UNIX

Programmation Système


```
int main(void){
  int pid1=0;
  int status;
  if ((pid1=fork()) == -1){
    perror("Creation fils ");
    exit(1);
  }
  if (pid1 == 0) {
    /* Recouvrement du fils */
    execlp("monProg", "monProg", NULL);
    perror("Lancement monProg");
    exit(2);
  }
```


ENAC

PROGRAMMATION SYSTEME UNIX : PLAN

Programmation système

Rappels de C

L'API système UNIX

CADRE DU COURS

Programmation système

- Système : Linux
- Langage de programmation : C
- Compilateur : gcc

PROGRAMMATION SYSTEME

COMMENT?

Faire appel dans les programmes utilisateurs aux services spécifiques d'un système d'exploitation

- création et coordination des tâches (programmation multitâche/concurrente)
- gestion spécifique des données
- communication
- autres ...

PROGRAMMATION SYSTEME

COMMENT?

Chaque système d'exploitation offre une API (Interface de programmation)

- Bibliothèque de fonctions (appels système)
- Spécifique au système d'exploitation
- Fonctions appelées dans les programmes
- Le processus utilisateur exécute du code système (mode noyau)

LA NORMALISATION

UNIX est une "famille" de systèmes, conformes à des normes

- Single UNIX Specification version 3 (POSIX 2001)
- XPG4
- POSIX 1003.1
- ...

LA NORME POSIX

Portable **O**perating **S**ystem **I**nterface uni**X**

- Spécification d'un ensemble de fonctions permettant de solliciter les services de base d'un système d'exploitation
- Ne constitue pas la définition d'un système d'exploitation
- Objectif : garantir le développement d'applications portables au niveau du code source, entre les systèmes d'exploitation conformes à la norme en masquant les spécificités du système

PROGRAMMATION SYSTEME

LA NORME POSIX

Portable **O**perating **S**ystem **I**nterface uni**X**

- Fournit une liste de points d'accès aux services du système
- Pour chaque fonction, le comportement attendu dans les différentes circonstances susceptibles de se produire est complètement défini.

PROGRAMMATION SYSTEME UNIX : PLAN

Programmation système

Rappels de C

- La bibliothèque standard C
- Les arguments de main()
- Paramètres d'un exécutable
- Environnement shell d'un processus
- Gestion des erreurs

L'API système UNIX

LA BIBLIOTHÈQUE STANDARD C

- Fonctions déclarées dans les fichiers d'entête :

assert.h ctype.h errno.h float.h limits.h
locale.h math.h setjmp.h signal.h stdarg.h
stddef.h stdio.h stdlib.h string.h time.h

- Disponibles quelque soit le système *(Unix, Windows, ...)* et l'environnement de développement *(gcc, Borland C, Visual Studio, ...)*

LES ARGUMENTS DE main( )

- main( ) : fonction à nombre de paramètres variable
- Prototypes de la fonction main()

```
int main ( void );

int main ( int argc , char **argv );

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

PARAMÈTRES D'UN EXÉCUTABLE

Passer des paramètres au lancement d'un exécutable

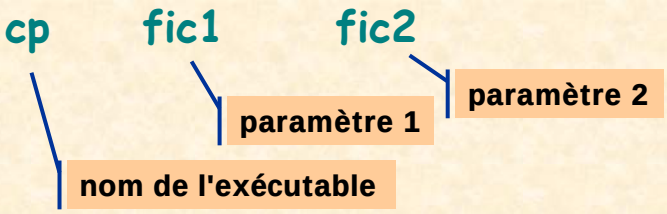
- Arguments de la fonction main()

  int main (int argc , char **argv)

  argc : nombre de chaînes de caractères de la ligne de commande

  argv : la ligne de commande, terminée par NULL

PARAMÈTRES D'UN EXÉCUTABLE




```
argc ---> 3
argv[0] ---> cp\0
argv[1] ---> fic1\0
argv[2] ---> fic2\0
argv[3] ---> NULL
```

PARAMÈTRES D'UN EXÉCUTABLE

Affichage d'un fichier dont le nom est passé en argument

```
int main ( int argc, char** argv ) {
  int fd, nb;
  char buf[512];
  if  ( argc != 2 ) {
          write( 2, "Nb arguments incorrect\n", 25);  exit(1); }
  if  ( ( fd = open( argv[1] , O_RDONLY ) ) == -1 ) {
          perror("Ouverture");  exit(1); }
  while ( ( nb = read( fd, buf, sizeof(buf) ) ) > 0 ) {
          write( 1, buf, nb ); }
  close(fd);   return(0);
}
```


ENVIRONNEMENT SHELL D'UN PROCESSUS

Accès à l'environnement shell d'un processus

- Arguments de la fonction main()

int main (int argc , char **argv , char **envp)

argc : nombre de chaînes de caractères de la ligne de commande

argv : la ligne de commande, terminée par NULL

envp : liste des variables d'environnement, terminée par NULL

ENVIRONNEMENT SHELL D'UN PROCESSUS

Affichage des variables d'environnement

```
int main ( int argc, char** argv, char** envp ) {
   int i=0;
   for (i=0; envp[i] != NULL; i++ )
        printf( "%s\n", envp[i] );
   printf("\n%d variables d'environnement initialisées\n", i );
   return(0);
}
```


ENVIRONNEMENT SHELL D'UN PROCESSUS

Affichage des variables d'environnement : extrait du résultat

```
LC_TIME=en_US.ISO8859-1
SUN_SUNRAY_TOKEN=Payflex.500940e200130100
PATH=/usr/bin:/opt/java/bin:/usr/local/bin:/usr/sfw/bin:/o
pt/sfw/bin:/usr/ccs/bin:/opt/staroffice6.0/program:/opt/Ac
robat5/bin:/opt/Netscape:/opt/mozilla:/opt/eclipse/bin:/op
t/sudo.1.6.7/bin:/usr/local/teTeX/bin/sparc-sun-
solaris2.9:/opt/SUNWspro/bin:/opt/gnat3.15p/bin:/opt/fuzz2
000:/opt/flex-254/bin:/opt/gawk-
311/bin:/usr/dt/bin:/usr/openwin/bin:/bin:/usr/bin:/usr/uc
b
OSTYPE=solaris
PWD=/home/people/profs/luter/PROG_SYST

49 variables d'environnement initialisées
```

ENVIRONNEMENT SHELL D'UN PROCESSUS

- Accès au contenu d'une variable d'environnement

```
char *getenv ( const char *name );
```

paramètre : nom de la variable
valeur de retour : contenu de la variable ou NULL (erreur)

```
#include <stdlib.h>
int main (void) {
    printf("Le shell courant est %s\n", getenv("SHELL"));
    return 0;
}
```


ENVIRONNEMENT SHELL D'UN PROCESSUS

- Créer ou modifier une variable d'environnement

```
int putenv ( const char *cmde );
```

paramètre : commande d'initialisation
valeur de retour : 0 ou -1 (erreur)

```
#include <stdlib.h>
int main (void) {
    putenv ( "MAVAR = 123");
    printf("MAVAR contient: %s\n", getenv("MAVAR"));
    return 0;
}
```

GESTION DES ERREURS

- Tout accès à une ressource système (fichier, mémoire, ...) est susceptible d'échouer !!!
- Il est **indispensable** de tester le code retour de chaque appel système et d'effectuer le traitement approprié pour éviter tout comportement erratique des programmes
- La bibliothèque standard C offre des fonctions qui permettent de récupérer les codes erreur du système

GESTION DES ERREURS

- Affichage d'un message utilisateur et du message d'erreur système sur la console

```
#include <stdio.h>
void  perror ( const char *msg );
```

paramètre: le message utilisateur

GESTION DES ERREURS

- Affichage d'un message utilisateur et du message d'erreur système sur la console : exemple

```
#include <stdio.h>
...
if ( ( fd = open( "fic", O_RDONLY ) ) == -1 ) {
      perror("ouverture fic");
      exit(1);
      }
```

Si le fichier n'existe pas, affichage du message :

" ouverture fic : no such file or directory"

GESTION DES ERREURS

- Accès au numéro d'erreur système : variable globale **int errno** déclarée dans **errno.h**

```
if ( ( fd = open( "fic",O_RDONLY ) ) == -1 ) {
      switch(errno) {
            case NOENT:  /* fichier inexistant */
                  fd = open ("fic", O_RDWR | O_CREAT, 0640);
                  break;
            default:   /* autre erreur */
                  perror("ouverture fic");
                  exit(1);
            }
```

PROGRAMMATION SYSTEME UNIX : PLAN

L'API système UNIX

- Généralités
- Fichiers
- Processus
- Communication multi-tâche
  - Programmation concurrente
  - Mécanismes UNIX
  - Critères de choix
  - Tubes
  - IPC
  - Signaux
- Threads Posix

PROGRAMMER EN C AVEC L'API UNIX : GENERALITES

GENERALITES

PROGRAMMER EN C AVEC L'API UNIX : GENERALITES

UN PEU D'AIDE ???

man nom_de_la_fonction

man fork

man -s2 nom_de_la_fonction

man -s2 time --> aide sur la fonction time()

man -s2 intro --> informations générales sur les appels système, entre autres les codes erreurs

PROGRAMMER EN C AVEC L'API UNIX : GENERALITES

LES FICHIERS D'ENTETE SPECIFIQUES UNIX

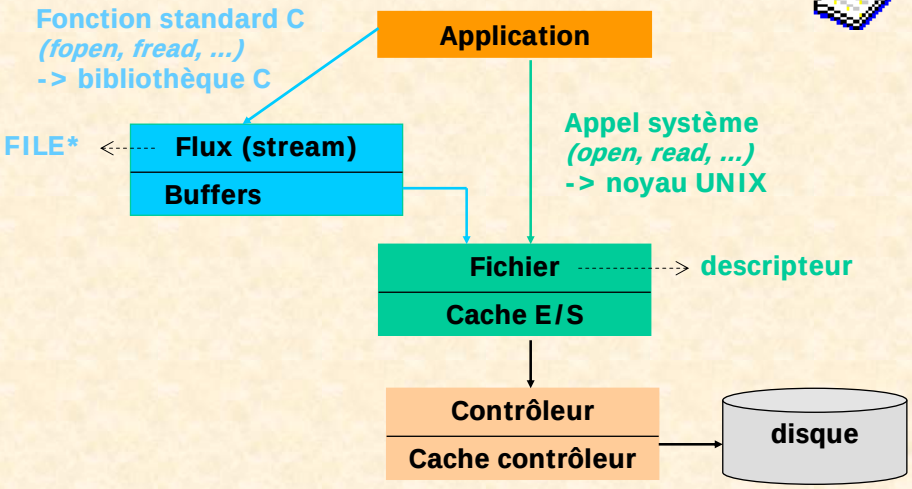
- **unistd.h**
  - Gestion des processus
  - Gestion des fichiers *(sauf open() et creat())*
- **fcntl.h**
  - open() et creat()
- **signal.h** gestion des signaux
- **sys/ipc.h** IPC

LES FICHIERS

NIVEAUX DE GESTION

- Gestion standard bibliothèque C ("haut niveau")
  - normalisée (C ANSI) + + +
  - bufferisation + + +
  - entrées/sorties formatées + + +
  - gestion par flux (stream) = descripteur de fichier + buffer
- Gestion par appel système Unix ("bas niveau")
  - appel direct aux primitives spécifiques du noyau
  - pas de bufferisation - - -
  - entrées/sorties non formatées - - -
  - accès aux fichiers spéciaux + + +
  - accès à des fonctionnalités spécifiques + + +

NIVEAUX DE GESTION



GESTION DE "BAS NIVEAU"

- Un fichier est manipulé par son descripteur système
  - une table des descripteurs par processus (héritée) : correspondance avec le fichier physique + pointeurs
  - 3 descripteurs ouverts par défaut au lancement d'un processus:
    0 : entrée standard, **1** : sortie standard
    2 : sortie d'erreur standard
  - nombre entier alloué à l'ouverture du fichier = entrée dans la table des descripteurs
  - L'entrée dans la table des descripteurs est supprimée à la fermeture du fichier

DESCRIPTEUR DE FICHIER

- Concerne tous les types de « fichiers »
  - fichiers ordinaires (fichiers disques)
  - fichiers spéciaux :
    - tubes de communication (pipelines)
    - sockets (réseau)
    - répertoires
    - périphériques

DESCRIPTEUR DE FICHIER

- Opérations associées aux descripteurs
  - Création, ouverture, fermeture
  - lecture, écriture
  - déplacement dans un fichier
  - duplication (--> redirections)
  - entrées/sorties asynchrones
  - opérations de contrôle sur les périphériques (ioctl)

DESCRIPTEUR DE FICHIER et FLUX

➢ dans /usr/include/stdio_impl.h:

```
struct __FILE_TAG {   /* une définition du type FILE */
    ssize_t _cnt;     /* number of available characters in buffer */
    unsigned char  *_ptr;  /*next character from/to here in buffer */
    unsigned char  *_base;  /* the buffer */
    unsigned char  _flag;    /* the state of the stream */
    unsigned char  _file;  /* UNIX System file descriptor */
    unsigned       __orientation:2;  /* the orientation of the stream */
    unsigned       __ionolock:1;    /* turn off implicit locking */
    unsigned       __seekable:1;    /* is file seekable? */
    unsigned       __filler:4;
};
```


PRIMITIVES D'ACCES AUX FICHIERS

➢ Ouverture / Création

```
#include <fcntl.h>
int  open ( const char *path, int oflags, mode_t mode );
```

path : le chemin d'accès au fichier

oflags : flags d'ouverture (O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_CREAT, O_TRUNC, O_EXCL, O_NONBLOCK)

mode : droits d'accès au fichier (seulement si mode = O_CREAT)

valeur de retour : le descripteur du fichier ou -1 (erreur)

PRIMITIVES D'ACCES AUX FICHIERS

- Lecture de données

```
size_t read ( int fd, char *buffer, size_t nb );
```

fd : le descripteur du fichier
buffer : adresse du buffer de réception des données lues
nb : nombre d'octets à lire
valeur de retour : le nombre d'octets lus ou -1 (erreur)

- Ecriture de données

```
size_t write ( int fd, char *buffer, size_t nb );
```

fd : le descripteur du fichier
buffer : adresse du buffer des données à écrire
nb : nombre d'octets à écrire
valeur de retour : le nombre d'octets écrits ou -1 (erreur)

PRIMITIVES D'ACCES AUX FICHIERS

- Déplacement du pointeur (position courante)

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

fd : le descripteur du fichier
offset : le déplacement (nombre d'octets)
methode : SEEK_SET, SEEK_CUR, SEEK_END
valeur de retour : l'offset ou -1 (erreur)

- Fermeture de fichier

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

fd : le descripteur du fichier
valeur de retour : -1 (erreur)

PRIMITIVES D'ACCES AUX FICHIERS

chmod( ), fchmod( ) Modification des droits d'accès

umask( ) Modification du masque

fcntl( ) Contrôle d'accès

fsync( ) Synchronisation des caches

fstat( ), stat( ) Récupération d'informations

...

LECTURE / ECRITURE DANS UN FICHIER

Exemple 1

EN PRATIQUE ...

Prise en main de l'environnement de programmation

- **Réalisation d'un filtre de conversion**

LES PROCESSUS

LES PROCESSUS

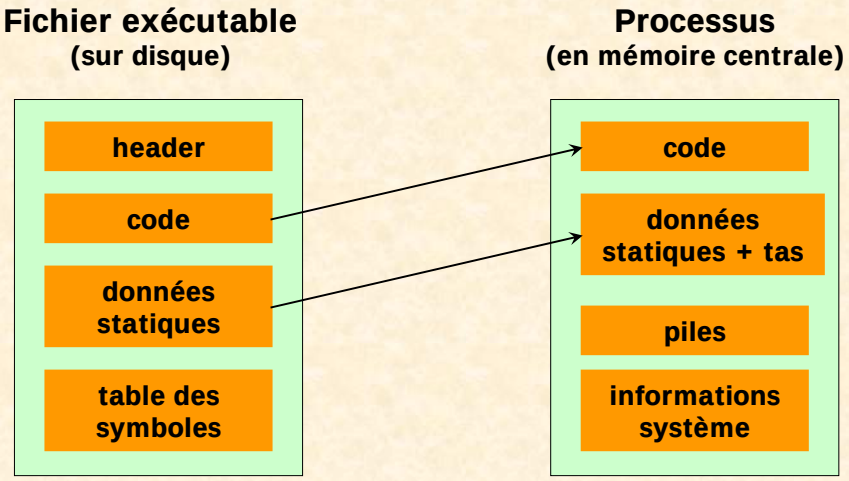
PROCESSUS

= image mémoire d'un programme en cours d'exécution

Espace d'adressage:

- code
- données
- piles utilisateur et système
- informations système (environnement, bloc de contrôle (PCB))

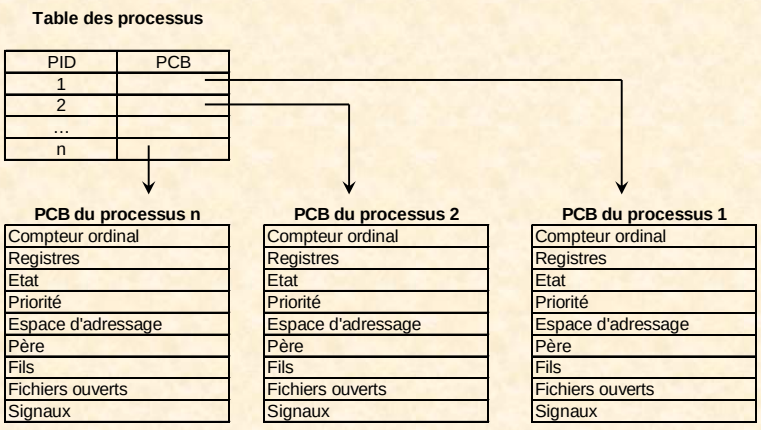
CHARGEMENT EN MEMOIRE



ESPACE D'ADRESSAGE D'UN PROCESSUS LINUX

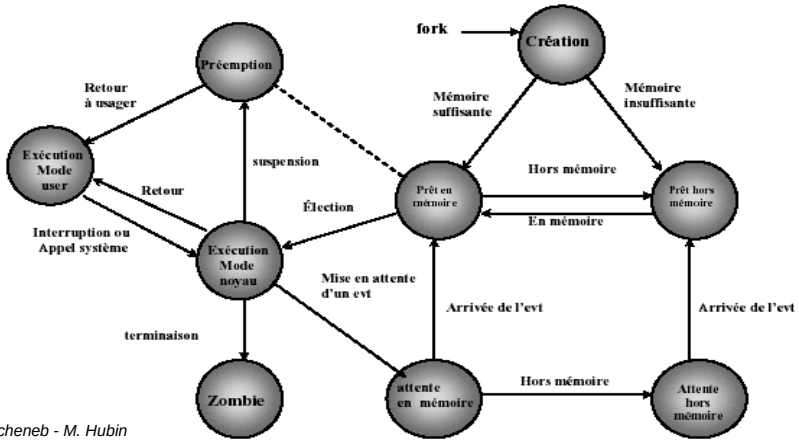
TABLE DES PROCESSUS

- Gérée par le système



ORDONNANCEMENT DES PROCESSUS

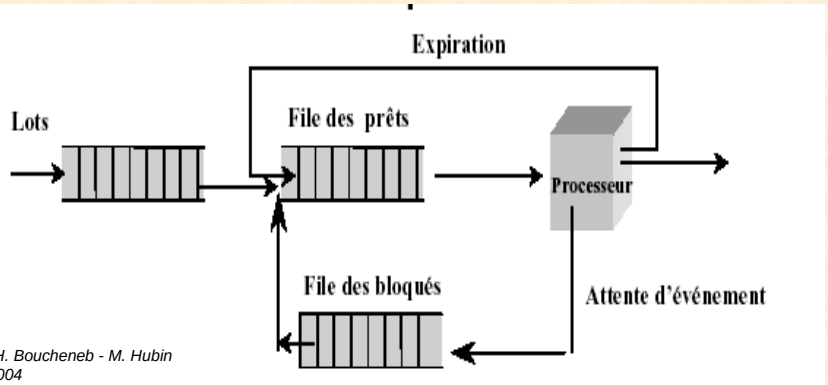
Cycle de vie



Source H. Boucheneb - M. Hubin
EPM - 2004

ORDONNANCEMENT DES PROCESSUS

Cycle de vie



Source H. Boucheneb - M. Hubin
EPM - 2004

LES PROCESSUS

CARACTERISTIQUES DES PROCESSUS

- Un numéro d'identification unique : PID (Process IDentifier)
- Un état (prêt, en cours d'exécution, en attente, ...)
- L'état des registres lors de la dernière suspension
- Une priorité
- Les ressources allouées (fichiers ouverts, mémoires, ...)
- Les signaux à capturer, à masquer, à ignorer et les actions associées
- Autres informations: son processus père, ses processus fils, son propriétaire, son groupe, ses variables d'environnement, ...

LES PROCESSUS

CARACTERISTIQUES DES PROCESSUS

- **Processus père et processus fils**

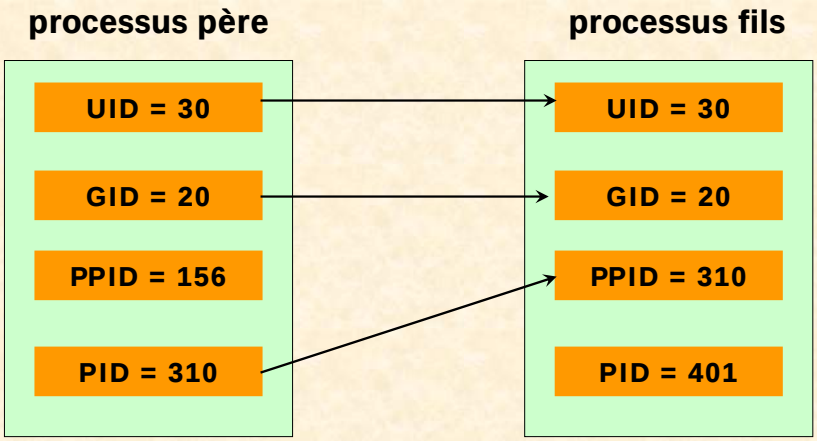
Un processus est toujours créé par un autre processus (à l'exception du processus de démarrage du système)

- **Lancement des commandes**
  - commande interne = code interne au shell
    ---> pas de nouveau processus
  - commande externe (exécutable)
    ---> le shell crée un processus fils
  - script
    ---> selon le mode de lancement

CARACTERISTIQUES DES PROCESSUS

- **PID** (Process IDentifier)
  - numéro unique
  - process **init** : PID 1
- **PPID** (Parent Process IDentifier)
  - numéro du processus père
- **UID** (User IDentifier) = utilisateur réel ***---> droits***
  - identifiant de l'utilisateur ayant lancé le processus
- **GID** (Group IDentifier) = groupe réel ***---> droits***
  - identifiant du groupe

CARACTERISTIQUES DES PROCESSUS : exemple



DROITS D'ACCÈS AUX RESSOURCES

Pour le propriétaire, le groupe et les autres utilisateurs

- lecture (r)
- écriture (w)
- exécution (x)

Un processus dispose des droits d'accès de son propriétaire effectif

DROITS D'ACCÈS AUX RESSOURCES

- tout processus lancé par un utilisateur dispose du numéro (RUID) de cet utilisateur (ou utilisateur réel)
- le processus dispose en plus du numéro de l 'utilisateur effectif (EUID), celui avec les droits duquel s 'exécute le programme.
- En règle générale, RUID = EUID, c 'est à dire que le processus se voit attribuer les droits de l 'utilisateur qui l 'a lancé.
- Les mêmes notions existent au niveau du groupe (RGID, EGID)

DROITS D'ACCÈS AUX RESSOURCES

- **bit setuid (s)**
  - Permet à un utilisateur (RUID) d 'avoir les droits d 'accès du propriétaire (EUID) d 'un programme le temps de son exécution
  - RUID != EUID
  - Positionné au niveau des droits du propriétaire à la place du droit en exécution
  - Exemple :

    passwd offre les droits de root sur /etc/passwd

DROITS D'ACCÈS AUX RESSOURCES

- **bit setgid (s)**
  - Permet à un utilisateur (RGID) d 'avoir les droits d 'accès du groupe propriétaire (EGID) d 'un programme le temps de son exécution
  - Les fichiers créés dans un répertoire setgid auront tous le même groupe que le répertoire

DROITS D'ACCÈS AUX RESSOURCES

- **"sticky bit" (t)**
  - pour les fichiers exécutables, permet de garder le code en mémoire après la fin de l 'exécution (accélération du lancement)
  - pour les répertoires, permet à un utilisateur d 'écrire dans le répertoire, mais ne l 'autorise à supprimer que les fichiers lui appartenant

    exemple : /tmp
  - Positionné au niveau des droits des autres utilisateurs à la place du droit en exécution

CARACTERISTIQUES DES PROCESSUS : PRIMITIVES

- **Récupération du PID**

  pid_t **getpid** (void);

  valeur de retour : le PID du processus courant
- **Récupération du PPID**

  pid_t **getppid** (void);

  valeur de retour : le PID du père du processus courant
- **Récupération du UID**

  pid_t **getuid** (void);

  valeur de retour : le UID du processus courant
- **Récupération du GID**

  pid_t **getgid** (void);

  valeur de retour : le GID du processus courant

CARACTERISTIQUES DES PROCESSUS : PRIMITIVES

Exemple 3

DEMARRAGE DU SYSTEME

1 - Mise en route de l'ordinateur

- le chargeur d'amorçage lance le système à partir du disque (ou autre support)
- création du processus ordonnanceur (PID 0)

2 - Duplication de l'ordonnanceur

- mise en place du système de fichiers, ...
- création du processus init (PID 1) et du démon de gestion des pages (PID 2)

3 - processus init

- En fonction du niveau de démarrage, lance les services système requis et reste en mémoire

DEMARRAGE DU SYSTEME

CREATION D'UN PROCESSUS

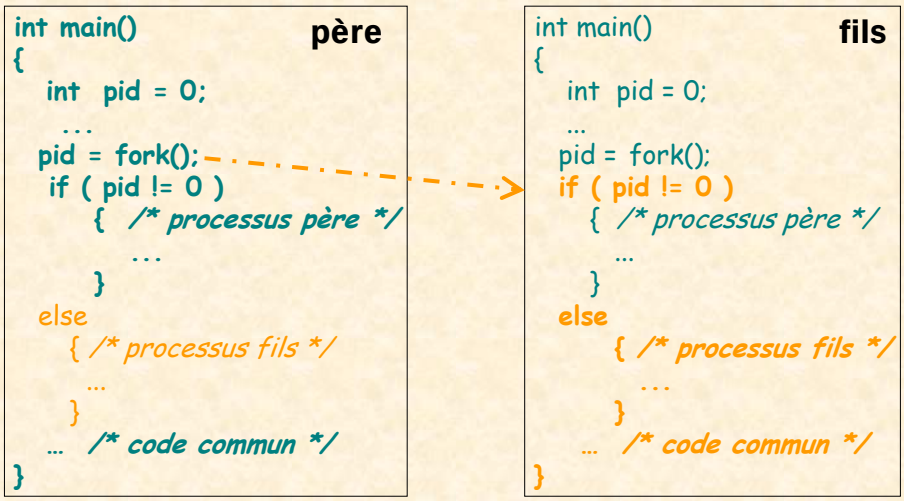
```
pid_t fork ( void );
```

valeur de retour : le PID du processus fils créé ou -1 (erreur)

Duplique le processus père : le fils en est une copie exacte

- Même code
- Recopie des zones de données
- Recopie de l'environnement :
  - descripteurs de fichiers ouverts
  - répertoire courant
  - priorité
  - masque des signaux
- Partage des pointeurs de fichiers ouverts 

CREATION D'UN PROCESSUS



TERMINAISON D'UN PROCESSUS

void **exit** (int status);

paramètre : le code retour du processus

- Ne revient jamais au programme appelant
- Libère toutes les ressources du processus
- Les processus fils sont "adoptés" par init (PID 1)
- Remarque : l'instruction **return** de la fonction main() invoque **exit()**

Terminaison d'un processus avec création d'un fichier core

void **abort** (void);

SYNCHRONISATION SUR LA FIN D'UN FILS

Quand un processus se termine, il reste à l'état "zombi" tant que son père ne s'est pas synchronisé

```
pid_t wait ( int *status );
```

paramètre : le code retour du processus fils
valeur de retour : le PID du fils ou -1 (pas de fils)

- si le processus n'a pas de fils : -1 et errno = ECHLD
- si le processus a au moins un fils "zombi" : le fils disparaît de la table des processus
- s'il y a au moins un fils en cours d'exécution (et pas de fils zombi) : attente du signal de fin du fils
- autant d'appels wait() nécessaires que de process fils
- possibilité d'attendre un fils particulier : **waitpid()**

SYNCHRONISATION SUR LA FIN D'UN FILS

```
int main(void) {
        int status, pid;
        if ( (pid  = fork() ) == 0) {
                /* processus fils */
                sleep(5);
                exit(0);
        }
        wait( &status);
        return 0;
}
```

CREATION DE PROCESSUS

Exemples 5 et 6

RECOUVREMENT DU CODE D'UN PROCESSUS

- Exécution d'un nouveau programme exécutable sans création de nouveau processus
- Le code et les données du processus courant sont remplacés par ceux du nouveau programme
- Conservation de l'environnement courant (fichiers, signaux, ...)
- Pas de retour à la fonction appelante (code écrasé !)
- Famille de fonctions exec*XX*()

RECOUVREMENT DU CODE D'UN PROCESSUS

Fonction	Recherche dans le PATH	Passage des arguments	Variables d'environnement
execl ()	NON	Liste	conservées
execlp ()	OUI	Liste	conservées
execle ()	NON	Liste	nouvelles
execv ()	NON	Tableau	conservées
execvp ()	OUI	Tableau	conservées
execve ()	NON	Tableau	nouvelles

RECOUVREMENT DU CODE D'UN PROCESSUS

Exemple 7

EXECUTION D'UNE COMMANDE SHELL

int **system (const char * commande);**

paramètre : la ligne de commande

valeur de retour : le code retour de la commande ou -1

```
system ( "clear" );
```


Illustration du concept de processus

processus

fork()

- **Transfert du contenu d'un fichier à travers des tubes : étape 0**

LES REDIRECTIONS

MECANISME

- Modification de la table des descripteurs de fichiers
- Changement du fichier physique associé à un descripteur
- Mécanisme :

  1- fermeture d'un descripteur --> libère son entrée dans la table des descripteurs

  2- duplication du descripteur d'un fichier ouvert : affecte au fichier un descripteur disponible dans la table

LES REDIRECTIONS

EXEMPLE

- Redirection de la sortie standard

0	clavier
1	écran
2	écran
3	/home/luter/toto

0	clavier
1	/home/luter/toto
2	écran
3	/home/luter/toto

fermeture desc. 1
+
duplication desc. 3

LES REDIRECTIONS

INTERET

- Réutilisation de code existant avec de nouvelles entrées/sorties
- Souvent combiné avec les fonctions exec*XX*( )
- Mécanisme shell du pipeline = inclut une redirection

LES REDIRECTIONS

PRIMITIVES

int **dup (** int fd**);**

fd : le descripteur du fichier à rediriger

valeur de retour : le nouveau descripteur (le plus petit disponible) ou -1 (erreur)

int **dup2 (** int fd, int newfd **);**

fd : le descripteur du fichier à rediriger

newfd : le nouveau descripteur (attention, s'il est déjà attribué, il est préalablement fermé)

valeur de retour : le nouveau descripteur ou -1 (erreur)

Remarque : `dup2(fd1, fd2);` Non interruptible, est équivalent à

```
close(fd2);
fd2=dup(fd1);
```


LES REDIRECTIONS

REDIRECTIONS

Exemple 2

MECANISMES UNIX DE COMMUNICATION MULTI-TÂCHE

MECANISMES UNIX

- Communication
  - fichiers
  - tubes (pipes)
  - tubes nommés
  - mémoire partagée
  - files de messages
  - sockets (réseau)
- Synchronisation
  - ensembles de sémaphores
  - signaux
  - verrous

COMMUNICATION MULTI-TACHE

MECANISMES UNIX

- Critères de choix
  - Communication ou synchronisation
  - Volume d'information
  - Rapidité des échanges
  - Protection des données
  - Conservation des données
  - Structuration des données

COMMUNICATION MULTI-TACHE

LES FICHIERS

- Critères de choix
  - Volume d'information **---> important**
  - **Communication** ou synchronisation
  - Rapidité des échanges **---> lente**
  - Protection des données **---> à synchroniser**
  - Conservation des données **---> oui**
    (Données indépendantes des processus)
  - Structuration des données **---> non**

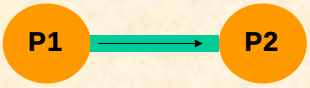
COMMUNICATION MULTI-TACHE

LES TUBES

- Critères de choix
  - Volume d'information **---> moyen**
  - **Communication** ou synchronisation
  - Rapidité des échanges **---> rapide**
  - Protection des données **---> synchronisé**
  - Conservation des données **---> non**
  - Structuration des données **---> non**

COMMUNICATION MULTI-TACHE

LES TUBES



- Mécanisme de communication unidirectionnelle entre processus
- Fonctionnement en FIFO
- Implantés en mémoire
- Gérés par le système de fichier
  - descripteurs dans la table des fichiers
  - droits
  - accès par les primitives de gestion des fichiers
  - mais ... pas de déplacement dans un tube

COMMUNICATION MULTI-TACHE

LES TUBES

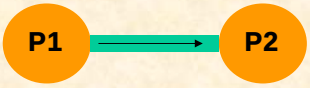
Synchronisation en lecture :

- le tube contient des données : elles sont consommées
- le tube ne contient pas de données :
  - existence d'un processus écrivain : attente
  - pas de processus écrivain* : retour et fin du tube (la fermeture du descripteur en écriture place un EOF dans le tube)

**processus écrivain : processus ayant un descripteur en écriture ouvert sur le tube*

COMMUNICATION MULTI-TACHE

LES TUBES



Synchronisation en écriture :

- place disponible dans le tube : écriture
- tube plein : attente
- pas de processus lecteur* : fin du processus écrivain (signal SIGPIPE) et fin du tube (message "Broken pipe")

**processus lecteur : processus ayant un descripteur en lecture ouvert sur le tube*

COMMUNICATION MULTI-TACHE

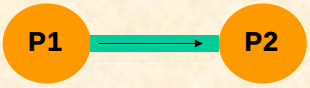
LES TUBES

2 types de tubes

- tube sans nom
  - communication entre processus de même famille
  - accessible uniquement par ses descripteurs
- tube nommé
  - communication entre processus "indépendants"
  - accessible par son chemin d'accès dans le système de fichiers

COMMUNICATION MULTI-TACHE

LES TUBES SANS NOM



- Création

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

fd : tableau de 2 descripteurs
fd[0] : lecture
fd[1] : écriture
valeur de retour : 0 (OK) ou -1 (erreur)

Exemples 8 et 9

LES TUBES NOMMES

➢ Création

#include <sys/stat.h>

int **mkfifo** (const char *nom, mode_t mode);

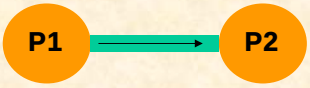
nom : chemin d'accès du tube
mode : droits d'accès aux fichiers
valeur de retour : 0 (OK) ou -1 (erreur)

ou

int **mknod** (const char *nom, mode_t mode2);

mode2 = mode | S_IFIFO

LES TUBES NOMMES



➢ Ouverture

int **open** (const char *path, int oflags, mode_t mode);

(cf. accès fichiers)

➢ Obtention d'un flux à partir d'un descripteur ouvert

FILE * **fdopen** (int fd , const char *mode);

fd : descripteur de fichier
mode : droits d'accès aux fichiers (cf. modes de fopen())
valeur de retour : flux ou NULL (erreur)

COMMUNICATION MULTI-TACHE

LES TUBES NOMMES

Exemple 10

EN PRATIQUE ...

Illustration du concept de processus

création de processus

recouvrement de processus

accès fichier

redirection

tubes

- **Transfert du contenu d'un fichier à travers des tubes : étapes 1 et 2**

COMMUNICATION MULTI-TACHE

LES SIGNAUX

- Critères de choix
  - Volume d'information **---> pas de données**
  - Communication ou **synchronisation**
  - Rapidité des échanges **---> asynchrone**
  - Protection des données **---> -**
  - Conservation de l'information **---> non**
  - Structuration des données **---> -**

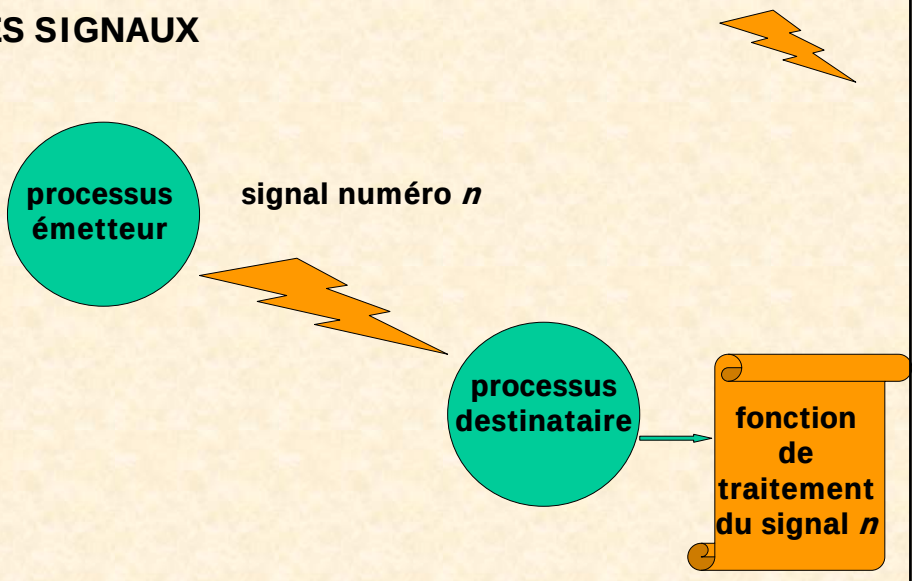
COMMUNICATION MULTI-TACHE

LES SIGNAUX

- Interruption logicielle
- Mécanisme asynchrone : à la réception d'un signal, le processus exécute un traitement (code d'une fonction)
- Modes de traitement
  - ignorance volontaire : pas pour tous les signaux
  - traitement par défaut : en général, exit()
  - exécution d'une fonction utilisateur (handle)

COMMUNICATION MULTI-TACHE

LES SIGNAUX



COMMUNICATION MULTI-TACHE

LES SIGNAUX

Catégories de signaux

- exceptions : lancées par le noyau en cas d'exécution anormale (/0, violation de segment, instruction illégale, ...)
- interaction du terminal : CTRL-C, CTRL-Z, ...
- interactions entre processus : synchronisation , réveil, alarme, ...

LES SIGNAUX

- Prise en compte d'un signal

  La prise en compte d'un signal n'est pas immédiate (mécanisme asynchrone)

  - processus actif : le signal est pris en compte lors d'une transition entre l'état "actif noyau" et l'état "actif utilisateur"
  - processus endormi : la réception d'un signal fait passer le processus à l'état "prêt". Le signal sera traité quand le processus sera actif.

LES SIGNAUX SOLARIS 9

Name	Value	Default	Event
SIGHUP	1	Exit	Hangup
SIGINT	2	Exit	Interrupt
SIGQUIT	3	Core	Quit
SIGILL	4	Core	Illegal Instruction
SIGTRAP	5	Core	Trace or Breakpoint Trap
SIGABRT	6	Core	Abort
SIGEMT	7	Core	Emulation Trap
SIGFPE	8	Core	Arithmetic Exception
SIGKILL	9	Exit	Killed
SIGBUS	10	Core	Bus Error
SIGSEGV	11	Core	Segmentation Fault
SIGSYS	12	Core	Bad System Call

LES SIGNAUX SOLARIS 9

Name	Value	Default	Event
SIGPIPE	13	Exit	Broken Pipe
SIGALRM	14	Exit	Alarm Clock
SIGTERM	15	Exit	Terminated
SIGUSR1	16	Exit	User Signal 1
SIGUSR2	17	Exit	User Signal 2
SIGCHLD	18	Ignore	Child Status Changed
SIGPWR	19	Ignore	Power Fail or Restart
SIGWINCH	20	Ignore	Window Size Change
SIGURG	21	Ignore	Urgent Socket Condition
SIGPOLL	22	Exit	Pollable Event
SIGSTOP	23	Stop	Stopped (signal)
SIGTSTP	24	Stop	Stopped (user)

LES SIGNAUX SOLARIS 9

Name	Value	Default	Event
SIGCONT	25	Ignore	Continued
SIGTTIN	26	Stop	Stopped (tty input)
SIGTTOU	27	Stop	Stopped (tty output)
SIGVTALRM	28	Exit	Virtual Timer Expired
SIGPROF	29	Exit	Profiling Timer Expired
SIGXCPU	30	Core	CPU time limit exceeded
SIGXFSZ	31	Core	File size limit exceeded
SIGWAITING	32	Ignore	Concurrency signal res.
SIGLWP	33	Ignore	Inter-LWP signal reserved
SIGFREEZE	34	Ignore	Check point Freeze
SIGTHAW	35	Ignore	Check point Thaw
SIGCANCEL	36	Ignore	Cancellation signal
SIGXRES	37	Ignore	Resource control exceeded

COMMUNICATION MULTI-TACHE

LES SIGNAUX

- Émettre un signal

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

pid : processus destinataire(s)
- **> 0** : PID du destinataire
- **= 0** : envoi à tous les processus du groupe
- **< 0** : émis par root : envoi du signal à tous les processus non système
  émis par un autre utilisateur : envoi du signal à tous ses processus

sig : numéro du signal

valeur de retour : 0 ou -1 (erreur)

COMMUNICATION MULTI-TACHE

LES SIGNAUX

- Émettre un signal

Exemple 14

COMMUNICATION MULTI-TACHE

LES SIGNAUX

- Paramétrer la réception d'un signal

2 méthodes de gestion des signaux :

C ANSI : gestion simplifiée. Les signaux sont systématiquement délivrés au processus. Chaque processus choisit le comportement à adopter (défaut, ignorance ou traitement personnalisé)

Posix : Chaque processus définit un masque des signaux (signaux bloqués qui ne seront pas délivrés). Pour les signaux non bloqués, choix du comportement à adopter.

COMMUNICATION MULTI-TACHE

LES SIGNAUX

- Paramétrer la réception d'un signal : C ANSI

Pour chaque signal pour lequel on veut modifier le comportement par défaut :

```
void (* signal ( int sig, void ( *action ( int ) ) ) ( int );
```

sig : numéro du signal

action : nom de la fonction de traitement *ou*
SIG_DFL : traitement par défaut *ou*
SIG_IGN : signal à ignorer (sauf SIGKILL)

valeur de retour : adresse de la fonction ou -1 (erreur)

La réception d'un signal annule le paramétrage

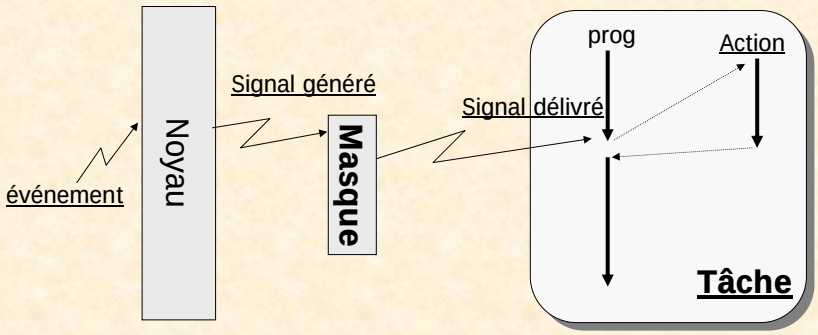
LES SIGNAUX

- Paramétrer la réception d'un signal : C ANSI

Exemple 12

LES SIGNAUX

- Paramétrer la réception d'un signal : POSIX



COMMUNICATION MULTI-TACHE

LES SIGNAUX

- Paramétrer la réception d'un signal : Posix

1- Mise en œuvre du masque de blocage des signaux

Préparation du masque

- création de l'ensemble de signaux (tout ou rien)
- ajout ou retrait de signaux

Application du masque

2- Définition du comportement spécifique des signaux

COMMUNICATION MULTI-TACHE

LES SIGNAUX

- Paramétrer la réception d'un signal : Posix

1- Mise en œuvre du masque de blocage des signaux

Préparation du masque

int **sigemptyset (sigset_t * ensemble);**

ensemble : ensemble de signaux à vider

valeur de retour : 0 ou -1 (erreur)

int **sigfillset (sigset_t * ensemble);**

ensemble : ensemble de signaux à remplir

valeur de retour : 0 ou -1 (erreur)

LES SIGNAUX

- Paramétrer la réception d'un signal : Posix

1- Mise en œuvre du masque de blocage des signaux

Préparation du masque

int **sigaddset** (sigset_t * ensemble, int numSignal);

ensemble : ensemble de signaux
numSignal : numéro du signal à ajouter
valeur de retour : 0 ou -1 (erreur)

int **sigdelset** (sigset_t * ensemble, int numSignal);

ensemble : ensemble de signaux
numSignal : numéro du signal à supprimer
valeur de retour : 0 ou -1 (erreur)

LES SIGNAUX

- Paramétrer la réception d'un signal : Posix

1- Mise en œuvre du masque de blocage des signaux

Application du masque

int **sigprocmask** (int methode, const sigset_t * ensemble,
sigset_t * ancien);

methode : SIG_SETMASK
ensemble : ensemble de signaux
ancien : sauvegarde de l'ancien masque
valeur de retour : 0 ou -1 (erreur)

LES SIGNAUX

- Paramétrer la réception d'un signal : Posix

2- Définition du comportement spécifique des signaux

```
int  sigaction ( int numSig, const struct sigaction * newSigAct,
                                 struct sigaction * oldSigAct );
```

numSig : numéro du signal

newSigAct : comportement du signal

oldSigAct : sauvegarde de l'ancien comportement

valeur de retour : 0 ou -1 (erreur)

LES SIGNAUX

- Paramétrer la réception d'un signal : Posix

2- Définition du comportement spécifique des signaux

```
struct sigaction {
    void (*sa_handler)( );  comportement (cf. signal())
    sigset_t sa_mask;  signaux à bloquer pendant sa_handler()
    int sa_flags;  combinaison OU de flags (cf. man)
    };
```

COMMUNICATION MULTI-TACHE

LES SIGNAUX

- Paramétrer la réception d'un signal : Posix

Exemple 11

COMMUNICATION MULTI-TACHE

LES SIGNAUX

- Mise en place d'une alarme

Exemple 13

LES IPC System V

IPC

- 3 outils de communication entre processus locaux:
  - mémoire partagée
  - ensembles de sémaphores
  - files de message
- Mécanismes de fonctionnement communs
  - identifiant unique obtenu via une clé (nommage)
  - non détruits automatiquement par le système
  - commandes de contrôle
  - fichier d'entête à inclure : sys/ipc.h

TIRAGE DE CLÉ

IPC

- A chaque combinaison (path, val) correspond une clé unique.
- Utilisation : obtenir un « nom » unique pour un IPC
- En cas de liens, tirage de la même clé !

```
key_t ftok ( const char * path, int val );
```

path : chemin d'accès d'un fichier existant
id : valeur combinée avec path pour obtenir la clé
valeur de retour : une clé ou -1 (erreur)

TIRAGE DE CLÉ

IPC

Exemple 15

COMMANDES DE CONTRÔLE DES IPC

IPC

- Liste des IPC présents:

  ipcs

Exemple:

```
IPC status from <running system> as of Wed Aug 25 09:05:23
T       ID      KEY       MODE       OWNER    GROUP
Message Queues:
Shared Memory:
m        0  0xffffffff --rw-rw-rw-   luter    profs
Semaphores:
s        0  0xffffffff --ra-r--r--    luter    profs
```

COMMANDES DE CONTRÔLE DES IPC

- Supprimer un IPC

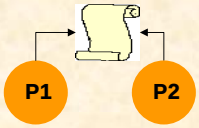
```
ipcrm  -option  numeroIPC
```

option : s -> ensemble de sémaphores
m ->mémoire partagée
q -> file de message

Exemple:

```
ipcrm -s 200
```

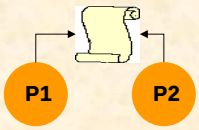

MÉMOIRE PARTAGÉE



- Critères de choix
  - Volume d'information **---> faible**
  - **Communication** ou synchronisation
  - Rapidité des échanges **---> rapide**
  - Protection des données **---> non synchronisé**
  - Conservation des données **---> non**
  - Structuration des données **---> oui**

COMMUNICATION MULTI-TACHE

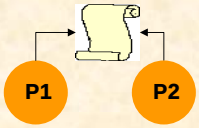
MÉMOIRE PARTAGÉE



- Allocation dynamique d'une zone de mémoire système
- Accessible par les processus :
  - disposant de la clé
  - autorisés (droits d'accès)
- Principe
  - Création ou ouverture de la zone de mémoire
  - Attachement à l'espace mémoire du processus
  - Manipulation par son adresse (variable)

COMMUNICATION MULTI-TACHE

MÉMOIRE PARTAGÉE

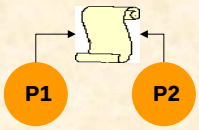


- Fonctions déclarées dans sys/shm.h
- Création ou ouverture d'une zone de mémoire partagée

```
int shmget ( key_t cle, size_t nbOctets, int flags );
```

cle : clé unique ou IPC_PRIVATE
nbOctets : taille de la zone en octets
flags : Combinaison OU de constantes:
IPC_CREAT : création ou ouverture
IPC_EXCL : création exclusive. Échoue si existe déjà
droits d'accès
valeur de retour : l'identifiant de la zone ou -1 (erreur)

MÉMOIRE PARTAGÉE



- Attachement d'une zone de mémoire partagée

```
void *  shmat ( int id, const void *adresse, int flags );
```

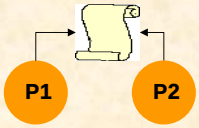
id : identifiant de la zone de mémoire
adresse : adresse d'attachement ou **NULL** (le système la choisit)
flags : Combinaison OU de constantes ou **0**: SHM_RND, SHM_RDONLY
valeur de retour : l'adresse de la zone ou -1 (erreur)

- Détachement d'une zone de mémoire partagée

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

adresse : adresse d'attachement
valeur de retour : 0 ou -1 (erreur)

MÉMOIRE PARTAGÉE

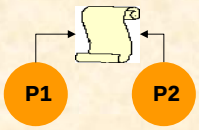


- Contrôle d'une zone de mémoire partagée

```
int  shmctl ( int id, int cmd, struct shmid_ds * infos);
```

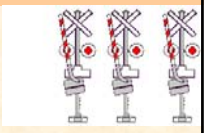
id : identifiant de la zone de mémoire
cmd : opération de contrôle (IPC_RMID: suppression, ...)
infos : informations. NULL si cmd = IPC_RMID
valeur de retour : 0 ou -1 (erreur)

MÉMOIRE PARTAGÉE



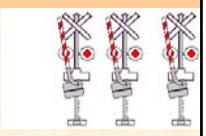
Exemple 16

ENSEMBLES DE SEMAPHORES



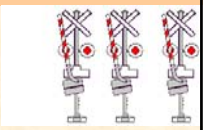
- Critères de choix
  - Volume d'information **---> pas de données**
  - Communication ou **synchronisation**
  - Rapidité des échanges **---> -**
  - Protection des données **---> -**
  - Conservation des données **---> -**
  - Structuration des données **---> -**

ENSEMBLES DE SEMAPHORES



- Gestion simultanée de *n* sémaphores : opérations atomiques
- Mutex ou compteur : valeur >= 0
- Enchaînement
  - Création ou ouverture de l'ensemble
  - Initialisation des sémaphores (si création)
  - Opérations atomiques sur *m* sémaphores parmi les *n* $P(S_i)$ ou $V(S_i)$

ENSEMBLES DE SEMAPHORES

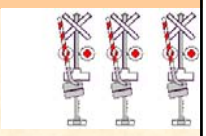


- Fonctions déclarées dans sys/sem.h
- Création ou ouverture d'un ensemble de sémaphores

```
int semget ( key_t cle, int nbSem, int flags );
```

cle : clé unique ou IPC_PRIVATE
nbSem : nombre de sémaphores de l'ensemble
flags : Combinaison OU de constantes:
IPC_CREAT : création ou ouverture
IPC_EXCL : création exclusive. Échoue si existe déjà
droits d'accès
valeur de retour : l'identifiant de l'ensemble ou -1 (erreur)

ENSEMBLES DE SEMAPHORES



- Opérations de contrôle sur un ensemble de sémaphores

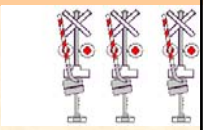
int **semctl (** int id, int semNum, int cmd, autre paramètre **);**

id : identifiant de l'ensemble

semNum : numéro du sémaphore concerné (de 0 à n - 1)
ou 0 si opération sur tous les sémaphores

cmd : opération
SETVAL: initialisation d'1 sémaphore
SETALL: initialisation de tous les sémaphores
GETVAL : lecture de la valeur d'un sémaphore
GETALL: lecture de la valeur de tous les sémaphores
IPC_RMID : suppression de l'ensemble
...

ENSEMBLES DE SEMAPHORES



- Opérations de contrôle sur un ensemble de sémaphores

int **semctl (** int id, int semNum, int cmd, autre paramètre **);**

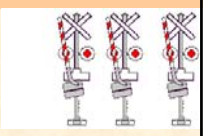
autre paramètre :

pour les commandes SETALL et GETALL :

```
union semun {
        int val;
        struct semid_ds *buf;
        ushort_t *array;      // valeurs des sémaphores
};
```

valeur de retour: pour GETVAL : la valeur du sémaphore
-1 : erreur

ENSEMBLES DE SEMAPHORES



- Opérations P et V sur les sémaphores d'un ensemble

int **semop (** int id, struct sembuf *sOps, size_t nbOps **);**

id : identifiant de l'ensemble

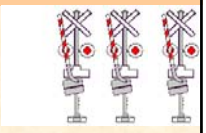
sOps : tableau de *nbOps* structures. Chaque structure décrit une des opérations.

nbOps : nombre d'opérations

valeur de retour : 0 ou -1 (erreur)

```
struct sembuf {
  short sem_num;   numéro du sémaphore concerné par l'opération
  short sem_op;    opération: +1 --> V(S),  -1 --> P(S)
  short sem_flg;   flags  };
```


ENSEMBLES DE SEMAPHORES



Exemples 18, 19, 20

COMMUNICATION MULTI-TACHE

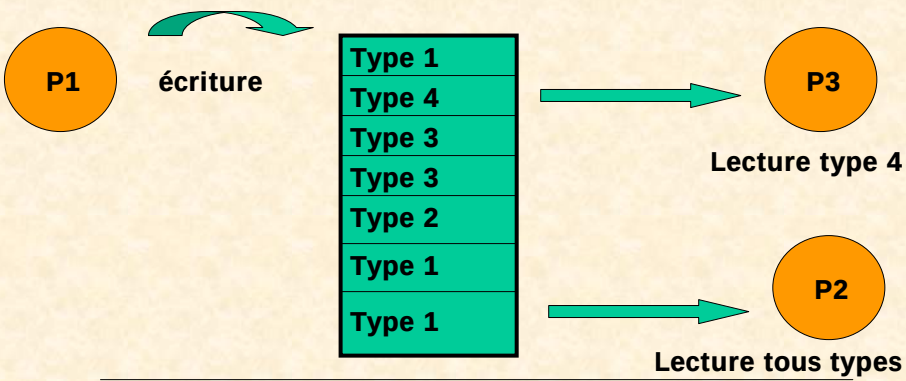
FILES DE MESSAGES

- Critères de choix
  - Volume d'information **---> moyen**
  - **Communication** ou synchronisation
  - Rapidité des échanges **---> oui**
  - Protection des données **---> oui**
  - Conservation des données **---> non**
  - Structuration des données **---> oui**

COMMUNICATION MULTI-TACHE

FILES DE MESSAGES

- Mécanisme de boîte aux lettres
- Message = type (numéro) + données



FILES DE MESSAGES

- Fonctions déclarées dans sys/msg.h
- Création ou ouverture d'une file de messages

int **msgget** (key_t cle, int flags);

cle : clé unique ou IPC_PRIVATE

flags : Combinaison OU de constantes:
IPC_CREAT : création ou ouverture
IPC_EXCL : création exclusive. Échoue si existe déjà
droits d'accès

valeur de retour : l'identifiant de l'ensemble ou -1 (erreur)

FILES DE MESSAGES

- Envoi d'un message

int **msgsnd** (int id, const void * message, int taille, int flags);

id : identifiant de la file de messages

message : adresse du message à envoyer, sous la forme

```
struct monMessage {
        long type;   -> le type du message
        ...          -> les données utilisateur
    };
```

taille : nombre d'octets des données utilisateur

flags : 0 ou IPC_NOWAIT (non bloquant)

valeur de retour : 0 (OK) ou -1 (erreur)

FILES DE MESSAGES

- Lecture d'un message

```
int msgrcv ( int id, const void * message, int taille,
                                        long type, int flags );
```

id : identifiant de la file de messages

message : adresse du message à envoyer (cf. msgsnd)

taille : nombre d'octets des données utilisateur

type : 0 : le premier message disponible
> 0 : premier message disponible du type donné
< 0 : premier message disponible du plus petit type <= au type donné

flags : MSG_EXCEPT (exclusion du type donné) ou IPC_NOWAIT (non bloquant) ou MSG_NOERROR (troncature en cas de message trop long)

valeur de retour : 0 (OK) ou -1 (erreur)

FILES DE MESSAGES

Exemple 17

EN PRATIQUE ...

Illustration du concept de processus Communication multitâche

processus

IPC

mémoire partagée

signaux

- **Suivi du déroulement d'une application non interactive**

LES THREADS POSIX

NOTION DE THREAD

= "fil" d'exécution

= "processus léger"

Fonction appartenant à un processus, qui peut s'exécuter en concurrence avec d'autres threads

=> unité d'ordonnancement

LES THREADS POSIX

NOTION DE THREAD

Les threads d'un même processus se partagent le même espace d'adressage

Les threads possèdent leurs propres :

- masque des signaux
- priorité
- pile
- registres

Intérêt : Gestion plus "légère", commutation de contexte plus rapide

LES THREADS POSIX

LA BIBLIOTHEQUE PTHREAD

- Implémentation des threads POSIX
- Fonctions **pthread_*XXX*( )** déclarées dans **pthread.h**
- Valeur retournée

  0 (OK) ou numéro de l'erreur
- Edition de liens avec gcc

  options **-lpthread -lposix4**
- **#define _REENTRANT** en 1ère ligne du code source

LES THREADS POSIX

CARACTERISTIQUES D'UN THREAD

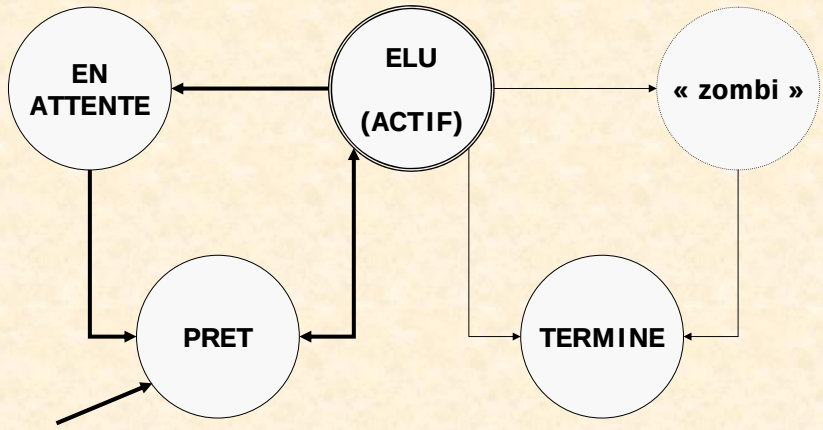
- Identifiant `pthread_t idThread;`
- Code `void * fonction(void * arg) { … }`
- Attributs
  - portée de l'ordonnancement ("scope")
    `PTHREAD_SCOPE_SYSTEM`
    `PTHREAD_SCOPE_PROCESS`
  - priorité
  - politique d'ordonnancement
    `SCHED_OTHER`
    `SCHED_FIFO`
    `SCHED_RR`

LES THREADS POSIX

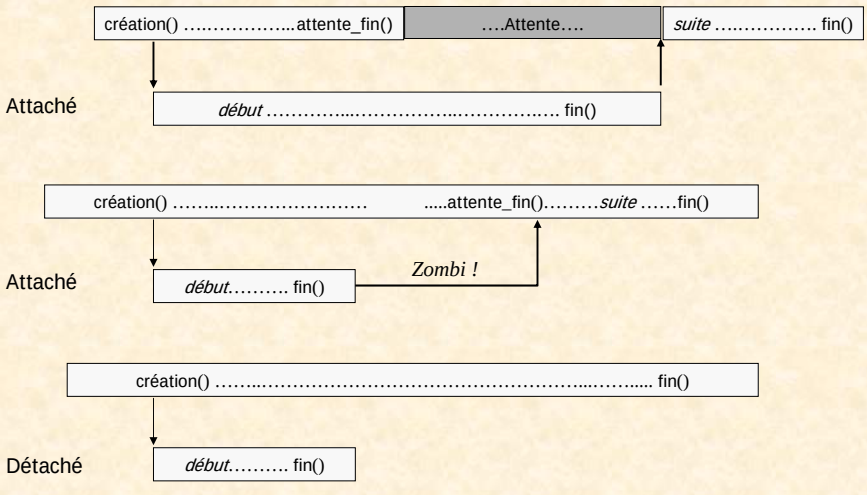
CARACTERISTIQUES D'UN THREAD

- Attributs
  - attachement
    `PTHREAD_CREATE_JOINABLE`
    `PTHREAD_CREATE_DETACHED`
  - pile : taille et adresse

CYCLE DE VIE D'UN THREAD



CYCLE DE VIE D'UN THREAD



LES THREADS POSIX

CREATION D'UN THREAD

```
int pthread_create ( pthread_t *tid, pthread_attr_t *attribut,
                     void * (* nomfonc) (void *arg), void *arg );
```

tid : l'identifiant du thread créé

attribut : les attributs du thread ou NULL (défaut)

nomfonc : le code du thread

arg : la valeur du paramètre de la fonction

valeur de retour : 0 (OK) ou numéro d'erreur

- Démarre l'exécution du thread

LES THREADS POSIX

TERMINAISON D'UN THREAD

```
void pthread_exit ( void *ret );
```

ret : valeur à renvoyer (si le thread est "attaché")

valeur de retour : aucune

- Termine l'exécution du thread appelant
- Si le thread a été créé avec l'attribut
  PTHREAD_CREATE_JOINABLE
  un autre thread peut attendre sa terminaison et récupérer la valeur renvoyée
- exit() termine **TOUS** les threads d'un processus

LES THREADS POSIX

ATTENTE DE LA FIN D'UN THREAD

int **pthread_join (pthread_t tid, void **ret);**

tid : l'identifiant du thread

ret : valeur renvoyée par le thread

valeur de retour : 0 (OK) ou numéro d'erreur

ANNULATION D'UN THREAD

int **pthread_cancel (pthread_t tid);**

tid : l'identifiant du thread

valeur de retour : 0 (OK) ou numéro d'erreur

IDENTIFIANT DU THREAD COURANT

pthread_t **pthread_self (void);**

valeur de retour : l'identifiant du thread courant

LES THREADS POSIX

INITIALISATION DES ATTRIBUTS

pthread_attr_init( )	attributs par défaut
pthread_attr_setdetachstate( )	détaché
pthread_attr_setguardsize( )	pile
pthread_attr_setinheritsched( )	héritage de l'ordonnancement
pthread_attr_setschedparam( )	priorité
pthread_attr_setschedpolicy( )	type d'ordonnancement
pthread_attr_setscope( )	attachement
pthread_attr_setstackaddr( )	pile
pthread_attr_setstacksize( )	pile

LES THREADS POSIX

CREATION D'UN THREAD

Exemple 21

LES THREADS POSIX

EXCLUSION MUTUELLE DES THREADS

- Accès concurrent à une variable globale

Exemple 22

EXCLUSION MUTUELLE DES THREADS: LES MUTEX

- Sémaphore d'exclusion mutuelle
- Utilisé pour résoudre des problèmes d'accès à une ressource critique
- Type de données **pthread_mutex_t mutex;**
- Attributs : il est possible de modifier les attributs d'un mutex, mais ceux-ci ne sont pas standard. On utilisera donc de préférence les attributs par défaut pour assurer la portabilité des applications.

EXCLUSION MUTUELLE DES THREADS: LES MUTEX

- Initialisation d'un mutex statique (attributs par défaut)

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
```

- Initialisation d'un mutex dynamique (attributs par défaut)

```
pthread_mutex_t *m;
m = malloc(sizeof(pthread_mutex_t));
pthread_mutex_init( &m, NULL );
```

- Destruction d'un mutex dynamique

```
pthread_mutex_destroy( &m );
```

EXCLUSION MUTUELLE DES THREADS: LES MUTEX

- P(mutex)

```
int pthread_mutex_lock ( pthread_mutex_ t *m );
```

m : le mutex
valeur de retour : 0 (OK) ou numéro d'erreur

- V(mutex)

```
int pthread_mutex_unlock ( pthread_mutex_ t *m );
```

m : le mutex
valeur de retour : 0 (OK) ou numéro d'erreur

EXCLUSION MUTUELLE DES THREADS: LES MUTEX

- Accès concurrent à une variable globale

Exemple 23

SYNCHRONISATION : SEMAPHORES POSIX

- Comptage de ressources
- Synchronisation de threads ou de processus
- norme Posix temps réel (Posix.1b)
- Type de données: **sem_t s;**
- **errno** positionnée en cas d'erreur
- **#include <semaphore.h>**

SYNCHRONISATION : SEMAPHORES POSIX

- Initialisation d'un sémaphore

int **sem_init (** sem_t *sem, int partage, unsigned int value **);**

sem : le sémaphore
partage : 0 (interne au processus)
value : valeur initiale
valeur de retour : -1 (erreur)

- Destruction d'un sémaphore

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

sem : le sémaphore
valeur de retour : -1 (erreur)

SYNCHRONISATION : SEMAPHORES POSIX

- P(sémaphore)

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

sem : le sémaphore
valeur de retour : -1 (erreur)

- P(sémaphore), non bloquant

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

sem : le sémaphore
valeur de retour : -1 (erreur)

- V(sémaphore)

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

sem : le sémaphore
valeur de retour : -1 (erreur)

SYNCHRONISATION : SEMAPHORES POSIX

- Cohorte

Exemple 24

EN PRATIQUE ...

Programmation Concurrente
Threads Posix

threads

sémaphores

- **Simulation d'un parking**

LES THREADS POSIX

SYNCHRONISATION : VARIABLES CONDITIONNELLES

- Synchronisation de threads
- Mécanismes :
  - Attente de la réalisation d'une condition ... en évitant l'attente active
  - Signalement de la réalisation d'une condition

thread 1	thread 2
while (! condition)	**réalisation condition**
attente ← - - - - - - - - - - **signal de réveil**	

SYNCHRONISATION : VARIABLES CONDITIONNELLES

- Synchronisation de threads
- Mécanismes :
  - Attente de la réalisation d'une condition ...
    en évitant l'attente active
  - Signalement de la réalisation d'une condition
- Évite certains interblocages
- Protection par mutex
- Matérialise une condition à exprimer

SYNCHRONISATION : VARIABLES CONDITIONNELLES

- Initialisation d'une variable conditionnelle

```
pthread_cond_t v = PTHREAD_COND_INITIALIZER;
```

- Attente sur une condition

```
int pthread_cond_wait( pthread_cond_t *v,
                       pthread_mutex_t *m );
```

v : la variable conditionnelle

m : le mutex associé

valeur de retour : 0 (OK) ou un code erreur

SYNCHRONISATION : VARIABLES CONDITIONNELLES

- Signalement d'une condition réalisée
  → débloque au moins un des threads en attente

```
int pthread_cond_signal ( pthread_cond_t *v );
```

v : la variable conditionnelle
valeur de retour : 0 (OK) ou un code erreur

- Signalement d'une condition réalisée
  → débloque tous les threads en attente

```
int pthread_cond_broadcast ( pthread_cond_t *v );
```

v : la variable conditionnelle
valeur de retour : 0 (OK) ou un code erreur

SYNCHRONISATION : VARIABLES CONDITIONNELLES

Thread attendant la condition	Thread signalant la condition
pthread_mutex_lock()	…
pthread_cond_wait() -> débloque le mutex	…
	…
Attente…	pthread_mutex_lock()
	pthread_cond_signal(): réveille le thread en attente
	pthread_mutex_unlock()
pthread_mutex_unlock()	…

SYNCHRONISATION : VARIABLES CONDITIONNELLES

Exemple 25

ATTENTION!!!

- **Aux entrées-sorties!!!**

  La table des descripteurs de fichiers est partagée
  En cas d'accès avec le même descripteur, utiliser **pread()** et **pwrite()**

Exemples 26 et 27

LES THREADS POSIX

ATTENTION!!!

- **Au fork() !!!**

  Seul le thread appelant est dupliqué

- **Aux fonctions !!!**

  Leur niveau de "résistance" au multithreading est indiqué dans le man

  MT-Safe : supporte le multithreading concurrent

  Async-Signal-Safe : peut être appelée d'un *handler* de signal

EN PRATIQUE ...

Programmation Concurrente
Threads Posix

threads

file d'attente

variables conditionnelles

- **Serveur d'affichage**

PROGRAMMATION SYSTEME UNIX : INDEX

PROGRAMMATION SYSTEME UNIX : INDEX

PROGRAMMATION SYSTEME UNIX : INDEX

➢ Threads Posix

➢ Sémaphores Posix