Introduction aux systèmes d'exploitation

Y. Guesnet

Département d'informatique
Université de Rouen

4 septembre 2023

Plan

1. Introduction
2. Processus
3. Les threads
4. Les fichiers
5. Communication et synchronisation
6. Les signaux

Plan

1. Introduction
2. Processus
3. Les threads
4. Les fichiers
5. Communication et synchronisation
6. Les signaux

Plan

1. Introduction
   - Présentation du cours

Le cours de systèmes d'exploitation

Objectifs

Ce cours a deux objectifs :

- Vous présenter les concepts présidant à l'élaboration d'un système d'exploitation.
  - Nous aborderons les fondements théoriques des différents outils proposés par les systèmes d'exploitation.
  - Nous verrons comment ces notions sont mises en œuvre à l'intérieur du système GNU/Linux.
- Vous initier à la programmation système.
  - Nous étudierons l'API proposée par le standard POSIX.
  - Nous utiliserons cette API via le langage C.

Les enseignements

Organisation

- 18 h de cours
- 16 h de TD
- 16 h de TP

Les enseignements

Évaluations

- Contrôle continu intégral.
- Deux contrôles : un écrit après les TD 4 ou 5 et un autre à la fin des TD.
- Un projet en fin de semestre.
- La note finale est obtenue en calculant la moyenne des trois notes.
- Pour être précis, la formule exacte est :
  ((CC1 + CC2) / 2) * 0.67 + TP * 0.33
- Un écrit de « seconde chance » en fin de semestre. La seconde chance peut remplacer une ou deux notes de contrôle (pas de projet).

Bibliographie

[1] Christophe Blaess. *Développement système sous Linux*. Eyrolles, Paris, 2011. ISBN : 978-2-2121-2881-9.

[2] Daniel P. Bovet et Marco Cesati. *Understanding the Linux Kernel*. O'Reilly Media, 2000. ISBN : 978-0-596-00002-8.

[3] Robert Love. *Linux Kernel Development*. Addison Wesley, 2010. ISBN : 978-0-672-32946-3.

[4] Andrew Tanenbaum. *Systèmes d'exploitation, 3e édition*. Pearson Education France, Paris, 2008. ISBN : 978-2-7440-7299-4.

[5] *The Open Group Base Specifications Issue 7, 2018 edition*. The Open Group. 2018. URL : https://pubs.opengroup.org/onlinepubs/9699919799/ (visité le 06/09/2019).

Plan

- Les systèmes d'exploitation
- Normes UNIX
- Les appels système

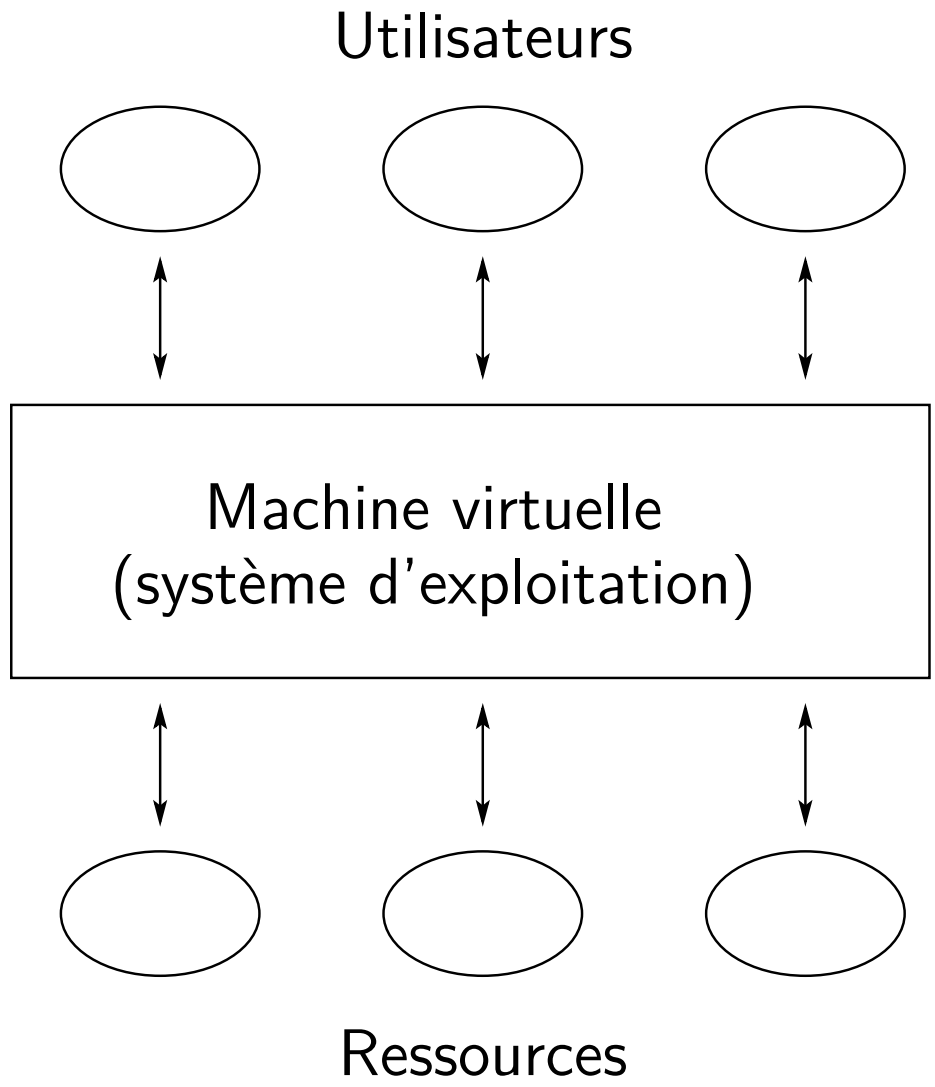
Les systèmes d'exploitation

Définition

Un système d'exploitation est une couche logicielle dont le rôle est de :

1. Gérer les périphériques
2. Fournir aux programmes une interface simplifiée avec le matériel

Les systèmes d'exploitation



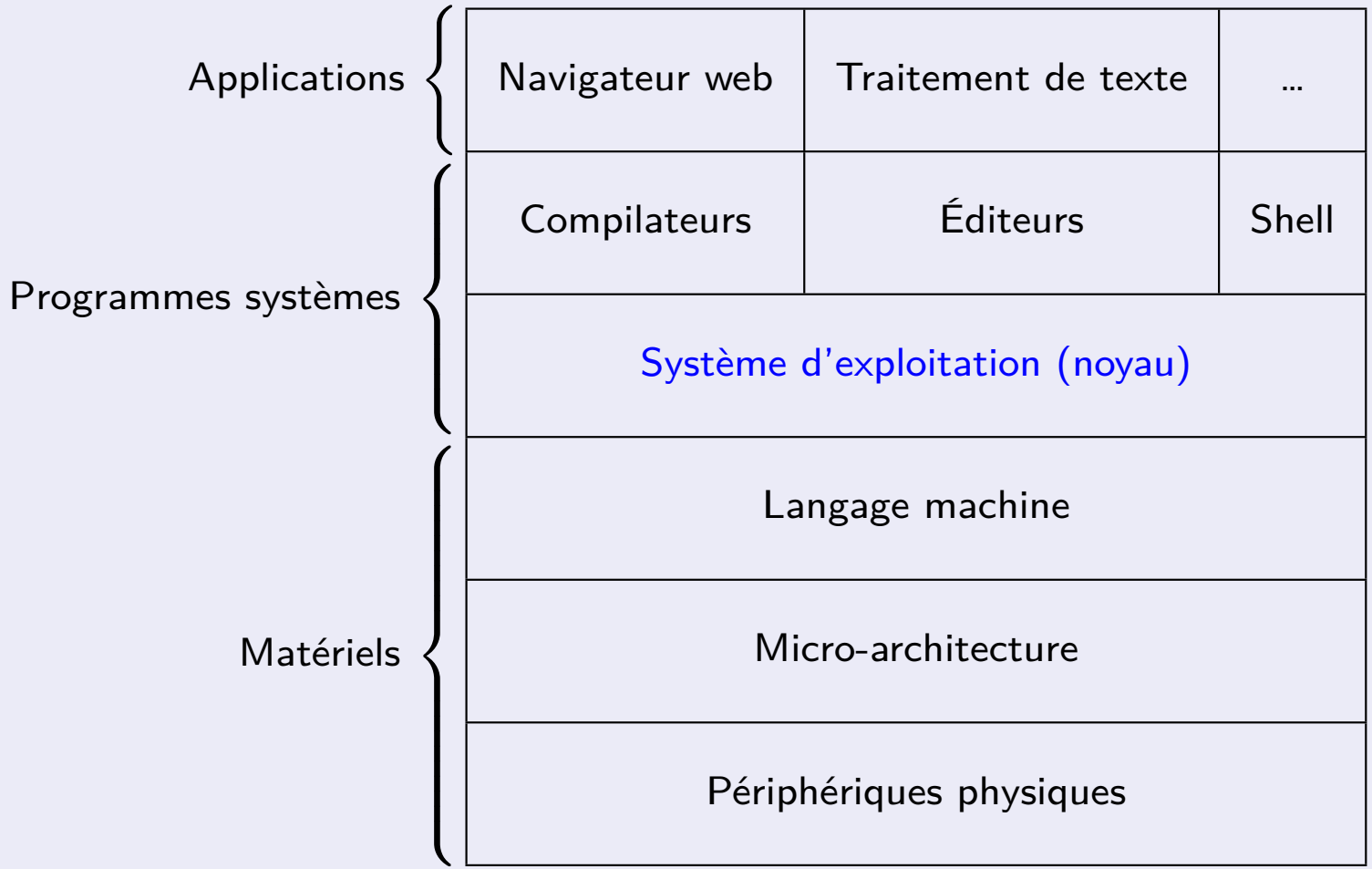
Les systèmes d'exploitation

Remarque

- Il est difficile de donner une définition précise d'un système d'exploitation.
- Selon les différents fournisseurs de systèmes, cette notion est plus ou moins large.
- Nous définissons ici les systèmes d'exploitation dans leur acceptation la plus stricte.

Les systèmes d'exploitation

Un système informatique



Où placer le gestionnaire de fenêtres ?

Les systèmes d'exploitation

Le rôle d'un système d'exploitation

Un système d'exploitation offre un ensemble de services à l'utilisateur. Parmi ceux-ci, on peut citer :

- Utilisation des différents périphériques au travers d'une interface unifiée (carte graphique, disques durs, ...)
- Accès aux fichiers (gestion des différents formats, sauvegarde, vérification d'intégrité, ...)
- Gestion des ressources (gestion de la mémoire, partage des périphériques, ...)
- Contrôle de l'utilisation des ressources (sécurité, statistiques, quotas, ...)

Les systèmes d'exploitation

Le marché : deux solutions ?

On distingue aujourd'hui deux grandes familles de systèmes d'exploitation

- Les systèmes UNIX (System V, BSD, Linux, Mac OS X, Oracle Solaris, ...)
- Les systèmes Windows (Windows 10, Windows 10 mobile, ...)

Remarque

- Il existe bien sûr d'autres systèmes d'exploitation n'appartenant pas à ces deux familles, comme par exemple VxWorks.
- Bien qu'utilisant le noyau Linux, on ne peut pas dire qu'Android fait partie de la famille UNIX.

Les systèmes d'exploitation

Différents types de systèmes d'exploitation

- Les systèmes pour mainframes (OS/390)
- Les systèmes d'exploitation « serveurs »
- Les systèmes multiprocesseurs
- Les systèmes personnels
- Les systèmes temps réel (VxWorks, QNX)
- Les systèmes embarqués (Windows CE, Android)
- Les systèmes pour smart cards
- ...

Plan

- Les systèmes d'exploitation
- Normes UNIX
- Les appels système

Les normes pour UNIX

Le standard UNIX

- POSIX est une famille de standards définie par l'IEEE. Il s'agit d'une standardisation des interfaces utilisateurs et logicielles des systèmes UNIX.
- SUS (Single UNIX Specification) est un ensemble de spécifications éditées par l'*Austin Group* permettant de certifier un système d'exploitation comme étant un UNIX.
- Désormais POSIX est édité conjointement par l'IEEE et *The Open Group*.

Les normes POSIX

Les différentes versions POSIX

- POSIX.1 definit les services du noyau.
- POSIX.1b ajoute des extensions temps réel.
- POSIX.1c ajoute des extensions pour les threads.
- POSIX.2 définit l'interpréteur de commandes et les programmes utilitaires.
- Il y a différentes versions de POSIX (POSIX, POSIX:2001, POSIX:2004, POSIX:2008, ...).

Utilisation de la norme POSIX dans les programmes

gcc et les normes UNIX

- On peut spécifier à gcc de respecter la norme POSIX.1 en définissant la macro *_POSIX_SOURCE*.
- La macro *_POSIX_C_SOURCE* permet de spécifier quelles fonctionnalités POSIX on désire (1 pour POSIX.1, 2 pour POSIX.2, 199309L pour POSIX.1b, 200112L pour POSIX.1:2001, ...)
- De même, on peut demander à gcc de respecter la norme SUS en donnant une valeur à la macro *_XOPEN_SOURCE* (500 pour SUSv2, 600 pour SUSv3, 700 pour SUSv4).

Comment feriez-vous concrètement pour définir une macro indiquant une norme ?

Plan

- Les systèmes d'exploitation
- Normes UNIX
- Les appels système

Les appels système

Définition

- Seul le noyau a un accès direct à toutes les ressources.
- Les appels système constituent l'interface entre le système d'exploitation et les programmes utilisateur.
- Les appels système sont fortement dépendants de la machine, ils sont la plupart du temps écrits en assembleur.
- Des bibliothèques sont fournies afin de pouvoir effectuer les appels système depuis les programmes C (il existe aussi des bibliothèques pour d'autres langages).

Fonctionnement d'un appel système

Exemple

Pour expliquer ce qui se passe lorsqu'on veut exécuter un appel système, prenons l'exemple d'un programme qui souhaite lire dans un fichier :

```
n = read(fd, buf, count);
```

Remarque

- Vous noterez qu'on emploie ici la fonction `read` qui est l'interface à l'appel système `read`
- Les fonctions réalisant les appels système du même nom sont parfois appelées fonctions « enveloppes »
- Nous n'utiliserons pas les fonctions de plus haut niveau (comme `fread`) fournies par la bibliothèque standard.

Fonctionnement d'un appel système

L'appel système *read*

La fonction *read* prend trois paramètres :

- Un descripteur de fichier *fd* associé au fichier dans lequel on veut lire.
- Une adresse *buf* correspondant à l'emplacement du tampon dans lequel on veut stocker les données lues.
- Le nombre d'octets *count* qu'on souhaite lire.

La fonction retourne le nombre d'octets lus ou -1 en cas d'erreur.

Fonctionnement d'un appel système

Fonctionnement du programme

- Comme pour toute fonction, lorsque le programme souhaite appeler la fonction `read`, il commence par empiler les paramètres de la fonction (ou les stocke dans des registres selon l'ABI utilisateur).
- Par exemple, avec l'ABI 32 bits, les paramètres sont empilés de la droite vers la gauche, ainsi le compilateur va empiler `count`, puis `buf` et enfin `fd`.
- Le programme passe ensuite à l'exécution de la fonction `read`.

Fonctionnement d'un appel système

La fonction *read*

La fonction *read* devra exécuter l'appel système *read*.

- Pour cela elle doit stocker le numéro de l'appel système (0 pour *read* sous x86-64) dans un registre particulier (*%rax* pour l'ABI 64 bits noyau).
- Elle stocke également les arguments de l'appel système (dans des registres dédiés sous Linux, jusqu'à six maximum)
- Elle peut ensuite exécuter l'instruction (assembleur) *syscall* (ou autre, comme lever l'exception `int 0x80` en 32 bits)
- L'instruction *syscall* permet de passer en mode privilégié (on parle aussi de mode noyau). Ce mode nous autorise, par exemple, à accéder directement aux différentes ressources du système.

Fonctionnement d'un appel système

La fonction *read*

- L'instruction *syscall* démarre ensuite l'exécution d'un code bien particulier du noyau.
- Ce code regarde le numéro de l'appel système transmis et saute à l'adresse correspondant au code de l'appel système.
- Une fois l'appel système terminé, on revient au mode utilisateur et on passe à l'instruction qui suit le *syscall*.
- La fonction *read* regarde la valeur du retour de l'appel système qui se trouve dans un registre
- Elle remplit éventuellement *errno* (valeur de retour négative)
- La fonction *read* peut enfin se terminer normalement en rendant la main au programme appelant.
- Ce dernier nettoie la pile en incrémentant le pointeur de pile.

Fonctionnement d'un appel système

La commutation de contexte

- Lors du passage en mode noyau, le système effectue une commutation de contexte.
- Le processeur doit alors sauvegarder l'état courant du processus.
- Le nouvel état de processus doit être chargé.
- La projection mémoire doit être modifiée.
- La totalité du cache mémoire est généralement invalidée.
- Cette commutation de contexte rend l'exécution d'un appel système très lent (on parle de 1000 à 10000 fois plus lent qu'un appel à une fonction standard, programme de test).

Fonctionnement d'un appel système

Prenons de bonnes habitudes

- Presque tous les appels système retournent une valeur. Celle-ci nous permet de vérifier que l'exécution de l'appel système s'est correctement déroulée. Nous devrons donc **toujours** vérifier la valeur de retour de l'appel système.
- Très souvent, lorsqu'une erreur se produit, la valeur -1 est retournée. On peut alors récupérer un code correspondant au type de l'erreur qui s'est produite via la variable globale *errno*.
- La fonction `void perror(const char *s);` permet d'afficher un message d'erreur correspondant au code d'erreur contenu dans *errno*.

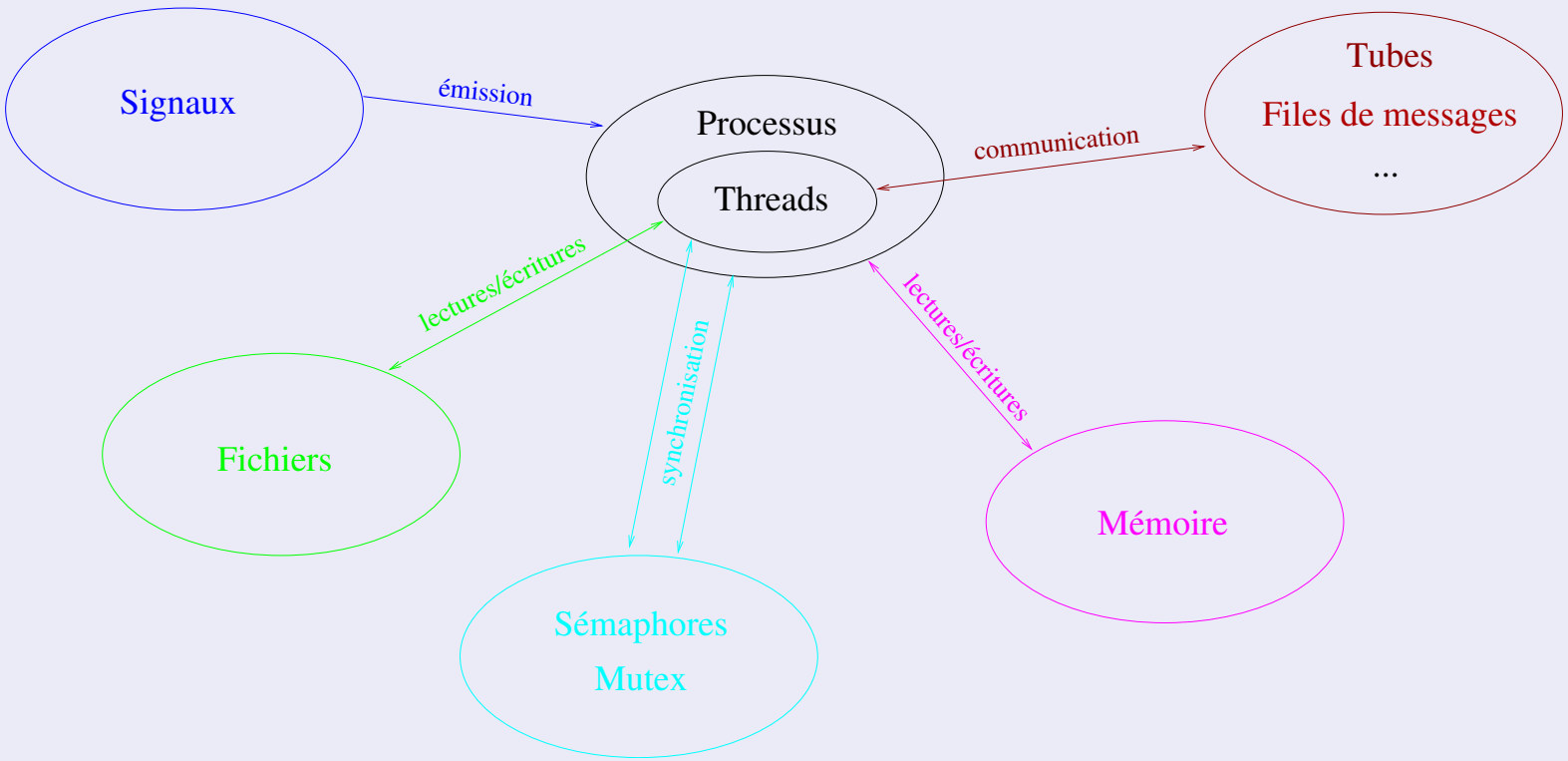
Fonctionnement d'un appel système

L'appel système *read*

```
#include <unistd.h> // read
#include <stdlib.h> // exit, malloc
#include <stdio.h>  // perror

...

int n, fd;
char* buf;
size_t count;

...

buf = malloc(count);
n = read(fd, buf, count);
if (n == -1) {
    perror("Erreur lors du read");
    exit(EXIT_FAILURE);
}

...
```

Ce que vous allez voir...

Les notions associées aux systèmes d'exploitation



Plan

1. Introduction
2. Processus
3. Les threads
4. Les fichiers
5. Communication et synchronisation
6. Les signaux

Plan

2 Processus
- Introduction
- Les appels système POSIX
- Ordonnancement

Définition

Processus

- Le processus est au cœur du fonctionnement de tout système d'exploitation.
- Un processus est une abstraction d'un programme en cours d'exécution.

Les processus

Le modèle de processus

- Les logiciels et le système d'exploitation sont constitués de processus.
- Un processus est un programme qui s'exécute avec un compteur ordinal, des registres (pointeur de pile, registres matériels, ...) et des variables.
- Le(s) processeur(s) bascule(nt) sans arrêt d'un processus à l'autre.
- Chaque processus dispose d'un espace d'adressage virtuel : un ensemble d'adresses mémoire allant de 0 à une limite donnée.
- L'espace d'adressage contient le programme exécutable, les données et la pile.

Les processus

La table des processus

- Les informations concernant les processus sont généralement contenues dans une table : la table des processus.
- Une entrée de la table des processus est un bloc de contrôle de processus (BCP).
- Lorsqu'un processeur bascule d'un processus à l'autre, il sauvegarde l'état du processus dans le BCP.
- Le BCP contient donc toute l'information nécessaire au redémarrage du processus.
- Un processus suspendu est donc constitué de son BCP et du contenu de son espace d'adressage (image *core*).

Les processus

La table des processus

Voici quelques champs qu'on peut généralement trouver dans le BCP :

- Registres, compteur ordinal, pointeur de pile (contexte d'unité centrale du processus courant ou **mot d'état**).
- État du processus, priorité, paramètres d'ordonnancement.
- Identifiant du processus, processus parent, groupe de processus.
- Signaux.
- Adresses des segments de texte, de données et de pile.
- Répertoire racine, de travail, descripteurs de fichiers.
- Utilisateur, groupe.
- Heure de début, temps de traitement, alarme.

Les processus

Création d'un processus

Il existe principalement quatre évènements pouvant mener à la création d'un processus :

- Initialisation du système.
- Création d'un processus par un autre processus.
- Requête utilisateur.
- Initiation d'un travail en traitement par lots (*mainframes*).

Fin d'un processus

Un processus peut se terminer pour diverses raisons :

- Arrêt normal (volontaire).
- Arrêt suite à une erreur (volontaire).
- Arrêt pour erreur fatale (involontaire).
- Arrêt par un autre processus (involontaire).

Les processus

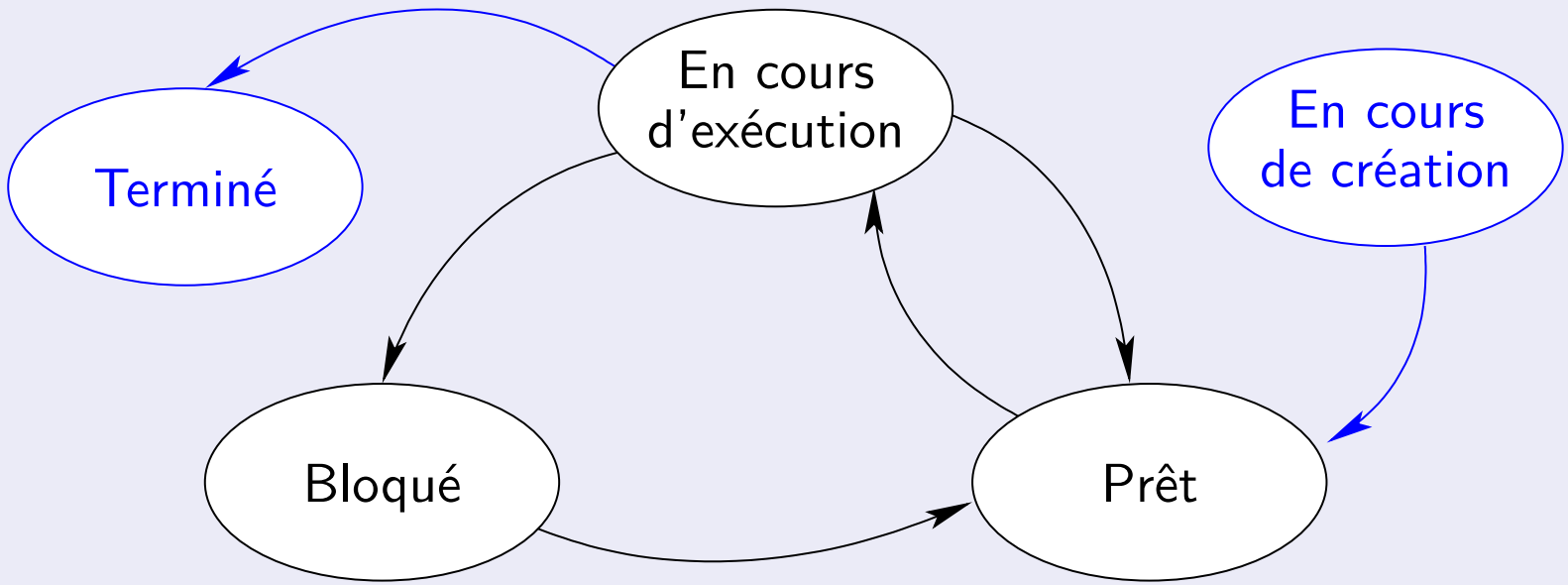
Hiérarchie de processus

- Sous UNIX, lorsqu'un processus crée un processus, les processus parent et enfant continuent d'être associés.
- Un processus n'a qu'un seul parent mais peut avoir 0, 1 ou plusieurs enfants.
- Un processus ainsi que l'ensemble de ses descendants forment un groupe de processus.
- Sous Windows, tous les processus sont égaux. On récupère juste un *handle* lorsqu'on crée un processus mais ce *handle* peut être transféré à un autre processus.

Les processus

Les états des processus

De manière générale, les processus peuvent passer par différents états :



Le cas UNIX

- Les systèmes UNIX distinguent deux types d'état bloqué selon l'évènement ayant amené cet état : un évènement interruptible par un signal (comme avec `sleep()`) ou un évènement ininterruptible (généralement une entrée/sortie).
- Il existe d'autres états sous Linux (comme « stoppé »).
- Il n'y a pas d'état « en cours de création », mais il y a un état « terminé ».
- Losqu'un processus est dans l'état « terminé », il peut attendre que le processus qui l'a créé soit mis au courant de sa « mort ». Nous dirons alors qu'il s'agit d'un processus **zombie**.

Plan

2. Processus
   - Introduction
   - Les appels système POSIX
   - Ordonnancement

Identifiants

Les identifiants de processus

- Chaque processus est repéré par un identifiant. Ce dernier peut être retrouvé à l'aide de l'appel système *getpid*.
- De plus, chaque processus peut connaître le processus qui l'a créé (son **père**) à l'aide de l'appel système *getppid*.
- Lorsqu'un processus se termine, tous ses fils sont rattachés au processus d'identifiant 1 (*init* ou *systemd*).

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

pid_t getpid(void);
pid_t getppid(void);
```

Ces appels réussissent toujours.

Voir l'exemple

Groupes et sessions

Les groupes de processus

- Chaque processus appartient à un groupe de processus. Le groupe de processus peut être connu *via* la fonction `getpgrp`.
- On peut changer le groupe d'un processus à l'aide de la fonction `setpgid`.
- D'autres appels système existent pour gérer les groupes de processus, nous ne présentons ici que les fonctions préconisées par POSIX.1-2001.
- Les groupes de processus sont surtout utilisés par les shells pour implémenter le contrôle des jobs.
- Les groupes de processus permettent, entre autre, d'envoyer un signal à un ensemble de processus.
- Un groupe de processus peut avoir un leader : il s'agit du processus qui a le même identifiant que celui du groupe.

Groupes et sessions

Les appels système

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

pid_t getpgrp(void);
int setpgid(pid_t pid, pid_t pgid);
```

- *setpgid* fixe le groupe de processus du processus *pid* à *pgid*. Si *pid* vaut 0 alors on fixe le groupe du processus appelant. Si *pgid* vaut 0 alors le groupe de processus du processus *pid* est fixé à *pid*. On ne peut changer que pour un groupe de processus qui appartient à la même session que l'ancien.
- *setpgid* retourne -1 en cas d'erreur (et *errno* est fixée) et 0 sinon.

Voir l'exemple

Groupes et sessions

Les sessions

- Les sessions réunissent des groupes de processus.
- Généralement les sessions sont attachées à un terminal de contrôle.
- Au sein d'une session un groupe de processus est en avant plan.
- Pour créer une nouvelle session un processus ne doit pas être leader de son groupe, la création d'une session créant un nouveau groupe de processus.

Groupes et sessions

Les appels système

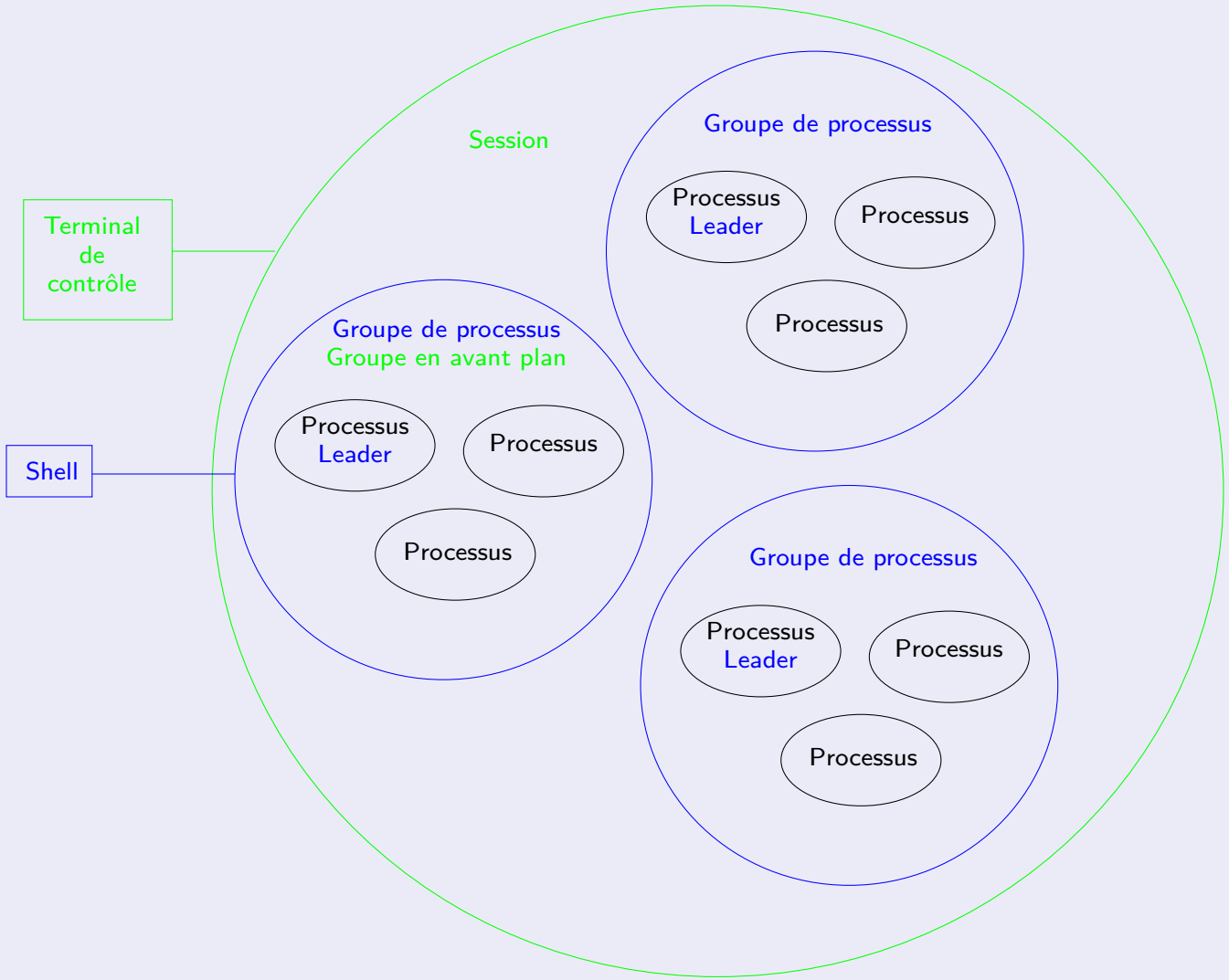
```
#include <sys/types.h>
#include <unistd.h>

pid_t getsid(pid_t pid);
pid_t setsid(void);
```

- *getsid*(0) retourne l'identifiant de session du processus appelant.
- *setsid* crée une nouvelle session si le processus n'est pas leader de son groupe. L'identifiant de la nouvelle session (et du nouveau groupe) est celui du processus appelant.
- *getsid* et *setsid* retournent -1 en cas d'erreur (et *errno* est fixée) et l'identifiant de la session sinon.

Voir l'exemple

Groupes et sessions

Pour résumer...



Utilisateur d'un processus

Utilisateur et processus

- Afin que le système puisse gérer correctement les permissions à attribuer à un processus, celui-ci est associé à un utilisateur.
- Chaque processus s'exécute sous une identité précise, il s'agit de l'*UID* (*User IDentifier*) du processus.
- Il existe plusieurs *UID* : l'*UID* réel, l'*UID* effectif et l'*UID* sauvé
- Sous Linux, il en existe même un quatrième dont nous ne parlerons pas : le *FS-UID*.

Utilisateur d'un processus

Utilisateur et processus

- L'*UID* réel est celui de l'utilisateur ayant lancé le programme.
- L'*UID* effectif correspond aux privilèges actuellement accordés au processus.
- L'*UID* sauvé est celui de l'ancien *UID* effectif lorsque ce dernier a été modifié par le processus.
- L'*UID* réel et l'*UID* effectif peuvent être différents si le fichier exécutable de la commande a son bit « Set-UID » de positionné.
- Le bit « Set-UID » permet de fixer l'*UID* effectif du processus à l'*UID* du propriétaire du fichier.

Utilisateur d'un processus

Utilisateur et processus

- Un processus ne peut que perdre des privilèges ou en retrouver des anciens.
- Un processus peut demander à remplacer son *UID* effectif par son *UID* réel puis récupérer son ancien *UID* (d'où l'utilité de l'*UID* sauvé).

Utilisateur d'un processus

Les appels système

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

uid_t getuid(void);
uid_t geteuid(void);
int setuid(uid_t euid);
int seteuid(uid_t euid);
int setreuid(uid_t ruid, uid_t euid);
```

Utilisateur d'un processus

Les appels système

- *getuid* retourne l'*UID* réel du processus.
- *geteuid* retourne l'*UID* effectif du processus.
- *setuid* fixe l'*UID* effectif du processus.
  - Il retourne -1 en cas d'échec et 0 sinon.
  - Attention, POSIX spécifie que si l'*UID* effectif était celui de root alors l'*UID* réel et l'*UID* sauvé sont également fixés. Cela empêche le processus de récupérer un *UID* effectif root.

Voir l'exemple

Utilisateur d'un processus

Les appels système

- *seteuid* fixe l'*UID* effectif du processus.
- Il retourne -1 en cas d'échec et 0 sinon.
- Comme pour *setuid*, un utilisateur non privilégié ne peut fixer son *UID* effectif qu'à son *UID* réel ou son *UID* sauvé.
- *seteuid* ne modifie pas l'*UID* sauvé.

Voir l'exemple

Utilisateur d'un processus

Les appels système

- *setreuid* fixe les *UID* réel et effectif. Une valeur -1 demande de ne pas modifier l'*UID* correspondant.
- On retrouve le même comportement pour l'*UID* effectif qu'avec *seteuid*.
- POSIX est très flou quant au comportement de *setreuid* vis-à-vis de l'*UID* réel.

Remarque

L'appel système *getresuid* qui permet de récupérer l'*UID* sauvé est une extension GNU.

Groupes d'utilisateurs d'un processus

Groupes d'utilisateurs et processus

- Un processus est également associé à un groupe d'utilisateurs *GID* (à ne pas confondre avec le groupe de processus).
- Comme pour l'*UID*, il existe un *GID* effectif, réel et sauvé. Le *GID* réel correspond au groupe principal de l'utilisateur ayant lancé le programme.
- Le *GID* effectif peut être différent si le fichier exécutable a son bit « Set-GID » positionné.

Groupes d'utilisateurs d'un processus

Les appels système

Voici les appels système permettant de gérer le groupe d'utilisateurs d'un processus. Ils fonctionnent de façon similaire à ceux concernant les *UID*.

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

gid_t getgid(void);
gid_t getegid(void);
int setgid(gid_t egid);
int setegid(gid_t egid);
int setregid(gid_t rgid, gid_t egid);
```

Groupes d'utilisateurs d'un processus

Les appels système

Un utilisateur pouvant appartenir à plusieurs groupes, un processus possède également une liste de groupes supplémentaires. Cette liste peut être obtenue à l'aide de l'appel système *getgroups* :

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

int getgroups(int size, gid_t list[]);
```

Groupes d'utilisateurs d'un processus

Les appels système

- *getgroups* stocke dans *list* l'ensemble des groupes supplémentaires du processus.
- POSIX ne spécifie pas si le groupe effectif doit être retourné dans *list*.
- *size* indique le nombre maximal d'éléments pouvant être stockés dans *list*.
- Si *size* est trop petite, une erreur est retournée (valeur de retour -1 et *errno* est fixée à *EINVAL*).
- En cas de succès, la valeur de retour indique le nombre d'éléments remplis.
- Si *size* vaut 0, *list* n'est pas modifiée et le nombre de groupes supplémentaires est retourné.

Voir l'exemple

Création

L'appel système *fork*

L'appel système *fork* permet de créer un nouveau processus qui sera une copie du processus effectuant l'appel système :

```
#include <unistd.h>

pid_t fork(void);
```

Le processus ayant initié l'appel à *fork* (le père) et le processus ayant été créé (le fils) reçoivent des valeurs de retour différentes :

- le père reçoit l'identifiant du processus créé ;
- le fils reçoit 0.
- -1 est retourné dans le père en cas d'erreur (le fils n'étant pas créé).

Voir l'exemple (1)
Voir l'exemple (2)

Terminaison

Fin normale d'un processus

Un processus se termine

- lorsqu'il rencontre un `return` dans la fonction *main*
- lorsqu'il fait appel à la fonction *exit*

```
#include <stdlib.h>

void exit(int status);
```

La valeur de retour du programme est *status* & 0377.

Attente de processus

Processus zombies

- Un processus qui se termine est placé dans l'état zombie.
- Le père d'un processus peut libérer les ressources occupées par un processus zombie en utilisant les appels système *wait* ou *waitpid*.

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Attente de processus

Appels système `wait` et `waitpid`

1. Par défaut, ces appels système attendent qu'un fils se termine.
2. Si un fils s'est déjà terminé alors ces appels retournent immédiatement.
3. Si le processus n'a pas de fils alors l'appel retourne -1 et `errno` est fixé à `ECHILD`.

- Si `status` n'est pas `NULL`, il contiendra des informations sur l'état du fils qui s'est terminé.
- Les informations contenues dans `status` peuvent être consultées *via* des macros comme `WIFEXITED`, `WEXITSTATUS` (*cf.* le `man`).

Voir l'exemple

Attente de processus

Appel système *waitpid*

Pour l'appel système *waitpid*, *pid* indique quels fils doivent être attendus :

- < -1 : attendre la fin d'un fils appartenant au groupe de processus d'identifiant *-pid*.
- -1 : attendre n'importe lequel des processus fils (idem que *wait*).
- 0 : attendre la fin d'un fils appartenant au même groupe de processus que l'appelant.
- > 0 : attendre la fin du fils d'identifiant *pid*.

Attente de processus

Appel système *waitpid*

option permet de modifier le fonctionnement de l'appel système :

- *WNOHANG* permet de ne pas bloquer si aucun fils ne s'est terminé.
- *WUNTRACED* permet d'attendre également les fils qui passent à l'état stoppé.
- *WCONTINUED* permet d'attendre également les fils stoppés qui ont été relancés.

Remarque

Il existe également un appel système *waitid* permettant de contrôler plus précisément les changements d'états des processus fils.

Recouvrement de processus

Appel système *execve*

Il est possible de recouvrir le code d'un processus par celui d'un autre programme en utilisant la fonction *execve*.

```
#include <unistd.h>

int execve(const char *filename, char *const argv[],
           char *const envp[]);
```

Recouvrement de processus

Appel système *execve*

- Les segments de texte, de données ainsi que la pile du processus appelant sont alors remplacés par ceux du programme *filename*.
- Les tableaux *argv* et *envp* doivent se terminer par la valeur *NULL*.
- Les tableaux *argv* et *envp* sont accessibles dans le programme appelé grâce à la fonction *main* et la variable externe *environ* :

```
extern char **environ; int main(int argc, char *argv[]);
```

- *envp* est un tableau de chaînes de la forme *clé=valeur*.

Exemple d'utilisation d'*execve*
Exemple d'utilisation d'*environ*

Recouvrement de processus

Variable *environ*

Traditionellement, sous UNIX, on pouvait déclarer la fonction *main* avec le prototype suivant :

```
int main(int argc, char *argv[], char *envp[]);
```

Cependant la norme ISO du langage C n'autorise pas la déclaration du paramètre *envp* dans la fonction *main*.

Recouvrement de processus

Appels système *exec*

En fait POSIX définit différentes variantes pour *exec* :

```
#include <unistd.h>

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg,
           ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

Remarque

La fonction *execvpe* est une extension GNU et n'est pas un appel système POSIX.

Recouvrement de processus

Appels système *exec*

- Le suffixe *l* indique que les arguments sont transmis les uns à la suite des autres comme paramètre de la fonction variadique.
- Le suffixe *v* indique que les arguments sont transmis à travers un tableau comme pour *execve*.
- Le suffixe *p* indique que le programme doit être recherché, comme le ferait le shell, en utilisant la variable *PATH*.
- Le suffixe *e* indique que le dernier paramètre de la fonction est un tableau contenant les variables d'environnement à transmettre.

Plan

2. Processus
   - Introduction
   - Les appels système POSIX
   - Ordonnancement

Ordonnancement

Définition

- À un instant donné, plusieurs processus peuvent être en concurrence pour l'obtention de temps processeur.
- Un choix doit alors être fait quant au prochain processus à exécuter.
- La partie du système d'exploitation qui effectue ce choix s'appelle l'**ordonnanceur**.
- La méthode utilisée pour réaliser le choix est la **politique d'ordonnancement**.

Classification

Comportement des processus

- Les processus alternent les phases de traitement et les requêtes d'entrée/sortie (E/S).
- Lorsqu'un processus est en attente d'une E/S, il est bloqué. Un bon ordonnanceur ne lui accorde donc pas de temps CPU.

Ordonnancement

Quand ordonnancer ?

- Création d'un nouveau processus.
- Fin d'un processus.
- Blocage d'un processus sur une E/S.
- Interruption :
  - E/S ;
  - Top d'horloge ;
  - Appel système.

Ordonnancement

Ordonnancement préemptif/non préemptif

- Un ordonnanceur **non préemptif** sélectionne un processus puis le laisse s'exécuter jusqu'à ce qu'il bloque ou qu'il libère volontairement le processeur.
- Un ordonnanceur **préemptif** sélectionne un processus puis le laisse s'exécuter pendant un délai déterminé. À la fin du délai, un autre processus peut être sélectionné.

Classification

Types de processus

On peut distinguer trois types de processus :

1. Les processus **interactifs** qui effectuent beaucoup d'E/S pour communiquer avec l'utilisateur.
   - Ces processus passent le plus clair de leur temps à être bloqués.
   - Par contre, ils doivent traiter le résultats des E/S rapidement car sinon l'utilisateur aura l'impression d'un système « qui se traine ».
   - On cherchera à traiter les processus selon les attentes des utilisateurs.
   - Un ordonnancement préemptif est alors obligatoire.

Classification

Types de processus (suite)

2. Les processus de **traitement** qui n'ont pas besoin d'interaction avec l'utilisateur.
   - Ces processus, qui tournent généralement en tâche de fond, peuvent être pénalisés par les ordonnanceurs.
   - Pour les environnement qui n'effectuent que des traitements de ce type (traitements par lots), un ordonnancement non préemptif ou préemptif avec de longs délais peut être utilisé.
   - On cherchera à optimiser le nombre de processus traités à l'heure ainsi qu'à réduire le délai entre la soumission et l'achèvement. Le processeur devra être occupé en permanence.

Classification

Types de processus (suite)

3. Les processus **temps réel** qui ont des exigences d'ordonnancement importantes.
   - Ils doivent être rapidement traités.
   - Les temps de réponses doivent être prévisibles.
   - Ce type de processus n'a pas forcément besoin d'ordonnancement préemptif car il s'agit la plupart du temps de processus courts, bloquant rapidement.

Types de processus (suite)

Dans tous les cas, l'algorithme d'ordonnancement devra faire en sorte que toute les parties du système soient occupées.

Définitions

Mesures

- Le **temps de réponse** est le temps écoulé entre le moment de l'émission d'une commande et la réception du résultat attendu.
- Le **délai de rotation** est la moyenne des temps de réponses.
- Pour les mainframes, on définit également la **capacité de traitement** qui est le nombre de jobs menés à bien par heure.

Politiques d'ordonnancement

Premier arrivé, premier servi

- L'ordonnancement **premier arrivé, premier servi** (ou FCFS pour *first-come first-served*) est un ordonnancement non préemptif qui consiste à exécuter les processus selon leur ordre d'arrivée.
- On dispose d'une seule file d'attente dans laquelle sont placés les processus au fur et à mesure de leur création.
- Lorsqu'un processus se bloque, le processus suivant dans la file d'attente est exécuté.
- Lorsqu'un processus est débloqué, il est replacé en fin de liste.

Politiques d'ordonnancement

Exemple

Prenons l'exemple de quatre processus étant arrivés quasi simultannément au temps 0 et nécessitant les temps d'exécution suivants :

A = 5ms B = 2 ms C = 3 ms D = 6 ms.

On aura un délai de rotation de 9,5 et les temps de réponses suivants :

$R_A = 5$ $R_B = 7$ $R_C = 10$ $R_D = 16$

Politiques d'ordonnancement

Exécution du job le plus court en premier

- L'ordonnancement **le job le plus court en premier** (*shortest job first*) est un ordonnancement non préemptif qui suppose que les délais d'exécution soient connus à l'avance.
- On dispose d'une liste dans laquelle sont placés les processus.
- L'ordonnanceur choisit le processus qui possède le plus court délai d'exécution.
- Ce type d'ordonnancement est optimal pour le délai de rotation si tous les processus sont disponibles au même moment.
- Ce type d'ordonnancement est surtout valable sur les systèmes à traitement par lot.

Politiques d'ordonnancement

Exemple

Avec l'exemple précédent :

A = 5ms B = 2 ms C = 3 ms D = 6 ms.

On aura un délai de rotation de 8,25 et les temps de réponses suivants :

$R_A = 10$ $R_B = 2$ $R_C = 5$ $R_D = 16$

Politiques d'ordonnancement

Tourniquet

- L'ordonnancement du **tourniquet** (ou ***round robin***, tournoi) est un algorithme préemptif où chaque processus se voit attribuer un intervalle de temps, appelé **quantum**.
- À l'issue de son quantum, le processus est interrompu et le processeur est attribué à un autre processus.
- Si le processus est bloqué ou se termine, l'ordonnancement s'effectue naturellement.
- L'algorithme utilise une file d'attente pour stocker les processus dans l'état prêt.

Politiques d'ordonnancement

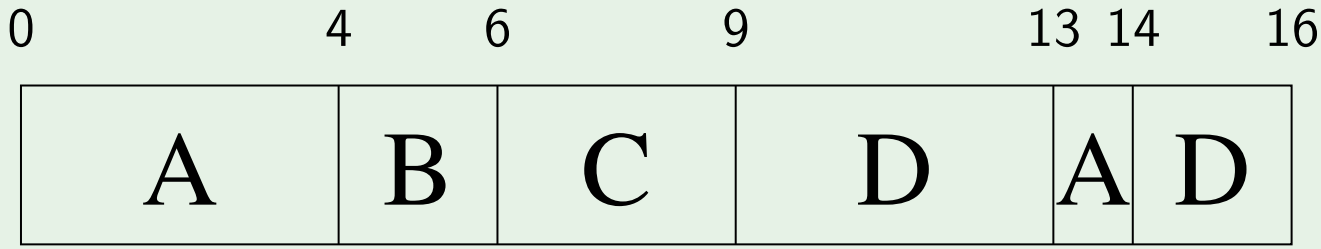
Exemple

Avec l'exemple précédent :

A = 5ms B = 2 ms C = 3 ms D = 6 ms.

On choisit un quantum de 4 ms.
On aura un délai de rotation de 11,25 et les temps de réponses suivants :

$R_A = 14$ $R_B = 6$ $R_C = 9$ $R_D = 16$



Politiques d'ordonnancement

Tourniquet

Toute la difficulté réside dans le choix du quantum : avec un quantum trop court le coût des changements de contexte fera chuter l'efficacité du processeur alors qu'avec un quantum trop long les temps de réponses aux requêtes interactives deviendront trop longs.
Par exemple, supposons que le temps de changement de contexte soit de 1 ms.

- Si on fixe le quantum à 4 ms le processeur passera 20 % de son temps à changer de contexte.
- Si on fixe le quantum à 100 ms, le temps passé à changer de contexte sera alors de 1 %. Mais si 10 utilisateurs appuient sur une touche en même temps et que chaque processus utilise tout son quantum, le 10ème verra sa demande traitée au bout de plus d'une seconde.

Politiques d'ordonnancement

Listes de priorités

- L'ordonnancement par **priorités** consiste à affecter une priorité à chaque processus.
- Le processus de plus haute priorité s'exécute en premier.
- L'algorithme peut être préemptif ou non préemptif.
- Pour éviter qu'un processus de haute priorité ne s'exécute indéfiniment, les priorités peuvent être dynamiques : elles sont réévaluées à chaque décision d'ordonnancement (voire à chaque interruption d'horloge).
- On peut gérer des listes de processus. Chaque liste contient les processus de même priorité.
- Chaque liste peut alors être gérée avec un algorithme d'ordonnancement différent.

Politiques d'ordonnancement

Remarque

- Il existe bien d'autres algorithmes d'ordonnancement.
- Certains prennent en compte le propriétaire de chaque processus afin de servir tous les utilisateurs de façon équitable.
- Les systèmes temps réel (avec ou sans tolérance) possèdent des algorithmes spécifiques.
- Les algorithmes vus doivent être adaptés à la gestion des threads (en espace utilisateur ou noyau).

Ordonnancement sous Linux

Exemple : le cas Linux

- Nous ne donnerons pas ici d'explications détaillées de l'algorithme (il diffère selon les versions de noyau). Mais nous essaierons de donner une idée de son mode de fonctionnement.
- L'ordonnancement sous Linux repose sur des listes de priorités.
- Les processus possèdent une priorité statique qui peut varier entre 0 et 99.
- Les processus de priorité statique nulle possèdent également une priorité dynamique (qui varie en fonction du temps d'inactivité du processus).

Ordonnancement sous Linux

Exemple : le cas Linux

- Les processus de priorité statique supérieure à 1 sont appelés processus temps-réel.
- L'ordonnanceur choisira toujours le processus de plus grande priorité statique.
- Un processus possède également un attribut indiquant le type d'ordonnancement qu'il faut lui appliquer.
- Les processus temps-réel peuvent être ordonnancés selon une politique de tourniquet (*SCHED_RR*) ou une politique de type FIFO (*SCHED_FIFO*).

Ordonnancement sous Linux

Exemple : le cas Linux

- Les processus de priorité statique nulle correspondent aux processus « normaux » de Linux.
- Il sont ordonnancés selon la valeur de leur priorité dynamique (*SCHED_OTHER*).
- Il existe également deux autres politiques d'ordonnancement qui peuvent être appliquées à un processus : *SCHED_IDLE* (processus à très faible priorité) et *SCHED_BATCH* (processus non interactif).

Ordonnancement sous Linux

Valeur de courtoisie

- La priorité dynamique d'un processus est calculée à partir de sa valeur de courtoisie et de sa période d'inactivité.
- On peut modifier la valeur de courtoisie d'un processus à l'aide de l'appel système *nice*.

```
#include <unistd.h>

int nice(int inc);
```

Ordonnancement sous Linux

nice

- *inc* est ajouté à la valeur de courtoisie du processus.
- *inc* peut être négatif (réservé à root).
- Plus la valeur est grande, moins le processus est prioritaire.
- Sous Linux (depuis le noyau 1.3.43), la valeur de courtoisie est comprise entre -20 et 19.
- POSIX spécifie que l'appel système retourne la nouvelle valeur de courtoisie.
- -1 est donc une valeur de retour possible. Afin de vérifier s'il s'agit d'une erreur ou pas, il faut fixer *errno* à 0 avant l'appel puis vérifier sa valeur ensuite.

Voir l'exemple

Ordonnancement sous Linux

Valeur de courtoisie

On peut également consulter et modifier la valeur de courtoisie de tous les processus d'un groupe de processus ou d'un utilisateur.

```
#include <sys/time.h>
#include <sys/resource.h>

int getpriority(int which, id_t who);
int setpriority(int which, id_t who, int prio);
```

- *which* peut prendre les valeurs *PRIO_PROCESS*, *PRIO_PGRP* ou *PRIO_USER*.
- *who* correspond à l'identifiant de l'objet indiqué par *which*.
- *prio* correspond à la nouvelle valeur de courtoisie.
- La nouvelle valeur est retournée (-1 en cas d'erreur).
- *getpriority* retourne la priorité la plus élevée des processus concernés.

Ordonnancement sous Linux

Céder le processeur

Un processus peut céder le processeur au prochain processus éligible grâce à l'appel système *sched_yield* :

```
#include <sched.h>

int sched_yield(void);
```

Ordonnancement sous Linux

Politiques d'ordonnancement

Les appels système *sched_getscheduler* et *sched_setscheduler* permettent de manipuler les politiques d'ordonnancement :

```
#include <sched.h>

int sched_setscheduler(pid_t pid, int policy,
                       const struct sched_param *param);
int sched_getscheduler(pid_t pid);

struct sched_param {
  ...
  int sched_priority;
  ...
};
```

Les politiques d'ordonnancement peuvent être *SCHED_OTHER*, *SCHED_BATCH*, *SCHED_IDLE*, *SCHED_FIFO* et *SCHED_RR*.

Ordonnancement sous Linux

Politiques d'ordonnancement

Les appels système *sched_getparam* et *sched_setparam* permettent de manipuler les paramètres de la politique d'ordonnancement du processus :

```
#include <sched.h>

int sched_setparam(pid_t pid,
                   const struct sched_param *param);
int sched_getparam(pid_t pid, struct sched_param *param);
```

Ordonnancement sous Linux

Politiques d'ordonnancement

Les intervalles des priorités statiques des différentes politiques d'ordonnancement peuvent être obtenus par les appels système suivants :

```
#include <sched.h>

int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```

Ordonnancement sous Linux

Ordonnancement des threads

Sous Linux, les politiques d'ordonnancement dépendent des threads. On peut également utiliser les appels système :

```
int pthread_setschedparam(pthread_t target_thread,
        int politique,
        const struct sched_param *param);

int pthread_getschedparam(pthread_t  target_thread,
        int  *politique,
        struct sched_param *param);

int pthread_setschedprio(pthread_t thread, int prio);
```

Remarque

Sous Mac OS X, seuls ces appels système sont disponibles.

Plan

1. Introduction
2. Processus
3. Les threads
4. Les fichiers
5. Communication et synchronisation
6. Les signaux

Plan

3. Les threads
   - Introduction
   - Les threads POSIX
   - Threads et concurrence
   - Threads et processus

Définition

Les threads

- Le terme de *thread* peut se traduire par « fil d'exécution »
- Il s'agit du déroulement particulier de l'exécution d'un programme
- Un thread est constitué d'une pile et d'un contexte d'exécution (registres du processeur et compteur ordinal)

Multi-threading

Processus et threads

- Jusqu'alors nous n'avons vu que des processus constitués d'un seul thread et de données (statiques et dynamiques).
- Un processus peut disposer de plusieurs threads, l'espace d'adressage est alors partagé par les différents threads
- Utiliser plusieurs threads permet de traiter différentes tâches en parallèle sans avoir à créer plusieurs processus gourmants en ressources
- Les threads sont aussi parfois appelés « processus légers »

Multi-threading

Processus et threads

Le tableau suivant récapitule quelques éléments qui sont partagés par les threads d'un même processus ainsi que ceux qui sont propres à chaque thread.

Données processus	**Données threads**
Variables globales	Compteur ordinal
Variable dynamiques	Registres
Processus enfants	Pile
Fichiers ouverts	État
Signaux et gestionnaires de signaux	Masque des signaux
...	...

Remarque

Il faut bien noter que les variables locales étant stockées dans la pile, elles sont propres à chaque threads.

Programmation multi-threads

Avantage

Les données étant partagées, la communication entre threads est plus facile à implémenter que la communication entre les processus.

L'inconvénient de l'avantage

Les données étant partagées, l'accès aux données est plus délicat à gérer lorsque plusieurs threads sont mis en œuvre

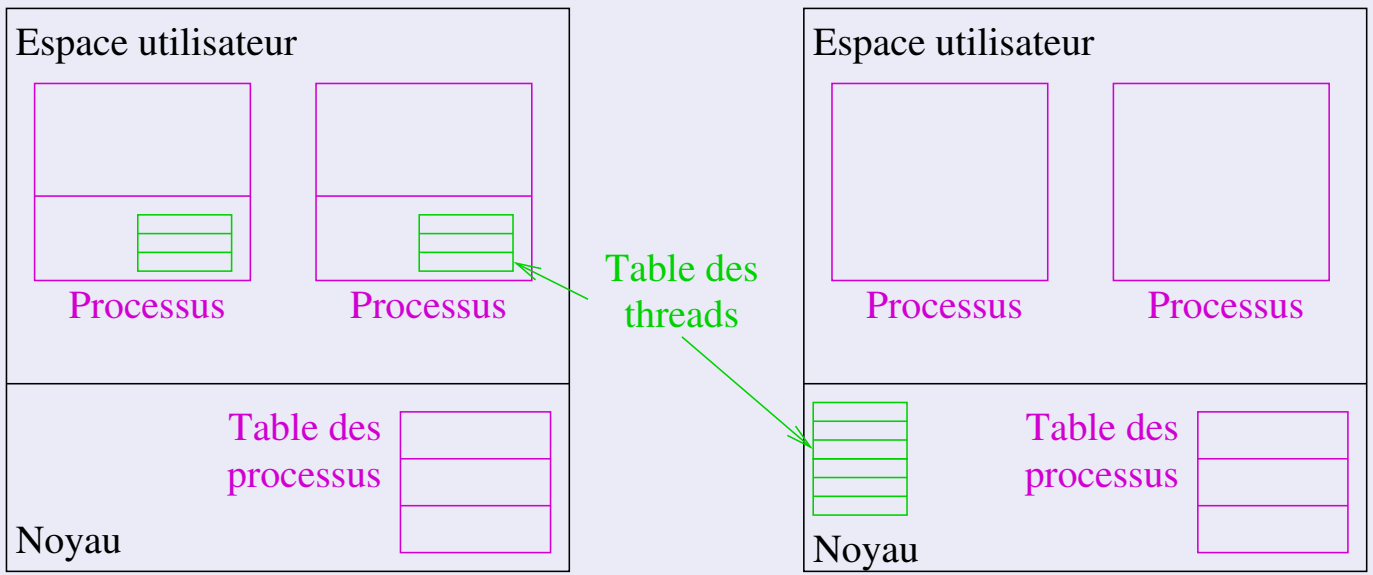
Implantation des threads

Espace utilisateur ou noyau

- Les threads peuvent être implantés dans l'espace utilisateur ou dans le noyau.
- Lorsque les threads sont implantés dans l'espace utilisateur, il s'agit d'une bibliothèque qui prend en charge les fonctions de manipulation des threads : le noyau ne voit que des processus (exemple : GNU pth).
- Lorsque les threads sont implantés dans le noyau, ce dernier gère alors une table des threads en plus de la table des processus. L'ordonnancement se fait au niveau des threads (NPTL : *Native POSIX Thread Library*).
- Il existe également des implantations mixtes ou les threads du noyau peuvent être multiplexés en plusieurs threads utilisateurs (comme sur NetBSD).

Implantation des threads

Espace utilisateur ou noyau



Implantation des threads

Avantages et inconvénients

- La gestion des appels systèmes bloquants est beaucoup plus facile avec les threads noyau
- L'implémentation des threads dans l'espace utilisateur permet d'éviter de nombreuses commutations espace utilisateur/noyau ce qui la rend nettement plus efficace.

Plan

3. Les threads
   - Introduction
   - Les threads POSIX
   - Threads et concurrence
   - Threads et processus

Les threads POSIX

threads POSIX

- POSIX définit une interface de programmation permettant de gérer les threads au travers de la norme POSIX.1c, *Threads extensions*.
- L'interface des threads POSIX est souvent appelée pthreads
- Cette interface peut être implémentée à l'aide de threads en espace utilisateur ou de threads noyau.

Les threads POSIX sous Linux

La bibliothèque NPTL

- Pour les TP, nous utiliserons la bibliothèque NPTL qui utilise des threads noyau.
- Les fonctions sont définies dans le fichier d'en-tête *pthread.h*
- Afin de pouvoir compiler et réaliser l'édition des liens en utilisant la bibliothèque NPTL, il faut ajouter avec gcc l'option -pthread

Exemple

```
gcc -c -Wall -pthread -o test_pthreads.o test_pthreads.c
gcc -pthread -o test_pthreads test_pthreads.o
```

Les threads POSIX sous Linux

ps et les thread

- Par défaut la commande *ps* n'affiche que les processus

```
  PID TTY          TIME CMD
 2526 pts/0    00:00:00 emacs
```

- Afin de visualiser les threads avec *ps* on peut utiliser l'option *m* :

```
# ps m
  PID TTY      STAT   TIME COMMAND
 2526 pts/0    -      0:00 emacs
    - -        Sl     0:00 -
    - -        Sl     0:00 -
    - -        Sl     0:00 -
```

ps et les thread

- On peut aussi ajuster l'affichage de *ps* avec l'option *o* :

```
# ps mo pid,tid,tty,stat,time,cmd
  PID    TID TT       STAT      TIME CMD
 2526      - pts/0    -      00:00:00 emacs
    -   2526 -        Sl     00:00:00 -
    -   2530 -        Sl     00:00:00 -
    -   2531 -        Sl     00:00:00 -
```

Les threads POSIX

Création d'un thread

Pour créer un nouveau thread, on utilise la fonction *pthread_create* :

```
int pthread_create(pthread_t * thread,
                   pthread_attr_t * attr,
                   void * (*start_routine)(void *),
                   void * arg);
```

- Cette fonction crée un nouveau thread qui va s'exécuter parallèlement au thread appelant
- L'identifiant du nouveau thread sera stocké dans la variable pointée par *thread*

Les threads POSIX

Création d'un thread

- Le nouveau thread exécutera la fonction *start_routine* passée en paramètre.
- *start_routine* attend un argument de type `void *` qui est spécifié à l'aide du paramètre *arg*.
- *attr* permet de spécifier les attributs du threads. *NULL* indique qu'il faut utiliser les attributs par défaut.
- La fonction retourne 0 en cas de succès et un code d'erreur sinon.

Voir l'exemple

Les identifiants de threads POSIX

pthread_t

- Les fonctions de gestion des threads utilisent des identifiants de thread pour repérer les threads à manipuler
- Ces identifiants sont de type *pthread_t*
- Ce type est un type opaque (entier, structure ou autre)
- Un thread peut récupérer son identifiant à l'aide de la fonction *pthread_self* :

```
pthread_t pthread_self(void);
```

- Pour savoir si deux identifiants désignent le même thread, on utilise la fonction *pthread_equal* :

```
int pthread_equal(pthread_t thread1,
                  pthread_t thread2);
```

Cette fonction retourne une valeur non nulle si *thread1* et *thread2* correspondent au même thread, 0 sinon.

Les identifiants de threads POSIX

TID et identifiant POSIX

- Attention : le TID affiché par *ps* est spécifique à Linux et ne correspond pas à l'identifiant de thread POSIX utilisé par les fonctions POSIX de manipulation des threads.
- L'appel système permettant de récupérer le TID d'un thread est *gettid* mais cet appel est spécifique à Linux et n'est pas portable
- De plus *gettid* ne possède pas de fonction « enveloppe » dans la glibc, il faut donc utiliser *syscall* pour l'utiliser

Vie et mort d'un thread

Que fait ce programme ?

```
void * run(void * arg) {
  printf("Bonjour\n");
  return NULL;
}

int main(void) {
  pthread_t th;
  if (pthread_create(&th, NULL, run, NULL) != 0) {
    fprintf(stderr, "Erreur\n");
    exit(EXIT_FAILURE);
  }
  return EXIT_SUCCESS;
}
```

Vie et mort d'un thread

Quelques définitions

- Nous appelerons **thread principal** le thread qui est lancé au démarrage du processus, il s'agit du thread qui exécute la fonction *main*
- Un thread qui n'est pas le thread principal sera un **thread secondaire**
- La fonction exécutée au lancement d'un thread secondaire (argument *start_routine* de *pthread_create*) sera la **fonction de démarrage** du thread

Vie et mort d'un thread

Terminaison du thread ou du processus

- Un appel à *exit* (ou un **return** dans la fonction *main*) termine le processus : tous les threads sont donc interrompus
- Un thread secondaire se termine lorsqu'il termine l'exécution de sa fonction de démarrage, les autres threads ne sont pas interrompus
- La fonction *pthread_exit* permet de stopper l'exécution d'un thread sans interrompre les autres threads :

```
void pthread_exit(void *retval);
```

retval sera la valeur du retour de thread, nous verrons plus loin comment la consulter (elle est différente de la valeur de retour du processus)

Vie et mort d'un thread

Que fait ce programme ? (le retour)

Dans l'exemple précédent, pour être sûr d'afficher « Bonjour », il faut terminer le thread principal avec *pthread_exit*.

```
void * run(void * arg) {
  printf("Bonjour\n");
  return NULL;
}

int main(void) {
  pthread_t th;
  if (pthread_create(&th, NULL, run, NULL) != 0) {
    fprintf(stderr, "Erreur\n");
    exit(EXIT_FAILURE);
  }
  pthread_exit(NULL);
}
```

Voir le source

Vie et mort d'un thread

Remarques

- Si le dernier thread qui se termine ne fait pas appel à `exit` (ou à `return` dans la fonction `main`), le processus se termine comme si on avait fait appel à `exit(0);`
- Sous Linux, lorsque le thread principal est interrompu par `pthread_exit` et qu'il reste des threads en cours d'exécution, le processus est placé en état zombie sans pour autant que son état puisse être récupéré avec `wait`.

Les attributs de thread

Les attributs de thread

- L'état d'un thread contient - entre autre - des attributs
- Ces attributs sont fixés à sa création et peuvent être modifiés par la suite
- Il existe deux types d'attributs de thread :
  - Les attributs permettant de fixer les propriétés d'ordonnancement du thread (4 attributs). Il s'agit d'une extension de POSIX, nous ne détaillerons pas ici ces attributs.
  - Un attribut permettant d'indiquer si le thread est **joignable** ou **détaché**.

Les threads joignables

Threads joignables

- Quand l'exécution d'un thread joignable se termine, il est placé dans un état similaire à l'état zombie pour les processus
- Afin de libérer définitivement les ressources utilisées par un thread joignable, un autre thread doit faire appel à la fonction *pthread_join* :

```
int pthread_join(pthread_t th, void **thread_return);
```

- L'appel à *pthread_join* est bloquant jusqu'à ce que le thread attendu se termine
- Si *thread_return* est non *NULL*, la valeur de retour du thread est stockée dans **thread_return*
- La valeur de retour d'un thread est la valeur de retour de la fonction de démarrage du thread ou la valeur de l'argument fourni à l'appel à *pthread_exit* qui a mené à la terminaison du thread

Les threads joignables

Remarques

- Contrairement aux processus, la valeur de retour d'un thread n'est pas un entier mais un pointeur
- Cette valeur devrait toujours correspondre à une adresse valide ou à `NULL`
- La valeur de retour récupérée par `pthread_join` peut également valoir `PTHREAD_CANCELED` qui ne correspond pas à une adresse valide et qui ne vaut pas `NULL`

Les threads joignables

Remarques (suite)

- La pile du thread étant libérée une fois qu'il a été joint, sa valeur de retour ne doit pas pointer sur une de ses variables automatiques (mauvais exemple)
- À l'exeption de *PTHREAD_CANCELED*, la sémantique de la valeur de retour d'un thread n'est pas définie par la norme : elle dépend de l'application
- Un thread détaché ne peut être joint : ses ressources sont libérées dès qu'il se termine

Exemple

Un exemple d'utilisation de thread joignable est disponible ici

Les attributs de thread

Les attributs de thread

- L'argument *attr* de la fonction *pthread_create* permet de spécifier les attributs du thread.
- Une valeur *NULL* permet de donner au thread les valeurs d'attributs par défaut
- Un autre moyen de fixer les attributs du thread aux valeurs par défaut est d'initialiser une variable de type *pthread_attr_t* à l'aide de la fonction *pthread_attr_init* :

```
int pthread_attr_init(pthread_attr_t *attr);
```

- Il faudra ensuite libérer les ressources utilisées par les attributs à l'aide de la fonction *pthread_attr_destroy()* :

```
int pthread_attr_destroy(pthread_attr_t *attr);
```

Voir l'exemple

Les attributs de thread

Joignable/détaché

- L'attribut « joignable » est l'attribut par défaut
- Pour fixer l'attribut, on peut utiliser *pthread_attr_setdetachstate* :

```
int pthread_attr_setdetachstate(pthread_attr_t *attr,
                                int detachstate);
```

  detachstate peut valoir *PTHREAD_CREATE_JOINABLE* ou *PTHREAD_CREATE_DETACHED*
- Il existe également une fonction *pthread_attr_getdetachstate* permettant de récupérer l'état de l'attribut dans une variable de type *pthread_attr_t*

Les attributs de thread

Joignable/détaché

- *pthread_detach* permet de modifier l'attribut du thread une fois celui-ci créé :

```
int pthread_detach(pthread_t th);
```

- On ne peut pas rendre joignable un thread détaché
- S'il y a déjà un thread bloqué avec *pthread_join* sur le thread alors *pthread_detach* est sans effet

Voir l'exemple

Transmettre des paramètres

Le paramètre de la fonction de démarrage

- La fonction de démarrage d'un thread a le prototype suivant :

```
void* start_routine(void *);
```

- On peut donc transmettre un paramètre de type pointeur à la fonction de démarrage
- Ce paramètre correspond au quatrième argument de la fonction *pthread_create*
- Il s'agit en général d'un pointeur sur une variable dynamique contenant les paramètres nécessaires à l'exécution du thread

Voir l'exemple

Transmettre des paramètres

Remarque

- Vous trouverez certains exemples où l'argument transmis au thread est en fait un entier qui est converti en `void*`.
- Comme la norme ISO indique que la conversion d'un entier en un pointeur et vice versa dépend de l'implémentation, ce type de manipulation est à proscrire.

Plan

3. Les threads
   - Introduction
   - Les threads POSIX
   - Threads et concurrence
   - Threads et processus

La gestion des erreurs

errno

- Les fonctions de manipulation des threads n'utilisent pas *errno*
- Cependant POSIX tient compte de l'existence de cette « variable » et stipule que *errno* est différente pour chaque thread
- Ainsi *errno* est définie comme étant une macro permettant de récupérer une *lvalue* spécifique à chaque thread.
- Sous Linux, *errno* fait appel à la fonction *__errno_location*() qui retourne l'adresse associée à la gestion des erreurs dans le thread

Les fonctions sûres

Que fait ce code ?

```
void* run(void* arg) {
  static int compteur = 0;

  printf("Valeur du compteur : %d\n", compteur);
  ++compteur;

  return NULL;
}

int main(void) {
  for (int i = 0; i < 100; ++i) {
    pthread_t th;
    if (pthread_create(&th, NULL, run, NULL) != 0) {
      fprintf(stderr, "pthread_create");
      exit(EXIT_FAILURE);
    }
  }
  pthread_exit(NULL);
}
```

Voir l'exemple

Les fonctions sûres

Les fonctions ré-entrantes

POSIX définit les fonctions ré-entrantes (*reentrant*) comme étant des fonctions dont l'effet, lorsqu'elles sont appelées par deux threads ou plus, est garanti être comme si les threads avaient exécutés la fonction l'un après l'autre dans un ordre indéfini même si, en réalité, les exécutions ont été entrelacées.

Les fonctions thread-safe

SUS définit les fonctions sûres pour les threads (*thread-safe*) comme étant des fonctions qui peuvent être appelées sans rique par plusieurs threads simultanément.

Les fonctions sûres

Remarques

- Que veut dire « sans risque » dans la définition des fonctions thread-safe ?
- Pour nous, une fonction sera thread-safe si elle est ré-entrante.
- La définition de « ré-entrance » est très fluctuante selon les sources. Par example, pour IBM, la ré-entrance se définit dans un contexte mono-threadé.
- De même la notion de thread-safety varie selon les systèmes. Ainsi, en C++ une fonction est thread-safe si elle synchronise l'accès à ses données partagées (ou plus exactement, s'il n'y a pas de *data race*).

Les fonctions sûres

Fonctions sûres

- Toute fonction qui n'utilise que des données locales (présentes sur la pile) sera une fonction ré-entrante.
- Toute fonction qui utilise un verrou afin qu'il y ait au plus un seul thread en train d'exécuter son code sera une fonction ré-entrante.

POSIX et les fonctions sûres

POSIX.1-2008 donne une liste exhaustive des fonctions qui ne sont pas nécessairement thread-safe (on peut consulter cette liste ici).

Les fonctions sûres

Version réentrantes

Certaines fonctions ne peuvent pas être ré-entrantes car elles retournent un pointeur sur une structure statique. C'est le cas, par exemple, de *getpwnam* :

```
struct passwd *getpwnam(const char *name);
```

Dans ces cas, POSIX propose une version ré-entrante avec un nom de fonction suffixée par *_r* :

```
int getpwnam_r(const char *name, struct passwd *pwd,
       char *buffer, size_t bufsize,
       struct passwd **result);
```

On peut remarquer que l'utilisation de la version ré-entrante de *getpwnam* rend la tâche du développeur un petit peu plus ardue...

Les fonctions sûres

Version réentrantes

De même, certaines fonctions ont besoin de mémoriser des données entre deux appels consécutifs. Elles ne sont alors pas sûres. C'est le cas de la fonction standard *strtok* :

```
char *strtok(char *restrict s1, const char *restrict s2);
```

Ici, une extension à la norme ISO, fournit une version ré-entrante de la fonction :

```
char *strtok_r(char *restrict s, const char *restrict s2,
       char **restrict lasts);
```

Les fonctions sûres

États internes

Certaines fonctions utilisent un état de configuration interne. Encore une fois, elles ne sont pas sûres. C'est le cas, par exemple de la fonction standard (C99) *mblen* :

```
int mblen(const char *s, size_t n);
```

Il faut alors utiliser la version permettant de transmettre l'état en paramètre :

```
size_t mbrlen(const char *restrict s, size_t n,
       mbstate_t *restrict ps);
```

Cette fonction n'est toutefois pas sûre si on fixe *ps* à *NULL* : cela revient à utiliser *mblen*.

Les fonctions sûres

Faites ce que je dis, pas ce que je fais...

- La norme POSIX n'assure pas que la fonction `strerror` soit sûre pour les threads.
- Il faut donc lui préférer la fonction `strerror_r` dès lors que deux threads sont suceptibles de l'utiliser.

Un exemple corrigé...

L'exemple précedent d'utilisation de `pthread_detach` est donc mal écrit. Il faudrait lui préférer la version utilisant `strerror_r` (ici).

Plan

3. Les threads
   - Introduction
   - Les threads POSIX
   - Threads et concurrence
   - Threads et processus

Duplication de processus multi-threadés

fork et threads

- Un appel à *fork* dans un processus contenant plusieurs threads provoque la création d'un nouveau processus ne contenant plus qu'un seul thread : il s'agit de la copie du thread qui a appelé *fork* dans le thread parent.
- Si on souhaite ré-initialiser des variables en cours d'utilisation par des threads autres que le thread appelant *fork*, on peut enregistrer des gestionnaires qui pourront être appelés lors de la duplication :

```
int pthread_atfork(void  (*prepare)(void),
       void  (*parent)(void),
       void  (*child)(void));
```

Plan

1. Introduction
2. Processus
3. Les threads
4. Les fichiers
5. Communication et synchronisation
6. Les signaux

Plan

4 Les fichiers
- Introduction
- Fichiers
- Les répertoires

Introduction

Stockage de l'information

Trois conditions essentielles doivent être remplies pour un stockage à long terme de l'information :

- Possibilité d'enregistrer une grande quantité d'information.
- Les informations doivent être persistantes : elle sont conservées après la fin du processus qui les utilise.
- Plusieurs processus doivent pouvoir accéder simultanément aux informations.

Afin de satisfaire ces conditions, on stocke généralement les informations sur des unités de stockage externe (disques durs, clef USB, ...).

Introduction

Fichiers

- Les disques peuvent être vus comme des suites séquentielles de blocs de taille fixe qui supportent deux opérations :
  - Lire le block k
  - Écrire le bloc k
- Les **fichiers** sont des abstractions permettant de faciliter le stockage de l'information par l'utilisateur.

Introduction

Système de fichiers

- La partie du système d'exploitation qui gère les fichiers est appelée le **système de fichiers** (ou système de gestion de fichiers, SGF).
- Il prend en charge la structure, le nommage, les accès, l'utilisation, la protection et l'implantation des fichiers.

Plan

4 Les fichiers
- Introduction
- Fichiers
- Les répertoires

Les fichiers

Les noms de fichiers

- Les fichiers sont une abstraction permettant de stocker des informations.
- On accède aux fichiers grâce à leur nom.
- De nombreux systèmes de fichiers gèrent des noms de fichiers en deux parties séparées par un point.
- La partie qui suit le point est appelée **extension**, elle correspond généralement au type du fichier.
- Les règles permettant de nommer un fichier diffèrent d'un système à l'autre (longueur, sensibilité à la casse, caractères spéciaux, ...).

Les fichiers

La structure des fichiers

- Un fichier peut être vu comme une suite d'octets. UNIX suit cette approche.
- D'autres systèmes d'exploitation gèrent des fichiers ayant une structure plus évoluée, il peut s'agir :
  - d'une suite d'enregistrements de longueur fixe ;
  - d'un ensemble d'enregistrements de longueurs variables et repérés par des clefs.

Les fichiers

Les types de fichiers

Selon les systèmes de fichiers, il peut exister différents types de fichiers :

- Les fichiers ordinaires.
- Les répertoires.
- Les liens.
- Les fichiers spéciaux caractères.
- Les fichiers spéciaux bloc.
- ...

Les fichiers

Les fichiers ordinaires

Les fichiers ordinaires peuvent être de deux types :

- texte : il s'agit de lignes de texte terminées par un saut de ligne (*CR* ou *LF* ou les deux selon les systèmes).
- binaire : les fichiers binaires ont une structure connue des programmes qui les exploitent.

Les fichiers

Les accès aux fichiers

Les systèmes d'exploitation peuvent proposer deux types d'accès aux fichiers :

- **Accès séquentiel** : les octets ou les enregistrements peuvent être lus dans l'ordre, en commençant par le début.
- **Accès direct** (ou accès aléatoire) : les octets ou les enregistrements peuvent être lus dans n'importe quel ordre.

Les fichiers

Les attributs de fichiers

- En plus du nom et des données, tous les systèmes d'exploitation associent des informations complémentaires aux fichiers.
- Ces informations constituent les **attributs** du fichier.
- Les attributs diffèrent d'un système d'exploitation à l'autre.

Les fichiers

Exemples d'attributs de fichiers

Attributs	**Signification**
Protection	Contrôle les modalités d'accès
Mot de passe	Mot de passe d'accès au fichier
Créateur	Créateur du fichier
Propriétaire	Propriétaire actuel du fichier
Date de création	Date et heure de création du fichier
Date du dernier accès	Date et heure du dernier accès au fichier
Date de modification	Date et heure de la dernière modification
Taille	Nombre d'octets du fichier
...	

L'appel système *stat*

Les attributs d'un fichier sous UNIX peuvent être consultés *via* les appels système *stat* :

```
int stat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

lstat permet de récupérer les informations d'un lien symbolique (et non pas celles du fichier visé).

L'appel système *stat*

La structure **struct** *stat* contient les champs suivants (issus du *man*) :

```
dev_t     st_dev;     // Périphérique
ino_t     st_ino;     // Numéro i-noeud
mode_t    st_mode;    // Type de fichier et protection
nlink_t   st_nlink;   // Nb liens matériels
uid_t     st_uid;     // UID propriétaire
gid_t     st_gid;     // GID propriétaire
dev_t     st_rdev;    // Type périphérique
off_t     st_size;    // Taille totale en octets
blksize_t st_blksize; // Taille de bloc pour E/S
blkcnt_t  st_blocks;  // Nombre de blocs alloués
time_t    st_atime;   // Heure dernier accès
time_t    st_mtime;   // Heure dernière modification
time_t    st_ctime;   // Heure dernier changement état
```

Voir l'exemple

Les fichiers sous UNIX

Les i-nœuds

- Un **i-nœud** est une structure associée à un fichier sur le disque.
- Les i-nœuds sont repérés par un identifiant unique, le numéro d'i-nœud.
- Le système maintient une table des i-nœuds permettant de récupérer les informations relatives aux fichiers ainsi que leur localisation sur le disque.

Modification des attributs

Pour modifier les autorisations d'accès à un fichier, on utilise les appels système *chmod* et *fchmod* :

```
#include <sys/stat.h>

int chmod(const char *path, mode_t mode);
int fchmod(int fd, mode_t mode);
```

- *path* est un chemin d'accès au fichier, il est déréférencé s'il s'agit d'un lien symbolique.
- Il faut être propriétaire du fichier (ou *root*) pour pouvoir changer les autorisations d'accès.
- Si le *GID* du fichier n'est aucun des groupes du processus appelant (et le processus n'est pas *root*), le bit Set-GID est effacé.
- Le bit Set-UID peut être effacé (dépend du système).

Les fichiers sous UNIX

Permissions

mode est un ou binaire des constantes suivantes :

- *S_IRUSR*, *S_IWUSR*, *S_IXUSR* et *S_IRWXU* servent à indiquer les autorisations pour le propriétaire du fichier
- *S_IRGRP*, *S_IWGRP*, *S_IXGRP* et *S_IRWXG* servent à indiquer les autorisations pour le groupe du fichier
- *S_IROTH*, *S_IWOTH*, *S_IXOTH* et *S_IRWXG* servent à indiquer les autorisations pour les autres utilisateurs
- *S_ISUID*, *S_ISGID* et *S_ISVTX* indiquent respectivement les bits Set-UID, Set-GID et Sticky bit.

Voir l'exemple

Modification des attributs

On peut modifier l'UID et le GID d'un fichier grâce aux appels :

```
#include <unistd.h>

int chown(const char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int lchown(const char *path, uid_t owner, gid_t group);
```

- *lchown* ne déréférence pas les liens symboliques.
- Seul *root* peut modifier le propriétaire d'un fichier.
- Un processus non privilégié ne peut changer le groupe d'un fichier que pour un groupe auquel il appartient.
- Les bits Set-UID et Set-GID sont effacés après un appel réussi à *chown* par un utilisateur non privilégié.

Les fichiers

Les opérations sur les fichiers

Voici les principaux appels système rencontrés dans les systèmes d'exploitation concernant les opérations sur les fichiers.

- **Création** Création d'un fichier vide, on peut éventuellement spécifier certains attributs.
- **Suppression** Suppression d'un fichier.
- **Ouvrir** Ouverture du fichier (permet son utilisation ultérieure), chargement de ses attributs et mise en cache d'un certain nombre de données.
- **Déplacer** Repositionne le pointeur de position courante du fichier.

Les fichiers

Les opérations sur les fichiers

- **Lire** Permet de lire un certain nombre d'octets du fichier et de les stocker en mémoire.
- **Écrire** Écrit des données dans le fichier.
- **Ajouter** Ne permet d'écrire des données qu'à la fin du fichier.
- **Renommer** Permet de modifier le nom d'un fichier.
- **Interroger** Permet de récupérer l'état d'un fichier.
- Différents appels système peuvent également exister pour modifier les attributs d'un fichier.

Les opérations sur les fichiers sous UNIX

Descripteurs de fichiers

- Sous UNIX, la manipulation des fichiers se fait à l'aide d'un **descripteur de fichier**.
- Les descripteurs de fichiers sont des entiers compris entre 0 et une valeur au moins égale à 20 (1024 sous Linux).
- Les constantes symboliques *STDIN_FILENO*, *STDOUT_FILENO* et *STDERR_FILENO* désignent respectivement l'entrée standard, la sortie standard et la sortie des erreurs.
- Les descripteurs permettent également de gérer d'autres objets que les fichiers (tubes, socket, ...).
- Chaque processus dispose de sa table de descripteurs.

Les opérations sur les fichiers sous UNIX

Ouverture de fichiers

Afin d'obtenir un descripteur de fichier, on utilise l'appel système *open* :

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

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

- *pathname* correspond au chemin du fichier à ouvrir.
- *flags* contient le mode d'ouverture ainsi que des attributs de création et/ou d'état.
- *mode* est utilisé lors de la création d'un fichier pour initialiser les autorisation d'accès.

open retournera toujours le plus petit descripteur de fichier non utilisé (ou -1 en cas d'erreur).

Les opérations sur les fichiers sous UNIX

Mode d'ouverture et attributs de création

Le paramètre *flags* contient l'un des modes d'ouverture suivants :

- *O_RDONLY* ouverture du fichier en lecture seule ;
- *O_WRONLY* ouverture du fichier en écriture seule ;
- *O_RDWR* ouverture du fichier en lecture-écriture.

D'autres attributs peuvent être spécifiés à l'aide d'un ou binaire. Par exemple, on peut utiliser les attributs de création :

- *O_CREAT* pour créer un fichier.
- *O_EXCL* à utiliser avec *O_CREAT* pour que l'ouverture échoue si le fichier existe déjà.
- *O_TRUNC* à utiliser avec *O_WRONLY* ou *O_RDWR* : si le fichier existe déjà, sa taille est ramenée à 0.

Parmi les attributs d'état, on trouve *O_APPEND* qui spécifie que les écritures auront lieu automatiquement à la fin de fichier.

Les opérations sur les fichiers sous UNIX

Mode d'ouverture et attributs de création

Que produira le mode suivant ?

`O_RDWR | O_CREAT | O_EXCL`

Voyez-vous dans quel cas nous pourrions utiliser le mode suivant ?

`O_WRONLY | O_CREAT | O_APPEND`

Les opérations sur les fichiers sous UNIX

Permissions

Lors d'une création de fichier (avec *O_CREAT*), on spécifie également les autorisations d'accès au fichier à l'aide du paramètre *mode* qui est un ou binaire des constantes vues précédemment avec *chmod*.
Voir l'exemple

Les opérations sur les fichiers sous UNIX

Création d'un fichier

- L'utilisateur effectif du processus est utilisé pour fixer le propriétaire d'un nouveau fichier.
- Attention, deux statégies peuvent être utilisées pour fixer le groupe de celui-ci :
  - certaines configurations utilisent le groupe effectif du processus ;
  - d'autres utilisent le groupe du répertoire contenant le fichier.
- Le choix de la statégie peut dépendre du système de fichier, des options de montage, *etc.*
- Les permissions sont filtrées par le `umask` du processus : `mode & ~umask`.

Les opérations sur les fichiers sous UNIX

Remarque

Il existe également un appel système *creat*(*pathname*, *mode*) équivalent à *open*(*pathname*, *O_CREAT* | *O_WRONLY* | *O_TRUNC*, *mode*).

Les opérations sur les fichiers sous UNIX

Masque de création de fichier

On peut modifier le masque de création de fichier avec l'appel système *umask* :

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

mode_t umask(mode_t mask);
```

- Le masque est hérité lors d'un *fork* et conservé au cours d'un *execve*.
- *umask* retourne la valeur précédente du masque.

Les opérations sur les fichiers sous UNIX

Question

Comment récupérer l'*umask* courant ?

Les opérations sur les fichiers sous UNIX

Fermeture d'un fichier

Lorsqu'on a fini d'utiliser un fichier, on utilise l'appel système *close* :

```
#include <unistd.h>

int close(int fd);
```

1. Après *close*, *fd* peut être réutilisé pour un autre fichier.
2. Si *fd* est le dernier descripteur référençant le fichier alors les ressources associées au fichier sont libérées de la mémoire.
3. Si *fd* était le dernier descripteur référençant un fichier en attente de suppression (*cf.* *unlink*), alors le fichier est effectivement supprimé.

Les opérations sur les fichiers sous UNIX

Remarque

- L'appel système `close` est la dernière chance de détecter si une erreur se produit durant une écriture différée dans le fichier.
- Il faut donc toujours vérifier si une erreur se produit afin de pouvoir avertir l'utilisateur que son fichier est peut-être corrompu.

Les opérations sur les fichiers sous UNIX

Lecture dans un fichier

Nous avons déjà vu qu'on utilise l'appel système *read* pour lire dans un fichier :

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

- *buf* est l'adresse du tampon dans lequel stocker les données lues.
- *count* est le nombre d'octets à lire.
- La fonction retourne le nombre d'octets effectivement lus ou -1 en cas d'erreur (*ssize_t* est un type signé).
- La position courante dans le fichier est avancée du nombre d'octets lus.

Voir l'exemple

Les opérations sur les fichiers sous UNIX

Lecture dans un fichier

Il existe également deux autres appels système (que nous ne détaillerons pas) permettant de lire dans un fichier :

```
#include <unistd.h>

ssize_t pread(int fd, void *buf, size_t count,
              off_t offset);

#include <sys/uio.h>

ssize_t readv(int fd,
              const struct iovec *iov, int iovcnt);
```

Les opérations sur les fichiers sous UNIX

Écriture dans un fichier

Pour écrire dans un fichier, nous pourrons utiliser l'appel système *write* :

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

- Si le descripteur autorise le positionnement, l'écriture prend place à la position courante.
- Si le fichier ouvert possède l'attribut *O_APPEND*, la position courante est déplacée à la fin du fichier avant l'écriture.
- Après l'écriture, la position courante est déplacée du nombre d'octets écrits.

Valeurs de retour de `read` et `write`

Que s'est-il passé ?

Cinq cas de retour possibles lors d'un `read` ou d'un `write` :

- la valeur de retour correspond au nombre d'octets demandés (paramètre `count`) : tout s'est correctement déroulé ;
- la valeur de retour est plus petite que `count` mais supérieure à 0 : la fonction a traité moins d'octets que demandé, soit la fin de fichier a été rencontrée (cas de `read`), soit il n'y a plus d'espace sur le disque (cas de `write`), soit un signal a interrompu la fonction ;
- la valeur de retour vaut 0 : aucun octet n'a été traité, la fonction `read` a atteint la fin de fichier ;
- la valeur de retour vaut `-1` et `errno` vaut `EINTR` : la fonction a été interrompue par un signal ;
- la valeur de retour vaut `-1` et `errno` est différente de `EINTR` : une erreur s'est produite ;

Les opérations sur les fichiers sous UNIX

Écriture dans un fichier

De même que pour *read*, il existe deux autres appels système permettant l'écriture dans un fichier :

```
#include <unistd.h>

ssize_t pwrite(int fd, const void *buf, size_t count,
               off_t offset);

#include <sys/uio.h>

ssize_t writev(int fd, const struct iovec *iov,
               int iovcnt);
```

Les opérations sur les fichiers sous UNIX

Synchronisation

On peut demander au système de synchroniser les données mises en cache avec celles du disque.

```
#include <unistd.h>

void sync(void);
int fsync(int fd);
```

- D'après POSIX, cette fonction ne fait que planifier la transmission des données aux controleurs de disque. Sous Linux, l'appel système est bloqué jusqu'à ce que la synchronisation soit effectuée.
- Les disques modernes ayant également de grands caches, rien n'assure que les données seront effectivement écrites sur le disque.

Les opérations sur les fichiers sous UNIX

Positionnement

On peut modifier la position courante dans un fichier grâce à l'appel système *lseek* :

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

off_t lseek(int fd, off_t offset, int whence);
```

whence peut valoir les valeurs suivantes :

- *SEEK_SET* : la position courante est fixée à *offset* octets.
- *SEEK_CUR* : la position courante est incrémentée de *offset* octets.
- *SEEK_END* : la position courante est incrémentée de *offset* octets après la fin du fichier.

Les opérations sur les fichiers sous UNIX

Question

Il n'y a pas d'appel système `ltell`.
Comment récupérer la position courante d'un fichier ouvert ?

Les tables liées aux fichiers

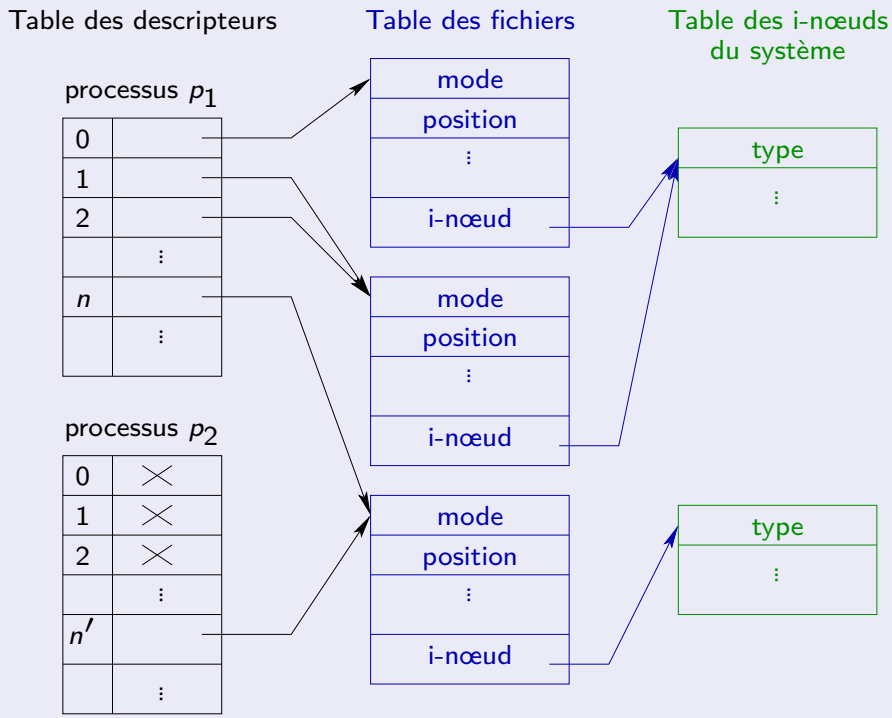
Tables de fichiers

Que fait le programme suivant : [source] ?
Pourquoi ?

Les tables liées aux fichiers

Tables de fichiers

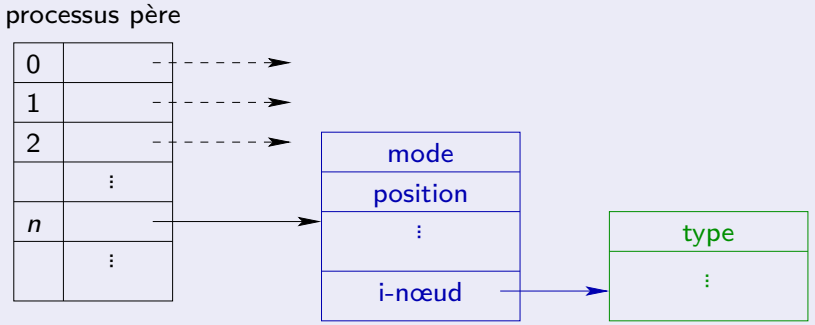
Afin de gérer les différents fichiers ouverts, le système d'exploitation utilise trois tables.

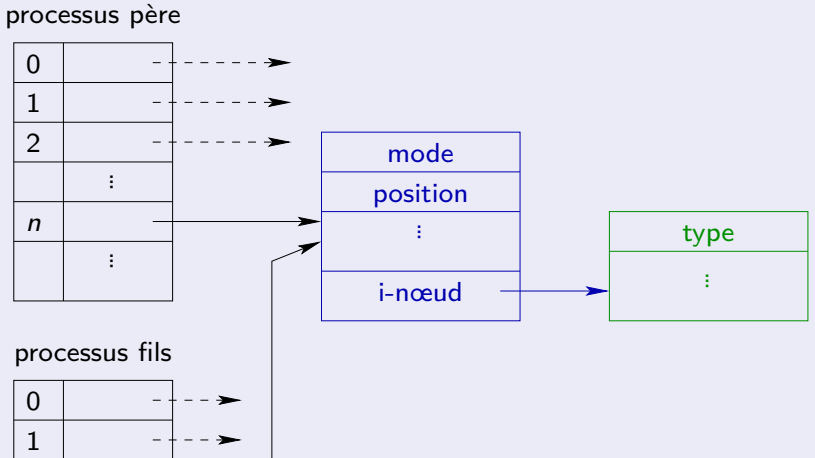


Les tables liées aux fichiers

Tables de fichiers : duplication de processus

Que se passe-t-il lorsqu'on duplique un processus ?





Les tables liées aux fichiers

Tables de fichiers : duplication de processus

Dans l'exemple précédent, le père attend que son fils se termine avant d'écrire dans le fichier. Si cela n'avait pas été le cas :

- dans un système mono-processeur, comme l'appel système `write` ne peut pas être préempté, les écritures se réalisent séquentiellement ;
- s'il y a plusieurs processeurs (ou plusieurs cœurs), deux processeurs peuvent appeler `write` en même temps : POSIX n'impose pas la non concurrence du write, il peut donc y avoir écrasement des données d'un processeur par l'autre.

Les opérations sur les fichiers sous UNIX

Duplication de descripteur

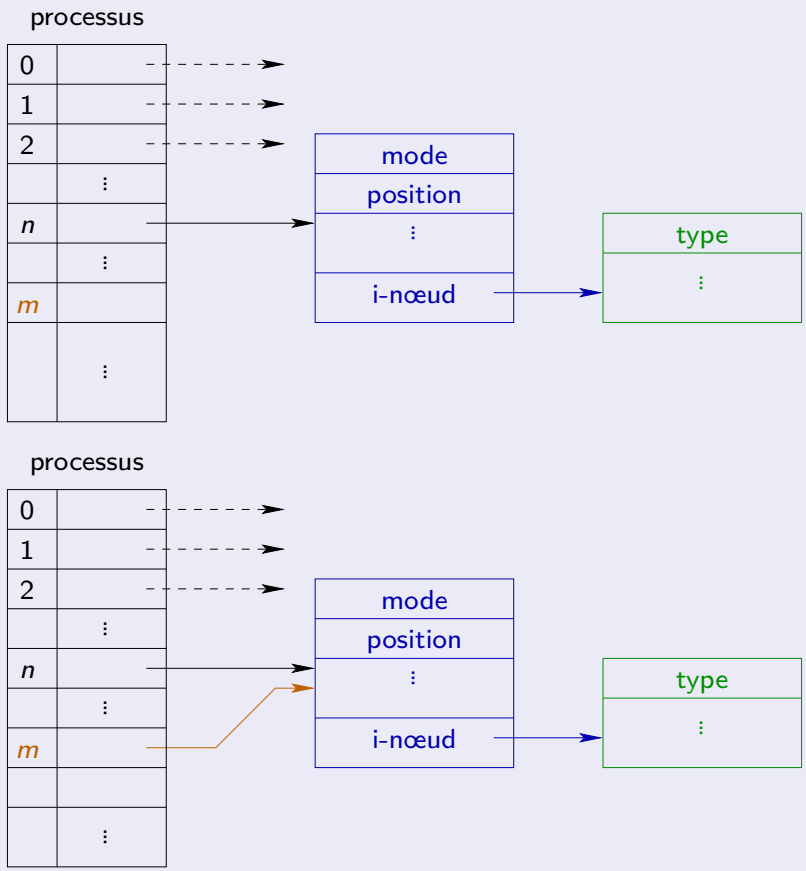
On peut dupliquer un descripteur de fichier grâce aux appels système *dup* et *dup2* :

```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

- *dup* retourne le plus petit descripteur inutilisé.
- *dup2* crée une copie de *oldfd* dans *newfd*.
- Si besoin, *newfd* est fermé avant d'être réaffecté.
- Que fait l'exemple suivant [source] ?
- Le même exemple avec *dup2* : [exemple] (à préférer afin d'être sûr que *STDOUT* n'est pas utilisé par un autre thread entre *close* et *dup*).

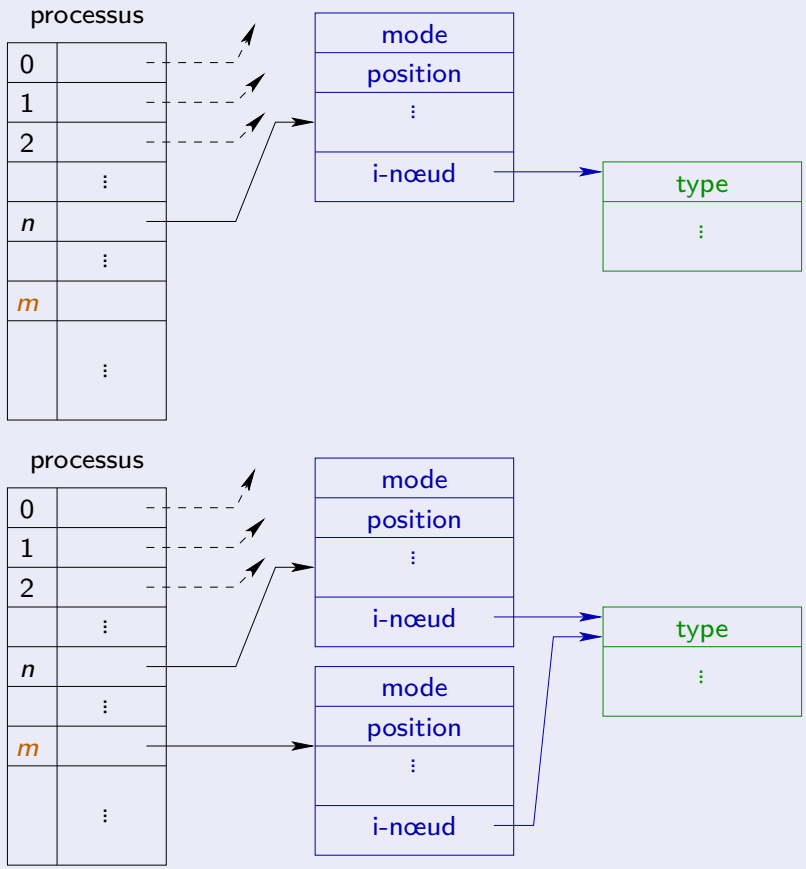
Les tables liées aux fichiers

Tables de fichiers : duplication de descripteur



Les tables liées aux fichiers

Tables de fichiers : ouverture d'un fichier



Plan

4 Les fichiers
- Introduction
- Fichiers
- Les répertoires

Les répertoires

Rappels

- Dans la plupart des systèmes de fichiers, les fichiers sont rangés dans des **répertoires** (ou dossiers).
- Généralement, l'imbrication des répertoires forme une **arborescence**.
- La racine de l'arbre ainsi décrit est appelée **répertoire racine**.
- Sous UNIX, le répertoire racine est désigné par le symbole « / »

Les répertoires

Rappels

- Un **chemin absolu** est le chemin complet permettant d'atteindre un fichier à partir du répertoire racine.
  - Sous UNIX, il sera de la forme $/rep_1/rep_2/.../rep_n/fichier$
- Un **chemin relatif** est le chemin permettant d'atteindre un fichier à partir du **répertoire courant** (ou répertoire de travail).
  - Sous UNIX, il sera de la forme $rep_1/rep_2/.../rep_n/fichier$.
- Sous UNIX, le repértoire courant et le répertoire parent sont indiqués par les symboles « . » et « .. ».

Les répertoires

Implémentation

Sous Unix, le répertoire peut être appréhendé comme étant une liste de couples (nom de fichier, i-nœud).

Les opérations sur les répertoires

Les appels système pour les répertoires varient beaucoup d'un système à l'autre. Nous présenterons ici quelques possibilités offertes par UNIX.

Les répertoires

Changement de répertoire de travail

Pour modifier le répertoire de travail d'un processus, on peut utiliser les appels système suivants :

```
#include <unistd.h>

int chdir(const char *path);
int fchdir(int fd);
```

Les répertoires

Répertoire de travail

On peut utiliser la fonction *getcwd* pour retrouver le répertoire de travail :

```
char *getcwd(char *buf, size_t size);
```

Si le tampon *buf* de taille *size* est trop petit pour contenir le chemin absolu du répertoire de travail alors la fonction retourne *NULL* et *errno* est fixée à *ERANGE*.

Les répertoires

Lecture du contenu d'un répertoire

Pour parcourir un répertoire, nous utiliserons les fonctions suivantes :

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *nom);
DIR *fdopendir(int fd);
int closedir(DIR *dirp);

struct dirent *readdir(DIR *dirp);
```

- *opendir* et *fdopendir* retournent un pointeur sur un flux répertoire qui pourra être ensuite utilisé par *readdir*.
- *readdir* retourne un pointeur sur l'entrée suivante du flux répertoire *dirp*.

Les répertoires

Lecture du contenu d'un répertoire

Les seuls champs définis par POSIX pour la stucture *dirent* sont :

```
ino_t          d_ino;     /* numéro de l'inode */
char           d_name[]; /* nom du fichier */
```

Voir l'exemple

Les répertoires

Lecture du contenu d'un répertoire

Il existe d'autres fonctions permettant de manipuler les répertoires :

```
#include <sys/types.h>
#include <dirent.h>

void rewinddir(DIR *dirp);
long int telldir(DIR *dirp);
int scandir(const char *dirp, struct dirent ***namelist,
    int (*filter)(const struct dirent *),
    int (*compar)(const struct dirent **,
                  const struct dirent **));
```

Les répertoires

Création d'un répertoire

On crée un répertoire à l'aide de l'appel système *mkdir* :

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

int mkdir(const char *pathname, mode_t mode);
```

- Le répertoire aura pour propriétaire l'UID effectif du processus.
- Pour le groupe cela dépend du bit Set-GID du parent ainsi que des options de montage (*cf.* le *man* pour plus de détails).

Les répertoires

Déplacer/renommer un fichier

L'appel système *rename* permet de renommer et de déplacer un fichier :

```
#include <stdio.h>

int rename(const char *oldpath, const char *newpath);
```

- Si *newpath* existe, il sera alors écrasé (si on en a le droit).
- Si *oldpath* est un répertoire alors *newpath*, s'il existe, doit être un répertoire vide.

Les répertoires

Création de liens

Sous UNIX, un fichier peut avoir plusieurs noms et se trouver dans différents répertoires. Pour créer un nouveau nom (avec éventuellement un nouvel emplacement) pour un fichier on utilise l'appel système *link* :

```
#include <unistd.h>

int link(const char *path, const char *newpath);
```

- Les différents noms associés à un même fichier sont appelés **liens physiques**.
- Sous UNIX, l'imbrication des répertoires forme donc un graphe et non pas un arbre.

Les répertoires

Suppression de liens

Pour supprimer un lien, on utilise :

```
#include <unistd.h>

int unlink(const char *pathname);
```

- Si le lien détruit était le dernier lien sur le fichier alors le fichier sera effacé.
- Un fichier n'est réellement effacé qu'après la fermeture du dernier descripteur de fichier le référençant.

Les répertoires

Suppression de répertoire

Linux ne permet pas la suppression des répertoires à l'aide de *unlink*. On utilise *rmdir* à la place :

```
#include <unistd.h>

int rmdir(const char *pathname);
```

On ne peut supprimer que des répertoires vides.

Les répertoires

Les liens symboliques

- Les liens symboliques sont un type de fichier particulier.
- Un lien symbolique est un fichier contenant le chemin d'accès du fichier vers lequel il pointe.
- Pour créer un lien symbolique on utilise l'appel système *symlink* :

```
#include <unistd.h>
int symlink(const char *path, const char *newpath);
```

- Pour lire le contenu d'un lien symbolique, on utilisera l'appel système *readlink* :

```
#include <unistd.h>
ssize_t readlink(const char *path,
                 char *buf, size_t bufsiz);
```

Voir l'exemple

Les répertoires

Les liens

L'utilisation des liens peut poser des problèmes.

- Pour les liens physiques :
  - Si un utilisateur A crée un fichier et qu'un utilisateur B crée un lien sur ce dernier.
  - A peut décider de supprimer son fichier mais il n'est pas forcément au courant que B a un lien sur celui-ci.
  - B se retrouve avec un fichier appartenant à A qu'il ne peut éventuellement pas supprimer.
  - De plus le fichier est toujours comptabilisé sur le quota de A.
- Les liens symboliques peuvent être couteux en temps lors de l'accès à un fichier.
- Les liens posent également des problèmes lors de la sauvegarde des systèmes de fichiers car ils peuvent amener à une sauvegarde multiple du même fichier.

Plan

1. Introduction
2. Processus
3. Les threads
4. Les fichiers
5. Communication et synchronisation
6. Les signaux

Plan

5 Communication et synchronisation
- La communication interprocessus
- Les tubes
- Synchronisation de tâches
- La mémoire partagée
- Les sémaphores
- La synchronisation des threads

Introduction

La communication interprocessus

- Il arrive souvent que les processus aient besoin de communiquer entre eux.
- Cette communication entre les processus est appelée communication interprocessus ou **IPC** (*inter-process communication*).
- Par extension, les IPC désigneront aussi les outils permettant la communication entre les processus.

Introduction

La communication interprocessus

Il existe essentiellement trois problèmes liés à la communication entre les processus :

1. Comment passer des informations d'un processus à un autre ?
2. Comment éviter les conflits lors des accès aux informations partagées ?
3. Comment ordonner les différentes actions réalisées par des processus dépendants les uns des autres ?

Introduction

La communication interprocessus

Dans ce cours nous présenterons les IPC suivantes :

- Les signaux (dans le chapitre suivant).
- Les tubes et les tubes nommés.
- Les segments de mémoires partagés.

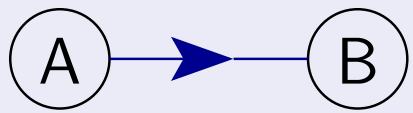
De plus, les sémaphores qui sont des outils de synchronisation sont parfois classés parmi les IPC.

Plan

5 Communication et synchronisation
- La communication interprocessus
- Les tubes
- Synchronisation de tâches
- La mémoire partagée
- Les sémaphores
- La synchronisation des threads

Les tubes

- Lorsque l'échange de messages se fait entre deux processus et toujours dans le même sens, on peut utiliser un tube.
- Un **tube** (ou *pipe*) est un canal unidirectionnel de données.
- Un tube peut être vu comme un pseudo-fichier, la lecture et l'écriture dans un tube pourra donc se faire (presque) comme dans un fichier.
- Le canal est unidirectionnel : on écrit à une extrémité du tube et on lit à l'autre extrémité.



- Un tube est un flux d'octets : il n'y a pas de frontières entre les messages.

Les tubes

Création

- Il existe deux types de tubes : les **tubes nommés** et les **tubes anonymes**.
- Lorsque le tube ne sera utilisé que par le processus ou ses fils, il pourra être anonyme.
- L'appel système *pipe* crée un tube anonyme :

```
#include <unistd.h>

int pipe(int pipefd[2]);
```

- En sortie, *pipefd*[0] contient la sortie du tube (descripteur ouvert en lecture) et *pipefd*[1] contient l'entrée (descripteur ouvert en écriture).

Les tubes

Exemple

Voici un exemple où un processus fils écrit des données sur le tube et le père lit ces données pour les afficher : voir l'exemple.

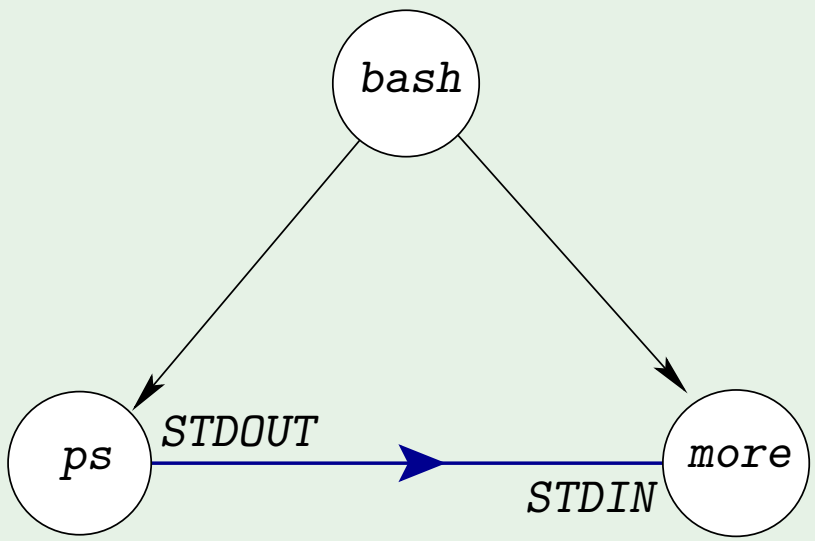
Les tubes

Exemple

Le shell utilise également des tubes. Par exemple la commande

```
# ps aux | more
```

produit l'arbre des processus suivant :



Les tubes

Utilisation des descripteurs associés à un tube

- Lorsqu'il n'y a plus de processus possédant un descripteur de tube ouvert en écriture, un appel à `read` sur la sortie du tube renvoie 0 (fin de fichier).
- Afin d'éviter qu'un processus reste indéfiniment bloqué sur un `read`, un lecteur doit fermer les descripteurs ouverts en écriture qu'il possède.

Les tubes

Taille du tampon

- Un tampon est associé à chaque tube.
- Les processus essayant d'écrire dans un tampon plein sont bloqués.
- La taille du tampon étant variable, les applications ne devraient pas dépendre de celle-ci : il devrait toujours y avoir un lecteur prêt à recevoir les données envoyées par les écrivains.
- POSIX impose que les écritures de moins de `PIPE_BUF` octets doivent être atomiques. Au-delà les données peuvent être entrelacées avec d'autres données (exemple).

Utilisation des descripteurs associés à un tube (suite)

- Lorsqu'il n'y a plus de descripteur de tube ouvert en lecture, un appel à `write` provoque l'émission d'un signal `SIGPIPE` dans le processus appelant (le traitement des signaux sera vu au chapitre suivant).
  Voir l'exemple
- Afin d'éviter qu'un processus écrive dans un tube sur lequel il n'y a plus de lecteur, un écrivain doit fermer les descripteurs ouverts en lecture qu'il possède.

Les tubes nommés

Création de tubes nommés

- Afin de permettre la communication par tube entre deux processus indépendants, on peut associer un nom dans le système de fichier à un tube : il s'agit alors d'un tube nommé.
- Pour créer un tube nommé, on utilise la fonction *mkfifo* :

```
int mkfifo(const char *pathname, mode_t mode);
```

- Pour supprimer un tube nommé, on utilise l'appel système *unlink*.

Les tubes nommés

Utilisation de tubes nommés

- Pour utiliser un tube nommé il suffit de l'ouvrir en lecture ou en écriture.
- Il faut ouvrir les deux extrémités pour pouvoir utiliser le tube : la première ouverture en lecture ou une écriture sera bloquante jusqu'à ce que la seconde ouverture soit réalisée.

Les tubes

Exemple

Voici l'exemple précédent adapté à un tube nommé : voir l'exemple.

Plan

5 Communication et synchronisation
- La communication interprocessus
- Les tubes
- Synchronisation de tâches
- La mémoire partagée
- Les sémaphores
- La synchronisation des threads

Synchronisation de la communication

Introduction

- Certaines IPC intégrent dans leur spécification la synchronisation des processus.
- Par exemple, lorsqu'on utilise un tube, il ne doit y avoir qu'un lecteur et qu'un écrivain. La synchronisation des écritures et des lectures dans le tube est alors réalisée au niveau du système.
- D'autres IPC délèguent à l'utilisateur la synchronisation des accès à l'IPC. C'est le cas, par exemple, de la mémoire partagée.

Situation de concurrence

Définition

- Une **situation de concurrence** (*race condition*) est un système qui fournit un résultat différent selon l'ordre d'exécution des acteurs (généralement des processus ou des threads).
- Ce type de situation se produit fréquemment lorsque des processus partagent des données, que ce soit en mémoire partagée ou dans des fichiers.
- Les fonctions suceptibles d'être exécutées simultanément par plusieurs threads sont également très propices à ce type de situation.
- Bien entendu, lorsque nous développerons des algorithmes, nous chercherons à éviter les situations de concurrence.

Section critique

Définition

- Un moyen d'éviter les situations de concurrence est d'empêcher que deux tâches (processus ou thread) utilisent simultanément la même ressource partagée (variable ou fichier) : il s'agit de **l'exclusion mutuelle**.
- La partie d'un programme faisant intervenir une ressource partagée est appelée **section critique**.
- Si on peut empêcher que deux processus se trouvent simultanément dans leurs sections critiques, on est sûr de réaliser l'exclusion mutuelle.

Section critique

Synchronisation

Réaliser l'exclusion mutuelle ne suffit pas. Afin que les tâches coopèrent correctement, les quatre conditions suivantes doivent être satisfaites :

1. Deux tâches ne doivent pas être en même temps dans leur section critique (exclusion mutuelle).
2. Il ne faut pas faire de supposition quant à la vitesse ou au nombre de processeurs.
3. Aucune tâche s'exécutant à l'extérieur de sa section critique ne doit bloquer les autres (interblocage).
4. Aucune tâche ne doit attendre indéfiniment pour pouvoir entrer dans sa section critique (famine).

L'exclusion mutuelle

Désactivation des interruptions

- Un premier moyen pour un processus de réaliser l'exclusion mutuelle est de désactiver les interruptions.
- Cette méthode ne fonctionne que pour des systèmes monoprocesseur.
- Il est de plus très dangereux de permettre aux utilisateurs de désactiver les interruptions.
- Cette méthode était utilisée à l'intérieur des systèmes d'exploitation sur les machines monocœurs.

L'exclusion mutuelle

Les verrous

- Une autre méthode consiste à utiliser des verrous.
- Un verrou est une variable partagée par les processus.
- Le verrou peut prendre les valeurs 0 ou 1.
- Un 1 indique qu'un processus est entré ou entre en section critique.
- Un 0 indique qu'aucun processus ne souhaite pour le moment entrer en section critique.

L'exclusion mutuelle

Une (pas si ?) mauvaise solution

Selon le principe du verrou, on pourrait écrire le code suivant :

```
// On suppose que verrou est une variable partagée
volatile int verrou = 0;

void entrerSectionCritique() {
  while (verrou == 1) {
    // On attend
  }
  verrou = 1;
  // On peut maintenant entrer en section critique
}

void sortirSectionCritique() {
  verrou = 0;
  // Un autre processus peut maintenant entrer
  // dans sa section critique
}
```

L'exclusion mutuelle

Une mauvaise solution

```
// Processus A                          | // Processus B

// Le verrou est à 0
// L'odonnanceur donne la main à A

while (verrou == 1) ;                   |

// L'ordonnanceur passe la main

                                        | while (verrou == 1) ;
                                        | verrou = 1;
verrou = 1;
// Section critique                     | // Section critique

// Les deux processus sont maintenant en même temps
// dans leur section critique.
```

L'exclusion mutuelle

Les verrous : une solution ?

- La solution proposée ne fonctionne pas car le test sur le verrou et l'affectation à 1 ne constituent pas une opération atomique.
- Certains processeurs proposent une instruction TSL (*Test and Set Lock*) qui permet en une seule opération de
  - fixer le verrou (un mot mémoire) à 1 ;
  - retourner (dans un registre) l'ancienne valeur du verrou.
- Les fonctions précédentes peuvent alors s'écrire :

```
volatile int verrou = 0;

void entrerSectionCritique() {
   while (TSL(verrou) == 1) ;
 }

void sortirSectionCritique() {
  verrou = 0;
}
```

L'exclusion mutuelle

Les verrous : une solution ?

- Les processeurs x86 proposent d'autres instructions que TSL.
- On peut, par exemple, utiliser l'instruction XCHG qui effectue un échange des valeurs contenues dans deux emplacements :

```
volatile int verrou = 0;

void entrerSectionCritique() {
  int x;
  do {
    x = 1;
    XCHG(verrou, x);
  } while (x == 1) ;
}

void sortirSectionCritique() {
  verrou = 0;
}
```

L'exclusion mutuelle

L'alternance stricte

- Parfois, on peut vouloir que les processus entrent dans leur section critique à tour de rôle.
- On peut alors utiliser la méthode de l'alternance stricte.
- On mémorise à qui c'est le tour d'entrer en section critique à l'aide d'une variable partagée *turn*.
- Voici une solution pour deux processus :

```
// Processus A
while (1) {
  while (turn != 0) ;
  // Section critique
  turn = 1;
}
```

```
// Processus B
while (1) {
  while (turn != 1) ;
  // Section critique
  turn = 0;
}
```

L'exclusion mutuelle

Exclusion mutuelle et attente active

- Il existe de nombreux algorithmes permettant de réaliser l'exclusion mutuelle de processus de façon logicielle (sans avoir besoin d'instruction atomique évoluée) et sans alternance stricte.
- Par exemple, l'algorithme de Peterson permet de réaliser l'exclusion mutuelle pour deux processus.
- L'algorithme de la boulangerie permet quant-à lui de réaliser l'exclusion mutuelle pour un nombre quelconque de processus.

Plan

5 Communication et synchronisation
- La communication interprocessus
- Les tubes
- Synchronisation de tâches
- La mémoire partagée
- Les sémaphores
- La synchronisation des threads

Introduction

Les variables partagées

- Nous avons jusqu'à présent parlé de variables partagées sans expliquer comment partager ces variables entre des processus.
- Des données partagées peuvent, bien sûr, être stockées dans des fichiers mais cela n'est *a priori* pas très efficace.
- Une autre solution consiste à utiliser un segment de mémoire qui sera partagée entre différents processus.

Introduction

Les variables partagées

- Les variables qui sont définies dans un segment de mémoire partagée peuvent être utilisées par les différents processus utilisant la mémoire partagée.
- Ce procédé ressemble à l'utilisation que font les threads des variables globales ou allouées sur le tas pour communiquer entre eux.
- Les segments de mémoire partagée sont dupliqués lors de la duplication de processus : un fils a donc accès à la même mémoire partagée que son père.

Introduction

Les objets de mémoire partagée

L'obtention d'un segment de mémoire partagée s'effectue en deux ou trois étapes :

1. Ouverture (et éventuellement création) de l'objet de mémoire partagée.
2. En cas de création, il faut ensuite définir la taille de la mémoire partagée.
3. Projection de la mémoire partagée dans l'espace d'adressage virtuel du processus.

Remarque

Sous Linux, les programmes utilisant les objets de mémoire partagée doivent être liés avec la bibliothèque `rt`.

Utilisation des objets de mémoire partagée

Création et ouverture d'un objet de mémoire partagée

Pour créer et ouvrir un objet de mémoire partagée on utilise la fonction *shm_open* :

```
#include <sys/mman.h>
int shm_open(const char *nom, int oflag, mode_t mode);
```

- *nom* doit être de la forme « */un_nom* ». *un_nom* ne doit pas contenir de caractère « / ».
- *oflag* contient *O_RDONLY* ou *O_RDWR* et des attributs *O_CREAT*, *O_EXCL* ou *O_TRUNC*.
- *mode* est utilisé pour définir les autorisations d'accès si l'objet est créé. Le propriétaire et le groupe étant ceux du processus.
- *shm_open* retourne un descripteur de fichier associé à l'objet.

Utilisation des objets de mémoire partagée

Création et ouverture d'un objet de mémoire partagée

Sous Linux, l'objet de mémoire partagée est créé dans un système de fichiers virtuel généralement monté sur `/dev/shm`.

Utilisation des objets de mémoire partagée

Définir la taille d'un objet de mémoire partagée

- Comme pour n'importe quel fichier, lorsque le système crée un objet de mémoire partagée, sa taille est nulle.
- Pour fixer la taille d'un objet de mémoire partagée on utilisera l'appel système *ftruncate* :

```
#include <unistd.h>
int ftruncate(int fd, off_t length);
```

- *ftruncate* fonctionne également avec les fichiers ordinaires :
  - Si la taille du fichier était plus grande que *length* alors le fichier est tronqué (et les données surnuméraires sont perdues).
  - Si la taille du fichier est plus petite que *length* alors des 0 sont ajoutés à la fin du fichier afin de lui donner la taille voulue.

Voir l'exemple

Utilisation des objets de mémoire partagée

Projeter un objet de mémoire partagée

Pour projeter l'objet de mémoire partagée dans l'espace d'adressage virtuel du processus, nous utiliserons l'appel système *mmap* :

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length,
           int prot, int flags,
           int fd, off_t offset);
```

- Projette *length* octets du fichier *fd* à partir de la position *offset* à l'adresse *addr*.
- Si *addr* est *NULL* le système choisit une adresse valide.
- *prot* indique la protection de la mémoire qui sera projetée à l'aide d'un *OU* binaire : *PROT_EXEC*, *PROT_READ* ou *PROT_WRITE*.
- *flags* : nous utiliserons *MAP_SHARED*.

Voir l'exemple

Libération des ressources

Libérer un objet de mémoire partagée

- Le descripteur d'un objet de mémoire partagée est fermé à l'aide de l'appel système *close* :

```
#include <unistd.h>
int close(int fd);
```

- La projection mémoire peut être libérée à l'aide de l'appel système *munmap* :

```
int munmap(void *addr, size_t length);
```

- À la terminaison d'un processus le descripteur est fermé et la projection est libérée.

Libération des ressources

Libérer un objet de mémoire partagée

- Les objets de mémoire partagée non supprimés sont persistants jusqu'à l'extinction de la machine.
- Pour supprimer un objet de mémoire partagée, on utilisera l'appel système *shm_unlink* :

```
int shm_unlink(const char *name);
```

1. Libère le nom utilisé par l'objet de mémoire partagée (un nouvel objet avec le même nom peut alors être créé).
2. L'objet de mémoire partagée n'est effectivement supprimé que lorsque tous les descripteurs le référençant sont supprimés.
3. La mémoire partagée n'est réellement libérée du système qu'une fois que tous les processus ont retiré leur projection.

Utilisation de la mémoire partagée

Exemple

Voici un exemple où nous utilisons la mémoire partagée pour stocker une variable qui permet de fixer un rendez-vous entre le père et le fils : voir l'exemple.

Plan

5 Communication et synchronisation
- La communication interprocessus
- Les tubes
- Synchronisation de tâches
- La mémoire partagée
- Les sémaphores
- La synchronisation des threads

L'attente passive

- Les algorithmes précédents ont le gros défaut d'utiliser l'**attente active**.
- Un processus qui attend de pouvoir entrer en section critique devrait pouvoir être retiré de la liste des processus éligibles et n'y être replacé qu'une fois que son attente est terminée : cela correspond à l'état bloqué.
- Les processus bloqués sont parfois appelés processus en sommeil en raison des premiers appels systèmes qui permettaient de bloquer et de débloquer les processus : *sleep* et *wakeup*.

Introduction

Les sémaphores

- En 1965, Dijkstra propose d'utiliser un nouveau type de variable entière sur laquelle deux opérations sont autorisées : P et V.
- P et V proviennent du néerlandais tester (*Proberen*) et incrémenter (*Verhogen*).
- Attention, POSIX nomme ces opérations *wait* pour P et *post* pour V : il ne faut pas assimiler *post* à P.

Les opérations P et V

Les sémaphores

- Un sémaphore peut être vu comme un nombre de ressources libres.
- L'opération V augmente ce nombre de 1.
- Un P commence par tester si le sémaphore est nul :
  - Si le sémaphore est strictement positif alors il est décrémenté.
  - S'il n'y a pas de ressource disponible (le sémaphore est à 0) alors :
    - Le processus est placé en attente qu'une ressource se libère.
    - Le processus ne sera réveillé et le sémaphore décrémenté que lorsque le sémaphore sera strictement positif.
- Les opérations sont atomiques.

Les sémaphores

Le problème de l'exclusion mutuelle

- Un exemple très classique d'utilisation des sémaphores est celui de l'exclusion mutuelle.
- Pour protéger une section critique, on peut utiliser un sémaphore S initialisé à 1 puis utiliser le code suivant :

```
P(S)
Section critique...
V(S)
```

- Un sémaphore qui ne peut prendre que les valeurs 0 et 1 (« verrouillé », « déverrouillé ») est un **mutex** (pour *mutual exclusion*).

Les sémaphores

Le problème du producteur/consommateur

- Pour donner un autre exemple d'utilisation des sémaphores, nous allons nous intéresser au problème (toujours très classique) du producteur/consommateur.
- On suppose que deux processus partagent un tampon commun de taille fixe.
- L'un deux, le producteur, place des données dans le tampon.
- Le deuxième, le consommateur, les récupère.
- Les problèmes se posent lorsque le producteur veut placer des données dans un tampon plein ou si le consommateur veut récupérer les données d'un tampon vide.
- Dans les deux cas, il faut mettre en sommeil le processus qui ne peut effectuer son opération.

Les sémaphores

Le problème du producteur/consommateur

```
donnee buffer[N];
semaphore mutex = 1; // Contrôle l'accès à la SC
semaphore vide = N;  // Compte les emplacements vides
semaphore plein = 0; // Compte les emplacements occupés
```

```
void produire(donnee d) {
  P(vide);
  P(mutex);
  ajouter(buffer, d);
  V(mutex);
  V(plein);
}
```

```
donnee consommer() {
  P(plein);
  P(mutex);
  donnee d = retirer(buffer);
  V(mutex);
  V(vide);

  return d;
}
```

Les sémaphores

Utilisation des sémaphores

De nombreuses autres situations peuvent nécessiter l'utilisation de sémaphores. Par exemple :

- l'établissement de rendez-vous entre processus ;
- la gestion de lecteurs et de rédacteurs, il faut alors décider qui est prioritaire : le rédacteur ou le lecteur ;
- le problème de la voie unique.

Les sémaphores POSIX

Les fonctions sur les sémaphores

- Un sémaphore POSIX est une variable de type *sem_t*.
- Les deux opérations sur les sémaphores sont accessibles *via* les fonctions *sem_wait* et *sem_post* :

```
#include <semaphore.h>

int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
```

- L'édition des liens doit être effectuée avec la bibliothèque *pthread*.
- Il existe deux types de sémaphores POSIX : les **sémaphores nommés** et les **sémaphores anonymes**.

Les sémaphores POSIX

Les sémaphores nommés

- Pour accéder à un sémaphore nommé déjà existant, il faut utiliser l'appel système *sem_open* :

```
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag);
```

- La variable *name* contient le nom du sémaphore.
- Le nom d'un sémaphore est de la forme « */un_nom* ». *un_nom* ne peut contenir de caractère « / ».
- Sous Linux, la longueur maximale d'un nom de sémaphore est de 251 caractères (*NAME_MAX* - 4).
- *oflag* indique le mode d'ouverture du sémaphore comme pour un fichier.

Les sémaphores POSIX

Les sémaphores nommés

- Pour créer un nouveau sémaphore nommé, il faut utiliser la fonction *sem_open* avec quatre arguments :

```
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag,
                mode_t mode, unsigned int value);
```

- *O_CREAT* doit être spécifié dans *oflag* (avec éventuellement *O_EXCL*).
- *mode* sera utilisé pour fixer les droits d'accès.
- *value* indique la valeur que doit prendre le nouveau sémaphore.
- Sous Linux, les sémaphores sont créés sur le système de fichiers virtuel des objets de mémoires partagées avec des noms de la forme « *sem.un_nom* ».

Les sémaphores POSIX

Les sémaphores nommés

- Pour « fermer » un sémaphore nommé ouvert, on utilisera la fonction *sem_close* :

```
#include <semaphore.h>
int sem_close(sem_t *sem);
```

- Les sémaphores ouverts sont automatiquement fermés lors de la fin du processus ou dans le cas d'un *execve*.
- Un sémaphore nommé est persistant dans le noyau, il existe tant que le système n'est pas éteint. Pour supprimer un sémaphore nommé, on utilisera *sem_unlink* :

```
#include <semaphore.h>
int sem_unlink(const char *name);
```

Les sémaphores POSIX

Les sémaphores nommés

Un exemple (complètement inutile) sur l'utilisation des sémaphores POSIX nommés est disponible ici.

Les sémaphores POSIX

Les sémaphores anonymes

- Lorsque, comme dans le cas précédent, le sémaphore ne sera utilisé que par des threads ou des processus issus du processus qui a créé le sémaphore, on peut utiliser un sémaphore anonyme.
- Le sémaphore doit alors être déclaré dans une partie commune aux entités qui vont l'utiliser.
- Par exemple, on pourra déclarer une variable globale pour les threads ou une variable en mémoire partagée pour un sémaphore utilisé par plusieurs processus.

Les sémaphores POSIX

Les sémaphores anonymes

- Le sémaphore sera une variable du type *sem_t* :

```
sem_t sem;
```

- Avant de pouvoir utiliser le sémaphore il faut l'initialiser :

```
int sem_init(sem_t *sem, int pshared,
             unsigned int value);
```

- *pshared* indique si le sémaphore ne sera utilisé que par les threads du même processus (valeur `0`) ou par plusieurs processus (valeur `1`).
- *value* indique la valeur d'initialisation du sémaphore.

Les sémaphores POSIX

Les sémaphores anonymes

- Un sémaphore anonyme doit toujours être détruit :

```
int sem_destroy(sem_t *sem);
```

- Ne pas détruire un sémaphore anonyme peut provoquer des fuites de mémoire.
- Détruire un sémaphore sur lequel sont bloqués des processus ou des threads produira des résultats indéfinis.

Les sémaphores POSIX

Exemple de sémaphores anonymes

L'exemple précédent (toujours aussi futile) revisité avec des sémaphores anonymes est disponible ici.

Un producteur-consommateur

Un exemple beaucoup plus complet est disponible ici.

Pour aller plus loin

Implantation

Voici une implantation possible des sémaphores.

- Les sémaphores peuvent être implantés à l'aide d'une variable en mémoire partagée.
- La variable contient la valeur du sémaphore.
- Afin d'éviter la famine, les sémaphores utilisent généralement une file (de type FIFO) des processus et des threads bloqués à cause d'une opération P.
- Les opérations P et V sont rendues atomiques à l'aide d'instructions évoluées ou à l'aide d'un verrou (stocké également dans la variable du sémaphore) et de l'utilisation d'instructions du type TSL.
- Ici, l'utilisation de l'attente active dans les fonctions P et V n'est pas dommageable car les opérations à effectuer sont très rapides.

Plan

5 Communication et synchronisation
- La communication interprocessus
- Les tubes
- Synchronisation de tâches
- La mémoire partagée
- Les sémaphores
- La synchronisation des threads

Synchronisation des threads

Synchronisation des threads POSIX

L'API des threads POSIX propose en plus des sémaphores trois autres moyens de synchronisation des threads :

- Les mutex
- Les conditions
- Les verrous de lecture/écritures

Nous ne détaillerons pas ces objets dans ce cours (par manque de temps) mais vous pouvez consulter les pages du manuel si vous en avez besoin.

Synchronisation des threads

Exemple d'utilisation d'un mutex

Un exemple d'utilisation d'un mutex pour réaliser une exclusion mutuelle est disponible. Voir l'exemple.

Synchronisation des threads

Exemple d'utilisation d'une condition

Un exemple d'utilisation d'une condition est disponible. Voir l'exemple.

Synchronisation des threads

Exemple d'utilisation d'un verrou de lecture/écriture

Un exemple d'utilisation d'un verrou de lecture/écriture est disponible. Voir l'exemple.

Plan

1. Introduction
2. Processus
3. Les threads
4. Les fichiers
5. Communication et synchronisation
6. Les signaux

Plan

6 Les signaux
- Introduction
- L'API POSIX pour les signaux standards
- Programmer avec les signaux
- Particularités de certains signaux
- Les signaux temps-réel

Introduction

Communication asynchrone

- Jusqu'à présent tous les moyens de communication étudiés étaient synchrones : le processus décidait du moment où il traitait les messages reçus.
- Les signaux sont un moyen de communication asynchrone : lorsqu'un signal est reçu, l'exécution du processus est interrompue afin de traiter le signal.

Introduction

Présentation des signaux

- Les signaux peuvent être vus comme une interruption logicielle.
- Les processus peuvent émettre des signaux vers les autres processus (ou vers eux-mêmes).
- De nombreux signaux sont envoyés directement par le noyau afin de communiquer avec les processus.

Introduction

Gestion des signaux

Lorsqu'il reçoit un signal, un processus peut :

- Ignorer le signal.
- Utiliser la routine de traitement par défaut.
- Spécifier une fonction à exécuter.
- Bloquer le signal en attendant d'être prêt pour le traiter.

À la fin du traitement du signal, si le processus ne s'est pas terminé, il reprend le cours normal de son exécution.

Plan

6 Les signaux
- Introduction
- L'API POSIX pour les signaux standards
- Programmer avec les signaux
- Particularités de certains signaux
- Les signaux temps-réel

Les signaux standards POSIX

Spécifier un signal

- POSIX définit un ensemble de signaux « standards » qui ont une sémantique particulière.
- Un numéro est associé à chaque signal mais il peut changer d'une architecture à l'autre.
- POSIX définit un ensemble de macros permettant de spécifier un numéro de signal de façon portable.
- Les macros correspondant aux numéros de signal sont toutes préfixées par `SIG`

Remarque

Sous Linux, il existe 31 signaux standards.

Les signaux standards POSIX

Quelques signaux POSIX.1-1990

Signal	Détails
SIGHUP	Perte du terminal de contrôle ou mort du leader du groupe (suivi de *SIGCONT*).
SIGINT	Interruption depuis le clavier (^C).
SIGQUIT	Demande de fin depuis le clavier (^\).
SIGKILL	Signal de fin immédiate (numéro 9).
SIGPIPE	Écriture dans un tube sans lecteur.
SIGALRM	Alarme.
SIGTERM	Demande de terminaison du processus (avant un *SIGKILL*).
SIGUSR1	Signal utilisateur.
SIGUSR2	Signal utilisateur.

Les signaux standards POSIX

Quelques signaux POSIX.1-1990 (suite)

Signal	Détails
SIGCHLD	Fils terminé.
SIGSTOP	Arrêt.
SIGTSTP	Arrêt depuis le terminal (^Z).
SIGCONT	Continuer si arrêté.
SIGTTIN	Lecture sur le terminal en arrière plan.
SIGTTOU	Écriture sur le terminal en arrière plan.

Remarque

La liste complête des signaux supportés par Linux peut être consultée avec *man 7 signal*.

Les signaux standards POSIX

Traitement par défaut

À chaque signal est associé un traitement par défaut. Il y a cinq comportements par défaut possibles :

1. Fin : terminaison du processus (*SIGHUP*, *SIGINT*, *SIGKILL*, *SIGPIPE*, *SIGALRM*, *SIGTERM*, *SIGUSR1*, *SIGUSR2*).
2. Core : terminaison du processus avec sauvegarde de son contexte d'exécution dans un fichier *core* (*SIGQUIT*).
3. Ignorer : ignorer le signal (*SIGCHLD*).
4. Stop : stopper le processus (*SIGSTOP*, *SIGTSTP*, *SIGTTIN*, *SIGTTOU*).
5. Cont : reprendre l'exécution du processus si celui-ci était stoppé (*SIGCONT*).

Les signaux standards POSIX

Remarque

Sur de nombreuses distributions, la génération d'un fichier *core* est désactivée par défaut.
Sous Linux, il suffit parfois de supprimer la limite de taille pour les fichiers *core* pour réactiver la génération pour les programmes ordinaires. :

```
# ulimit -c unlimited
```

L'API POSIX pour la gestion des signaux

Envoyer un signal

On peut envoyer un signal à un processus en utilisant l'appel système *kill* :

```
#include <signal.h>

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

- $pid > 0$: le signal est envoyé au processus correspondant.
- $pid = 0$: le signal est envoyé à tous les processus appartenant au même groupe que l'appelant.
- $pid = -1$: le signal est envoyé à tous les processus (sauf 1).
- $pid < -1$: le signal est envoyé à tous les processus du groupe *-pid*.

L'API POSIX pour la gestion des signaux

Envoyer un signal

- De manière générale, pour pouvoir envoyer un signal à un autre processus, l'UID effectif ou réel de l'émetteur doit être identique à l'UID sauvé ou réel du récepteur.
- Un processus privilégié peut envoyer un signal à n'importe quel processus.
- Un processus peut envoyer un signal `SIGCONT` à tous processus appartenant à la même session que lui.

L'API POSIX pour la gestion des signaux

Remarques

- L'appel système `kill` est mal nommé puisqu'il ne tue pas forcément le processus cible.
- On peut utiliser le pseudo-signal 0 avec `kill` afin de déterminer si on peut atteindre par un signal un processus donné.

L'API POSIX pour la gestion des signaux

Les ensembles de signaux

- Les appels système gérant les signaux manipulent des ensembles de signaux.
- Les ensembles de signaux (*sigset_t*) peuvent être manipulés à l'aide des fonctions suivantes :
- `int sigemptyset(sigset_t *set);`
- `int sigfillset(sigset_t *set);`
- `int sigaddset(sigset_t *set, int signum);`
- `int sigdelset(sigset_t *set, int signum);`
- `int sigismember(const sigset_t *set, int signum);`

L'API POSIX pour la gestion des signaux

Les ensembles de signaux

On pourra créer un ensemble de signaux ne contenant que le signal *SIGCHLD* avec le code suivant :

```
sigset_t set;
if (sigemptyset(&set) == -1) {
  perror("sigemptyset");
  exit(EXIT_FAILURE);
}
if (sigaddset(&set, SIGCHLD) == -1) {
  perror("sigaddset");
  exit(EXIT_FAILURE);
}
```

L'API POSIX pour la gestion des signaux

Remarques

- Le type *sigset_t* est opaque, selon les implémentations, ce peut être un entier ou un tableau : on ne peut donc le manipuler qu'à l'aide de ces 5 fonctions.
- Question : quelle opération manque à l'appel ?

L'API POSIX pour la gestion des signaux

Et pourtant...

Ce qui est dit précédemment est vrai pour des codes compatibles avec la norme ISO mais POSIX ajoute une extension à cette norme :

```
sigset_t: [CX] Integer or structure type of an object
               used to represent sets of signals.
```

Nous pourrons donc copier des ensembles de signaux...

L'API POSIX pour la gestion des signaux

Bloquer un signal

Un processus possède un ensemble des signaux bloqués. Cet ensemble est spécifié à l'aide de *sigprocmask* :

```
int sigprocmask(int how, const sigset_t *set,
                sigset_t *oldset);
```

- *how* peut prendre les valeurs :
  - *SIG_BLOCK* les signaux présents dans *set* sont ajoutés à l'ensemble des signaux bloqués.
  - *SIG_UNBLOCK* les signaux présents dans *set* sont retirés de l'ensemble des signaux bloqués.
  - *SIG_SETMASK* l'ensemble des signaux bloqués est fixé à *set*.
- Si *oldset* est non *NULL*, l'ancienne valeur de l'ensemble des signaux bloqués est stocké dans **oldset*.
- On ne peut pas bloquer *SIGKILL* et *SIGSTOP* (les demandes sont ignorées silencieusemeent).

L'API POSIX pour la gestion des signaux

Les ensembles de signaux

Pour bloquer tous les signaux (sauf *SIGKILL* et *SIGSTOP*), nous pourrons utiliser le code suivant :

```
sigset_t set;
if (sigfillset(&set) == -1) {
  perror("sigfillset");
  exit(EXIT_FAILURE);
}
if (sigprocmask(SIG_SETMASK, &set, NULL) == -1) {
  perror("sigprocmask");
  exit(EXIT_FAILURE);
}
```

L'API POSIX pour la gestion des signaux

Bloquer un signal

- Si le processus bloque un signal qui arrive, ce dernier est placé en attente.
- Lorsque le processus débloque un signal qui a été reçu, celui-ci est alors traité.
- Tout se passe comme si les processus possédaient dans leur BCP un masque des signaux standards reçus et non traités.
- Cela signifie que si un processus reçoit plusieurs fois un même signal bloqué, une seule occurence du signal sera traitée lorsque le processus débloquera le signal.
- Il en est de même si le processus reçoit plusieurs fois le même signal avant que l'ordonnanceur ne décide de lui donner la main.

L'API POSIX pour la gestion des signaux

Transmission du masque des signaux

- Lors d'un *fork*, le fils possède une copie du masque des signaux de son père.
- Le masque des signaux est préservé au travers d'un *execve*.

Transmission des signaux

- Lors d'un *fork*, le fils possède un ensemble des signaux en attente vide (il n'hérite donc pas de celui de son père).
- L'ensemble des signaux en attente est préservé au travers d'un *execve*.

L'API POSIX pour la gestion des signaux

Traiter un signal

Nous avons vu qu'un processus peut associer une fonction à exécuter à un signal. L'appel système *sigaction* permet d'examiner et de modifier l'action associée à un signal :

```
int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
```

- Si *act* est non *NULL*, l'action spécifiée est associée au signal *signum*.
- Si *oldact* est non *NULL*, l'ancienne action associée au signal y est stockée.

L'API POSIX pour la gestion des signaux

Traiter un signal

- La structure `struct sigaction` contient les champs :

```
void      (*sa_handler)(int);
sigset_t    sa_mask;
int         sa_flags;
```

- `sa_handler` pointe sur la fonction à exécuter lorsque le signal `signum` est reçu.
- Le numéro du signal est transmis en paramètre lorsque le noyau fait appel à la fonction.
- La fonction exécutée lors de la réception est appelée le **gestionnaire** (*handler*) du signal.
- Les valeurs particulières `SIG_IGN` et `SIG_DFL` peuvent être affectées à `sa_handler` pour demander que le signal soit ignoré ou qu'il soit traité par l'action définie par défaut.

L'API POSIX pour la gestion des signaux

Traiter un signal

- *sa_mask* indique les signaux qui seront ajoutés au masque des signaux bloqués pendant l'exécution du gestionnaire du signal.
    - Le signal ayant provoqué l'appel au gestionnaire est automatiquement ajouté au masque des signaux bloqués.
    - Le masque des signaux retrouve son état d'origine au retour du gestionnaire.
- *sa_flags* permet de modifier le comportement par défaut de *sigaction*.
- Pour l'instant, nous fixerons *sa_flags* à 0.

L'API POSIX pour la gestion des signaux

Remarque

- La structure `struct sigaction` contient d'autres champs que nous n'utiliserons pas ici. Il **ne faut** donc **pas** initialiser la structure de manière statique :

```
struct sigaction act = {SIG_DFL, set, 0};
```

- De toute manière cette initialisation est incorrecte car elle suppose qu'on a le droit d'écrire quelque chose comme

```
act.sa_mask = set;
```

L'API POSIX pour la gestion des signaux

Exemple

Voici un exemple de programme qui intercepte le signal `SIGINT` afin d'afficher un message avant de se terminer.
Voir l'exemple

L'API POSIX pour la gestion des signaux

Traiter un signal

- Les processus stockent les actions associées à chaque signal dans les **dispositions des signaux**.
- Les dispositions des signaux forment un attribut de processus (stocké dans le BCP).
- Lors d'un `fork`, les dispositions des signaux sont hérités par le fils.
- Lors d'un `execve`, les dispositions des signaux sont réinitialisées aux valeurs par défaut.

Plan

6 Les signaux
- Introduction
- L'API POSIX pour les signaux standards
- Programmer avec les signaux
- Particularités de certains signaux
- Les signaux temps-réel

Interruption d'un appel système par un signal

Appels système lents et signaux

- Un appel système lent est un appel qui peut être amené à bloquer le processus indéfiniement.
- Par défaut, si un signal arrive alors que le processus est bloqué sur un appel système, le processus est réveillé et l'appel système échoue avec une erreur `EINTR`.
- Un exemple avec `read` ici.

Interruption d'un appel système par un signal

Appels système lents et signaux

- Certains appels système peuvent être automatiquement redémarrés après le retour du gestionnaire si on a spécifié l'attribut *SA_RESTART* dans le champ *sa_flags* lors de l'appel à *sigaction*.
- Un exemple avec *read* ici.

Remarque

Si *read* est interrompu par un signal alors qu'il avait déjà lu un certain nombre de données alors il retournera le nombre de données déjà lues.

Interruption d'un appel système par un signal

Remarques

- Certains appels système peuvent être lents ou pas en fonction de leurs paramètres.
- Par exemple, *read* est un appel système lent s'il lit des données sur un terminal mais pas s'il lit les données dans un fichier.
- De même *open* est un appel système lent s'il tente d'ouvrir un tube.
- *SA_RESTART* ne rédémarre pas tous les appels système lents.
- Une description détaillée des appels systèmes lents et du comportement de *SA_RESTART* sur ces appels système peut être consultée dans le manuel (*man 7 signal*).

Interruption d'une fonction réutilisée dans le gestionnaire

Fonctions sûres

- Un gestionnaire de signal peut interrompre le programme à n'importe quel endroit.
- Selon l'état du programme au moment de l'interruption, l'appel de certaines fonctions dans le gestionnaire du signal peut provoquer des résultats aléatoires.
- D'après le C ANSI, un gestionnaire ne devrait que modifier des variables globales de type `sig_atomic_t` et déclarées `volatile`.

Interruption d'une fonction réutilisée dans le gestionnaire

Fonctions sûres

- POSIX donne une liste de **fonctions sûres** pouvant être utilisées sans risque dans le gestionnaire du signal.
- Le comportement de l'appel à une fonction non sûre dans un gestionnaire qui a interrompu une fonction non sûre n'est pas définit.
- Les fonctions sûres sont dites *async-signal-safe*.

Remarque

Les fonctions *async-signal-safe* sont *thread-safe* mais la réciproque n'est pas forcément vraie.

Interruption d'une fonction réutilisée dans le gestionnaire

Fonctions sûres

Voici une liste de fonctions sûres que nous avons déjà rencontrées ou que nous allons rencontrer (la liste complète peut être consultée *via* `man 7 signal`).

```
stat, chmod, chown,
open, close, lseek, read, write,
mkdir, rmdir, link, unlink,
pipe,
dup, dup2,
execve, execle,  execv, execl,
fork, wait, waitpid,
getpid, getppid, getuid, ...
kill, sigaction, sigprocmask, sigsuspend, alarm,
sigemptyset, sigaddset, ...
```

Interruption d'une fonction réutilisée dans le gestionnaire

Fonctions non sûres

- On remarque, par exemple, l'absence de `malloc` et de `free` dans l'ensemble des fonctions sûres.
- `printf` est également une fonction non sûre : un gestionnaire devrait pouvoir disposer de son propre flux de sortie.
- Un gestionnaire devrait toujours s'assurer que, à la fin de la fonction, `errno` à la même valeur qu'à l'entrée.

Attente d'un signal

On peut attendre l'arrivée de certains signaux à l'aide de la fonction *sigsuspend* :

```
int sigsuspend(const sigset_t *mask);
```

- L'appel *sigsuspend* remplace temporairement le masque des signaux bloqués par **mask* puis endort le processus jusqu'à l'arrivée d'un signal ni bloqué, ni ignoré.
- Lorsqu'un tel signal arrive, son gestionnaire est exécuté puis *sigsuspend* restaure le masque des signaux avant de retourner -1 (et *errno* est fixé à *EINTR*).

Voir l'exemple

Plan

6 Les signaux
- Introduction
- L'API POSIX pour les signaux standards
- Programmer avec les signaux
- Particularités de certains signaux
- Les signaux temps-réel

Un signal particulier : *SIGALRM*

L'alarme

- Le signal *SIGALRM* est souvent utilisé pour indiquer qu'un délai maximal est écoulé.
- L'appel système *alarm* permet de programmer l'envoi du signal *SIGALRM* au bout de *nb_sec* secondes :

```
#include <unistd.h>
unsigned int alarm(unsigned int nb_sec);
```

- Si *nb_sec* vaut 0, aucune alarme n'est planifiée.
- *alarm* annule une précédente alarme éventuellement programmée.
- *alarm* retourne le nombre de secondes qu'il restait pour la précédente alarme.

Voir l'exemple

Le signal *SIGCHLD*

La chasse aux zombies

- Nous avons vus qu'un processus devrait toujours éliminer ses fils zombies à l'aide d'un *wait*.
- Mais si le père a aussi un travail à réaliser, il ne devrait pas attendre que ses fils meurent avant de pouvoir vaquer à ses occupations.
- Un signal *SIGCHLD* est envoyé au père à chaque fois qu'un de ses fils se termine.
- Un processus peut donc effectuer des *wait* lorsqu'il reçoit *SIGCHLD*.
- La difficulté réside dans le fait que plusieurs *SIGCHLD* peuvent être émis avant que le processus n'ait la main.
  Voir l'exemple.

Le signal *SIGCHLD*

Un vaccin contre les zombies

On peut également demander au système de ne pas transformer les fils en zombies lorsqu'ils se terminent en spécifiant *SA_NOCLDWAIT* dans *sa_flags* de la structure *sigaction* lors de l'appel à *sigaction*.
Voir l'exemple.

Le signal *SIGPIPE*

Rappel

Nous avons vu que *SIGPIPE* est émis lorsqu'un processus tente d'écrire dans un tube sans lecteur.

Plan

6 Les signaux
- Introduction
- L'API POSIX pour les signaux standards
- Programmer avec les signaux
- Particularités de certains signaux
- Les signaux temps-réel

Les signaux temps-réel

Propriétés

- Les signaux temps-réel peuvent être vus comme une extension aux signaux *SIGUSR1* et *SIGUSR2*.
- Posix.1b assure la présence d'au moins huit signaux temps réel.
- Les signaux temps-réel se distinguent des signaux standards par les points suivants :
  1. sauvegarde des occurences des signaux bloqués ;
  2. priorité associée aux signaux ;
  3. informations supplémentaires fournies au gestionnaire.

Les signaux temps-réel

Manipulation des signaux temps-réel

- Les signaux temps-réel n'ont pas de noms.
- On utilise directement leurs numéros qui s'étendent de *SIGRTMIN* à *SIGRTMAX*.
- Les signaux standards et les signaux temps-réel se partagent l'ensemble des numéros de signaux.
- Pour faciliter la lecture du code, on définiera généralement les signaux qu'on utilise à l'aide d'une constante symbolique :

```
#define SIG_PRET (SIGRTMIN + 2)
```

- *SIGRTMIN* et *SIGRTMAX* peuvent être des variables dont la valeur n'est accessible qu'à l'éxécution. Voir l'exemple

Les signaux temps-réel

Empilement des signaux temps-réel

- Les signaux temps-réel peuvent être « empilés ».
- Cela signifie que toutes les occurences d'un signal sont mémorisées et quelles seront donc toutes délivrées au processus cible.
- Il ne s'agit pas vraiment d'un empilement : les occurences d'un signal sont délivrées dans l'ordre où elles ont été émises.
- Pour s'assurer qu'un signal temps-réel est empilé, on utilise `sigqueue` à la place de `kill`.

Remarque

- Posix indique qu'un système doit permettre d'empiler au moins 32 signaux par processus.
- Linux ne suit pas la norme et gère les signaux à empiler au niveau de l'utilisateur (*cf.* `RLIMIT_SIGPENDING`).

Les signaux temps-réel

Prorité des signaux

- Lorsque le noyau doit délivrer plusieurs signaux à un processus, il commence toujours par le signal de plus petit numéro.
- Ainsi, plus le numéro d'un signal temps-réel est petit, plus la priorité du signal est élevée.

Remarque

- Posix ne précise pas l'ordre de délivrance des signaux standards.
- De même, Posix ne précise pas l'ordre de délivrance des signaux lorsqu'il y a des signaux temps-réel et des signaux standards en attente.

Les signaux temps-réel

Informations transmisent par un signal

- Un signal temps-réel peut transporter une petite quantité d'informations.
- En particulier, l'utilisateur peut transmettre une valeur de type `union sigval` :

```
union sigval {
  int    sival_int;
  void *sival_ptr;
};
```

Il s'agit donc, soit d'un entier, soit d'un pointeur.

Les signaux temps-réel

Informations transmisent par un signal

- L'information transmise par le signal est contenue dans une structure de type *siginfo_t*.
- *siginfo_t* contient au moins les champs :

```
int             si_signo;   // Numéro du signal
int             si_code;    // Origine du signal
int             si_errno;   // Si <>0, erreur ayant
                            // engendré le signal
pid_t           si_pid;     // PID de l'émetteur
                            // du signal
uid_t           si_uid;     // UID réel de l'émetteur
union sigval    si_value;   // Valeur transmise avec
                            // le signal
... // D'autres informations dépendant du signal
```

Les signaux temps-réel

Gestionnaire de signal temps-réel

- Le gestionnaire d'un signal temps-réel doit être capable de recevoir l'information transmise par le signal.
- Le prototype d'un signal temps-réel sera de la forme :

```
void gestionnaire(int signum,
                  siginfo_t *info,
                  void *context);
```

- *info* contient l'information transmise par le signal.
- *context* est un pointeur sur un *ucontext_t* correspondant au contexte du thread qui a reçu le signal au moment de la réception du signal (nous n'en dirons pas plus...).

Les signaux temps-réel

Action associée à un signal temps-réel

- Rappel : pour associer un gestionnaire à un signal on utilise une structure `struct sigaction` qu'on transmet à l'appel système *sigaction*.
- La structure `struct sigaction` contient en fait au moins un champ supplémentaire :

```
struct sigaction {
  void      (*sa_handler)(int);
  void      (*sa_sigaction)(int, siginfo_t *, void *);
  sigset_t    sa_mask;
  int         sa_flags;
  ...
};
```

- Le gestionnaire d'un signal temps-réel doit être placé dans le champ *sa_sigaction* et le champ *sa_flags* doit contenir l'attribut *SA_SIGINFO*.

Les signaux temps-réel

Exemple de définition d'un gestionnaire de signal temps-réel

```
struct sigaction action;
action.sa_sigaction = gestionnaire_temps_reel;
action.sa_flags = SA_SIGINFO;
if (sigfillset(&action.sa_mask) == -1) {
  perror("sigfillset");
  exit(EXIT_FAILURE);
}

// On associe l'action à SIGRTMIN
if (sigaction(SIGRTMIN, &action, NULL) == -1) {
  perror("sigaction");
  exit(EXIT_FAILURE);
}
```

Les signaux temps-réel

Exemple de définition d'un gestionnaire de signal temps-réel

Un autre exemple plus complet est ici.

Les signaux temps-réel

Émission d'un signal temps réel

Pour émettre un signal temps-réel en lui attachant une valeur, on utilise *sigqueue* au lieu de *kill* :

```
#include <signal.h>

int sigqueue(pid_t pid, int sig,
             const union sigval valeur);
```

La structure *siginfo_t* du gestionnaire temps-réel qui recevra le signal aura le champ *si_code* à *SI_QUEUE* et le champ *si_value* à *valeur*.

Les signaux temps-réel

Exemple d'envoi de signal

Voici l'exemple d'une commande permettant d'envoyer un signal temps-réel avec une valeur entière : sigqueue.