

Programmation Système

Département Informatique

IUT2 de Grenoble

BUT2 - Ressource 3.05

Organisation de l'enseignement R3.05

semaine	Cours	TD	TP
01	1h30		
02-04	1h30	2h	2h

Evaluation [20% de l'UE3]

examen sur table 2h

Encadrants [**Prenom.Nom@univ-grenoble-alpes.fr**]


Cours TD TP : **Cedric.Gerot** ; TD TP : **Laurent.Bonnaud**
Jean-Pierre.Chevallet **Pierre-Francois.Dutot**

Langages utilisés


Shell, C.

Organisation du cours

Cours conçu entre autres à partir de :

 Silberschatz, Galvin, and Gagne
Operating System Concepts - 10th Edition.
Wiley, 2018.

Autre lecture très recommandable :

 Tanenbaum and Bos
Modern Operating Systems - 5th Edition.
Prentice Hall, 2023.

Organisation du cours

1. Système d'exploitation
2. Processus
3. Partage des ressources
4. Système de Gestion de Fichiers
5. Entrées/Sorties

Organisation du cours

1. Système d'exploitation
2. Processus
3. Partage des ressources
4. Système de Gestion de Fichiers
5. Entrées/Sorties

1. Système d'Exploitation

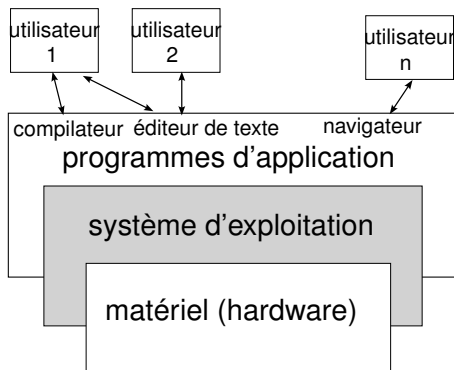
1.1. Définition

1.2. Retour sur les composants matériels

1.3. Fonctions d'un système d'exploitation

Définition d'un système d'exploitation [R1.04]

Un ensemble de programmes
qui permettent aux **applications lancées par les utilisateurs**
de fonctionner sur les **composants matériels** constituant un ordinateur
tout en veillant à exploiter au mieux ces ressources matérielles.



Définition d'un système d'exploitation [R1.04]

2 critères a priori contradictoires à satisfaire :

- qualité du service offert aux utilisateurs
- efficacité de l'exploitation des ressources

1. Système d'Exploitation

1.1. Définition

1.2. Retour sur les composants matériels

1.3. Fonctions d'un système d'exploitation

Composants matériels d'un ordinateur [R1.03]

Les différents composants peuvent s'organiser selon :

- Processeur(s)
- Mémoires
- Périphériques
- Bus

Composants matériels d'un ordinateur [R1.03]

Les différents composants peuvent s'organiser selon :

- Processeur(s)
- Mémoires
- Périphériques
- Bus

Composants - Processeur [R1.03]

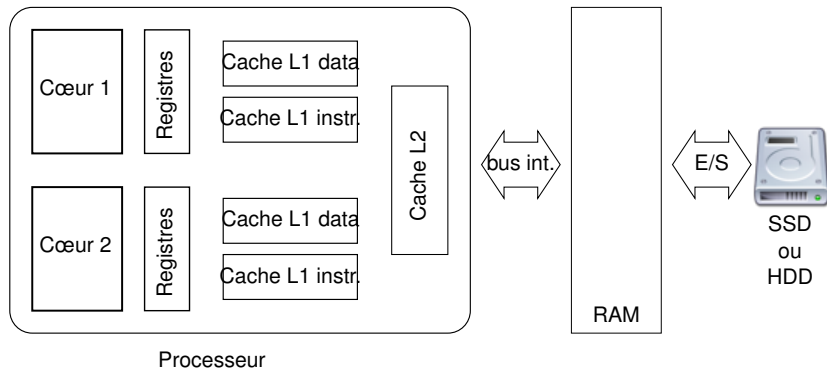
- centre de calcul et de manipulation de données
- est gouverné par une horloge
- lit des instructions en mémoire et les exécute :
 - accès mémoire
 - opération arithmétique
 - opération logique
 - contrôle (branchement etc.)
- utilise des registres
 - de données (pour les manipuler localement)
 - d'adresses :
 - IP (Instruction Pointeur) : compteur ordinal
 - SP (Stack Pointer) : adresse de pile
 - CS (Code Segment) : adresse du segment de code

Composants - Processeur [R1.03]

- centre de calcul et de manipulation de données
- est gouverné par une horloge
- lit des instructions en mémoire et les exécute :
 - accès mémoire
 - opération arithmétique
 - opération logique
 - contrôle (branchement etc.)
- utilise des registres
 - de données (pour les manipuler localement)
 - d'adresses :
 - IP (Instruction Pointeur) : compteur ordinal
 - SP (Stack Pointer) : adresse de pile
 - CS (Code Segment) : adresse du segment de code

Composants - Mémoires [R2.04a]

Exemple de hiérarchie

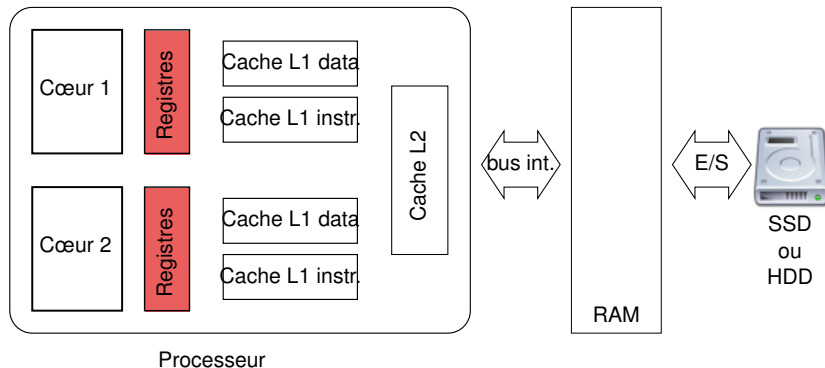


Composants - Mémoires [R2.04a]

Exemple de hiérarchie

taille : < 1Kio

tps d'accès : 0,25 à 0,5 ns

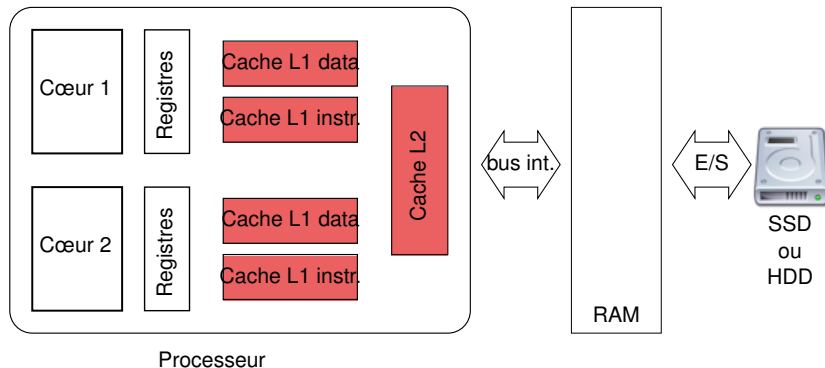


Composants - Mémoires [R2.04a]

Exemple de hiérarchie

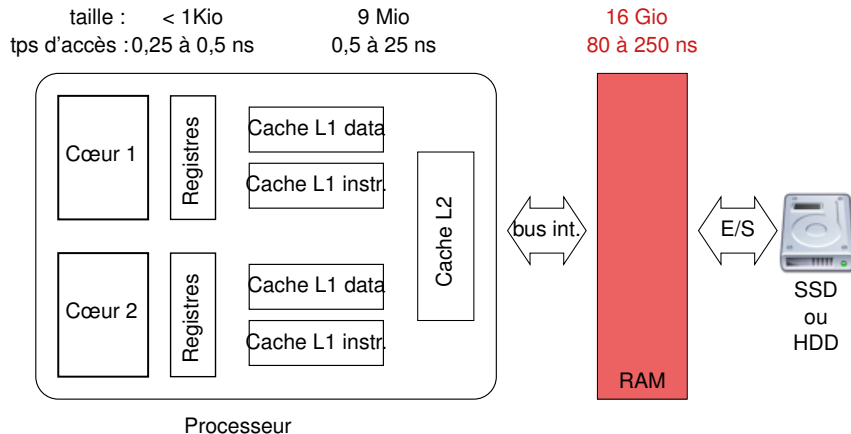
taille : < 1Kio
tps d'accès : 0,25 à 0,5 ns

9 Mio
0,5 à 25 ns



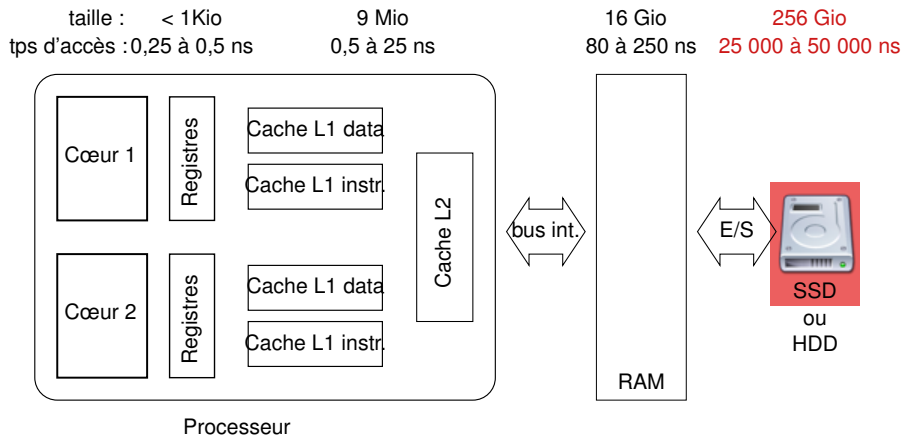
Composants - Mémoires [R2.04a]

Exemple de hiérarchie



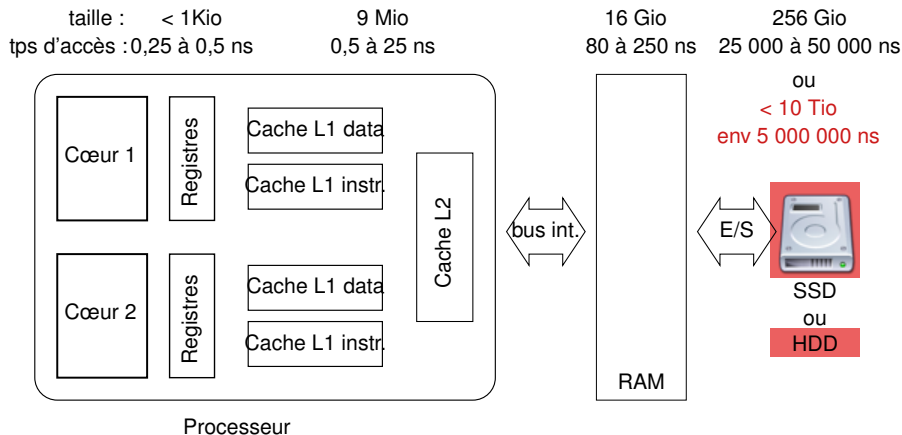
Composants - Mémoires [R2.04a]

Exemple de hiérarchie



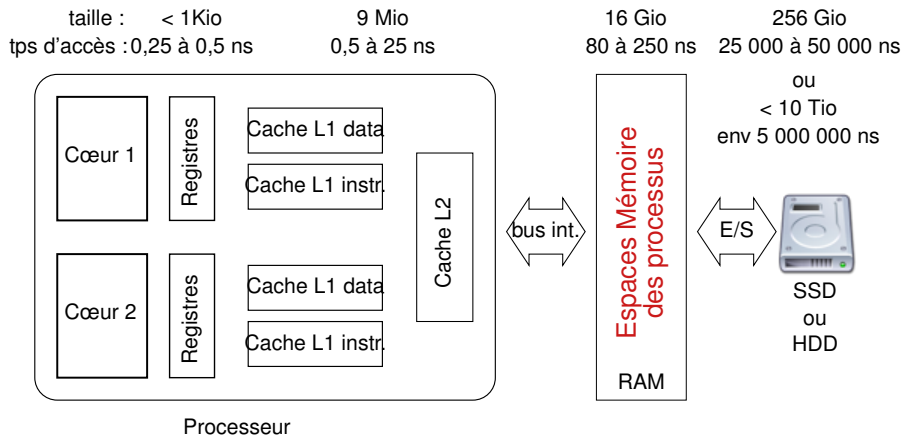
Composants - Mémoires [R2.04a]

Exemple de hiérarchie



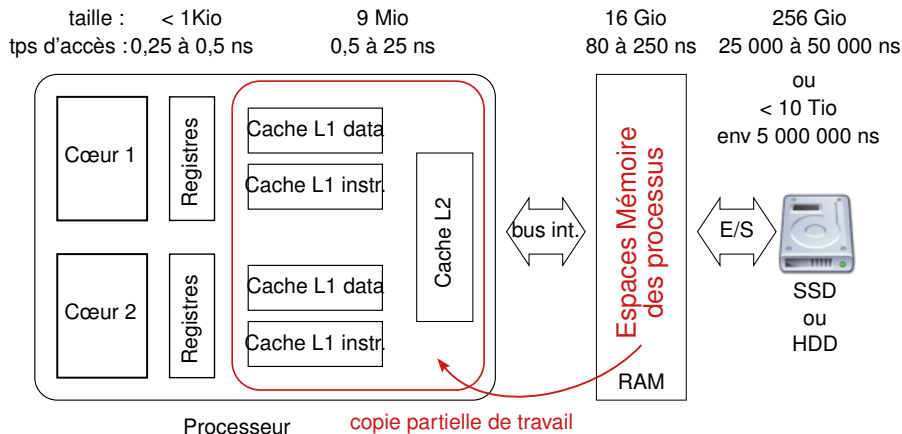
Composants - Mémoires [R2.04a]

Exemple de hiérarchie



Composants - Mémoires [R2.04a]

Exemple de hiérarchie



Composants - Mémoires [R2.04a]

Registres

- permettent au CPU d'exécuter ses instructions
- compilateur (ou programmeur) : y conserver les données manipulées

Caches

- sur matériel accessible rapidement (mais cher)
- version de travail, partielle de la RAM
- localité temporelle : les données utilisées changent peu
- localité spatiale : les données utilisées se suivent (ex : instr.)
- Cache L1 : spécialisé selon data ou instr (localités spatiales ≠)
- géré par le matériel (m-à-j en cas d'écriture etc.)
- programmeur / compilateur : espace de travail contenu en cache
 - manipuler les données en blocs
 - prévoir les données susceptibles d'être utilisées (*prefetch*)

Composants - Mémoires [R2.04a]

Registres

- permettent au CPU d'exécuter ses instructions
- compilateur (ou programmeur) : y conserver les données manipulées

Caches

- sur matériel accessible rapidement (mais cher)
- version de travail, partielle de la RAM
- **localité temporelle** : les données utilisées changent peu
- **localité spatiale** : les données utilisées se suivent (ex : instr.)
- Cache L1 : spécialisé selon data ou instr (localités spatiales \neq)
- géré par le matériel (m-à-j en cas d'écriture etc.)
- programmeur / compilateur : *espace de travail* contenu en cache
 - manipuler les données en blocs
 - prévoir les données susceptibles d'être utilisées (*prefetch*)

Composants - Mémoires [R2.04a]

Registres

- permettent au CPU d'exécuter ses instructions
- compilo (ou programmeur) : y conserver les données manipulées

Caches

- sur matériel accessible rapidement (mais cher)
- version de travail, partielle de la RAM
- **localité temporelle** : les données utilisées changent peu
- **localité spatiale** : les données utilisées se suivent (ex : instr.)
- Cache L1 : spécialisé selon data ou instr (localités spatiales \neq)
- géré par le matériel (m-à-j en cas d'écriture etc.)
- programmeur / compilo : *espace de travail* contenu en cache
 - manipuler les données en blocs
 - prévoir les données susceptibles d'être utilisées (*prefetch*)

Composants - Mémoires [R2.04a]

Registres

- permettent au CPU d'exécuter ses instructions
- compilo (ou programmeur) : y conserver les données manipulées

Caches

- sur matériel accessible rapidement (mais cher)
- version de travail, partielle de la RAM
- **localité temporelle** : les données utilisées changent peu
- **localité spatiale** : les données utilisées se suivent (ex : instr.)
- Cache L1 : spécialisé selon data ou instr (localités spatiales \neq)
- géré par le matériel (m-à-j en cas d'écriture etc.)
- programmeur / compilo : *espace de travail* contenu en cache
 - manipuler les données en blocs
 - prévoir les données susceptibles d'être utilisées (*prefetch*)

Composants - Mémoires [R2.04a]

Registres

- permettent au CPU d'exécuter ses instructions
- compilateur (ou programmeur) : y conserver les données manipulées

Caches

- sur matériel accessible rapidement (mais cher)
- version de travail, partielle de la RAM
- **localité temporelle** : les données utilisées changent peu
- **localité spatiale** : les données utilisées se suivent (ex : instr.)
- Cache L1 : spécialisé selon data ou instr (localités spatiales \neq)
- géré par le matériel (m-à-j en cas d'écriture etc.)
- programmeur / compilateur : *espace de travail* contenu en cache
 - manipuler les données en blocs
 - prévoir les données susceptibles d'être utilisées (*prefetch*)

1. Système d'Exploitation

1.1. Définition

1.2. Retour sur les composants matériels

1.3. Fonctions d'un système d'exploitation

Fonctions d'un système d'exploitation

- exécuter un programme
 - charger un programme en mémoire
 - l'exécuter (=> un processus)
 - le permettre de s'achever (gestion d'erreur si fin anormale)
- donner l'illusion d'une machine démesurée
 - multiples processus *simultanés*
 - mémoire "*infinie*"
- gérer et partager les ressources

Fonctions d'un système d'exploitation

- exécuter un programme
 - charger un programme en mémoire
 - l'exécuter (=> un processus)
 - le permettre de s'achever (gestion d'erreur si fin anormale)
- donner l'illusion d'une machine démesurée
 - multiples processus *simultanés*
 - mémoire "*infinie*"
- gérer et partager les ressources

Organisation du cours

1. Système d'exploitation

2. Processus

3. Partage des ressources

4. Système de Gestion de Fichiers

5. Entrées/Sorties

2. Processus

2.1. Révisions R1.04

2.2. Descripteur de processus

2.3. Protection : mode noyau et appels systèmes

2.4. En pratique : **fork()**, **exit()**, **waitpid()**, **execvp()**

2. Processus

2.1. Révisions R1.04

2.2. Descripteur de processus

2.3. Protection : mode noyau et appels systèmes

2.4. En pratique : `fork()`, `exit()`, `waitpid()`, `execvp()`

Définition d'un processus [R1.04]

Définition

processus = exécution d'un programme

Toute exécution est demandée par un autre processus

- chaque processus a un et un seul *père* : celui qui a demandé son exécution
- les processus sont organisés en un arbre

Arbre de processus [R1.04]

```

systemd--accounts-daemon
    |-acpid
    |-agent
    [...]
    |-gam_server
    |-gmenudbusmenupr
    [...]
    |
    | -ksmsserver--konsole--bash--pstree
    |                                     `-bash--gedit
    |-kwin_x11
    |-lightdm--Xorg
    |           |-Xtigervnc
    |           |-lightdm--lightdm-gtk-gre
    |           |-lightdm
    |           `-lightdm--startplasma-x11--ssh-agent
    [...]
    `-xsettingsd

```

Arbre de processus [R1.04]

A la destruction d'un père

- W*s : ses fils meurent avec lui
- U*x : ses fils sont adoptés par **systemd** / **init**

```
systemd+-accounts-daemon
|-acpid
|-agent
[...]|
|-gam_server
|-gmenudbusmenupr
[...]|
|   |-ksmsserver--konsole--bash--pstree
|   |                                     `--bash--gedit
|   |-kwin_x11
[...]|
|   `--xsettingsd
```

```
systemd+-accounts-daemon
|-acpid
|-agent
[...]|
|-gam_server
|   |-gedit
|   |-gmenudbusmenupr
[...]|
|   |-ksmsserver--konsole--bash--pstree
|   |-kwin_x11
[...]|
|   `--xsettingsd
```

Attributs et ressources d'un processus [R1.04]

Attributs

- PID : Process IDentifier ; PPID : Parent Process ID
- UID : User IDentifier
- S : State (état)
- PR : PRiority ; NI : NIceness (courtoisie)
- etc.

Ressources

- zone mémoire ;
- fichiers ouverts (3 par défaut : `stdin`, `stdout`, `stderr`) ;
- etc.

Définition d'un processus

processus = exécution d'un programme
= image mémoire + descripteur

Attributs et ressources d'un processus [R1.04]

Attributs

- PID : Process IDentifier ; PPID : Parent Process ID
- UID : User IDentifier
- S : State (état)
- PR : PRiority ; NI : NIceness (courtoisie)
- etc.

Ressources

- zone mémoire ;
- fichiers ouverts (3 par défaut : `stdin`, `stdout`, `stderr`) ;
- etc.

Définition d'un processus

processus = exécution d'un programme
= image mémoire + descripteur

2. Processus

2.1. Révisions R1.04

2.2. Descripteur de processus

2.3. Protection : mode noyau et appels systèmes

2.4. En pratique : `fork()`, `exit()`, `waitpid()`, `execvp()`

Descripteur de processus

identifiant
état
priorité
copie de registres
informations gestion mémoire
informations ressources (fichiers...)

Descripteur de Processus
(*Process Control Block*)

Où en est l'exécution du programme ?

- compteur ordinal : adresse de l'instruction en cours d'exécution
- registre de pile : adresse de la tête de pile

Descripteur de processus

identifiant
état
priorité
copie de registres
informations gestion mémoire
informations ressources (fichiers...)

Descripteur de Processus
(*Process Control Block*)

Où en est l'exécution du programme ?

- compteur ordinal : adresse de l'instruction en cours d'exécution
- registre de pile : adresse de la tête de pile

Descripteur de processus

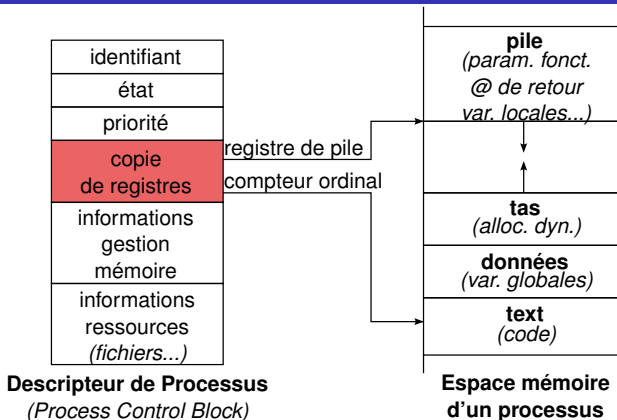
identifiant
état
priorité
copie de registres
informations gestion mémoire
informations ressources (fichiers...)

Descripteur de Processus
(*Process Control Block*)

Où en est l'exécution du programme ?

- compteur ordinal : adresse de l'instruction en cours d'exécution
- registre de pile : adresse de la tête de pile

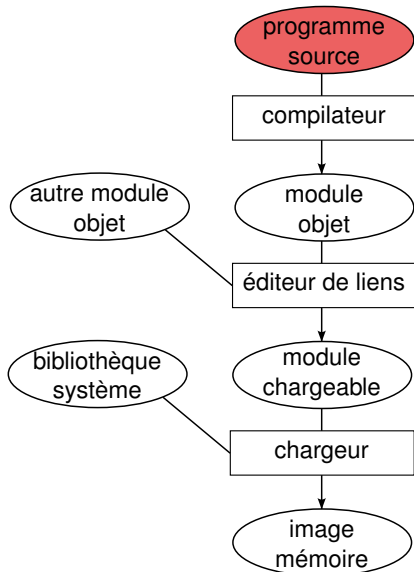
Descripteur de processus



Où en est l'exécution du programme ?

- compteur ordinal : adresse de l'instruction en cours d'exécution
- registre de pile : adresse de la tête de pile

Image mémoire d'un processus [R3.16]

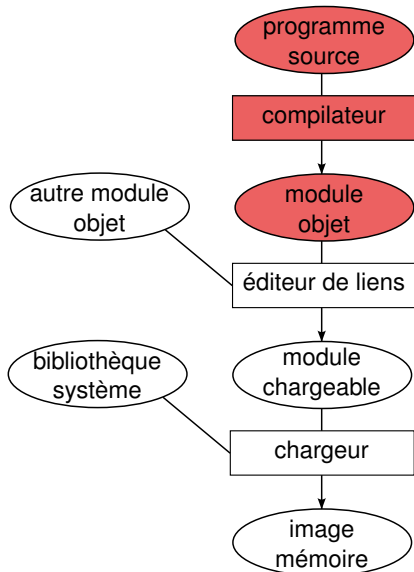


Fichier prog.c

```
#define N 16384
char T[N];

void main(void)
{
    for(int i=0; i<N; i++)
        T[i]='*';
}
```

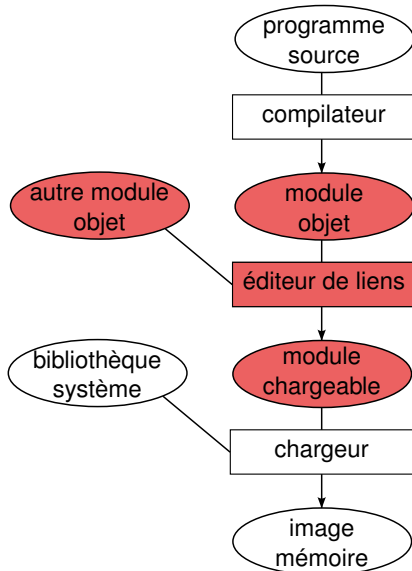
Image mémoire d'un processus [R3.16]



Fichier `prog.o`

```
gcc -c prog.c
```

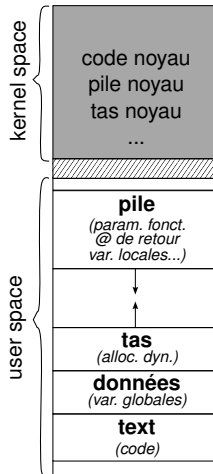
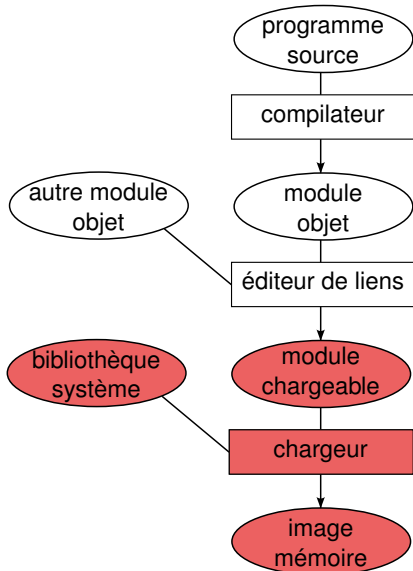
Image mémoire d'un processus [R3.16]



Fichier `prog`

```
gcc -o prog prog.o
```

Image mémoire d'un processus [R3.16]



Espace mémoire d'un processus
(adresses logiques)

2. Processus

2.1. Révisions R1.04

2.2. Descripteur de processus

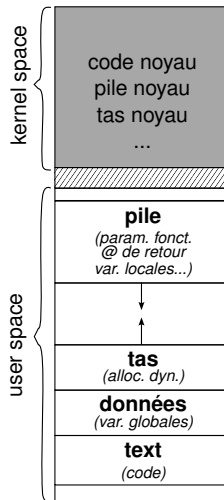
2.3. Protection : mode noyau et appels systèmes

2.4. En pratique : **fork()**, **exit()**, **waitpid()**, **execvp()**

Un accès au matériel protégé : user/kernel mode

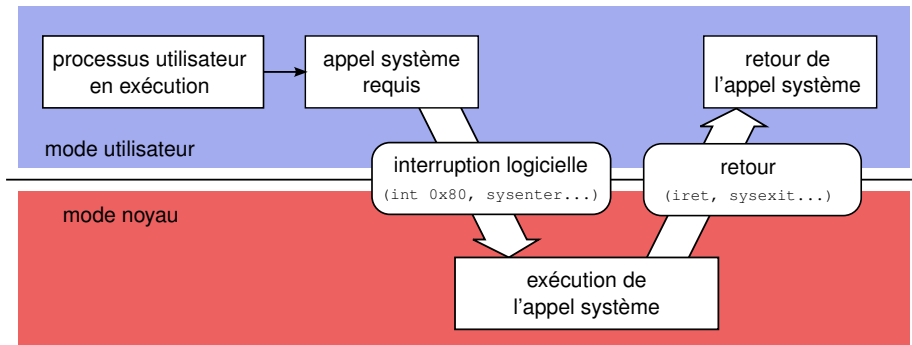
Deux espaces : user/noyau

- **code du noyau** : partagé par toutes les images mémoire processus
- processus en **mode user** : seul l'espace user accessible
- \neq degrés d'isolation possibles : un ou deux espaces mémoires pour chaque processus
- pour accéder à l'espace noyau : **appel système \Rightarrow mode noyau**



Espace mémoire d'un processus
(adresses logiques)

Appels systèmes : Mode dual



Pour isoler de l'utilisateur les ressources critiques de la machine

Deux modes d'exécution de processus :

- mode utilisateur
- mode noyau

CPL (*Current Privilege Level*) : un champ du registre CS (*Code Segment*)

Appels système pour la commande `cp toto titi`

Exécuter le programme `cp`

```
execve("/bin/cp", "toto", "titi")
```

Consulter les propriétés du fichier destination

```
stat64("titi", ...) = -1 ENOENT
```

Consulter les propriétés du fichier source

```
stat64("toto", ...) = 0
```

Ouvrir le fichier source

```
open("toto", ...) = 3
```

Créer le fichier destination

```
open("titi", ...) = 4
```

Copier

```
read(3, ...)
```

```
write(4, ...)
```

Fermer les fichiers

```
close(4)
```

```
close(3)
```

Terminer normalement

```
exit_group(0)
```

Programmation - Appels systèmes

<https://syscalls.mebeim.net/>

Définition

- routines mises à disposition du programmeur système
- écrites en C (ou assembleur)
- permettent des actions élémentaires

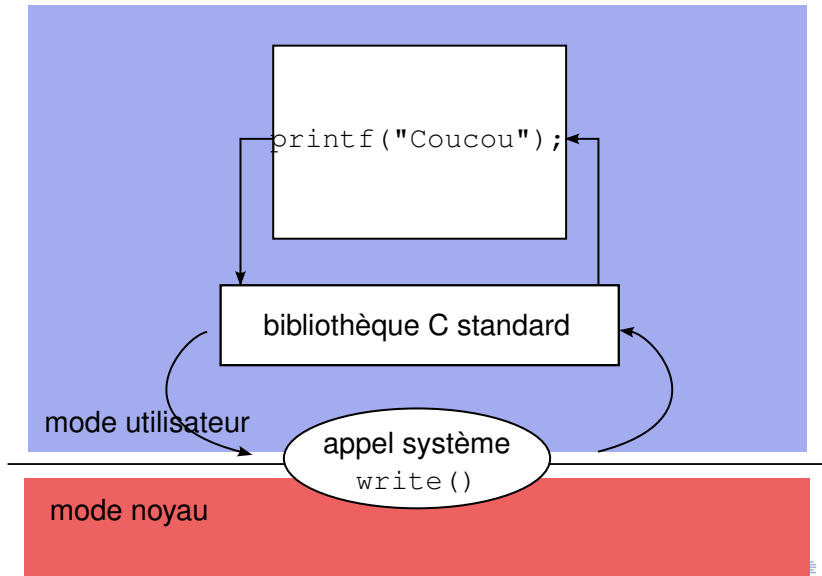
API : *Application Programming Interface*

interface de programmation pour faciliter l'utilisation de ces routines

Exemples d'API

- Bibliothèque C standard (ex : `printf()`)
- POSIX (Linux, Windows OS, Mac OSX)
- Win32 (Windows OS)

L'API aux appels système permet cette séparation



2. Processus

2.1. Révisions R1.04

2.2. Descripteur de processus

2.3. Protection : mode noyau et appels systèmes

2.4. En pratique : **fork()**, **exit()**, **waitpid()**, **execvp()**

Création de processus : `fork()`



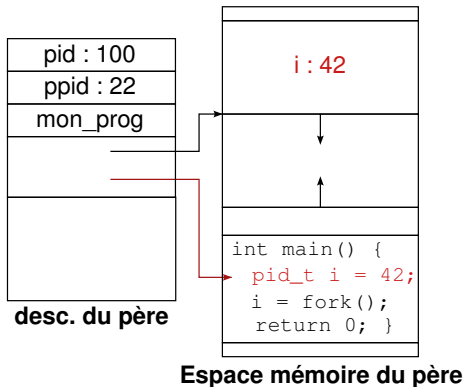
Synopsis

```
#include <unistd.h>
pid_t fork(void);
```

Description

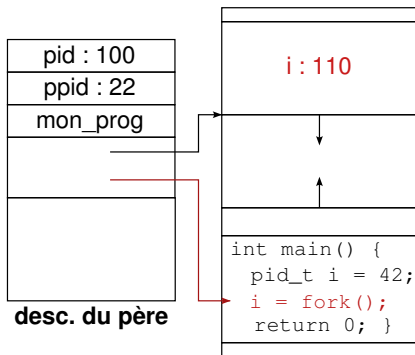
- crée un processus fils sans arrêter le père
- **tout** l'environnement du père est **dupliqué**,
- seules différences :
 - **PID** et **PPID**
 - la valeur de retour de la fonction `fork()` =
 - **0** dans l'environnement du fils
 - **PID du fils** dans l'environnement du père

Création de processus : `fork()`

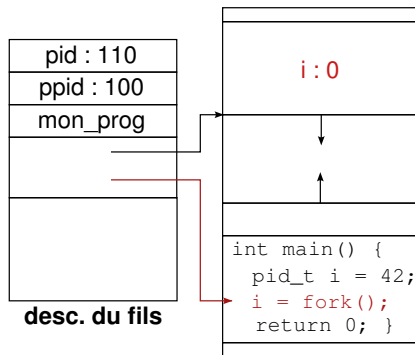


```
systemd+-accounts-daemon
|
[... ]
|-ksmserver--konsole--bash--mon_prog
[... ]
```

Création de processus : `fork()`



Espace mémoire du père



Espace mémoire du fils

```
systemd+-accounts-daemon
```

```
|
```

```
[...]
```

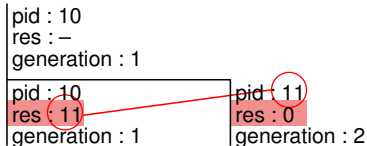
```
| -kmsserver--konsole--bash--mon_prog--mon_prog
```

```
[...]
```


Création de processus : `fork()`

Exemple : 1 père, 3 fils, 3 petits-fils,
1 arrière-petit-fils

```
int main(void)
{
    int generation = 1;
    pid_t res;
    res = fork();
    if (res==0) generation ++;
    printf("%d \n", generation);
    return EXIT_SUCCESS;
}
```

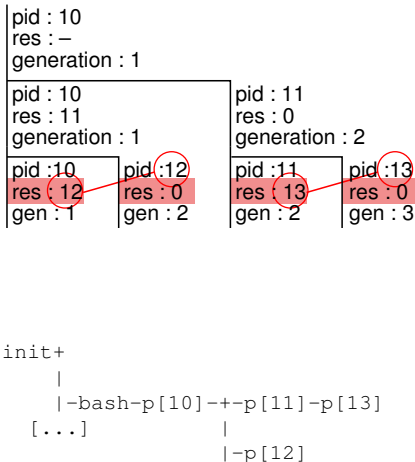


```
init+
    |
    | -bash-p[10]--p[11]
    [...]
    
```

Création de processus : `fork()`

Exemple : 1 père, 3 fils, 3 petits-fils,
1 arrière-petit-fils

```
int main(void)
{
    int generation = 1;
    pid_t res;
    res = fork();
    if (res==0) generation ++;
    res = fork();
    if (res==0) generation ++;
    printf("%d \n", generation);
    return EXIT_SUCCESS;
}
```



Création de processus : `fork()`



Exemple : 1 père, 3 fils, 3 petits-fils,
1 arrière-petit-fils

```
int main(void)
{
    int generation = 1;
    pid_t res;
    res = fork();
    if (res==0) generation ++;
    res = fork();
    if (res==0) generation ++;
    res = fork();
    if (res==0) generation ++;
    printf("%d \n", generation);
    return EXIT_SUCCESS;
}
```

```
pid : 10
res : -
generation : 1
```

```
pid : 10
res : 11
generation : 1
```

```
pid : 11
res : 0
generation : 2
```

```
pid : 10
res : 12
gen : 1
```

```
pid : 12
res : 0
gen : 2
```

```
pid : 11
res : 13
gen : 2
```

```
pid : 13
res : 0
gen : 3
```

```
p : 10 p : 14
r : 14 r : 0
```

```
p : 12 p : 16
r : 16 r : 0
```

```
p : 11 p : 15
r : 15 r : 0
```

```
p : 13 p : 17
r : 17 r : 0
```

```
g : 1 g : 2
```

```
g : 2 g : 3
```

```
g : 2 g : 3
```

```
g : 3 g : 4
```

```
init+
```

```
| -bash-p[10]--p[11]-p[13]-p[17]
```

```
[...]
```

```
| -p[15]
```

```
| -p[12]-p[16]
```

```
`-p[14]
```

Terminaison d'un processus : `exit()`

Synopsis

```
#include <stdlib.h>
void exit(int status);
```

Description

- provoque une terminaison normale du processus, les ressources utilisées sont libérées, mais son descripteur reste ;
- **status** définit la valeur renvoyée au père via un champ du descripteur de processus du fils (utiliser **EXIT_SUCCESS**, **EXIT_FAILURE** ou **errno**)

Réaction du père :

- il a déclaré ignorer la terminaison de ses fils : le descripteur du fils disparaît
- il attend la terminaison de ses fils : le descripteur du fils disparaît
- sinon : le fils devient **zombie**

Terminaison d'un processus : `exit()`

Synopsis

```
#include <stdlib.h>
void exit(int status);
```

Description

- provoque une terminaison normale du processus, les ressources utilisées sont libérées, mais son descripteur reste ;
- **status** définit la valeur renvoyée au père via un champ du descripteur de processus du fils (utiliser **EXIT_SUCCESS**, **EXIT_FAILURE** ou **errno**)

Réaction du père :

- il a déclaré ignorer la terminaison de ses fils : le descripteur du fils disparaît
- il attend la terminaison de ses fils : le descripteur du fils disparaît
- sinon : le fils devient **zombie**

Terminaison d'un processus : `exit()`

Synopsis

```
#include <stdlib.h>
void exit(int status);
```

Description

- provoque une terminaison normale du processus, les ressources utilisées sont libérées, mais son descripteur reste ;
- **status** définit la valeur renvoyée au père via un champ du descripteur de processus du fils
(utiliser **EXIT_SUCCESS**, **EXIT_FAILURE** ou **errno**)

Réaction du père :

- il a déclaré ignorer la terminaison de ses fils : le descripteur du fils disparaît
- il attend la terminaison de ses fils : le descripteur du fils disparaît
- sinon : le fils devient **zombie**

Terminaison d'un processus : `exit()`

Synopsis

```
#include <stdlib.h>
void exit(int status);
```

Description

- provoque une terminaison normale du processus, les ressources utilisées sont libérées, mais son descripteur reste ;
- **status** définit la valeur renvoyée au père via un champ du descripteur de processus du fils
(utiliser **EXIT_SUCCESS**, **EXIT_FAILURE** ou **errno**)

Réaction du père :

- il a déclaré ignorer la terminaison de ses fils : le descripteur du fils disparaît
- il attend la terminaison de ses fils : le descripteur du fils disparaît
- sinon : le fils devient **zombie**

Terminaison d'un processus : `waitpid()`



Synopsis

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *wstatus);
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

Description

- peut suspendre l'exécution du processus (selon `options`)...
- jusqu'au changement d'état de l'un de ses fils (identifié par `pid`)
- et récupère alors des informations sur lui (via `wstatus`)

Terminaison d'un processus : `waitpid()`

Exemple 1 : père boucle (état R), fils se termine (état Z)

```
int main(void)
{
    pid_t res; int i = 0;
    res = fork();
    if (res != 0)
        while (1) {
            printf("%d\n", i);
            i++; }
    return EXIT_SUCCESS;
}
```

Exemple 2 : père attend (état S), fils boucle (état R)

```
int main(void)
{
    pid_t res; int i = 0;
    res = fork();
    if (res != 0)
        waitpid(-1, NULL, 0);
        // ou wait(NULL);
    else
        while (1) {
            printf("%d\n", i);
            i++; }
    return EXIT_SUCCESS;
}
```

Pour tuer un processus zombie, il faut tuer son père.

Exécution d'un programme : `execvp()`



Synopsis

```
#include <unistd.h>
int execvp(const char *file, char *const argv[]);
```

Description

- **remplace** l'image mémoire du processus appelant par celle de l'exécution du logiciel **file**
- **argv** : tableau des mots de la ligne de commande à exécuter (dernier mot : **NULL**).
- utilisation de **PATH** pour trouver **file**

Exécution d'un programme : `execvp()`

Exécution de `ls -l -a`

```
int main(void)
{
    char* arg[]={"ls", "-l", "-a", NULL};
    execvp(arg[0], arg);
    return EXIT_FAILURE;
}
```

Une autre exécution...

```
int main(void)
{
    char* arg[]={"./march_hare", NULL};
    pid_t res = fork();
    if (res != 0)
        wait(NULL);
    else
        execvp(arg[0], arg);
    return EXIT_SUCCESS;
}
```