Système d'exploitation et programmation système

Cours 2

EISTI

2007/2008

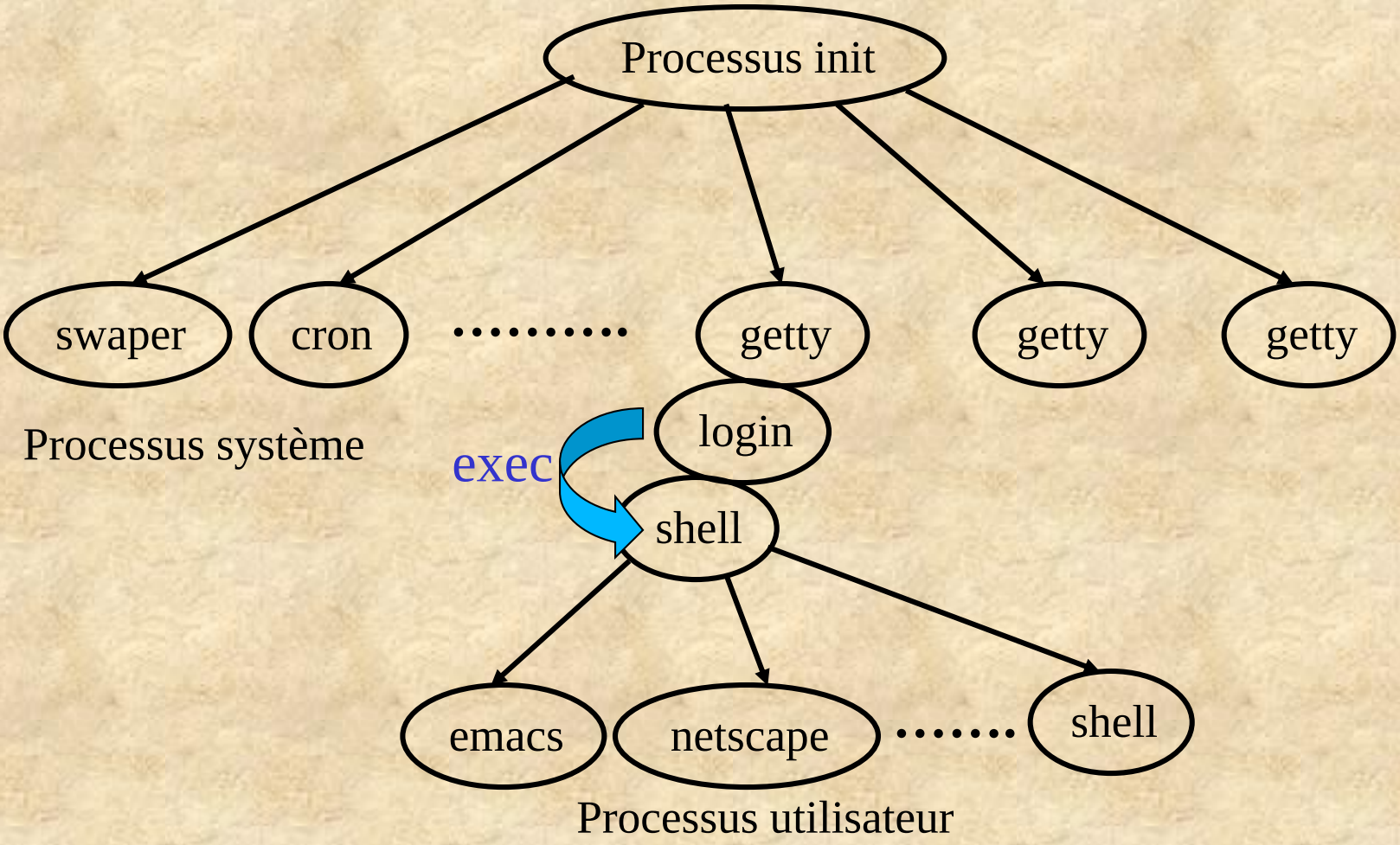
Département informatique

Processus

Processus

- Un programme produit par compilation et édition de lien est un objet **inerte** correspondant au contenu d'un fichier sur disque de nom par défaut a.out
- Un processus est un objet **dynamique** correspondant à l'exécution des instruction d'un programme
- Le processus est l'entité d'exécution dans le système linux
- Dans le système linux, il existe deux types de processus :
  - Processus système
    - swapper
    - crons
    - getty
  - Processus utilisateur qui correspondant à l'éxécution :
    - d'une commande
    - d'une application

Arborescence des processus dans linux



la commande shell :

pstree : affiche l'arborescence des processus

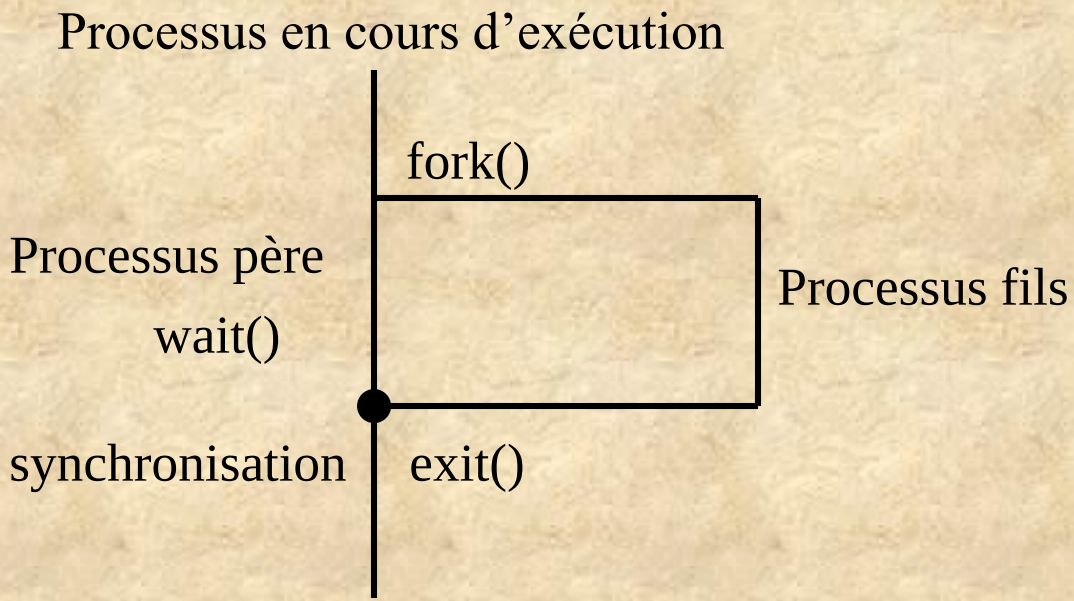
ps [option] : (Processus Status) donne la liste des processus actifs selon certains critères

Processus

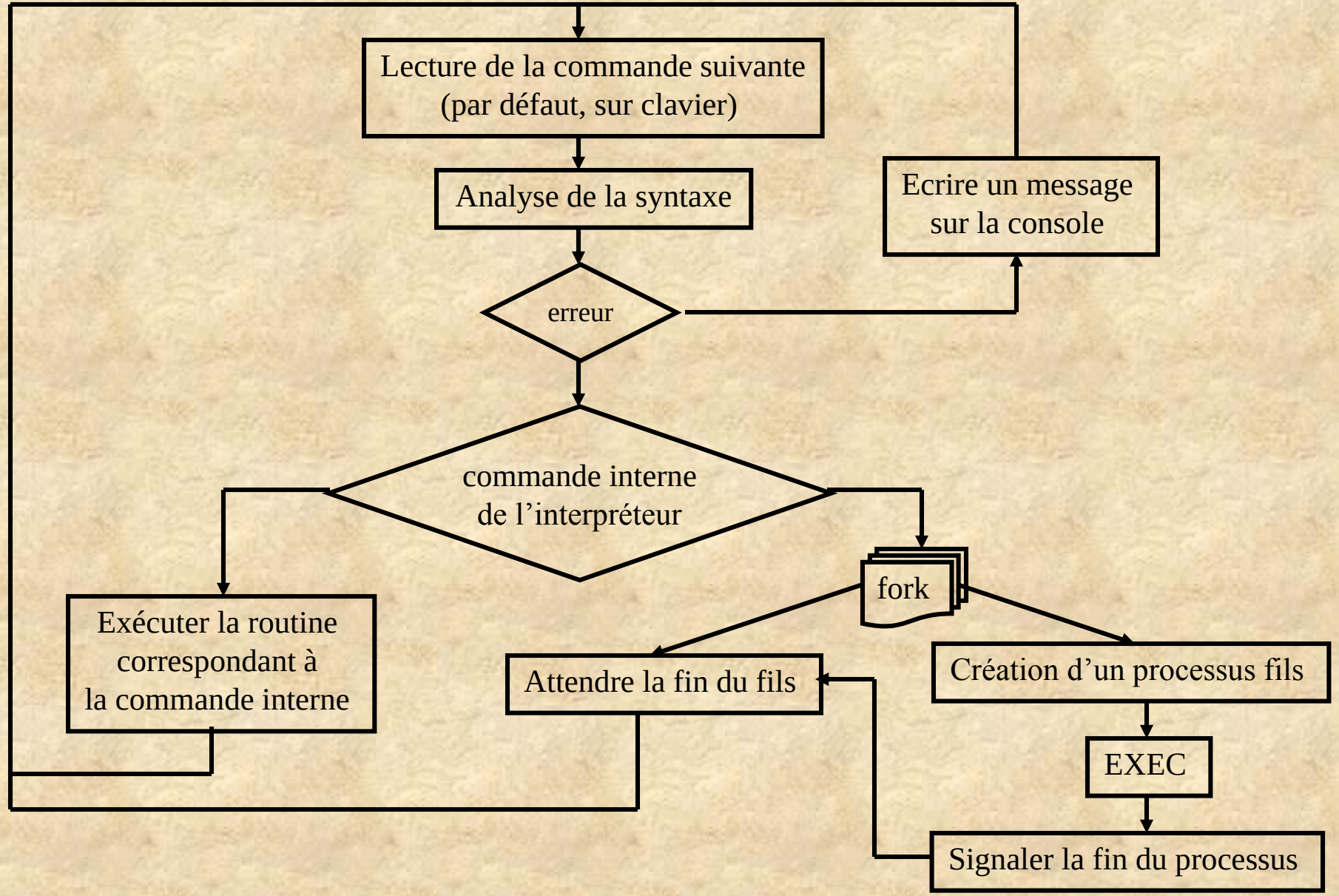
- Lors de l'initialisation du système linux, un premier processus, nommé *init* est créé avec un PID = 1
- Chaque processus est identifié par un numéro unique (PID)
- La commande ps donne la liste des procesus
- La création d'un processus se fait par duplication
- Un processus peut demander au système sa duplication en utilisant la primitive fork()
- On appelle le processus créateur le processus père. Le processus créé est appelé processus fils
- Le système crée une copie exacte du processus original avec un PID différent
- Le numéro du processus père est noté PPID
- Le processus fils peut executer un nouveau code à l'aide des primitives EXEC

Execution d'une commande par un shell

- Le shell se duplique (fork); on donc 2 processus shell identiques
- Le père se met en attente de la fin du fils (wait)
- Le shell fils remplace son exécutable par selui de la commande **compress**
- La commande **compress** s'exécute et compacte le fichier toto ; lorsqu'elle termine, le processus fils disparait
- Le père est alors réactivé, et affiche le prompt suivant

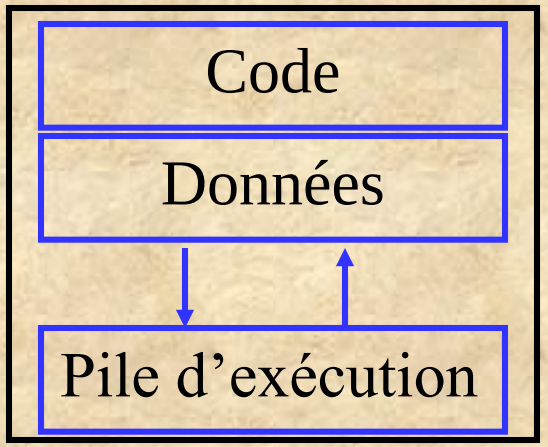


Exécution d'une commande par un shell



Espace d'adressage d'un processus

- Tout processus linux a un espace d'adressage constitué de 3 segments :



- Le code correspond aux instructions, en langage d'assemblage, du programme à exécuter.
- La zone de données contient les variables globales ou statiques du programme ainsi que les allocations dynamiques de mémoire.
- Enfin, les appels de fonctions, avec leurs paramètres et leurs variables locales, viennent s'empiler sur la pile.

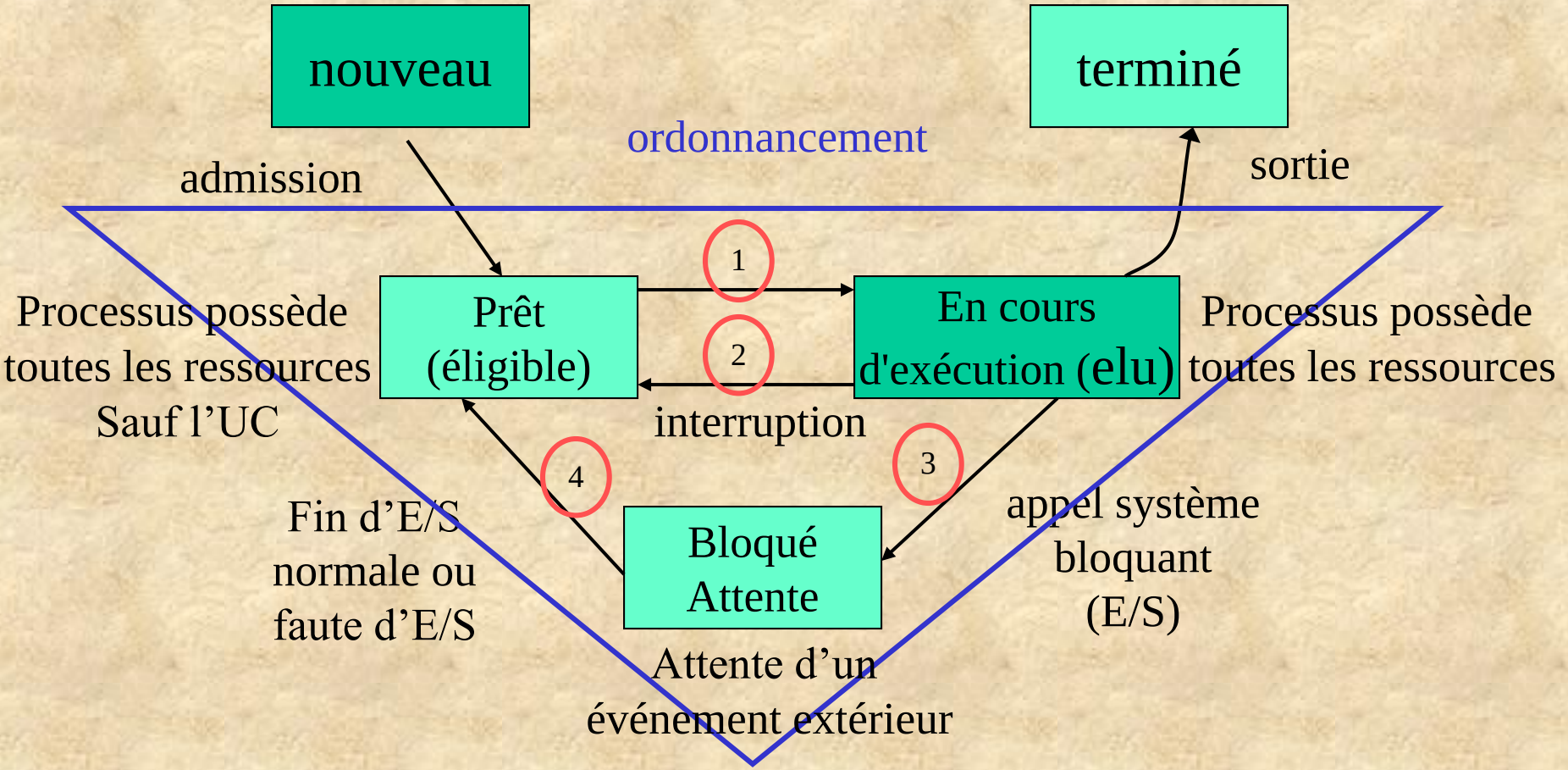
- Contexte d'un processus
  - Le contenu de son espace d'adressage
  - Le contenu des registres matériels
  - Le compteur ordinal
  - Les variables
  - Les structure de données du noyaux qui ont un rapport avec le processus

Vie d'un processus

- Appel système fork par le père. Le processus père continue son exécution en arrière plan.
  - Appel système exec par le processus fils.
  - Déroulement du programme.
  - Fin du programme, envoi du code retour.
- Récupération du processus fils à l'état zombie par le processus père.
- Si le processus père à fini son exécution avant le processus fils, le processus fils à l’état zombi est "rattaché" au processus originel. Son PPID est alors le PID de init.
- Tous processus linux qui se termine possede une valeur retour appelé code de retour (*exit status*) à laquelle le père peut accéder

Etats des processus

- Le processus peut être dans un certain nombre d'états connus : c'est un automate.
- L'ordonnanceur gère l'allocation du temps CPU aux processus.



Caractéristiques d'un processus

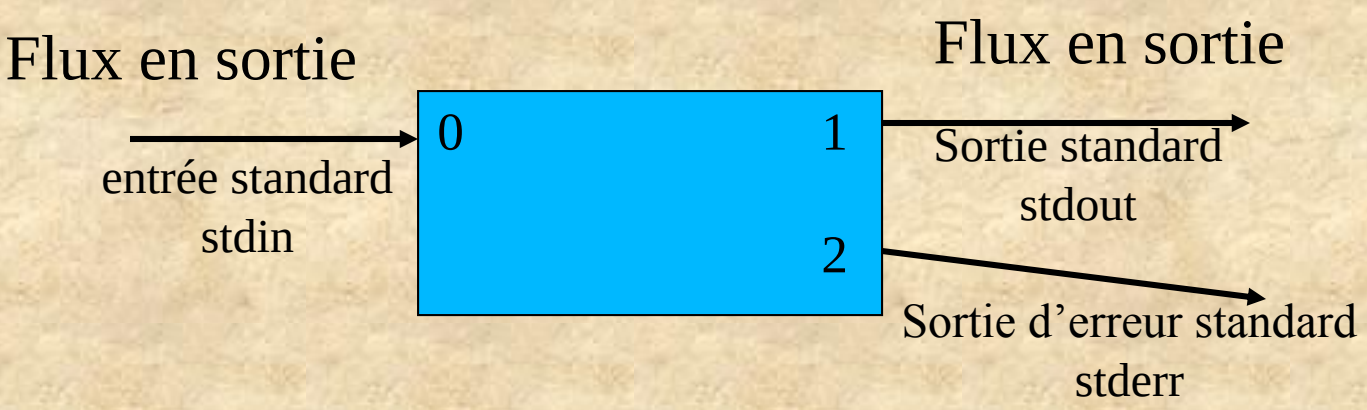
- PID
- PPID (PID du père)
- Etat (O/S/R/Z/T)
- Priorité
- Zone de code executable
- Zone de données manipulées
- Repertoire courant
- Table des descripteurs (fichiers ouverts)
- le masque de création de fichier(*umask)*
- la taille maximale des fichiers que

ce processus peut créer, appelée *ulimit*

- Un état des registres
- Répertoire courant et VarEnv
- Pile d'exécution
- Un propriétaire (UID) et son groupe (GID)
- Un terminal d’attachement
- Le temps UC consommé

Caractéristiques d'un processus

- Tout processus possède à sa création 3 descripteurs de fichiers:
  - 0 : stdin (entrée standard)
  - 1 : stdout (sortie standard)
  - 2 : stderr (sortie erreur)



- Par défaut ces trois descripteurs sont reliés au terminal correspondant à la session

Redirection

- Redirection de l'entrée standard

  - commande < nom_de_fichier

- Redirection de la sortie standard

  - commande > nom_de_fichier

  - commande >> nom_de_fichier (redirection sans écrasement)

- Redirection de la sortie en erreur standard

  - commande 2> nom_de_fichier (en Bourne Shell)

  - commande >& nom_de_fichier (en c-shell)

Programmation sous LINUX II

void **main** (int *argc*, char * *argv[]*, char ***arge*)

argc : nombre de paramètres passés plus le nom de l'executable

argv : tableau contenant les paramètres passés

arge : tableau contenant les variables d'environnements

Exemple : On lance un programme en C **prog** sous un shell en récupérant l'environnement de ce shell

$set HOME =/home/ing1/dupond

PATH=../usr/bin/

LOGNAME=dupond

$**prog** param1 param2

argc	3		
argv[0]	**prog**	arge[0]	"HOME =/home/ing1/dupond"
argv[1]	param1	arge[1]	"PATH=../usr/bin/"
argv[2]	param2	arge[2]	"LOGNAME=dupond"

Variables d'environnement

- On peut accéder aux variables d'environnement d'un processus par l'intermédiaire de deux fonctions

#include <stdlib.h>

char * **getenv**(const * *variables_environnement*)

récupère un pointeur sur la valeur de cette vatriable ou retourne NULL

int **putenv**(const char * *chaine*)

modifie ou ajoute une variable d'environnement : chaine : "variable=valeur"

retourne 0 dans le cas corrects

Manipulation des fichiers en C (open)

- Les fichiers seront toujours manipulés avec des primitives bas niveau (non bufferisées) :

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

int desc;
int open (const char * ref, int mode_ouverture, mode_t droits);
```

//demande une nouvelle entrée dans la table des des fichiers ouverts du système

desc = (ref, mode_ouverture, droits);

Paramètre mode_ouverture

Le paramètre mode_ouverture est construit par disjonction bit à bit des constantes définies dans fcntl.h

O_RDONLY	En lecture
O_WRONLY	En écriture
O_RDWR	En lecture est en écriture
O_NONBLOCK	Ouverture dans un mode non bloquant
O_APPEND	En ajout à la fin du fichier
O_CREAT	Crée fichier s'il n'existe pas
O_TRUNC	Remet le fichier à 0 s'il existe
O_EXCL	Echoue si le fichier existe

Manipulation des fichiers en C (close)

- Les fichiers seront toujours manipulés avec des primitives bas niveau (non bufferisées) :

#include <unistd.h>

Cette primitive permet de libérer un descripteur dans la tables des descripteurs

int **close** (int *desc*)

Ces primitives permettent la lecture et l'ecriture dans un fichier de descripteur *desc*

ssize_t **read** (int *desc*, void **ptr*, size_t *nb_octets*)

nb_octets : le nombre (int) d'octets que l'utilisateur voudrait lire

ssize_t : reçoit le nombre d'octets réellement lus

la fonction read détecte la fin de fichier et renvoie -1.

ssize_t **write** (int *desc*, void **ptr*, size_t *nb_octets*)

Modifier l'offset par rapport à une origine dans le fichier de descripteur *desc*

off_t **lseek** (int *desc*, off_t *offset*, int *origine*)

Fonction **stat**

La fonction **stat** permet de récupérer des informations sur un fichier dont on a l'accès en lecture des répertoires intervenant dans la référence de celui-ci. Il faut passer à cette fonction, par adresse, une structure dans laquelle on récupère les données sur le fichier de référence Ref.

#include <sys/stat.h>

int **stat**(const char **Ref*, struct stat **ptr_structure*)

int **fstat**(const int *desc*, struct stat **ptr_structure*)

où la structure stat est définie de la façon suivante dans <sys/stat.h>

Structure **stat**

struct stat

{

dev_t st_dev; identification du disque logique

ino_t st_ino; numéro d'inode du fichier

mode_t st_mode; type et droits d'accès

nlink_t st_nlink; nombre de liens physiques

uid_t st_uid; propriétaire

gid_t st_gid; groupe du propriétaire

off_t st_size; taille

time_t st_atime; date de dernier accès

time_t st_mtime; date de dernière modification

time_t st_ctime; date de dernière modification de l'inode

}

Le champ st_mode est exprimé sur 12 bits (dont 4 pour le type du fichier). Une valeur de ce champ est une combinaison logique par l'opérateur de disjonction | des constantes suivantes:

Structure **stat**

Nom symbolique du bit	**Interprétation du bit**
S_ISUID, S_ISGID	le set_uid et le set_gid un autre bit pour le sticky bit
S_IRUSR	lecture par le propriétaire
S_IWUSR	écriture par le propriétaire
S_IXUSR	exécution par le propriétaire
S_IRWXU	lecture, écriture et exécution par le propriétaire
S_IRGRP	lecture par les membres du groupe propriétaire
S_IWGRP	écriture par les membres du groupe propriétaire
S_IXGRP	exécution par les membres du groupe propriétaire
S_IRWXG	lecture, écriture et exécution par le groupe
S_IROTH	lecture par les autres
S_IWOTH	écriture par les autres
S_IXOTH	exécution par les autres
S_IRWXO	lecture, écriture et exécution par les autres

Exemple d’utilisation de **stat**

struct stat *variable*;

int *retour*;

retour = stat("myfic",&*variable*);

if (*retour* != -1) printf("L'utilisateur est : %d ",*variable*.st_uid);

else perror("stat:");

Multitâches

Création et contrôle de plusieurs processus

MULTIPROCESSING

Définition

- On appelle multiprocessing le fait de pouvoir faire traiter une tâche complexe par une seule machine en utilisant plusieurs processeurs en même temps.
- On va donc s'intéresser à des processus gérés par le même système d'exploitation, s'exécutant en parallèle, s'échangeant et partageant des données et se synchronisant.

Avantages / Inconvénients

- Gain de temps de traitement
- Meilleure utilisation d'une machine multiprocesseurs
- Problèmes d'accès concurrents (synchronisation, inter-blocages, verrous…)

Primitive fork

- L'appel de la primitive fork demande au système d'effectuer une copie exacte du processus en cours d'exécution.
- Si cette primitive a réussi, un nouveau processus est créé et exécute le même programme.
- Il hérite de la pile d'exécution et donc ce nouveau processus "débute" par le retour de la primitive fork
- Dans le processus père, fork() retourne le PID du processus fils crée
- Dans le processus fils, fork() retourne la valeur 0
- Si echec de fork(), alors pas de fils créé et fork retourne -1.

Création d'un nouveau processus

- Appel de la primitive fork :

```
#include <unistd.h>
  void main()
  {
    pid_t m_pid;
    m_pid = fork();
    printf("Retour de la fonction fork : %d\n",m_pid);
  }
```

Caractéristiques héritées

- UID, identifiant du ou des propriétaires
- GID, identifiant du groupe
- Toutes les valeurs des variables…

#include <unistd.h>

getpid(); retourne le PID du processus appelant

getppid(); retourne le PID du père de processus

char * **getcwd**(char * *buf*, size_t *taille*);

La référence absolue du repertoire de travail d'un processus peut être obtenue dans la chaine *buf* de taille *taille*

Caractéristiques non héritées

- PID
- PPID
- Temps d'exécutions (initialisés à zéro)
- Les signaux pendants
- La priorité d'exécution si elle a été modifiée
- Les verrous sur fichiers

Cas spécifique des descripteurs de fichiers

- Après un fork, les entrées de la table des descripteurs pointent sur la même entrée de la table des fichiers ouverts.
- Le nombre de descripteurs de la table des fichiers ouverts est incrémenté.
- L'offset est commun aux deux processus.

Fin de vie d'un processus et synchronisation

- Rappel : tout processus passe en état zombie lorsque son exécution est terminée.
- Un processus zombie occupe une entrée dans la table des processus (le nombre est limité !)
- Le processus père peut alors accéder aux informations relatives à cette terminaison

Primitives lancées à partir de C

- Il est possible depuis un programme C de lancer une commande qui sera interprétée par le shell

  #include <stdlib.h>

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

- La primitive exit() est une autre fonction qui ne retourne aucune valeur, puisqu'elle termine le processus qui l'appelle

  #include <stdlib.h>

  void **exit** (int *status*)

  L'argument *status* est un entier qui indique au shell (ou au père de façon générale) qu'une erreur s'est produite. On laisse à zéro pour indiquer une fin normale

Primitives sleep

- La primitive sleep() est similaire à la commande shell sleep. Le processus est bloqué durant le nombre de secondes spécifié, sauf s'il reçoit entre temps un signal.

  #include <unistd.h>

  int **sleep** (int *secondes*)

Primitives de recouvrement exec

- La primitive execlp() permet le recouvrement d'un processus par un autre exécutable

void **execlp**(char * *cmd*, char **arg*, ….,NULL)

cmd : est une chaîne de caractère qui indique la commande à exécuter

arg : spécifie les arguments de cette commande (argv) + NULL

 car on ne connaît pas la taille de la liste a priori

Exemple : execution de **ls –l /usr**

execlp("ls", "ls", "-l", "/usr", NULL)

- La primitive execlp() retourne –1 en cas d'erreur, sinon rien si tout se passe bien

Recouvrement avec la famille execv

Les primitives du type execv permettent de passer les arguments au programme sous la forme d'un tableau.

#include <stdio.h>

int **execv**(const char **nom*, const char **argv[ ])

int **execvp**(const char **nom*, const char **argv[ ])

Les paramètres passés sont :

- le nom du fichier exécutable (avec son chemin d'accès pour execv)

- le tableau des arguments

Exemple d'utilisation :

```
char *arg[ ];
arg[0] = "myprog";
arg[1] = "toto";
arg[2] = "tata";
arg[3] = "titi";
execv("/home/ing1/dupond/myprog",arg);
execvp("myprog",arg);
```

La fonction execvp cherche de même automatiquement le chemin d'accès au programme dans le PATH.

Primitives wait, waitpid

La primitive wait permet de récupérer les informations de terminaison et de supprimer les processus zombie.

– Si l'appelant possède au moins un fils non zombie l'appel est bloquant

– Si l'appelant ne possède aucun fils (ni en exécution ni zombie) le retour est immédiat et vaut –1

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int * status);
```

Le retour est le PID du processus fils qui a été récupéré.

Primitives waitpid

- La primitive waitpid permet sélectionner un processus fils de pid particulier

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

- Le paramètre ***pid*** permet de sélectionner le processus attendu de la manière suivante :
  - <-1 tout processus fils dans le groupe |pid|
  - -1 tout processus fils
  - 0 tout processus fils du même groupe que l'appelant
  - > 0 processus fils d'identité pid
- Le paramètre ***options*** permet entre autre de choisir entre les modes :
  - 0 (WUNTRACED) bloquant : recevoir l'information concernant également les fils bloqués
  - 1 (WNOHANG) non-bloquant : ne pas bloquer si aucun fils ne s'est terminé
- La fonction renvoie :
  - -1 en cas d'erreur (en mode bloquant ou non bloquant) ;
  - 0 en cas d'échec (processus demandé existant mais ni terminé ni stoppé) en mode non bloquant ;
  - le pid du processus fils zombi pris en compte sinon.

Interprétation à l'aide de macros

- Si *status* est non NULL, **wait** et **waitpid** y stockent l'information sur la terminaison du fils. Cette information peut être analysée avec les macros suivantes :

non null si le fils s'est terminé

- `#define WIFEXITED(stat) ((int)((stat)&0xFF) == 0)`

Donne le code retour tel qu'il a été mentionné dans l'appel exit() ou dans le return de la fonction main(). Cette macro ne peut être évaluée que si WIFEXITED est non null

- `#define WEXITSTATUS(stat) ((int)(((stat)>>8)&0xFF))`

indique que le fils s'est terminé à cause d'un signal non intercepté

- `#define WIFSIGNALED(stat) ((int)((stat)&0xFF) > 0 && (int)((stat)&0xFF00) == 0)`

Interprétation à l'aide de macros

donne le nombre de signaux qui ont causé la fin du fils. Cette macro ne peut être évaluée que si WIFSIGNALED est non null

- **`#define WTERMSIG(stat) ((int)((stat)&0x7F))`**

indique que le fils est actuellement arrêté; Cette macro n'a de sens que si l'on a effectué l'appel avec l'option WUNTRACED

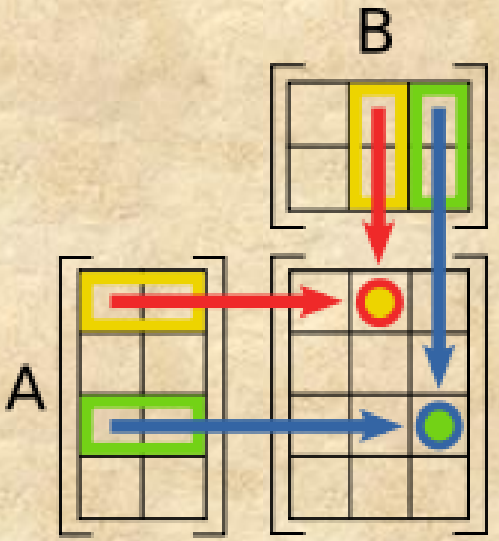
- **`#define WIFSTOPPED(stat) ((int)((stat)&0xFF) == 0177 && int)((stat)&0xFF00) != 0)`**

donne le nombre de signaux qui ont causé l'arrêt du fils. Cette macro ne peut être évaluée que si WIFSTOPPED est non null

- **`#define WSTOPSIG(stat) ((int)(((stat)>>8)&0xFF))`**

Exemple d’utilisation de la Primitive waitpid

Exemple calcul du produit matriciel :



WEXITSTATUS(*status*) donne le code de retour tel qu'il a été mentionné dans l'appel **exit()** ou dans le **return** de la routine **main**

$$\forall i,j : c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj}$$

Synchronisation

- L’effet de sleep est très différent de celui de wait: wait bloque jusqu’à ce qu’une condition précise soit verifiée

  (exemple la mort d’un fils), alors que sleep attend pendant une durée fixe

⇨ **sleep ne doit jamais être utilisé pour tenter de synchroniser deux processus**

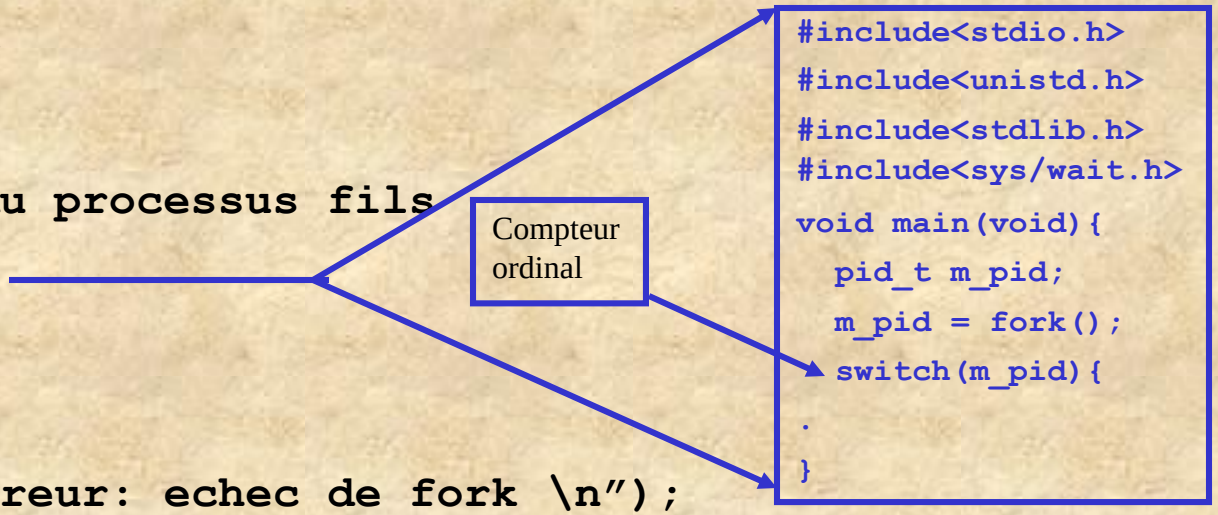
Multiprocessing

processus père

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

void main(void){
  pid_t m_pid;//PID du processus fils
  m_pid = fork();
  switch(m_pid){
  case -1 :
          printf("Erreur: echec de fork \n");
  case 0 :
          printf(" Processus fils : pid = %d\n", getpid())
          exit(0) fin du processus fils
          break;
  default :
        printf("Ici le pere, le fils a un PID : %d\n", m_pid);
        wait(0);// attente la fin du fils
  }
}
```

Création d'un processus fils



FIN