

Unix - e2i5 - Introduction à la programmation système sous Linux

Nicolas Palix

UJF/Polytech

2017

1 Programmation système ?

2 Programmer en C sous Linux

- Integrated Development Environment (IDE)
- Éditer son code
- Compiler
- Moteur de production (Build automation)
- Déboguer (Debug)

3 Les bibliothèques importantes de la programmation système

4 Gestion de fichiers

- Principes fondamentaux
- Implantation Unix/Linux

Définition

Dans les cours d'Unix précédents, vous avez appris à utiliser le système d'exploitation Linux.

Programmation système

Développement de programmes qui font partie du système d'exploitation d'un ordinateur, qui en réalisent les fonctions, qui utilisent les fonctions avancées de celui-ci.

Définition

Dans les cours d'Unix précédents, vous avez appris à utiliser le système d'exploitation Linux.

Programmation système

Développement de programmes qui font partie du système d'exploitation d'un ordinateur, qui en réalisent les fonctions, qui utilisent les fonctions avancées de celui-ci.

Exemples

L'accès aux fichiers, la gestion des processus, la programmation réseau, les entrées/sorties, la gestion de la mémoire...

But du cours

Le but de ce cours est de vous apprendre comment interagir avec un système d'exploitation de type Unix, et mettre en oeuvre les différents mécanismes système classiques.

Pourquoi la programmation système ?

Les principales applications :

- La création de plusieurs processus
- Tout ce qui a trait à l'accès de ressources de manière *concurrente* par plusieurs processus s'exécutant sur le même processeur.
 - Partage de données entre plusieurs processus (un exemple classique est le producteur/consommateur)
 - Partage de ressources entre plusieurs processus, avec une possible exclusion mutuelle.
 - ...
- Les communications entre processus (échanger des données qui ne sont pas partagées)
- La synchronisation entre processus (ne pas lire des données qui ne sont pas complètement écrite)

Pourquoi sous Linux et comment ?

- Les concepts sont généraux et se retrouvent dans tous les systèmes d'exploitation
- Les sources du noyau de Linux sont accessibles et open source,
- Les interfaces sont normalisées (norme POSIX)
- Le noyau Linux est écrit en C,

Pour faire de la programmation système sous Linux, il faut donc savoir comment coder en C sous Linux.

1 Programmation système ?

2 Programmer en C sous Linux

- Integrated Development Environment (IDE)
- Éditer son code
- Compiler
- Moteur de production (Build automation)
- Déboguer (Debug)

3 Les bibliothèques importantes de la programmation système

4 Gestion de fichiers

- Principes fondamentaux
- Implantation Unix/Linux

Les différentes tâches

Pour programmer sous Linux, on a besoin :

- D'éditer son code,
- De le compiler,
- De le lier,
- De le corriger.

La plupart du temps, on utilise un environnement de développement intégré, *IDE*, tels que Eclipse ou Netbeans qui regroupe l'ensemble de ces fonctionnalités. Pourquoi utiliser autre chose ?

- Lorsque l'on travaille sur des machines distantes, il est parfois difficile d'utiliser des outils nécessitant d'exporter l'affichage graphique.
- Les IDE reposent sur la plupart des outils que nous allons lister pour chacune des fonctionnalités qu'ils offrent.
- Il est parfois utile de comprendre ce que l'on génère avec un IDE (pour voir ce qui ne va pas).

L'IDE Eclipse

- Eclipse (projet de la Fondation Eclipse) : environnement de développement **libre, extensible, et polyvalent.**
- Crée en 2001 sous l'impulsion d'IBM pour contrer Sun qui a son propre IDE (NetBeans).
- Architecture volontairement orienté vers l'ajout de greffons.
- Permet de gérer des projets dans plus de 20 langages différents.
- Maintenu, mis à jour régulièrement.
- Très répandu en entreprise.
- Mais ce n'est pas le seul !

1 Programmation système ?

2 Programmer en C sous Linux

- Integrated Development Environment (IDE)
- Éditer son code
- Compiler
- Moteur de production (Build automation)
- Déboguer (Debug)

3 Les bibliothèques importantes de la programmation système

4 Gestion de fichiers

- Principes fondamentaux
- Implantation Unix/Linux

Les outils à disposition

Si l'on ne dispose pas d'affichage graphique, on doit passer par un éditeur en ligne de commande, tels que :

- vi : date de ... 1976. Installé par défaut sur la majorité des Linux.
- emacs : date aussi de 1976. À peine moins installé que vi. Utilise LISP pour ses greffons.
- nano : date de 2000. Plus accessible pour les Windowsiens. Équivalent au bloc-notes.
- pico : éditeur de texte par défaut pour le logiciel pine (email). Sous licence. nano est basée sur l'interface pico.



<http://www.luc-damas.fr/humeurs/pour-coder-plus-vite/>

Éditeur de texte emacs

```
1 int main(int argc, char **argv) {
2     return 0;
3 }
4
5 // Local Variables:
6 // mode: c
7 // coding: utf-8
8 // End:
```

```
1 ;; ~/.emacs
2 (setq-default indent-tabs-mode t)
3 (setq-default tab-width 8)
4 (setq c-basic-offset 8)
5 (setq c-default-style "linux" c-basic-offset 8)
6 ;; style: gnu|k&r|bsd|linux|java|awk|python
```

Éditeur de texte vi

vi : Visual Interface

- Écrit par Bill Joy en 1976 (Unix BSD),
- Vi est l'un des éditeurs de texte CLI les plus populaires sous Linux (avec Emacs).
- Éditeur en mode texte, ce qui signifie que chacune des actions se fait à l'aide de commandes texte. (Très pratique en cas de non fonctionnement de l'interface graphique ...).
- Très puissant par un système de commandes (et de macros)
- On ne quitte jamais le clavier des doigts, donc très rapide quand on a l'habitude

vi : modes de fonctionnement

Vi possède plusieurs modes de fonctionnement. Les 2 principaux modes sont :

- Mode commande : mode à l'appel de l'éditeur. De nombreuses commandes peuvent être effectuées avec des séquences de touches simples :
 - Exemple 1 : d3w pour "delete 3 words" efface les 3 mots qui suivent
 - Exemple 2 : c2fa pour "change le texte jusqu'à ce qu'il trouve le 2e a"
 - Les touches tapées en mode commande ne sont pas insérées dans le texte
- Mode insertion :
 - Le texte tapé est inséré dans le document
 - Passage d'un mode à l'autre ESC ou a A i I ...

vi : commandes de base

- Depuis la console, pour appeler vi : vi <fichier>
Crée le fichier s'il n'existe pas (si on a les droits ...)
- Pour sortir en sauvegardant un fichier : :wq (write and quit)
- Pour sortir sans sauvegarder : :quit! (ou q!)

Autres commandes de base

Déplacements :

- h, j, k, l pour se déplacer caractère par caractère
- w, b pour se déplacer par mot
- ngg pour aller à la n^e ligne (identique à :n)
- ^, \$ pour aller en début ou fin de ligne

Effacement :

- x pour effacer un caractère
- dd pour effacer une ligne (ndd pour effacer n lignes)

Recherche :

- /motif pour rechercher un motif (n pour trouver le suivant)

Remplacement :

- r remplace le caractère sur lequel on se trouve par celui qui sera tapé après le r
- cw pour "change word"

Commande à l'appel :

- vi -c "10,%s/Deux/Trois/g|:wq" fichier

Pour coder proprement...

- = indente la ligne courante
- =10j indente les 10 prochaines lignes
- gg=G indente le programme entier
 - gg va à la première ligne
 - = indente
 - G jusqu'à la fin
- :set number numérote les lignes
- :syntax on active la coloration syntaxique
- Sauvegarder votre configuration dans ~/.vimrc

... configurez votre éditeur

<http://vimdoc.sourceforge.net/htmldoc/options.html#option-summary>

```
1 // modeline
2 // vim: set style=linux-kernel
3
4 // Shift Width: Indent size (8) ; (No) AutoIndent
5 // (No) Expand Tab: Indent with spaces instead of tabs
6 // vim: set ts=8 sw=8 ai noet
7 // vim: set syntax=c fenc=utf-8 ff=unix
```



<http://www.luc-damas.fr/humeurs/lindentation-cest-important/>

Conclusion sur vi

- On utilisera une version améliorée : vim
- Vim : logiciel libre. Son code source a été publié pour la première fois en 1991 par Bram Moolenaar, son principal développeur.
- VI aMélioré car il possède un langage de macros.

Il existe aussi une version graphique de vi permettant l'utilisation de la souris (gvim) non installée dans la salle Linux 216. Pour apprendre à utiliser vim, vimtutor

Compiler : gcc

GCC et gcc

GCC (GNU Compiler Collection, à l'origine GNU C Compiler).

- Permet de compiler du C,
- Mais aussi du C++, Ada, Fortran, Pascal, VHDL, D, Objective C, Java...
- On parle de gcc quand on veut compiler du C.

gcc (gnu c compiler)

- **gcc est utilisé pour compiler le noyau Linux**
- gcc a de très nombreuses options (une quarantaine). Les plus utilisées sont :
 - -I : include
 - -L : link
 - -l : librairies à lier
 - -O<number> : optimisation du code
 - -Werror : Warning traités comme des erreurs
 - -Wall : (Presque) tous les warning possibles sont annoncés
 - -D : passer des paramètres

gcc

- Beaucoup d'options : beaucoup de possibilités de se tromper !
- Un exemple d'appel gcc pris au hasard :

```
1 /usr/bin/gcc -Dgras_EXPORTS -O0 -Wall -Wunused
2 -Wmissing-prototypes -Wmissing-declarations
3 -Wpointer-arith -Wchar-subscripts -Wcomment
4 -Wformat -Wwrite-strings -Wno-unused-function
5 -Wno-unused-parameter -Wno-strict-aliasing
6 -Wno-format-nonliteral
7 -Werror -L/usr/lib -l/usr/include -g3 -fPIC
8 -I/home/npalix/projects/simgrid/simgrid
9 -I/home/npalix/projects/simgrid/simgrid/include
10 -I/home/npalix/projects/simgrid/simgrid/src
11 -I/home/npalix/projects/simgrid/simgrid/src/include
12 -I/home/npalix/projects/simgrid/simgrid/build
13 -I/home/npalix/projects/simgrid/simgrid/build/include
14 -I/home/npalix/projects/simgrid/simgrid/build/src
15 -o CMakeFiles/gras.dir/src/gras/Virtu/gras_module.c.o
16 -c /home/npalix/projects/simgrid/src/gras/Virtu/gras_module.c
```

gcc

- gcc est le compilateur standard sous Linux
- Il existe des portages sur les principaux OS (Windows, MacOS) et pour beaucoup de microprocesseurs (AMD64, ARM, DEC Alpha, M68k, MIPS, PowerPC, SPARC, x86, Hitachi H8).
- Il existe des alternatives (PathScale, TinyCC) mais leur utilisation reste exotique.

On utilisera gcc pour compiler. Mais comme tout compilateur, il a ses inconvénients ...

- gcc est un outil très puissant ...
- ... utilisable "à la main" quand on travaille sur des micro-projets,
- mais il est **vital** d'utiliser des **outils de build** dès que l'on passe à des projets de taille même moyenne

Moteur de production

Définition

Construction de l'exécutable à partir des sources fournis et des bibliothèques à lier.

Il existe beaucoup d'outils :

- make : Outil sous-jacent à pas mal d'IDE. Se base sur des Makefile.
- ant : Même utilisation que make, mais plus facile d'accès. Utilisé par NetBeans pour générer du code. Orienté java à la base. Syntaxe XML.
- Automake/Autoconf : Outils GNU pour générer des Makefile portables.
- cmake : Moteur de production multiplates-formes. 2 phases : on génère les fichiers de build (Makefile par exemple) et ensuite on utilise ces fichiers. De plus en plus utilisé (gcc par exemple pour se compiler)
- Et bien d'autres : Rant, Rake, Remake, Scons, pmake, cook, jam, dmake, hmake...

make

Maintient la cohérence entre un programme exécutable et les fichiers sources permettant de le produire.

- Basé sur les dates de dernière modification des fichiers
- Nécessite un script contenant des règles
- Chaque règle indique le moyen de construire une cible particulière à partir des ressources dont elle a besoin
- Si une cible est moins récente que l'une de ses dépendances, elle est automatiquement reconstruite.

Exécution :

```
make [-f fichier] [cible]
```

- Si l'option -f est absente, le fichier du répertoire courant appelé Makefile ou makefile est utilisé.
- Si la cible est absente, la première cible est considérée.

Makefile : syntaxe

pour les commentaires jusqu'à fin de ligne Chaque déclaration de règle contient :

- Sur la première ligne : <cible> ':' <liste des dépendances>
- Sur les suivantes : une tabulation suivie des commandes à exécuter

Exemple

```
1 # Construction du programme executable
2 executable : fich1.o fich2.o fich3.o
3         →gcc -ansi -Wall -o executable fich1.o fich2.o fich3.o
4 # Construction du fichier fich1.o
5 fich1.o : fich1.c fich1.h fichier_entete.h
6         →gcc -ansi -Wall -o fich1.o -c fich1.c
```

Makefile : un exemple

```
1 CC=gcc
2 RM=rm -f
3 CFLAGS= -Werror -Wall
4
5 SRC=hello.c main.c
6 OBJECTS=$(SRC:.c=.o)
7
8 all: executable
9
10 # %.o : %.c similaire .c.o
11 # $< premiere dependance
12 %.o: %.c fichier.h
13 → @echo "[ $< ]"
14 → $(CC) $(CFLAGS) -o $@ -c $<
15
16 # $@ indique le nom de la cible
17 executable: $(OBJECTS)
18 → $(CC) $(CFLAGS) -o $@ $(OBJECTS)
19
20 # Si un fichier clean existe – execution de la commande
21 .PHONY : clean
22 clean:
23 → $(RM) $(OBJECTS) executable *~
```

- Construction automatique de la liste des fichiers objets
- Commande silencieuse pour ne pas afficher les messages

1 Programmation système ?

2 Programmer en C sous Linux

- Integrated Development Environment (IDE)
- Éditer son code
- Compiler
- Moteur de production (Build automation)
- Déboguer (Debug)

3 Les bibliothèques importantes de la programmation système

4 Gestion de fichiers

- Principes fondamentaux
- Implantation Unix/Linux

Déboguer

Définition

Partir à la recherche d'erreurs dans le code, de fuite mémoire, etc.

Il existe aussi beaucoup d'outils de débogue, comme par exemple :

- gdb : The GNU Project Debugger. Richard Stallman, 1988. Standard sous Linux, mis à jour régulièrement. Vous permet de :
 - De démarrer vos programmes, en spécifiant quoi que ce soit que puisse changer son comportement
 - Faire stopper votre programme quand certaines conditions sont réalisées.
 - Examiner ce qui s'est passé, une fois que votre programme a stoppé.
 - Changer des choses dans votre programme, pour corriger les effets d'un bug et en chercher un autre.
- ddd : Interface graphique pour gdb.

Déboguer (suite et fin)

- Valgrind : libre, pour déboguer, effectuer du profilage de code et mettre en évidence des fuites mémoires.
 - Utilisation de la mémoire (free, memory links, etc)
 - Exploration de la pile
 - Statistique d'utilisation
 - Utilisation de zones mémoires non allouées ou non initialisées...
- kdbg : que vous avez utilisé l'an dernier.

1 Programmation système ?

2 Programmer en C sous Linux

- Integrated Development Environment (IDE)
- Éditer son code
- Compiler
- Moteur de production (Build automation)
- Déboguer (Debug)

3 Les bibliothèques importantes de la programmation système

4 Gestion de fichiers

- Principes fondamentaux
- Implantation Unix/Linux

Les bibliothèques de la programmation système

La bibliothèque standard C de la norme ISO est minimaliste (24 en standards)

- Fonctions mathématiques,
- Manipulation de chaînes de caractères,
- Conversion de types,
- Entrée/sortie maniant les fichiers et les terminaux

Les bibliothèques de la programmation système

La bibliothèque standard C de la norme ISO est minimaliste (24 en standards)

- Fonctions mathématiques,
- Manipulation de chaînes de caractères,
- Conversion de types,
- Entrée/sortie maniant les fichiers et les terminaux

Les fonctions habituellement classiques dans les autres langages, tels que Java, ne sont donc pas des standards de la norme ISO !

Norme POSIX

Pour remédier à cela, la norme POSIX a été créée.

POSIX

- Famille de standards IEEE 1003 (début 1988)
- POSIX : Portable Operating System Interface for UniX
- Interfaces utilisateurs, interfaces logicielle
- Librairie POSIX C : donne les entêtes pour les fonctionnalités manquantes dans l'ISO/ANSI

18 fichiers d'entêtes qui viennent compléter les fonctions systèmes déjà offertes.

Les entêtes POSIX

cpio.h	Magic numbers for the cpio archive format. (deprecated)
dirent.h	Allows the opening and listing of directories.
fcntl.h	File opening, locking and other operations.
grp.h	User group information and control.
pthread.h	Defines an API for creating and manipulating POSIX threads.
pwd.h	passwd (user information) access and control.
tar.h	Magic numbers for the tar archive format.
termios.h	Allows terminal I/O interfaces.
unistd.h	Various essential POSIX functions and constants.
utime.h	inode access and modification times.

Les entêtes POSIX - 2

sys/ipc.h	Inter-process communication (IPC).
sys/msg.h	POSIX message queues.
sys/sem.h	POSIX semaphores.
sys/stat.h	File information (stat et al.).
sys/time.h	Time and date functions and structures.
sys/types.h	Various data types used elsewhere.
sys/utsname.h	name and related structures.
sys/wait.h	Status of terminated child processes (see wait)

1 Programmation système ?

2 Programmer en C sous Linux

- Integrated Development Environment (IDE)
- Éditer son code
- Compiler
- Moteur de production (Build automation)
- Déboguer (Debug)

3 Les bibliothèques importantes de la programmation système

4 Gestion de fichiers

- Principes fondamentaux
- Implantation Unix/Linux

Généralités sur les fichiers

- Un fichier est un objet typé dont le type permet de définir l'ensemble des opérations applicables.
- D'un point de vue interne, à chaque fichier correspond une entrée dans une table contenant l'ensemble des attributs (type, propriétaires, droits, ...).
- Une telle entrée est appelée i-node (*index node*). Un fichier a donc comme identification un couple constitué de l'identification de la table dans laquelle est enregistré ses caractéristiques (on parle de disque logique), et de l'indice dans cette table.

Système de gestion de fichiers

① Régulier (*regular*)

- Texte : codage ASCII, UTF-8, ISO-8859-15 (latin 9)
- Binaire : code machine d'un programme, données multimédia

② Répertoires (*directory*) : Liste de fichiers

③ Lien (*link*) : Lien symbolique donnant le chemin vers un autre fichier.

④ Spéciaux (*special*) :

- Caractère (*character*) : imprimante, terminal, mémoire, modem, ...
- Périphériques blocs (*block device*) : Similaire aux fichiers spéciaux mais le mode d'accès est orienté blocs de données au lieu de caractère.
Utilisé pour les disques durs.

⑤ *Socket* : sockets Unix, similaires aux sockets TCP/IP. Utilise le système de droits des fichiers.

⑥ Tube nommé (*named pipe*) : Plus ou moins comme les sockets Unix sans la sémantique réseau.

Connaître le type des fichiers

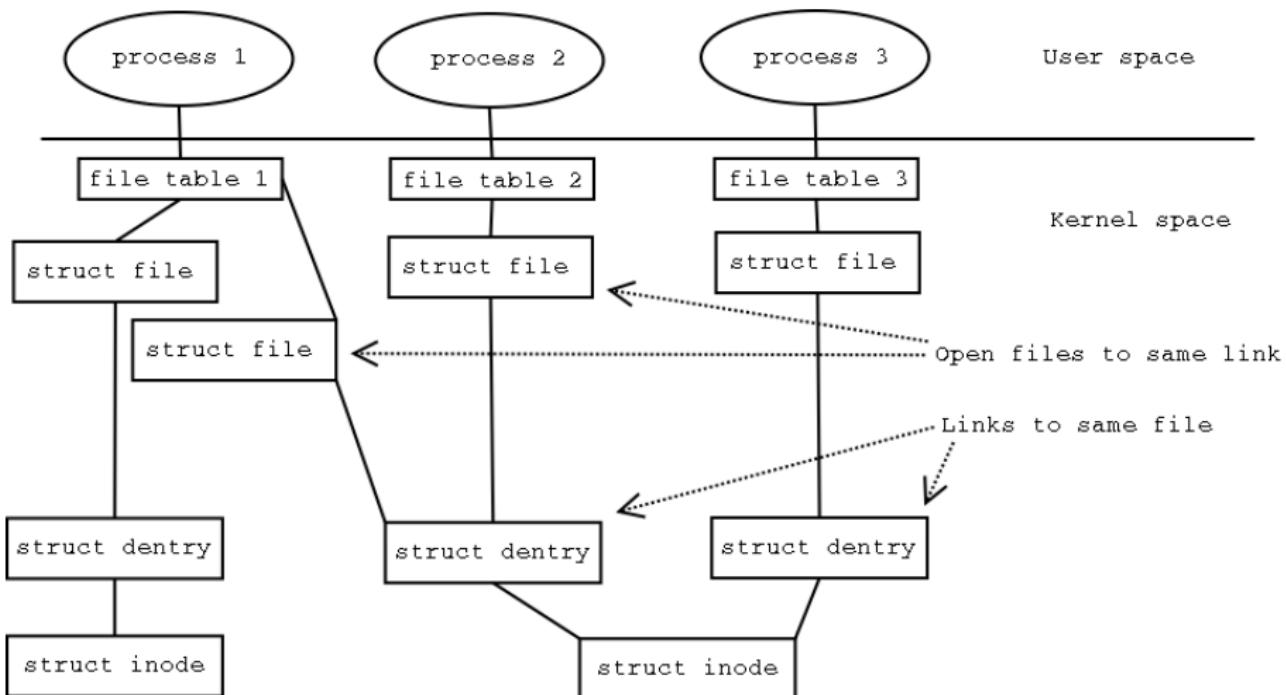
Ces types sont visibles avec la commande ls -l

- Regular	d Directory	c Character device
l Link	s Socket	p Named pipe
b Block device		

Exemple avec des entrées de /dev

```
lrwxrwxrwx  1 root    root          13 Oct 15 14:48 fd -> /proc/self/fd
drwxr-xr-x  4 root    root         300 Oct 15 14:48 input
crw-----  1 root    root        1,  11 Oct 15 14:48 kmsg
srw-rw-rw-  1 root    root          0 Oct 15 14:48 log
brw-rw----  1 root disk        7,   0 Oct 15 14:48 loop0
```

Système de fichiers virtuel



i-node

Structure d'un i-node :

- L'identification du propriétaire et du groupe
- Le type de fichier et droits d'accès des différents utilisateurs
- La taille du fichier exprimée en nombre de caractères (pas de sens pour les fichiers spéciaux)
- Le nombre de liens physiques
- Les trois dates significatives (lecture, modification du fichier et modification du nœud)
- L'identification de la ressource associée (pour les fichiers spéciaux)
- Adresse sur le disque
- ...

La commande stat donne les informations sur un nœud.

Exemple

```
login@host:~$ stat exo.c
  File: 'exo.c'
  Size: 331          Blocks: 8          IO Block: 4096   regular file
Device: 900h/2304d      Inode: 47579146      Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1010/ frousse)  Gid: ( 1002/      sls)
Access: 2008-02-14 09:55:40.000000000 +0100
Modify: 2007-10-30 16:06:42.000000000 +0100
Change: 2007-10-30 16:06:42.000000000 +0100
```

Information sur les i-nodes en C

On accède en C à toutes ces informations par la structure stat définie dans <sys/stat.h>

```
struct stat {  
    dev_t      st_dev;      /* ID of device containing file */  
    ino_t      st_ino;      /* inode number */  
    mode_t     st_mode;     /* protection */  
    nlink_t    st_nlink;    /* number of hard links */  
    uid_t      st_uid;      /* user ID of owner */  
    gid_t      st_gid;      /* group ID of owner */  
    dev_t      st_rdev;     /* device ID (if special file) */  
    off_t      st_size;     /* total size, in bytes */  
    blksize_t  st_blksize;  /* blocksize for filesystem I/O */  
    blkcnt_t   st_blocks;   /* number of blocks allocated */  
    time_t     st_atime;    /* time of last access */  
    time_t     st_mtime;    /* time of last modification */  
    time_t     st_ctime;    /* time of last status change */  
};
```

On peut changer le contenu de cette structure avec la fonction stat

```
int stat(const char *ref, struct stat *ptr);
```

Système de gestion de fichiers

- Il existe une table de fichiers par partition. Chacune correspond donc à une arborescence indépendante. La racine de ses arborescences a comme index 2 en ext2/3/4 (`ls -id /`)
- De manière générale, certains indexes sont réservés. L'index 0 dénote un échec/une erreur.
- Les différentes arborescences peuvent être reliées entre elles par un mécanisme de *montage* : il s'agit de greffer la racine d'une arborescence non encore montée, en un point accessible depuis la racine absolue du système.
 - La commande de montage est `mount`. Par exemple, pour monter votre clé USB : `mount /dev/sda1 /media/CleUSB`. On obtient la racine du disque logique avec `dmesg`
 - La commande pour démonter est `umount` : `umount /media/CleUSB`

Système de gestion de fichiers

- Le montage de systèmes de fichiers permet l'accès à des systèmes distants, tel que NFS (Network File System) ou SMB.
- L'objectif fondamental est de permettre un accès aux fichiers par l'intermédiaire des mêmes fonctions génériques (open, read, write) indépendamment du système accédé.
- Problème de sécurité et de cohérence des informations dans le cas des systèmes distants.

Les primitives de base : open, close, read, write

open : ouverture de descripteur

open permet à un processus de réaliser l'ouverture de fichier, càd demande l'allocation d'une nouvelle entrée dans la table des fichiers ouverts du système. Si c'est possible, on charge l'i-node correspondant en mémoire. Un descripteur est alloué dans la table des descripteurs du processus. La fonction est

```
#include <fcntl.h>
int open(const char *ref, int flags [, mode_t mode]);
```

- ref est le nom du fichier.
- flags est obtenu par disjonction logique avec O_RDONLY, O_WRONLY, O_RDWR, O_CREAT, ...
- mode définit les droits d'accès au fichier. Ce paramètre est optionnel.

Les primitives de base : open, close, read, write

open : ouverture de descripteur

open permet à un processus de réaliser l'ouverture de fichier, càd demande l'allocation d'une nouvelle entrée dans la table des fichiers ouverts du système. Si c'est possible, on charge l'i-node correspondant en mémoire. Un descripteur est alloué dans la table des descripteurs du processus. La fonction est

```
#include <fcntl.h>
int open(const char *ref, int flags [, mode_t mode]);
```

En cas de succès, la valeur renvoyée est l'indice, dans la table des descripteurs du processus, du descripteur alloué (on l'appelle le descripteur). De plus, la position courante dans le fichier est égale à 0 (début de fichier).

Attention au mode bloquant !!!

Les primitives de base : open, close, read, write

Exemple : Pour créer un fichier

- Fonction creat()
- `open ("fichier", O_WRONLY|O_CREAT|O_TRUNC, 666);`
`O_WRONLY|O_CREAT|O_TRUNC` similaire "w" pour fopen()

Les primitives de base : open, close, read, write

close : fermeture de descripteur

```
int close(int descripteur);
```

read : lecture de fichier

```
ssize_t read (int descripteur, void *ptr, size_t nb_octets);
```

Demande de lecture d'au plus nb_octets caractères via le descripteur. Les caractères lus sont écrits dans l'espace d'adressage du processus à l'adresse ptr.

write : écriture de fichier

```
ssize_t write (int descripteur, void *ptr, size_t nb_octets);
```

Demande d'écriture dans le fichier descripteur de nb_octets caractères trouvés à l'adresse ptr.

Unix - e2i5 - Processus

Nicolas Palix

Polytech

2017

1 La notion de processus

- Définition
- Caractéristiques et propriétés des processus
- Les processus et le système d'exploitation
- Relations entre processus
- Spécificités Unix

2 Manipulation des processus sous Unix

- Les commandes de manipulation
- API de manipulation de processus

3 Exemples

Définition

Processus

Entité dynamique représentant l'exécution d'un programme sur un processeur

Du point de vue du système :

- Espace d'adressage (mémoire, contient données + code)
- État interne (compteur d'exécution, fichiers ouverts, etc.)

Programme != processus

Un programme peut être exécuté plusieurs fois et se trouver dans plusieurs unités d'exécution en même temps

Vous pouvez utiliser le même *programme* que moi, mais ça ne sera pas le même *processus* que moi. Un *processus* a une durée de vie qui correspond à la fin de son *exécution*, mais rien ne l'empêche d'exécuter plusieurs *programmes* les uns à la suite des autres.

Le processus doit donc connaître à chaque instant :

- Le code du programme
- Le pointeur d'instruction
- L'état de la pile
- Les variables

Exemples de processus

Exemples :

- **Exécution** d'un programme
- La copie d'un fichier sur disque
- Transmission d'une séquence de données sur un réseau
- Un terminal

Un processus :

- Utilise le CPU (exécution)
- Utilise des ressources qui peuvent être partagées
- Des ressources partagées peuvent être utilisables uniquement par un (ou n processus simultanément) (disques, imprimante, carte d'acquisition, contrôleurs)
- On doit donc gérer d'éventuels **inter-blocages (*deadlock*)** (voir cours synchronisation de processus).

Contre-exemple

Lancer une commande dans un terminal : nouveau processus créé. Le terminal lance un nouveau processus "fils" qui exécute la commande.

- La commande est un **nouveau processus à part**. *ps -jH (voir plus loin) permet de bien voir ça.*
- Mettre un "&" ne change en fait rien à la nature du processus. Sans le "&" le processus du shell (processus *père*) attend que la commande (le processus *fils*) lui redonne accès au terminal auquel il est associé. Avec le "&", le terminal est utilisable par le processus *père* (la commande est exécutée en tâche de fond).
- Par contre, un "&" détache le processus du terminal de son père. Si son père est tué, le processus survit, car (c'est un cas particulier) la destruction du père n'implique que la destruction des processus lancés en *foreground* dans le terminal.

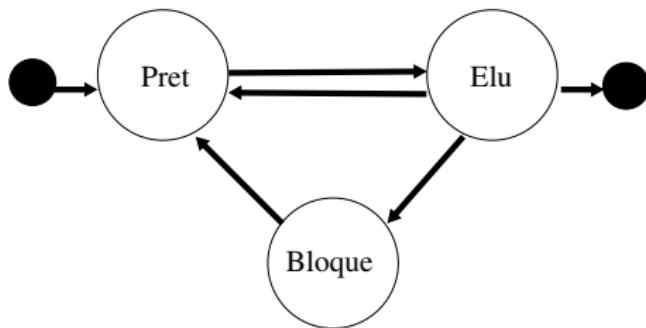
Les processus et le système d'exploitation

- Les processus permettent d'exécuter plusieurs programmes "simultanément" sur un seul CPU : **on appelle cela du pseudo-parallélisme.**
- HS : **Le parallélisme est l'exécution de plusieurs processus sur plusieurs CPU, d'où "pseudo".**
- Pour pouvoir exécuter plusieurs processus, le système d'exploitation choisit d'exécuter un sous-ensemble des instructions d'un processus, puis de passer à un autre processus et de faire de même.
- Choisir la taille de ce sous-ensemble d'instructions, l'ordre dans lequel on choisit d'exécuter ces sous-ensembles, est appelé une **politique d'ordonnancement** (avec ou sans priorité entre les processus, équitable ou non ...). Un algorithme d'ordonnancement est un algorithme qui permet de choisir la prochaine tâche à exécuter.

Les différents états d'un processus (vu du processus)

Un processus peut être dans 3 états possibles (un seul à la fois) :

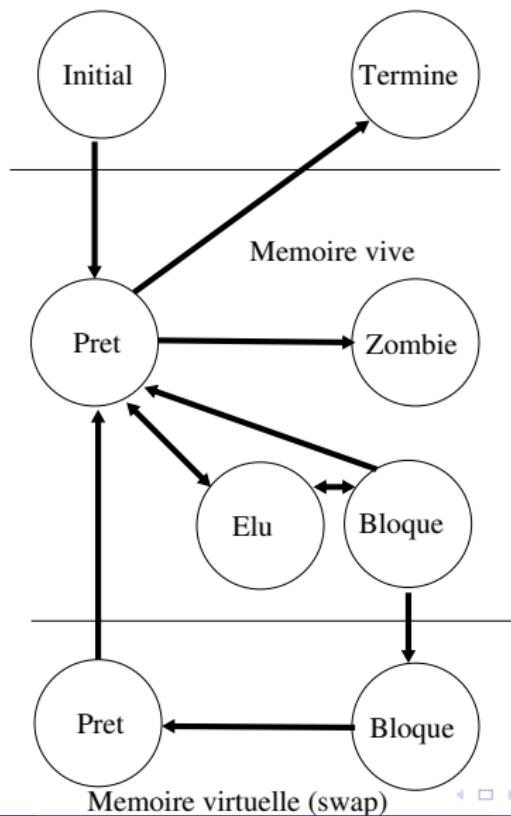
- élu (en cours d'exécution) -> processus OK, processeur OK
- prêt (suspendu provisoirement pour qu'un autre processus s'exécute)
-> processus OK, processeur non OK (occupé)
- bloqué (attendant un événement extérieur pour continuer) ->
processus non OK, même si processeur OK



Les états d'un processus (vu du système)

- Les processus doivent exister pour avoir un état.
- Les processus doivent avoir un identifiant pour pouvoir être manipulés : c'est le PID (Process Id).
- Le système se charge de les tuer et de les créer : il manipule donc plus d'états
- Certains états sont communs à tous les SE : initialisé (attend de passer à prêt), terminé.
- D'autres moins standards
 - Zombie : terminé mais son père n'a pas traité la notification,
 - Orphelin : son père est mort mais il s'exécute encore. Il sera adopté par le processus init d'ID 1,
 - Swappé (prêt ou endormi)

Les états d'un processus (vu du système)



Propriétés de l'ordonnancement

Plusieurs processus sont prêts à être exécutés. Le SE doit faire un choix selon un critère :

- Équité : chaque processus doit avoir du temps processeur
- Efficacité : le processeur doit être utilisé à 100%
- Temps de réponse : l'utilisateur devant sa machine ne doit pas trop attendre
- Temps d'exécution : une séquence d'instructions ne doit pas trop durer
- Rendement : il faut faire le plus de choses en une heure

Choix fait par l'algorithme d'ordonnancement qui peut être :

- Sans réquisition : un processus est exécuté jusqu'à la fin. peut être inefficace et dangereux (ex : exécution d'une boucle sans fin)
- Avec réquisition : à chaque signal d'horloge, le SE reprend la main, décide si le processus courant a consommé son quota de temps machine et alloue éventuellement le processeur à un autre processus

Il existe de nombreux algorithmes d'ordonnancement avec réquisition

Ordonnancement des processus : tourniquet

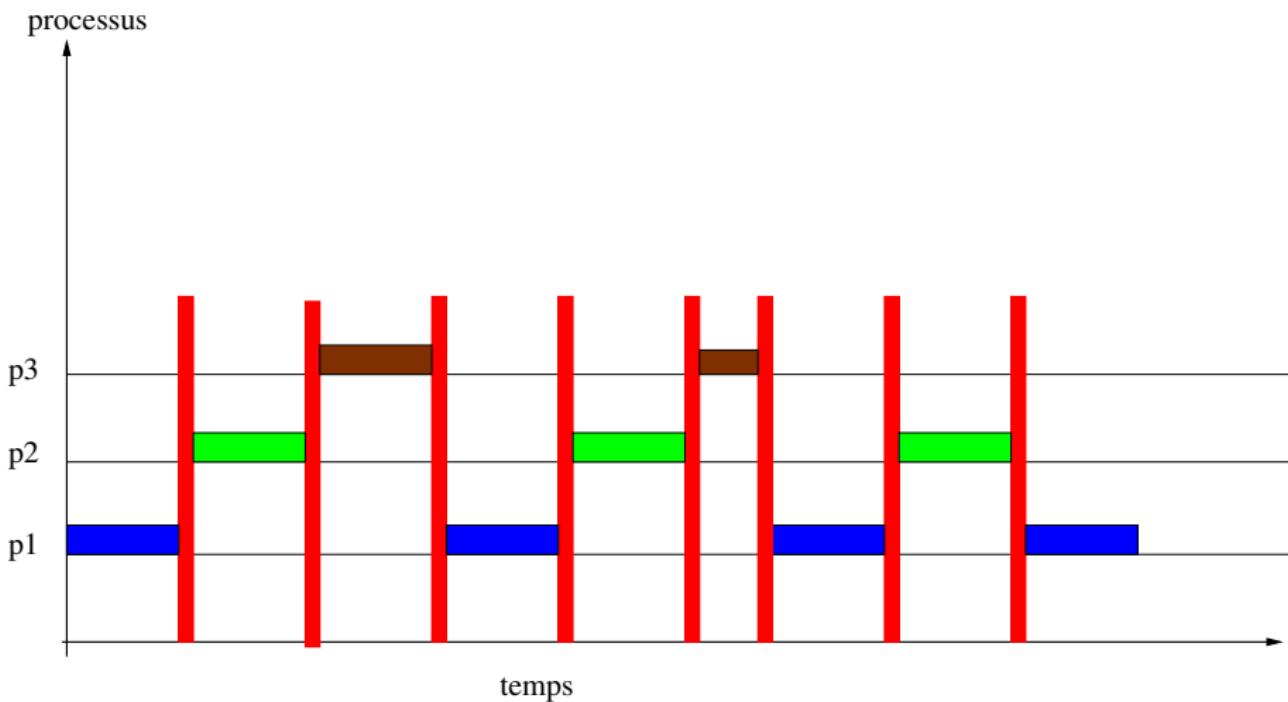
Chaque processus possède un quantum d'exécution :

- Si le processus a fini dans cet intervalle : au suivant !
- S'il n'a pas fini : le processus passe en fin de liste, et au suivant !

Le passage d'un processus à un autre (**commutation de tâche**) a un coût...
Problème : réglage du quantum par rapport au temps de commutation

- Quantum trop petit : le processeur passe son temps à commuter
- Quantum trop grand : augmentation du temps de réponse d'une commande (même simple)
- Réglage correct : $\text{Quantum}/\text{commutation} = 5$

Tourniquet : exemple



Ordonnancement avec priorité

Inconvénient du tourniquet = processus de même priorité !

Ordonnancement avec priorité :

- Plusieurs files d'attente plus ou moins prioritaires
- La priorité d'un processus peut décroître au cours du temps pour ne pas bloquer les autres files d'attente

Tourniquet priorité 1	P1	P2	P9	P8
Tourniquet priorité 2	P7	P4		
Tourniquet priorité 3	P3	P5	P6	

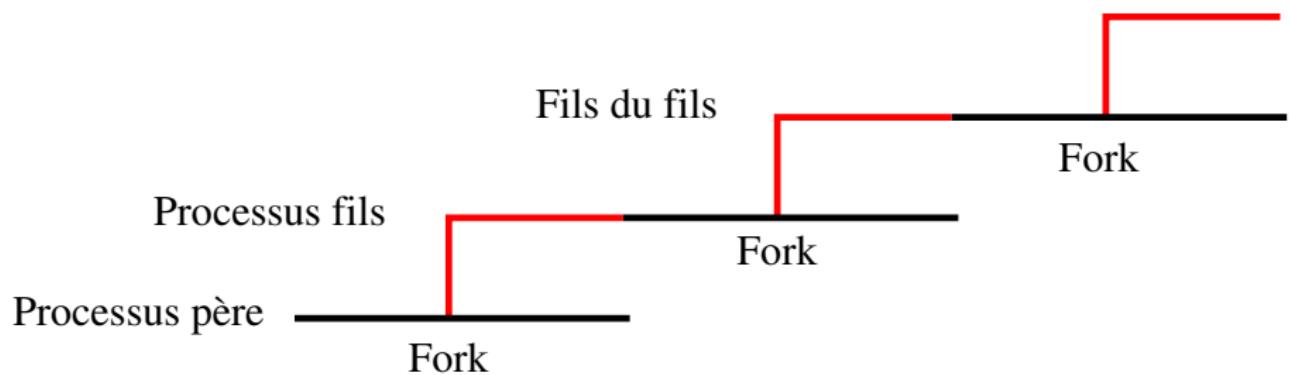
3 formes de relations

On peut donner 3 grandes formes de relations entre processus :

- Les **relations de parenté** : Chaque processus est créé par un processus appelé **père**. Il existe donc des relations de parenté entre processus
- Des **relations d'échange** : Les processus échangent très souvent des informations entre eux. On appelle cela la **communication inter-processus** (IPC : Inter Process Communication).
- Les **relations d'attente** : un processus attend un autre, un processus attend n secondes, un processus attend tous ses fils ...

On verra toutes les méthodes d'échange dans le cours portant sur la synchronisation (le prochain cours) et dans celui sur les IPC (Inter Process Communication (signaux, tubes, etc)).

Parenté



<http://www.luc-damas.fr/humeurs/the-fork-is-strong-in-your-family/>

Attente

Plusieurs processus veulent accéder à une ressource exclusive (c-à-d ne pouvant être utilisée que par un seul à la fois) :

- Processeur (cas du pseudo-parallélisme)
- Mémoire
- Imprimante, carte son

⇒ Une solution possible parmi d'autres : FCFS : premier arrivé, premier servi (les suivants attendent leur tour).

Plusieurs processus collaborent à une tâche commune. Souvent, ils doivent se synchroniser :

- p1 produit un fichier, p2 imprime le fichier
- p1 met à jour un fichier, p2 consulte le fichier
- La synchronisation se ramène à : p1 doit attendre que p2 ait franchi un certain point de son exécution. **Pour l'instant, on supposera que ce point est la terminaison, mais il existe des méthodes plus efficaces.**

Attendre

Faire attendre un processus

- `sleep(n)` : se bloquer pendant n secondes
- `wait` et `waitpid` : attendre la mort d'un de ses fils.
- `pause()` : se bloquer jusqu'à la réception d'un signal (cf. plus tard)

Processus Unix

État d'un processus

Mémoire virtuelle et contexte d'exécution (pile et registres CPU)

Identifiants de processus Unix

- PID (Processus Id)
- PPID (Parent Processus Id)

Types de processus

- Processus noyau : exécution de tâches d'administration (migrations, gestion des interruptions, ...)
- Processus systèmes (*daemons*) : exécution de tâches générales, souvent contrôlées par root
- Processus utilisateurs

les modes de manipulation

Il existe 2 façons différentes de manier les processus :

- Soit via un ensemble de commandes utilisable via un shell. Utile surtout pour manipuler les différentes exécutions de programmes en cours, les stopper, les dupliquer, observer leur comportement.
- Soit via une API. Utilisation beaucoup plus large. On peut souvent avoir envie de lancer plusieurs exécutions parallèles pour ne pas attendre par exemple la fin d'exécution d'une tâche longue dans un programme. **Il existe un autre moyen de manipuler différents fils d'exécutions qui est utilisé plus souvent, les processus légers (ou *threads*), que nous verrons plus tard.**

Spécificités Unix

2 modes d'exécution depuis le shell :

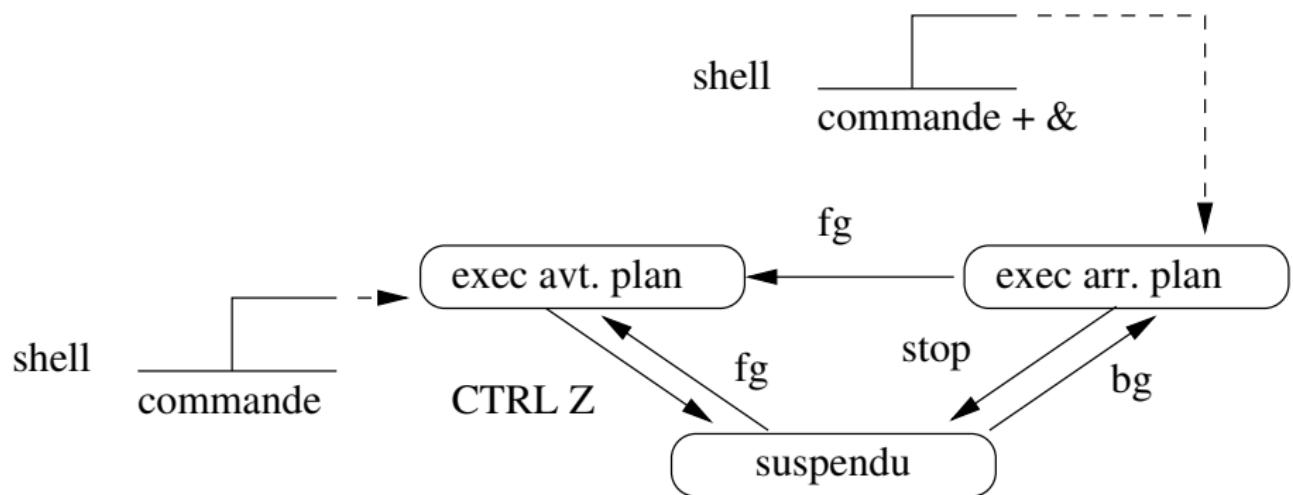
Interactif (foreground)

- Le plus fréquent : on tape une commande et on attend un résultat
- Interruption de la commande par CTRL+C
- Suspension de la commande par CTRL+Z

Arrière-plan (background)

- commande &
- La commande est lancée, mais on rend le contrôle à l'utilisateur
- Le lien avec le tty de la console est coupé.

Shell : manipulation de tâches



Ligne de commande : tuer, créer, mettre en arrière-plan...

- kill : permet de tuer un processus en donnant son PID. **On peut passer un numéro de signal à kill (cf plus tard). Pour insister : -9**
 - On ne peut arrêter que ses processus
 - Variante : killall (commande)
- & : lance en arrière plan
- bg : passe en arrière plan
- fg : passe en avant plan
- jobs : liste les jobs. Options :
 - jobs -p : liste seulement les PIDs.
 - jobs -l : seulement les jobs qui ont changé de statut
 - jobs -r : seulement les jobs en état d'exécution
 - jobs -s : seulement les jobs stoppés

Gérer les priorités

- **nice** : Exécuter un programme avec une priorité modifiée.
 - nice [OPTION] [COMMANDÉ [PARAM]...]
 - L'intervalle des valeurs possibles va de -20 (priorité la plus favorable) à 19 (la moins favorable).
- **renice** : Modifier la priorité des processus en cours d'exécution.
 - renice [-n] priorité [[-p] pid ...] [[-g] pgrp ...] [[-u] utilisateur...]
 - Les utilisateurs, autres que le superutilisateur, peuvent seulement modifier la priorité des processus dont ils sont propriétaires

Gérer les exécutions

À une date donnée / Exécution différée

- at : fichier des commandes exécuté à une date fixée
 - Pas d'interaction avec l'utilisateur
 - Envoi des résultats par email possible
- atq : liste les tâches en attente
- atrm : pour en effacer ...

File d'attente (batch)

- batch : la commande est placée dans une file d'attente
- La file d'attente est vidée en fonction de la charge du processeur
- Envoi des résultats par email possible

Périodiquement

- crontab : permet d'édition un fichier spécial contenant les tâches à exécuter régulièrement (par utilisateur)
- Permet de lancer une commande régulièrement (mois, jour, heure, minutes, etc ...)
- Un service scrute ce fichier à intervalle régulier

Observer les processus

- ps : affiche des renseignements sur une sélection de processus actifs.
 - Cliché instantané
 - beaucoup d'options
 - ps auxfw : infos longues, tous les processus
 - ps -jH : Arborescence de tous les processus descendant du shell courant
- top : vision dynamique des renseignements sur les processus actifs
 - Quelques options. top par défaut est très lisible
 - Utile pour voir quels sont les processus gourmands.

L'API

```
#include <unistd.h> /* _exit, fork */  
#include <stdlib.h> /* exit */
```

Les principales commandes

- `fork()` : créer un processus fils
- `wait()` et `waitpid()` : attendre un processus fils
- `exit()` : terminer le processus courant
- `getpid()` et `getppid()` : retourne le pid et le ppid
- `execl()` : exécute un programme

Par le détail

- `pid_t fork()` : le processus appelant se clone
 - Retourne 0 dans le processus fils
 - Retourne le PID du fils dans le père
 - -1 en cas d'erreur (trop de processus, pas assez de mémoire, ...)
- `void exit(int status)` : arrêter le processus courant.
- `pid_t wait(int *status)` et `pid_t waitpid(pid_t pid, int *status, int options)`
 - Attendre la mort d'un de ses fils
 - `wait(NULL)` donne le fonctionnement ci-dessus
 - `wait(&status)` affectera la valeur de sortie choisie par le fils.

Par le détail - 2

- `pid_t getpid()` et `pid_t getppid()`
 - Renvoie le pid du processus appelant ou de son père
 - -1 en cas d'erreur
- `int execl(const char *path, const char *arg, ..., NULL)`
- `int execv(const char *path, char *const argv[])`
 - path : chaîne de caractères donnant le nom du fichier qui contient le programme à exécuter
 - arg0 : nom du fichier (sans répertoire)
 - arg[1-n] : paramètres du programme
 - La liste des paramètres doit se terminer par un NULL
 - Exemple : `execlp("ls", "ls", "*.*", NULL);`
 - Variantes : `execvpe`, `execvp`, `execle`, `execlp`
 - Spécifier l'environnement : paramètre additionnel `char *const envp[]`
 - Recherche l'exécutable demandé dans les répertoires spécifiés par PATH

Exemples d'utilisation -1

Créer un processus fils et afficher :

- Dans le fils : son PID et le PID du père
- Dans le père : son PID

```
if ((n=fork())<0) {  
    perror("fork");  
    exit();  
}  
if (n!=0)  
    printf("Je suis le processus père %d de %d\n", getpid(), n);  
else  
    printf("Je suis le fils de %d, mon pid est le %d\n",  
           getppid(), getpid());
```

Exemples d'utilisation -2

```
for (i=1; i<=4; i++) {  
    pid = fork();  
    if (pid == 0)  
        printf("%d\n", getpid());  
}
```

Combien de processus ?

Exemples d'utilisation -2

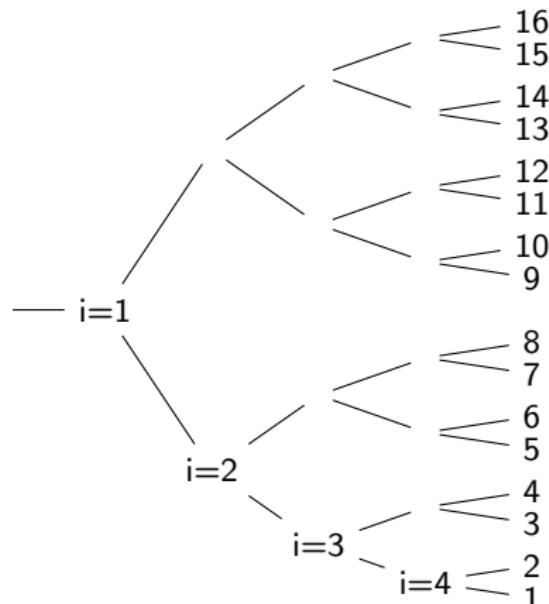
```
for (i=1; i<=4; i++) {
    pid = fork();
    if (pid = 0)
        printf("%d\n", getpid());
}
```

Combien de processus ?

Chaque processus en crée un deuxième :

$$2 * 2 * 2 * 2 = 2^4 = 16$$

15 pid affichés par les fils + le processus père initial



Exemples d'utilisation -3

```
for (i=1; i<=4; i++)
{
    pid = fork();
    if (pid == 0)
        break;
    else
        printf("%d\n", pid);
}
```

Que fait ce code ?

Exemples d'utilisation -3

```
for (i=1; i<=4; i++)
{
    pid = fork();
    if (pid == 0)
        break;
    else
        printf("%d\n", pid);
}
```

Que fait ce code ? Le fils avorte à chaque lancement de processus.

Exemples d'utilisation -4

Comment lancer une commande externe et reprendre son exécution après ?

Exemples d'utilisation -4

Comment lancer une commande externe et reprendre son exécution après ?

```
pid_t pid;  
if (!(pid = fork())) {  
    execl("/bin/ls", "ls", "-l", "/tmp", NULL);  
} else {  
    waitpid(pid, NULL, 0)  
};
```

Unix - e2i5 - Communication entre processus

Nicolas Palix

Polytech

2017

1 IPC

2 Signaux

- Définition
- Manipulation des signaux
 - Comment manipuler les signaux
 - Ligne de commande
 - API

3 Tubes

- Présentation
- Manipulation des tubes

4 Primitives de recouvrement

- Exemple d'utilisation

Définition

- Un processus est un fil d'exécution qui s'exécute sur une machine
- Plusieurs processus cohabitent simultanément
- Les processus se partagent les ressources de la machine (périphériques, CPU, fichiers ...)

Définition

- Un processus est un fil d'exécution qui s'exécute sur une machine
- Plusieurs processus cohabitent simultanément
- Les processus se partagent les ressources de la machine (périphériques, CPU, fichiers ...)

Pourquoi interagir avec les processus ?

- Les processus interagissent entre eux et doivent donc pouvoir communiquer entre eux

Définition

- Un processus est un fil d'exécution qui s'exécute sur une machine
- Plusieurs processus cohabitent simultanément
- Les processus se partagent les ressources de la machine (périphériques, CPU, fichiers ...)

Pourquoi interagir avec les processus ?

- Les processus interagissent entre eux et doivent donc pouvoir communiquer entre eux
- Le système doit pouvoir avertir les processus en cas de défaillance d'un composant du système

Définition

- Un processus est un fil d'exécution qui s'exécute sur une machine
- Plusieurs processus cohabitent simultanément
- Les processus se partagent les ressources de la machine (périphériques, CPU, fichiers ...)

Pourquoi interagir avec les processus ?

- Les processus interagissent entre eux et doivent donc pouvoir communiquer entre eux
- Le système doit pouvoir avertir les processus en cas de défaillance d'un composant du système
- L'utilisateur doit pouvoir gérer les processus (arrêt, suspension, ..)

Définition

- Un processus est un fil d'exécution qui s'exécute sur une machine
- Plusieurs processus cohabitent simultanément
- Les processus se partagent les ressources de la machine (périphériques, CPU, fichiers ...)

Pourquoi interagir avec les processus ?

- Les processus interagissent entre eux et doivent donc pouvoir communiquer entre eux
- Le système doit pouvoir avertir les processus en cas de défaillance d'un composant du système
- L'utilisateur doit pouvoir gérer les processus (arrêt, suspension, ..)

Les méthodes de communications entre processus sont souvent désignés par l'acronyme IPC : Inter-Process Communications

IPC

Il existe de multiples moyens de réaliser une communication inter-processus :

- Par fichiers (\simeq tous les systèmes)
- **Signaux** (Unix/Linux/MacOS, pas vraiment sous Windows)
- Sockets (\simeq tous les systèmes)
- Files d'attente de messages (\simeq tous les systèmes)
- **Pipe/tubes** (tous les systèmes POSIX, Windows)
- **Named pipe/tubes nommés** (tous les systèmes POSIX, Windows)
- Sémaphores (tous les systèmes POSIX, Windows)
- Mémoire partagée (tous les systèmes POSIX, Windows)
- Passage de messages : MPI, RMI, CORBA ...
- ...

Aujourd'hui on s'intéresse aux signaux et aux tubes.

1 IPC**2** Signaux

- Définition
- Manipulation des signaux
 - Comment manipuler les signaux
 - Ligne de commande
 - API

3 Tubes

- Présentation
- Manipulation des tubes

4 Primitives de recouvrement

- Exemple d'utilisation

Signaux

Définition

Un signal : moyen de communication indiquant une action à entreprendre à partir de conventions préétablies.

- Différents signaux pour différentes actions préétablies
- Conventions :
 - On peut récupérer les signaux et définir l'action via des *handlers*
 - Il existe des actions prédéfinies

Signaux

Définition

Un signal : moyen de communication indiquant une action à entreprendre à partir de conventions préétablies.

- Différents signaux pour différentes actions préétablies
- Conventions :
 - On peut récupérer les signaux et définir l'action via des *handlers*
 - Il existe des actions prédéfinies

Caractéristiques - Généralités

- On ne connaît pas l'émetteur
- Chaque signal est identifié par un numéro et un nom symbolique définis dans **signal.h**
- Les signaux ont des noms qui supposent un événement, mais rien ne permet de savoir si il s'est *réellement* produit.

Caractéristiques - Vocabulaire

États des signaux

- Un signal **pendant** (*pending*) est un signal envoyé mais pas encore pris en compte
- Un signal est **délivré** lorsqu'il est pris en compte par le processus qui le reçoit
- On peut dans certaines versions d'Unix différencier volontairement la délivrance de certains signaux : les signaux peuvent être dans l'état **bloqué** ou **masqué**.

Caractéristiques - Vocabulaire 2

Types de signaux

- Interruption : événement extérieur au processus
 - Frappe au clavier
 - Signal déclenché par programme : primitive kill, ...
- Déroutement : événement intérieur au processus généré par le matériel
 - FPE (floating point error)
 - Violation mémoire ...

Liste des signaux (kill -l)

SIGABRT	terminaison anormale du processus.
SIGALRM	alarme horloge
SIGFPE	erreur arithmétique
SIGHUP	rupture de connexion
SIGILL	instruction illégale
SIGINT	interruption terminal
SIGKILL	terminaison impérative. Ne peut être ignoré ou intercepter
SIGPIPE	écriture dans un conduit sans lecteur disponible
SIGQUIT	signal quitter du terminal
SIGSEGV	accès mémoire invalide
SIGTERM	signal 'terminer' du terminal
SIGUSR1	signal utilisateur 1
SIGUSR2	signal utilisateur 2
SIGCHLD	processus fils stoppé ou terminé
SIGCONT	continuer une exécution interrompue
SIGSTOP	interrompre l'exécution. Ne peut être ignoré ou intercepter
SIGTSTP	signal d'arrêt d'exécution généré par le terminal
SIGTTIN	processus en arrière plan essayant de lire le terminal
SIGTTOUT	processus en arrière plan essayant d'écrire sur le terminal
SIGBUS	erreur accès bus
SIGPOLL	événement interrogable
SIGPROF	expiration de l'échéancier de profilage
SIGSYS	appel système invalide
SIGTRAP	point d'arrêt exécution pas à pas
SIGURG	donnée disponible à un socket avec bande passante élevée
SIGVTALRM	échéancier virtuel expiré
SIGXCPU	quota de temps CPU dépassé
SIGXFSZ	taille maximale de fichier dépassée

Nota : liste incomplète !



Signaux particuliers

Certains signaux ont des statuts particuliers :

- SIGKILL ne peut pas être intercepté, bloqué ou ignoré. Cela permet de tuer un signal même en cas de processus récalcitrant.
- SIGSTOP est dans le même cas, pour pouvoir stopper un processus (stopper pour reprendre plus tard, pas arrêter)
- SIGCONT ne peut être pris en charge par un *handler*. Pourquoi ? SIGCONT est envoyé à un processus pour lui faire reprendre son exécution (après un SIGSTOP). Il est donc logique qu'il ne puisse être pris en charge par un *handler*...

Émetteurs d'un signal

3 entités peuvent émettre un signal :

- Un processus : pour se coordonner avec les autres, pour gérer une exécution multi-processus plus ou moins complexe (par exemple du *branch-and-bound*).
- Le système (souvent pour avertir d'une défaillance/interruption matérielle)
- L'utilisateur pour gérer ses tâches. Remarque : quand on tape au clavier CTRL+C, ce n'est pas le clavier, mais **l'interpréteur de commande** qui traduit ce caractère comme un signal. Pour vous en souvenir : dans un éditeur de texte, CTRL+C ne tue pas l'éditeur !

Traitement des signaux

À chaque type de signal est associé un handler par défaut. Le plus commun est la terminaison du programme.

- Les 5 traitements par défaut disponibles sont :
 - Terminaison du processus (macro SIGDFL)
 - Terminaison du processus avec image mémoire : fichier core (action lancé lors d'un SIGQUIT - pas de macro définie)
 - Signal ignoré (sans effet) (macro SIGIGN)
 - Suspension du processus (SIGSTOP, pas de macro définie)
 - Continuation : reprise d'un processus stoppé (SIGCONT, pas de macro définie)
- Un processus peut ignorer un signal en lui associant le handler SIGIGN.
- Les signaux SIGKILL, SIGCONT et SIGSTOP ne peuvent avoir que le handler SIGDFL.
- SIGDFL et SIGIGN sont les deux seules macros prédéfinies.

Réception d'un signal par un processus actif

- Si le processus exécute un programme utilisateur : traitement immédiat du signal reçu,
- S'il se trouve dans une fonction du système (*system call*) : le traitement du signal est différé jusqu'à ce qu'il revienne en mode utilisateur (*user mode*), c'est à dire lorsqu'il sort de cette fonction.

1 IPC

2 Signaux

- Définition
- **Manipulation des signaux**
 - Comment manipuler les signaux
 - Ligne de commande
 - API

3 Tubes

- Présentation
- Manipulation des tubes

4 Primitives de recouvrement

- Exemple d'utilisation

Comment manipuler les signaux

On doit pouvoir manipuler les signaux, c'est à dire :

- Pouvoir envoyer un signal spécifique à un processus donné (ou un groupe de processus)
- Pouvoir définir les *handlers* mis en place pour prendre en charge chacun des signaux reçus par chacun des processus

Comment manipuler les signaux

On doit pouvoir manipuler les signaux, c'est à dire :

- Pouvoir envoyer un signal spécifique à un processus donné (ou un groupe de processus)
- Pouvoir définir les *handlers* mis en place pour prendre en charge chacun des signaux reçus par chacun des processus

Il existe 2 moyens de manipuler les signaux :

- Par la ligne de commande
- Par une API

Gérer les processus en ligne de commande

bash

- CTRL+C : SIGINT
- CTRL+Z : SIGSTOP
- kill : envoi un signal spécifique (cf cours précédent)
- suspend : envoie un signal SIGSTOP, et attend un signal SIGCONT pour reprendre
- trap [-lp] [arg] [sigspec ...] : utilise le programme arg quand un signal SIGSPEC est reçu par le shell.
- bg, fg, & ... passent tous par des signaux.

Remarques sur la manipulation en ligne de commande

- Pas tout à fait standardisé (d'où : bash)
- La liste des raccourcis claviers qui correspondent à des signaux est configurable, extensible et modifiable via **stty**. **stty** gère la configuration de la ligne de terminal (paramètres spéciaux, d'entrée, de sortie, de contrôle, locaux... , caractères spéciaux). **Utilisation dangereuse et malaisée ...**

Vue d'ensemble de l'API de manipulation des signaux

Deux principales commandes :

- kill pour envoyer des signaux,
- signal pour définir une action correspondante à un signal donné.

Vue d'ensemble de l'API de manipulation des signaux

Deux principales commandes :

- kill pour envoyer des signaux,
- signal pour définir une action correspondante à un signal donné.

Et tout un tas d'autres :

- pause pour attendre un signal
- sigsuspend pour attendre un signal masqué auparavant
- sigemptyset pour vider la liste des signes masqués, sigaddset pour en ajouter ...
- ...

Il existe 2 API, l'une POSIX et l'autre non.

Tuer un processus : primitive kill

La primitive kill permet d'émettre un signal *sig* vers un ou plusieurs processus :

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

pid représente :

- Si $pid > 0$, le pid du processus
- Si $pid = 0$, tous les processus dans le même groupe que le processus émetteur
- Si $pid = -1$, non définie dans POSIX
- Si $pid < -1$, tous les processus du groupe *pid*

sig est le numéro du signal que l'on désire envoyer. Quand *sig* est égal à 0, permet de tester l'existence d'un processus.

ATTENTION : cet appel système est particulièrement mal nommé, car il ne tue que très rarement un processus ...

Exemple d'envoi d'un signal

```
1 #include <sys/types.h>
2 #include <unistd.h>
3 #include <sys/wait.h>
4 #include <signal.h>
5
6 int main(int argc, char **argv) {
7     pid_t p;
8     int etat;
9
10    if ((p=fork()) == 0) {
11        /* Child process wait forever */
12        while (1);
13        exit(2);
14    }
15
16    /* Father process */
17    sleep(10);
18    printf("send SIGUSR1 signal to the child %d\n", p);
19    kill(p, SIGUSR1);
20    // Father wait until the end of its child.
21    p = waitpid(p, &etat, 0);
22    printf("Child %d exit status: %d\n", p, etat >> 8);
23    return (0);
24 }
```

Mise en place d'un handler

Les signaux (autres que SIGKILL, SIGCONT et SIGSTOP) peuvent avoir un handler spécifique installé par un processus :

- La primitive signal() fait partie du standard de C et non de la norme de POSIX :

```
1 #include <signal.h>
2 void (*signal (int sig, void (*p_handler)(int)))(int);
```

- Elle installe le handler spécifié par p_handler pour le signal sig. La valeur renvoyée est un pointeur sur la valeur ancienne du handler.
- La fonction de handler est exécutée par le processus à la délivrance d'un signal. Cette fonction ne retourne pas de valeur (variable globale ...). Elle reçoit le numéro du signal. À la fin de l'exécution de cette fonction, l'exécution du processus reprend au point où elle a été suspendue.
- Quelques remarques :
 - Un processus endormi est réveillé par l'arrivée d'un signal
 - Les signaux sont sans effet sur un processus zombie

Remarque : POSIX vs non POSIX

Blocage des signaux

La norme POSIX définit un ensemble de fonctions pour la manipulation des signaux (Voir documentation : sigprocmask, sigpending,...)

Installation de la fonction de handler

- POSIX : structure sigaction à remplir, puis fonction sigaction()
- Structure sigaction dépend des versions de Linux ... (ne jamais l'initialiser de façon statique ...)
- Non POSIX (mais tellement plus simple) : signal(...) vu précédemment

Préférer la version POSIX si vous devez gérer des signaux dans une application portable (mais plus difficile à mettre en œuvre).

Exemple de mise en place handler

```
1 #include <stdio.h>
2 #include <signal.h>
3
4 void hand(int signum) {
5     printf("Press Ctrl-C\n");
6     printf("Will stop next time.\n");
7     signal(SIGINT, SIG_DFL);
8 }
9
10 int main(int argc, char **argv) {
11     signal(SIGINT, hand);
12     for (;;) { }
13     return 0;
14 }
```

Exemple de mise en place handler : déroutement

```
1 #include <signal.h>
2 #include <stdio.h>
3
4 void Hand_sigfpe(int sig) {
5     printf("\nErreur division par 0 !\n");
6     exit(1);
7 }
8
9 void main(int argc, char **argv) {
10    int a, b, r;
11    signal(SIGFPE, Hand_sigfpe);
12    printf("Taper a : "); scanf("%d", &a);
13    printf("Taper b : "); scanf("%d", &b);
14    r = a/b;
15    printf("La division de a par b = %d\n", r);
16 }
```

Autres primitives : pause

La primitive pause() bloque en attente le processus appelant jusqu'à l'arrivée d'un signal.

```
1 #include <unistd.h>
2 int pause (void);
```

- À la prise en compte d'un signal, le processus peut :
 - Se terminer car le handler associé est SIGDFL ;
 - Passer à l'état stoppé ; à son réveil, il exécutera de nouveau pause() et s'il a été réveillé par SIGCONT ou SIGTERM avec le handler SIGIGN, il se mettra de nouveau en attente d'arrivée d'un signal ;
 - Exécuter le handler correspondant au signal intercepté.
- pause() ne permet pas d'attendre un signal de type donné ni de connaître le nom du signal qui l'a réveillé.

Exemple d'attente

```
1 #include <stdio.h>
2 #include <signal.h>
3 #include <unistd.h>
4
5 int nb_req=0;
6 int pid_fils, pid_pere;
7
8 void Hand_Pere(int sig) {
9     nb_req++;
10    printf("Parent: handling request %d of the child.\n",
11          nb_req);
12 }
13
14 int main() {
```

Exemple d'attente - suite

```
14 int main() {
15     if ((pid_fils = fork()) == 0) { /* Child */
16         pid_pere = getppid();
17         /* Wait for the parent to activate its handler and pause */
18         sleep(2);
19         for (i=0; i<10; i++) {
20             printf("Child send a signal to its parents.\n");
21             kill (pid_pere, SIGUSR1); /* Service requested */
22         }
23         exit(0);
24     }
25     else { /* Parent */
26         signal(SIGUSR1, Hand_Pere);
27         while(1) {
28             pause(); /* Wait for a service request */
29             sleep(5); /* Serve */
30         }
31     }
32     return(0);
33 }/* END OF MAIN */
```

Remarques supplémentaires

- Il faut généralement réinstaller le handler à la délivrance d'un signal pour prendre en compte le suivant (suivant les versions du SE).
- Une fonction de handler doit rester courte
- Installer le handler avant l'arrivée du signal
- Un processus peut se mettre en attente de réception d'un signal :
 - Non POSIX : pause(). La fonction pause ne permet ni d'attendre l'arrivée d'un signal de type précis, ni de savoir quel signal arrive (récupérer la valeur du paramètre dans la fonction de handler)
 - POSIX : sigsuspend()
- Fonction particulière :

```
unsigned int alarm(unsigned int secondes);
```

Génère le signal SIGALRM après un nombre de secondes passé en paramètre à la fonction alarm

1 IPC

2 Signaux

- Définition
- Manipulation des signaux
 - Comment manipuler les signaux
 - Ligne de commande
 - API

3 Tubes

- Présentation
- Manipulation des tubes

4 Primitives de recouvrement

- Exemple d'utilisation

Tubes ?

Définition : tube

Mécanisme de communication unidirectionnel.

Caractéristiques

- Possède deux extrémités, une pour y lire et l'autre pour écrire
- La lecture dans un tube est destructrice : les données lues sont supprimées du tube
- Les tubes permettent la communication d'un flot continu de caractères (mode stream)
- Un tube a une capacité finie
- La gestion des tubes se fait en mode FIFO
- Il existe 2 types de tubes : les tubes ordinaires, et les tubes nommés.

Tubes : Fonctionnement

- À un tube est associé un nœud du système de gestion de fichiers (son compteur de liens est = 0 car aucun répertoire ne le référence pour un tube ordinaire).
 - Le tube sera supprimé et le nœud correspondant libéré lorsque plus aucun processus ne l'utilise.
 - Un tube ordinaire n'a pas de nom

Tube ordinaire : accès

- Impossible pour un processus d'ouvrir un tube ordinaire à l'aide de la fonction `open()`
- Il faut donc connaître le descripteur associé à ce fichier

L'existence d'un tube correspond à la possession d'un descripteur qui peut être acquis de deux manières :

- Un appel à la primitive de création de tube `pipe` ;
- Par héritage : un processus fils hérite de son père des descripteurs de tubes, entre autres : un tube est limité aux processus issus de la même descendance du créateur du tube ordinaire

Tube nommé : accès

- Ils font partie de la norme POSIX sous le nom de fifo
- Ils ont toutes les propriétés des tubes ordinaires, avec en plus :
 - Ils ont une référence dans le système de fichiers
 - Un processus connaissant cette référence peut récupérer un descripteur via la fonction open
- Autorisent la communication entre différents processus.

Interfaces

Il existe plusieurs possibilités pour utiliser des pipe. Des interfaces en shell (Le "|/" est un pipe ordinaire) et dans beaucoup de langages (dont C).

- Les interfaces entre tubes nommés et tubes ordinaires diffèrent ...
Mais le fonctionnement peu :
 - On récupère un descripteur : soit en lecture, soit en écriture.
 - Suivant le type de descripteur que l'on possède, on lit ou on écrit dedans
 - Si il n'y a pas de lecteur pour le tube et que l'on écrit :
 - Un SIGPIPE est envoyé (pipe ordinaire)
 - Le SIGSTOP est envoyé (pipe nommé, le processus est donc suspendu jusqu'à ce qu'un lecteur arrive (SIGCONT))
- En C, il faut inclure `unistd.h` pour les tubes (nommés ou non).

Manipulation des tubes ordinaires : création

La fonction pipe() permet de créer un tube ordinaire :

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

Alloue un nœud sur le disque et crée 2 descripteurs dans la table du processus appelant. Le tableau de 2 entiers est passé en paramètre et contient au retour ces descripteurs.

- p[0] pour la sortie – lecture
- p[1] pour l'entrée – écriture

Manipulation des tubes ordinaires : lecture - algorithme

Algorithme :

- Si le tube n'est pas vide, on extrait au plus TAILLE_BUFS caractères qui sont placés à l'adresse buf
- Si le tube est vide
 - Si le nombre d'écrivains est nul,
la fin du fichier est atteinte ($nb_lu = 0$)
 - Si le nombre d'écrivains n'est pas nul,
 - Si la lecture est bloquante (par défaut), le processus est mis en sommeil jusqu'à ce que le tube ne soit plus vide
 - Si la lecture est non bloquante (cf fonction fcntl),
retour immédiat avec -1

Risque d'auto-blocage ou d'inter-blocage : Ne conserver que les descripteurs utiles. Fermer systématiquement tous les autres !

Manipulation des tubes ordinaires : lecture - exemple

```
1 #include <stdio.h>
2 #include <unistd.h>
3
4 #define TAILLE_BUF 50
5
6 int main(int argc, char **argv) {
7     int p[2];
8     int nb_lu;
9     char buf[TAILLE_BUF];
10
11    if (pipe(p) == -1)
12        /* Error handling */
13
14    close(p[1]);
15
16    nb_lu = read(p[0], buf, TAILLE_BUF);
17
18
19    return 0;
20}
```

Manipulation des tubes ordinaires : écriture - algorithme

Algorithme

- Si le nombre de lecteurs dans le tube est nul, le signal SIGPIPE est délivré au processus écrivain (par défaut, il termine le processus . . .).
- Si le nombre de lecteurs est non nul
 - Si l'écriture est bloquante (cf fonction fcntl), le retour n'a lieu que si n caractères ont été écrits (écriture atomique si $n < \text{PIPE_BUF}$ définie dans <limits.h>). Le processus est donc susceptible de s'endormir.
 - Si l'écriture est non bloquante
 - Si $n > \text{PIPE_BUF}$, le retour est inférieur à n
 - Si $n \leq \text{PIPE_BUF}$, l'écriture atomique est réalisée (si tube non plein)

Manipulation des tubes ordinaires : écriture - exemple

```
1 int main(int argc, char **argv) {
2     int tube[2];
3     int n;
4     char buf[50];
5
6     if (pipe(tube) == -1) {
7         printf("Erreur de pipe !!!\n");
8         exit(0);
9     }
10    ... /* Fork - Child creation */
11    {
12        /* Parent code */
13        close(tube[0]);
14        ...
15        n = strlen(buf) + 1;
16        write(tube[1], buf, n);
17        ...
18    }
19    ...
20 }
```

Manipulation des tubes nommés : création

Dans <sys/types.h> et <sys/stat.h>

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

- `ref` indique le chemin d'accès au fichier,
- `mode` est construit comme pour la fonction `open`.

Manipulation des tubes nommés : lecture et écriture

Écriture :

```
1 int main(int argc, char **argv) {
2     char s[30] = "bonjour!!!";
3     int num, fd;
4
5     if ( mkfifo("fifo", 0666) == -1)
6         perror("mkfifo");
7     fd = open("fifo", O_WRONLY);
8     ...
9     write(fd, s, strlen(s));
10    ...
11 }
```

Lecture :

```
1 char s[30];
2 int num, fd;
3
4 if ( mkfifo("fifo", 0666) == -1)
5     perror("mkfifo");
6 fd = open("fifo", O_RDONLY);
7 ...
8 read(fd, s, 30);
9 ...
10 }
```

Définition

- Primitives de recouvrement : `exec*`
- Ensemble de fonctions permettant à un processus de charger en mémoire un nouveau programme binaire en vue de son exécution :
 - Remplacement de l'espace d'adressage du processus par un nouveau programme (segment texte, données, pile, tas)
 - Pas de retour de recouvrement, sauf si il n'a pas pu avoir lieu
 - Pas de création de processus
 - Lance le nouveau programme

L'utilisation des pipes ordinaires ne peut se faire qu'entre membres d'une même descendance de processus, en héritant des descripteurs. Qu'en est-il quand on exécute un `execv` ?

Déroulement de l'exécution d'une primitive de recouvrement

Déroulement d'un tel appel système :

- Recherche du fichier à exécuter (utilisation de la variable PATH)
- Vérification des droits d'accès en exécution de l'utilisateur
- Vérification que le fichier est exécutable (nombre magique)
- Sauvegarde des paramètres de l'appel à la fonction de recouvrement
- Création du nouvel espace d'adressage (texte, données, pile, tas)
- Restitue les paramètres précédemment sauvegardés
- Initialisation du contexte matériel

Les différentes primitives de recouvrement

execv, execvp, execvpe, execl, execlp, execle

- execve : la primitive appelée par toutes les autres
- "l" : liste, "v" : vecteur. liste : arguments séparés dans l'appel de la commande, vecteur : un pointeur vers un tableau d'arguments.
- "l" : premier argument doit être le nom de la commande. Si on utilise un vecteur, le premier élément du vecteur doit être le nom de la commande aussi ...
- "e" final : environnement. Quand il y a le "e" permet de passer un tableau de chaînes de char vers des variables d'environnements. Sans, l'environnement est la copie de celui de l'appelant.
- "p" final : path (chemin) . Quand il y a un "p", il y a un argument qui peut être un nom de fichier ("p" parce que l'on peut utiliser la variable PATH). Si il contient un "/" alors l'argument est considéré comme un chemin, sinon on cherche dans le PATH l'exécutable.

Exemples :

```
1  execlp("ls", "ls", "-l", "-a", NULL);
2  execl("/bin/ls", "ls", "-l", "-a", NULL);
3  char *mes_arg[4] = {"ls", "-l", "-a", NULL};
4  execv("/bin/ls", mes_arg);
```

Duplication des descripteurs

La duplication des descripteurs permet à un processus d'acquérir un nouveau descripteur (dans sa table de descripteurs) synonyme d'un descripteur déjà existant :

- Ce mécanisme est principalement utilisé pour rediriger les E/S standard
- Ce mécanisme repose sur le fait que le descripteur dupliqué prendra toujours la place du plus petit descripteur libre dans la table.
- À la création d'un processus, la table des descripteurs contient 3 descripteurs :
 - TableDesc[0] : stdin
 - TableDesc [1] : stdout
 - TableDesc [2] : stderr

Duplication des descripteurs

On supprime une référence dans la table des descripteurs avec la fonction

```
int close(int fd);
```

Par exemple, pour supprimer la référence à stdin :

```
close(0);
```

La fonction dup (dans <unistd.h>) associe le descripteur au plus petit numéro disponible dans la table

```
int dup(int descripteur);
```

Par exemple, si p[0] est la sortie d'un tube ordinaire

```
close(0); dup(p[0]);
```

ce qui remplace l'entrée standard (le clavier) par la sortie du tube p.

La fonction dup2 force le descripteur1 à la place du descripteur2 (réalise un close(descripteur2 si besoin))

```
int dup2(int descripteur1, int descripteur2);
```

Exemple

```
1 int main(int argc, char **argv) {
2     ...
3     if (pid_fils == 0) { /* Child code */
4         close(0);
5         dup(tube[0]);
6         close(tube[1]);
7         close(tube[0]);
8         if (execlp("./affichage", "./affichage", 0) == -1)
9             printf("Erreur \u00e9xeclp\u00e9 !!!\n");
10        exit(0);
11    }
12    else { /* Parent code */
13        close(tube[0]);
14        strcpy(buf, "bonjour\u00e9 !!!\n");
15        write(tube[1], &buf, strlen(buf) + 1);
16        sleep(1);
17        kill(pid_fils, SIGKILL);
18        wait(0);
19    }
20    return 0;
21 }
```

Exemple - suite

```
1  /* affichage.c */
2  int main(int argc, char **argv) {
3      char c;
4
5      do {
6          scanf("%c", &c);
7          printf("%c", c);
8      } while(1);
9
10     return 0;
11 }
```

Unix - e2i5 - Synchronisation des processus : sémaphores et variables d'exclusion mutuelle

Nicolas Palix

Polytech

2017

1 Rappels

- Processus
- IPC

2 Synchronisation entre processus

- Problèmes de synchronisation
- Famine
- Inversion de priorité
- Les philosophes
- Section critique

3 Implémentation en C sous Linux

- Implémentation System V
- Implémentation POSIX

Définition

- Un processus est un fil d'exécution qui s'exécute sur une machine
- Plusieurs processus cohabitent simultanément, souvent liés entre eux par des liens de famille.
- Les processus se partagent les ressources de la machine (périphériques, CPU, fichiers ...)

Définition

- Un processus est un fil d'exécution qui s'exécute sur une machine
- Plusieurs processus cohabitent simultanément, souvent liés entre eux par des liens de famille.
- Les processus se partagent les ressources de la machine (périphériques, CPU, fichiers ...)

Pourquoi interagir avec les processus ?

- Les processus interagissent entre eux et doivent donc pouvoir communiquer entre eux
- Le système doit pouvoir avertir les processus en cas de défaillance d'un composant du système
- L'utilisateur doit pouvoir gérer les processus (arrêt, suspension, ..)

Définition

- Un processus est un fil d'exécution qui s'exécute sur une machine
- Plusieurs processus cohabitent simultanément, souvent liés entre eux par des liens de famille.
- Les processus se partagent les ressources de la machine (périphériques, CPU, fichiers ...)

Pourquoi interagir avec les processus ?

- Les processus interagissent entre eux et doivent donc pouvoir communiquer entre eux
- Le système doit pouvoir avertir les processus en cas de défaillance d'un composant du système
- L'utilisateur doit pouvoir gérer les processus (arrêt, suspension, ..)

Les méthodes de communications entre processus sont souvent désignés par l'acronyme IPC : Inter-Process Communications

IPC

Il existe de multiples moyens de réaliser une communication inter-processus :

- Par fichiers (\simeq tout système)
- *Signaux* (Unix/Linux/MacOS, pas vraiment sous Windows)
- Sockets (\simeq tout système)
- Files d'attente de message (\simeq tout système)
- *Pipe/tubes* (tout système POSIX, Windows)
- *Named pipe/tubes nommés* (tout système POSIX, Windows)
- Sémaphores (tout système POSIX, Windows)
- Mémoire partagée (tout système POSIX, Windows)
- Passage de message : MPI, RMI, CORBA ...
- ...

IPC

Il existe de multiples moyens de réaliser une communication inter-processus :

- Par fichiers (\simeq tout système)
- *Signaux* (Unix/Linux/MacOS, pas vraiment sous Windows)
- Sockets (\simeq tout système)
- Files d'attente de message (\simeq tout système)
- *Pipe/tubes* (tout système POSIX, Windows)
- *Named pipe/tubes nommés* (tout système POSIX, Windows)
- **Sémaphores** (tout système POSIX, Windows)
- Mémoire partagée (tout système POSIX, Windows)
- Passage de message : MPI, RMI, CORBA ...
- ...

Aujourd'hui on s'intéresse aux **sémaphores**.

Accès aux ressources

Types d'accès aux ressources

- Exclusif : un seul processus peut utiliser cette ressource à un instant donné (ex : une imprimante ne peut imprimer qu'un document à la fois).
- Simultané : quand on dispose de n ressources indifférenciées (ex : n imprimantes, n têtes de lectures sur un disque).

Accès aux ressources

Types d'accès aux ressources

- Exclusif : un seul processus peut utiliser cette ressource à un instant donné (ex : une imprimante ne peut imprimer qu'un document à la fois).
- Simultané : quand on dispose de n ressources indifférenciées (ex : n imprimantes, n têtes de lectures sur un disque).

Type et mode d'accès

- Lecture : souvent non exclusif (ex : n processus peuvent lire simultanément la même donnée).
- Écriture : écriture simultanée ainsi que lecture de la variable par d'autre processus **interdites**.

Accès : reposent sur **l'atomicité** des actions. Peuvent concerner des portions de code.

Outils pour la synchronisation des accès

Sémaphores

Mécanisme de **synchronisation** fourni par le système (Dijkstra, 1965), permet de gérer les accès de plusieurs processus à une ressource.

Outils pour la synchronisation des accès

Sémaphores

Mécanisme de **synchronisation** fourni par le système (Dijkstra, 1965), permet de gérer les accès de plusieurs processus à une ressource. 2 primitives pour les proc. (+ création et destruction) et 1 compteur :

- $P(x)$ (*Proberen*, Essayer/Prendre) et $V(x)$: (*Verhogen*, Augmenter/Libérer). x : id (clef) du sémaphore.
- Compteur (caché) dans le sémaphore i :
 - Initialisation : $n =$ nb. proc. pouvant utiliser la ressource simultanément
 - $i--$ à chaque fois qu'on autorise un processus à y accéder ($P(x)$). Si $i = 0$, on attend qu'un processus appelle $V(x)$. (appel bloquant).
 - $i++$ quand un processus libère la ressource ($V(x)$).

Outils pour la synchronisation des accès

Sémaphores

Mécanisme de **synchronisation** fourni par le système (Dijkstra, 1965), permet de gérer les accès de plusieurs processus à une ressource. 2 primitives pour les proc. (+ création et destruction) et 1 compteur :

- $P(x)$ (*Proberen*, Essayer/Prendre) et $V(x)$: (*Verhogen*, Augmenter/Libérer). x : id (clef) du sémaphore.
- Compteur (caché) dans le sémaphore i :
 - Initialisation : $n =$ nb. proc. pouvant utiliser la ressource simultanément
 - $i--$ à chaque fois qu'on autorise un processus à y accéder ($P(x)$). Si $i = 0$, on attend qu'un processus appelle $V(x)$. (appel bloquant).
 - $i++$ quand un processus libère la ressource ($V(x)$).

Cas particulier du sémaphore : verrous / exclusion mutuelle

- $n = 1$: **exclusion mutuelle** : 1 seul proc. utilise la ressource à la fois.
- Mécanisme appelé **verrou ou mutex** (*Mutual Exclusion*).

Problèmes de synchronisation

Problèmes courants

- **Interblocage (*deadlock*)** : plusieurs processus ont besoin de ressources **simultanément**, mais chacun n'en acquière qu'un sous-ensemble.
- **Famine** : 1 ou plusieurs processus n'ont pas accès à 1 ou n ressources car d'autre processus + prioritaires les verrouillent.
- **Inversion de priorité** : Un processus verrouille la ressource k , puis un processus + prioritaire l'interrompt et essaie de verrouiller k .

Problèmes de synchronisation

Problèmes courants

- **Interblocage (*deadlock*)** : plusieurs processus ont besoin de ressources **simultanément**, mais chacun n'en acquière qu'un sous-ensemble.
- **Famine** : 1 ou plusieurs processus n'ont pas accès à 1 ou n ressources car d'autre processus + prioritaires les verrouillent.
- **Inversion de priorité** : Un processus verrouille la ressource k , puis un processus + prioritaire l'interrompt et essaie de verrouiller k .

Solutions possibles (algorithmiques)

- **Interblocage (*deadlock*)** : Gérer intelligemment l'ordre d'acquisition
- **Famine** : compliqué à éviter, peut de temps à autre être résolu en utilisant **l'attente active**.
- **Inversion de priorité** : Éviter de partager des verrous entre processus de priorités différentes

Interblocage

Exemple

Soit P_1, P_2 des processus et R_1, R_2 des ressources.

- P_1 et P_2 ont tous les 2 besoin de R_1 et R_2
- P_1 verrouille R_1
- P_2 verrouille R_2

Jamais P_1 et P_2 ne pourront finir leur exécution : **interblocage**

Interblocage

Exemple

Soit P_1, P_2 des processus et R_1, R_2 des ressources.

- P_1 et P_2 ont tous les 2 besoin de R_1 et R_2
- P_1 verrouille R_1
- P_2 verrouille R_2

Jamais P_1 et P_2 ne pourront finir leur exécution : **interblocage**

Interblocage

Exemple

Soit P_1, P_2 des processus et R_1, R_2 des ressources.

- P_1 et P_2 ont tous les 2 besoin de R_1 et R_2
- P_1 verrouille R_1
- P_2 verrouille R_2

Jamais P_1 et P_2 ne pourront finir leur exécution : **interblocage**

- N'intervient pas nécessairement sur des situations aussi simples
- Un processus qui attend une ressource est à l'état bloqué (cf cours précédent), d'où le terme d'**interblocage**.

Famine

Exemple : lecteur et écrivains

- / lecteurs, potentiellement e écrivains
- Lecture **interdit** l'écriture.
- L'écriture **interdit** la lecture ET l'écriture.
- Le verrou est sur la donnée partagée.

Famine

Exemple : lecteur et écrivains

- / lecteurs, potentiellement e écrivains
- Lecture **interdit** l'écriture.
- L'écriture **interdit** la lecture ET l'écriture.
- Le verrou est sur la donnée partagée.

Famine

- Quand / est beaucoup plus grand que e : famine pour l'écrivain.
- Passer par des priorités : **dangereux**, potentiellement **inversion de priorité**

Inversion de priorité

Exemple

- P_1 et P_2 2 processus avec $\text{Prio}(P_1) > \text{Prio}(P_2)$
- P_2 commence à t_0 et prend un verrou sur la ressource R_1
- P_1 est **préemptif** et tente de prendre le verrou sur R_1 : P_1 attend que P_2 libère le verrou et est donc bloqué. Si un processus P_3 de priorité p t.q. $\text{Prio}(P_1) > p > \text{Prio}(P_2)$ existe, P_3 va s'exécuter seul.

Inversion de priorité

Exemple

- P_1 et P_2 2 processus avec $\text{Prio}(P_1) > \text{Prio}(P_2)$
 - P_2 commence à t_0 et prend un verrou sur la ressource R_1
 - P_1 est **préemptif** et tente de prendre le verrou sur R_1 : P_1 attend que P_2 libère le verrou et est donc bloqué. Si un processus P_3 de priorité p t.q. $\text{Prio}(P_1) > p > \text{Prio}(P_2)$ existe, P_3 va s'exécuter seul.
-
- Souvent dans des cas plus complexes.
 - Difficile à détecter ...

Un exemple concret : NASA Mars PathFinder (1997)

Le robot explorateur perd des données régulièrement (il est sur Mars).

Le contexte

- Mémoire partagée protégée par un verrou
- Processus de gestion de bus sur la mémoire partagée priorité haute
- Écriture en mémoire partagée (récupération de données), basse priorité
- Routine de communication, priorité moyenne (pas d'accès mémoire).
- Pas d'exécution de la gestion de bus pendant un certain temps : redémarrage système.

Un exemple concret : NASA Mars PathFinder (1997)

Le robot explorateur perd des données régulièrement (il est sur Mars).

Le contexte

- Mémoire partagée protégée par un verrou
- Processus de gestion de bus sur la mémoire partagée priorité haute
- Écriture en mémoire partagée (réécriture de données), basse priorité
- Routine de communication, priorité moyenne (pas d'accès mémoire).
- Pas d'exécution de la gestion de bus pendant un certain temps : redémarrage système.

Le problème

- Écriture verrouille la mémoire partagée.
- La gestion du bus attend ce verrou.
- La routine de communication s'exécute alors, jusqu'au redémarrage.

Problème (heureusement) corrigé à distance ...

Problème classique : les philosophes

- $n \geq 5$ philosophes autour d'une table. Ils passent leur temps à manger ou penser.
- Devant eux : une assiette de spaghetti. A gauche de chaque assiette : une fourchette.
- Il leur faut 2 fourchettes pour manger : celles autour de leur assiette.
- 3 états possibles pour les philosophes :
 - Manger : durée déterminée et finie.
 - Penser : durée indéterminée.
 - Avoir faim : ils veulent manger mais les fourchettes sont prises : durée déterminée et finie. Sinon : **famine**.

Problème classique : les philosophes

- $n \geq 5$ philosophes autour d'une table. Ils passent leur temps à manger ou penser.
- Devant eux : une assiette de spaghetti. A gauche de chaque assiette : une fourchette.
- Il leur faut 2 fourchettes pour manger : celles autour de leur assiette.
- 3 états possibles pour les philosophes :
 - Manger : durée déterminée et finie.
 - Penser : durée indéterminée.
 - Avoir faim : ils veulent manger mais les fourchettes sont prises : durée déterminée et finie. Sinon : **famine**.

Problème

Les philosophes doivent manger chacun à leur tour, on veut éviter la famine :

- Complexé en distribué
- Local : peut être résolu grâce aux sémaphores

Les philosophes : solution (trop) naïve

- Fourchettes = ressources partagées (un sémaphore par fourchette).
- Prendre une fourchette = $P(id_{fork})$, la libérer = $V(id_{fork})$.
- Philosophes (proc.) et fourchettes (ress.) numérotées de 0 à n .

```
int philosophe(int i, . . .)
{
    ...
    while (1)
    {
        penser();
        P(semid[i]); // On attend la fourchette de gauche
        P(semid[(i + 1) % N]); // On attend la fourchette de droite
        manger();
        V(semid[i]); // On libère la fourchette de gauche
        V(semid[(i + 1) % N]); // On libère la fourchette de droite
    }
}
```

Résoudre le problème des philosophes

Solution naïve : problèmes

- **Interblocages possibles** : si tous les philosophes prennent la fourchette de gauche, tout le monde est bloqué.
- Famine ...

Résoudre le problème des philosophes

Solution naïve : problèmes

- **Interblocages possibles** : si tous les philosophes prennent la fourchette de gauche, tout le monde est bloqué.
- Famine ...

Solution quand n est pair

- Les philosophes d' id pair prennent d'abord la fourchette de gauche, et ensuite celle de droite.
- Ceux d' id impair font l'inverse (d'abord droite, ensuite gauche).

Résoudre le problème des philosophes

Solution naïve : problèmes

- **Interblocages possibles** : si tous les philosophes prennent la fourchette de gauche, tout le monde est bloqué.
- Famine ...

Solution quand n est pair

- Les philosophes d' id pair prennent d'abord la fourchette de gauche, et ensuite celle de droite.
- Ceux d' id impair font l'inverse (d'abord droite, ensuite gauche).

Solution pour le cas général

Se baser sur les états des philosophes voisins pour voir si l'on peut manger, et avoir un sémaphore par philosophe. Lorsqu'un philosophe a fini de manger, il réveille ses éventuels voisins qui attendent. Nécessite l'**exclusion mutuelle** quand on regarde les états de ses voisins !

Solution pour le cas général

Code initialisation :

```
booléen Etat[0..4];
pour i := 0 à 4
    faire Etat[i] := Pense;
```

DemandeFourchettes :

```
P(Mutex);
    si Etat[gauche[i]] <> Mange
        et Etat[droite[i]] <> Mange
    alors début
        Etat[i] := Mange;
        V(SemPriv[i]);
    fin
    sinon Etat[i] := Attend;
V(Mutex);
P(SemPriv[i]);
```

RestituerFourchettes :

```
P(Mutex);
    si Etat[gauche[i]] = Attend
        et Etat[gauche[gauche[i]]] <> Mange
    alors début
        Etat[gauche[i]] := Mange;
        V(SemPriv[gauche[i]]);
    fin;
    Idem pour droite;
    Etat[i] := Pense;
V(Mutex);
```

Code philosophe :

```
répéter
    Penser;
    DemanderFourchettes;
    Manger;
    RestituerFourchettes;
jusqu'à plat = vide;
```

Exclusion mutuelle et section critique

Exclusion mutuelle : un seul processus à la fois peut utiliser une ressource donnée :

- Une ressource physique (ex : imprimante)
- Une variable partagé en écriture en peut être lue
- Une portion de code ne peut être effectuée simultanément par plusieurs processus : section de code concernée : **section critique**

Exclusion mutuelle et section critique

Exclusion mutuelle : un seul processus à la fois peut utiliser une ressource donnée :

- Une ressource physique (ex : imprimante)
- Une variable partagé en écriture en peut être lue
- Une portion de code ne peut être effectuée simultanément par plusieurs processus : section de code concernée : **section critique**

Avant d'utiliser une ressource, ou d'entrer en section critique, un processus fait une demande explicite :

- Si la ressource est libre, il la prend et indique qu'elle est utilisée
- Si la ressource est déjà utilisée, il attend

Exclusion mutuelle et section critique

Exclusion mutuelle : un seul processus à la fois peut utiliser une ressource donnée :

- Une ressource physique (ex : imprimante)
- Une variable partagé en écriture en peut être lue
- Une portion de code ne peut être effectuée simultanément par plusieurs processus : section de code concernée : **section critique**

Avant d'utiliser une ressource, ou d'entrer en section critique, un processus fait une demande explicite :

- Si la ressource est libre, il la prend et indique qu'elle est utilisée
- Si la ressource est déjà utilisée, il attend

Un sémaphore peut être utilisé pour cela. C'est lui qui indique si la ressource est disponible ou non. **Très souvent utile quand on partage des accès ou des variables.**

Implémentations disponibles

2 implémentations sont disponibles :

- Une version issue de System V, une version d'Unix, qui est un rajout aux fonctionnalités d'origine du système.
- Une version POSIX, donc plus portable.

Implémentations disponibles

2 implémentations sont disponibles :

- Une version issue de System V, une version d'Unix, qui est un rajout aux fonctionnalités d'origine du système.
- Une version POSIX, donc plus portable.

Chacune a ses forces et ses faiblesses :

- La version System V est ... complexe à utiliser, voire contre-intuitive. POSIX est intuitif.
- Les sémaphores POSIX sont moins gourmands en mémoire, passe mieux à l'échelle et sont supposément plus rapides.
- System V permet plus de contrôles et d'opérations (permissions, valeurs d'incrément, mais elles ne sont pas souvent nécessaires)
- Les sémaphores POSIX ne sont pas forcément persistants.
- L'avenir semble promis à POSIX, mais l'implémentation complète est moins répandue que celle pour l'interface System V.

Implémentations disponibles

2 implémentations sont disponibles :

- Une version issue de System V, une version d'Unix, qui est un rajout aux fonctionnalités d'origine du système.
- Une version POSIX, donc plus portable.

Chacune a ses forces et ses faiblesses :

- La version System V est ... complexe à utiliser, voire contre-intuitive. POSIX est intuitif.
- Les sémaphores POSIX sont moins gourmands en mémoire, passe mieux à l'échelle et sont supposément plus rapides.
- System V permet plus de contrôles et d'opérations (permissions, valeurs d'incrément, mais elles ne sont pas souvent nécessaires)
- Les sémaphores POSIX ne sont pas forcément persistants.
- L'avenir semble promis à POSIX, mais l'implémentation complète est moins répandue que celle pour l'interface System V.

Puisque l'utilisation est simple et rapide en POSIX, on va s'attacher à la version System V...

Implémentation en C sous Linux

Avec la version SystemV d'Unix sont apparus 3 nouveaux mécanismes de communications (ou synchronisation) entre les processus locaux : *files de messages, sémaphores, mémoire partagée*. Ces 3 mécanismes sont extérieurs au système de fichiers :

- Pas de descripteur (pas de référence à un fichier)
- Mécanismes de communication (ou synchronisation) pour des processus sans lien de parenté mais s'exécutant sur la même machine
- Le système gère une table de files pour chacun de ces mécanismes :
 - Le système assure la gestion de ces files/IPC
 - La commande ipcs donne la liste des IPC courants
 - La commande ipcrm permet de détruire les IPC qui vous appartiennent
 - Chaque IPC dispose d'une identification interne (entier positif ou nul). Les processus qui veulent utiliser ces objets doivent connaître l'identificateur.
- Les objets sont identifiés par un **système de clé**. L'attribution et la gestion des clés est un des points les plus délicats pour ces objets.

Opérations

Soit S la valeur numérique (compteur) d'un sémaphore x :

- $P(x, n)$: Si $(S - n) \geq 0$, alors $S = S - n$ et le processus continue, sinon le processus est stoppé
- $V(x, n)$: $S = S + n$ et tous les processus en attente du sémaphore sont débloqués (dans l'ordre des demandes ...)
- $Z(x)$: le processus qui effectue cette opération est suspendu jusqu'à ce que $S = 0$
- Quand on parle des opérations P et V sur un sémaphore, on considère que $n = 1$
- Atomicité des opérations P et V (suite non interruptible)
- Existence d'un mécanisme mémorisant les processus faisant P

Usage des commandes ipcs et ipcrm

ipcs : *Show information about IPC objects*

ipcs fournit des informations sur les fonctionnalités IPC.

Voir man ipcs

ipcrm : *Remove IPC objects*

Utiliser pour détruire en ligne de commande des objets IPC non nettoyés par un programme : file de messages, sémaphores, mémoire partagée

ipcrm shmlmsglsem id...

Voir man ipcrm

Astuces

```
for i in `ipcs -s | cut -f2 -d' ' | sed -n '/[0-9]/p' ;  
do ipcrm -s $i ; done
```

Opérations system V

En Unix SystemV, l'implémentation des sémaphores est beaucoup plus riche qu'une simple implémentation de P et V :

- Famille de sémaphores (plusieurs sémaphores d'une même famille).
Intérêts :
 - Atomicité des opérations sur tous les membres de la famille
 - Acquisition simultanée d'exemplaires multiples de ressources différentes
 - Définition des opérations `Pn()` et `Vn()`
 - Diminution ou augmentation de la valeur du sémaphore de n de façon atomique
 - Fonction `Z()`
- Les constantes et les prototypes des fonctions sont définies dans
`<sys/sem.h>` et `<sys/ipc.h>`

Obtention de la clef

Les IPC sont identifiés par un mécanisme de clé. Ces clés jouent le rôle de références pour les fichiers. Ces clés ont des valeurs numériques.

- Le type de ces variables est `key_t` défini dans `<sys/types.h>`
- Chaque type d'IPC possède son propre ensemble de clés : **Important pour la portabilité des applications.**

Le problème de clé peut se résoudre avec la fonction `ftok` :

```
key_t ftok(const char *, int num);
```

Cette fonction rend une clé obtenue à partir du nom d'un fichier existant et d'une valeur entière

- Les mêmes paramètres fourniront les mêmes clés (sauf déplacement de fichiers, qui dans ce cas change de numéro d'index ...)
- Avec `IPC_PRIVATE` (clé privée) quand tous les processus qui utiliseront l'objet IPC sont des descendants de celui qui le crée

Exemples d'obtention de clefs

```
int main()
{
SEMAPHORE semid;
. .
if ((semid = semget(IPC_PRIVATE, nb_sem, IPC_CREAT|IPC_EXCL|0666)) == -1)
/* Erreur */
. .
}
```

Un autre exemple :

```
int main()
{
SEMAPHORE semid;
key_t cle;
. .
if ((cle = ftok("Exo2.c", 15)) == -1)
/* Erreur */
. .
if ((semid = semget(cle, nb_sem, IPC_CREAT|IPC_EXCL|0666)) == -1)
/* Erreur */
. .
}
```

System V : création

La fonction `semget` permet de créer une famille de sémaphores :

```
int semget(key_t clef, int nb_sem, int option);
```

- clef : id système de l'IPC (type spécifique)
- nb_sem : nombre de sémaphores dans la famille
- Les membres de la famille sont numérotés de 0 à nb_sem - 1
- option : Ou logique entre plusieurs constantes
 - IPC_CREAT : Crée l'objet (l'IPC) si il n'existe pas
 - IPC_EXCL : avec IPC_CREAT, signale par une erreur que l'objet existe
 - Droit en lecture-écriture
- Retourne un identificateur pour le sémaphore (ou la famille de sémaphore).

System V : création

La fonction `semget` permet de créer une famille de sémaphores :

```
int semget(key_t clef, int nb_sem, int option);
```

- clef : id système de l'IPC (type spécifique)
- nb_sem : nombre de sémaphores dans la famille
- Les membres de la famille sont numérotés de 0 à nb_sem - 1
- option : Ou logique entre plusieurs constantes
 - `IPC_CREAT` : Crée l'objet (l'IPC) si il n'existe pas
 - `IPC_EXCL` : avec `IPC_CREAT`, signale par une erreur que l'objet existe
 - Droit en lecture-écriture
- Retourne un identificateur pour le sémaphore (ou la famille de sémaphore).

Exemple :

```
int semid;
int nombre_de_sem = 4;

semid = semget(cle, nombre_de_sem, IPC_CREAT | IPC_EXCL | 0666);
if (semid < 0)
{
    printf("Erreur semget\n");
    exit(0);
}
```

Initialisation d'une famille de sémaphores

La fonction `semctl` permet d'initialiser une famille de sémaphores :

```
int semctl(int semid, int semnum, int option, ... /* arg */);
```

Initialisation (et consultation) des valeurs des sémaphores. La fonction `semctl` rend -1 en cas d'erreur.

En fonction de option, `semnum` et `arg` ont les valeurs suivantes :

- `SETALL`, `semnum` est ignoré, `arg` reçoit un pointeur sur un tableau d'entiers courts pour initialiser les valeurs de toute la famille
- `SETVAL`, `semnum` indique le numéro du sémaphore dans la famille, `arg` reçoit l'entier correspondant à la valeur d'initialisation
- `GETALL` et `GETVAL` permettent de connaître la valeur des sémaphores
- `IPC_RMID` pour supprimer la famille de sémaphore (`semnum` et `arg` sont ignorés)

Structure sembuf

C'est la structure à remplir (une par sémaphore de la famille, donc un tableau de structures si plusieurs membres) pour demander l'opération à effectuer. Définie dans <sys/sem.h>. Les flags : IPC_NOWAIT (non bloquant) et SEM_UNDO (terminaison).

```
struct sembuf {  
    // Numéro du sémaphore dans la famille  
    unsigned short int sem_num;  
    short            sem_op;  // Définition de l'action  
    short            sem_flg; // Option pour l'opération  
};
```

Si `sem_op` est :

- > 0 V(`semid, n`)
- $= 0$ Z
- < 0 P(`semid, n`)

Opération sur les sémaphores avec semop()

La fonction `semop` permet de réaliser un ensemble d'opérations sur une famille de sémaphores :

```
int semop(int semid, struct sembuf *tab_str, int nb_op);
```

- `semid` est l'identificateur de la famille
- `tab_str` est un tableau de structures `sembuf` (`nb_op` structures)
- `nb_op` nombre d'opérations à effectuer

Retourne 0 si OK, -1 en cas d'échec.

Opérations réalisées de façon atomique :

- Toutes ou aucune : annule les $i - 1$ ième si la i ème opération ne peut être effectuée
- L'ordre des opérations n'est pas indifférent (sauf si toutes bloquantes ou toutes non bloquantes)

Exemple de P() et de V()

Définition d'un type opaque

```
typedef int SEMAPHORE;
```

Exemple de P()

```
int P(SEMAPHORE sem)
{
    struct sembuf sb;

    sb.sem_num = 0;
    sb.sem_op = -1;
    sb.sem_flg = SEM_UNDO;

    return semop(sem, &sb, 1);
}
```

Exemple de V()

```
int V(SEMAPHORE sem)
{
    struct sembuf sb;

    sb.sem_num = 0;
    sb.sem_op = 1;
    sb.sem_flg = SEM_UNDO;

    return semop(sem, &sb, 1);
}
```

Norme POSIX

Déclaré dans <semaphore.h>. 2 types de sémaphores :

- Anonymes : utilisable seulement dans le processus qui l'a créé
- Nommés : utilisable par des processus indépendants

Création, Initialisation, Destruction :

- `sem_t mon_semaphore;`
- `int sem_init(sem_t *semap, int partage, unsigned int valeur);`
- `int sem_destroy(sem_t *semap);`

Pour accéder à un sémaphore nommé :

- `sem_t *sem_open(const char *nom, int options, mode_t mode, unsigned int valeur);`
- `int sem_close(sem_t *semap);`

Opérations :

- `int sem_wait(sem_t *semap);` équivalent à P(semid)
- `int sem_post(sem_t *semap);` équivalent à V(semid)

Unix - e2i5 - Les processus légers

Nicolas Palix

Polytech

2017

1 Les processus légers

- Rappel sur la notion de processus UNIX
- Définition de processus légers
- Avantages des threads
- Les différents états d'un processus
- Threads POSIX
- Création d'un thread
- Attente de la terminaison d'un thread
- Terminaison
- Gestion des threads
- Section Critique

2 Mécanismes de synchronisation

- MUTEX
- Variables conditionnelles

Rappel sur la notion de processus UNIX

Processus

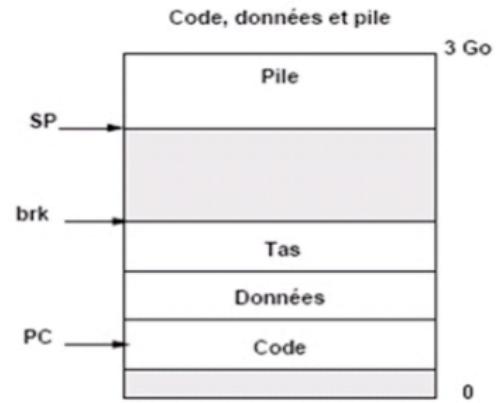
Unité d'exécution (unité de partage du temps processeur et de la mémoire)

Processus UNIX classiques (lourds)

- Crées par copie indépendantes des ressources du processus père
- Ressources séparées (espace mémoire, descripteurs,...)
- Un processus lourd = un contexte + un espace d'adressage

Contexte du processus

- Contexte d'exécution
 - Registres généraux
 - Registre d'état
 - Pointeur de pile (SP)
 - Compteur ordinal (PC)
- Contexte noyau
 - Tables des descripteurs
 - Pointeur brk



Processus UNIX classiques (lourds)

Avantage :

Pas de risque d'écrasement des données d'un autre processus.

Inconvénients :

- Espace d'adressage propre (pas de partage de la mémoire simple)
- Copies de toutes les variables et ressources nécessaires (pile, registres, fichiers ouverts...)
- Mécanismes de communication lourds (pipe, mémoire partagée...)
- Mécanismes de synchronisation lourds (signaux...)
- Création par des appels systèmes lents
- Changement de contexte lent
- ...

Définition de processus légers

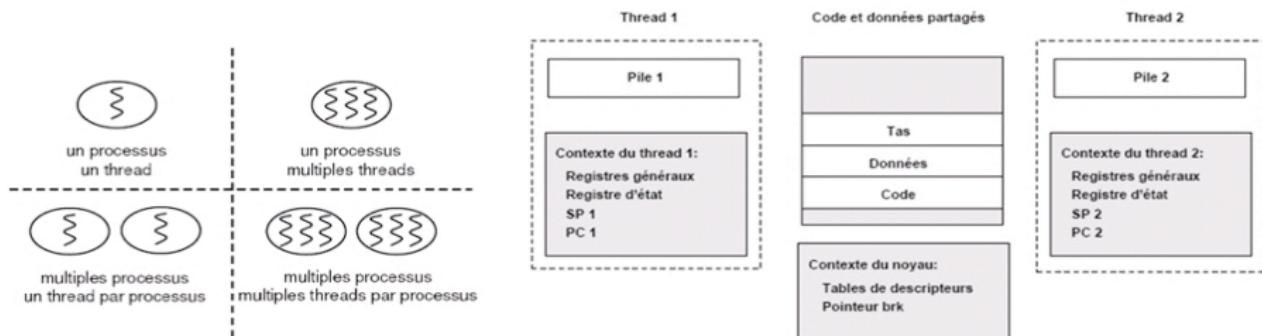
Thread – fil

Un thread est un fil d'exécution, une subdivision d'un processus. Plusieurs traductions : activité (tâche), fil d'exécution, *lightweight process* (lwp) ou processus léger (par opposition au processus lourd créé par fork)

Les threads permettent de dérouler plusieurs suites d'instructions, en PARALLÈLE, à l'intérieur du même processus. **Un thread exécute une fonction.**

Partage de ressources avec les autres threads

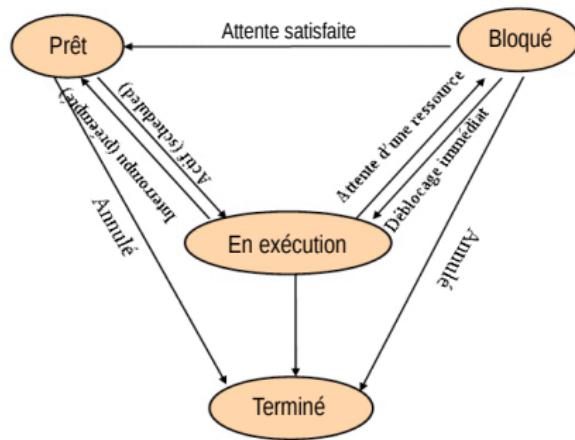
- Le code
- Les variables
- Le tas
- La table des fichiers ouverts
- Ressources propres à chaque thread
- La pile
- Les registres



Avantage des threads

- Parallélisme et multiprocesseur
 - Machine multiprocesseurs
 - Plusieurs threads au sein d'une application
- Débit
 - Un seul thread
attente à chaque requête au système
 - Plusieurs threads
Le thread qui a fait la requête attend, un autre thread peut poursuivre son exécution
- Communication entre threads
 - Plus rapide et plus efficace
- Ressources systèmes
 - Moins coûteux
 - Augmentation de sa rapidité d'exécution
- Facilité de la mise en œuvre du programme

Les différents états d'un processus



- Prêt : Le fil est prêt à être exécuté.
- En exécution : Le fil est en cours d'exécution sur un processeur.
- Bloqué : Le fil est en attente sur une synchronisation ou sur la fin d'une opération.
- Terminé : Le fil a terminé son exécution ou a été annulé.

Interface POSIX pour les threads

- Bibliothèque pthread : #include <pthread.h>
- Compilation avec -lpthread
- En général, valeur de retour est 0 si OK

Plus d'informations : `man pthreads`

Création d'un thread

`pthread_create()`

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

`thread` est initialisée à la création du processus.

`attr` définit les attributs à appliquer au nouveau thread. Si ce paramètre est omis (NULL), le processus subit la politique d'ordonnancement par défaut et n'est pas détachable.

`fonction` est un pointeur sur la fonction qui sera exécutée.

`arg` permet de fixer les arguments passés en paramètre à la fonction à sa création.

Attente de la terminaison d'un thread

- La fonction `pthread_join()` permet au processus d'attendre la fin d'un processus léger, et de récupérer son code de retour.
Équivalent du `wait()` pour les processus.
- Les ressources du processus ne sont libérées qu'à la terminaison du dernier thread.

`pthread_join()`

```
int pthread_join (pthread_t thread, void ** vrtval);
```

`thread` identifie le thread

`vrtval` est l'adresse de la zone de récupération de la valeur de retour.

Terminaison

La terminaison du processus léger se fait avec un code de retour :

- Par appel à la fonction `pthread_exit()`
- Lorsque la fonction associée au thread se termine par `return retval` (appel implicite à `pthread_exit()`)

`pthread_exit()`

```
void pthread_exit (void *vrtval);
```

`vrtval` est l'adresse de la valeur de retour ou `NULL`.

Exemple 1 - hello.c 1/2

```
1 void *thread_hi ( void* nb ) ;
2 void *thread_boujour ( void* nb ) ;
3 int main ( int argc , char **argv ) {
4     pthread_t tid[2];
5     int i=1, j=2;
6     void *ret_val1 , ret_val2;
7     /* creation de deux threads */
8     printf("Processus %i : cree 2 threads et les attend.\n", getpid());
9     int ret1 = pthread_create(&tid[0], NULL, thread_hi, (void *) i);
10    int ret2 = pthread_create(&tid[1], NULL, thread_boujour, (void *) j);
11    if (ret1 || ret2) {
12        fprintf(stderr, "Probleme lors de la creation d'un thread.\n");
13        return 1;
14    }
15    ret1 = pthread_join( tid[0] , &ret_val1 );
16    ret2 = pthread_join( tid[1] , &ret_val2 );
17    if (ret1 || ret2) {
18        fprintf (stderr, "Probleme lors de l'attente d'un thread.\n");
19        return 1;
20    }
21    printf("Le thread 1 s'est termine avec le code %d.", (int) ret_val1);
22    printf("Le thread 2 s'est termine avec le code %d.", (int) ret_val2 );
23    return 0;
24 }
```

Exemple 1 - hello.c 2/2

```
1 void *thread_hi (void *nb) {
2     int id = (int) nb;
3     printf ("Hi ! Thread %d du processus %i.\n", id , getpid ());
4     return (void *) id;
5 }
6
7 void *thread_bonjour(void *nb) {
8     int id = (int) nb;
9     printf( "Bonjour ! Thread %d du processus %i .\n", id , getpid ());
10    return (void *) id;
11 }
```

```
Processus 3125: cree deux threads et les attend
Hi ! Thread 1 du processus 3125
Bonjour ! Thread 2 du processus 3125
Le thread 1 s'est termine avec le code 1.
Le thread 2 s'est termine avec le code 2.
```

Exemple 2

```
1 static int valeur = 0;
2 void main(void) {
3     pthread_t tid1, tid2;
4     void *fond(void);
5     pthread_set_concurrency(2);
6     pthread_create(&tid1, NULL,
7                     (void *(*())() fond, NULL);
8     pthread_create(&tid2, NULL,
9                     (void *(*())() fond, NULL);
10    pthread_join(tid1, NULL);
11    pthread_join(tid2, NULL);
12    printf("valeur=%d\n", valeur);
13 }
14
15 void fond(void) {
16     int i;
17     pthread_t tid;
18     for (i=0; i <1000000; i++)
19         valeur = valeur + 1;
20     tid = pthread_self();
21     printf("tid=%d valeur=%d\n", tid, valeur);
22 }
```

Quelle valeur sera affichée ?

- Le code fond() est exécuté en concurrence par 2 threads.
- La variable globale valeur est partagée entre les threads.

Exemple 2

```
1 static int valeur = 0;
2 void main(void) {
3     pthread_t tid1, tid2;
4     void *fond(void);
5     pthread_set_concurrency(2);
6     pthread_create(&tid1, NULL,
7                     (void *(*())() fond, NULL);
8     pthread_create(&tid2, NULL,
9                     (void *(*())() fond, NULL);
10    pthread_join(tid1, NULL);
11    pthread_join(tid2, NULL);
12    printf("valeur=%d\n", valeur);
13 }
14
15 void fond(void) {
16     int i;
17     pthread_t tid;
18     for (i=0; i <1000000; i++)
19         valeur = valeur + 1;
20     tid = pthread_self();
21     printf("tid=%d valeur=%d\n", tid, valeur);
22 }
```

Quelle valeur sera affichée ?

```
tid : 4 valeur = 1342484
tid : 5 valeur = 1495077
valeur = 1495077
```

- Le code `fond()` est exécuté en concurrence par 2 threads.
- La variable globale `valeur` est partagée entre les threads.

Exemple 2

```
1 static int valeur = 0;
2 void main(void) {
3     pthread_t tid1, tid2;
4     void *fond(void);
5     pthread_set_concurrency(2);
6     pthread_create(&tid1, NULL,
7                     (void *(*())() fond, NULL);
8     pthread_create(&tid2, NULL,
9                     (void *(*())() fond, NULL);
10    pthread_join(tid1, NULL);
11    pthread_join(tid2, NULL);
12    printf("valeur=%d\n", valeur);
13 }
14
15 void fond(void) {
16     int i;
17     pthread_t tid;
18     for (i=0; i <1000000; i++)
19         valeur = valeur + 1;
20     tid = pthread_self();
21     printf("tid=%d valeur=%d\n", tid, valeur);
22 }
```

Quelle valeur sera affichée ?

```
tid : 4 valeur = 1342484
tid : 5 valeur = 1495077
valeur = 1495077
```

```
tid : 4 valeur = 1279685
tid : 5 valeur = 1596260
valeur = 1596260
```

```
tid : 5 valeur = 1958900
tid : 4 valeur = 2000000
valeur = 2000000
```

- Le code fond() est exécuté en concurrence par 2 threads.
- La variable globale valeur est partagée entre les threads.

Section critique

Section critique

C'est une partie de code telle que deux threads ne peuvent s'y trouver au même instant.

- Il est nécessaire d'utiliser des sections critiques lorsqu'il y a accès à des ressources partagées par plusieurs threads.
- Les **mécanismes de synchronisation** sont utilisés pour résoudre les problèmes de sections critiques et plus généralement pour bloquer et débloquer des threads suivant certaines conditions.

1 Les processus légers

- Rappel sur la notion de processus UNIX
- Définition de processus légers
- Avantages des threads
- Les différents états d'un processus
- Threads POSIX
- Création d'un thread
- Attente de la terminaison d'un thread
- Terminaison
- Gestion des threads
- Section Critique

2 Mécanismes de synchronisation

- MUTEX
- Variables conditionnelles

Mécanismes de synchronisation

Dans la programmation concurrente, le terme de synchronisation se réfère à deux concepts distincts (mais liés) :

- **La synchronisation de processus ou tâche** : mécanisme qui vise à bloquer l'exécution des différents processus à des points précis de leur programme de manière à ce que tous les processus passent les étapes bloquantes au moment prévu par le programmeur.
- **La synchronisation de données** : mécanisme qui vise à conserver la cohérence entre différentes données dans un environnement multitâche.

Les problèmes liés à la synchronisation rendent toujours la programmation plus difficile.

API Mutex

Un **Mutex** (*Mutual exclusion*) est une primitive de synchronisation permettant d'éviter que des ressources partagées d'un système ne soient utilisées en même temps.

Nom de la fonction	Rôle de la fonction
<code>pthread_mutex_init</code>	Le verrou est créé et mis à l'état <i>unlock</i>
<code>pthread_mutex_destroy</code>	Le verrou est détruit
<code>pthread_mutex_lock</code>	Si le verrou est déjà pris, le thread est bloqué
<code>pthread_mutex_trylock</code>	Renvoie une erreur si le verrou <i>locked</i> , le thread N'est PAS bloqué
<code>pthread_mutex_unlock</code>	Rend le verrou et libère un thread

Linux package for the manpages under Debian-based system

`apt-get install manpages-posix-dev`

Création/destruction de mutex

- Allocation

```
1 pthread_mutex_t lock;  
2  
3 pthread_mutex_t *mp;  
4 mp = malloc(sizeof(pthread_mutex_t));
```

- Initialisation (obligatoire)

```
1 pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;  
2  
3 int pthread_mutex_init(pthread_mutex_t *mp,  
4                         const pthread_mutexattr_t *attr);
```

- Destruction

```
1 int pthread_mutex_destroy(pthread_mutex_t *mp);
```

Verrouillage/Déverrouillage de mutex

- Verrouillage (appel bloquant)

```
1 int pthread_mutex_lock(pthread_mutex_t *mp);
```

- Déverrouillage (uniquement par le propriétaire et sur un verrou pris)

```
1 int pthread_mutex_unlock(pthread_mutex_t *mp);
```

Exemple de mutex

```
1 #include <pthread.h>
2 int cpt; /* Variable partagée */
3
4 /* Équivalent à pthread_mutex_init avec attr=NULL */
5 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
6
7 /* Fonction réalisant une opération sur cpt */
8 void* fons(void* arg) {
9     pthread_mutex_lock(&mutex); /* Prise du mutex */
10    cpt++;
11    pthread_mutex_unlock(&mutex); /* Libération du mutex */
12    pthread_exit(NULL); /* Terminaison du thread */
13 }
14
15 int main() {
16     int i;
17     pthread_t thread_id[2];
18
19     cpt = 0;
20     pthread_create(&thread_id[0], NULL, fons, (void *) NULL);
21     pthread_create(&thread_id[1], NULL, fons, (void *) NULL);
22     for (i=0; i<2; i++)
23         pthread_join(thread_id[i], NULL); /* Attend l'arrêt des threads */
24     pthread_mutex_destroy(&mutex); /* Destruction des mutex */
25 }
```

Variables conditionnelles

Les variables conditionnelles (conditions variables), s'utilisent conjointement à un verrou, elles évitent les interblocages.

Nom de la fonction	Rôle de la fonction
<code>pthread_cond_init(&cv)</code>	La variable conditionnelle cv est initialisée
<code>pthread_cond_destroy(&cv)</code>	La variable conditionnelle cv est détruite
<code>pthread_cond_wait(&cv, &V)</code>	Fait passer le thread à l'état bloqué ET rend le verrou V de façon atomique. Sort de l'état bloqué et essaie de reprendre V sur un cond_signal ou cond_broadcast
<code>pthread_cond_signal(&cv)</code>	Libère un des threads bloqués sur cv
<code>pthread_cond_broadcast(&cv)</code>	Libère tous les threads bloqués sur cv
<code>pthread_cond_timedwait()</code>	Attend un signal de libération pendant un certain temps

Conclusion

- Les caractéristiques des processus légers
- Le standard POSIX des processus légers
- Les fonctions de thread :
 - Création d'un thread
 - Attente de la terminaison d'un thread
 - Terminaison d'un thread
- Les mécanismes de synchronisation

Unix - e2i5 - Mémoire partagée

Nicolas Palix

Polytech

2017

1 Rappels

- Processus
- IPC

2 Mémoire partagée

- Définition
- Implémentation
- System V
- POSIX

3 Recommandations

Définition

- Un processus est un fil d'exécution qui s'exécute sur une machine
- Plusieurs processus cohabitent simultanément, souvent liés entre eux par des liens de famille.
- Les processus se partagent les ressources de la machine (périphériques, CPU, fichiers ...)

Définition

- Un processus est un fil d'exécution qui s'exécute sur une machine
- Plusieurs processus cohabitent simultanément, souvent liés entre eux par des liens de famille.
- Les processus se partagent les ressources de la machine (périphériques, CPU, fichiers ...)

Pourquoi interagir avec les processus ?

- Les processus interagissent entre eux et doivent donc pouvoir communiquer entre eux
- Le système doit pouvoir avertir les processus en cas de défaillance d'un composant du système
- L'utilisateur doit pouvoir gérer les processus (arrêt, suspension, ..)

Définition

- Un processus est un fil d'exécution qui s'exécute sur une machine
- Plusieurs processus cohabitent simultanément, souvent liés entre eux par des liens de famille.
- Les processus se partagent les ressources de la machine (périphériques, CPU, fichiers ...)

Pourquoi interagir avec les processus ?

- Les processus interagissent entre eux et doivent donc pouvoir communiquer entre eux
- Le système doit pouvoir avertir les processus en cas de défaillance d'un composant du système
- L'utilisateur doit pouvoir gérer les processus (arrêt, suspension, ..)

Les méthodes de communications entre processus sont souvent désignés par l'acronyme IPC : Inter-Process Communications

IPC

Il existe de multiples moyens de réaliser une communication inter-processus :

- Par fichiers (\simeq tout système)
- *Signaux* (Unix/Linux/MacOS, pas vraiment sous Windows)
- Sockets (\simeq tout système)
- Files d'attente de message (\simeq tout système)
- *Pipe/tubes* (tout système POSIX, Windows)
- *Named pipe/tubes nommés* (tout système POSIX, Windows)
- *Sémaphores* (tout système POSIX, Windows)
- **Mémoire partagée** (tout système POSIX, Windows)
- Passage de message : MPI, RMI, CORBA ...
- ...

IPC

Il existe de multiples moyens de réaliser une communication inter-processus :

- Par fichiers (\simeq tout système)
- *Signaux* (Unix/Linux/MacOS, pas vraiment sous Windows)
- Sockets (\simeq tout système)
- Files d'attente de message (\simeq tout système)
- *Pipe/tubes* (tout système POSIX, Windows)
- *Named pipe/tubes nommés* (tout système POSIX, Windows)
- *Sémaphores* (tout système POSIX, Windows)
- **Mémoire partagée** (tout système POSIX, Windows)
- Passage de message : MPI, RMI, CORBA ...
- ...

Aujourd'hui on s'intéresse à la **mémoire partagée**.

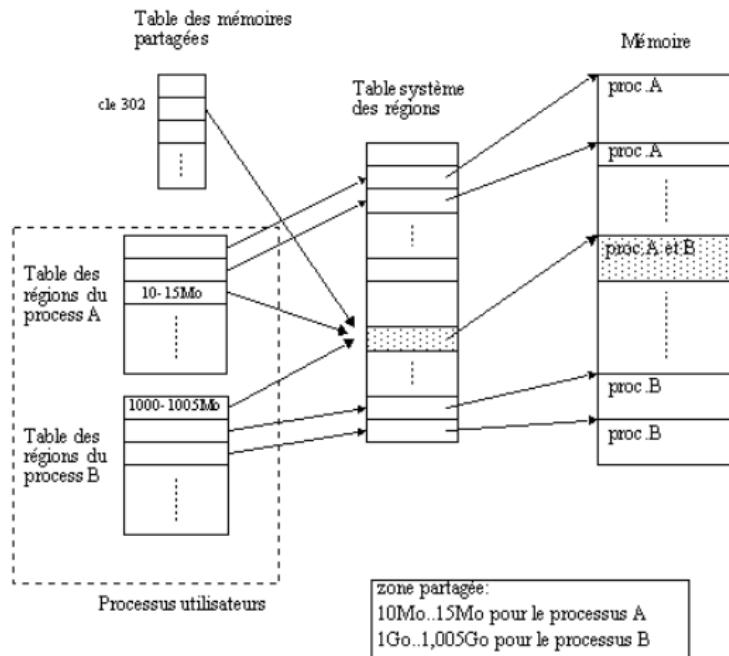
Pourquoi partager de la mémoire

- Quand on fait un `fork()` et que l'on crée donc un processus, tout l'espace mémoire du père est copié pour servir d'espace mémoire au fils : les 2 sont donc indépendants
- Ce n'est pas le cas lors de la création d'un thread (processus léger)
- Partager de la mémoire est un mécanisme simple pour échanger des informations d'un processus à l'autre (modulo les questions de synchronisation des accès)
- La mémoire partagée permet de le faire.

Mémoire partagée

- Seule manière de partager une zone mémoire entre deux processus distincts : partagée par l'intermédiaire de son espace d'adressage
- La mémoire partagée est un mécanisme de communication qui n'implique pas de duplication de l'information (à l'inverse des fichiers, tubes, ... qui supposent une recopie de l'espace utilisateur vers le mode noyau). Donc, approprié pour le partage de gros volume de données
- Les segments de mémoire partagée ont une existence indépendante des processus :
 - Pas de destruction à la mort du dernier processus qui y accède
 - Un processus peut demander le rattachement d'un segment à son espace d'adressage. Il pourra ensuite y accéder en utilisant son adresse

Plan mémoire de 2 processus et mémoire partagée



(Jean François Pique – Université de la Méditerranée (Aix – Marseille II))

Implémentation en C sous Linux

Comme souvent, 2 implémentations :

- La mémoire partagée System V (`shmget(2)`, `shmop(2)`, etc.) est une ancienne API de sémaphores.
- La mémoire partagée POSIX offre une interface plus simple et mieux conçue ; d'un autre côté, la mémoire partagée POSIX est moins largement disponible (particulièrement sur d'anciens systèmes) que la mémoire partagée System V.
- POSIX propose aussi une extension (XSI) permettant l'utilisation des primitives System V.
- En TP on utilisera l'implémentation System V.

Dans les 2 cas :

- L'espace mémoire partagée doit être protégé par des sémaphores !
- Absence de destruction à la fin du processus créateur
- Philosophie fichier.
- La principale différence se trouve au niveau de la façon d'obtenir un descripteur

Implémentation en C sous Linux : System V

Les IPC SystemV offre une solution élégante

- Une fonction `shmget()` permet à partir d'une clé d'obtenir l'identifiant d'un segment de mémoire partagée existant ou d'en créer un au besoin
- Une fonction `shmat()` permet d'attacher le segment de mémoire partagée dans l'espace d'adressage du processus
- Une fonction `shmdt()` permet de détacher le segment si on ne l'utilise plus
- Une fonction `shmctl()` permet de paramétrier ou de supprimer un segment de mémoire partagée.

Les constantes et les prototypes des fonctions sont définis dans `sys/shm.h`

shmget et shmat

```
int shmget(key_t cle, int taille, int option);
```

- cle : clé obtenue avec ftok() ou IPC_PRIVATE
- taille : indique la taille du segment (inférieure ou égale si segment existant) en octets
- option : IPC_CREAT et IPC_EXCL et 9 bits d'autorisation
- La fonction retourne un identificateur du segment

Remarque : la taille est arrondie au multiple supérieur de la taille des pages mémoire du système (4KO sur un PC) donc éviter de créer trop de petits segments... Préférer une grosse structure ou un tableau

```
void *shmat(int shmid, const void *adr, int option);
```

- shmid est l'identificateur de segment de mémoire partagée
- adr est l'adresse désirée de l'attachement (uniquement pour écrire un simulateur ou un débuggeur), ou NULL si on laisse le système le placer
- option : cf docs (SHM_RND, SHM_RDONLY, ...)
- La fonction retourne un pointeur sur la première adresse du segment ou -1 si erreur

shmdt et shmctl

```
int shmdt(const void *adr);
```

Détachement du segment dont la demande d'attachement par shmat a été réalisée à l'adresse adr

```
int shmctl(int shmid, int op, struct shmid_ds *p_shm);
```

Réalise différentes opérations de contrôle. Les valeurs de op sont IPC_RMID, IPC_STAT, IPC_SET, ...

Exemple

```
typedef struct {
    int tab[NB_MAX];
} MEM_TAB;
MEM_TAB *ptr;

/* On cree la memoire partagee */
if ((shmid = shmget(IPC_PRIVATE, sizeof(MEM_TAB), IPC_CREAT | 0600)) == -1)
    gestion_erreur("Erreur de creation de la memoire partagee");

/* Attachement de la memoire partagee */
if ((ptr = (MEM_TAB *) shmat(shmid, NULL, 0)) == NULL)
    gestion_erreur("Erreur au l attachement de la memoire");

for (i = 0; i < NB_MAX; i++)
    ptr->tab[i] = i;

/* On detache la memoire partagee */
if (shmctl(ptr) == -1)
    gestion_erreur("Erreur de detachment de la memoire partagee");

/* On rend la memoire partagee */
if (shmctl(shmid, IPC_RMID, NULL) == -1)
    gestion_erreur("Erreur avec shmctl a la destruction");
.
```

Implémentation en C sous Linux : POSIX

- `shm_open` : Créer et ouvrir un nouvel objet, ou ouvrir un objet existant. Semblable à `open()`. Retourne un descripteur de fichiers identifiant la mémoire partagée.
- `ftruncate` : Définir la taille de la mémoire partagée (par défaut nulle)
- `mmap` : Projeter l'objet en mémoire partagée dans l'espace d'adresses virtuel du processus appelant.
- `munmap` : Déprojeter l'objet en mémoire partagée de l'espace d'adresses virtuel du processus appelant.
- `shm_unlink` : Supprimer le nom d'un objet en mémoire partagée.
- `close` : Fermer le descripteur de fichier alloué avec `shm_open` lorsqu'on en a plus besoin.
- `fstat` : Obtenir une structure `stat` décrivant l'objet en mémoire partagée (similaire au fichier)
- `fchown, fchmod` : `chmod, chown` pour la mémoire partagée

Recommendations

- L'attachement à la mémoire partagée est propagé au fils lors de sa création avec `fork()`
Toujours penser à détacher le processus créé si nécessaire.
- L'attachement à la mémoire partagée n'est pas conservé par un appel à `execve()`
Penser à attacher le processus si nécessaire
- La création d'un segment de mémoire partagée est arrondie au multiple supérieur de la taille d'une page (4 Koctets sur un PC)
 - Éviter les petits segments de mémoire partagée
 - Préférer de les regrouper au sein d'un tableau ou d'une structure
- La mémoire partagée est une ressource critique et les accès doivent être synchronisés. **Utiliser un sémaphore d'exclusion mutuel.**