

Systèmes d'Exploitation

*concepts et implémentation appliqués
à l'architecture Intel x86*

Stéphane Duverger

Airbus
Toulouse, FRANCE

TLS-SEC

Introduction

La phase de démarrage

Les modes opératoires

La segmentation

Les niveaux de privilèges

Les interruptions et exceptions

La pagination

Les composants d'un OS

Conclusion

Introduction

La phase de démarrage

Les modes opératoires

La segmentation

Les niveaux de privilèges

Les interruptions et exceptions

La pagination

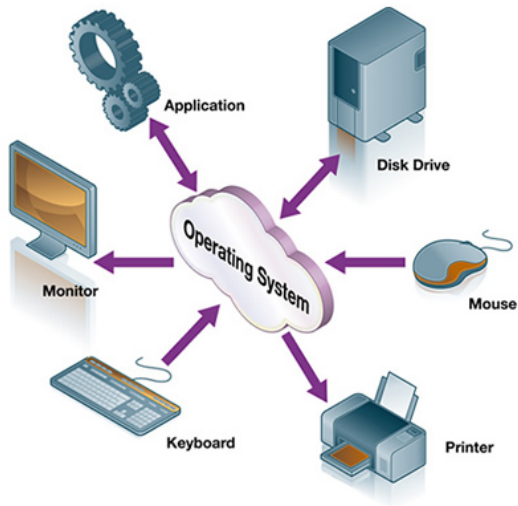
Les composants d'un OS

Conclusion

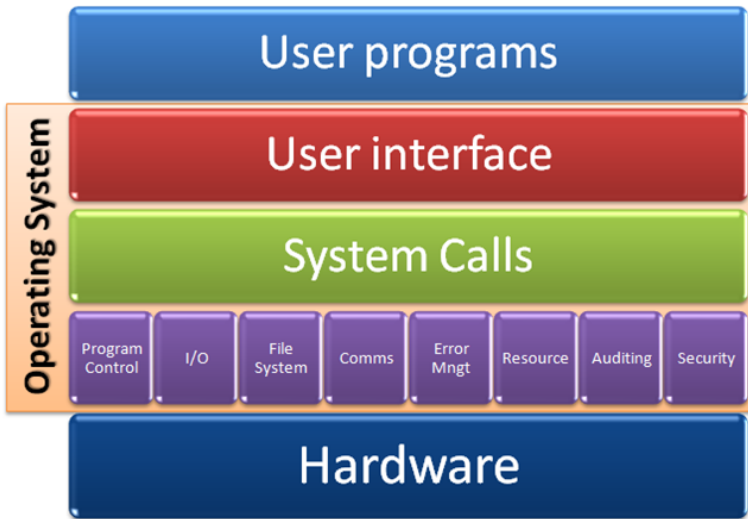
Qu'est-ce qu'un Système d'Exploitation ?

(Operating System, O.S.)

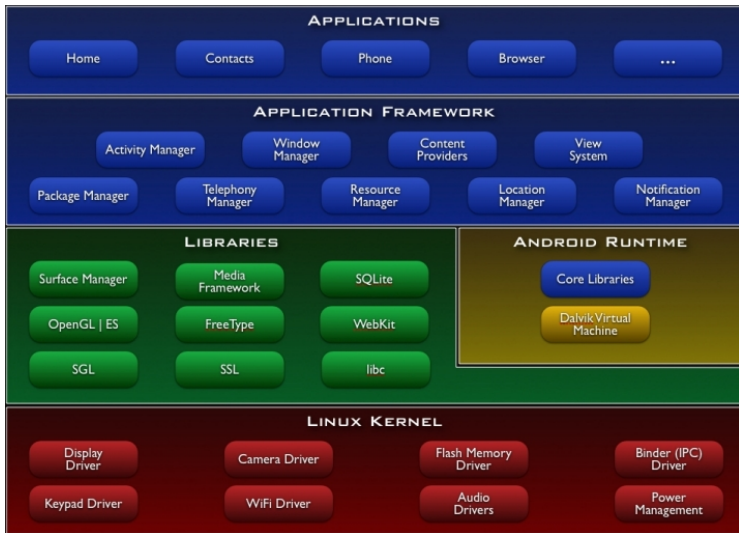
Qu'est-ce qu'un OS ? (level *n00b*)



Qu'est-ce qu'un OS ? (level *intermediate*)



Qu'est-ce qu'un OS ? (level skilled)



Les grands noms des OS

- Pour le poste client
 - Microsoft
 - Dos, Win3.1, NT4
 - Win95/98/Me/2k
 - WinXP/Vista/7/8/8.1/10

Les grands noms des OS

- Pour le poste client
 - Microsoft
 - Dos, Win3.1, NT4
 - Win95/98/Me/2k
 - WinXP/Vista/7/8/8.1/10
 - Mac OS (< 10), IBM OS/2

Les grands noms des OS

- Pour le poste client
 - Microsoft
 - Dos, Win3.1, NT4
 - Win95/98/Me/2k
 - WinXP/Vista/7/8/8.1/10
 - Mac OS (< 10), IBM OS/2
 - Unix
 - Linux, et toutes ses distributions
 - les BSDs : FreeBSD, NetBSD, OpenBSD
 - MacOS X (Mach + NextStep)
 - les tordus : GNU Hurd, L4, ...
 - les propriétaires : SGI, SCO, SunOS, Terminaux X

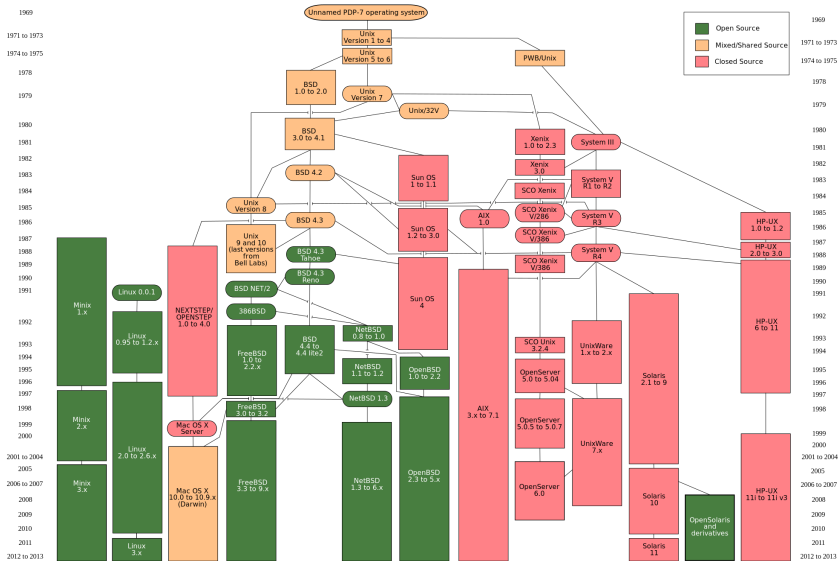
Les grands noms des OS

- Pour le poste client
 - Microsoft
 - Dos, Win3.1, NT4
 - Win95/98/Me/2k
 - WinXP/Vista/7/8/8.1/10
 - Mac OS (< 10), IBM OS/2
 - Unix
 - Linux, et toutes ses distributions
 - les BSDs : FreeBSD, NetBSD, OpenBSD
 - MacOS X (Mach + NextStep)
 - les tordus : GNU Hurd, L4, ...
 - les propriétaires : SGI, SCO, SunOS, Terminaux X
- Pour l'embarqué
 - Windows CE, Palm OS
 - android (dérivé de Linux)
 - iOS (iPhone, iPad)
 - WatchOS (Apple watch ... *seriously*)
 - consoles de jeux (Xbox, PSx, ...)

Les grands noms des OS

- Pour le poste client
 - Microsoft
 - Dos, Win3.1, NT4
 - Win95/98/Me/2k
 - WinXP/Vista/7/8/8.1/10
 - Mac OS (< 10), IBM OS/2
 - Unix
 - Linux, et toutes ses distributions
 - les BSDs : FreeBSD, NetBSD, OpenBSD
 - MacOS X (Mach + NextStep)
 - les tordus : GNU Hurd, L4, ...
 - les propriétaires : SGI, SCO, SunOS, Terminaux X
- Pour l'embarqué
 - Windows CE, Palm OS
 - android (dérivé de Linux)
 - iOS (iPhone, iPad)
 - WatchOS (Apple watch ... *seriously*)
 - consoles de jeux (Xbox, PSx, ...)
- Et du coté des serveurs
 - Windows Server 2003/2008
 - Cisco IOS, BlueCoat ProxySG, ...
 - Les Unix propriétaires : AIX, HPUNIX, SunOS

La famille des Unix



A quoi sert un OS ?

A quoi sert un OS ?

Offrir des services

- Gérer des ressources matérielles, des I/O
- En temps et en espace
- Exécuter des applications

A quoi sert un OS ?

Offrir des services

- Gérer des ressources matérielles, des I/O
- En temps et en espace
- Exécuter des applications

Quelques services

- communication entre applications
 - système de fichiers
 - IPC

A quoi sert un OS ?

Offrir des services

- Gérer des ressources matérielles, des I/O
- En temps et en espace
- Exécuter des applications

Quelques services

- communication entre applications
 - système de fichiers
 - IPC
- communication extérieur (réseau)
 - piles protocolaires (OSI)
 - drivers wifi, ethernet, ...

A quoi sert un OS ?

Offrir des services

- Gérer des ressources matérielles, des I/O
- En temps et en espace
- Exécuter des applications

Quelques services

- communication entre applications
 - système de fichiers
 - IPC
- communication extérieur (réseau)
 - piles protocolaires (OSI)
 - drivers wifi, ethernet, ...
- couches graphiques
 - gestion du framebuffer
 - accélération matérielle (3D)

A quoi sert un OS ?

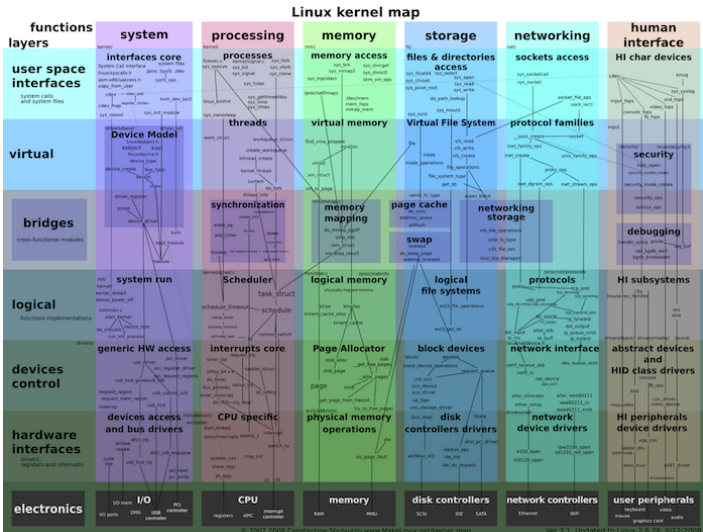
Offrir des services

- Gérer des ressources matérielles, des I/O
- En temps et en espace
- Exécuter des applications

Quelques services

- communication entre applications
 - système de fichiers
 - IPC
- communication extérieur (réseau)
 - piles protocolaires (OSI)
 - drivers wifi, ethernet, ...
- couches graphiques
 - gestion du framebuffer
 - accélération matérielle (3D)
- **Sécurité**
 - isolation
 - limitation
 - gestion d'erreurs

Un OS offre donc beaucoup de services ...



Quelques chiffres

Nombre de lignes de code source

- OS simpliste (projet d'étudiant) ~ 8K
- Unix v1 ~ 10K
- Linux 3.2.51 ~ 10M
- Windows NT4.0 ~ 11M
- Windows 7 ~ 40M

<http://www.informationisbeautiful.net/visualizations/million-lines-of-code/>

```
Totals grouped by language (dominant language first) :
ansic :      9681628 (96.83%)
asm :        245406 (2.45%)
xml :        40377 (0.40%)
perl :       15103 (0.15%)
sh :         4235 (0.04%)
cpp :        3530 (0.04%)
yacc :       2979 (0.03%)
python :    2840 (0.03%)
lex :        1726 (0.02%)
awk :         708 (0.01%)
pascal :     231 (0.00%)
lisp :       218 (0.00%)
sed :        30 (0.00%)

Total Physical Source Lines of Code (SLOC)                = 9,999,011
```

Le noyau de l'OS

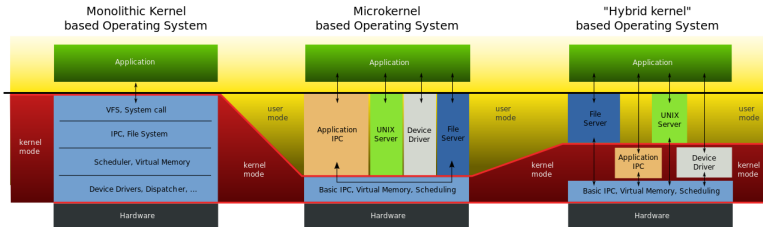
- tout ne nous intéresse pas dans un OS
 - les couches graphiques utilisateur
 - les programmes tiers (navigateurs, explorateur de fichiers, ...)
 - les applications de maintenance/gestion de l'OS
- le noyau (*kernel*) est son composant majeur et **critique**

Le noyau de l'OS

- tout ne nous intéresse pas dans un OS
 - les couches graphiques utilisateur
 - les programmes tiers (navigateurs, explorateur de fichiers, ...)
 - les applications de maintenance/gestion de l'OS
- le noyau (*kernel*) est son composant majeur et **critique**
- il impose une architecture, des APIs (*appels système*)
 - noyau monolithique
 - micro-noyau
 - exo-noyau, hybride, ...

Le noyau de l'OS

- tout ne nous intéresse pas dans un OS
 - les couches graphiques utilisateur
 - les programmes tiers (navigateurs, explorateur de fichiers, ...)
 - les applications de maintenance/gestion de l'OS
- le noyau (*kernel*) est son composant majeur et **critique**
- il impose une architecture, des APIs (*appels système*)
 - noyau monolithique
 - micro-noyau
 - exo-noyau, hybride, ...
- selon l'architecture
 - plus ou moins de composants dits *kernel land*
 - plus privilégiés par opposition au *user land*



Comment marche `malloc()` ?

But de ce cours

Rappels sur les principes des OS

- quel que soit leur(s) design/architecture/concepts
- pas de théorie sur
 - les ordonnanceurs, le temps réel
 - la gestion abstraite de la mémoire (LRU, ...)
 - les architectures de noyaux (monolithic, micro, ...)

But de ce cours

Rappels sur les principes des OS

- quel que soit leur(s) design/architecture/concepts
- pas de théorie sur
 - les ordonnanceurs, le temps réel
 - la gestion abstraite de la mémoire (LRU, ...)
 - les architectures de noyaux (monolithic, micro, ...)

Implémentation pour Intel x86

- aborder la gestion de certaines ressources (CPU, MMU)
- sans choix de design
- uniquement des contraintes matérielles
- on ne verra pas l'implémentation
 - d'un système de fichiers
 - d'une pile TCP/IP
 - de drivers, ...

But de ce cours

Rappels sur les principes des OS

- quel que soit leur(s) design/architecture/concepts
- pas de théorie sur
 - les ordonnanceurs, le temps réel
 - la gestion abstraite de la mémoire (LRU, ...)
 - les architectures de noyaux (monolithic, micro, ...)

Implémentation pour Intel x86

- aborder la gestion de certaines ressources (CPU, MMU)
- sans choix de design
- uniquement des contraintes matérielles
- on ne verra pas l'implémentation
 - d'un système de fichiers
 - d'une pile TCP/IP
 - de drivers, ...

Programme bien chargé !

But de ce cours

Objectif

- aborder la programmation des OS
- en particulier du point de vue du CPU
- faire de vous de meilleurs développeurs

But de ce cours

Objectif

- aborder la programmation des OS
- en particulier du point de vue du CPU
- faire de vous de meilleurs développeurs

Plan d'action

- la phase de boot
- les modes d'exécution du CPU (protégé essentiellement)
- la mémoire (segmentation, pagination)
- les exceptions/interruptions
- les appels systèmes
- les processus/tâches utilisateurs
- quelques mots sur :
 - gestionnaire de mémoire physique/virtuelle
 - gestionnaire d'interruptions, drivers
 - ordonnanceur, accès concurrents (lock, préemption, ...)

Introduction

La phase de démarrage

Les modes opératoires

La segmentation

Les niveaux de privilèges

Les interruptions et exceptions

La pagination

Les composants d'un OS

Conclusion

Au commencement était le BIOS

État initial du CPU

- mode réel, Boot Strap Processor (BSP)
- premières instructions :
 - $0xffffffff0$ ($max_addr - max_insn_sz$)
 - la flash ROM du BIOS y est *mappée*
 - puis saute en $0xf0000$ (*BIOS code area*)

Au commencement était le BIOS

État initial du CPU

- mode réel, Boot Strap Processor (BSP)
- premières instructions :
 - `0xffffffff0` (`max_addr - max_insn_sz`)
 - la flash ROM du BIOS y est *mappée*
 - puis saute en `0xf0000` (*BIOS code area*)

Basic Input Output System (B.I.O.S.)

- responsable de l'initialisation des composants essentiels du système
 - contrôleur de RAM
 - table de gestion des interruptions
 - tables ACPI (gestion de l'énergie entre autre)

Au commencement était le BIOS

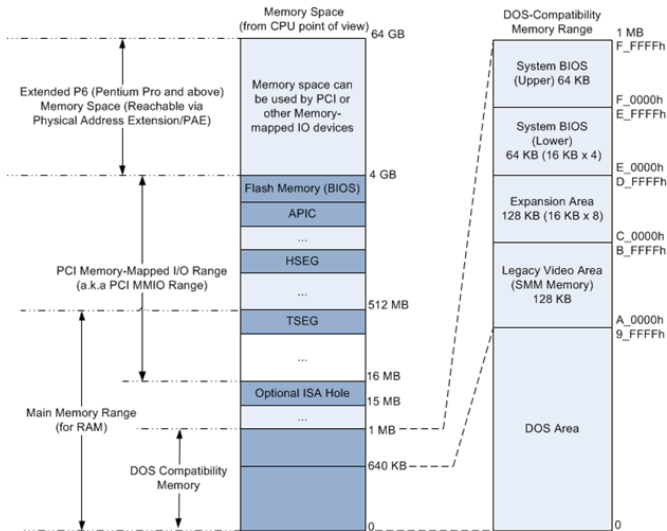
État initial du CPU

- mode réel, Boot Strap Processor (BSP)
- premières instructions :
 - `0xffffffff0` (`max_addr - max_insn_sz`)
 - la flash ROM du BIOS y est *mappée*
 - puis saute en `0xf0000` (*BIOS code area*)

Basic Input Output System (B.I.O.S.)

- responsable de l'initialisation des composants essentiels du système
 - contrôleur de RAM
 - table de gestion des interruptions
 - tables ACPI (gestion de l'énergie entre autre)
- initialement le BIOS servait à abstraire le matériel
 - int 0x13 pour les HDDs
 - int 0x15 pour la mémoire (*system map*)
 - int 0x1a pour le temps
 - Ralf Brown's Interrupt List
<http://www.ctyme.com/rbrown.htm>

État de la mémoire aux origines



Boot Sector & Boot Loader

Secteur d'amorçage (boot sector)

- le BIOS maintient une liste de périphériques de démarrage
 - HDD
 - disquette (... oui je sais)
 - clé USB
 - ...

Boot Sector & Boot Loader

Secteur d'amorçage (boot sector)

- le BIOS maintient une liste de périphériques de démarrage
 - HDD
 - disquette (... oui je sais)
 - clé USB
 - ...
- cas d'un HDD :
 - tables de partitions, *boot flag*
 - Master Boot Record, premier secteur du disque CHS(0,0,0)
 - Volume Boot Record, premier secteur d'une partition bootable

```
# fdisk /dev/sda
```

```
Command (m for help) : p
```

Device	Boot	Start	End	Blocks	Id	System
/dev/sda1	*	2048	522190847	261094400	83	Linux
/dev/sda2		522192894	524285951	1046529	5	Extended
/dev/sda5		522192896	524285951	1046528	82	Linux swap

Boot Sector & Boot Loader

Boot Sector

- contenu dans 512 octets, se termine par "\x55\xAA" (marqueur)
- chargé par le BIOS en 0x7c00
- chargera en mémoire le Boot Loader

```
# dd if=/dev/sda bs=512 count=1 of=boot_sector
# objdump -D -b binary -m i386 boot_sector
00000000 <.data> :
   0 : eb 63                jmp     0x65
   2 : 90                    nop
[... ]
  65 : fa                    cli
  66 : 90                    nop
  67 : 90                    nop
  68 : f6 c2 80             test   $0x80,%dl
  6b : 74 05                je     0x72
  6d : f6 c2 70             test   $0x70,%dl
  70 : 74 02                je     0x74
  72 : b2 80                mov    $0x80,%dl
  74 : ea 79 7c 00 00 31 c0  ljmp   $0xc031,$0x7c79
[... ]
 1fe : 55 aa
```

Boot Sector & Boot Loader

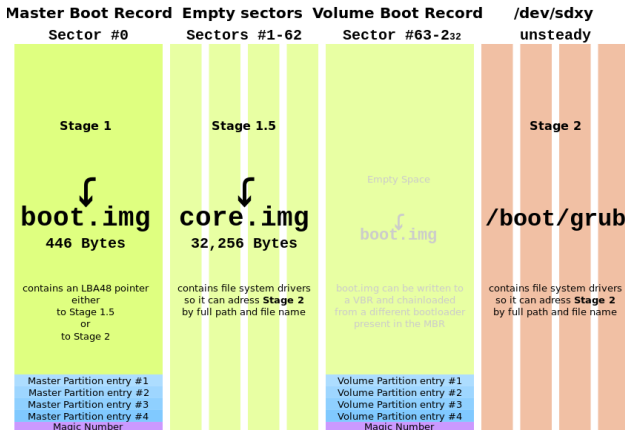
Boot Loader

- son but est de charger le noyau de l'OS
- simplifier son démarrage (mode protégé, ...)
- plus gros qu'un boot sector (mini-os parfois)
- passerelle entre le noyau et le BIOS :
 - détecte les périphériques de stockage
 - parcourt les systèmes de fichiers
 - récupère les *system map* (ie. taille de la RAM)

```
[ 0.000000] e820 : BIOS-provided physical RAM map :
[ 0.000000] BIOS-e820 : [mem 0x0000000000000000-0x000000000009fbff] usable
[ 0.000000] BIOS-e820 : [mem 0x000000000009fc00-0x000000000009ffff] reserved
[ 0.000000] BIOS-e820 : [mem 0x00000000000dc000-0x00000000000fffff] reserved
[ 0.000000] BIOS-e820 : [mem 0x0000000000100000-0x000000000affffff] usable
[ 0.000000] BIOS-e820 : [mem 0x000000000aff0000-0x000000000affffff] ACPI data
[ 0.000000] BIOS-e820 : [mem 0x000000000afff000-0x000000000affffff] ACPI NVS
[ 0.000000] BIOS-e820 : [mem 0x00000000fd000000-0x00000000fd7fffff] reserved
[ 0.000000] BIOS-e820 : [mem 0x00000000fec00000-0x00000000fec0ffff] reserved
[ 0.000000] BIOS-e820 : [mem 0x00000000fee00000-0x00000000fee0ffff] reserved
[ 0.000000] BIOS-e820 : [mem 0x00000000ffc00000-0x00000000ffffffffff] reserved
[ 0.000000] BIOS-e820 : [mem 0x00000000100000000-0x0000000014ffffff] usable
```

Quelques bootloaders célèbres

- LILO, historique et minimaliste
- u-Boot, dans l'embarqué (ARM, ...)
- GRUB, spécification multi-boot, très courant



Each **partition table entry** comprises of **16 octets**:

Flag	Start CHS	Type	End CHS	Start LBA	Size
1	3	1	3	4	4 octets

Introduction

La phase de démarrage

Les modes opératoires

La segmentation

Les niveaux de privilèges

Les interruptions et exceptions

La pagination

Les composants d'un OS

Conclusion

Modes d'opération du CPU

Les différents modes

- réel
- virtuel 8086
- system management (SMM)
- protégé :
 - IA32e (x86-64)
 - virtualisé (Intel VMX/ AMD SVM)
 - chiffré (Intel SGX/ AMD SEV)

Modes d'opération du CPU

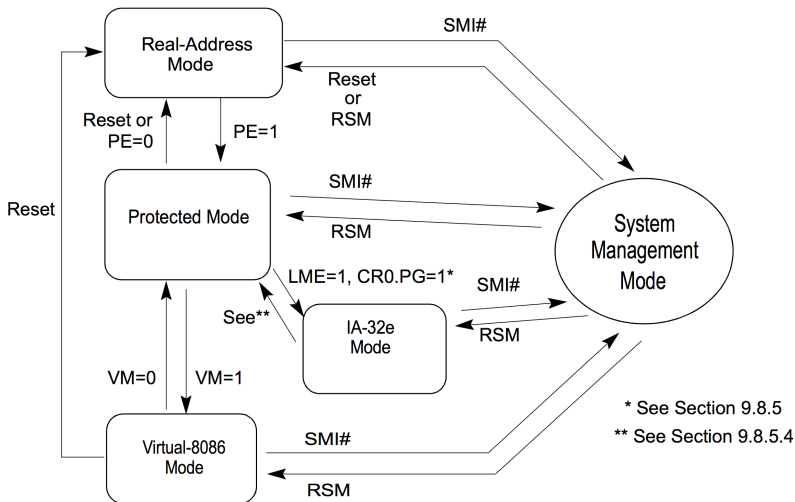
Les différents modes

- réel
- virtuel 8086
- system management (SMM)
- protégé :
 - IA32e (x86-64)
 - virtualisé (Intel VMX/ AMD SVM)
 - chiffré (Intel SGX/ AMD SEV)

Leurs conséquences

- le cpu est dans un seul mode à la fois
- transitions de modes peuvent être complexes
- environnement d'exécution est très différent selon le mode
 - quantité de mémoire adressable
 - modes d'adressages (niveau instruction assembleur)
 - gestion des interruptions/exceptions
 - instructions invalides/non autorisées

Les transitions de modes



Rappels ASM x86

Registres généraux

- multi-usage `[r,e]ax`, `[r,e]bx`, `[r,e]cx`, `[r,e]dx`, `[r,e]si`, `[r,e]di`
- pour la pile `[r,e]sp`, `[r,e]bp`
- en 64 bits `r8-r15`
 - selon le mode d'exécution (16,32,64) et/ou l'usage de préfixe
 - `ax` = 16 bits, `eax` = 32 bits, `rax` = 64 bits

Rappels ASM x86

Registres généraux

- multi-usage [r,e]ax, [r,e]bx, [r,e]cx, [r,e]dx, [r,e]si, [r,e]di
- pour la pile [r,e]sp, [r,e]bp
- en 64 bits r8-r15
 - selon le mode d'exécution (16,32,64) et/ou l'usage de préfixe
 - ax = 16 bits, eax = 32 bits, rax = 64 bits

Registres de segments (sélecteurs)

cs, ss, ds, es, fs, gs

```
mov    [ax], 0xdead    DS :ax = 0xdead
mov    fs:[ax], 0xdead  FS :ax = 0xdead
push   ax              SS :sp = ax
jump   ax              saute en CS :ax
movs   ax              ES :di = DS :si
```

Rappels ASM x86

Registres généraux

- multi-usage [r,e]ax, [r,e]bx, [r,e]cx, [r,e]dx, [r,e]si, [r,e]di
- pour la pile [r,e]sp, [r,e]bp
- en 64 bits r8-r15
 - selon le mode d'exécution (16,32,64) et/ou l'usage de préfixe
 - ax = 16 bits, eax = 32 bits, rax = 64 bits

Registres de segments (sélecteurs)

cs, ss, ds, es, fs, gs

```
mov    [ax], 0xdead    DS :ax = 0xdead
mov    fs:[ax], 0xdead  FS :ax = 0xdead
push   ax              SS :sp = ax
jump   ax              saute en CS :ax
movs   ax              ES :di = DS :si
```

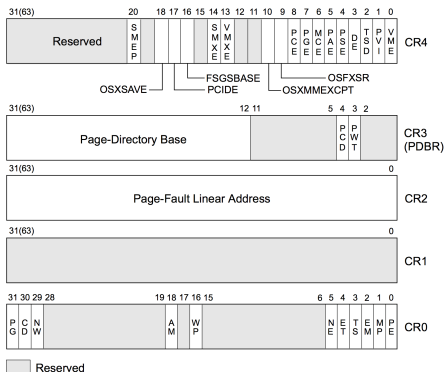
Registres systèmes

- controle : cr0, cr2, cr3, cr4
- segmentation : gdtr, idtr, ltr, tr
- model specific : MSRs

Les transitions de modes

Les registres de contrôle

- permettent d'activer des fonctionnalités du cpu
- dont le passage mode réel/protégé (cr0.pe)
- d'activer la pagination (cr0.pg, cr3)
- ...



Le mode réel

Historique

- premiers x86 : 8086, 80186, 80286 (pseudo mode protégé)
- état initial du cpu
- permet interaction facile avec le BIOS
- mode d'exécution 16 bits, adressage de 20 bits
- pas de protections (niveaux de privilèges)

Le mode réel

Historique

- premiers x86 : 8086, 80186, 80286 (pseudo mode protégé)
- état initial du cpu
- permet interaction facile avec le BIOS
- mode d'exécution 16 bits, adressage de 20 bits
- pas de protections (niveaux de privilèges)

Premiers OS de l'IBM/PC compatible

- MS-DOS
- OS/2
- Windows < 3.1

Le mode réel

Des segments de 64K

- adressage mémoire en mode 16 bits

```
mov ax, 0x1234  
mov [ax], 0xdead
```

Le mode réel

Des segments de 64K

- adressage mémoire en mode 16 bits

```
mov ax, 0x1234
mov [ax], 0xdead
```

- registre d'offset sur 16 bits $\Rightarrow 2^{16} = 64\text{KB}$
- bus d'adressage sur 20 bits $\Rightarrow 2^{20} = 1\text{MB}$
- usage des registres de segments
- calcul du cpu : $\text{adresse} = \text{selecteur} * 16 + \text{offset}$

```
mov ds, 0x100
mov [0x234x], 0xdead ---> 0x100<<4 + 0x234 = 0x1234
```

Le mode réel

Des segments de 64K

- adressage mémoire en mode 16 bits

```
mov ax, 0x1234
mov [ax], 0xdead
```

- registre d'offset sur 16 bits $\Rightarrow 2^{16} = 64\text{KB}$
- bus d'adressage sur 20 bits $\Rightarrow 2^{20} = 1\text{MB}$
- usage des registres de segments
- calcul du cpu : $\text{adresse} = \text{selecteur} * 16 + \text{offset}$

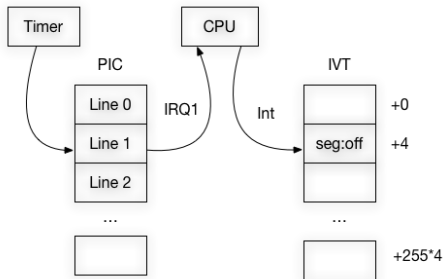
```
mov ds, 0x100
mov [0x234x], 0xdead ---> 0x100<<4 + 0x234 = 0x1234
```

```
mov ds, 0xffff
mov [0xffff], 0xdead ---> 0xffff<<4 + 0xffff = 0x10fff
> 1MB ! wrap around
```

Le mode réel

Les interruptions ... en 1 slide

- périphériques génèrent des interruptions
- le contrôleur d'interruptions (PIC, APIC) signale au CPU un index
- le cpu consulte une table de vecteurs située en 0 (IVT)
- chaque entrée fait 32 bits : 16 bits offset, 16 bits segment
- nombre d'entrées limité à 256



```
# dd if=/dev/mem bs=4 count=20 | hexdump
0000000 72a0 f000 72a0 f000 72a1 f000 72a0 f000
0000010 72a0 f000 72dc f000 72a0 f000 72a0 f000
0000020 ad91 f000 a3c5 f000 72cc f000 72cc f000
0000030 72cc f000 72cc f000 8517 f000 72cc f000
0000040 0110 c000 72ea f000 72f4 f000 8897 f000

int 0x13 ----> IVT[0x13] = [0x13*4] = [0x4c] = 8897 f000
handler = 0xf000*16 + 0x8897 = 0xf8897
```

Le mode réel

Montre rapidement ses limites

- les OS deviennent plus gros/complexes/gourmands
- arrivée du multimédia
- limitation de la mémoire à 1MB inacceptable
- si une application plante, tout plante !

Le mode réel

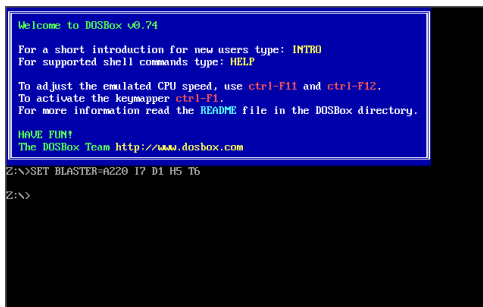
Montre rapidement ses limites

- les OS deviennent plus gros/complexes/gourmands
- arrivée du multimédia
- limitation de la mémoire à 1MB inacceptable
- si une application plante, tout plante !
- ... passage en mode protégé

Les modes exotiques

Virtual 8086

- permet l'émulation *hardware* du mode réel depuis le mode protégé
 - interruptions du mode réel
 - interception des I/Os
 - gestion de la mémoire < 1MB laissée à l'OS
- cas concret : *DosBox*



```
Welcome to DOSBox v0.74

For a short introduction for new users type: INTRO
For supported shell commands type: HELP

To adjust the emulated CPU speed, use ctrl-F11 and ctrl-F12.
To activate the keymapper ctrl-F1.
For more information read the README file in the DOSBox directory.

HAVE FUN!
The DOSBox Team http://www.dosbox.com

Z:\>SET BLASTER=A220 I7 D1 H5 T6

Z:\>
```

Les modes exotiques

System Management Mode

- mode le plus privilégié
- utilisé par les firmwares (BIOS)
- configuré au boot (SMRAM)
- accessible via une System Management Interrupt (SMI)
- généralement difficilement accessible

Les modes exotiques

System Management Mode

- mode le plus privilégié
- utilisé par les firmwares (BIOS)
- configuré au boot (SMRAM)
- accessible via une System Management Interrupt (SMI)
- généralement difficilement accessible

VMX

- gestion de la virtualisation matérielle
- 2 modes d'exécution
 - `vmx-root`, pour l'hyperviseur (VMM)
 - `vmx-nonroot`, pour la(es) machine(s) virtuelle(s) (VM)
- permet de filtrer la plupart des évènements systèmes :
 - instructions sensibles
 - interruptions/exceptions
 - la gestion de la mémoire
 - accès aux périphériques
- proposer une vision contrôlée de la machine

Le mode protégé

Adressage 32 bits mais pas que

- arrivé avec le 80386 (d'où les i386 ...)
- support des modes 16 et 32 bits
- émulation du mode réel (v8086)

Le mode protégé

Adressage 32 bits mais pas que

- arrivé avec le 80386 (d'où les i386 ...)
- support des modes 16 et 32 bits
- émulation du mode réel (v8086)

Qui dit protégé ... dit

- segmentation, pagination
- niveaux de privilèges
- changement de tâche matériel (*task switch*)

Le mode protégé

Introduction

La phase de démarrage

Les modes opératoires

La segmentation

Les niveaux de privilèges

Les interruptions et exceptions

La pagination

Les composants d'un OS

Conclusion

Gestion mémoire du mode protégé

Espace mémoire en 32 bits

- espace linéaire de 32 bits $\Rightarrow 2^{32} = 4\text{GB}$
- pendant un temps supérieur à la RAM installée
- ségmentation possible de cet espace
- adressage relatif au début d'un segment (adresse logique)
- permet une isolation des portions de l'espace linéaire

Gestion mémoire du mode protégé

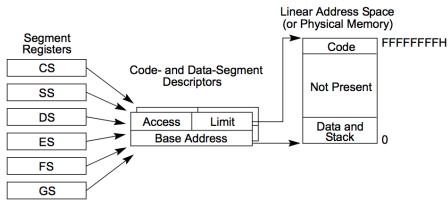
Espace mémoire en 32 bits

- espace linéaire de 32 bits $\Rightarrow 2^{32} = 4\text{GB}$
- pendant un temps supérieur à la RAM installée
- ségmentation possible de cet espace
- adressage relatif au début d'un segment (adresse logique)
- permet une isolation des portions de l'espace linéaire

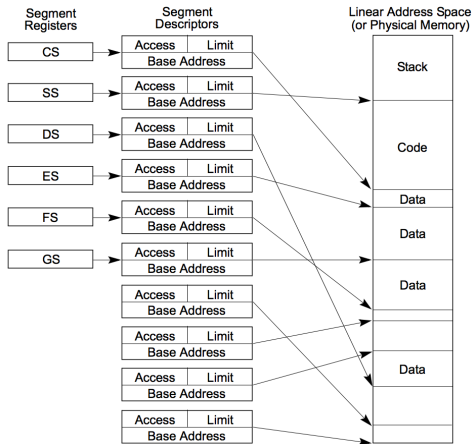
La ségmentation

- extension du principe de segments du mode réel
- modèles *flat/protected flat/multi-segment*
- des segments avec des propriétés : type, taille, droits

Modèles de ségmentation

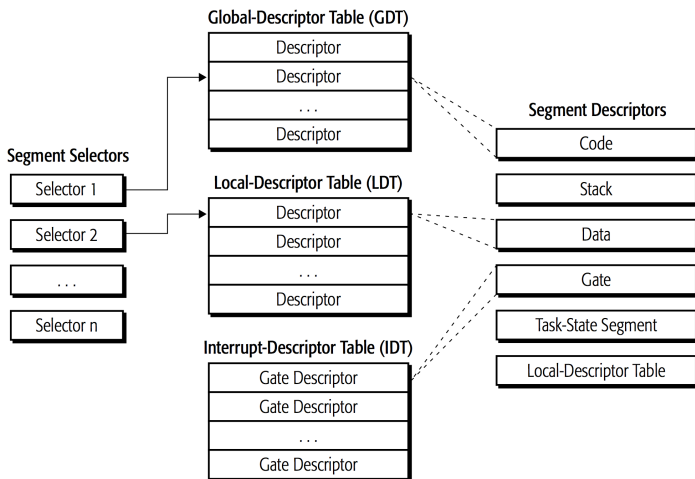


Flat



Multi-Segments

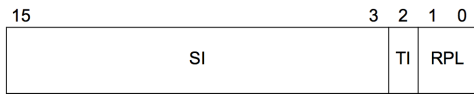
Des tables de descripteurs de segments



Du sélecteur au descripteur de segment

Le sélecteur de segment

- utilisé par les registres de segment
- permet l'accès à un descripteur
- chaque sélecteur définit
 - un niveau de privilège
 - une table (locale ou globale)
 - un indice de descripteur dans cette table
- partie visible au programmeur
- le cpu charge le descripteur dans des parties cachées du registre



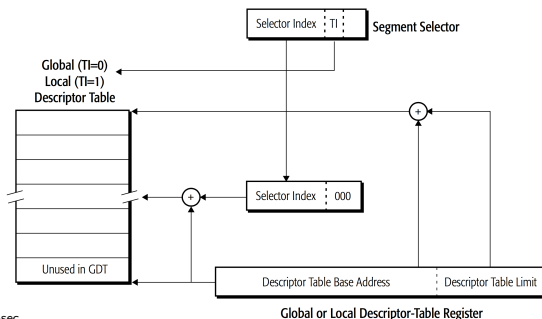
Bits	Mnemonic	Description	R/W
15-3	SI	Selector Index	R/W
2	TI	Table Indicator	R/W
1-0	RPL	Requestor Privilege Level	R/W

Registre de segment en mode protégé

Du sélecteur au descripteur de segment

Tables de descripteurs

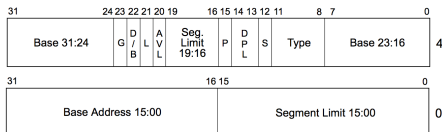
- Global Descriptor Table (GDT), `gdtr`
 - unique pour chaque coeur d'un cpu
 - 8192 entrées max, première entrée vide
- Local Descriptor Table (LDT), `ldtr`
 - locale à une tâche
 - son propre descripteur se trouve dans la GDT
- Interrupt Descriptor Table (IDT), `idtr`
 - unique pour chaque coeur d'un cpu
 - utilisée pour les exceptions/interruptions



Du sélecteur au descripteur de segment

Le descripteur de segment

- base et limite
- type de segment :
 - code (X,RX), data (R,W,RW)
 - système : TSS, Task Gate, Interrupt Gate, Call Gate

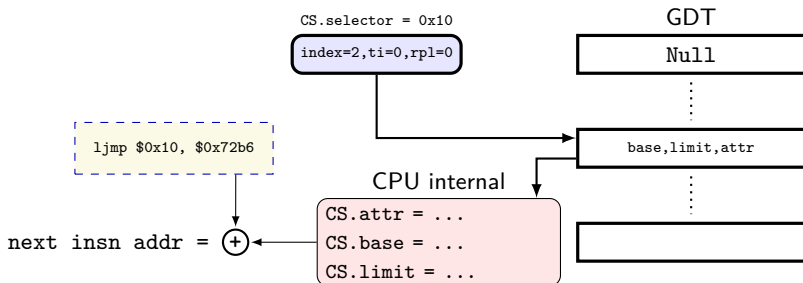


- L — 64-bit code segment (IA-32e mode only)
- AVL — Available for use by system software
- BASE — Segment base address
- D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
- DPL — Descriptor privilege level
- G — Granularity
- LIMIT — Segment Limit
- P — Segment present
- S — Descriptor type (0 = system; 1 = code or data)
- TYPE — Segment type

Du sélecteur au descripteur de segment

Exemple de far jump

- encodage x86 permet de faire un saut long
- donne le sélecteur et l'offset
- le sélecteur est chargé dans `cs` et l'offset dans `eip`
- l'adresse de la prochaine instruction est calculée comme suit :



Introduction

La phase de démarrage

Les modes opératoires

La segmentation

Les niveaux de privilèges

Les interruptions et exceptions

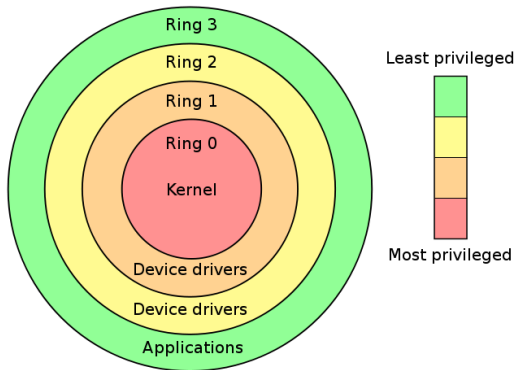
La pagination

Les composants d'un OS

Conclusion

Un anneau pour les gouverner tous

- mode réel n'avait qu'un seul niveau (*ring 0*)
- mode protégé introduit des anneaux (*ring level*)
- permet d'isoler des composants logiciels de niveau
 - de confidentialité différents
 - d'intégrité différents



Niveau de privilèges des sélecteurs et descripteurs

Identification du niveau de privilèges

- CPL, Current Privilege Level, contenu dans `cs`
- RPL, Requestor Privilege Level, au niveau du sélecteur
- DPL, Descriptor Privilege Level, au niveau du descripteur
- potentiellement très complexe
 - interprétation différente selon le type de segment
 - transferts inter-segments (Gate)

Niveau de privilèges des sélecteurs et descripteurs

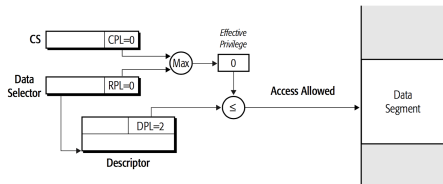
Identification du niveau de privilèges

- CPL, Current Privilege Level, contenu dans `cs`
- RPL, Requestor Privilege Level, au niveau du sélecteur
- DPL, Descriptor Privilege Level, au niveau du descripteur
- potentiellement très complexe
 - interprétation différente selon le type de segment
 - transferts inter-segments (Gate)

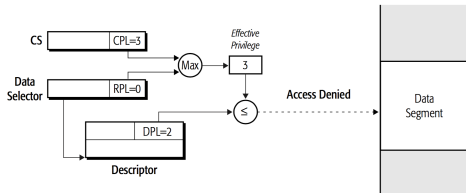
Changements de niveau de privilèges

- on ne peut accéder à un segment de données plus privilégié
- on ne peut charger un descripteur de **code** qu'à **niveau de privilège équivalent**
 - aussi bien plus faible que plus privilégié
 - passer par des *call gate* ou *interrupt gate*
 - cas classique des appels systèmes (ring 3 vers ring 0)
 - historiquement on passait par une interruption (ie. `int 0x80`)

Exemple de vérification de niveau de privilèges



Example 2: Privilege Check Passes



Example 1: Privilege Check Fails

Faisons le point

Des descripteurs sécurisants

- les niveaux de privilèges isolent les composants : Qui ?
- les bases et limites aussi à leur manière : Où/Combien ?
- les attributs également (read, write, ...) : Quoi/Comment ?

Faisons le point

Des descripteurs sécurisants

- les niveaux de privilèges isolent les composants : Qui ?
- les bases et limites aussi à leur manière : Où/Combien ?
- les attributs également (read, write, ...) : Quoi/Comment ?

Du point de vue de la sécurité

- simulation d'une architecture harvard (contre von Neumann)
- imaginez un descripteur par buffer ? anti-overflow
- Linux Kernel PaX protection : SEGMEXEC
<https://pax.grsecurity.net/docs/segmexec.txt>

Faisons le point

Des descripteurs sécurisants

- les niveaux de privilèges isolent les composants : Qui ?
- les bases et limites aussi à leur manière : Où/Combien ?
- les attributs également (read, write, ...) : Quoi/Comment ?

Du point de vue de la sécurité

- simulation d'une architecture harvard (contre von Neumann)
- imaginez un descripteur par buffer ? anti-overflow
- Linux Kernel PaX protection : SEGMEXEC
<https://pax.grsecurity.net/docs/segmexec.txt>

Et pourtant ...

- les OS modernes n'utilisent pas la segmentation (modèle flat)
- quasiment disparue en 64 bits
- basent leur sécurité sur la pagination

Introduction

La phase de démarrage

Les modes opératoires

La segmentation

Les niveaux de privilèges

Les interruptions et exceptions

La pagination

Les composants d'un OS

Conclusion

Les interruptions et les exceptions

Intérêt

- éviter l'attente active et bloquer le CPU
- apporter une composante événementielle à l'OS
- être réveillé à la demande
 - appuye d'une touche clavier
 - arrivée d'un paquet réseau
 - écoulement d'un quantum de temps

Gestion des interruptions et des exceptions

Les sources d'exceptions

- générées par le CPU, en cas d'erreurs (#GP, #NP)
- générées par du code, (#BP)
- différentes natures
 - *fault*, on la gère et on re-exécute l'instruction
 - *trap*, on la gère et on continue après l'instruction
 - *abort*, non récupérable
- priorités entre exceptions générées simultanément
- code d'erreur accompagne parfois les exceptions

Gestion des interruptions et des exceptions

Les sources d'interruptions

- générées par les périphériques
- ou par du code (`int 0x80`)
- différentes natures
 - *irq*, pour les périphériques
 - *smi*, par le mode SMM
 - *nmi*, par des périphériques souvent liés à l'ACPI
 - *ipi*, entre coeurs d'un CPU, entre CPUs (*smp*)
- certaines peuvent être masquées (via l'instruction `cli`)
- une fois masquées, le CPU est in-interruptible
- sauf dans le cas des *nmi* (Non Maskable Interrupts)

Quelques exceptions usuelles

General Protection Fault :#GP

- survient dans de nombreux cas
 - erreurs liées à la segmentation (taille, droits, pvl)
 - configuration invalide des registres de controle, MSRs
- du type *fault*
- fournit un code d'erreur
 - sélecteur de segment fautif, si erreur de segmentation
 - ou 0 pour les autres erreurs

Quelques exceptions usuelles

General Protection Fault :#GP

- survient dans de nombreux cas
 - erreurs liées à la segmentation (taille, droits, pvl)
 - configuration invalide des registres de contrôle, MSRs
- du type *fault*
- fournit un code d'erreur
 - sélecteur de segment fautif, si erreur de segmentation
 - ou 0 pour les autres erreurs

Break Point :#BP

- survient lorsque l'instruction `int3` est exécutée
- du type *trap*
- pas de code d'erreur

Quelques exceptions usuelles

Invalid Opcode :#UD

- survient lorsqu'une instruction est invalide
- du type *fault*
- pas de code d'erreur

Quelques exceptions usuelles

Invalid Opcode :#UD

- survient lorsqu'une instruction est invalide
- du type *fault*
- pas de code d'erreur

Divide Error :#DE

- survient lorsque l'on effectue une division par zéro
- du type *fault*
- pas de code d'erreur

Les fautes de pages (#PF, *page fault*)

Concerne les erreurs liées à la pagination

- page non présente ($\text{pte.p/pde.p} = 0$)
- accès read/write
- accès user/supervisor

Les fautes de pages (#PF, *page fault*)

Concerne les erreurs liées à la pagination

- page non présente ($\text{pte.p}/\text{pde.p} = 0$)
- accès read/write
- accès user/supervisor

Code d'erreur et CR2

- code d'erreur indique la nature de la faute
- le registre `cr2` contient l'adresse fautive
- parfois différente de l'endroit où la faute est générée

```
0x804801e  mov  [0x1234], eax
```

- `cr2 = 0x1234`, faute générée en `0x804801e`

Gestion des interruptions et des exceptions

Fonctionnement

- qui dit interruption, dit reprise d'exécution
- nécessaire de sauvegarder l'état du CPU au moment de l'interruption
- on va parler
 - de pile d'interruption
 - d'Interrupt Gate (descripteur de segment d'interruption)
 - d'appels systèmes (implémentation historique)

Gestion des interruptions et des exceptions

Fonctionnement

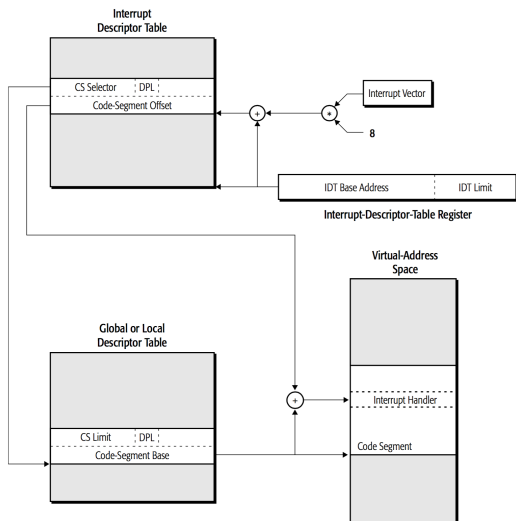
- qui dit interruption, dit reprise d'exécution
- nécessaire de sauvegarder l'état du CPU au moment de l'interruption
- on va parler
 - de pile d'interruption
 - d'Interrupt Gate (descripteur de segment d'interruption)
 - d'appels systèmes (implémentation historique)

Comme pour la segmentation

- les gestionnaires (*handlers*) sont placés dans une table
- on parle d'Interrupt Descriptor Table (IDT)
- ressemble à une GDT, avec des descripteurs spécifiques
- localisée grace au registre `idtr`

La table des interruptions IDT

- les descripteurs référencent des segments de code et un *handler*
- indiquent également le DPL nécessaire pour y accéder (généralement ring 0)
- sauf le *handler* d'appels systèmes qui sera en ring 3 (linux 0x80)



Le contexte d'interruption

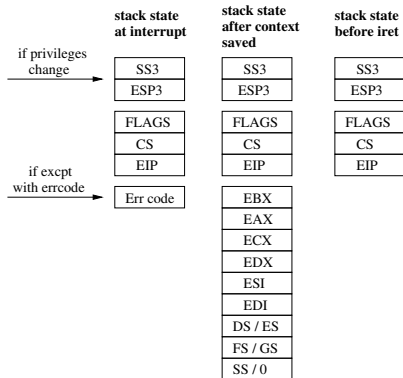
Le changement de contexte

- un contexte indique un état du CPU à un instant précis
- cela implique
 - le niveau de privilèges courant
 - la valeur des registres généraux (eax, ebx, ... esp)
 - le registre d'état (EFLAGS)
- dans le cas de l'arrivée d'une interruption/exception, on peut potentiellement changer de niveau de privilèges (ring 3 \Rightarrow ring 0)
- le CPU sauvegarde pour nous les informations du niveau de provenance
- le noyau fait le reste

Le contexte d'interruption

En trois étapes

- au moment de l'interruption, le CPU sauvegarde le minimum et détermine s'il y a eu changement de niveau de privilèges
- avant de traiter l'interruption, l'OS sauvegarde ce qu'il pense être nécessaire de restaurer
- au retour de l'interruption, l'OS utilise une instruction (`iret`) spécifique



Le contexte d'interruption

Comment le CPU sait où sauvegarder ?

- dans le cas d'une transition ring 3 \Rightarrow ring 0
 - le CPU cherche à changer de pile (registre esp)
 - sauver les informations du ring 3 dans le ring de destination
 - où trouver la nouvelle valeur à mettre dans esp ?

Le contexte d'interruption

Comment le CPU sait où sauvegarder ?

- dans le cas d'une transition ring 3 \Rightarrow ring 0
 - le CPU cherche à changer de pile (registre esp)
 - sauver les informations du ring 3 dans le ring de destination
 - où trouver la nouvelle valeur à mettre dans esp ?
- grâce au TSS (Task State Segment)

Le contexte d'interruption

Comment le CPU sait où sauvegarder ?

- dans le cas d'une transition ring 3 \Rightarrow ring 0
 - le CPU cherche à changer de pile (registre esp)
 - sauver les informations du ring 3 dans le ring de destination
 - où trouver la nouvelle valeur à mettre dans esp ?
- grâce au TSS (Task State Segment)

Changement de tâche matériel

- descripteur de segment spécifique dans la GDT
- dont l'index est chargé dans le registre tr
- mis à jour par l'OS à chaque changement de tâche
- en particulier son champ esp0

Le contexte d'interruption

Comment le CPU sait où sauvegarder ?

- dans le cas d'une transition ring 3 \Rightarrow ring 0
 - le CPU cherche à changer de pile (registre esp)
 - sauver les informations du ring 3 dans le ring de destination
 - où trouver la nouvelle valeur à mettre dans esp ?
- grâce au TSS (Task State Segment)

Changement de tâche matériel

- descripteur de segment spécifique dans la GDT
- dont l'index est chargé dans le registre tr
- mis à jour par l'OS à chaque changement de tâche
- en particulier son champ esp0
- consulté par le CPU à chaque changement de privilèges
- initialement utilisé pour effectuer des changements de tâches matériel
- pas utilisé dans les OS modernes

Les appels systèmes

Historiquement

- géré grâce à une interruption accessible depuis le ring 3
- Interrupt Gate avec $DPL=3$ dans l'IDT
- les programmes utilisent `int N`
- où N correspond à l'indice dans l'IDT du descripteur

Les appels systèmes

Historiquement

- géré grâce à une interruption accessible depuis le ring 3
- Interrupt Gate avec DPL=3 dans l'IDT
- les programmes utilisent `int N`
- où N correspond à l'indice dans l'IDT du descripteur

De nos jours

- on utilise les *fast system call*
- en passant par les instructions `sysenter`, `sysexit`
- l'OS configure des MSRs
 - `MSR_SYSENTER_CS`
 - `MSR_SYSENTER_EIP`
 - `MSR_SYSENTER_ESP`

Les appels systèmes : exemple sous Linux 32 bits

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

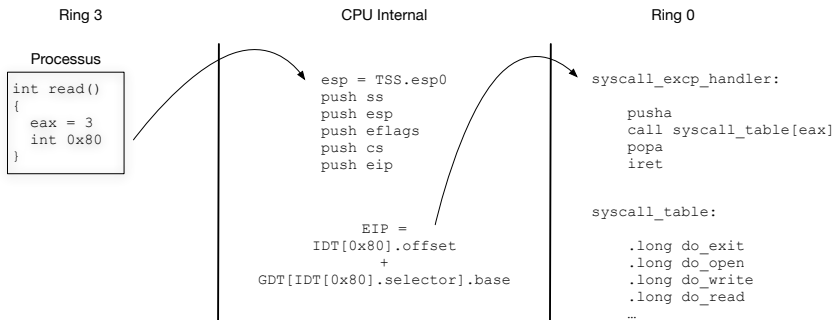
void main(int argc, char **argv)
{
    char buffer[10];

    int fd = open(argv[1], O_RDONLY);
    read(fd, buffer, 10);
}

$ gcc -static -m32 test.c -o test
$ objdump -D test

08048e24 <main> :
8048e24 : 55                push   %ebp
8048e25 : 89 e5            mov    %esp,%ebp
...
8048e74 : e8 47 3d 02 00   call   806cbc0 <__libc_read>
...
0806cbc0 <__libc_read> :
806cbc0 : 65 83 3d 0c 00 00 00  cmp    $0x0,%gs :0xc
806cbc7 : 00
806cbc8 : 75 25            jne    806cbef <__read_nocancel+0x25>
...
0806cbca <__read_nocancel> :
806cbca : 53                push   %ebx
806cbcb : 8b 54 24 10      mov    0x10(%esp),%edx
806cbcf : 8b 4c 24 0c      mov    0xc(%esp),%ecx
806cbd3 : 8b 5c 24 08      mov    0x8(%esp),%ebx
806cbd7 : b8 03 00 00 00   mov    $0x3,%eax
806cbd8 : ff 15 f0 99 0e 08 call   *0x80e99f0
...
080e99f0 <_dl_sysinfo> :
80e99f0 : 70 ed 06 08
...
0806ed70 <_dl_sysinfo_int80> :
806ed70 : cd 80            int    $0x80
806ed72 : c3                ret
```

Les appels systèmes : exemple sous Linux 32 bits



Introduction

La phase de démarrage

Les modes opératoires

La segmentation

Les niveaux de privilèges

Les interruptions et exceptions

La pagination

Les composants d'un OS

Conclusion

Gestion mémoire du mode protégé

La pagination

- l'adresse linéaire n'est plus l'adresse physique finale (RAM)
- elle devient une adresse virtuelle
- traduite en adresse physique via des tables par la MMU
- un peu de jargon
 - on parle d'espace d'adressage (*address space*)
 - de *mapping* mémoire
 - page tables/directory, memory management unit (MMU)
 - page table entries (pte)

Gestion mémoire du mode protégé

La pagination

- l'adresse linéaire n'est plus l'adresse physique finale (RAM)
- elle devient une adresse virtuelle
- traduite en adresse physique via des tables par la MMU
- un peu de jargon
 - on parle d'espace d'adressage (*address space*)
 - de *mapping* mémoire
 - page tables/directory, memory management unit (MMU)
 - page table entries (pte)

Intérêt

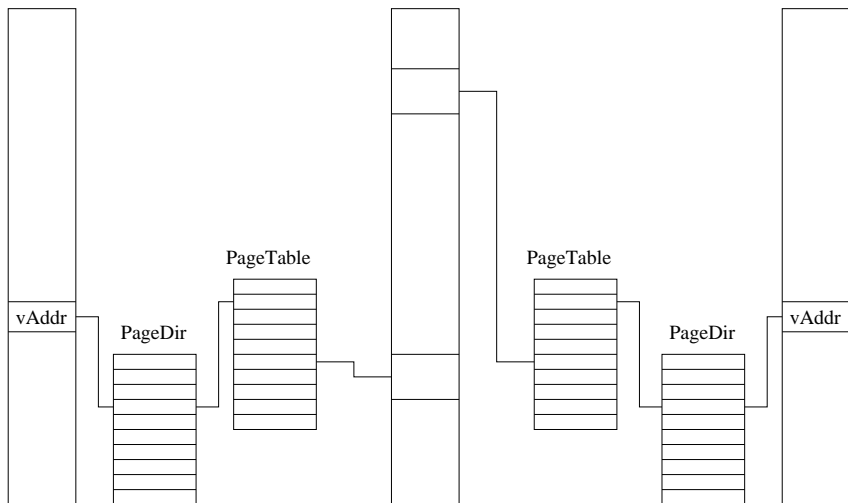
- optimiser la consommation mémoire
 - demand paging (ie. la pile)
 - shared memory (ie. des *threads*)
- charger plusieurs fois le meme programme au meme instant
 - adresses virtuelles identiques (ie. fichier exécutable)
 - adresses physiques distinctes

Espaces d'adressage de 2 exécutables

User vspace 1

Physical Mem

User Vspace 2



La pagination en mode 32 bits

Répertoire et tables

- un répertoire de pages (*page directory*, PGD)
- une ou plusieurs table(s) de pages (*page tables*, PTB)
- le répertoire et les tables contiennent uniquement des **adresses physiques** !
- les entrées du répertoire et des tables sont assez similaires

La pagination en mode 32 bits

Répertoire et tables

- un répertoire de pages (*page directory*, PGD)
- une ou plusieurs table(s) de pages (*page tables*, PTB)
- le répertoire et les tables contiennent uniquement des **adresses physiques** !
- les entrées du répertoire et des tables sont assez similaires

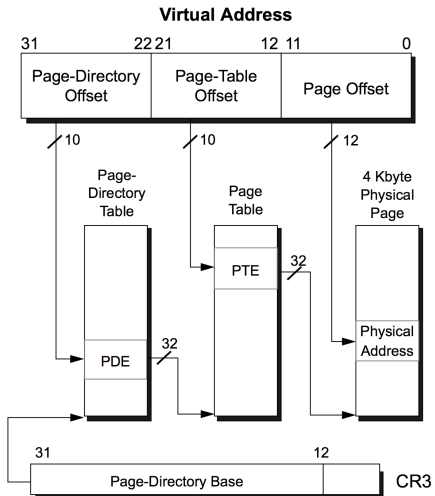
Dans le détail

- le PGD voit son adresse physique stockée dans `cr3`
- chaque table contient au plus 1024 entrées de 4 octets
- une entrée de table (*PTE*) définit une page de 4KB
- une table de pages couvre donc $1024 * 4KB = 4MB$ d'espace virtuel
- une entrée de répertoire (*PDE*) peut définir
 - une table de pages de 1024 entrées
 - une page de 4MB (*bit pse*)
- un répertoire de pages couvre donc $1024 * 4MB = 4GB$ d'espace virtuel

La pagination en mode 32 bits

Traduction d'une adresse virtuelle en adresse physique

- adresse de 32 bits découpée en 3 parties
- 10 bits d'index dans le PGD ($2^{10} = 1024$)
- 10 bits d'index dans la PTB
- 12 bits d'*offset* dans la page ($2^{12} = 4096$)



La pagination en mode 32 bits

Détail des entrées de PTE/PDE

- Present, génère #PF (*page fault*) si 0
- User (ring 3), Supervisor (Ring 0,1,2)
- PSE (page de 4MB ou table de pages)
- G (global), mapping persistant
- bits de gestion des caches (PAT, PCD, PWT)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory ¹												Ignored					PCD	PWT	Ignored			CR3										
Bits 31:22 of address of 4MB page frame					Reserved (must be 0)				Bits 39:32 of address ²				PAT	Ignored	G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 4MB page								
Address of page table												Ignored					Q	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table							
Ignored																	Q	PDE: not present														
Address of 4KB page frame												Ignored	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page										
Ignored																	Q	PTE: not present														

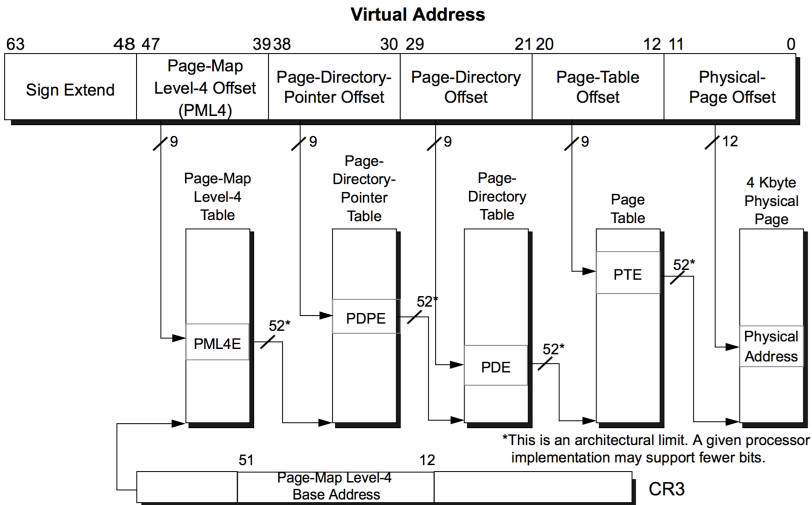
Les différents modes de pagination

De nombreuses variations

- taille des adresses virtuelles/physiques
- taille des pages adressables

Mode		Physical-Address Extensions (CR4.PAE)	Page-Size Extensions (CR4.PSE)	Page-Directory Pointer Offset	Page-Directory Page Size	Resulting Physical-Page Size	Maximum Virtual Address	Maximum Physical Address	
Long Mode	64-Bit Mode	Enabled	–	PDPE.PS=0	PDE.PS=0	4 Kbyte	64-bit	52-bit	
	Compatibility Mode				PDE.PS=1	2 Mbyte			
					PDPE.PS=1	–			1 Gbyte
Legacy Mode		Enabled	–	PDPE.PS=0	PDE.PS=0	4 Kbyte	32-bit	52-bit	
					PDE.PS=1	2 Mbyte		52-bit	
		Disabled			Disabled	–		4 Kbyte	32-bit
					Enabled	PDE.PS=0		4 Kbyte	32-bit
						PDE.PS=1		4 Mbyte	40-bit

Par curiosité ... en 64 bits



Et après la traduction ?

Les Translation Lookaside Buffers (TLBs)

- traduction couteuse
- chaque traduction est conservée
 - dans des caches spécifiques (les TLBs)
 - temporairement, une écriture dans *cr3* les invalide
 - on parle alors de *TLB flush*
- les entrées Global sont conservées (sauf si écriture dans *cr4*)

Et après la traduction ?

Les Translation Lookaside Buffers (TLBs)

- traduction couteuse
- chaque traduction est conservée
 - dans des caches spécifiques (les TLBs)
 - temporairement, une écriture dans *cr3* les invalide
 - on parle alors de *TLB flush*
- les entrées Global sont conservées (sauf si écriture dans *cr4*)

Dans la pratique, que fait un OS ?

- le noyau expose sa mémoire
 - dans tous les espaces d'adressage
 - en mode *supervisor*, bit U=0
 - en mode global, bit G=1
- chaque tache possède son propre espace d'adressage
- maintenu par le noyau uniquement (sécurité)
- chaque changement de tache implique un chagement d'espace d'adressage
- et une perte des caches de traduction (*write cr3*)
- sauf des entrées globales

Comment un OS met à jour les tables de pages ?

Le problème

- pagination activée \Rightarrow toute adresse est virtuelle
- or les tables contiennent des adresses physiques (ie. celles des autres tables)
- comment lire/modifier une table quand on ne connaît que son adresse physique ?

Comment un OS met à jour les tables de pages ?

Le problème

- pagination activée \Rightarrow toute adresse est virtuelle
- or les tables contiennent des adresses physiques (ie. celles des autres tables)
- comment lire/modifier une table quand on ne connaît que son adresse physique ?

Plusieurs solutions

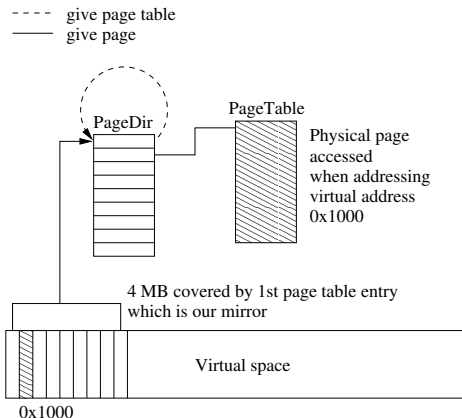
- *identity mapping* ... misérable
- correspondance linéaire virtuelle/physique (noyau linux, `PAGE_OFFSET`)
- un peu plus métaphysique ... le *self-mapping*

Le principe du *self-mapping*

- une entrée du PGD est utilisée comme *self-map index*
- l'adresse de la table de pages à cet index est le PGD lui-meme
- conséquence : annulation d'un niveau d'indirection
- les pages finales accédées seront les tables de pages
- inconvénient : on perd 4MB d'espace virtuel

Le principe du *self-mapping*

- une entrée du PGD est utilisée comme *self-map index*
- l'adresse de la table de pages à cet index est le PGD lui-meme
- conséquence : annulation d'un niveau d'indirection
- les pages finales accédées seront les tables de pages
- inconvénient : on perd 4MB d'espace virtuel



Sécurité et pagination

Une sécurité toute relative

- protection noyau/tache, bit User/Supervisor
- protection des données, bit Read/Write
- historiquement Read \Rightarrow Execute, **WTF !**
- impossibilité d'obtenir $W \oplus X$?

Sécurité et pagination

Une sécurité toute relative

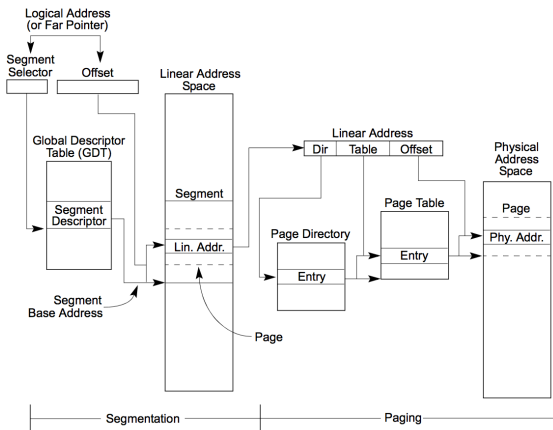
- protection noyau/tache, bit User/Supervisor
- protection des données, bit Read/Write
- historiquement Read \Rightarrow Execute, **WTF !**
- impossibilité d'obtenir $W \oplus X$?

Des améliorations au fil du temps

- différenciation iTLB (instructions), dTLB (données)
- Linux Kernel PaX protection : PAGEEXEC
<https://pax.grsecurity.net/docs/pageexec.txt>
- protection exécution, bit NX (uniquement PAE, 64 bits)
- d'autres protections (via cr0)
 - WP (write protect)
 - SMEP/SMAP (execution/access prevention)

Vue d'ensemble d'un accès mémoire

La mémoire : putting it all together



Du fichier exécutable au processus

au format ELF 32 bits pour Linux

Du fichier exécutable au processus

Compilation

- le compilateur catégorise les éléments du code source
- il les place dans des *sections*
 - le code se retrouve dans une section `.text`
 - les variables globales initialisées dans `.data`
 - les variables globales constantes dans `.rodata`
 - les variables globales non-initialisées dans `.bss`
 - les éléments dynamiques (tas, pile) sont hors scope
 - ils n'existent pas avant l'exécution du fichier !

Du fichier exécutable au processus

Compilation

- le compilateur catégorise les éléments du code source
- il les place dans des *sections*
 - le code se retrouve dans une section `.text`
 - les variables globales initialisées dans `.data`
 - les variables globales constantes dans `.rodata`
 - les variables globales non-initialisées dans `.bss`
 - les éléments dynamiques (tas, pile) sont hors scope
 - ils n'existent pas avant l'exécution du fichier !
- chaque section possède des attributs
 - adresse de base, taille, des droits (`rxw`)
 - des contraintes d'alignement
 - certaines peuvent être d'une taille nulle dans le fichier
 - et avoir une taille conséquente dans le processus chargé en mémoire
 - cas de la section `.bss`
 - contient uniquement des variables non initialisées
 - au chargement de l'exécutable, cette section sera mise à 0

Du fichier exécutable au processus

```

#include <stdio.h>

int globale_non_initialisee ;
int globale_initialisee = 1234 ;

int main(int argc, char **argv)
{
    if(argc != 2)
        globale_non_initialisee = 3 ;

    printf("hello world\n") ;
    return 0 ;
}

```

example.c

```

.data
.align    4
.type    globale_initialisee, @object
.size    globale_initialisee, 4
globale_initialisee :
.long    1234

.section .rodata
.LC0 :
.string  "hello world"

.text
.globl  main
.type  main, @function
main :
pushl  %ebp
movl   %esp, %ebp
andl   $-16, %esp
subl   $16, %esp
subl   $2, 8(%ebp)
cmpl   $2, 8(%ebp)
je     .L2
movl   $3, globale_non_initialisee
.L2 :
movl   $.LC0, (%esp)
call   puts
movl   $0, %eax
leave

```

gcc -S example.c

Du fichier exécutable au processus

Édition de liens

- le *linker* (*ld*) prépare l'exécutable final
- il est responsable du choix des adresses attribuées à chaque section
- leur agencement également
- il se base sur un fichier de configuration (*ldscript*)

Du fichier exécutable au processus

Édition de liens

- le *linker* (*ld*) prépare l'exécutable final
- il est responsable du choix des adresses attribuées à chaque section
- leur agencement également
- il se base sur un fichier de configuration (*ldscript*)
- sous linux, pour le format ELF, on en trouve dans `/usr/lib/ldscripts`

```
$ ls /usr/lib/ldscripts/
elf32_x86_64.x      elf32_x86_64.xsw  elf_i386.xr      elf_k10m.xdc
elf32_x86_64.xbn  elf32_x86_64.xu   elf_i386.xs      elf_k10m.xdw
elf32_x86_64.xc    elf32_x86_64.xw   elf_i386.xsc     elf_k10m.xn
elf32_x86_64.xd    elf_i386.x        elf_i386.xsw     elf_k10m.xr
elf32_x86_64.xdc  elf_i386.xbn     elf_i386.xu      elf_k10m.xs
elf32_x86_64.xdw  elf_i386.xc      elf_i386.xw      elf_k10m.xsc
elf32_x86_64.xn   elf_i386.xd      elf_k10m.x       elf_k10m.xsw
elf32_x86_64.xr   elf_i386.xdc     elf_k10m.xbn     elf_k10m.xu
elf32_x86_64.xs   elf_i386.xdw     elf_k10m.xc      elf_k10m.xw
elf32_x86_64.xsc  elf_i386.xn      elf_k10m.xd      elf_l10m.x
```


Du fichier exécutable au processus : édition de liens

```

/* Default linker script, for normal executables */
OUTPUT_FORMAT("elf32-1386", "elf32-1386", "elf32-1386")
OUTPUT_ARCH(i386)
ENTRY(_start)
SEARCH_DIR("/usr/lib32-linux-gnu/lib32");
SEARCH_DIR("/usr/x86_64-linux-gnu/lib32");
SEARCH_DIR("/usr/lib");
SECTIONS
{
    /* Read-only sections, merged into text segment : */
    . = SEGMENT_START("text-segment", 0x08048000) + SIZEOF_HEADERS;

    .text          :
    {
        *(.text.unlikely .text.*_unlikely .text.unlikely.*)
        *(.text.exit .text.exit.*)
        *(.text.startup .text.startup.*)
        *(.text.hot .text.hot.*)
        *(.text.stub .text.*.gnu.linkonce.t.*)
        /* .gnu.warning sections are handled specially by elf32.em. */
        *(.gnu.warning)
    }

    .data          :
    {
        *(.data.data.*.gnu.linkonce.d.*)
        SORT(CONSTRUCTORS)
    }

    __bss_start = .;
    .bss           :
    ...
}

```

script LD elf_i386.x

```

Section Headers :
[Nr] Name      Type        Addr       Off       Size   ES Flg Lk Inf Al
  [ 0]          NULL       00000000   000000   000000 00      0  0  0
...
[13] .text      PROGBITS   08048320   000320   0001a2 00  AX  0  0 16
[14] .fini      PROGBITS   080484c4   0004c4   000014 00  AX  0  0  4
[15] .rodata    PROGBITS   080484d8   0004d8   000014 00  A   0  0  4
...
[24] .data      PROGBITS   0804a018   001018   00000c 00  WA  0  0  4
[25] .bss       NOBITS     0804a024   001024   000008 00  WA  0  0  4
...
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) o (OS specific), p (processor specific)

Program Headers :
Type      Offset  VirtAddr  PhysAddr  FileSiz MemSiz  Flg Align
PHDR     0x000034 0x08048034 0x08048034 0x00120 0x00120 R E 0x4
INTERP   0x000154 0x08048154 0x08048154 0x00013 0x00013 R 0x1
          [Requesting program interpreter : /lib/ld-linux.so.2]
LOAD     0x000000 0x08048000 0x08048000 0x005c8 0x005c8 R E 0x1000
LOAD     0x000f08 0x08049f08 0x08049f08 0x0011c 0x00124 RW 0x1000
DYNAMIC  0x000f14 0x08049f14 0x08049f14 0x000e8 0x000e8 RW 0x4
NOTE     0x000168 0x08048168 0x08048168 0x00044 0x00044 R 0x4

```

\$ readelf -e example

Du fichier exécutable au processus

Chargement du fichier exécutable

- l'appel système `execve()` est responsable du chargement
- plusieurs cas possibles (binaires statiques/dynamiques)
- dans l'ensemble :
 - le noyau crée un espace d'adressage vide (PGD,PTB)
 - il *parse* le fichier au format ELF
 - alloue la mémoire physique pour chaque *program header*
 - chaque page lue depuis le fichier est copiée en mémoire physique
 - les pages de mémoire physique sont *mappées* aux adresses virtuelles indiquées dans le fichier ELF
 - l'exécutable est devenu un **processus** !

Explorons la mémoire du processus
à l'aide de Ramooflax

Exploration de la mémoire d'un processus

Programme d'exemple

- attente active de l'appuyé sur une touche
- donne une chance de le trouver avant sa terminaison

```
const unsigned long long magic = 0xdeadbabedeadb00b ;

int main(void)
{
    char *buffer ;
    char key ;

    printf("type any key to continue\n") ;
    fcntl(0, F_SETFL, fcntl(0, F_GETFL)|O_NONBLOCK) ;
    while(read(0, &key, 1) == -1 && errno == EAGAIN) ;

    buffer = (char*)malloc(20) ;

    printf("stack variable 'key'    is at 0x%.8x\n"
           "heap variable 'buffer' is at 0x%.8x\n",
           &key, buffer) ;

    return 0 ;
}
```

Exploration de la mémoire d'un processus

Programme d'exemple

- attente active de l'appuyé sur une touche
- donne une chance de le trouver avant sa terminaison

```
const unsigned long long magic = 0xdeadbabedeadb00b ;

int main(void)
{
    char *buffer ;
    char key ;

    printf("type any key to continue\n") ;
    fcntl(0, F_SETFL, fcntl(0, F_GETFL)|O_NONBLOCK) ;
    while(read(0, &key, 1) == -1 && errno == EAGAIN) ;

    buffer = (char*)malloc(20) ;

    printf("stack variable 'key'   is at 0x%.8x\n"
           "heap variable 'buffer' is at 0x%.8x\n",
           &key, buffer) ;

    return 0 ;
}
```

Comment trouver ce processus dans toute la mémoire du système ?

Exploration de la mémoire d'un processus

Grace à Ramooflax

- hyperviseur qui sert de debugger
- interface distante en python (script)
- recherche du processus en fonction de son *magic*

```
const unsigned long long magic = 0xdeadbabe00b ;
```

- l'adresse de la variable *magic* est récupérée via

```
$ nm -f sysv malloc | grep magic  
magic |08048660| R | OBJECT|00000008| |.rodata
```

Exploration de la mémoire d'un processus : script Ramooflax

- filtre les accès au registre cr3


```
vm.cpu.filter_write_cr(3, hook_cr3)
```
- vérifie si l'adresse du *magic* existe


```
vm.mem.translate(magic_addr)
```
- vérifie la valeur du *magic*

```
vm.mem.read_qword(magic_addr) == magic_val
```
- installe un *breakpoint* dans le processus


```
vm.cpu.breakpoints.add_insn(magic_bp)
```
- nous donne un shell interactif

```
#!/usr/bin/env python

import sys, struct
from ramooflax import VM, CPUFamily, log, PgMsk

def hook_cr3(vm) :
    cr3 = vm.cpu.sr.cr3 & PgMsk.addr
    try :
        vm.mem.translate(magic_addr)
        if vm.mem.read_qword(magic_addr) == magic_val :
            log("info", "found process cr3 0x%x" % cr3)

            vm.cpu.breakpoints.add_insn(magic_bp)
            vm.cpu.release_write_cr(3)
            return True
    except :
        pass

    return False

#####
#### MAIN ####
#####
peer = "172.16.131.128 :1337"
vm = VM(CPUFamily.Intel, peer)

magic_addr = 0x8048660
magic_bp   = 0x804858e
magic_val  = 0xdeadbabedeadb00b

log.setup(info=True, fail=True,
          gdb=False, vm=True,
          ads=False, brk=True, evt=False)

vm.attach()
vm.stop()

vm.filter_detach(lambda vm : vm.cpu.del_active_cr3())

vm.cpu.filter_write_cr(3, hook_cr3)

vm.interact2(dict(globals()), **locals())
```

Vue d'ensemble du mode protégé

ie. ordonnancement

Exemple d'ordonnancement

Le contexte

- considérons un processus (A) s'exécutant en ring 3 du mode protégé
- celui-ci est interrompu par l'apparition d'une interruption d'horloge (*irq 0*)
- le noyau (K) en ring 0 traite l'interruption
- et décide d'ordonnancer un autre processus (B)

Exemple d'ordonnancement

Le contexte

- considérons un processus (A) s'exécutant en ring 3 du mode protégé
- celui-ci est interrompu par l'apparition d'une interruption d'horloge (*irq 0*)
- le noyau (K) en ring 0 traite l'interruption
- et décide d'ordonnancer un autre processus (B)

Cette vue d'ensemble regroupe

- la mémoire physique
- les espaces d'adressage des processus A et B
- les mappings virtuels/physiques correspondant
- ainsi que la mécanique du mode protégé, à savoir
 - traitement d'une interruption
 - consultation des tables IDT/GDT, du TSS
 - changement de niveau de privilèges
 - mise à jour du répertoire de pages et des TLBs
 - retour d'interruption/reprise d'un processus

Exemple d'ordonnancement

Introduction

La phase de démarrage

Les modes opératoires

La segmentation

Les niveaux de privilèges

Les interruptions et exceptions

La pagination

Les composants d'un OS

Conclusion

Concepts et composants d'un OS

Démarrage d'un OS

- détection de son environnement
 - cartographie mémoire (taille, zones réservées)
 - périphériques disponibles (bus PCI, ...)
- mise en place d'une GDT, IDT
- **passage en mode protégé**

Concepts et composants d'un OS

Démarrage d'un OS

- détection de son environnement
 - cartographie mémoire (taille, zones réservées)
 - périphériques disponibles (bus PCI, ...)
- mise en place d'une GDT, IDT
- **passage en mode protégé**
- mise en place d'un gestionnaire de mémoire physique (`get_page()`)
- allocation/configuration des PGD/PTBs
- **activation de la pagination**

Concepts et composants d'un OS

Démarrage d'un OS

- détection de son environnement
 - cartographie mémoire (taille, zones réservées)
 - périphériques disponibles (bus PCI, ...)
- mise en place d'une GDT, IDT
- **passage en mode protégé**
- mise en place d'un gestionnaire de mémoire physique (`get_page()`)
- allocation/configuration des PGD/PTBs
- **activation de la pagination**
- mise en place d'un gestionnaire de mémoire virtuelle (`kmalloc()`)
- initialisation des drivers de périphériques
- initialisation des composants haut niveau :
 - systèmes de fichiers (montage des disques, `rootfs`)
 - pile réseau
- démarrage de la première tâche utilisateur (`/sbin/init`)

Concepts et composants d'un OS

Des gestionnaires

- interruptions/exceptions
- appels systèmes
- mémoire physique/virtuelle
- tâches/processus/threads
- ordonnanceurs/changement de contexte
- communication entre tâches (IPC, fichiers, signaux)

Concepts et composants d'un OS

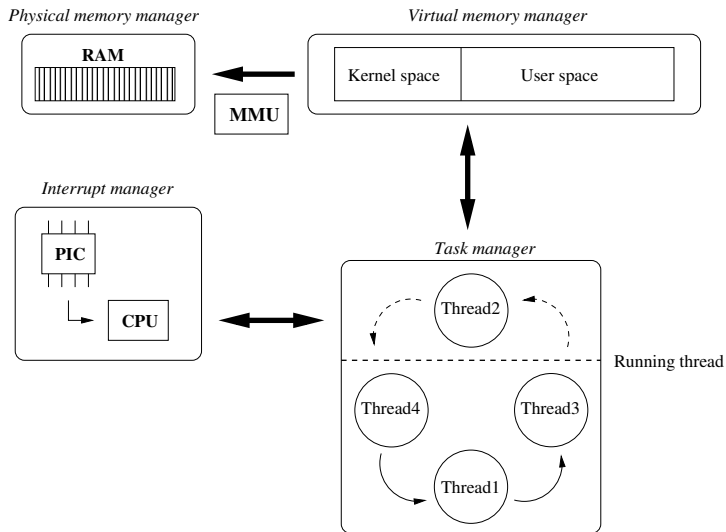
Des gestionnaires

- interruptions/exceptions
- appels systèmes
- mémoire physique/virtuelle
- tâches/processus/threads
- ordonnanceurs/changement de contexte
- communication entre tâches (IPC, fichiers, signaux)

Autres composants

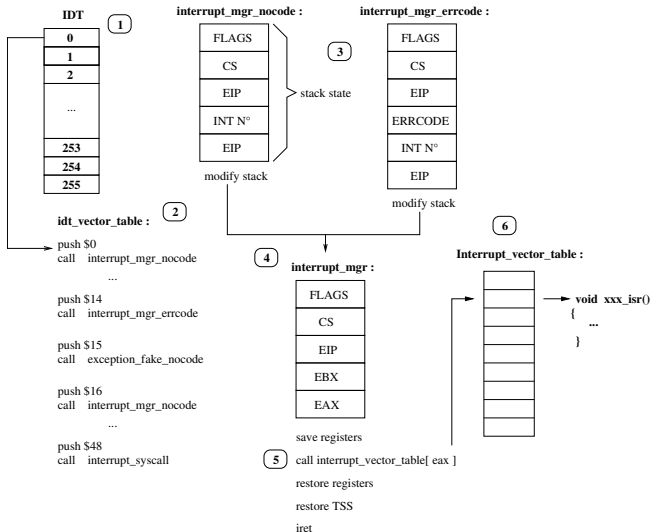
- préemption, participatifs (yield)
- verrous (spin lock, sémaphores, mutex)
- les drivers
 - timers (8254, HPET)
 - contrôleurs d'interruptions (PIC, APIC, ioAPIC)
 - Bus PCI, Memory Mapped I/O

Concepts et composants d'un OS



Le gestionnaire d'interruptions

Gestionnaire d'interruptions



Gestionnaire d'interruptions

Interruptibilité

- l'arrivée d'une interruption bloque les suivantes
- il faut les traiter le plus rapidement possible

Gestionnaire d'interruptions

Interruptibilité

- l'arrivée d'une interruption bloque les suivantes
- il faut les traiter le plus rapidement possible

Top Half/Bottom Half

- découper le traitement en 2 parties
 - top half :
 - enregistre l'interruption
 - in-interruptible
 - prépare un traitement plus poussé pour plus tard

Gestionnaire d'interruptions

Interruptibilité

- l'arrivée d'une interruption bloque les suivantes
- il faut les traiter le plus rapidement possible

Top Half/Bottom Half

- découper le traitement en 2 parties
 - top half :
 - enregistre l'interruption
 - in-interruptible
 - prépare un traitement plus poussé pour plus tard
 - bottom half :
 - traitement réel de l'interruption (paquet reseau, clavier)
 - via une tâche noyau dédiée ou non
 - interruptible

Les appels systèmes

Les appels systèmes

Porte d'entrée du noyau pour les processus

- basés sur un mécanisme matériel de changement de niveau de privilèges
 - soit une interruption : `int 0x80`
 - soit des instructions spécifiques : `sysenter`, `sysexit`
- fournissent des services bas niveau aux processus

Les appels systèmes

Porte d'entrée du noyau pour les processus

- basés sur un mécanisme matériel de changement de niveau de privilèges
 - soit une interruption : `int 0x80`
 - soit des instructions spécifiques : `sysenter`, `sysexit`
- fournissent des services bas niveau aux processus
- **ne pas confondre avec la bibliothèque C !**
 - reste en userland (`ring3`)
 - se base sur les appels systèmes
 - fournit une abstraction des concepts du noyau
 - `malloc()` \Rightarrow `sbrk()`
 - `system()` \Rightarrow `fork()` + `execve()`
 - `fopen()` \Rightarrow `open()` + `bufferisation`

Les appels systèmes

Porte d'entrée du noyau pour les processus

- basés sur un mécanisme matériel de changement de niveau de privilèges
 - soit une interruption : `int 0x80`
 - soit des instructions spécifiques : `sysenter`, `sysexit`
- fournissent des services bas niveau aux processus
- **ne pas confondre avec la bibliothèque C !**
 - reste en userland (`ring3`)
 - se base sur les appels systèmes
 - fournit une abstraction des concepts du noyau
 - `malloc()` \Rightarrow `sbrk()`
 - `system()` \Rightarrow `fork()` + `execve()`
 - `fopen()` \Rightarrow `open()` + `bufferisation`

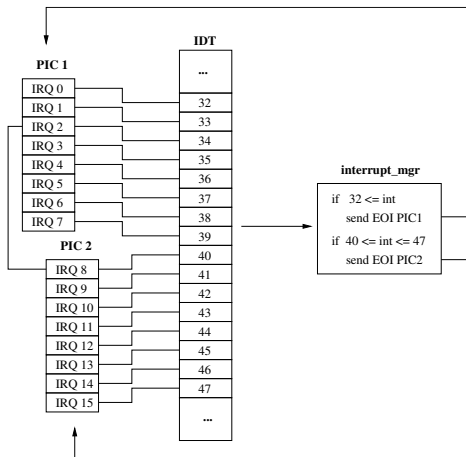
Quelques uns sous linux

- fichiers : `open()`, `close()`, `read()`, `write()`
- mémoire : `sbrk()`, `mmap()`, `munmap()`, `mprotect()`
- réseau : `socket()`, `listen()`, `bind()`, `accept()`
- tâches : `execve()`, `clone()`, `fork()`, `exit()`, `wait()`
- communication : `signal()`, `kill()`, `sigaction()`

Les périphériques

Les contrôleurs d'interruptions

- permettent de recevoir les évènements des périphériques
- historique PIC (master, slave, cascade)
- APIC, x2APIC, ioAPIC (multi-coeurs)



Les Entrées/Sorties : Port IOs, MMIOs

Deux modes de gestion des périphériques

- soit via des instructions spécifiques : `in`, `out`
- soit via un mapping mémoire directe vers le périphérique : `mmio`

Les Entrées/Sorties : Port IOs, MMIOs

Deux modes de gestion des périphériques

- soit via des instructions spécifiques : `in`, `out`
- soit via un mapping mémoire directe vers le périphérique : `mmio`

Les ports

- spécifie un port sur lequel lire/écrire
- ce port est relié au périphérique
- quelques ports célèbres
 - PIC 0x20, 0xa0
 - Clavier 0x60
 - Série 0x3f8
- assez pénible

Les Entrées/Sorties : Port IOs, MMIOs

Deux modes de gestion des périphériques

- soit via des instructions spécifiques : `in`, `out`
- soit via un mapping mémoire directe vers le périphérique : `mmio`

Les ports

- spécifie un port sur lequel lire/écrire
- ce port est relié au périphérique
- quelques ports célèbres
 - PIC 0x20, 0xa0
 - Clavier 0x60
 - Série 0x3f8
- assez pénible

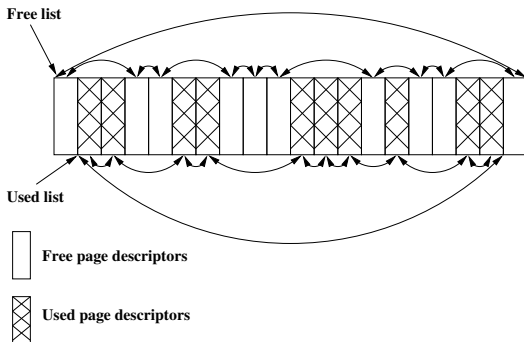
La mémoire

- la mémoire du périphérique est projetée dans l'espace linéaire
- généralement des adresse hautes 0xfffffe00
- accès à l'espace PCI, aux tables ACPI
- permet une configuration plus aisé, lit/écrit directement la mémoire

Les questionnaires de mémoire

Gestionnaire de mémoire physique

- on parle d'allocateur de pages
- référencer toutes les pages de mémoire physique
- gestion via des listes doublement chaînées des pages libres/utilisées
- structure de données qui gère des *descripteurs de pages*
 - *méta-informations*
 - pointeur suivant/précédent
 - qui la détient
 - quel est son rôle



Gestionnaire de mémoire virtuelle

Allocation/Libération

- équivalent du couple *malloc()*, *free()*
- avec des stratégies de cache (performance)
- donner rapidement des objets de taille prédéfinie
- cas dans le noyau linux
 - slab allocator
 - slub allocator

Gestionnaire de mémoire virtuelle

Allocation/Libération

- équivalent du couple *malloc()*, *free()*
- avec des stratégies de cache (performance)
- donner rapidement des objets de taille prédéfinie
- cas dans le noyau linux
 - slab allocator
 - slub allocator

API de Mapping mémoire

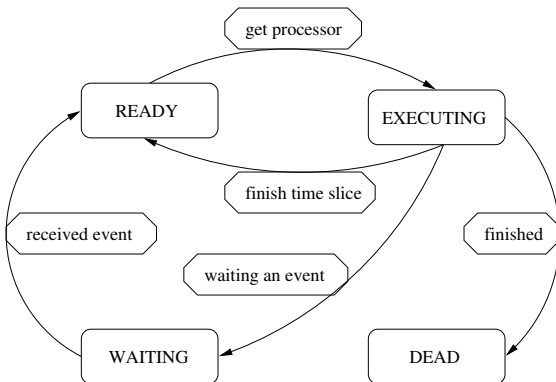
- mapper des pages via les PDE(s)/PTE(s)
- donner de la mémoire aux processus
- gérer de la mémoire partagée

L'ordonnancement

Ordonnancement : *scheduling*

État des tâches

- le noyau gère des listes de tâches
- dans des états différents (disponibles, bloquées, mortes)
- de niveaux de priorités différents



Ordonnancement : *scheduling*

Gestion du multi-tâches

- donner l'illusion que tout s'exécute en même temps
- grâce au temps partagé ... *interruption de l'horloge*
- le noyau détermine une fréquence d'interruption
- choix de la fréquence est sensible
 - trop rapide, le noyau passe son temps à être interrompu
 - trop lent, le noyau n'est pas assez interactif (pénible en GUI)
- stratégie de gestion du temps des processus
 - ordonnanceur temps réel
 - ordonnanceur avec files de priorités simples
 - dépend de la nature du système cible (station multimédias, embarqué)

Ordonnancement : coopératif ou préemptif

Coopératif

- soit le processus relâche le processeur
- quand il n'a plus rien à faire, ou le décide
- on parle de multi-tâches *coopératif*
- `yield()`

Ordonnancement : coopératif ou préemptif

Coopératif

- soit le processus relâche le processeur
- quand il n'a plus rien à faire, ou le décide
- on parle de multi-tâches *coopératif*
- `yield()`

Préemptif

- le noyau alloue un *quantum* de temps à chaque tâche
- le processus est interrompu par le noyau à l'expiration du quantum
- on parle de multi-tâches *préemptif*
- différentes parties peuvent être préemptibles :
 - les processus utilisateurs uniquement
 - le noyau uniquement
 - les deux

Ordonnancement : *scheduling*

La pile noyau

- différentes natures d'entités ordonnançables
 - processus utilisateurs, threads d'un processus utilisateurs
 - threads du noyau (idle, bottom half)
- chaque entité prend un chemin d'exécution unique
- nécessaire de disposer d'une pile par entité
- on parle de pile noyau (*kernel stack*)

Ordonnancement : *scheduling*

La pile noyau

- différentes natures d'entités ordonnançables
 - processus utilisateurs, threads d'un processus utilisateurs
 - threads du noyau (idle, bottom half)
- chaque entité prend un chemin d'exécution unique
- nécessaire de disposer d'une pile par entité
- on parle de pile noyau (*kernel stack*)

Le changement de contexte : *context switch*

- on va sauver l'état (registres) de la tâche sortante, dans sa pile noyau

Ordonnancement : *scheduling*

La pile noyau

- différentes natures d'entités ordonnançables
 - processus utilisateurs, threads d'un processus utilisateurs
 - threads du noyau (idle, bottom half)
- chaque entité prend un chemin d'exécution unique
- nécessaire de disposer d'une pile par entité
- on parle de pile noyau (*kernel stack*)

Le changement de contexte : *context switch*

- on va sauver l'état (registres) de la tâche sortante, dans sa pile noyau
- on va charger l'état de la tâche entrante, depuis sa pile noyau

Ordonnancement : *scheduling*

La pile noyau

- différentes natures d'entités ordonnançables
 - processus utilisateurs, threads d'un processus utilisateurs
 - threads du noyau (idle, bottom half)
- chaque entité prend un chemin d'exécution unique
- nécessaire de disposer d'une pile par entité
- on parle de pile noyau (*kernel stack*)

Le changement de contexte : *context switch*

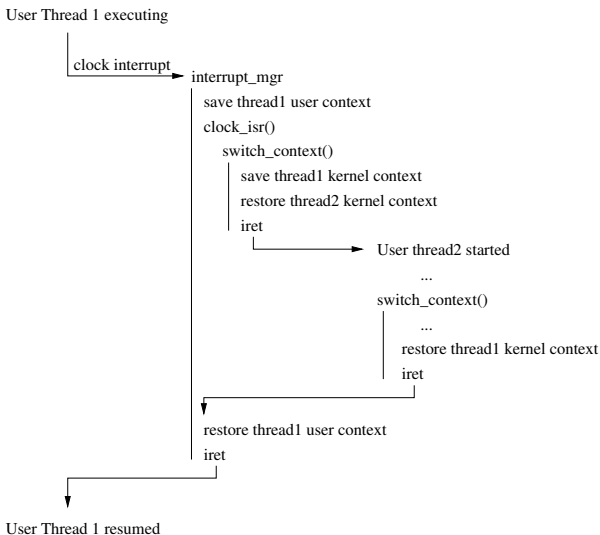
- on va sauver l'état (registres) de la tâche sortante, dans sa pile noyau
- on va charger l'état de la tâche entrante, depuis sa pile noyau
- ce changement de pile s'appelle un *context switch*
- généralement accompagné d'un changement d'espace d'adressage
- mais uniquement pour les processus utilisateurs
- les threads noyaux n'ont besoin que de la mémoire du noyau
- or, elle est disponible (mappée) dans tous les espaces d'adressage

Ordonnancement : changement de contexte

refaire le schéma avec des piles kernels pour p1,p2

montrer la mise à jour de TSS.esp0

insister sur le fait que `schedule()` ne fait que changer de pile



Ordonnancement : exclusion mutuelle

Multitâches et ressources partagées

- plusieurs tâches accèdent simultanément (multi-coeurs, smp) aux mêmes ressources
- nécessité de gérer une sorte d'exclusivité d'accès

Ordonnancement : exclusion mutuelle

Multitâches et ressources partagées

- plusieurs tâches accèdent simultanément (multi-coeurs, smp) aux mêmes ressources
- nécessité de gérer une sorte d'exclusivité d'accès
- principe des sémaphores (IPC)
- équivalents à quelques détails près aux *spin-lock()*, *mutex()*
- bloquer des tâches tant qu'une ressource est utilisée
- libérer une tâche dès que la ressource est disponible

doit-être implémenté par la tâche ! ... *race condition*

La communication

Communication : les fichiers

L'API classique

- localisation par chemin : nom, path
- droits associés au fichier : utilisateur, groupe, rwx
- `open("/etc/passwd", O_RDONLY)`

Communication : les fichiers

L'API classique

- localisation par chemin : nom, path
- droits associés au fichier : utilisateur, groupe, rwx
- `open("/etc/passwd", O_RDONLY)`
- ensuite opérations lecture/écriture/déplacement
- `read()`, `write()`, `lseek()`
- noyau gère la position dans le fichier
- il gère également l'allocation de son contenu (inode)

Communication : les fichiers

L'API classique

- localisation par chemin : nom, path
- droits associés au fichier : utilisateur, groupe, rwx
- `open("/etc/passwd", O_RDONLY)`
- ensuite opérations lecture/écriture/déplacement
- `read()`, `write()`, `lseek()`
- noyau gère la position dans le fichier
- il gère également l'allocation de son contenu (inode)

Gérer un fichier comme une zone de mémoire

- demander au noyau d'ouvrir le fichier en mémoire
- lire/écrire dedans comme dans un buffer
- sans passer par `read()`, `write()`, `lseek()`

une idée ?

Communication : les fichiers

L'API classique

- localisation par chemin : nom, path
- droits associés au fichier : utilisateur, groupe, rwx
- `open("/etc/passwd", O_RDONLY)`
- ensuite opérations lecture/écriture/déplacement
- `read()`, `write()`, `lseek()`
- noyau gère la position dans le fichier
- il gère également l'allocation de son contenu (inode)

Gérer un fichier comme une zone de mémoire

- demander au noyau d'ouvrir le fichier en mémoire
- lire/écrire dedans comme dans un buffer
- sans passer par `read()`, `write()`, `lseek()`
une idée ?
- *memory mapped files* : `mmap()`
- sert à mapper de la mémoire (page par page)
- mais permet également de mapper un fichier en mémoire

Introduction

La phase de démarrage

Les modes opératoires

La segmentation

Les niveaux de privilèges

Les interruptions et exceptions

La pagination

Les composants d'un OS

Conclusion

Conclusion

Cela reste un rappel ...

- non exhaustif
- intéressez-vous à l'ARM
- lisez des livres *conceptuels*
- lisez des livres d'implémentations
 - Linux, Windows, BSDs
 - un-Higgs H*
<https://sites.google.com/site/stephaneduverger/publications>
- lisez du code
 - kernel.org (git, <http://lxr.free-electrons.com>)
 - github.com (rechercher : kernel, os, ...)
 - OSdev.org
<http://wiki.osdev.org>
- ou des forums, blogs, ...
 - <http://www.kernelmode.info>
 - <http://www.osdever.net/tutorials>

Quelques liens (1)

Architecture matérielle

- Les manuels Intel
<https://www-ssl.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>
- Les manuels AMD et les BIOS & Kernel Developer's Guide
<http://developer.amd.com/resources/documentation-articles/developer-guides-manuals/>
- What every programmer should know about memory, Ulrich Drepper
<http://www.akkadia.org/drepper/cpumemory.pdf>
- Protected Mode Software Architecture, Tom Shanley

Concepts

- Les systèmes d'exploitation - conception et mise en oeuvre, Andrew Tanenbaum
- Operating System Concepts, 8th Edition, Avi Silberschatz, Peter Baer Galvin, Greg Gagne
<http://codex.cs.yale.edu/avi/os-book/OS8/os8c/slide-dir/>

Quelques liens (2)

Implémentation

- Understanding the Linux Kernel, 3rd Edition, Bovet & Cesati
- Windows Internals, 6th Edition, Russinovich & Ionescu
- The Design and Implementation of the 4.4 BSD Operating System, McKusick
- UNIX Internals : The New Frontiers, Vahalia Uresh
- Lion's commentary on Unix 6th Edition
- MIT xv6 OS
<http://pdos.csail.mit.edu/6.828/2014/xv6/book-rev8.pdf>
- Créez votre OS (Linux magazine)
<http://sos.enix.org/fr/SOSDownload>

Vos commentaires

Votre avis compte

- remarques ?
- améliorations ?
- manquements ?
- suggestions ?
- questions ?

stephane.duverger@gmail.com