ENSICAEN
Computer Science

ERRATAS

Architecture et Technologie des Ordinateurs

Hugo Descoubes - Novembre 2013

Ce document associé à l'enseignement ''Architecture et technologie des ordinateurs'' liste les erreurs présentes au sein des supports de cours ainsi que sur les supports télédiffusés et propose une nouvelle définition ou justification.

Ne pas hésiter à me contacter (hugo.descoubes@ensicaen.fr) si d'autres erreurs ou maladresses sont constatées.

Merci de votre compréhension,

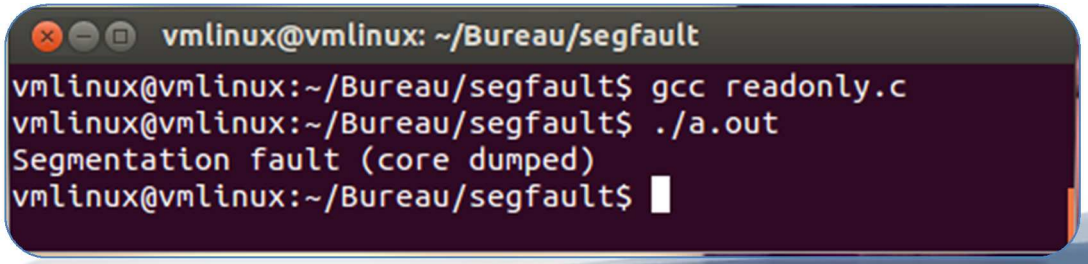
Hugo Descoubes

- ***Partie Gestion mémoire :*** *slide 10, niveaux de privilèges*

Durant le cours vidéo en ligne, j'utilise souvent à tort le mot priorité au lieu de privilège. Ceci est notamment dû à de mauvaises habitudes liées au vocabulaire associé à la programmation concurrente. Programmation sur systèmes d'exploitation temps réel ou non.

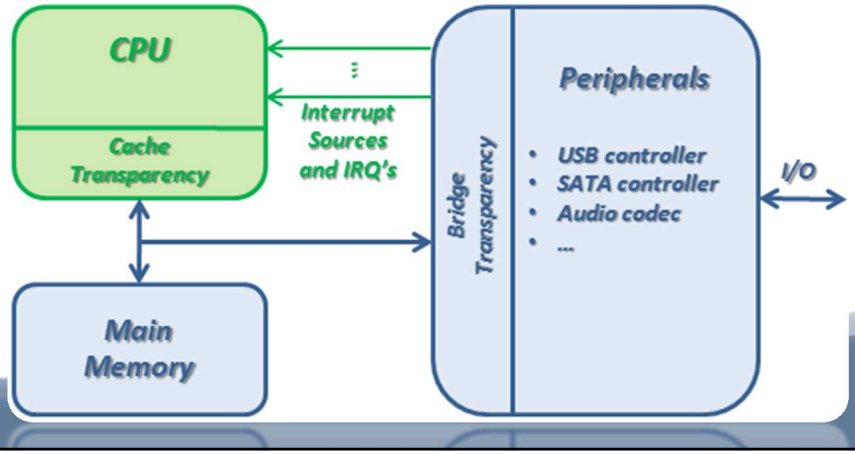
- ***Partie Gestion mémoire :*** *segmentation mémoire*

Dans cette partie, nous allons approfondir et rectifier quelques informations présentées durant le cours en ligne. Intéressons-nous à la segmentation mémoire et notamment au célèbre ***segmentation fault (core dumped)*** *de linux. Pour bien assimiler cette partie, nous nous intéresserons aux exceptions matérielles relevées par le processeur, aux mécanismes de préemption par le kernel et de communication avec les tâches applicatives à la source du défaut.*

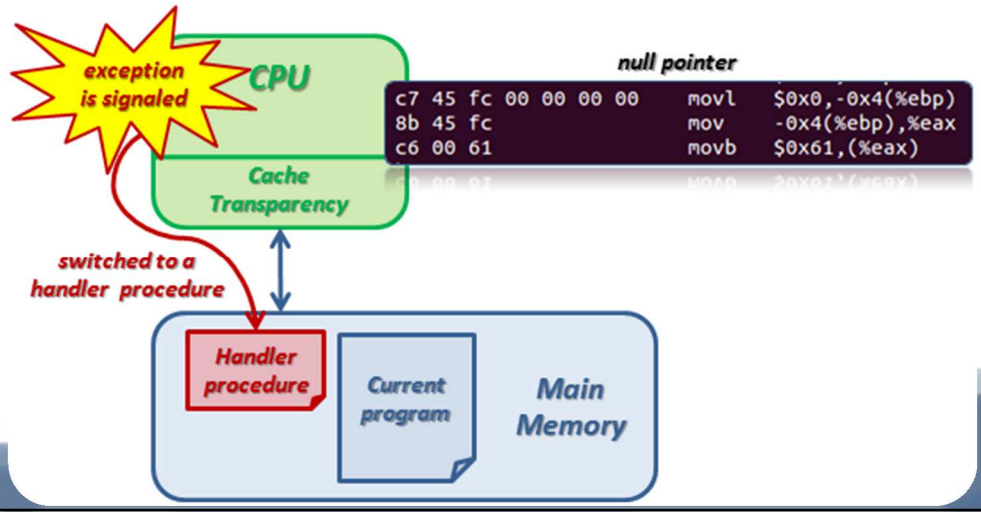


Avant d'appréhender la partie propre au système d'exploitation, attardons nous sur les mécanismes d'interruption d'un programme en cours d'exécution. Comme pour une grande majorité des processeurs à architecture CPU, deux mécanismes cohabitent :

- ***Interruption :*** *Evènement matériel asynchrone de communication typiquement utilisé par les périphériques (vu durant les enseignements de systèmes embarqués).*



- ***Exception :*** *Evènement matériel synchrone généré par le CPU (synchrone au regard du fonctionnement d'un CPU dont les traitements restent synchronisés sur une référence d'horloge, pas au regard de la probabilité d'occurrence). Ces évènements sont relevés par le CPU lorsque celui-ci détecte une voire plusieurs conditions prédéfinies durant l'exécution d'une instruction (violation de privilège, division flottante par zéro, accès illégal en mémoire …).*



Lorsqu'une interruption ou une exception se produit, le CPU stoppe l'exécution du programme en cours et donne la main à une procédure spécifiquement écrite pour traiter l'évènement matériel venant de se produire. Ces fonctions sont appelées ISR's (Interrupt Software/Service Routine) dans le cadre des interruptions. Observons les familles d'exceptions et interruptions supportées sur architecture Intel IA-32 :

Vector No.	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point instruction.[2]
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		Reserved	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory.[3]
18	#MC	Machine Check	Error codes (if any) and source are model dependent.[4]
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction[5]
20-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT *n* instruction.

Chacune de ces familles d'exception englobe différents types de défauts. Prenons en une à titre d'exemple, celle associée aux opérations arithmétiques flottantes hors instructions vectorielles (#MF ou Math Fault) :

Interrupt 16—x87 FPU Floating-Point Error (#MF)

Exception Class Fault.

Description

Indicates that the x87 FPU has detected a floating-point error. The NE flag in the register CR0 must be set for an interrupt 16 (floating-point error exception) to be generated. (See Section 2.5, "Control Registers," for a detailed description of the NE flag.)

NOTE

SIMD floating-point exceptions (#XM) are signaled through interrupt 19.

While executing x87 FPU instructions, the x87 FPU detects and reports six types of floating-point error conditions:

- Invalid operation (#I)
  - Stack overflow or underflow (#IS)
  - Invalid arithmetic operation (#IA)
- Divide-by-zero (#Z)
- Denormalized operand (#D)
- Numeric overflow (#O)
- Numeric underflow (#U)
- Inexact result (precision) (#P)

Sur architecture Intel, les exceptions matérielles sont répertoriées en 3 grandes classes :

- ***Fault :*** *Lorsqu'une exception de ce type arrive, elle peut en général être corrigée et peut potentiellement autoriser la continuité d'exécution du programme ayant causé le défaut (dépend de la stratégie de l'OS). Sous Linux, si un défaut de ce type est détecté, le kernel prend l'initiative d'envoyer un signal (SIGSEGV, SIGFPE, SIGILL, SIGBUS) au processus à la cause du défaut. Par défaut, le processus est alors mis à mort.*

- ***Abort :*** *Défaut critique pour le système, le processus en cause n'est pas autorisé à reprendre la main*

- ***Trap:*** *Ce type d'exception n'est pas un défaut matériel, prenons l'exemple de l'exception #BP ou Break Point. Il s'agit, dans le cas présent, d'un opcode de 1 octet (instruction breakpoint = INT3) remplaçant le premier octet de l'opcode de chaque instruction du programme sous test. Ce type d'exception est utilisé par les outils de debuggage, par exemple gdb (signal UNIX SIGTRAP). En effet, le debugger sera alors appelé à l'exécution de chaque instruction.*

```c
/**
* @file sigtrap.c
* @brief exception #BP et signal UNIX SIGTRA
* @author
* @date novembre 2013
*/
#include <stdio.h>

/**
* @fn void main (void)
* @brief program entry point
*/
int main(int argc, char **argv) {
    __asm__("int3");
return 0;
}
```

```
080483b4 <main>:
 80483b4:       55                      push   %ebp
 80483b5:       89 e5                   mov    %esp,%ebp
 80483b7:       cc                      int3
 80483b8:       b8 00 00 00 00          mov    $0x0,%eax
 80483bd:       5d                      pop    %ebp
 80483be:       c3                      ret
```

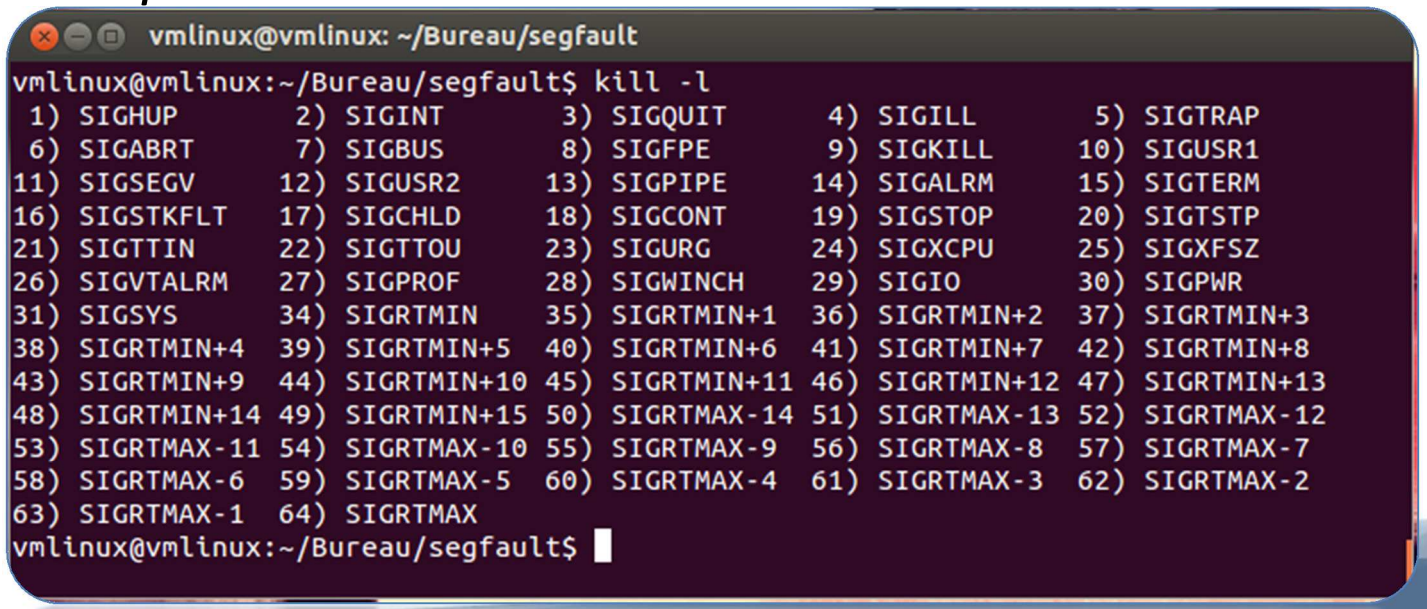
```
vmlinux@vmlinux: ~/Bureau/segfault
vmlinux@vmlinux:~/Bureau/segfault$ gcc sigtrap.c
vmlinux@vmlinux:~/Bureau/segfault$ ./a.out
Trace/breakpoint trap (core dumped)
vmlinux@vmlinux:~/Bureau/segfault$
```

Intéressons nous maintenant aux signaux UNIX et à l'implémentation sur système UNIX-like comme Linux. Attention, ne pas confondre :

- ***interruptions*** *et* ***exceptions :*** *événements matériels*

- ***signaux :*** *notifications logicielles asynchrones envoyées par le kernel à un processus ou thread cible suite à un évènement matériel ou logiciel. Les appels système* ***kill*** *et* ***signal*** *permettent respectivement d'envoyer un signal (évènement logiciel) et de capturer un signal (évènement matériel ou logiciel) depuis un processus.*

En cas d'occurrence, le kernel stoppe l'exécution du processus cible (fait au niveau hardware par le CPU si exception matérielle), celui-ci exécute alors une procédure spécifiquement écrite par le développeur afin de traiter le signal. Sinon, une procédure par défaut est appliquée. Observons les signaux supportés par Linux :



Observons les signaux système associés à des exceptions matérielles :

- ***SIGSEGV :*** *le plus célèbre, signal générant le fameux* ***segmentation fault (core dumped)****. Plusieurs exceptions matérielles peuvent en être à la source : #PF page fault (page non présente en mémoire physique, exécution d'une page non-executable…), #GP General Protection (nombreux défauts principalement associés à des accès mémoire illégaux : écriture sur segment read-only, lecture d'un segment execute-only, dépassement taille limite de segment, violation de privilège, null segment selector …), …*

- ***SIGILL :*** *exécution d'un opcode invalide (exception #UD Invalid Opcode Exception)*

- ***SIGBUS :*** *détection d'erreur sur bus physique. Par exemple détection de défauts d'alignement (exception #AC Alignement Check Exception) ou d'adresses physiques invalides (exception #MC Machine-Check, architecture CPU dépendant).*

- ***SIGFPE :*** *détection d'opérations arithmétiques erronées, par exemple division par zéro, valeur dé-normalisée, overflow ou underflow arihtmétique … (exceptions #XM SIMD floating point, #MF x87 floating point …).*

- ***SIGTRAP :*** *vu précédemment, principalement utilisé par les outils de debug. Il ne s'agit pas de défaut mais d'exception matérielles voulues.*

Prenons quelques exemples permettant d'illustrer quelques unes des exceptions précédemment présentées. :

- ***SIGBUS :*** *défaut d'alignement (activation matérielle nécessaire côté processeur)*

```c
/**
* @fn void main (void)
* @brief program entry point
*/
int main(int argc, char **argv) {
    int *pInt;
    char *pArea;

    // enable alignment checking
    #if defined(__GNUC__)
    # if defined(__i386__)
        // x86 architecture
        __asm__("pushf\norl $0x40000,(%esp)\npopf");
    # elif defined(__x86_64__)
         // x86_64 architecture
        __asm__("pushf\norl $0x40000,(%rsp)\npopf");
    # endif
    #endif

    // malloc() always provides aligned memory
    pArea = (char *) malloc(sizeof(int)+1);

    // increment the pointer by value different of modulo sizeof(*pInt)
    // making it misaligned
    pInt = (int *) (pArea+=3);                    // nok
    //pInt = (int *) (pArea+=sizeof(*pInt));      // ok

    // unaligned access (dereference pointer)
    *pInt = 51;

    return 0;
```

```
vmlinux@vmlinux: ~/Bureau/segfault
vmlinux@vmlinux:~/Bureau/segfault$ gcc buserror.c
vmlinux@vmlinux:~/Bureau/segfault$ ./a.out
Bus error (core dumped)
vmlinux@vmlinux:~/Bureau/segfault$
```

- ***SIGFPE :*** *erreur arithmétique, division par zéro (exception #MF)*

```c
/**
* @fn void main (void)
* @brief program entry point
*/
int main(int argc, char **argv) {
    volatile int value = 3, zero = 0.0;
    value /= zero;

return 0;
}
```

```
vmlinux@vmlinux: ~/Bureau/segfault
vmlinux@vmlinux:~/Bureau/segfault$ gcc divzero.c
vmlinux@vmlinux:~/Bureau/segfault$ ./a.out
Floating point exception (core dumped)
vmlinux@vmlinux:~/Bureau/segfault$
```

- ***SIGSEGV :*** *erreur de segmentation (probablement exception #PF)*

```c
/**
* @fn void main (void)
* @brief program entry point
*/
int main(int argc, char **argv) {
    volatile char* ptr; // uninitialized pointer

    printf("current stack area = 0x%x\n", (unsigned int) &ptr);
    printf("ptr point anywhere = 0x%x\n", (unsigned int) ptr);

    *ptr = 'H';

return 0;
}
```

```
vmlinux@vmlinux: ~/Bureau/segfault
vmlinux@vmlinux:~/Bureau/segfault$ gcc anywhere.c
vmlinux@vmlinux:~/Bureau/segfault$ ./a.out
current stack area = 0xbf985dbc
ptr point anywhere = 0xb77b6ff4
Segmentation fault (core dumped)
vmlinux@vmlinux:~/Bureau/segfault$
```

- ***SIGSEGV :*** *écriture en zone read-only (exception #GP)*

```c
/**
* @fn void main (void)
* @brief program entry point
*/
int main(int argc, char **argv) {
    char *str = "Hello World";

    *str = 'a';

return 0;
}
```

```
vmlinux@vmlinux: ~/Bureau/segfault
vmlinux@vmlinux:~/Bureau/segfault$ gcc readonly.c
vmlinux@vmlinux:~/Bureau/segfault$ ./a.out
Segmentation fault (core dumped)
vmlinux@vmlinux:~/Bureau/segfault$
```

```
Contents of section .rodata:
 8048498 03000000 01000200 48656c6c 6f20576f  ........Hello Wo
 80484a8 726c6400                             rld.
```

- ***SIGSEGV :*** *intéressons-nous au célèbre* ***stack overflow****. En mode réel, celui-ci est notamment détecté par l'exécution d'instructions des familles PUSH et POP capables de lever l'exception #SS (Stack Fault) en cas de dépassement de limite du Stack Segment. Dans les autres modes mémoire, il est en général détecté par exception matérielle #PF (Page Fault), la pile étant de taille multiple de la taille d'une page mémoire. L'instruction PUSH est également capable de lever l'exception #PF (hors mode réel).*

```
080483b4 <main>:
 80483b4:       55                      push   %ebp
 80483b5:       89 e5                   mov    %esp,%ebp
 80483b7:       83 e4 f0                and    $0xfffffff0,%esp
 80483ba:       e8 f5 ff ff ff          call   80483b4 <main>
 80483bf:       b8 00 00 00 00          mov    $0x0,%eax
 80483c4:       c9                      leave
 80483c5:       c3                      ret
```

```
/**
* @fn void main (void)
* @brief program entry point
*/
int main(void) {
    main();
return 0;
}
```

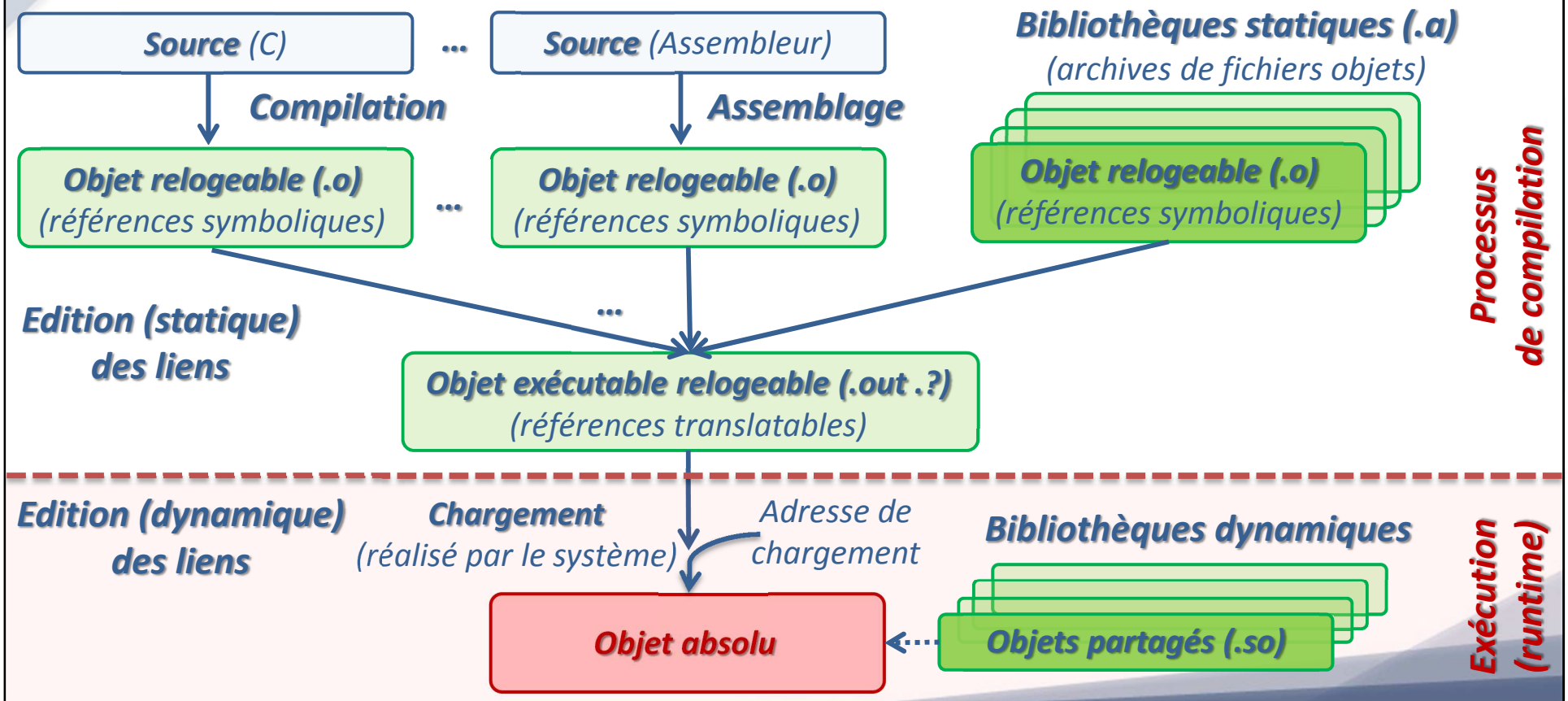
```
vmlinux@vmlinux: ~/Bureau/segfault
vmlinux@vmlinux:~/Bureau/segfault$ gcc stackoverflow.c
vmlinux@vmlinux:~/Bureau/segfault$ ./a.out
Segmentation fault (core dumped)
vmlinux@vmlinux:~/Bureau/segfault$
```

- ***Fichier ELF (Executable and Linkable Format) :***

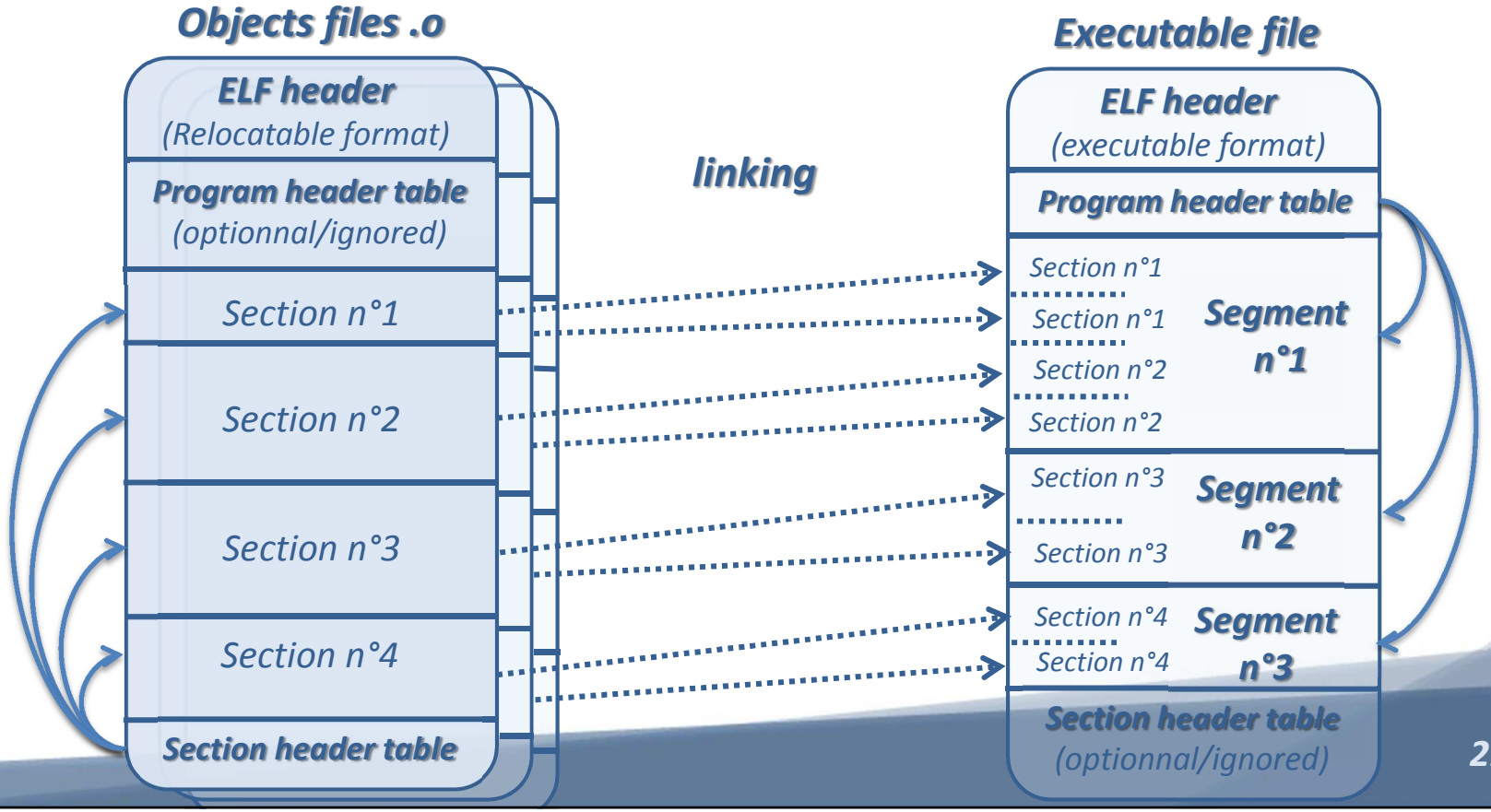
Le format de fichier ELF sert à l'enregistrement de programmes compilés (fichiers objets, exécutables, bibliothèques statiques et dynamiques, modules kernel). Prenons les principales extensions de fichiers compilés rencontrées sous Linux (tous des fichiers ELF) .o (object), .a (archive de .o), .so (shared object), .ko (kernel object). Ce format plus flexible unifie et remplace les anciens format a.out et COFF.

Le format ELF est extrêmement répandu sur systèmes UNIX-like (Linux, FreeBSD, Solaris, OpenBSD, Android …) ainsi que sur grand nombre d'autres plateformes (PS-2, PS-3, PSP, Wii, SymbianOS v9 …).

Avant de présenter le format de fichier ELF, présentons le cycle de vie d'un programme (en vert, fichiers ELF) :



*Un fichier ELF est toujours constitué d'une en-tête de fichier (cf. fichier **elf.h**), le reste de la structure diffère en fonction du type de fichier compilé (exécutable, bibliothèque partagée, objet …) :*



*Observons, en utilisant la commande **readelf** proposée avec **binutils** (ou la commande **objdump**), les en-têtes de fichiers ELF respectivement pour un fichier objet .o, exécutable et la bibliothèque standard du C, qui est un objet partagé .so :*

```
vmlinux@vmlinux:~/Desktop/segfault$ readelf -e objdump-minimal.o
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          340 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           40 (bytes)
  Number of section headers:         13
  Section header string table index: 10
```

```
vmlinux@vmlinux:~/Desktop/segfault$ readelf -e ./a.out
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x80483f0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          4428 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         30
  Section header string table index: 27
```

```
vmlinux@vmlinux:~/Desktop/segfault$ readelf -e /lib/i386-linux-gnu/libc.so.6
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x19630
  Start of program headers:          52 (bytes into file)
  Start of section headers:          1732720 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         10
  Size of section headers:           40 (bytes)
  Number of section headers:         35
  Section header string table index: 34
```

*Rappelons que pour les **langages compilés**, deux grands types d'allocations mémoire sont rencontrés pour la gestion des variables :*

- ***Allocations statiques :** Allocation mémoire à la compilation (compile-time). Prenons l'exemple des variables globales, locales qualifiées de static … Chaque chaîne de compilation C est très structurée, classe les variables statiques par famille et les range dans des sections spécifiques (sections présentent dans fichier ELF). Observons quelques-unes des principales sections :*
  - ***.bss :** variables statiques non-initialisées. Par définition initialisées à zéro au démarrage*
  - ***.data :** variables statiques initialisées*
  - ***.rodata :** variables statiques en lecture seule (read-only)*

Observons le contenu des sections d'un fichier objet élémentaire après compilation. Accès aux variables par adressage relatif aux adresses de base des sections :

```c
/**
* @file objdump-minimal.c
* @author
* @date novembre 2013
*/
#include <stdio.h>

char data[]="Bonjour le Monde\n";

/**
* @fn void main (void)
* @brief program entry point
*/
int main(int argc,char* argv[]){
    char* rodata="Hello World\n";

    printf("%s%s", data, rodata);

return 0;
}
```

```
vmlinux@vmlinux:~/Desktop/segfault$ gcc -c objdump-minimal.c
vmlinux@vmlinux:~/Desktop/segfault$ objdump -s objdump-minimal.o

objdump-minimal.o:     file format elf32-i386

Contents of section .text:
 0000 5589e583 e4f083ec 20c74424 1c000000  U.....  .D$....
 0010 00b80d00 00008b54 241c8954 2408c744  .......T$..T$..D
 0020 24040000 00008904 24e8fcff ffffb800  $.......$.......
 0030 000000c9 c3                          .....
Contents of section .data:
 0000 426f6e6a 6f757220 6c65204d 6f6e6465  Bonjour le Monde
 0010 0a00                                 ..
Contents of section .rodata:
 0000 48656c6c 6f20576f 726c640a 00257325  Hello World..%s%
 0010 7300                                 s.
```

Section .text : *binaire du programme*

Adresses relatives dans les sections *(représentation hexadécimale)*

Contenu des sections *(représentation hexadécimale)*

Contenu des sections *(représentation sous forme de caractères)*

- ***Allocations dynamiques :*** *Allocation mémoire à l'exécution (run-time). Vu par la suite.*
  - ***Gestion par la pile (ou stack) :*** *variables locales, paramètres, valeur de retour, adresse de retour et contexte d'exécution de fonction. Nous parlons également souvent d'allocation automatique.*
  - ***Gestion par le tas (ou heap) :*** *fonctions malloc, free et variantes.*

Observons l'en-tête contenant la table des sections pour le programme précédemment compilé (fichier objet).

```c
/**
* @file objdump-minimal.c
* @author
* @date novembre 2013
*/
#include <stdio.h>

char data[]="Bonjour le Monde\n";

/**
* @fn void main (void)
* @brief program entry point
*/
int main(int argc,char* argv[]){
    char* rodata="Hello World\n";

    printf("%s%s", data, rodata);

return 0;
}
```

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 000035 00  AX  0   0  4
  [ 2] .rel.text         REL             00000000 000444 000020 08     11   1  4
  [ 3] .data             PROGBITS        00000000 00006c 000012 00  WA  0   0  4
  [ 4] .bss              NOBITS          00000000 000080 000000 00  WA  0   0  4
  [ 5] .rodata           PROGBITS        00000000 000080 000012 00   A  0   0  1
  [ 6] .comment          PROGBITS        00000000 000092 00002b 01  MS  0   0  1
  [ 7] .note.GNU-stack   PROGBITS        00000000 0000bd 000000 00      0   0  1
  [ 8] .eh_frame         PROGBITS        00000000 0000c0 000038 00   A  0   0  4
  [ 9] .rel.eh_frame     REL             00000000 000464 000008 08     11   8  4
  [10] .shstrtab         STRTAB          00000000 0000f8 00005f 00      0   0  1
  [11] .symtab           SYMTAB          00000000 000360 0000c0 10     12   9  4
  [12] .strtab           STRTAB          00000000 000420 000024 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)

There are no section groups in this file.

There are no program headers in this file.
```

- ***Section .symtab :*** *cette section, nommée* ***table des symboles*** *est essentielle. La compilation est un processus dépendant du langage et de l'architecture du CPU mais indépendant du mapping mémoire. Les binaires compilés (fichiers objets) travaillent par références symboliques, la table des symboles lie les symboles à des adresses relatives vers différentes sections.*

```c
/**
* @file objdump-minimal.c
* @author
* @date novembre 2013
*/
#include <stdio.h>

char data[]="Bonjour le Monde\n";

/**
* @fn void main (void)
* @brief program entry point
*/
int main(int argc,char* argv[]){
    char* rodata="Hello World\n";

    printf("%s%s", data, rodata);

return 0;
}
```

```
Symbol table '.symtab' contains 12 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS objdump-minimal.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    1
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000     0 SECTION LOCAL  DEFAULT    5
     6: 00000000     0 SECTION LOCAL  DEFAULT    7
     7: 00000000     0 SECTION LOCAL  DEFAULT    8
     8: 00000000     0 SECTION LOCAL  DEFAULT    6
     9: 00000000    18 OBJECT  GLOBAL DEFAULT    3 data
    10: 00000000    53 FUNC    GLOBAL DEFAULT    1 main
    11: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
```

Section .data

Section .text

Adresse dans la section

Nous pouvons également rencontrer une table de relocation (liste de trous à compléter avec la table des symboles). A titre indicatif, les bibliothèques dynamiques sont liées à l'exécution par le chargeur du noyau au démarrage du programme. La section ***.plt (procedure linkage table)*** *effectue une redirection à l'exécution vers l'adresse absolue de la procédure cible (printf dans notre cas).*

```c
/**
* @file objdump-minimal.c
* @author
* @date novembre 2013
*/
#include <stdio.h>

char data[]="Bonjour le Monde\n";

/**
* @fn void main (void)
* @brief program entry point
*/
int main(int argc,char* argv[]){
    char* rodata="Hello World\n";

    printf("%s%s", data, rodata);

return 0;
}
```

```
080483e4 <main>:
main():
 80483e4:       55                      push   %ebp
 80483e5:       89 e5                   mov    %esp,%ebp
 80483e7:       83 e4 f0                and    $0xfffffff0,%esp
 80483ea:       83 ec 20                sub    $0x20,%esp
 80483ed:       c7 44 24 1c f0 84 04    movl   $0x80484f0,0x1c(%esp)
 80483f4:       08
 80483f5:       b8 fd 84 04 08          mov    $0x80484fd,%eax
 80483fa:       8b 54 24 1c             mov    0x1c(%esp),%edx
 80483fe:       89 54 24 08             mov    %edx,0x8(%esp)
 8048402:       c7 44 24 04 14 a0 04    movl   $0x804a014,0x4(%esp)
 8048409:       08
 804840a:       89 04 24                mov    %eax,(%esp)
 804840d:       e8 ee fe ff ff          call   8048300 <printf@plt>
 8048412:       b8 00 00 00 00          mov    $0x0,%eax
 8048417:       c9                      leave
 8048418:       c3                      ret
```

ENSICAEN
Computer Science

Merci de votre attention !