



IMT Mines Albi-Carmaux
École Mines-Télécom

Systèmes d'exploitation

**Cours de licence d'informatique
INU Champollion**

**Paul Gaborit
IMT Mines Albi
Centre Génie Industriel**

2022

Première partie

Introduction

■ Paul Gaborit :

- paul.gaborit@mines-albi.fr ✉
- De formation universitaire (Nantes, Rennes puis Toulouse)
- Enseignant-chercheur au centre génie industriel de l'IMT Mines Albi



À envoyer par mail à paul.gaborit@mines-albi.fr :

- 1 Adresse de messagerie électronique.
- 2 Nom, prénom, âge.
- 3 Projet professionnel (quels diplômes visés ? quels métiers ?).
- 4 Systèmes d'exploitation utilisés.
- 5 Quels langages de programmation avez-vous...
 - ... appris/utilisés (au moins une fois) ?
 - ... pratiqués (projets ou grands programmes) ?
- 6 Si vous aimez programmer...
 - ... quels langages préférez-vous utiliser ?
 - ... pour quels types de programmes ?

- 18 heures de cours/TD (6 séances \times 3h),
- 10 heures de TP en 2 groupes (5 séances \times 2h),
- Contrôles : intermédiaire (jeudi 14 février) et final (*date à préciser*).

- Lundi 17 janvier (09:00 – 12:00) : cours 1
 - Lundi 24 janvier (09:00 – 12:00) : cours 2
 - Lundi 31 janvier (08:00 – 10:00 ou 10:15 – 12:15) : TP 1
 - Lundi 7 février (09:00 – 12:00) : cours 3
 - Lundi 14 février (08:00 – 10:00 ou 10:15 – 12:15) : TP 2
 - **Jeudi** 24 février (09:00 – 12:00) : cours 4 (et *contrôle mi-parcours*)
-

- Lundi 14 mars (09:00 – 12:00) : cours 5
 - Lundi 2 mars (09:00 – 12:00) : cours 6
 - Lundi 28 mars (08:00 – 10:00 ou 10:15 – 12:15) : TP 3
 - Lundi 4 avril (08:00 – 10:00 ou 10:15 – 12:15) : TP 4
 - Lundi 11 avril (08:00 – 10:00 ou 10:15 – 12:15) : TP 5
-

- Examens : à *définir* (première session), à *définir* (deuxième session)

Objectifs

- Comprendre comment un système d'exploitation gère plusieurs processus.
- Comprendre la notion de parallélisme, ses intérêts et ses contraintes et pratiquer la programmation parallèle à travers l'interface POSIX.

Contenu

- Le noyau du système et les primitives
- Concepts de processus et de ressources
- Les processus Unix
- Politiques d'ordonnancement des processus
- Les fichiers, les tubes, les signaux, le partage de mémoire...

Objectifs

- Comprendre comment un système d'exploitation gère plusieurs processus.
- Comprendre la notion de parallélisme, ses intérêts et ses contraintes et pratiquer la programmation parallèle à travers l'interface POSIX.

Contenu

- Le noyau du système et les primitives
- Concepts de processus et de ressources
- Les processus Unix
- Politiques d'ordonnancement des processus
- Les fichiers, les tubes, les signaux, le partage de mémoire...

Via :

- La théorie (un peu...)
- Des algorithmes (quelques-uns...)
- La pratique (commandes, appels systèmes et quelques utilitaires...)

- Tous les programmes fournis sont écrits en langage C et sont prévus pour être compilés sous Unix (testés sur Debian/Ubuntu mais ils devraient fonctionner sur n'importe quel système Linux et même Unix).
- Tous les fichiers sources des exemples et des exercices sont des pièces-jointes (📎) de ce support en PDF (utilisez un *viewer* de PDF qui sait y accéder : **adobe reader**, **sumatra**, **evince**, **atril** ou même **firefox**... mais ni **view** ni **chrome**).

Deuxième partie

Le système d'exploitation

Système d'exploitation

Système d'exploitation



**Un ensemble d'entités
communicantes... mais
lesquelles ?**

Système d'exploitation

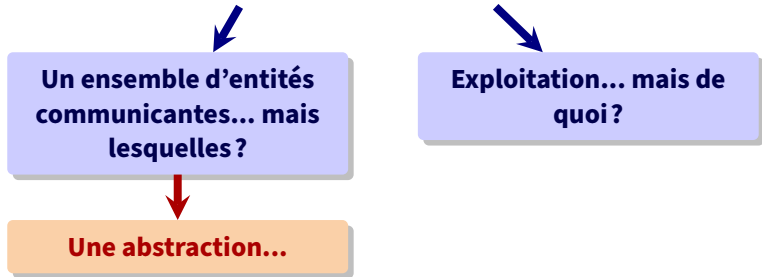


**Un ensemble d'entités
communicantes... mais
lesquelles ?**

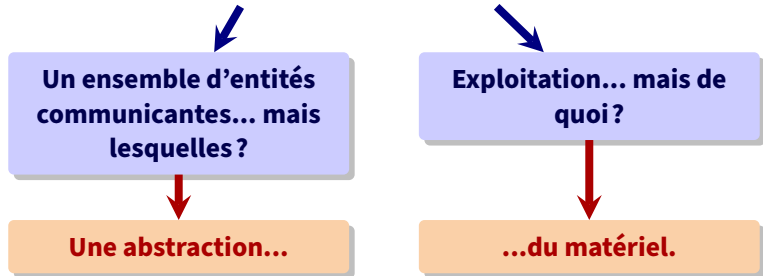


**Exploitation... mais de
quoi ?**

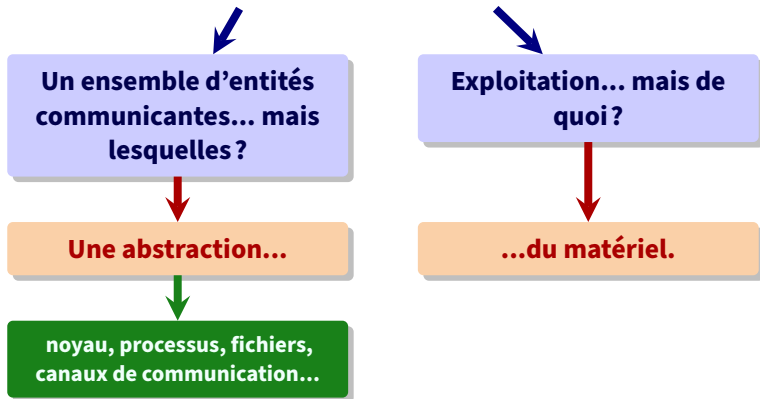
Système d'exploitation



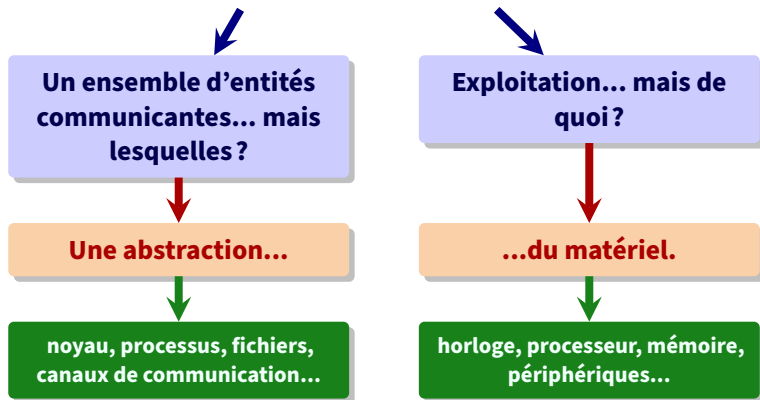
Système d'exploitation



Système d'exploitation



Système d'exploitation



Deuxième partie

Le système d'exploitation

- 1 Les fonctions du système d'exploitation
- 2 Les appels systèmes
- 3 Un exemple : Unix

Les fonctions du système d'exploitation

11a/69

- Présenter une abstraction du matériel
 - homogénéisation,
 - ne pas se préoccuper des détails,
- Assurer l'accès à ces ressources de manière cohérente
 - exclusion mutuelle,
 - non famine,
- Répartir le temps d'utilisation du processeur
- Répartir l'usage de la mémoire
 - pagination,
 - mémoire virtuelle,
 - mémoire partagée,
- Garantir la sécurité du système
 - identité,
 - droit d'accès,
- Permettre à des processus de communiquer
 - mémoire partagée,
 - synchronisation des processus,
 - canaux de communications.

Les fonctions du système d'exploitation

11b/69

- Présenter une abstraction du matériel
 - homogénéisation,
 - ne pas se préoccuper des détails,
- Assurer l'accès à ces ressources de manière cohérente
 - exclusion mutuelle,
 - non famine,
- Répartir le temps d'utilisation du processeur
 - plusieurs processus,
 - différentes priorités,
- Répartir l'usage de la mémoire
 - pagination,
 - mémoire virtuelle,
 - mémoire partagée,
- Garantir la sécurité du système
 - identité,
 - droit d'accès,
- Permettre à des processus de communiquer
 - mémoire partagée,
 - synchronisation des processus,
 - canaux de communications.

Les fonctions du système d'exploitation

11c/69

- Présenter une abstraction du matériel
 - homogénéisation,
 - ne pas se préoccuper des détails,
 - Assurer l'accès à ces ressources de manière cohérente
 - exclusion mutuelle,
 - non famine,
 - Répartir le temps d'utilisation du processeur
- Répartir l'usage de la mémoire
 - pagination,
 - mémoire virtuelle,
 - mémoire partagée,
 - Garantir la sécurité du système
 - identité,
 - droit d'accès,
 - Permettre à des processus de communiquer
 - mémoire partagée,
 - synchronisation des processus,
 - canaux de communications.

Les fonctions du système d'exploitation

11d/69

- Présenter une abstraction du matériel
 - homogénéisation,
 - ne pas se préoccuper des détails,
 - Assurer l'accès à ces ressources de manière cohérente
 - exclusion mutuelle,
 - non famine,
 - Répartir le temps d'utilisation du processeur
- Répartir l'usage de la mémoire
 - pagination,
 - mémoire virtuelle,
 - mémoire partagée,
 - Garantir la sécurité du système
 - identité,
 - droit d'accès,
 - Permettre à des processus de communiquer
 - mémoire partagée,
 - synchronisation des processus,
 - canaux de communications.

Les fonctions du système d'exploitation

11e/69

- Présenter une abstraction du matériel
 - homogénéisation,
 - ne pas se préoccuper des détails,
- Assurer l'accès à ces ressources de manière cohérente
 - exclusion mutuelle,
 - non famine,
- Répartir le temps d'utilisation du processeur
- Répartir l'usage de la mémoire
 - pagination,
 - mémoire virtuelle,
 - mémoire partagée,
- Garantir la sécurité du système
 - identité,
 - droit d'accès,
- Permettre à des processus de communiquer
 - mémoire partagée,
 - synchronisation des processus,
 - canaux de communications.

Les fonctions du système d'exploitation

11f/69

- Présenter une abstraction du matériel
 - homogénéisation,
 - ne pas se préoccuper des détails,
- Assurer l'accès à ces ressources de manière cohérente
 - exclusion mutuelle,
 - non famine,
- Répartir le temps d'utilisation du processeur
 - plusieurs processus,
 - différentes priorités,
- Répartir l'usage de la mémoire
 - pagination,
 - mémoire virtuelle,
 - mémoire partagée,
- Garantir la sécurité du système
 - identité,
 - droit d'accès,
- Permettre à des processus de communiquer
 - mémoire partagée,
 - synchronisation des processus,
 - canaux de communications.

Deuxième partie

Le système d'exploitation

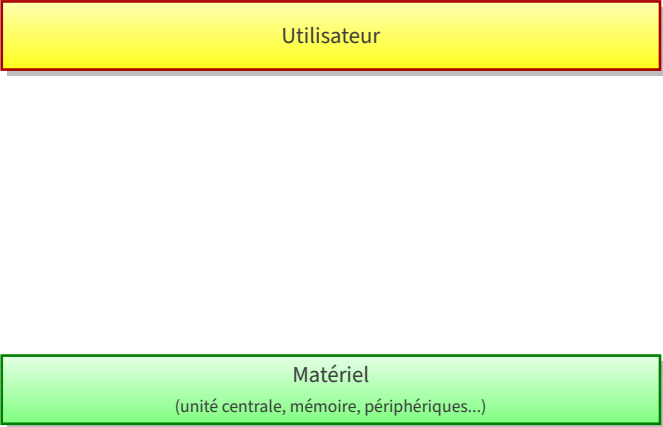
- 1 Les fonctions du système d'exploitation
- 2 Les appels systèmes
- 3 Un exemple : Unix



IMT Mines Albi-Carmaux
École Mines-Télécom

Répartition fonctionnelle

13a/69



Utilisateur

Matériel

(unité centrale, mémoire, périphériques...)

<https://makelinux.github.io/kernel/map/>

Répartition fonctionnelle

13b/69



Utilisateur

Noyau du système d'exploitation
(gestion des processus, de la mémoire, des périphériques)

Matériel
(unité centrale, mémoire, périphériques...)

<https://makelinux.github.io/kernel/map/>

Répartition fonctionnelle

13c/69

Utilisateur

Bibliothèque standard du système d'exploitation

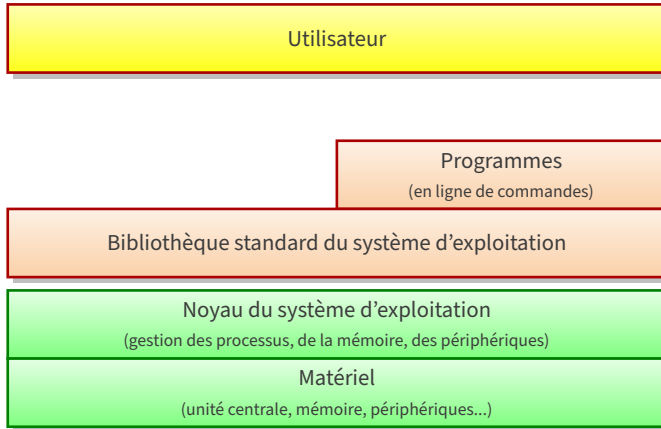
Noyau du système d'exploitation
(gestion des processus, de la mémoire, des périphériques)

Matériel
(unité centrale, mémoire, périphériques...)

<https://makelinux.github.io/kernel/map/>

Répartition fonctionnelle

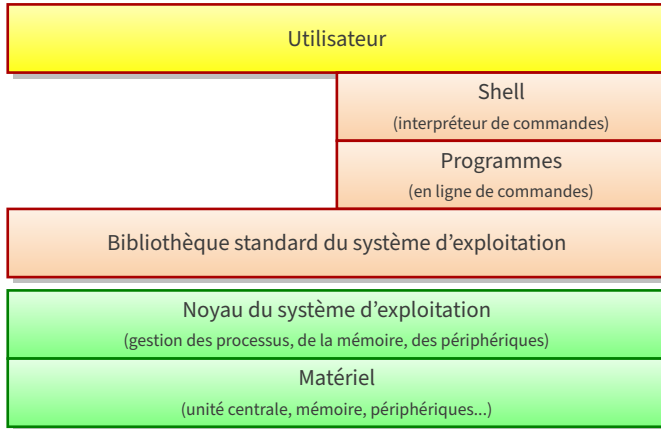
13d/69



<https://makelinux.github.io/kernel/map/>

Répartition fonctionnelle

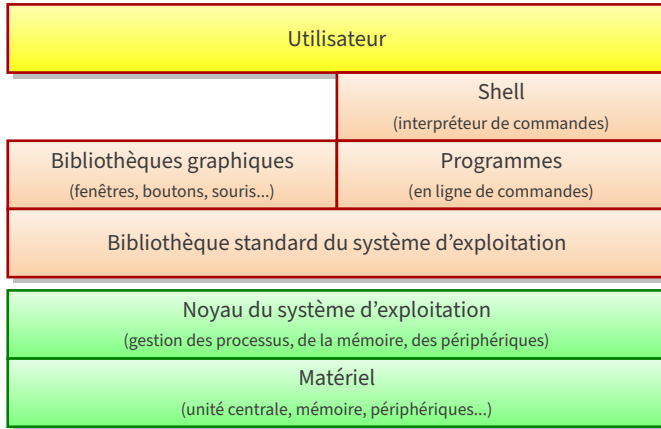
13e/69



<https://makelinux.github.io/kernel/map/>

Répartition fonctionnelle

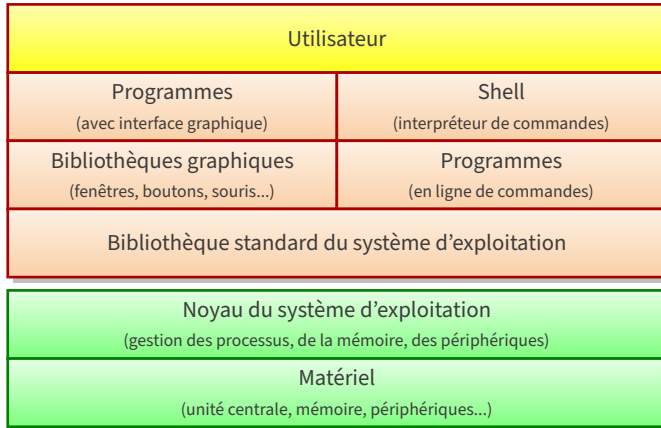
13f/69



<https://makelinux.github.io/kernel/map/>

Répartition fonctionnelle

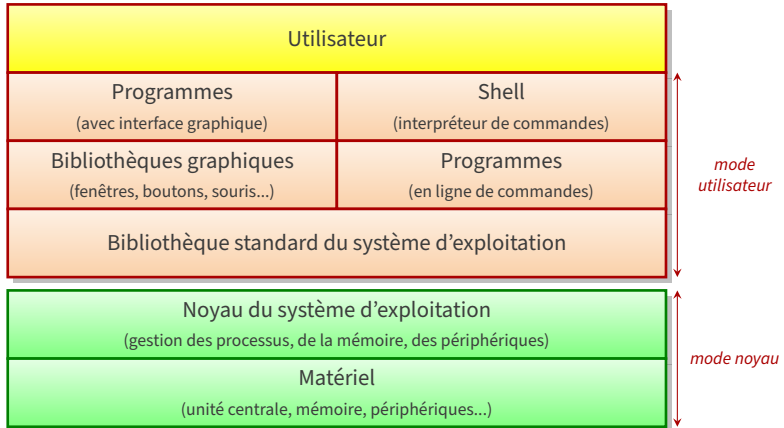
13g/69



<https://makelinux.github.io/kernel/map/>

Répartition fonctionnelle

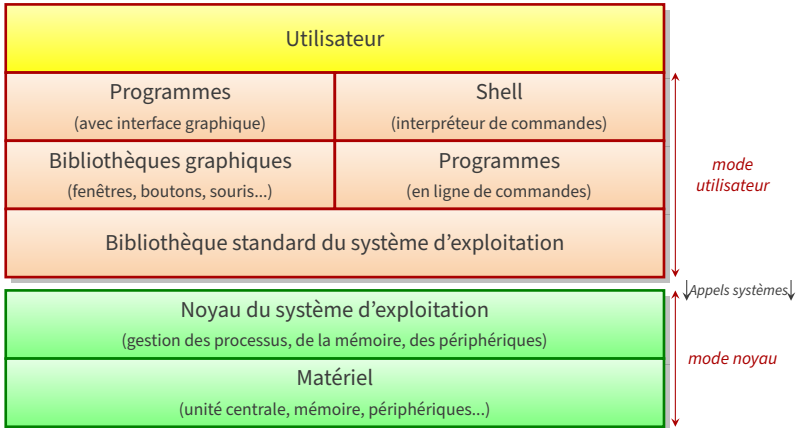
13h/69



<https://makelinux.github.io/kernel/map/>

Répartition fonctionnelle

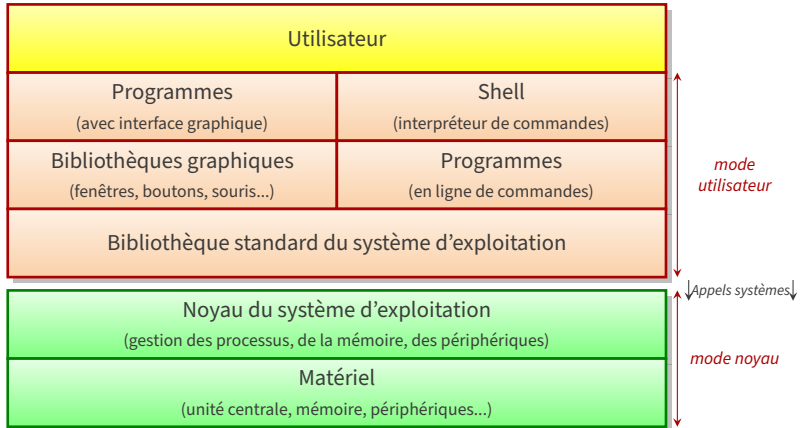
13i/69



<https://makelinux.github.io/kernel/map/>

Répartition fonctionnelle

13j/69



Attention : on triche ! L'utilisateur ne devrait pas se placer au-dessus mais en dessous puisqu'il interagit avec le matériel...

<https://makelinux.github.io/kernel/map/>

Deuxième partie

Le système d'exploitation

- 1 Les fonctions du système d'exploitation
- 2 Les appels systèmes
- 3 Un exemple : Unix

Pourquoi choisir Unix (en libre) ?

- complet,
- normalisé (POSIX),
- documenté,
- libre (accès aux sources),
- plus de 50 ans d'expérience...

Les raisons du choix d'un Unix (libre)

16/69

- L'accès à la documentation est aussi une très bonne raison !

👉 Si vous administrez votre propre machine Debian/Ubuntu, pour pouvoir lire la documentation (du système et du langage C), nous vous conseillons d'installer les packages `man-db`, `manpages-dev`, `manpages-fr-dev` et `glibc-doc` :

```
%% sudo apt install man-db manpages-dev manpages-fr-dev glibc-doc
```

Travaux Pratiques

- La commande `man`

- pour lire la page de manuel de la commande `man` :

```
%% man man
```

- pour lire la page de manuel en anglais de la commande `man` :

```
%% man -L C man
```

- Pour chercher toutes les pages de manuel via un mot clé (ici `shell`) :

```
%% man -k shell
```

- Pour chercher toutes les pages de manuel via un mot clé (ici `bash`) en se limitant à la section (1) (les commandes) :

```
%% man -s 1 -k shell
```

- La commande `xman`

Troisième partie

Les processus

Troisième partie

Les processus

4 Introduction

- Qu'est-ce qu'un processus?
- Quelques commandes pour voir les processus

5 La création de processus

6 Ordonnancement

7 Les signaux

Un processeur, plusieurs processus

19/69

Un processus

Un processus est l'exécution séquentielle d'une suite d'instructions (généralement appelée *programme*).

Un processeur

Un processeur est l'entité matérielle qui réalise effectivement les instructions. Un processeur (à un cœur) n'exécute qu'une seule instruction à la fois.

Plusieurs processus pour un seul processeur

Le processeur partage son temps entre plusieurs processus. Mais d'un point de vue utilisateur, on considère que les processus se déroulent en parallèle.

Description d'un processus

20/69

Un processus est caractérisé par :

- un numéro unique (PID = Process ID)
- un numéro de processus parent (PPID = Parent Process ID)
- un espace mémoire et un contexte d'exécution qui lui sont propres
- un utilisateur créateur identifié par son RUID (Real User ID)
- un utilisateur effectif identifié par son EUID (Effective User ID)

L'arbre des processus

- Le premier processus de PID=0 (le noyau) crée le processus `init` de PID=1.
- Tous les autres processus sont des descendants de `init`.
- Un processus crée un processus descendant direct (en fait un clone) via l'appel système `fork()`.
- Un processus peut remplacer le programme en cours d'exécution par un autre programme grâce à l'appel système `execve()`.

Commandes pratiques pour les processus

21/69

commandes

- ps
- pstree (spécifique à Linux)
- top

exercices

- afficher vos processus.
- afficher tous les processus.
- afficher uniquement l'ID du processus et de son processus parent, suivi du propriétaire, suivi du nom de la commande en cours
- afficher un arbre des processus avec **pstree** puis avec **ps**.

Troisième partie

Les processus

4 Introduction

5 La création de processus

- Usage d'Unix et de fork()
- Simulation de temps de calcul
- Changement de programme
- Mesure des temps d'exécution

6 Ordonnancement

7 Les signaux

Un premier programme (incomplet) créant un processus enfant

23/69

Fichier `fork1.c`

```
1 int main() {  
2     pid_t pid_child;  
3  
4     printf("Bonjour\n");  
5  
6     pid_child = fork();  
7     if (pid_child == 0) {  
8         printf("... je suis le processus enfant\n");  
9     } else {  
10        printf("... je suis le processus parent\n");  
11    }  
12  
13    return 0;  
14 }
```

Commande de compilation

```
%% cc -Wall -o fork1 fork1.c
```

ou :

```
%% gcc -Wall -o fork1 fork1.c
```

Les ajouts nécessaires

24a/69

Que manque-t-il?

Les ajouts nécessaires

24b/69

Que manque-t-il ?

Toutes les pages de manuel des fonctions systèmes indiquent les fichiers d'en-têtes à inclure...

Fichier `fork1.c` (avec les lignes d'en-têtes)

 `fork1.c`

```
1 #include <sys/types.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 int main() {
7     pid_t pid_child;
8
9     printf("Bonjour\n");
10
11     pid_child = fork();
12     if (pid_child == 0) {
13         printf("... je suis le processus enfant\n");
14     } else {
15         printf("... je suis le processus parent\n");
16     }
17
18     return 0;
19 }
```

Les ajouts nécessaires

25a/69

Que manque-t-il encore ?

Les ajouts nécessaires

25b/69

Que manque-t-il encore ?

Il faudrait **toujours** vérifier les valeurs retournées par les appels systèmes pour gérer d'éventuelles erreurs...

Les ajouts nécessaires

25c/69

Que manque-t-il encore ?

Il faudrait **toujours** vérifier les valeurs retournées par les appels systèmes pour gérer d'éventuelles erreurs...

Pour afficher des messages d'erreur du système :

La fonction **perror()** permet d'afficher son propre message d'erreur suivi du message d'erreur du système.

Pour interrompre un programme en cours :

La fonction **exit()** permet d'interrompre l'exécution d'un programme (sans terminer les fonctions en cours).

Un premier programme (complet) créant un processus enfant

26/69

Fichier `fork2.c`

 `fork2.c`

```
1 #include <sys/types.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 int main() {
7     pid_t pid_child;
8
9     printf("Bonjour\n");
10
11     pid_child = fork();
12     if (pid_child == -1) {perror("fork() impossible"); exit(1);}
13     if (pid_child == 0) {
14         printf("... je suis le processus enfant\n");
15     } else {
16         printf("... je suis le processus parent\n");
17     }
18
19     return 0;
20 }
```

Appel système pour créer un processus

27/69

Création d'un processus par clonage

- La fonction `fork()` duplique (clone) le processus en cours. Elle retourne -1 en cas d'erreur.
- Le processus original (le *parent*) et son clone (l'*enfant*) se distinguent par la valeur retournée par `fork()`. Cette valeur vaut :
 - 0 (zéro) du côté de l'enfant,
 - le PID du processus enfant du côté du parent.

Simulation du temps de calcul

28/69

Modifions le déroulement de ce programme...

... par l'ajout d'un appel à `sleep()`.

Fichier `fork3.c`

 `fork3.c`

```
1 #include <sys/types.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 int main() {
7     pid_t pid_child;
8
9     printf("Bonjour\n");
10
11     pid_child = fork();
12     if (pid_child == -1) {perror("fork() impossible");exit(1);}
13     if (pid_child == 0) {
14         sleep(1);
15         printf("... je suis le processus enfant\n");
16     } else {
17         printf("... je suis le processus parent\n");
18     }
19
20     return 0;
21 }
```

Attente des processus enfants et valeur de retour d'un processus

29/69

- Les processus créés par `fork()` se déroulent de manière indépendante.
- Un processus devrait attendre la fin de ses processus enfants avant de se terminer.
- Chaque processus retourne un résultat à son processus parent (son *status*).
Pour voir ce *status* avec le shell, il suffit de demander à voir la variable `$?`.
`echo $?`
- Ce résultat est la valeur retournée par `main()` ou la valeur retournée par `exit()`.

Exercices de mise en œuvre

30/69

exercices

- 1 Utiliser `wait()` pour synchroniser le parent et l'enfant dans le programme précédent. ► `fork-wait.c`
 - 2 Utiliser `rand(3)`, `srand(3)` et `getpid()` pour faire varier aléatoirement le temps d'exécution du processus parent et du processus enfant. ► `fork-rand.c`
 - 3 Afficher (en temps qu'erreur) le *status* de l'enfant et du parent à chaque appel du programme. ► `fork-status.c`
 - 4 Utiliser `nanosleep()` pour utiliser des temps d'attente moins longs. ► `fork-nano.c`
- ...suite...

Programme fork-wait.c

31/69

[retour](#) Fichier fork-wait.c

 fork-wait.c

```
1 #include <sys/types.h>
2 #include <sys/wait.h>
3 #include <unistd.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 int main() {
8     pid_t pid_child;
9     int status_child;
10    printf("Bonjour\n");
11
12    pid_child = fork();
13    if (pid_child == -1) {perror("fork() impossible");exit(1);}
14    if (pid_child == 0) {
15        sleep(1);
16        printf("... je suis le processus enfant\n");
17    } else {
18        printf("... je suis le processus parent\n");
19        pid_child = wait(&status_child);
20        if (pid_child == -1) {perror("wait() error");}
21    }
22
23    return 0;
24 }
```


Programme fork-rand.c

32/69

[retour](#)

Fichier fork-rand.c

 fork-rand.c

```
1 #include <sys/types.h>
2 #include <sys/wait.h>
3 #include <unistd.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 int main() {
8     pid_t pid_child;
9     int status_child;
10    printf("Bonjour\n");
11
12    pid_child = fork();
13    if (pid_child == -1) {perror("fork() impossible");exit(1);}
14    if (pid_child == 0) {
15        srand(getpid());
16        sleep(rand()%3);
17        printf("... je suis le processus enfant\n");
18    } else {
19        srand(getpid());
20        sleep(rand()%3);
21        printf("... je suis le processus parent\n");
22        pid_child = wait(&status_child);
23        if (pid_child == -1) {
24            perror("wait() error");
25        }
26    }
27
28    return 0;
29 }
```

Programme fork-status.c

33/69

[retour](#)

Fichier fork-status.c

 fork-status.c

```
1 #include <sys/types.h>
2 #include <sys/wait.h>
3 #include <unistd.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 int main() {
8     pid_t pid_child;
9     int status_child;
10    printf("Bonjour\n");
11
12    pid_child = fork();
13    if (pid_child == -1) {perror("fork() impossible");exit(1);}
14    if (pid_child == 0) {
15        srand(getpid());
16        sleep(rand()%3);
17        printf("... je suis le processus enfant\n");
18    } else {
19        srand(getpid());
20        sleep(rand()%3);
21        printf("... je suis le processus parent\n");
22        pid_child = wait(&status_child);
23        if (pid_child == -1) {
24            perror("wait() error");
25        } else {
26            fprintf(stderr, "status enfant: %d\n", WEXITSTATUS(status_child));
27        }
28    }
29    return rand()%256; // 'status' aléatoire
30 }
```

Programme fork-nano.c (fonction mssleep())

34/69

Fichier **fork-nano.c** (la fonction mssleep)

 fork-nano.c

```
1 void mssleep(long ms) {  
2     struct timespec delai;  
3     delai.tv_sec = ms/1000;  
4     delai.tv_nsec = (ms%1000)*1000000;  
5     int res = nanosleep(&delai, NULL);  
6     if (res != 0) {perror("nanosleep");}  
7 }
```

Programme fork-nano.c (fonction main())

35/69

◀ retour

Fichier fork-nano.c (la fonction main)

🔗 fork-nano.c

```
1 int main() {
2     pid_t pid_child;
3     int status_child;
4     printf("Bonjour\n");
5
6     pid_child = fork();
7     if (pid_child == -1) {perror("fork() impossible");exit(1);}
8     if (pid_child == 0) {
9         srand(getpid());
10        mssleep(rand()%3000);
11        printf("... je suis le processus enfant\n");
12    } else {
13        srand(getpid());
14        mssleep(rand()%3000);
15        printf("... je suis le processus parent\n");
16        pid_child = wait(&status_child);
17        if (pid_child == -1) {
18            perror("wait() error");
19        } else {
20            printf("status enfant: %d\n", WEXITSTATUS(status_child));
21        }
22    }
23
24    return rand() % 256;
25 }
```

Changer le programme exécuté par un processus

36/69

Changement de programme...

Un processus peut remplacer le programme en cours d'exécution par un autre programme lu depuis un fichier : `execve()`.

Exercices préparatoires

- 1 Écrire un programme qui appelle la commande `ls` pour afficher le contenu du répertoire courant (via `fork()` et `execve()`) puis attend qu'elle soit terminée.
[ls-via-fork-execve.c](#)
- 2 Faire un programme similaire en utilisant cette fois `vfork()` à la place de `fork()`.
[ls-via-vfork-execve.c](#)
- 3 Faire un programme similaire en utilisant `system()` au lieu de `fork()/execve()`.
[ls-via-system.c](#)

ls via fork() et execve()

37/69

[retour](#)

Fichier ls-via-fork-execve.c

 ls-via-fork-execve.c

```
1 #include <sys/types.h>
2 #include <sys/wait.h>
3 #include <unistd.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 int main() {
8     pid_t pid_child;
9     int status_child;
10
11     pid_child = fork();
12     if (pid_child == -1) {perror("fork() impossible");exit(1);}
13
14     if (pid_child == 0) { // l'enfant
15         char * argv[]={"/bin/ls", NULL};
16         char * envp[]={NULL};
17         int res = execve("/bin/ls", argv, envp);
18         if (res == -1) {perror("execve");}
19     } else { // le parent
20         pid_child = wait(&status_child);
21         if (pid_child == -1) {perror("wait() error");}
22     }
23
24     return 0;
25 }
```

ls via vfork() et execve()

38/69

[◀ retour](#)

Fichier `ls-via-vfork-execve.c`

 `ls-via-vfork-execve.c`

```
1 #include <sys/types.h>
2 #include <sys/wait.h>
3 #include <unistd.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 int main() {
8     pid_t pid_child;
9     int status_child;
10
11     pid_child = vfork();
12     if (pid_child == -1) {perror("fork() impossible");exit(1);}
13
14     if (pid_child == 0) { // l'enfant
15         char * argv[]={"/bin/ls", NULL};
16         char * envp[]={NULL};
17         int res = execve("/bin/ls", argv, envp);
18         if (res == -1) {perror("execve");}
19     } else { // le parent
20         pid_child = wait(&status_child);
21         if (pid_child == -1) {perror("wait() error");}
22     }
23
24     return 0;
25 }
```

ls via system()

39/69

[retour](#)

Fichier ls-via-system.c

 ls-via-system.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5
6     int res = system("/bin/ls");
7     if (res == -1) {perror("system()");}
8
9     return 0;
10 }
```


Mesure des temps d'exécution

40/69

Exercices de mesure des temps d'exécution

- 1 Tenter de mesurer les temps total d'exécution (via `/usr/bin/time -p`) des trois programmes de l'exercice précédent.

► comment mesurer les temps d'exécution

- 2 Ajouter une boucle dans chacun des trois programmes pour mesurer le temps d'exécution de 100 opérations.

► `ls-via-fork-execve-loop.c`

► `ls-via-vfork-execve-loop.c`

► `ls-via-system-loop.c`

- 3 Quelles différences notables de temps d'exécution entre les 3 programmes ? À votre avis, pourquoi ?

► comparaison des temps d'exécution

Mesurer les temps d'exécution

41/69

- Pour mesurer les temps d'exécution d'un **commande**, il existe plusieurs méthodes. L'une d'entre elles consiste à utiliser le programme `/usr/bin/time` avec l'option `-p` (pour obtenir une sortie conforme à la norme POSIX) :

```
%% /usr/bin/time -p commande
```

- Cette commande n'est pas toujours installée dans les distributions Linux. Le package Debian/Ubuntu qui fournit cette commande s'appelle tout simplement **time**.

Quelles mesures de temps d'exécution ?

Via la commande `/usr/bin/time -p` :

real : temps réellement attendu ;

user : temps consacré par le processeur à l'exécution des instructions du processus (en mode *utilisateur*) ;

sys : temps consacré par le processeur à l'exécution des instructions des appels systèmes du processus (en mode *noyau*) ;

La somme de **user** et **sys** peut être inférieure ou supérieure à **real**.

ls via fork() et execve() (× 100)

42/69

◀ retour

Fichier ls-via-fork-execve-loop.c

🔗 ls-via-fork-execve-loop.c

```
1 #include <sys/types.h>
2 #include <sys/wait.h>
3 #include <unistd.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 int main() {
8     pid_t pid_child;
9     int status_child;
10    int i; // compteur de boucle
11
12    for (i = 0; i < 100; i++) { // pour mesurer le temps de 100 opérations
13        pid_child = fork();
14        if (pid_child == -1) {perror("fork() impossible");exit(1);}
15
16        if (pid_child == 0) { // l'enfant
17            char * argv[]={"/bin/ls", NULL};
18            char * envp[]={NULL};
19            int res = execve("/bin/ls", argv, envp);
20            if (res == -1) {perror("execve");}
21        } else { // le parent
22            pid_child = wait(&status_child);
23            if (pid_child == -1) {perror("wait() error");}
24        }
25    }
26
27    return 0;
28 }
```

ls via vfork() et execve() (× 100)

43/69

◀ retour

Fichier ls-via-vfork-execve-loop.c

🔗 ls-via-vfork-execve-loop.c

```
1 #include <sys/types.h>
2 #include <sys/wait.h>
3 #include <unistd.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 int main() {
8     pid_t pid_child;
9     int status_child;
10    int i; // compteur de boucle
11
12    for (i = 0; i < 100; i++) { // pour mesurer le temps de 100 opérations
13        pid_child = vfork();
14        if (pid_child == -1) {perror("fork() impossible");exit(1);}
15
16        if (pid_child == 0) { // l'enfant
17            char * argv[]={"/bin/ls", NULL};
18            char * envp[]={NULL};
19            int res = execve("/bin/ls", argv, envp);
20            if (res == -1) {perror("execve");}
21        } else { // le parent
22            pid_child = wait(&status_child);
23            if (pid_child == -1) {perror("wait() error");}
24        }
25    }
26
27    return 0;
28 }
```

ls via system() (× 100)

44/69

[retour](#) Fichier ls-via-system-loop.c

 ls-via-system-loop.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5     int i; // compteur de boucle
6
7     for (i = 0; i < 100; i++) { // pour mesurer le temps de 100 opérations
8         int res = system("/bin/ls");
9         if (res == -1) {perror("system()");}
10    }
11    return 0;
12 }
```

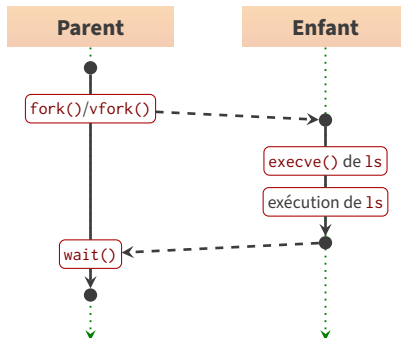
Comparaison des temps d'exécution

45/69

- On s'aperçoit que les versions via `fork()` ou `vfork()` sont toujours plus rapides que celle utilisant `system()`.
- En fait `system()` fait appel à un `shell` pour interpréter la ligne de commande (risques!). Puis ce `shell` lance un second processus pour le `ls` lui-même.
- Sur Linux et la plupart des Unix récents, les appels `vfork()` et `fork()` sont quasiment similaires en terme de performance... grâce à l'utilisation de la fonctionnalité *COW* (*Copy-on-write*) proposée par la mémoire virtuelle.

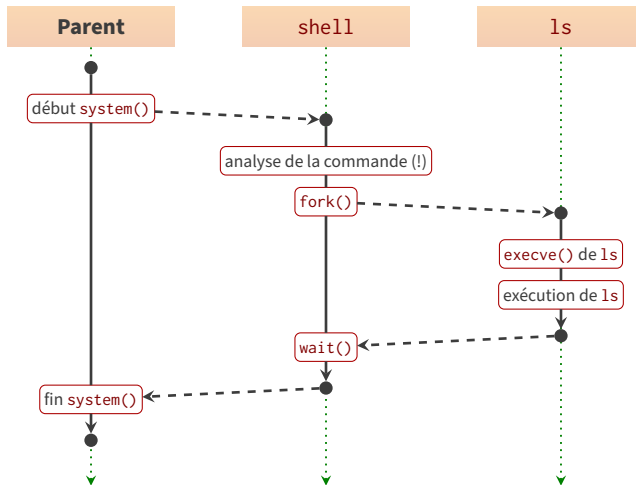
Détail du fonctionnement avec `fork()` ou `vfork()`

46/69



Détail du fonctionnement avec system()

47/69



Les processus « *zombie* »

48/69

Exercice

Reprendre le programme `ls-via-exec.c` et dans la partie du code spécifique au processus parent, ajouter un appel à `sleep()` avant le `wait()`. Exécuter ce programme et faire appel à `ps` pour voir l'état du processus enfant (qui exécute le programme `ls`).

► `ls-zombie.c`

Programme ls-zombie.c

49/69

Fichier ls-zombie.c

ls-zombie.c

```
1 #include <sys/types.h>
2 #include <sys/wait.h>
3 #include <unistd.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 int main() {
8     pid_t pid_child;
9     int status_child;
10
11     pid_child = fork();
12     if (pid_child == -1) {perror("fork() impossible");exit(1);}
13
14     if (pid_child == 0) { // l'enfant
15         char * argv[]={"/bin/ls", NULL};
16         char * envp[]={NULL};
17         int res = execve("/bin/ls", argv, envp);
18         if (res == -1) {perror("execve");}
19     } else { // le parent
20         sleep(10);
21         pid_child = wait(&status_child);
22         if (pid_child == -1) {perror("wait() error");}
23     }
24
25     return 0;
26 }
```

```
% ps -ef | grep ls
gaborit  13591 11138  0 15:27 pts/0    00:00:00 ./ls-zombie
gaborit  13592 13591  0 15:27 pts/0    00:00:00 [ls] <defunct>
gaborit  13594 12267  0 15:27 pts/1    00:00:00 grep ls
```

Processus *zombie*

On dit d'un processus qu'il est « *zombie* » ou « défunt » (*defunct*) lorsqu'il est terminé mais que son processus parent n'a toujours pas récupéré son *status*.

Autres caractéristiques standard d'un processus

51/69

arguments sous la forme de paramètres de la fonction principale du programme :

`int argc` le nombre d'arguments reçus
`char *argv[]` un tableau de chaînes de caractères (un tableau de pointeurs vers des caractères) contenant les arguments (le premier – `argv[0]` – est le nom du programme lui-même).

environnement une liste de chaînes de caractères de la forme `nom=valeur`.
Généralement un processus hérite de l'environnement de son processus parent.

- voir `env(1)`, `environ(7)` ainsi que `getenv()`, `setenv()` et `putenv()`.
- variables classiques : `PATH`, `HOME`, `USER`, `LANG`...

flux un processus débute généralement avec trois flux ouverts :

- `stdin` – un flux d'entrée,
- `stdout` – un flux de sortie,
- `stderr` – un flux de sortie d'erreur.

Troisième partie

Les processus

4 Introduction

5 La création de processus

6 Ordonnancement

- Principes
- Les interruptions
- Les états d'un processus

7 Les signaux

Ordonnancement

L'ordonnancement consiste pour le système à choisir le prochain processus à rendre actif parmi les processus prêts. Ce choix est effectué par l'ordonnanceur (en anglais *scheduler*).

Deux modes

collaboratif le système donne la main à un processus et attend que celui-ci lui rende (ex : MacOS avant MacOS X, Windows avant NT).

préemptif le système donne la main à un processus mais il l'interrompra quoiqu'il arrive au bout d'un quantum de temps fixé (tous les systèmes d'exploitation modernes).

Ordonnancement circulaire (tourniquet ou *round robin*)

- Le processus C est actif. Les autres sont dans la file d'attente.



- Lorsque le processus A a épuisé son quantum de temps, il est interrompu puis le processus F devient actif et C est placé en fin de file d'attente.



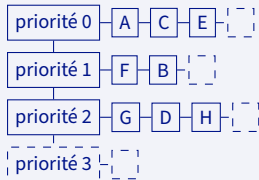
Quantum de temps

On appelle *quantum* la durée maximale durant laquelle le processeur exécute les instructions d'un processus avant qu'il soit interrompu par le système afin de commuter vers un autre processus.

Quelle durée pour le quantum de temps ?

- La commutation entre deux processus prend un temps non négligeable !
 - Quantum trop court : le processeur perd son temps à changer de processus.
 - Quantum trop long : les processus attendent plus longtemps et sont moins réactifs.
- Le choix d'un bon quantum est donc un compromis entre *performance* et *réactivité*.
- Sur les systèmes modernes (tel Linux), il varie dynamiquement (entre $\frac{1}{2}$ ms et 10 ms par exemple) en fonction du type d'activités des processus.

Ordonnancement circulaire avec priorité



- Les niveaux de priorité les plus petits sont servis en premier.
- Certains processus sont dans la file d'attente mais ne sont pas éligibles (en attente d'une ressource).

Comment attribuer les priorités ?

- Les processus du système sont, au départ, plus prioritaires (d'un niveau plus bas) que ceux de l'utilisateur.
- Le niveau de priorité d'un processus évolue avec le temps :
 - on augmente le niveau de priorité d'un processus (il devient moins prioritaire) qui consomme beaucoup de temps CPU.
 - on diminue le niveau de priorité d'un processus (il devient plus prioritaire) qui attend longtemps.

Les interruptions

54_a/69

Définition

Une interruption est une commutation de l'état du processeur déclenchée par un signal matériel (ce signal peut-être envoyé de l'extérieur ou de l'intérieur).

Les interruptions

54b/69

Un mécanisme indispensable au temps partagé

- présent sur presque tous les processeurs (vecteurs d'interruptions, les interruptions du BIOS...)
- autorise un système préemptif

Repose sur la sauvegarde de l'état du processeur

- le compteur ordinal (adresse de la prochaine instruction)
- les registres d'état du processeur (mode utilisateur/noyau, etc.)
- les registres de données
- les registres d'état du processus (droits, adresses, priorité, etc.)

Différents types d'interruptions

55/69

types d'interruptions

- externe** : intervention physique de l'utilisateur, certaines entrées/sorties...
- déroutement** : erreur interne du processeur, débordement, division par zéro, défaut de page mémoire...
- système** : les *appels systèmes* – demandes d'entrées/sorties, de changement d'état...

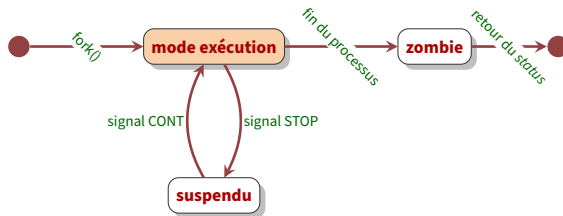
niveaux d'interruptions (exemple de priorité)

- 1 horloge interne (partage du temps entre les processus)
- 2 disques, console, carte son, carte réseau...
- 3 autres périphériques (parfois eux-mêmes classés)
- 4 appel système
- 5 autres interruptions (interruptions utilisateur)

Les états d'un processus

Point de vue utilisateur

56a/69



Légende



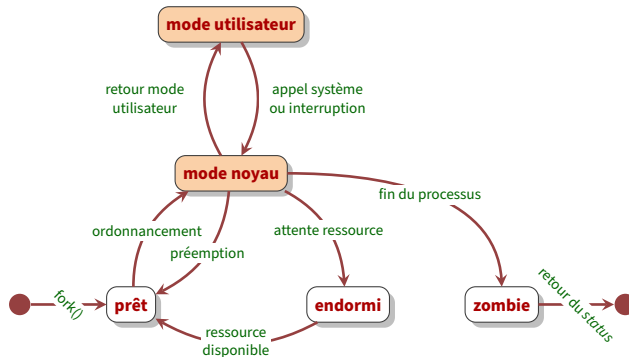
Le processus utilise le CPU

Le processus n'utilise pas le CPU

Les états d'un processus

Point de vue noyau

56b/69



Légende



Le processus utilise le CPU

Le processus n'utilise pas le CPU

Troisième partie

Les processus

4 Introduction

5 La création de processus

6 Ordonnancement

7 **Les signaux**

- Définition
- Choisir le comportement lors de la réception d'un signal
- Envoi de signaux
- Mise en œuvre

Définition

Un **signal** est comme un avertissement envoyé à un processus. À la réception d'un signal, le processus est interrompu et il effectue éventuellement le traitement de ce signal. Un *signal* est une forme d'**interruption logicielle**.

Provenance des signaux

- 1 depuis un terminal via certaines combinaisons de touches : **Ctrl-C**, **Ctrl-Z** (sous Unix), **Ctrl-** (pour les signaux **SIGINT**, **SIGTSTP**, **SIGQUIT**).
- 2 de manière interne, par le processus lui-même : **SIGSEGV** en cas d'erreur d'adressage, **SIGFPE** en cas de division par zéro...
- 3 par d'autres processus en lien avec certains événements : le processus parent envoie **SIGHUP** à ses enfants lorsque il est déconnecté de son terminal ou lorsqu'il meurt, l'enfant envoie **SIGCHLD** à son parent lorsqu'il se termine...
- 4 explicitement par un processus (via l'appel système **kill(2)**) ou par l'utilisateur (via la commande **kill**).

Les comportements par défaut

ignoré le signal est simplement ignoré.

fin le processus est arrêté (ses ressources sont libérées, les fichiers éventuellement ouverts sont fermés).

mort comme le comportement précédent mais un fichier **core** (une image de l'état du processus) est généré.

suspension le processus est suspendu (en attente de reprise).

Exemples de signaux sous Unix

signal	description	action par défaut
SIGHUP	déconnexion détectée par le parent ou mort du parent	fin
SIGINT	interruption depuis le terminal (Ctrl-C)	fin
SIGQUIT	demande « Quitter » depuis le terminal (Ctrl-\)	mort
SIGILL	instruction illégale	mort
SIGFPE	erreur mathématique virgule flottante	mort
SIGKILL	signal « KILL »	(!) fin
SIGSEGV	référence mémoire invalide	mort
SIGPIPE	écriture dans un tube sans lecteur	fin
SIGALRM	temporisation alarm(2) écoulée	fin
SIGTERM	signal de fin	fin
SIGUSR1	signal utilisateur 1	fin
SIGUSR2	signal utilisateur 2	fin
SIGCHLD	processus enfant arrêté ou terminé	ignoré
SIGCONT	continuer si suspendu	ignoré
SIGSTOP	suspension du processus	(!) suspension
SIGTSTP	suspension du processus depuis le terminal (Ctrl-Z)	suspension

Note : les actions marquées d'un (!) ne sont pas modifiables...

Gestion des signaux

- Chaque processus choisit sa politique de gestion des différents signaux.
- Un signal d'un type donné peut être :
 - capté** : une fonction de traitement est associée à la réception de ce signal.
 - ignoré** : le signal est purement et simplement ignoré.
 - par défaut** : on laisse le système adopter le comportement par défaut (celui défini à la création du processus).
- Certains signaux ne peuvent être ni ignorés ni captés :
 - **SIGKILL** : tue le processus.
 - **SIGSTOP** : suspend le processus.

Choisir le mode de traitement d'un signal

- Le fichier d'en-tête :

```
#include <signal.h>
```

- Définition du type pointeur vers une fonction de traitement de signaux :

```
typedef void (*sighandler_t)(int);
```

- La fonction `signal(2)` permet d'attacher ou de détacher une fonction de traitement (un `handler`) associée à un type de signal (de numéro `signum`) :

```
sighandler_t signal(int signum, sighandler_t handler);
```

- Retourne soit un pointeur vers la fonction de gestion du signal précédemment attachée ou `SIG_ERR` en cas d'erreur.
- On peut passer deux valeurs particulières comme `handler` :

`SIG_IGN` pour ignorer le signal.

`SIG_DFL` pour adopter le comportement par défaut.

- La fonction `pause(2)` permet d'attendre un signal.

Note importante

L'appel système `signal(2)` ne sert pas à envoyer des signaux !

Exemple d'interception de signaux

60/69

Fichier `signaux.c`

 `signaux.c`

```
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <unistd.h>
4 #include <signal.h>
5
6 void traite_signal(int signum) {
7     printf("Reception du signal %d\n", signum);
8 }
9
10 int main () {
11     int i;
12     int signaux[] = {SIGINT, SIGSTOP, SIGALRM, SIGTSTP};
13     for (i=0; i < sizeof(signaux)/sizeof(int); i++) {
14         if (signal(signaux[i], &traite_signal) == SIG_ERR) {
15             printf("%2d: signal %2d non captable !\n", i, signaux[i]);
16         } else {
17             printf("%2d: signal %2d ok.\n", i, signaux[i]);
18         }
19     }
20
21     printf("PID: %d\n", getpid());
22     while (1) {
23         if (pause() == -1) {perror("fin de pause");}
24     }
25 }
```

Envoi de signaux

61/69

L'envoi de signaux

- L'utilisateur peut envoyer un signal au processus de PID `pid` via la commande `kill(1)` :

```
% kill -SIGNAL pid
```

(où **SIGNAL** est le nom d'un signal sans le préfixe **SIG**. Ex : **QUIT**)
- Un processus peut envoyer un signal au processus de PID `pid` via l'appel système `kill(2)` :

```
int kill(pid_t pid, int sig);
```

(où **sig** est l'une des constantes associées aux numéros des signaux. Ex : **SIGINT**).
- Le système vérifie que l'utilisateur ou le processus émetteur a bien le droit d'envoyer un signal au processus destinataire.

Ne confondez pas...

La commande `kill(1)` (décrite en section 1 du manuel et utilisable en ligne de commande) et la fonction et appel système `kill(2)` (décrite en section 2 du manuel et utilisable depuis un programme).

Envoi de signaux

62/69

Exercice

- 1 Reprenez le programme précédent et compilez-le après avoir ajouter le signal `SIGQUIT` dans la liste.
- 2 Vérifiez que vous ne pouvez plus arrêter ce processus depuis votre terminal. Pourquoi?
- 3 Ouvrez un autre terminal et tuez votre processus en utilisant la commande `kill`. Quels signaux pouvez-vous envoyer?
- 4 Essayer de rendre votre processus non « *tuable* » ? Comment avez-vous fait ou sinon pourquoi n'est-ce pas faisable?

Énoncé

- 1 Écrire un programme qui crée deux processus (via un appel à `fork(2)`) : le premier affiche tous les entiers **pairs** compris entre 1 et 20 et le second affiche tous les entiers **impairs** compris entre 1 et 20. L'exécuter plusieurs fois. Que constate-t-on ?
- 2 Utiliser la fonction `msleep()` précédemment écrite pour ajouter un peu d'attente aléatoire dans les deux processus. L'exécuter plusieurs fois. Que constate-t-on ?
- 3 Utiliser les signaux pour synchroniser les deux processus (parent et enfant) afin d'afficher tous les nombres dans l'ordre.
- 4 Vérifier qu'il n'y a aucun risque d'interblocage, sinon proposer une solution.
- 5 Supprimer les appels à `msleep()` et constater que tout semble bien marcher...

Problème – Q.1

64/69

Fichier `de1a20.c`

 `de1a20.c`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <unistd.h>
5
6 int main(void) {
7     pid_t pid_enfant;
8     int i;
9
10    if ((pid_enfant = fork()) == -1) {perror("fork"); exit(1);}
11
12    if (pid_enfant == 0) { // l'enfant
13        for (i = 1; i <= 20; i += 2) {
14            printf("%d\n", i);
15        }
16    } else { // le parent
17        for (i = 2; i <= 20; i += 2) {
18            printf("%d\n", i);
19        }
20    }
21
22    return 0;
23 }
```

Problème – Q.2

65/69

Fichier `de1a20-mssleep.c` (sans les en-têtes ni la fonction `mssleep`)


 `de1a20-mssleep.c`

```
1 int main(void) {
2     pid_t pid_enfant, pid_child;
3     int i;
4
5     if ((pid_enfant = fork()) == -1) {perror("fork"); exit(1);}
6     if (pid_enfant == 0) { // l'enfant
7         srand(getpid());
8         for (i = 1; i <= 20; i += 2) {
9             mssleep(rand()%100); printf("%d\n", i);
10        }
11    } else { // le parent
12        srand(getpid());
13        for (i = 2; i <= 20; i += 2) {
14            mssleep(rand()%100); printf("%d\n", i);
15        }
16        pid_child = wait(NULL);
17        if (pid_child == -1) {perror("wait() error");}
18    }
19
20    return 0;
21 }
```

Problème – Q.3

66/69

Fichier `de1a20-synchro.c` (sans les en-tête ni la fonction `mssleep`)

 `de1a20-synchro.c`

```
1 void handler(int sig) {}
2
3 int main(void) {
4     pid_t pid_enfant;
5     int i;
6     if (signal(SIGUSR1, handler) == SIG_ERR) {perror("signal"); exit(1);}
7     if ((pid_enfant = fork()) == -1) {perror("fork"); exit(1);}
8     if (pid_enfant == 0) { // l'enfant
9         srand(getpid());
10        pid_t pid_parent = getppid();
11        for (i = 1; i <= 20; i += 2) {
12            mssleep(rand()%100); printf("%d\n", i);
13            kill(pid_parent, SIGUSR1);
14            pause();
15        }
16    } else { // le parent
17        srand(getpid());
18        for (i = 2; i <= 20; i += 2) {
19            pause();
20            mssleep(rand()%100); printf("%d\n", i);
21            kill(pid_enfant, SIGUSR1);
22        }
23    }
24    return 0;
25 }
```

Problème – Q.4

67/69

Fichier `de1a20-synchro-ok.c` (sans les en-têtes ni la fonction `mssleep`)

 `de1a20-synchro-ok.c`

```
1 int flag = 1;
2 void handler(int sig) {flag = 0;}
3
4 int main(void) {
5     pid_t pid_enfant;
6     int i;
7     if (signal(SIGUSR1, handler) == SIG_ERR) {perror("signal"); exit(1);}
8     if ((pid_enfant = fork()) == -1) {perror("fork"); exit(1);}
9     if (pid_enfant == 0) {// l'enfant
10         srand(getpid());
11         pid_t pid_parent = getppid();
12         for (i = 1; i <= 20; i += 2) {
13             mssleep(rand()%100); printf("%d\n", i);
14             kill(pid_parent, SIGUSR1);
15             if (flag) {pause();} flag = 1;
16         }
17     } else {// le parent
18         srand(getpid());
19         for (i = 2; i <= 20; i += 2) {
20             if (flag) {pause();} flag = 1;
21             mssleep(rand()%100); printf("%d\n", i);
22             kill(pid_enfant, SIGUSR1);
23         }
24     }
25     return 0;
26 }
```

Problème – Q.5

68/69

Fichier `de1a20-synchro-ok-nosleep.c` (sans les en-têtes)

 `de1a20-synchro-ok-nosleep.c`

```
1 int flag = 1;
2 void handler(int sig) {flag = 0;}
3
4 int main(void) {
5     pid_t pid_enfant;
6     int i;
7     if (signal(SIGUSR1, handler) == SIG_ERR) {perror("signal"); exit(1);}
8     if ((pid_enfant = fork()) == -1) {perror("fork"); exit(1);}
9     if (pid_enfant == 0) { // l'enfant
10         pid_t pid_parent = getppid();
11         for (i = 1; i <= 20; i += 2) {
12             printf("%d\n", i);
13             kill(pid_parent, SIGUSR1);
14             if (flag) {pause();} flag = 1;
15         }
16     } else { // le parent
17         for (i = 2; i <= 20; i += 2) {
18             if (flag) {pause();} flag = 1;
19             printf("%d\n", i);
20             kill(pid_enfant, SIGUSR1);
21         }
22     }
23     return 0;
24 }
```

Il reste un risque...

69/69

- Dans les deux versions précédentes, malgré l'utilisation de la variable globale **flag** pour mémoriser la réception d'un signal, il persiste un risque d'interblocage. Pour supprimer ce risque, il faudrait que le test du **flag** suivi de la mise en **pause()** éventuelle soit une opération atomique (une sorte de *test-and-set*).
- Les signaux seuls ne constituent donc pas un moyen fiable de synchroniser plusieurs processus. Ils restent tout de même utiles pour d'autres usages (ex : demande de relecture d'un fichier de configuration...).