=<optimized out>, fini=<optimized out>,
      rtdl_fini=<optimized out>, stack_end=0x7fffffff108)
    at ../csu/libc-start.c:291
#3 0x000000000000049c9 in ?? ()
```


Les commandes principales

- `break <ligne>` : place un point d'arrêt dans le code.
- `run <args>` : lance l'exécutable avec les arguments spécifiés.
- `where` : affiche la pile d'appel d'exécution.
- `continue` : continue l'exécution précédemment interrompue.
- `set <variable>=<valeur>` : modifie la valeur de la variable.

Exemple de backtrace de tous les threads d'un processus utilisateur

(gdb) thread apply all where

Thread 1 (LWP 20141):

```
#0 0x00007fd16877807a in __GI___waitpid (pid=-1, stat_loc=0x7ffffbce8c220, options=10) at
↳ ../sysdeps/unix/sysv/linux/waitpid.c:29
#1 0x000000000044706d in ?? ()
#2 0x0000000000044854b in wait_for ()
#3 0x000000000004384bf in execute_command_internal ()
#4 0x0000000000043851e in execute_command ()
#5 0x0000000000042139e in reader_loop ()
#6 0x0000000000041fdb1 in main ()
```

Autres commandes

- up/down : déplacement dans la stack d'exécution.

Registres gdb

(gdb) info registers

rax	0xffffffffffffdfc	-516
rbx	0x0	0
rcx	0x7f0d6900a1b1	139695572951473
rdx	0x0	0
rsi	0x0	0
rdi	0x7fff561480b0	140734637572272
rbp	0x7fff561480f0	0x7fff561480f0
rsp	0x7fff561480e0	0x7fff561480e0
r8	0x557f6f251770	94005813909360
r9	0x7f0d6930d9d0	139695576111568
r10	0x62f	1583
r11	0x246	582
r12	0x557f6f251590	94005813908880
r13	0x7fff561481e0	140734637572576
r14	0x0	0
r15	0x0	0
rip	0x557f6f2516ca	0x557f6f2516ca <loop+27>
...		

Autres commandes

- up/down : déplacement dans la stack d'exécution.
- info registers : affiche les registres.
- info source : Information sur les sources du binaire actuellement analysé.

Information sur les sources

```
(gdb) info source
Current source file is src.c
Compilation directory is /some/where
Located in /some/where/src.c
Contains 16 lines.
Source language is c.
Producer is GNU C11 7.2.0 -mtune=generic -march=x86-64 -g.
Compiled with DWARF 2 debugging format.
Does not include preprocessor macro info.
```

- up/down : déplacement dans la stack d'exécution.
- info registers : affiche les registres.
- info source : Information sur les sources du binaire actuellement analysé.
- disassemble : affiche le code assembleur d'une fonction.

```

Dump of assembler code for function func:
0x0000557f6f2516af <+0>:      push    %rbp
0x0000557f6f2516b0 <+1>:      mov     %rsp,%rbp
0x0000557f6f2516b3 <+4>:      sub     $0x10,%rsp
0x0000557f6f2516b7 <+8>:      movl    $0x0,-0x4(%rbp)
0x0000557f6f2516be <+15>:     jmp     0x557f6f2516ca <func+27>
0x0000557f6f2516c0 <+17>:     mov     $0x3e8,%edi
0x0000557f6f2516c5 <+22>:     callq   0x557f6f251580 <test+5>
-> 0x0000557f6f2516ca <+27>:     cmpl    $0x0,-0x4(%rbp)
0x0000557f6f2516ce <+31>:     sete    %al
0x0000557f6f2516d1 <+34>:     movzbl  %al,%eax
0x0000557f6f2516d4 <+37>:     mov     %eax,%edi
0x0000557f6f2516d6 <+39>:     callq   0x557f6f25169a <print>
0x0000557f6f2516db <+44>:     test    %eax,%eax
0x0000557f6f2516dd <+46>:     jne     0x557f6f2516c0 <func+17>
0x0000557f6f2516df <+48>:     nop
0x0000557f6f2516e0 <+49>:     leaveq  %rsp
0x0000557f6f2516e1 <+50>:     retq

```

◀ ◻ ▶ ◀ ◻ ▶ ◀ ≡ ▶ ◀ ≡ ▶ ≡

Un des gros intérêts de gdb est qu'il peut analyser ce qu'il s'est passé sur un processus de façon post-mortem.

Les corefiles

Les fichiers cores ou corefiles contiennent l'ensemble de la mémoire d'un processus. Ces fichiers sont générés de différentes façons :

- lorsque le processus a effectué une erreur de segmentation. (`ulimit -c`).
- lorsque le processus a effectué une erreur système (`ulimit -c`).
- à la demande : il est possible de demander au système de créer un corefile à tout moment (`gcore <PID>`).

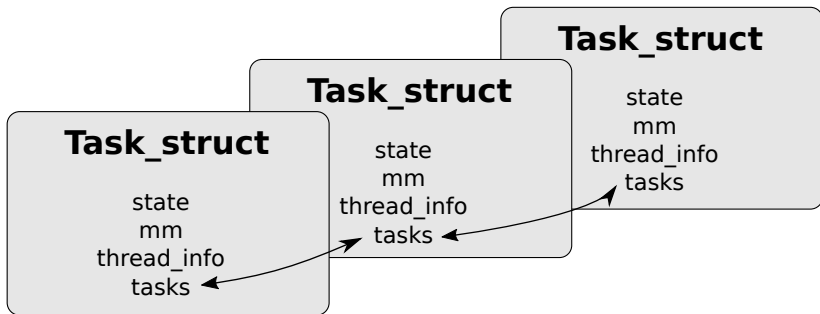
Exemple de génération d'un corefile

```
$ cat /proc/sys/kernel/core_pattern  
|/usr/share/apport/apport %p %s %c %P  
$ ulimit -c unlimited  
$ gcore 21768
```

Sommaire

- 1 Debugger
- 2 Les instructions
- 3 Processus utilisateur
- 4 **Noyau**
 - Au delà de l'espace utilisateur
 - Logs kernel
 - debugfs
 - Crash
 - Autres outils
- 5 Question

Mais qu'est-ce donc que cette `task_struct` ??



Mais qu'est-ce donc que cette `task_struct` ??

```

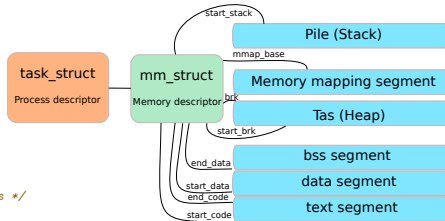
struct task_struct {
...
    /* -1 unrunnable, 0 runnable, >0 stopped: */
    volatile long                state;
    void                        *stack;
    atomic_t                    usage;
...
    unsigned int                cpu;
...

    struct mm_struct            *mm;
    struct mm_struct            *active_mm;

    /* Per-thread vma caching: */
    struct vmcache                vmacache;

struct mm_struct {
    struct vm_area_struct *mmap;                /* list of VMAs */
...
    unsigned long mmap_base;                /* base of mmap area */
...
    unsigned long start_code, end_code, start_data, end_data;
    unsigned long start_brk, brk, start_stack;
...
};

```



- La *task_struct* est une représentation d'un processus au sein du noyau.
- Elle contient une structure mémoire, la *mm_struct* dans laquelle le noyau chargera les différents segments du binaire.
- C'est ensuite le scheduler qui se chargera d'organiser le lancement des différents processus représentés par ces *task_struct*.

Au delà de l'espace utilisateur

Attention, on s'accroche (bis)



Et les *syscalls* dans le noyau ?

Reprenons notre exemple d'appel à un syscall :

Le code source - Assembleur x86_64

```
.text
.globl _start

_start:
    mov $60, %rax
    mov $2, %rdi
    syscall
```

- Les syscalls sont définis par architecture
 - `arch/x86/entry/syscalls/syscall_32.tbl` : processeur x86
 - `arch/x86/entry/syscalls/syscall_64.tbl` : processeur x86_64
 - `arch/arm/tools/syscall.tbl` : processeur arm

Et les *syscalls* dans le noyau ?

Reprenons notre exemple d'appel à un syscall :

Le code source - Assembleur x86_64

```
.text
.globl _start

_start:
    mov $60, %rax
    mov $2, %rdi
    syscall
```

syscall_64.tbl

```
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The abi is "common", "64" or "x32" for this file.
#
0      common      read      sys_read
1      common      write     sys_write
2      common      open      sys_open
3      common      close     sys_close
...
60     common      exit      sys_exit
...
```


L'implémentation des *syscalls*

- Le code déclenché par l'appel à un *syscall* est déclaré via les macros *SYSCALL_DEFINE**n*.
- *n* correspond au nombre de paramètres de l'appel système.
- Dans notre exemple *sys_exit* n'a qu'un paramètre.
- Il est donc défini avec la macro *SYSCALL_DEFINE1*.

L'implémentation des *syscalls*

- Le code déclenché par l'appel à un *syscall* est déclaré via les macros *SYSCALL_DEFINE**n*.
- *n* correspond au nombre de paramètres de l'appel système.
- Dans notre exemple *sys_exit* n'a qu'un paramètre.
- Il est donc défini avec la macro *SYSCALL_DEFINE1*.

exit.c

```
SYSCALL_DEFINE1(exit, int, error_code)
{
    do_exit((error_code&0xff)<<8);
}
```

L'implémentation des *syscalls*

exit.c

```
void __noretturn do_exit(long code)
{
    struct task_struct *tsk = current;
    int group_dead;
```

L'implémentation des *syscalls*

exit.c

```
void __noreturn do_exit(long code)
{
    struct task_struct *tsk = current;
    int group_dead;

    tsk->exit_code = code;
    taskstats_exit(tsk, group_dead);

    exit_mm();

    if (group_dead)
        acct_process();
    trace_sched_process_exit(tsk);

    exit_sem(tsk);
    exit_shm(tsk);
    exit_files(tsk);
    exit_fs(tsk);
    if (group_dead)
        disassociate_ctty(1);
    exit_task_namespaces(tsk);
    exit_task_work(tsk);
    exit_thread(tsk);

    ...

    exit_notify(tsk, group_dead);
    proc_exit_connector(tsk);
    mpol_put_task_policy(tsk);

    do_task_dead();
}
```

Les *syscalls*

Pourquoi tant de code dans le syscall exit ?

Les syscalls

Lors de l'appel à notre programme, le syscall *exit* est effectué, mais ce n'est pas tout, le lancement de la commande appelle également un autre syscall : *execve*

Les *syscalls*

Lors de l'appel à notre programme, le syscall `exit` est effectué, mais ce n'est pas tout, le lancement de la commande appelle également un autre syscall : `execve`

_____ strace de notre programme

```
$ strace ./add
execve("./add", [ "./add" ], 0x7ffcfcbe7a20 /* 45 vars */) = 0
exit(2)                                = ?
+++ exited with 2 +++
```

Les syscalls

Le syscall `execve` en a 3 paramètres c'est donc avec la macro *`SYSCALL_DEFINE3`* qu'il sera défini.

```

                                execve.c
SYSCALL_DEFINE3(execve,
    const char __user *, filename,
    const char __user *const __user *, argv,
    const char __user *const __user *, envp)
{
    return do_execve(getname(filename), argv, envp);
}

```

Les *syscalls*

execve.c

```
int do_execve(struct filename *filename,
             const char *__user *const __argv,
             const char *__user *const __envp)
{
    struct user_arg_ptr arg = { .ptr.native = __argv };
    struct user_arg_ptr envp = { .ptr.native = __envp };
    return do_execveat_common(AT_FDCWD, filename, arg, envp, 0);
}
```

Les syscalls

```
----- execve.c -----  
static int do_execveat_common(int fd, struct filename *filename,  
                             struct user_arg_ptr argv,  
                             struct user_arg_ptr envp,  
                             int flags)  
{  
    char *pathbuf = NULL;  
    struct linux_binprm *bprm;
```

Les *syscalls*

```

static int do_execveat_common(int fd, struct filename *filename,
                             struct user_arg_ptr argv,
                             struct user_arg_ptr envp,
                             int flags)
{
    char *pathbuf = NULL;
    struct linux_binprm *bprm;

    bprm = kzalloc(sizeof(*bprm), GFP_KERNEL);
    if (!bprm)
        goto out_files;

    retval = prepare_bprm_creds(bprm);
    if (retval)
        goto out_free;

    check_unsafe_exec(bprm);
    current->in_execve = 1;

    file = do_open_execat(fd, filename, flags);
    retval = PTR_ERR(file);
    if (IS_ERR(file))
        goto out_unmark;

    sched_exec();

    retval = bprm_mm_init(bprm);
    if (retval)
        goto out_unmark;

    bprm->argc = count(argv, MAX_ARG_STRINGS);
    if ((retval = bprm->argc) < 0)
}

```

Les syscalls

- execve ouvre le fichier binaire et le monte en mémoire.
- execve met en place la *mm_struct* avec les différents segment du binaire. (*bprm_mm_init*)
- En suite le noyau lance l'exécution.

Les syscalls

Mais où est la *task_struct* ?

```

$ strace -f -p 5678
5678 write(2, "./code/add ", 11) = 11
5678 read(0, "\r", 1) = 1
5678 write(2, "\n", 1) = 1
5678 clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x7fd69ebd0310) = 15690
5678 setpgid(15690, 15690) = 0
15690 getpid() = 15690
15690 execve("./code/add", [ "./code/add", 0x2791310 /* 45 vars */ ] = 0
15690 exit(2) = ?
15690 +++ exited with 2 +++
5678 <... wait4 resumed> [{WIFEXITED(s) && WEXITSTATUS(s) == 2}], WSTOPPED|WCONTINUED, NULL) = 15690
5678 --- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_EXITED, si_pid=15690, si_uid=1000, si_status=2, si_utime=0, si_stime=0} ---
5678 rt_sigprocmask(SIG_BLOCK, NULL, [], 8) = 0
5678 write(1, "\[33]0:mat@vortex:~/Documents/2017-"... 59) = 59

```

Les *syscalls*

Le syscall clone en a 3 paramètres c'est donc avec la macro
SYSCALL_DEFINE3 qu'il sera défini.

Les syscalls

Le syscall clone en a 5 paramètres c'est donc avec la macro
SYSCALL_DEFINE5 qu'il sera défini.

Les *syscalls*

Le syscall clone en a 5 paramètres c'est donc avec la macro `SYSCALL_DEFINE5` qu'il sera défini.

CLONE(2)
man clone
Linux Programmer's Manual

CLONE(2

NAME

```
clone, __clone2 - create a child process
```

SYNOPSIS

```
/* Prototype for the glibc wrapper function */
```

```
#define GNU_SOURCE
```

```
#include <sched.h>
```

```
int clone(int (*fn)(void *), void *child_stack,
          int flags, void *arg, ...
          /* pid_t *ptid, void *newtls, pid_t *ctid */ );
```

```
/* For the prototype of the raw system call, see NOTES */
```

NOTES

```
long clone(unsigned long flags, void *child_stack,
          int *ptid, int *ctid,
          unsigned long newtls);
```

Les syscalls

Le syscall clone en a 5 paramètres c'est donc avec la macro
SYSCALL_DEFINE5 qu'il sera défini.

 fork.c

```
#ifndef CONFIG_CLONE_BACKWARDS
SYSCALL_DEFINE5(clone, unsigned long, clone_flags, unsigned long, newsp,
                 int __user *, parent_tidptr,
                 unsigned long, tls,
                 int __user *, child_tidptr)
#elif defined(CONFIG_CLONE_BACKWARDS2)
SYSCALL_DEFINE5(clone, unsigned long, newsp, unsigned long, clone_flags,
                 int __user *, parent_tidptr,
                 int __user *, child_tidptr,
                 unsigned long, tls)
#elif defined(CONFIG_CLONE_BACKWARDS3)
SYSCALL_DEFINE6(clone, unsigned long, clone_flags, unsigned long, newsp,
                 int, stack_size,
                 int __user *, parent_tidptr,
                 int __user *, child_tidptr,
                 unsigned long, tls)
#else
SYSCALL_DEFINE5(clone, unsigned long, clone_flags, unsigned long, newsp,
                 int __user *, parent_tidptr,
                 int __user *, child_tidptr,
                 unsigned long, tls)
#endif
{
    return _do_fork(clone_flags, newsp, 0, parent_tidptr, child_tidptr, tls);
}
#endif
```

A set of small navigation icons typically found in Beamer presentations, including symbols for back, forward, search, and other slide controls.

Les syscalls

- *clone* duplique la *task_struct* courante.
- Cette dernière est réinitialisée dans la foulée.
- En suite le noyau insère cette nouvelle tâche dans sa liste de *task_struct*.
- Une nouveau processus fils a été créé.

Résumé

- Le shell appelle le syscall `clone` qui duplique la `task_struct` courante.

Résumé

- Le shell appelle le syscall `clone` qui duplique la `task_struct` courante.
- Cette nouvelle tâche appelle le syscall `execve` qui va remplacer le code de la `task_struct` par celui du binaire spécifié.

Résumé

- Le shell appelle le syscall *clone* qui duplique la *task_struct* courante.
- Cette nouvelle tâche appelle le syscall *execve* qui va remplacer le code de la *task_struct* par celui du binaire spécifié.
- Le syscall *exit* est enfin appelé.

- Le shell appelle le syscall *clone* qui duplique la *task_struct* courante.
- Cette nouvelle tâche appelle le syscall *execve* qui va remplacer le code de la *task_struct* par celui du binaire spécifié.
- Le syscall *exit* est enfin appelé.
- La *task_struct* créée par le syscall *clone* est nettoyée.

Les outils...

dmesg

dmesg permet d'accéder aux messages du noyau.

- Donne les premières informations permettant de comprendre un problème provenant du noyau.
- Accès à tous les messages depuis le dernier démarrage de la machine.
- En interne, c'est le fichier */proc/kmsg* qui est lu dans un ring buffer.

dmesg

dmesg permet d'accéder aux messages du noyau.

- Donne les premières informations permettant de comprendre un problème provenant du noyau.
- Accès à tous les messages depuis le dernier démarrage de la machine.
- En interne, c'est le fichier `/proc/kmsg` qui est lu dans un ring buffer.

```

                                dmesg
[ 0.072938] PM: Registering ACPI NVS region [mem 0xbf641000-0xbf683fff] (274432 bytes)
[ 0.073006] clocksource: jiffies: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns: 7645041785100000 ns
[ 0.073019] futex hash table entries: 1024 (order: 4, 65536 bytes)
[ 0.073091] pinctrl core: initialized pinctrl subsystem
[ 0.073189] RTC time: 7:49:09, date: 11/27/17
[ 0.073278] NET: Registered protocol family 16
[ 0.082796] cpuidle: using governor ladder
[ 0.087453] cpuidle: using governor menu
[ 0.087462] PCCT header not found.
[ 0.087548] ACPI: bus type PCI registered
[ 0.087552] acpiphp: ACPI Hot Plug PCI Controller Driver version: 0.5
[ 0.087617] PCI: MMCONFIG for domain 0000 [bus 00-ff] at [mem 0xe0000000-0xefffffff] (base 0xe0000000)
[ 0.087623] PCI: not using MMCONFIG
[ 0.087626] PCI: Using configuration type 1 for base access
[ 0.087744] NMI watchdog: enabled on all CPUs, permanently consumes one hw-PMU counter.
```

- `-T` : Affiche les timestamps dans un format lisible
- `-w` : Affiche les nouveaux messages dès leur réception

Debugfs

Debugfs est ce qu'on appelle un pseudo système de fichiers, il permet d'accéder aux fonctions dites de debug du noyau.

Accéder à debugfs

```
# mkdir /mnt/debug
# mount -t debugfs none /mnt/debug
```

Contenu de debugfs

```
# ls /sys/kernel/debug/
acpi          intel_powerclamp  regulator
bdi           iosf_sb           sched_features
btrfs        kprobes           sleep_time
cleancache    kvm               sunrpc
clk           mce               suspend_stats
dma_buf       mei0              tracing
dri           pinctrl           usb
dynamic_debug pkg_temp_thermal virtio-ports
extfrag       pm_qos            wakeup_sources
fault_around_bytes pstate_snb        x86
frontswap     pwm               zswap
gpio          ras
hid           regmap
```

De nombreuses possibilités y sont offertes, nous allons uniquement nous concentrer sur deux d'entre-elles :

- dynamic_debug
- tracing

dynamic_debug

Le `dynamic_debug` est une fonctionnalité du noyau permettant d'activer les messages de debugging du noyau.
 Le fichier `dynamic_debug/control` permet de lister et de contrôler ces activations.

```

----- Messages de debug activés -----
# awk '$3 != "=" /sys/kernel/debug/dynamic_debug/control
# filename:lineno [module]function flags format
init/main.c:741 [main]initcall_blacklisted =p "initcall %s blacklisted\012"
init/main.c:717 [main]initcall_blacklist =p "blacklisting initcall %s\012"
arch/x86/kernel/cpu/mtrr/main.c:491 [main]mtrr_del_page =p "mtrr: no MTRR for %lx000,%lx000 found\012"
arch/x86/kernel/cpu/mtrr/main.c:399 [main]mtrr_check =p "mtrr: size: 0x%lx base: 0x%lx\012"
arch/x86/kernel/cpu/mtrr/generic.c:444 [generic]print_mtrr_state =p "TOM2: %016llx aka %lldM\012"
arch/x86/kernel/cpu/mtrr/generic.c:441 [generic]print_mtrr_state =p " %u disabled\012"
arch/x86/kernel/cpu/mtrr/generic.c:439 [generic]print_mtrr_state =p " %u base %0*X%05X000 mask %0*X%05X000 %s\012"
arch/x86/kernel/cpu/mtrr/generic.c:426 [generic]print_mtrr_state =p "MTRR variable ranges %sabled:\012"

----- Activation d'un message de debug -----
# echo "func SYSC_init_module +p" >/sys/kernel/debug/dynamic_debug/control
awk '/SYSC_init_module/' /sys/kernel/debug/dynamic_debug/control
kernel/module.c:3604 [module]SYSC_init_module =p "init_module: umod=%p, len=%lu, uargs=%p\012"
  
```

Une fois activés, ces messages sont consultables via `dmesg`.

[Documentation/admin-guide/dynamic-debug-howto.rst](#)

tracing

Le tracing dans le noyau permet de suivre les différentes fonctions activées pendant une certaine période pour un ou plusieurs processus. Le principe consiste à définir un ensemble de filtre, de choisir une fonction de suivi (tracing) et de l'activer.

- *set_ftrace_pid* permet de définir le processus à observer.
- *current_tracer* donne la fonction de tracing utilisée.
- *tracing_on* active ou désactive le tracing.

Le résultat est stocké dans un ringbuffer dans le fichier :

- /sys/kernel/debug/tracing/trace

```
# tracer: function
#
#          _-----> irqs-off
#          /-----> need-resched
#          | /-----> hardirq/softirq
#          || /-----> preempt-depth
#          ||| /-----> delay
#
# TASK-PID   CPU#  ||||   TIMESTAMP   FUNCTION
#  ||        |   ||||   |
wmiirc-25517 [000] ....  1971.833158: acpi_ps_free_op <-acpi_ps_delete_parse_tree
wmiirc-25517 [000] ....  1971.833158: acpi_os_release_object <-acpi_ps_free_op
wmiirc-25517 [000] ....  1971.833158: kmem_cache_free <-acpi_os_release_object
wmiirc-25517 [000] ....  1971.833158: acpi_ps_get_arg <-acpi_ps_delete_parse_tree
wmiirc-25517 [000] ....  1971.833158: acpi_ps_get_opcode_info <-acpi_ps_get_arg
wmiirc-25517 [000] ....  1971.833158: acpi_ps_free_op <-acpi_ps_delete_parse_tree
wmiirc-25517 [000] ....  1971.833158: acpi_os_release_object <-acpi_ps_free_op
wmiirc-25517 [000] ....  1971.833158: kmem_cache_free <-acpi_os_release_object
wmiirc-25517 [000] ....  1971.833159: acpi_ps_free_op <-acpi_ps_delete_parse_tree
wmiirc-25517 [000] ....  1971.833159: acpi_os_release_object <-acpi_ps_free_op
wmiirc-25517 [000] ....  1971.833159: kmem_cache_free <-acpi_os_release_object
```

La mise en place du tracing étant relativement complexe, l'outil *perf* permet de faciliter son utilisation.

- `perf list` : liste l'ensemble des évènements disponibles
- `perf record` : enregistre une trace
- `perf report` : affiche le résultat d'une trace précédemment effectuée

```

Utilisation de perf
# cd /tmp
# perf record -e ext4:ext4_free_inode -a
^C[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.823 MB perf.data (2 samples) ]

# perf report
```


- `perf list <module>` : liste les évènements disponibles pour le module spécifié.
- `perf stat` : Affiche les statistique CPU.

Utilisation de perf

```
$ sudo perf stat -a sleep 1
```

Performance counter stats for 'system wide':

4003,736886	cpu-clock (msec)	#	3,999 CPUs utilized
280	context-switches	#	0,070 K/sec
11	cpu-migrations	#	0,003 K/sec
1 934	page-faults	#	0,483 K/sec
45 999 809	cycles	#	0,011 GHz
36 771 055	instructions	#	0,80 insn per cycle
7 734 723	branches	#	1,932 M/sec
247 430	branch-misses	#	3,20% of all branches

1,001166320 seconds time elapsed

L'outil perf permet également de créer ses propres sondes.

- perf probe [-m <module>] -F : Affiche les fonctions disponibles pour l'observation.
- perf probe [-m <module>] -V fonction : Affiche les paramètres disponibles pour une fonction donnée.
- perf probe [-m <module>] "fonction param..." : crée une sonde pour la fonction et affiche les paramètres *param* sélectionnés.
- perf script : affiche le résultat.

Utilisation de perf probe

```
[root@vm0 ~]# perf probe -m xfs -V xfs_create
Available variables at xfs_create
    @<xfs_create+0>
        struct xfs_inode*      ip
        struct xfs_name*       name
[root@vm0 ~]# perf probe -m xfs xfs_create name ip
[root@vm0 ~]# perf record -e probe:xfs_create -aRg touch /tmp/test
[root@vm0 ~]# perf script
touch 1798 [001] 11968.735301: probe:xfs_create: (fffffffc02b1b90) name=0xffff8bb2c8a67c30 ip=0x0
60c01 xfs_create (/lib/modules/3.10.0-862.14.4.el7.x86_64/kernel/fs/xfs/xfs.ko.xz)
5e244 xfs_vn_mknod (/lib/modules/3.10.0-862.14.4.el7.x86_64/kernel/fs/xfs/xfs.ko.xz)
5e283 xfs_vn_create (/lib/modules/3.10.0-862.14.4.el7.x86_64/kernel/fs/xfs/xfs.ko.xz)
42c193 vfs_create ([kernel.kallsyms])
42ff80 do_last ([kernel.kallsyms])
430227 path_openat ([kernel.kallsyms])
431dbd do_filp_open ([kernel.kallsyms])
41e0d7 do_sys_open ([kernel.kallsyms])
41e1fe sys_open ([kernel.kallsyms])
92579b system_call ([kernel.kallsyms])
efa30 __GI___libc_open (/usr/lib64/libc-2.17.so)
```

Crash

Présentation

- Crash est une version améliorée de gdb destinée à faciliter le debugging du noyau.
- Il se base sur le fichier `/proc/kcore` ou une copie qui contient la mémoire du noyau à un instant T
- Permet une analyse post-mortem ou en live d'un problème ou d'un mauvais comportement

Crash

Pré-requis

Pour pouvoir utiliser *crash*, il est impératif d'obtenir les symboles de debug du noyau que l'on souhaite debugger

- Soit le noyau est compilé par la distribution et il n'y a qu'à installer les packages :
 - linux-image-XXX.YYY-ZZZ-generic-dbgsym (Ubuntu/Debian)
 - kernel-debuginfo (RedHat/Fedora/CentOS)
 - ...
- Soit le noyau a été compilé depuis les sources et il faut recompiler le noyau avec les symboles de debug :
 - apt source linux (Ubuntu/Debian)
 - kernel-XXX.YYY.src.rpm (RedHat/Fedora/CentOS)
 - <https://www.kernel.org/>

Crash

Kexec/Kdump

Le challenge est de pouvoir récupérer la mémoire du noyau qui vient de planter.

- Kdump est un service qui pre-charge via partir de kexec un noyau et d'un initrd minimal
- La mémoire du noyau de capture doit être réservée au boot du noyau
 - `crashkernel=auto`
- Une fois le service activé, le noyau de capture est prêt à être déclenché et à procéder à une capture.

Crash

Utilisation

Lancement

crash [noyau corefile]

- Sans argument, *crash* analyse le système en live
- En lui spécifiant le noyau avec ses symboles de debug et un corefile précédemment généré, on lance une analyse post-mortem

Crash

Utilisation

Lancement

`crash` [noyau corefile]

```
_____ crash _____
  KERNEL: /usr/lib/debug/boot/vmlinux-4.4.0-97-generic
 DUMPFILE: /proc/kcore
   CPUS: 4
  DATE: Sun Nov 26 21:28:48 2017
 UPTIME: 12 days, 16:41:39
LOAD AVERAGE: 1.01, 1.19, 1.14
  TASKS: 713
 NODENAME: dakoro
 RELEASE: 4.4.0-97-generic
 VERSION: #120-Ubuntu SMP Tue Sep 19 17:28:18 UTC 2017
 MACHINE: x86_64 (3093 Mhz)
  MEMORY: 6 GB
   PID: 28479
 COMMAND: "crash"
  TASK: ffff8800ad5faa00 [THREAD_INFO: ffff8800799ec000]
   CPU: 0
  STATE: TASK_RUNNING (ACTIVE)
```


Lister l'ensemble des tâches du système

`ps [PID | task | commande]`

Lister l'ensemble des tâches du système

ps [PID | task | commande]

							ps	
PID	PPID	CPU	TASK	ST	%MEM	VSZ	RSS	COMM
0	0	0	ffffffffff81e11500	RU	0.0	0	0	[swapper/0]
0	0	1	ffff8801b7a38e00	RU	0.0	0	0	[swapper/1]
0	0	2	ffff8801b7a39c00	RU	0.0	0	0	[swapper/2]
0	0	3	ffff8801b7a3aa00	RU	0.0	0	0	[swapper/3]
1	0	1	ffff8801b79e8000	IN	0.1	185492	4296	systemd
...								
964	1	0	ffff8801b2d79c00	IN	0.0	44920	1556	avahi-daemon
970	1	1	ffff8801b34e8e00	IN	0.0	26044	184	atd
972	1	0	ffff8801b34e9c00	IN	0.1	138412	3676	freshclam
974	1	2	ffff8801b34e8000	IN	0.0	166456	2012	thermald
990	1	3	ffff8801b53a8e00	IN	0.0	276204	2268	accounts-daemon
997	1	3	ffff8801b53ab800	IN	0.0	256396	1220	rsyslogd
1009	1	2	ffff8800b5f30000	IN	0.0	44332	3204	dbus-daemon
1052	1	1	ffff8800b5f34600	IN	0.0	495104	1020	osspd
1053	1	1	ffff8800b5f35400	IN	0.0	495104	1020	osspd
1054	1	3	ffff8800b5f32a00	IN	0.0	495104	1020	osspd
1055	1	3	ffff8800b5f36200	IN	0.0	495104	1020	osspd

Lister l'ensemble des tâches du système

ps [PID | task | commande]

PID	PPID	CPU	TASK	ST	%MEM	VSZ	ps	
							RSS	COMM
0	0	0	ffffffffff81e11500	RU	0.0	0	0	[swapper/0]
0	0	1	ffff8801b7a38e00	RU	0.0	0	0	[swapper/1]
0	0	2	ffff8801b7a39c00	RU	0.0	0	0	[swapper/2]
0	0	3	ffff8801b7a3aa00	RU	0.0	0	0	[swapper/3]
1	0	1	ffff8801b79e8000	IN	0.1	185492	4296	systemd
...								
964	1	0	ffff8801b2d79c00	IN	0.0	44920	1556	avahi-daemon
970	1	1	ffff8801b34e8e00	IN	0.0	26044	184	atd
972	1	0	ffff8801b34e9c00	IN	0.1	138412	3676	freshclam
974	1	2	ffff8801b34e8000	IN	0.0	166456	2012	thermald
990	1	3	ffff8801b53a8e00	IN	0.0	276204	2268	accounts-daemon
997	1	3	ffff8801b53ab800	IN	0.0	256396	1220	rsyslogd
1009	1	2	ffff8800b5f30000	IN	0.0	44332	3204	dbus-daemon
1052	1	1	ffff8800b5f34600	IN	0.0	495104	1020	osspd
1053	1	1	ffff8800b5f35400	IN	0.0	495104	1020	osspd
1054	1	3	ffff8800b5f32a00	IN	0.0	495104	1020	osspd
1055	1	3	ffff8800b5f36200	IN	0.0	495104	1020	osspd

- Affichage de *PID PPID*
 - Adresse de la *task_struct*
 - Status de la tâche
- Mémoire virtuellement disponible
 - Mémoire effectivement utilisée
 - Ligne de commande

Choisir une autre tâche

set [PID | task]

- Par défaut l'ensemble des commandes prennent la tâche courante comme référence
- Pour changer de tâche courante, on utilise la commande `set`
- La sélection de la tâche se fait par *PID* ou par adresse de *task_struct*

Choisir une autre tâche

set [PID | task]

- Par défaut l'ensemble des commandes prennent la tâche courante comme référence
- Pour changer de tâche courante, on utilise la commande `set`
- La sélection de la tâche se fait par *PID* ou par adresse de *task_struct*

```
crash> set 1
PID: 1
COMMAND: "systemd"
TASK: ffff8801b79e8000 [THREAD_INFO: ffff8801b79f0000]
CPU: 1
STATE: TASK_INTERRUPTIBLE
```

Afficher la pile d'exécution d'une tâche

bt [PID]

- Affiche la pile d'exécution d'une tâche.
- Avec l'option *-f* affiche le contenu complet de la pile.
- C'est cette fonction qui rend crash incontournable pour comprendre ce qu'il se passe sur le système.

Afficher la pile d'exécution d'une tâche

bt [PID]

- Affiche la pile d'exécution d'une tâche.
- Avec l'option `-f` affiche le contenu complet de la pile.
- C'est cette fonction qui rend crash incontournable pour comprendre ce qu'il se passe sur le système.

```
crash> bt
PID: 1      TASK: ffff8801b79e8000 CPU: 0  COMMAND: "systemd"
#0 [ffff8801b79f3d40] __schedule at ffffffff8183efce
#1 [ffff8801b79f3d90] schedule at ffffffff8183f6b5
#2 [ffff8801b79f3da8] schedule_hrtimeout_range_clock at ffffffff81842ca3
#3 [ffff8801b79f3e50] schedule_hrtimeout_range at ffffffff81842cd3
#4 [ffff8801b79f3e60] ep_poll at ffffffff81259c40
#5 [ffff8801b79f3f10] sys_epoll_wait at ffffffff8125af28
#6 [ffff8801b79f3f50] entry_SYSCALL_64_fastpath at ffffffff818437f2
RIP: 00007f5e7aa689d3 RSP: 00007fff1c6f6470 RFLAGS: 00000293
RAX: ffffffff8183efce RBX: 00005610aabade00 RCX: 00007f5e7aa689d3
RDX: 00000000000000af RSI: 00007fff1c6f6480 RDI: 0000000000000004
RBP: 0000000000000000 R8: 00007fff1c6f6480 R9: 225870c9894f4527
R10: 00000000ffffffff R11: 0000000000000293 R12: 0000000000000000
R13: 00007fff1c6f4810 R14: 00007fff1c6f4820 R15: 00005610a8ad28e3
ORIG_RAX: 00000000000000e8 CS: 0033 SS: 002b
```


dis fonction

- Permet de comprendre l'enchaînement de la pile d'appel

Afficher le code assembleur d'une fonction

dis fonction

- Permet de comprendre l'enchaînement de la pile d'appel

dis

```
crash> dis schedule
0xffffffff8183f680 <schedule>: callq 0xffffffff818460c0 <ftrace_graph_caller>
0xffffffff8183f685 <schedule+5>: push    %rbp
0xffffffff8183f686 <schedule+6>: mov     %gs:0xd400,%rax
0xffffffff8183f68f <schedule+15>: mov     %rsp,%rbp
0xffffffff8183f692 <schedule+18>: push    %rbx
0xffffffff8183f693 <schedule+19>: mov     (%rax),%rdx
0xffffffff8183f696 <schedule+22>: test    %rdx,%rdx
0xffffffff8183f699 <schedule+25>: je      0xffffffff8183f6a5 <schedule+37>
0xffffffff8183f69b <schedule+27>: cmpq    $0x0,0x6f0(%rax)
0xffffffff8183f6a3 <schedule+35>: je      0xffffffff8183f6c3 <schedule+67>
0xffffffff8183f6a5 <schedule+37>: mov     %gs:0x14304,%rbx
0xffffffff8183f6ae <schedule+46>: xor     %edi,%edi
0xffffffff8183f6b0 <schedule+48>: callq   0xffffffff8183ec50 <_schedule>
0xffffffff8183f6b5 <schedule+53>: mov     -0x3ff8(%rbx),%rax
0xffffffff8183f6bc <schedule+60>: test    $0x8,%al
0xffffffff8183f6be <schedule+62>: jne     0xffffffff8183f6ae <schedule+46>
0xffffffff8183f6c0 <schedule+64>: pop     %rbx
0xffffffff8183f6c1 <schedule+65>: pop     %rbp
0xffffffff8183f6c2 <schedule+66>: retq
```

dis fonction

- Permet de comprendre l'enchaînement de la pile d'appel

```

dis
crash> dis schedule
0xffffffff8183f680 <schedule>:  callq  0xffffffff818460c0 <ftrace_graph_caller>
0xffffffff8183f685 <schedule+5>:      push   %rbp
0xffffffff8183f686 <schedule+6>:      mov    %gs:0xd400,%rax
0xffffffff8183f68f <schedule+15>:     mov    %rsp,%rbp
0xffffffff8183f692 <schedule+18>:     push   %rbx
0xffffffff8183f693 <schedule+19>:     mov    (%rax),%rdx
0xffffffff8183f696 <schedule+22>:     test   %rdx,%rdx
0xffffffff8183f699 <schedule+25>:     je     0xffffffff8183f6a5 <schedule+37>
0xffffffff8183f69b <schedule+27>:     cmpq   $0x0,0x6f0(%rax)
0xffffffff8183f6a3 <schedule+35>:     je     0xffffffff8183f6c3 <schedule+67>
0xffffffff8183f6a5 <schedule+37>:     mov    %gs:0x14304,%rbx
0xffffffff8183f6ae <schedule+46>:     xor     %edi,%edi
0xffffffff8183f6b0 <schedule+48>:     callq  0xffffffff8183ec50 <__schedule>
0xffffffff8183f6b5 <schedule+53>:     mov    -0x3ff8(%rbx),%rax
0xffffffff8183f6bc <schedule+60>:     test   $0x8,%al
0xffffffff8183f6be <schedule+62>:     jne    0xffffffff8183f6ae <schedule+46>
0xffffffff8183f6c0 <schedule+64>:     pop     %rbx
0xffffffff8183f6c1 <schedule+65>:     pop     %rbp
0xffffffff8183f6c2 <schedule+66>:     retq

```

- Adresse de l'instruction
- Offset de l'instruction
- Code assembleur

Afficher la définition d'un symbole

whatis symbole

- Donne la définition d'une structure
- Donne le type d'un symbole
- Donne le prototype d'une fonction

Afficher la définition d'un symbole

whatis symbole

- Donne la définition d'une structure
- Donne le type d'un symbole
- Donne le prototype d'une fonction

crash> *whatis* struct mm_struct

```
struct mm_struct {  
    struct vm_area_struct *mmap;  
    ...  
    unsigned long mmap_base;  
    unsigned long mmap_legacy_base;  
    ...  
    unsigned long start_code;  
    unsigned long end_code;  
    unsigned long start_data;  
    unsigned long end_data;  
    unsigned long start_brk;  
    unsigned long brk;  
    unsigned long start_stack;  
    unsigned long arg_start;  
    unsigned long arg_end;  
    ...  
    struct uprobes_state uprobes_state;  
    void *bd_addr;  
    atomic_long_t hugetlb_usage;  
}  
SIZE: 968
```

Afficher la définition d'un symbole

print[/format] symbole

- Affiche la valeur d'un symbole
- Si */format* est spécifié, affiche toutes les valeurs dans ce format (x : hexadecimal, d :décimal).
- Pour les structures, la commande *struct* est plus rapide à utiliser

Afficher la définition d'un symbole

print[/format] symbole

- Affiche la valeur d'un symbole
- Si */format* est spécifié, affiche toutes les valeurs dans ce format (x : hexadecimal, d :décimal).
- Pour les structures, la commande *struct* est plus rapide à utiliser

```
crash> print (struct list_head) modules
$11 = {
    next = 0xffffffffc09f2508,
    prev = 0xffffffffc00052c8
}
crash> print/x modules
$12 = {
    next = 0xffffffffc09f2508,
    prev = 0xffffffffc00052c8
}
crash> print/d modules
$13 = {
    next = -1063312120,
    prev = -1073720632
}
crash> print (struct list_head *) modules
$14 = (struct list_head *) 0xffffffffc09f2508
```

THE BEATLES

HELP!



Afficher l'aide en ligne d'une commande

help commande

- Affiche l'aide et les options de la commande

Berkley Packet Filter

Les outils BCC basés sur BPF sont les derniers outils de tracing des fonctions kernel. Ils permettent de façon très efficace de visualiser les fonctions kernel sollicitées à un instant *t*.

BPF

```

[root@vortex ~]# /usr/local/share/bcc/tools/gethostlatency
TIME      PID      COMM          LATms HOST
17:00:55  7291    ping          34.48 google.fr
17:01:22  7658    ping          5.64 192.168.21.254
17:01:34  7805    ping          5.06 192.168.21.254
[root@vortex ~]# /usr/local/share/bcc/tools/tcplife
PID  COMM  LADDR          LPORT RADDR          RPORT TX_KB RX_KB MS
13894 Socket Thr 192.168.21.10 40692 54.204.39.132 443   0    3 333.51
13894 Socket Thr 192.168.21.10 51588 93.184.220.29 80     0    0 5068.39
13894 Socket Thr 192.168.21.10 41800 66.117.28.86 443   0    2 5208.45
13894 Socket Thr 192.168.21.10 42536 193.0.160.129 443   0    2 5251.87
13894 Socket Thr 192.168.21.10 58196 104.94.13.245 443   0    0 5369.00
13894 Socket Thr 192.168.21.10 50638 2.18.139.190 443   0    4 420.49
[root@vortex ~]# /usr/local/share/bcc/tools/softirqs
Tracing soft irq event time... Hit Ctrl-C to end.
SOFTIRQ      TOTAL_usecs
hi            1073
rcu           9771
tasklet      18345
timer        44685
sched        53143
    
```

SystemTap

SystemeTap permet de modifier le comportement du noyau en insérant du code à n'importe quelle adresse.

- Modification de modules
- Visualisation de variables et de structures
- ...

Exemple de script

```
global reads
probe vfs.read
{
    reads[execname()] ++
}

probe timer.s(3)
{
    printf("=====\n")
    foreach (count in reads-)
        if (reads[count] >= 1024)
            printf("%s : %dkB \n", count, reads[count]/1024)
        else
            printf("%s : %dB \n", count, reads[count])
}
```

Sommaire

- 1 Debugger
- 2 Les instructions
- 3 Processus utilisateur
- 4 Noyau
- 5 Question

