Systèmes d'Exploitation

TPs SecOS (https://github.com/agantet/secos-ng)

A. Gantet (inspiré de secos de S.Duverger !)

TLS-SEC 2023-2024

1 secos - contexte

2 secos - correction TP0 : Démarrage

3 secos - correction TP1 : Segmentation

4 secos - correction TP2 : Interruptions et exceptions

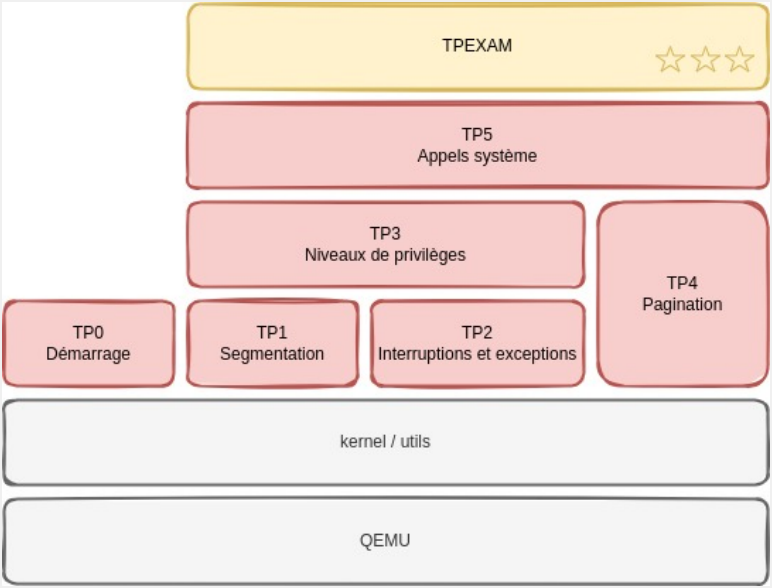
5 secos - correction TP3 : Mise en place de niveau de privilèges

6 secos - correction TP4

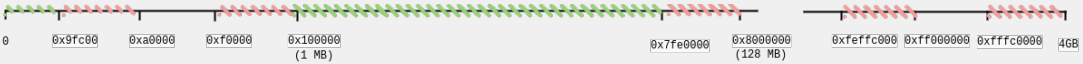
7 secos - correction TP5

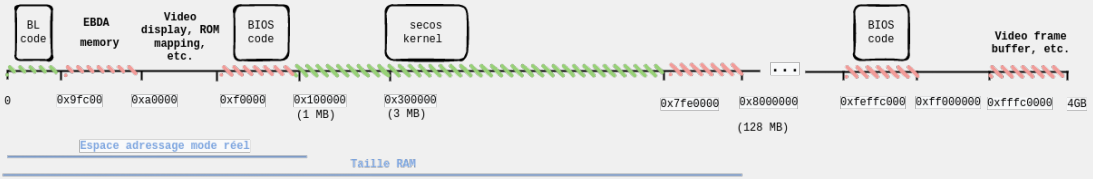
8 secos - bilan

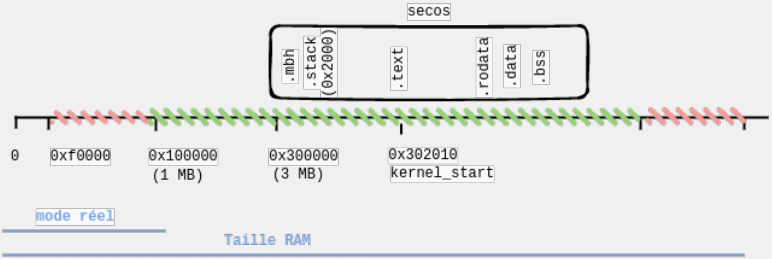
- Sous qemu
- ”BIOS” + ”Bootloader” exécutés
  - MMU et gestion des interruptions potentiellement pré-configurés
- CPU passé du mode réel au mode protégé
- secos chargé et exécuté (en ring 0)
  - code minimaliste pour le moment (print msg + halted)


TPEXAM
TP5
Appels système
TP3
Niveaux de privilèges
TP4
Pagination
TP0
Démarrage
TP1
Segmentation
TP2
Interruptions et exceptions
kernel / utils
QEMU

- Optionnel
- Jusqu'à 8 points bonus sur la note de l'examen final
- En binôme
- Date maximale de rendu : 31 décembre minuit
- Critères de notation
  - (/1) format demandé respecté (archive + patch git fonctionnel)
  - (/1) compilation fonctionnelle
  - (/3) exécution conforme au comportement attendu
  - (/2) sécurité / exactitude / beauté de la rédaction du code
  - (/1) clarté de la documentation


0
0x9fc00
0xa0000
0xf0000
0x100000
(1 MB)
0x7fe0000
0x8000000
(128 MB)
0xfeffc000
0xff000000
0xfffc0000
4GB


BL code
EBDA memory
Video display, ROM mapping, etc.
BIOS code
secos kernel
BIOS code
Video frame buffer, etc.
0
0x9fc00
0xa0000
0xf0000
0x100000
(1 MB)
0x300000
(3 MB)
0x7fe0000
0x8000000
(128 MB)
0xfeffc000
0xff000000
0xfffc0000
4GB
Espace adressage mode réel
Taille RAM


secos
.mbh
.stack (0x2000)
.text
.rodata
.data
.bss
0
0xf0000
0x100000
(1 MB)
0x300000
(3 MB)
0x302010
kernel_start
mode réel
Taille RAM

TPo - Question 3,4 - Utilisation de la mémoire physique à disposition

- Lecture / Ecriture dans une zone ”disponible”?

TP0 - Question 3,4 - Utilisation de la mémoire physique à disposition

- Lecture / Ecriture dans une zone ”disponible”? OK!

TP0 - Question 3,4 - Utilisation de la mémoire physique à disposition

- Lecture / Ecriture dans une zone ”disponible”? OK!
- Lecture / Ecriture dans une zone ”réservée”?

- Lecture / Ecriture dans une zone ”disponible” ? OK !
- Lecture / Ecriture dans une zone ”réservée” ? Lecture ok, écriture inefficace

TP0 - Question 3,4 - Utilisation de la mémoire physique à disposition

- Lecture / Ecriture dans une zone ”disponible” ? OK !
- Lecture / Ecriture dans une zone ”réservée” ? Lecture ok, écriture inefficace
- Lecture / Ecriture en dehors de la taille de la mémoire physique ?

- Lecture / Ecriture dans une zone ”disponible” ? OK !
- Lecture / Ecriture dans une zone ”réservée” ? Lecture ok, écriture inefficace
- Lecture / Ecriture en dehors de la taille de la mémoire physique ? Ne lève pas de faute – oh wait... une GDT serait-elle donc déjà en place ?...

- Configuration de grub inconnue et non maitrisée

- Configuration de grub inconnue et non maitrisée
- Nécessité de
  - Définir une table de descripteurs de segments (GDT, stockée en RAM) selon la configuration souhaitée
    - Première entrée nulle
    - Code, 32 bits RX, flat, indice 1
    - Données, 32 bits RW, flat, indice 2
  - Mettre à jour pour que la nouvelle configuration s'applique
    - GDTR pour référencer cette nouvelle table
    - mais aussi les sélecteurs de segments CS, DS, SS, ES, FS, GS

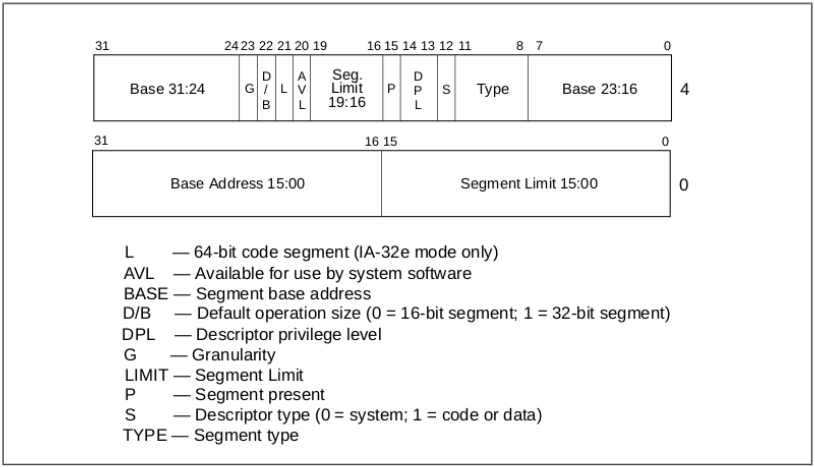


Figure 3-8. Segment Descriptor

Table 3-1. Code- and Data-Segment Types

Type Field					Descriptor Type	Description
Decimal	11	10 E	9 W	8 A		
0	0	0	0	0	Data	Read-Only
1	0	0	0	1	Data	Read-Only, accessed
2	0	0	1	0	Data	Read/Write
3	0	0	1	1	Data	Read/Write, accessed
4	0	1	0	0	Data	Read-Only, expand-down
5	0	1	0	1	Data	Read-Only, expand-down, accessed
6	0	1	1	0	Data	Read/Write, expand-down
7	0	1	1	1	Data	Read/Write, expand-down, accessed
		C	R	A		
8	1	0	0	0	Code	Execute-Only
9	1	0	0	1	Code	Execute-Only, accessed
10	1	0	1	0	Code	Execute/Read
11	1	0	1	1	Code	Execute/Read, accessed
12	1	1	0	0	Code	Execute-Only, conforming
13	1	1	0	1	Code	Execute-Only, conforming, accessed
14	1	1	1	0	Code	Execute/Read, conforming
15	1	1	1	1	Code	Execute/Read, conforming, accessed

```
seg_desc_t my_gdt[6];

my_gdt[0].raw = 0ULL;

my_gdt[1].limit_1 = 0xffff;    //:16;     /* bits 00-15 of the segment limit */
my_gdt[1].base_1 = 0x0000;     //:16;     /* bits 00-15 of the base address */
my_gdt[1].base_2 = 0x00;       //:8;      /* bits 16-23 of the base address */
my_gdt[1].type = 11;//Code,RX  //:4;      /* segment type */
my_gdt[1].s = 1;               //:1;      /* descriptor type */
my_gdt[1].dpl = 0; //ringo     //:2;      /* descriptor privilege level */
my_gdt[1].p = 1;               //:1;      /* segment present flag */
my_gdt[1].limit_2 = 0xf;       //:4;      /* bits 16-19 of the segment limit */
my_gdt[1].avl = 1;             //:1;      /* available for fun and profit */
my_gdt[1].l = 0; //32bits      //:1;      /* longmode */
my_gdt[1].d = 1;               //:1;      /* default length, depend on seg type */
my_gdt[1].g = 1;               //:1;      /* granularity */
my_gdt[1].base_3 = 0x00;       //:8;      /* bits 24-31 of the base address */
```

```
my_gdt[2].limit_1 = 0xffff;    //:16;     /* bits 00-15 of the segment limit */
my_gdt[2].base_1 = 0x0000;     //:16;     /* bits 00-15 of the base address */
my_gdt[2].base_2 = 0x00;       //:8;      /* bits 16-23 of the base address */
my_gdt[2].type = 3; //data,RW  //:4;      /* segment type */
my_gdt[2].s = 1;               //:1;      /* descriptor type */
my_gdt[2].dpl = 0; //ringo     //:2;      /* descriptor privilege level */
my_gdt[2].p = 1;               //:1;      /* segment present flag */
my_gdt[2].limit_2 = 0xf;       //:4;      /* bits 16-19 of the segment limit */
my_gdt[2].avl = 1;             //:1;      /* available for fun and profit */
my_gdt[2].l = 0; // 32 bits    //:1;      /* longmode */
my_gdt[2].d = 1;               //:1;      /* default length, depend on seg type */
my_gdt[2].g = 1;               //:1;      /* granularity */
my_gdt[2].base_3 = 0x00;       //:8;      /* bits 24-31 of the base address */

gdt_reg_t my_gdtr;
my_gdtr.addr = (long unsigned int)my_gdt;
my_gdtr.limit = sizeof(my_gdt) - 1;
set_gdtr(my_gdtr);
```

- Parce que la segmentation a surtout un intérêt quand elle n'est pas flat :)
- Définition d'un descripteur de segment court (32 octets)
- Chargement du sélecteur ES avec ce nouveau descripteur
- Utilisation de ES via memcpy qui utilise l'instruction REP MOVSB
  - Comportement attendu : #GP si la copie dépasse la taille du segment défini dans le descripteur, cf doc Intel :
  - *"#GP when : Exceeding the segment limit when accessing the CS, DS, ES, FS, or GS segments.*

MMU TP1 (Q9) - Un vrai segment, non flat

```
char  src[64];
char *dst = 0;
my_gdt[3].limit_1 = 0x20;      //:16;     /* bits 00-15 of the segment limit */
my_gdt[3].base_1 = 0x0000;     //:16;     /* bits 00-15 of the base address */
my_gdt[3].base_2 = 0x60;       //:8;      /* bits 16-23 of the base address */
my_gdt[3].type = 3; //data,RW  //:4;      /* segment type */
my_gdt[3].s = 1;               //:1;      /* descriptor type */
my_gdt[3].dpl = 0; //ringo     //:2;      /* descriptor privilege level */
my_gdt[3].p = 1;               //:1;      /* segment present flag */
my_gdt[3].limit_2 = 0x0;       //:4;      /* bits 16-19 of the segment limit */
my_gdt[3].avl = 1;             //:1;      /* available for fun and profit */
my_gdt[3].l = 0; // 32 bits    //:1;      /* longmode */
my_gdt[3].d = 1;               //:1;      /* default length, depend on seg type */
my_gdt[3].g = 0;               //:1;      /* granularity */
my_gdt[3].base_3 = 0x00;       //:8;      /* bits 24-31 of the base address */

seg_sel_t my_es;
my_es.index = 3;
my_es.ti = 0;
my_es.rpl = 0;
set_es(my_es);

_memcpy8(dst, src, 64); // _memcpy8(dst, src, 32);
```

- ***"secos est parti en boucle infini"***
  1. C'est sûrement qu'il y a une erreur de programmation,
  2. que le CPU lève une exception, mais que le handler n'est pas encore bien configuré
  3. que le CPU lève donc une nouvelle exception, mais que le handler n'est toujours pas bien configuré
  4. que le CPU lève encore une faute, et quand triple fault –> reboot et repart pour un tour
- ***"au moment du chargement de ES, le CPU lève une #GP"***
  - C'est sûrement dû à une mauvaise définition du descripteur de segment, cf. doc Intel :
    - *#GP when : "Loading the SS, DS, ES, FS, or GS register with a segment selector for a system segment."*
    - *#GP when : "Loading the DS, ES, FS, or GS register with a segment selector for an execute-only code segment."*
- ***"j'ai bien fait mon travail, mais la #GP attendue pour le memcpy 64 n'est pas au rendez-vous"***
  - C'est sûrement qu'on utilise un CPU émulé par QEMU et que QEMU a mal reproduit le comportement réel des CPU Intel :/ (d'où le conseil d'utilisation de KVM)

TP1 (Q12,Q13) - tentative de démarrage de code ring 3

But : démarrer un programme en ring 3

- Configurer une GDT avec des nouveaux segments, avec des DPL ring 3
- Comment tester si le passage au ring 3 a fonctionné correctement ?

But : démarrer un programme en ring 3

- Configurer une GDT avec des nouveaux segments, avec des DPL ring 3
- Comment tester si le passage au ring 3 a fonctionné correctement ?
  - Doc cause #GP : "Attempting to execute a privileged instruction when the CPL is not equal to 0"
  - exécution d'une instruction privilégiée : si GP, on a bien changé de ring :)
  - Exemple : `mov %eax, cr0`

- Mise à jour des sélecteurs de données (DS, ED, FS, GS) : trivial
- Mise à jour des sélecteurs SS et CS ?
  - Avec un simple MOV ?

TP1 (Q12,Q13) - tentative de démarrage de code ring 3

- Mise à jour des sélecteurs de données (DS, ED, FS, GS) : trivial
- Mise à jour des sélecteurs SS et CS ?
  - Avec un simple MOV ? Pas possible :(
    - Loading the SS register with a segment selector for a read-only segment (unless the selector comes from a TSS during a task switch, in which case an invalid-TSS exception occurs)
    - Loading the SS, DS, ES, FS, or GS register with a segment selector for a system segment
    - Loading the SS register with the segment selector of an executable segment or a null segment selector
  - Via un farjump ?

TP1 (Q12,Q13) - tentative de démarrage de code ring 3

- Mise à jour des sélecteurs de données (DS, ED, FS, GS) : trivial
- Mise à jour des sélecteurs SS et CS ?
  - Avec un simple MOV ? Pas possible :(
    - Loading the SS register with a segment selector for a read-only segment (unless the selector comes from a TSS during a task switch, in which case an invalid-TSS exception occurs)
    - Loading the SS, DS, ES, FS, or GS register with a segment selector for a system segment
    - Loading the SS register with the segment selector of an executable segment or a null segment selector
  - Via un farjump ? Interdit...

TP1 (Q12,Q13) - tentative de démarrage de code ring 3

- Mise à jour des sélecteurs de données (DS, ED, FS, GS) : trivial
- Mise à jour des sélecteurs SS et CS ?
  - Avec un simple MOV ? Pas possible :(
    - Loading the SS register with a segment selector for a read-only segment (unless the selector comes from a TSS during a task switch, in which case an invalid-TSS exception occurs)
    - Loading the SS, DS, ES, FS, or GS register with a segment selector for a system segment
    - Loading the SS register with the segment selector of an executable segment or a null segment selector
  - Via un farjump ? Interdit...

La suite aux prochains TPs :)

Niveau matériel, on a

- Une table de descripteurs d'interruption (IDT), pointeurs vers routine à exécuter
- le registre CPU IDTR à renseigner avec l'adresse de cette table
- Des informations stockées sur la pile lors de l'arrivée d'une int.
  - EFLAGS, CS, EIP, Error Code seulement si pas de changement de niveau de privilèges
  - ESP et SS en supplément en cas de changement de niveau

L'OS doit

- Implémenter des routines de traitement d'interruption
- Les référencer correctement dans la table de descripteurs d'interruption
- Configurer le registre IDTR

Niveau matériel, on a

- Une table de descripteurs d'interruption (IDT), pointeurs vers routine à exécuter
- le registre CPU IDTR à renseigner avec l'adresse de cette table
- Des informations stockées sur la pile lors de l'arrivée d'une int.
  - EFLAGS, CS, EIP, Error Code seulement si pas de changement de niveau de privilèges
  - ESP et SS en supplément en cas de changement de niveau

L'OS doit

- Implémenter des routines de traitement d'interruption
- Les référencer correctement dans la table de descripteurs d'interruption
- Configurer le registre IDTR

-> 3 exceptions déjà en partie gérées dans secos (NMI, PF, GP), le reste est à coder !

- But : s'entrainer en implémentant la routine de #BP et la référencer dans l'IDT
- La tester via l'instruction INT3

- Ecriture d'une fonction en C
- Mise à jour de l'IDT avec l'adresse cette fonction

Problème ?

- Ecriture d'une fonction en C
- Mise à jour de l'IDT avec l'adresse cette fonction

Problème ?

```
0030401e <bp_handler>:
  30401e:       55                      push   %ebp
  30401f:       89 e5                   mov    %esp,%ebp
  304021:       83 ec 08                sub    $0x8,%esp
  304024:       83 ec 0c                sub    $0xc,%esp
  304027:       68 f6 48 30 00          push   $0x3048f6
  30402c:       e8 a8 f0 ff ff          call   3030d9 <printf>
  304031:       83 c4 10                add    $0x10,%esp
  304034:       90                      nop
  304035:       c9                      leave
  304036:       c3                      ret

00304037 <bp_trigger>:
  304037:       55                      push   %ebp
  304038:       89 e5                   mov    %esp,%ebp
  30403a:       cc                      int3
  30403b:       90                      nop
  30403c:       5d                      pop    %ebp
  30403d:       c3                      ret
```

- CALL/RET vs INT/IRET
- Ici : INT/RET : dépilement d'adresse de retour alors que la pile avait été remplie par l'arrivée d'une interruption (donc avec EFLAGS, CS, EIP, Error Code) et ESP pointe donc sur un code d'erreur, pas sur un

Problème ?

- CALL/RET vs INT/IRET
- Ici : INT/RET :
  - dépilement d'adresse de retour alors que la pile avait été remplie par l'arrivée d'une interruption
    - Contenait donc EFLAGS, CS, EIP, Error Code
  - chargement de EIP avec la valeur "Error Code" (0x4c)
  - Erreur #UD (invalid opcode)
  - secos panic car la gestion #UD n'est pas encore implémenté !

Problème ?

- CALL/RET vs INT/IRET
- Ici : INT/RET :
  - dépilement d'adresse de retour alors que la pile avait été remplie par l'arrivée d'une interruption
    - Contenait donc EFLAGS, CS, EIP, Error Code
  - chargement de EIP avec la valeur "Error Code" (0x4c)
  - Erreur #UD (invalid opcode)
  - secos panic car la gestion #UD n'est pas encore implémenté !

-> Forcer l'utilisation de IRET

```
void bp_handler()
{
   debug("#BP furtif\n");
   asm volatile ("leave ; iret"); //leave: Set ESP to EBP, then pop EBP
}
```

La routine de traitement doit

- Sauvegarder tous les GPRs
- Traiter l'interruption comme on le souhaite
- Restaurer les registres sauvegardés
- Utiliser IRET pour retourner à l'exécution précédente

Moralité : préférer l'assembleur pour coder une routine d'INT

But : démarrer un programme en ring 3

- Configurer une GDT avec des nouveaux segments, avec des DPL ring 3
- Comment tester si le passage au ring 3 a fonctionné correctement ?

But : démarrer un programme en ring 3

- Configurer une GDT avec des nouveaux segments, avec des DPL ring 3
- Comment tester si le passage au ring 3 a fonctionné correctement ?
  - Doc cause #GP : "Attempting to execute a privileged instruction when the CPL is not equal to 0"
  - execution d'une instruction privilégiée : si GP, on a bien changé de ring :)
  - Exemple : `mov %eax, cr0`

- Mise à jour des sélecteurs de données (DS, ED, FS, GS) : trivial
- Mise à jour des sélecteurs SS et CS ?
  - Avec un simple MOV ?

- Mise à jour des sélecteurs de données (DS, ED, FS, GS) : trivial
- Mise à jour des sélecteurs SS et CS ?
  - Avec un simple MOV ? Pas possible :(
    - Loading the SS register with a segment selector for a read-only segment (unless the selector comes from a TSS during a task switch, in which case an invalid-TSS exception occurs)
    - Loading the SS, DS, ES, FS, or GS register with a segment selector for a system segment
    - Loading the SS register with the segment selector of an executable segment or a null segment selector
  - Via un farjump ?

- Mise à jour des sélecteurs de données (DS, ED, FS, GS) : trivial
- Mise à jour des sélecteurs SS et CS ?
  - Avec un simple MOV ? Pas possible :(
    - Loading the SS register with a segment selector for a read-only segment (unless the selector comes from a TSS during a task switch, in which case an invalid-TSS exception occurs)
    - Loading the SS, DS, ES, FS, or GS register with a segment selector for a system segment
    - Loading the SS register with the segment selector of an executable segment or a null segment selector
  - Via un farjump ? Interdit...

IRET sait traiter les changements de niveaux de privilèges si la pile est correctement remplie. Idée :

- Empiler (PUSH/PUSHF) SS, puis ESP, puis EFLAGS, CS, puis EIP (comme l'aurait fait un CPU lors de l'arrivée d'une faute) avec
  - 'CS' : un sélecteur de descripteur de segment de code ring 3
  - EIP : l'adresse de début du code ring 3 qu'on cherche à exécuter !
- Exécuter un IRET

IRET sait traiter les changements de niveaux de privilèges si la pile est correctement remplie. Idée :

- Empiler (PUSH/PUSHF) SS, puis ESP, puis EFLAGS, CS, puis EIP (comme l'aurait fait un CPU lors de l'arrivée d'une faute) avec
  - 'CS' : un sélecteur de descripteur de segment de code ring 3
  - EIP : l'adresse de début du code ring 3 qu'on cherche à exécuter !
- Exécuter un IRET

Résultat ?

IRET sait traiter les changements de niveaux de privilèges si la pile est correctement remplie. Idée :

- Empiler (PUSH/PUSHF) SS, puis ESP, puis EFLAGS, CS, puis EIP (comme l'aurait fait un CPU lors de l'arrivée d'une faute) avec
  - 'CS' : un sélecteur de descripteur de segment de code ring 3
  - EIP : l'adresse de début du code ring 3 qu'on cherche à exécuter !
- Exécuter un IRET

Résultat ? TSS invalide au moment du traitement de la #GP attendue...

```
qemu: fatal: invalid tss type
EAX=003042b0 EBX=0002be40 ECX=00304bc0 EDX=00000022
ESI=0002bfc2 EDI=0002bfc3 EBP=00301fe4 ESP=00301fe4
EIP=003042b3 EFL=00000082 [--S----] CPL=3 II=0 A20=1 SMM=0 HLT=0
ES =0023 00000000 ffffffff 00cff300 DPL=3 DS   [-WA]
CS =001b 00000000 ffffffff 00cffa00 DPL=3 CS32 [-R-]
SS =0023 00000000 ffffffff 00cff300 DPL=3 DS   [-WA]
DS =0023 00000000 ffffffff 00cff300 DPL=3 DS   [-WA]
FS =0023 00000000 ffffffff 00cff300 DPL=3 DS   [-WA]
GS =0023 00000000 ffffffff 00cff300 DPL=3 DS   [-WA]
LDT=0000 00000000 0000ffff 00008200 DPL=0 LDT
TR =0000 00000000 0000ffff 00008b00 DPL=0 TSS32-busy
GDT=     003057c0 0000002f
IDT=     00304fc0 000007ff
```

TP3 - IRET à la rescousse

IRET sait traiter les changements de niveaux de privilèges si la pile est correctement remplie.

```
set_ds(d3_sel);
set_es(d3_sel);
set_fs(d3_sel);
set_gs(d3_sel);

TSS.so.esp = get_ebp();
TSS.so.ss  = do_sel;
tss_dsc(&GDT[ts_idx], (offset_t)&TSS);
set_tr(ts_sel);

asm volatile (
"push %0    \n" // ss
"push %%ebp \n" // esp
"pushf      \n" // eflags
"push %1    \n" // cs
"push %2    \n" // eip
"iret"
::
"i"(d3_sel),
"i"(c3_sel),
"r"(&userland)
);
```

TP4 - rappel des registres clé de la pagination

- CR3
- CR0 (bit 31 : paging)

Q1* : A l'aide de la fonction get_cr3(), afficher la valeur courante du registre CR3 dans tp.c.

Q1* : A l'aide de la fonction get_cr3(), afficher la valeur courante du registre CR3 dans tp.c.

Difficulté rencontrable : le type renvoyé par `get_cr3` et comment le stocker dans le type `cr3_reg_t`

Q1* : A l'aide de la fonction get_cr3(), afficher la valeur courante du registre CR3 dans tp.c.

Difficulté rencontrable : le type renvoyé par `get_cr3` et comment le stocker dans le type `cr3_reg_t`
Exemple de solution :

```
// Q1
cr3_reg_t cr3 = {.raw = get_cr3()};
debug("CR3 = 0x%x\n", (unsigned int) cr3.raw);
// end Q1
```

Résultat :

```
CR3 = 0x0
```

Q2, Q3, Q4

- Q2* : Allouer un PGD de type (pde32_t*) à l'adresse physique 0x600000 et mettre à jour CR3 avec cette adresse.
- Q3* : Modifier le registre CR0 de sorte à activer la pagination dans tp.c. Que se passe-t-il ? Pourquoi ?
- Q4* : Un certain nombre de choses restent à configurer avant l'activation de la pagination. Comme pour le PGD, allouer également une PTB de type (pte32_t*) à l'adresse 0x601000.

Q2, Q3, Q4

- Q2* : Allouer un PGD de type (pde32_t*) à l'adresse physique 0x600000 et mettre à jour CR3 avec cette adresse.
- Q3* : Modifier le registre CR0 de sorte à activer la pagination dans tp.c. Que se passe-t-il ? Pourquoi ?
- Q4* : Un certain nombre de choses restent à configurer avant l'activation de la pagination. Comme pour le PGD, allouer également une PTB de type (pte32_t*) à l'adresse 0x601000.

```
// Q2
// 0x600000 = 0x001<<(10+12) | 0x300<<10 | 0x000
// 0x001 (10bits) -- 0x300 (10bits) -- 0x000 (12 bits)
// 00 0000 0001   -- 10 0000 0000   -- 0000 0000 0000
pde32_t *pgd = (pde32_t*)0x600000;
set_cr3((uint32_t)pgd);
// Q3
// uint32_t cr0 = get_cr0();
// set_cr0(cr0|CR0_PG);
// encore un peu tôt d'activer la pagination à ce stade :)
// notamment car le pgd est vide !
// Q4
pte32_t *ptb = (pte32_t*)0x601000;
// end Q4
```

- Q5* : Le but va être maintenant d'initialiser la mémoire virtuelle en "identity mapping"
- Q6 : Une fois la pagination activée, essayer d'afficher le contenu d'une entrée de votre PTB. Que se passe-t-il ? Trouver la solution pour être capable de modifier les entrées de votre PTB une fois la pagination activée.

Méthode :

- Trouver la plus grande adresse physique utilisée $addr_m$
- En déduire le nombre d_m d'entrées de PGD suffisante pour identity-mapper l'ensemble des adresses entre 0 et ce max : $d_m = addr_m >> (10 + 12)$
- Pour chaque entrée i de PGD, remplir des PTB de sorte que $ptb_d[t].addr = ((d << 10)|t) << 12$
- Activer ensuite la pagination dans cr0

Exemple de mise en oeuvre :

```
pde32_t *pgd  = (pde32_t*)0x600000;
pte32_t *ptb0 = (pte32_t*)0x601000;
pte32_t *ptb1 = (pte32_t*)0x602000;
memset((void*)pgd, 0, PAGE_SIZE);
set_cr3((uint32_t)pgd);

for(int i=0;i<1024;i++) {
  pg_set_entry(&ptb0[i], PG_KRN|PG_RW, 0<<10 + i);
}
pg_set_entry(&pgd[0], PG_KRN|PG_RW, page_nr(ptb0));

for(int i=0;i<1024;i++) {
  pg_set_entry(&ptb1[i], PG_KRN|PG_RW, 1<<10 + i);
}
pg_set_entry(&pgd[1], PG_KRN|PG_RW, page_nr(ptb1));

uint32_t cr0 = get_cr0();
set_cr0(cr0|CR0_PG);
```

Q7* : Avant d'activer la pagination, on souhaiterait faire en sorte que l'adresse virtuelle 0xC0000000 permette de modifier votre PGD après activation de la pagination. Comment le réaliser ?

Q7* : Avant d'activer la pagination, on souhaiterait faire en sorte que l'adresse virtuelle 0xc0000000 permette de modifier votre PGD après activation de la pagination. Comment le réaliser ?

But : $addr_p$ 0xc0000000 == $addr_p$ 0x600000 == 0x600000

Q7* : Avant d'activer la pagination, on souhaiterait faire en sorte que l'adresse virtuelle 0xc0000000 permette de modifier votre PGD après activation de la pagination. Comment le réaliser ?

But : $addr_p$ 0xc0000000 == $addr_p$ 0x600000 == 0x600000
Méthode

- Déterminer l'index de PGD à mettre à jour
- Déterminer l'index de PTB à mettre à jour
- Créer les entrées correspondantes et mettre à jour la base de PTB à 0x600000

Exemple de résolution

```
pte32_t  *ptb3    = (pte32_t*)0x603000;
uint32_t *target  = (uint32_t*)0xc0000000;
int      pgd_idx = pd32_idx(target);
int      ptb_idx = pt32_idx(target);
debug("%d %d\n", pgd_idx, ptb_idx);
/**/
memset((void*)ptb3, 0, PAGE_SIZE);
pg_set_entry(&ptb3[ptb_idx], PG_KRN|PG_RW, page_nr(pgd));
pg_set_entry(&pgd[pgd_idx], PG_KRN|PG_RW, page_nr(ptb3));
/**/
debug("PGD[0] = 0x%x | target = 0x%x\n",
      (unsigned int) pgd[0].raw, (unsigned int) *target);

768 0
PGD[0] = 0x601023 | target = 0x601023
```

Q8 : Faire en sorte que les adresses virtuelles 0x700000 et 0x7ff000 mappent l'adresse physique 0x2000. Affichez la chaîne de caractères à ces adresses virtuelles.

Q8 : Faire en sorte que les adresses virtuelles 0x700000 et 0x7ff000 mappent l'adresse physique 0x2000. Affichez la chaîne de caractères à ces adresses virtuelles.

Exemple de résolution

```
char *v1 = (char*)0x700000; // 7 memoire partagee
  char *v2 = (char*)0x7ff000;
  ptb_idx = pt32_idx(v1);
  pg_set_entry(&ptb2[ptb_idx], PG_KRN|PG_RW, 2);
  ptb_idx = pt32_idx(v2);
  pg_set_entry(&ptb2[ptb_idx], PG_KRN|PG_RW, 2);
  debug("%p = %s | %p = %s\n", v1, v1, v2, v2);

  0x700000 = /kernel.elf | 0x7ff000 = /kernel.elf
```

Q9 : Effacer la première entrée du PGD. Que constatez-vous ? Expliquez pourquoi ?

Q9 : Effacer la première entrée du PGD. Que constatez-vous ? Expliquez pourquoi ?

- A priori : supprime le mapping configuré pour toutes les adresses virtuelles entre 0 et $0x3ff000$
- devrait lever une faute, à la prochaine traduction d'adresse (prochaine EIP rencontrée par exemple)

Q9 : Effacer la première entrée du PGD. Que constatez-vous ? Expliquez pourquoi ?

- A priori : supprime le mapping configuré pour toutes les adresses virtuelles entre 0 et 0x3ff000
- devrait lever une faute, à la prochaine traduction d'adresse (prochaine EIP rencontrée par exemple)
- Mais attention aux TLB ! qui peuvent avoir gardé en mémoire les traductions précédemment effectuées !
  - Invalidable par l'instruction FLUSH
  - Invalidable par rechargement de CR3

Q1* : Un squelette de fonction userland est fourni dans tp.c. Reprendre le code du TP 3 pour modifier tp() de manière à démarrer du code ring 3.

```
init_gdt();
set_ds(d3_sel);
set_es(d3_sel);
set_fs(d3_sel);
set_gs(d3_sel);
TSS.s0.esp = get_ebp();
TSS.s0.ss  = d0_sel;
tss_dsc(&GDT[ts_idx], (offset_t)&TSS);
set_tr(ts_sel);
```

Q2 : En s'inspirant du TP2, mettre à jour les offsets du descripteur 48 de l'IDT pour installer la fonction syscall_isr() comme gestionnaire de l'interruption 48.

```
int_desc_t *dsc;
idt_reg_t  idtr;
get_idtr(idtr);
dsc = &idtr.desc[48];
// kernel syscall handler configuration instead of the previous handler
dsc->offset_1 = (uint16_t)((uint32_t)syscall_isr);
dsc->offset_2 = (uint16_t)(((uint32_t)syscall_isr)>>16);
```

Q3* : Pourquoi observe-t-on une #GP ? Corriger le problème de sorte qu'il soit autorisé d'appeler l'interruption "48" avec un RPL à 3.

```
void userland() {
   uint32_t arg =  0x2023;
   asm volatile ("int $48"::"a"(arg));
   while(1);
}
tp() {
...
uint32_t   ustack = 0x600000;
asm volatile (
  "push %0 \n" // ss
  "push %1 \n" // esp pour du ring 3 !
  "pushf   \n" // eflags
  "push %2 \n" // cs
  "push %3 \n" // eip
  "iret"
  ::
   "i"(d3_sel),
   "m"(ustack),
   "i"(c3_sel),
   "r"(&userland)
  );
```

Lève une faute CPU car pour le moment le RPL du descripteur de l'entrée 48 dans l'IDT est à 0 !
Solution : rajouter un `dsc->dpl = 3;` au moment de la configuration de l'IDT

Q4* : Modifier la fonction syscall_handler() pour qu'elle affiche une chaîne de caractères dont l'adresse se trouve dans le registre "ESI". Nous venons de créer un appel système permettant d'afficher un message à l'écran et prenant son argument via "ESI". Essayer cet appel système depuis votre fonction userland().

```
void __regparm__(1) syscall_handler(int_ctx_t *ctx) {
   debug("SYSCALL eax = 0x%x\n", (unsigned int) ctx->gpr.eax.raw);
   // Q4
   debug("print syscall: %s", (char *)ctx->gpr.esi.raw);
   // end Q4
}
```

TP5 - Q5 Faille de sécurité dans l'implémentation de ce handler ?

Q5 : Quel problème de sécurité y a-t-il à l'implémentation de syscall_handler() ? Essayez de pirater ce service, depuis userland (), afin de lire de la mémoire du noyau. Modifier le code de syscall_handler pour corriger ce problème.

Q5 : Quel problème de sécurité y a-t-il à l'implémentation de syscall_handler() ? Essayez de pirater ce service, depuis userland (), afin de lire de la mémoire du noyau. Modifier le code de syscall_handler pour corriger ce problème.

Rappel du contexte

- Application s'exécutant en ring 3
- Pouvant utiliser l'appel système 48 pour lire ce qui est stocké à une adresse arbitraire qu'elle choisit en paramètre

Que pourrait-il mal se passer ?...

Q5 : Quel problème de sécurité y a-t-il à l'implémentation de syscall_handler() ? Essayez de pirater ce service, depuis userland (), afin de lire de la mémoire du noyau. Modifier le code de syscall_handler pour corriger ce problème.

Rappel du contexte

- Application s'exécutant en ring 3
- Pouvant utiliser l'appel système 48 pour lire ce qui est stocké à une adresse arbitraire qu'elle choisit en paramètre

Que pourrait-il mal se passer ?...
-> Rien n'empêche l'application de demander à lire n'importe quelle adresse, dont la mémoire du noyau !

Affichage d'une zone de mémoire qui nous intéresse...

```
void userland() {
    asm volatile ("int $48"::"S"(0x304935));
    // will print secos-xxxx-xxxx !! (in the kernel memory !)
    while(1);
}
```

Affichage d'une zone de mémoire qui nous intéresse...

```
void userland() {
    asm volatile ("int $48"::"S"(0x304935));
    // will print secos-xxxx-xxxx !! (in the kernel memory !)
    while(1);
}
```

Idée de correctif : rajouter une vérification sur la valeur de `ctx->gpr.esi.raw` au début du handler, et n'afficher le résultat que dans le cas où l'adresse demandée appartient bien à l'espace d'adressage de la tâche appelante !

Affichage d'une zone de mémoire qui nous intéresse...

```
void userland() {
    asm volatile ("int $48"::"S"(0x304935));
    // will print secos-xxxx-xxxx !! (in the kernel memory !)
    while(1);
}
```

Idée de correctif : rajouter une vérification sur la valeur de `ctx->gpr.esi.raw` au début du handler, et n'afficher le résultat que dans le cas où l'adresse demandée appartient bien à l'espace d'adressage de la tâche appelante !
Moralité : code exposé par le noyau via des appels système à garder dans le collimateur niveau sécurité ! Le moindre oubli de vérification peut être fatal...

- TPo : découverte de la mémoire physique à disposition

TP0 à 5

- TP0 : découverte de la mémoire physique à disposition
- TP1 : Configuration de la segmentation, obligatoire en mode protégé
  - Mode flat le plus utilisé

TP0 à 5

- TP0 : découverte de la mémoire physique à disposition
- TP1 : Configuration de la segmentation, obligatoire en mode protégé
  - Mode flat le plus utilisé
- TP2 : Configuration de la gestion des interruptions
  - Dont format attendu d'un code de handler (iret)
  - Dont implémentation de handlers d'exception comme GP

- TP0 : découverte de la mémoire physique à disposition
- TP1 : Configuration de la segmentation, obligatoire en mode protégé
  - Mode flat le plus utilisé
- TP2 : Configuration de la gestion des interruptions
  - Dont format attendu d'un code de handler (iret)
  - Dont implémentation de handlers d'exception comme GP
- TP3 : Apprentissage de démarrage de la première tâche ring 3 après que le noyau ait fini de configurer IDT et GDT
  - Avec l'astuce du "fake iret" ! (limitation archi x86)

- TP0 : découverte de la mémoire physique à disposition
- TP1 : Configuration de la segmentation, obligatoire en mode protégé
  - Mode flat le plus utilisé
- TP2 : Configuration de la gestion des interruptions
  - Dont format attendu d'un code de handler (iret)
  - Dont implémentation de handlers d'exception comme GP
- TP3 : Apprentissage de démarrage de la première tâche ring 3 après que le noyau ait fini de configurer IDT et GDT
  - Avec l'astuce du "fake iret" ! (limitation archi x86)
- TP4 : Configuration de la pagination
  - Dont mise en place de l'identity mapping pour le bon transfert de l'ensemble des objets du monde non-paginé vers le monde paginé
  - Dont exemple de mise en place de zone de mémoire partagée

- TP0 : découverte de la mémoire physique à disposition
- TP1 : Configuration de la segmentation, obligatoire en mode protégé
  - Mode flat le plus utilisé
- TP2 : Configuration de la gestion des interruptions
  - Dont format attendu d'un code de handler (iret)
  - Dont implémentation de handlers d'exception comme GP
- TP3 : Apprentissage de démarrage de la première tâche ring 3 après que le noyau ait fini de configurer IDT et GDT
  - Avec l'astuce du "fake iret" ! (limitation archi x86)
- TP4 : Configuration de la pagination
  - Dont mise en place de l'identity mapping pour le bon transfert de l'ensemble des objets du monde non-paginé vers le monde paginé
  - Dont exemple de mise en place de zone de mémoire partagée
- TP5 : Rédaction de routine d'appel système
  - Proche TP2 mais pour une interruption au lieu d'une exception

- TP0 : découverte de la mémoire physique à disposition
- TP1 : Configuration de la segmentation, obligatoire en mode protégé
  - Mode flat le plus utilisé
- TP2 : Configuration de la gestion des interruptions
  - Dont format attendu d'un code de handler (iret)
  - Dont implémentation de handlers d'exception comme GP
- TP3 : Apprentissage de démarrage de la première tâche ring 3 après que le noyau ait fini de configurer IDT et GDT
  - Avec l'astuce du "fake iret" ! (limitation archi x86)
- TP4 : Configuration de la pagination
  - Dont mise en place de l'identity mapping pour le bon transfert de l'ensemble des objets du monde non-paginé vers le monde paginé
  - Dont exemple de mise en place de zone de mémoire partagée
- TP5 : Rédaction de routine d'appel système
  - Proche TP2 mais pour une interruption au lieu d'une exception
- TP Exam ? Jusqu'à 8 points bonus

- Optionnel
- Jusqu'à 8 points bonus sur la note de l'examen final
- En binôme
- Date maximale de rendu : 31 décembre minuit
- Enoncé dans le README sur github, comme pour les autres TP
- Critères de notation
  - (/1) format demandé respecté (archive + patch git fonctionnel)
  - (/1) compilation fonctionnelle
  - (/3) exécution conforme au comportement attendu
  - (/2) sécurité / exactitude / beauté de la rédaction du code
  - (/1) clarté de la documentation

- Hint 1 : n'oubliez pas de vous servir du mode debug de qemu, bien pratique pour comprendre ce qu'il faut corriger pour avancer !
- Hint 2 : explorez les fonctionnalités déjà implémentées de `secos-ng/include` ou de `secos-ng/core` dans vos implémentations pour gagner du temps :)