Chapitre 4: Gestion des processus

Plan de travail

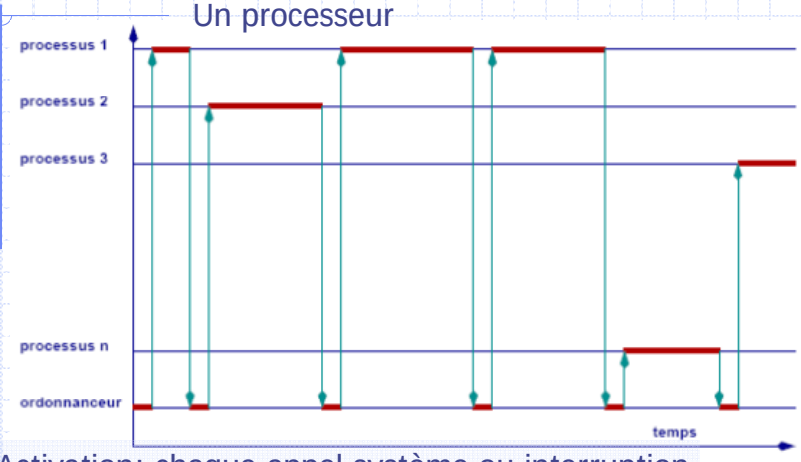
- Qu'est-ce qu'un processus ?
  concept, identification
- Création de processus
- Description d'un processus
- Environnement d'un processus
- Terminaison d'un processus
- Information sur un processus (état…)
- Chargement d'un processus

Qu'est-ce qu'un processus: Définitions

- Instruction = indécomposable et indivisible
- Processeur = ...
- Processus = suite temporelle d'exécutions d'instructions
- Processus = exécution d'un programme (instructions + données)
- Processus = entité dynamique (création --> disparition)

Les processus (système temps partagé)

Un processeur

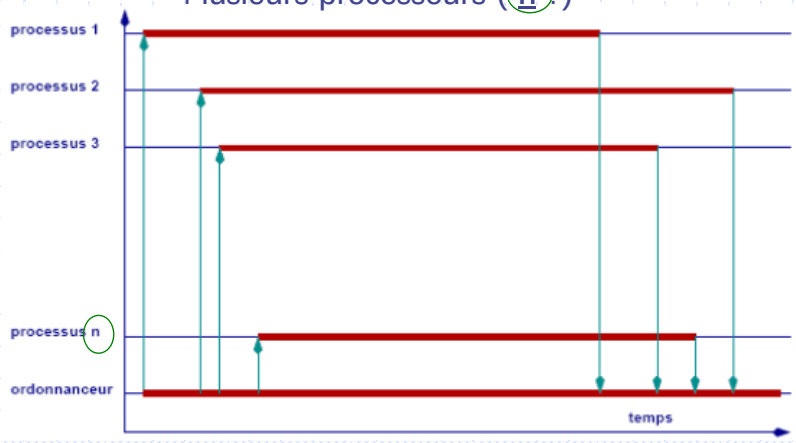


Activation: chaque appel système ou interruption

- Ordonnanceur : un processus ⇒ un coût
- Nécessité d'avoir des opérations atomique (appels systèmes)

Les processus (système temps partagé)

Plusieurs processeurs (n ?)



- Ordonnanceur : un processus ⇒ un coût
- Nécessité d'avoir des opérations atomique (appels systèmes)

Notion de processus UNIX (1)

- Approche intuitive
  - UNIX = système "temps partagé"
  - Plusieurs processus en même temps
  - Un seul processus actif
  - Attente des autres (dans/hors mémoire)
- Contenu d'un processus
  - Code
  - Données
  - Valeur des registres généraux
  - État des fichiers ouverts
  - Répertoire courant
  - ...

Le nécessaire pour l'exécution et la reprise d'exécution

Notion de processus UNIX (2)

- Processus (quelques centaines / machine)
  - Associé à un utilisateur
  - Sur une seule machine ou distribué sur le réseau
  - Compétition entre processus
  - Variété de mécanismes de communication / synchron.
  - Ordonnanceur: mode privilégié, dans le noyau
- Sous-processus (Threads – quelques milliers / machine)
  - Partage données, code, M.V. mais ¬ pile exécution
  - Même utilisateur, même machine
  - Très peu d'infos, peu de compétition,
  - Ordonnanceur: simple, mode utilisateur

Identification d'un processus

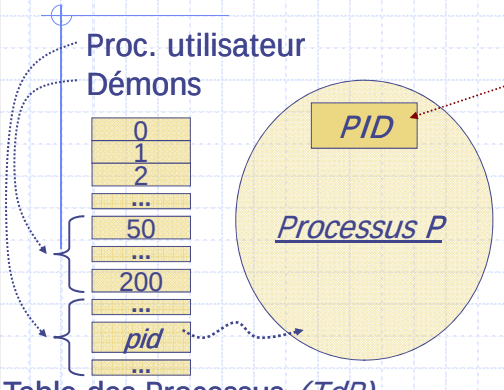


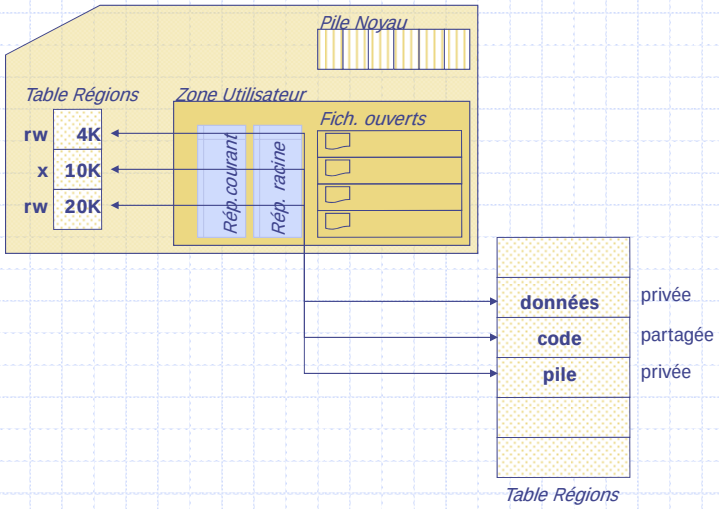
Table des Processus *(TdP)*

Process Identifier :
- unique
- généré

init :
- exécute /etc/rc. (initial.)
- gère mono ou multi
- processus privilégié
- lance processus connexion

- 0: scheduler, swapper (intégré au noyau)
- 1: /sbin/init (père de tous les processus)
- 2: gestionnaire de pagination (intégré au noyau)

Contexte d'un processus: Notion de région (1)



Contexte d'un processus: Notion de région (2)

- Structure
  - Pointeur: inode du fichier associé
  - Type: code, pile, données privées, partagées
  - Taille
  - Localisation (mémoire physique)
  - Etat: Fonction (verrouillée, demandée, en chargement, valide)
  - Compteur: nombre de processus qui la référence
- Opérations
  - Allocation/libération
  - (Dé)Verrouillage
  - Attacher/détacher une région
  - ... ➔ Exemple: voir appels systèmes *exec*, *sbrk*

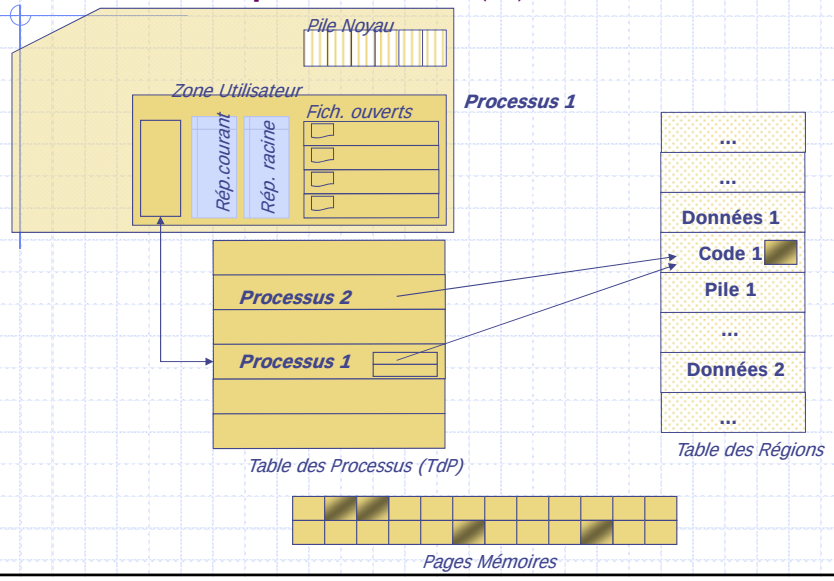
Contexte d'un processus : Autres éléments (1)

- Contenu (régions) :
  - Code du programme
  - Données
  - Pile utilisateur du processus
  - Mémoire partagée
- Quelques éléments :
  - Compteur ordinal: adresse virtuelle
  - Registre d'état processeur
  - Elément de la table des processus
    - ➔ Etat + contrôle général du processus
  - Correspondance des adresses
    - ➔ virtuelle / physique

Contexte d'un processus : Autres éléments (2)

- Quelques éléments (*suite*)
  - Répertoires: courant, racine
  - Table des descripteurs
  - Pointeur (table des processus)
  - Identificateurs de l'utilisateur
  - Compteurs de temps
  - Tableau de réaction aux signaux
    - →*Voir cours gestion des signaux*
  - Terminal de connexion éventuel (voir *ps*)
  - Champs: erreur, valeur de retour appels systèmes
  - Champs limites: taille processus, ...
  - Modes de permission (voir *umask*)
  - Paramètres E/S: volume, adresse, ...

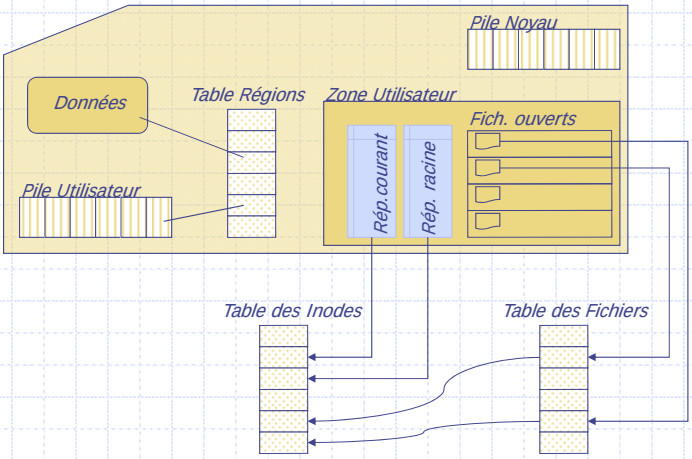
Intégration dans le système: Table des processus (1)



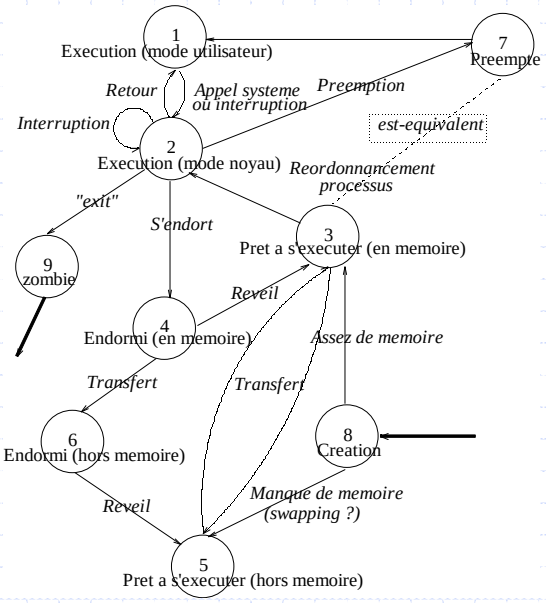
Intégration dans le système: Table des processus (2)

- Contenu d'un élément
  - Etat d'un processus
  - Localisation de la zone U
    → Mémoire principale/secondaire
  - Identificateurs d'utilisateur
    → *Voir gestion des signaux*
  - Identificateurs de processus
    → Lien de parenté
  - Evènement ayant provoqué le sommeil (*sleep*)
  - Paramètres d'ordonnancement
    → Ordre d'exécution
  - Signaux envoyés au processus (non traités)
  - Paramètres de calcul de priorité
    → Temps d'exécution, ...

Intégration dans le système: Entrée-sortie



Etats d'un processus



Changements d'états d'un processus

- Processus en attente:
  - Etats: endormi-en-mémoire, endormi-transféré
  - Réveil (changement d'état): par interruption
- Préemption:
  - Mode utilisateur: oui
  - Mode noyau: Oui si se termine
- Exécution d'une interruption:
  passage en mode noyau
- Prêt à s'exécuter en mémoire
  - Ordonnanceur / horloge
  - Gestion des priorités

Ordonnancement des processus: principes de base (1)

Politique générale

- Sélection de l'élu:
  {préempté, en mémoire, prêt s'exécuter}
- Plus haute priorité ou Max_{temps} (prêt s'exécuter)
- Changement de processus:
  dépassement du quantum de temps
- Re-évaluation de la priorité:
  - mode noyau → mode utilisateur
  - mode noyau → endormi-en-mémoire
  - réajustement périodique: {prêt s'exécuter}

→ Gestion des changements d'états

Ordonnancement des processus: principes de base (2)

- Calcul de la priorité
  - Calcul du temps utilisé en exécution
  - Recalcul de la priorité:
    - ➔ Changement de files de priorité
  - Paramètres:
    - Priorité dynamique :
      - Le moins récemment “en exécution”
      - Raison de l'attente
    - Priorité statique (*nice*)
- En pratique
  - Programme interactif: plus prioritaire
  - U.C. récemment utilisé: moins prioritaire

Utilisateurs et groupes effectifs/réels

- Le bit SETUID
  - Structure processus: utilisateur effectif/réel
  - l'utilisateur associé au login (réel)
  - avec les droits d'un utilisateur donné (effectif)
  - *chmod* : *chmod u+s* fichier (fichier exécutable)
  - exemple: commande *passwd*

→ idem pour le groupe

- Utilisation des groupes secondaires
  - groupe primaire/standard vs groupes secondaires
  - *newgrp* : prend le groupe passé en paramètre
  - Vérification des droits (groupe secondaire ?)
  - Fichier */etc/group*

Norme POSIX

Routine	*Description*
pid = fork ()	Création fils
e = waitpid (pid,&etat,opts)	Attendre fils
e = execve (nom,argv,envp)	Remp. image
exit (etat)	Term. process.
e = sigaction (sig,&act,&oact)	Spécif. action
e = kill (pid, sig)	Envoi signal
residuel = alarm (seconde)	Prog. SIG...
pause()	Susp. process.

Processus et Windows (sous-système POSIX)

Norme POSIX 1 :
- Contrôle de processus
- Communication inter-processus
- Entrées/sorties « caractères »
- ...

Ne contient pas :
- création de processus léger,
- création de fenêtre
- « RPC » et « sockets »
- ...

Services Windows pour UNIX
- 2000 fonctions
- 300 utilitaires

Pas d'accès aux fonctions de Windows

Création d'un processus (1)

Syntaxe :

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork ( void );
```

Retour :

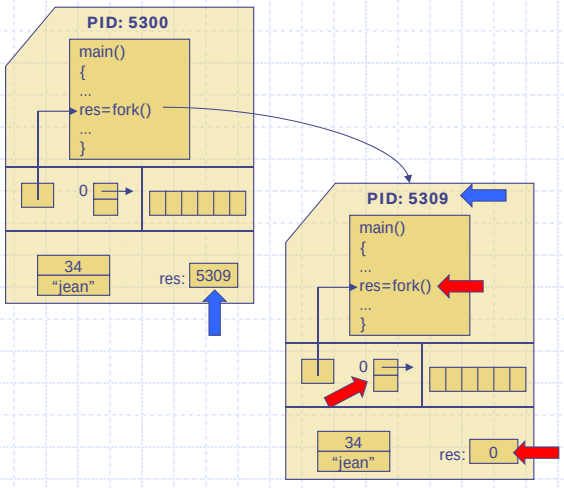
- 0 dans le fils, *PID* dans le père, si OK,
- -1, si *erreur*

Partage :
- code
- mém. *read_only*

Duplication :
- information
- données

- *Nouveau pid, nouvelle entrée dans la TdP*
- « héritage » d'informations
- Qui s'exécute? Quand? À partir d'où?

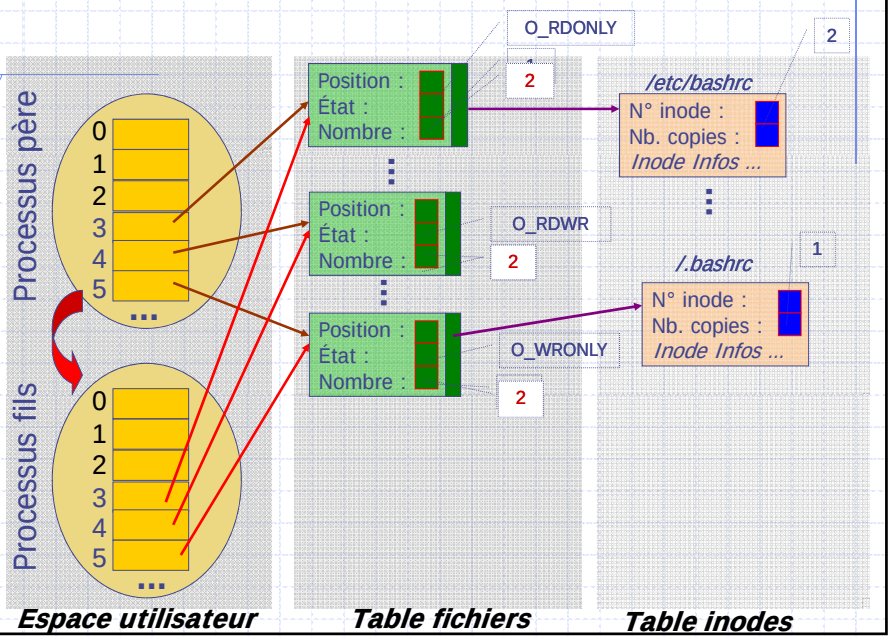
Création de processus (2)



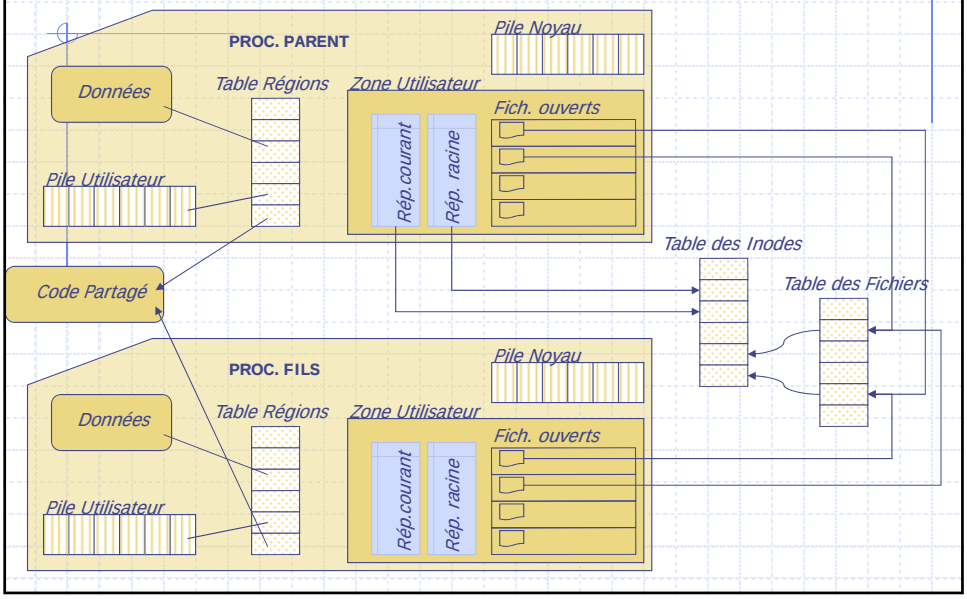
Création de processus (3)

- Appel système **fork**
  - Deux processus identiques
  - Deux processus indépendants
  - Exécution des processus: voir ordonnanceur
  - Duplication *vs* partage
- Informations dupliquées
  - Table des descripteurs
  - Buffers
  - Curseur d'exécution
  - Variables globales
  - ...

État des tables après un fork



fork et les structures internes



fork: exemple simple

```
#include <sys/types.h>
int main () {
        int pid;
        ...
        if ( (pid = fork ()) < 0) erreur;
        if (pid == 0)
                /* Traitement fils */
        else
                /* Traitement père */
}
```

fork: 2ème exemple (1)

```
#include "stevens.h"
int glob = 6; /* variable externe dans zone de data */
char mess[ ] = "un message sur stdout\n";
int main(void) {
  int var; /* variable automatique sur la pile */
  pid_t pid;
  int l=strlen(mess);
  var = 88;
  if ( write(1, mess, l) != l)ERRNO_EXIT;
  printf("Avant le fork\n");
      /* On ne vide pas les tampons de stdout */
```


fork: 2ème exemple (2)

```
  if ( (pid = fork()) < 0) ERRNO_EXIT;
  else if (pid == 0) { /* === Processus FILS == */
    glob++ ; var++ ;
  } else { /* === Processus PÈRE ============ */
    sleep(2);
  }
/* ===== Processus PÈRE et FILS ============= */
  printf("pid = %d, glob = %d, var = %d\n",
        getpid(), glob, var);
  exit(0);
}/* main */
```

fork: 2ème exemple (exécution)

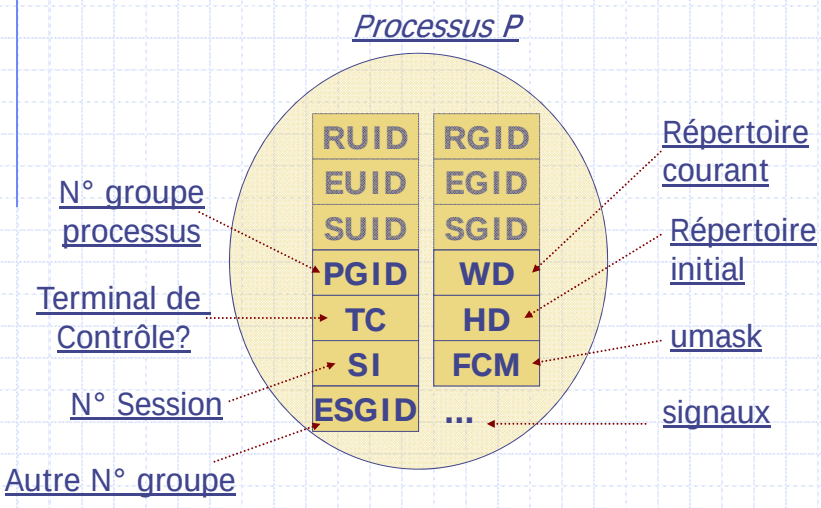
```
$ fork1
un message sur stdout
Avant le fork
pid = 20616, glob = 7, var = 89
pid = 20615, glob = 6, var = 88
```

```
$ fork1 > res
$ cat res
un message sur stdout
Avant le fork
pid = 20619, glob = 7, var = 89
Avant le fork
pid = 20618, glob = 6, var = 88
$
```

stdout associé :
- terminal
- fic. disque

Moment de la Synchronisation?

Information d'un processus (1)



Information d'un processus (2)

Hérité
- TDF
  - « close-on-exec »
- espace mémoire
  - Pile exécution
  - Code
  - Données statiques
  - Tas
  - Variables environnement
- mémoires partagées
- limites de ressource
- sauvegarde des registres

Non hérité
- PID
- PPID
- temps calcul
- signaux et alarmes
- verrous
- code retour fork

Efficacité?
- vfork (partage)
- *copy on write*

Obtenir les Informations du processus

Syntaxe :

```
#include <sys/types.h> #include <unistd.h>
pid_t getpid ( void );
pid_t getppid ( void );
uid_t getuid ( void );
uid_t geteuid ( void );
gid_t getgid ( void );
gid_t getegid ( void );
```

- PID du processus
- PID du père
- Propriétaire du processus
- Groupe du processus

} *Réel ou effectif*

Modifier les Informations du processus

Syntaxe :

```
#include <sys/types.h> #include <unistd.h>
int setuid ( uid_t uid );
int setgid ( gid_t gid );
int setreuid ( uid_t ruid, uid_t euid );
int setregid ( gid_t rgid, gid_t egid );
int seteuid (uid_t euid);
int setegid (gid_t egid );
```

Retour :

- 0, si succès,
- -1, si *erreur*

User pour R,E,S

Group pour R,E,S

User:R,E

Group:R,E

User:E

Group:E

- Rappel: Forte dépendance OS
- PID et PPID: non modifiable

Terminaison d'un processus

Normale *vs* anormale

P

```
main {
  ...
  exit(int)
}
```

Ou return implicite?

P

```
main {
  ...
  abort()
}
```

Si non ignorée

SIGABRT

signaux SIGKILL...

P

```
main {
  ...
}
```

exit(3) = _exit(2) + vidage buffer E/S

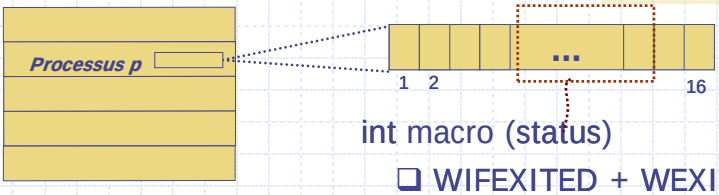
_exit(2) = ferm. desc. fic. + code retour + SIGCHLD

abort(3) = ferm. desc. fic. + « dump »

Etat d'un processus

0 = vrai : OK
>0 = faux = KO

Table des Processus



int macro (status)

- ❑ WIFEXITED + WEXITSTATUS
- ❑ WIFSIGNALED + WTERMSIG
- ❑ WIFSTOPPED + WSTOSIG

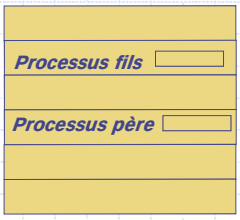
Et la commande *ps* ?

- ❑ un code de retour (exit) - terminaison normale
- ❑ le numéro du signal - terminaison anormale
- ❑ le numéro du signal - arrêt provisoire (stop)
- ❑ un indicateur si fichier **core** (dump)

Terminaison *vs* stop Libération ressources ou possib. SIGCONT

Après la mort d'un processus?

Table des Processus



Zombie

- ✓Descripteur
- ✓État
- ✓¬mémoire
- ✓¬ressource

- ❑ Le père est alerté par un signal SIGCHLD
- ❑ L'entrée disparait quand le père à récupéré l'état (wait)

Orphelin: et si le père meurt avant…

- ❑ init (éternel) adopte le processus fils
- ❑ init fera le wait…

Reconnaître un orphelin?

wait: Synchronisation père/fils

Syntaxe :

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

Retour :

- *PID* du processus terminé, si OK,
- -1, si *erreur* (plus de fils à attendre)

Pour recevoir l'état

Quel fils ?

- Processus s'endort – Et si déjà un zombie?
- Mort d'un fils: signal SIGCHILD
- Réveil du père
- Lien de parenté: groupe de processus → voir signaux
- Autres: ***wait, waitpid, wait3, wait4***

wait: exemple simple (1)

```
#include <sys/types.h>
int main () {
    int pid, tmp;
    ...
    if ( (pid = fork ()) < 0) erreur;
    if (pid == 0)
        /* Traitement fils */
    else {
        /* Traitement père */
        tmp = wait (0)
}
```

Informations possible

Utilisation: synchronisation entre le père et n fils

n = 1: tmp = P.I.D. du fils

n > 1: tmp = P.I.D. d'un fils

wait: exemple simple (2)

```
#include <signal.h>
int main () {
        int pid, res, etat;
        if ( (pid = fork()) == 0) {
                ...
                exit (1)
        }
        else
                res = wait (&etat);
        ...
}
```

Etat du processus

waitpid: Synchronisation père/fils

Syntaxe :

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

l'état

Retour :

- *PID* du processus terminé, si OK
- 0, si plus de processus à attendre (cas de l'option `WNOHANG`) ,
- -1, si *erreur* (plus de fils à attendre)

- ❑ < -1 : processus de ce groupe
- ❑ = -1 : ∀ processus fils
- ❑ = 0 : processus du groupe
- ❑ = > 0 : ce processus

- ❑ rien: blocage normal
- ❑ WNOHANG: pas de blocage
- ❑ WUNTRACED : processus stoppés non interrogés

waitpid: exemple simple

```
#include <signal.h>
int main () {
  int pid, res, etat;
  if ( (pid = fork()) == 0) {
          ...
          exit (1)
    }
  else
     if (waitpid(pid, &etat, WNOHANG) == -1)
          printf(« le processus fils n'est pas terminé »);
  ...
}
```


Wait3/4: Synchronisation père/fils

Syntaxe :

```
#include <sys/types.h> #include <sys/wait.h>
#include <sys/resource.h> #include <sys/time.h>
pid_t wait3 (int* status, int options,
             struct rusage* rusage);
pid_t wait4 (pid_t pid, int* status,
      int options, struct rusage* rusage);
```

Retour :

- comme *waitpid* : *PID* du processus terminé, si OK
- 0, si plus de processus à attendre ,
- -1, si *erreur*

- ✓ temps
- ✓ mémoire
- ✓ nb swap
- ✓ nb signaux
- ✓ ...

- ≈ waitpid + ressources utilisées (voir getrusage(2))
- processus stoppé ou terminé

Evaluer le temps d'exécution (1)

Syntaxe :

```
#include <sys/times.h>
clock_t times ( struct tms* buf )
struct tms {
  clock_t tms_utime ; /* CPU user du processus) */
  clock_t tms_stime ; /* CPU système du processus */
  clock_t tms_cutime ; /* CPU user de ses fils */
  clock_t tms_cstime ; /* CPU système de ses fils */
}
```

Retour : temps réel *(heure de la montre)*, passé depuis la création du processus.,

pulsations d'horloge (sysconf)

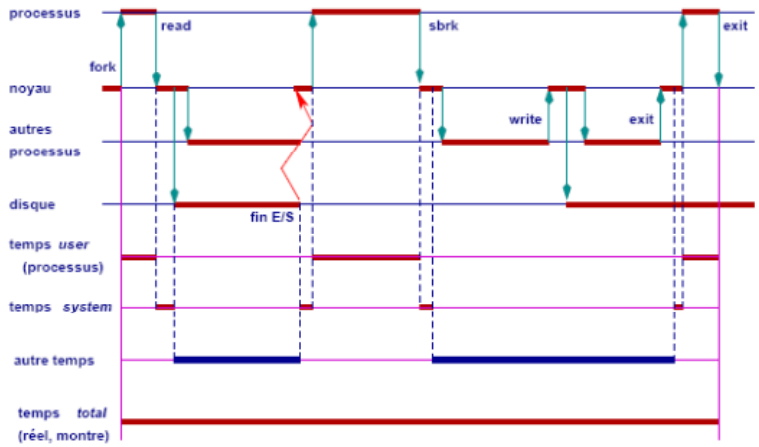
Programme *vs* Appels syst.

cumulation

time (shell - options) : `% time gcc -c mon_prog.c`

- ✓ temps total = temps réel de times
- ✓ temps user = tms_utime + tms_cutime
- ✓ temps system = tms_stime + tms_cstime

Evaluer le temps d'exécution (2)



- ✓ temps total = temps *user* + temps *system* + autre temps
- ✓ autre temps = temps E/S + temps pour autres processus

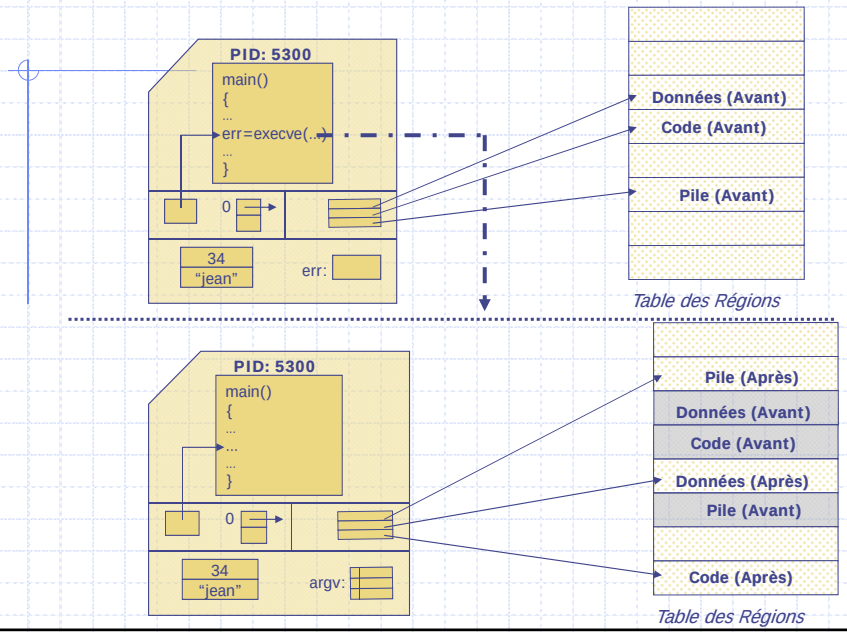
Evaluer le temps d'exécution (3)

```
#include <sys/times.h> #include <unistd.h>
struct tms buf1, buf2;
double top1, top2; long ticks;
if ((ticks= sysconf (_SC_CLK_TCK)) <0)
      ERRNO_EXIT;
top1 = (double) times(&buf1);
...  /* un calcul */
top2 = (double) times(&buf2);
printf("REAL: %7.2f USER: %7.2f\n",
  (top2 - top1)/ticks,
  (buf1.tms_utime - buf2.tms_utime)/ticks)
```

Nb puls. / sec

Evaluer un temps de calcul en secondes :
temps « réel » et « user »

Exécution d'un nouveau code:



execve: exécution d'un nouveau code

Syntaxe :

```
#include <unistd.h>
int execve ( const char* path,
             char* const argv[ ],
             char* const envp[ ] );
```

Retour :

- pas de retour si *OK*,
- -1, si *erreur*

L'exécutable

Arguments effectifs

Variables d'environnement

envp[*i*] = nom=valeur\0

- chemin relatif ou absolu
- chaque argument termine par \0
- Dernier élément des tableaux est NULL
- Rappel: argv[0] = nom point d'entrée

Exécution d'un nouveau code (1)

- Appels systèmes et routines *exec*
  - un appel système: *execve*
  - Une famille de routines C
  - → *execp, execvp, execl, execlp, execle*
  - Les structures *argv, argc, envp*
  - La variable *PATH* ou chemin absolu/relatif
- Principales actions
  - Sauvegarde paramètres de *exec*
  - Désallocation des régions
  - Chargement éventuel de l'inode et du contenu
  - Allocation des nouvelles régions

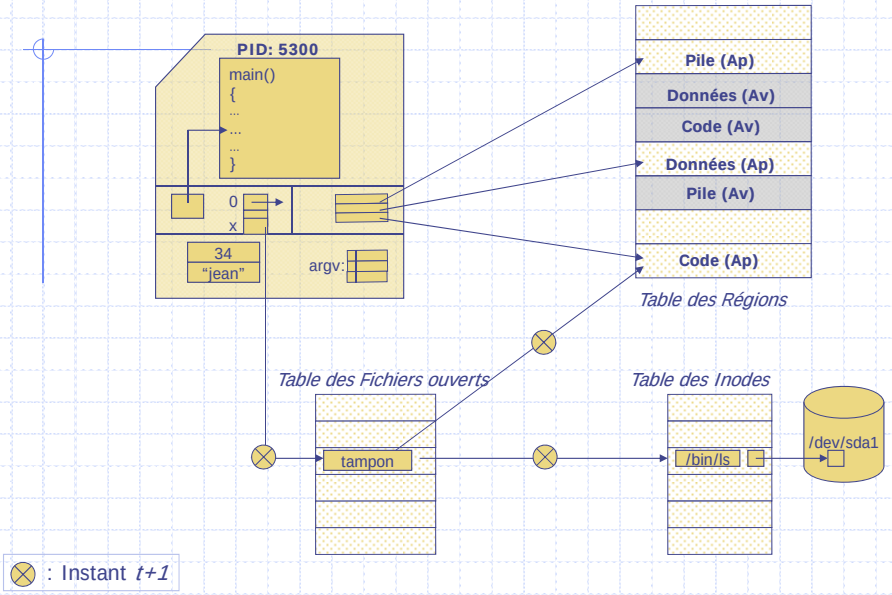
Exécution d'un nouveau code (2)

- Principales actions (suite)
  - Zone utilisateur:
    - Descripteurs non modifiés : 0, 1, 2
    - Autres descripteurs fermés
    - Effacement de la capture des signaux
      → voir *cours sur les signaux*
    - Curseur: au départ
  - Chargement exécutable

➔ Remplacement par nouveau code + exécution

Si FD_CLOEXEC est positionné (fcntl) ⇒ descripteur fermé

execve et les structures internes



exec(ve): exemple simple

```
#include <sys/types.h>
int main (int argc, char* argv[], char* envp[])
   {
        int pid, tmp;
         ...
        argv[0] = " ls ";
        argv[1] = " -l ";
        argv[2] = NULL;
        execve ("/bin/ls", argv, envp)
   }
```

(char* 0)

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

execl ("/bin/ls", "ls", "-l", NULL)

execle ("/bin/ls", "ls", "-l", NULL, envp)

exec: exemple simple d'utilisation

```
#include <sys/types.h>
int main ()
{  int pid, tmp;
    ...
    if ( (pid = fork ()) < 0) erreur;
    if (pid == 0)
          /* Traitement fils */
          execve ("/bin/ls", argv, envp)
 else {
          /* Traitement père */
          tmp = wait (0)
    }
}
```

Modif poss. si S_ISUID / S_ISGID :
- ✓ EUID , EGID
- ✓ SUID, SGID
- ✓ pas RUID , RGID

printf("Erreur…");

Fonctionnement du shell

Interprétation d'une commande
- Analyse syntaxique
- Substitutions
- Exécution:
  - Création d'un shell fils
  - Processus fils:
    - Redirections et aboutements
    - Application de *exec*
  - Processus père:
    - Arrière plan vs premier plan
    - Attente d'un signal (*wait*)

Variables d'environnement et exportation :
. commande = modif. possib.
commande = ¬ modification

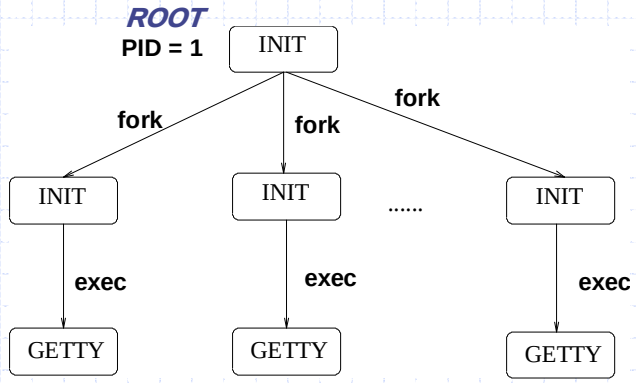
Fonctionnement du shell

```
#include <sys/types.h>
int main () {
    int pid, tmp, status;
    ...
    if ( (pid = fork ()) < 0) erreur;
    if (pid == 0)
          /* Traitement fils */
          execve (commande analysée, argv, envp)
    else {
          /* Traitement père */
          res = wait (&status)
          /* traitement de status */
    } ... }
```

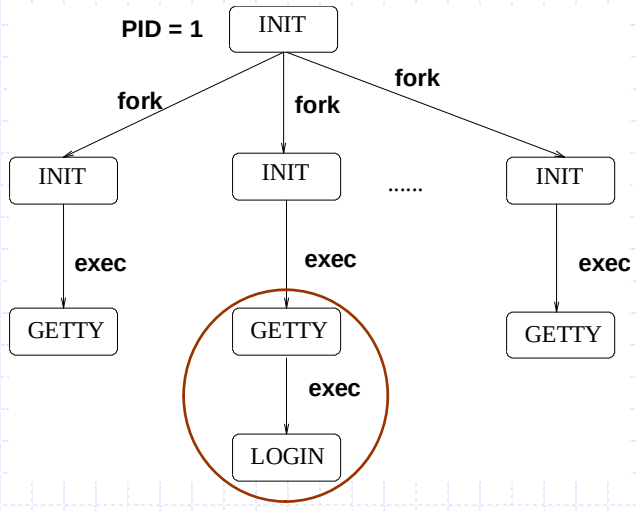
➔ Exécution en premier plan
➔ Et en arrière plan ?

WEXITSTATUS(status)
code de retour exit()

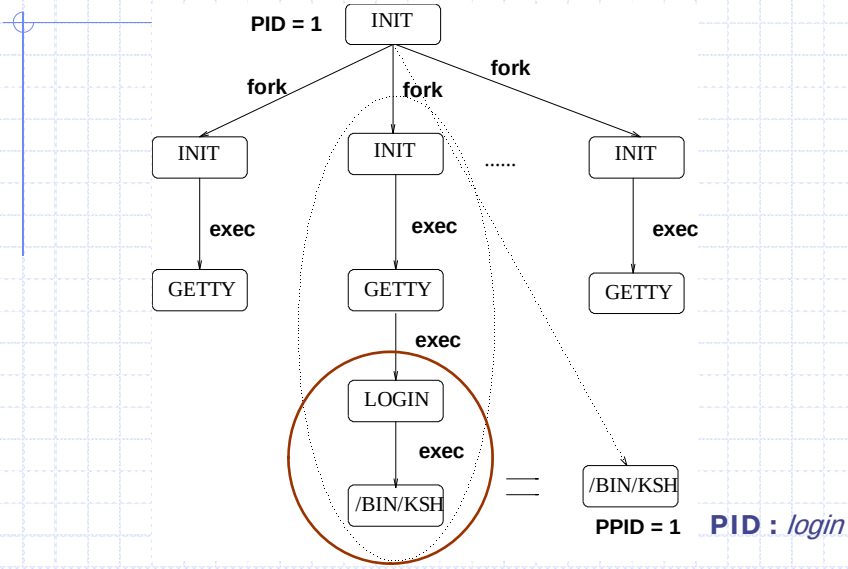
Ouverture d'une session (attente)



Ouverture d'une session (En cours)



Ouverture d'une session (Réalisée)



system: exécution d'un sous-shell

Syntaxe :

```
#include <stdlib.h>
int system (const char* cmdstring);
```

Ligne de commande

Retour :

- 0 à 126, si OK :
  statut de terminaison du shell (*exit* ou signal).,
- -1, si échec de *fork*
- 127, si échec de exec

- ❑ Pas supporté par tous les OS [*if (system(NULL))* *oui*]
- ❑ Bon exercice:
  - ✓ fork
  - ✓ execve
  - ✓ waitpid
  - ✓ ...

Exemples :

```
system("rm -f *.o");
system("date > prog.log");
```

Exercice: implémentation de system

```
#include ...
int system (const char* cmd){
  pid_t pid; int status;
  /* test d'implémentation :  c'est vrai sous Unix */
  if (cmd == NULL) return(1);
  if ( (pid = fork()) == 0) { /* **** FILS **** */
    execl("/bin/sh", "sh", "-c", cmd, (char*)0);
    _exit(127); /* échec de execl */
/* **** PERE **** */
  } else if (pid <0) { status= -1;
  } else { /* Attente robuste: même si waitpid est */
      /* interrompu par un signal */
    while (waitpid(pid, &status, 0) < 0)
      if (errno != EINTR){/* waitpid interrompu */
        status = -1; /* autre cas => erreur */
        break; }
  }
  return (status);
}
```

trop de fork

Pas de traitement des signaux

Execution: binaires vs scripts shells

- Choix basé sur le début du fichier (voir file(1))
  - Binaire : nombre magique (protocole de chargement)
  - Script : #! *Chemin interprète*
- Reliure du programme binaire
  - Statique : protocoles de chargement (à la demande…)
  - Dynamique (charge aussi le relieur dynamique)
    Bibliothèque partageable *nom*.so
- Chargement et exécution :
  - Binaire : c'est le programme (préparé par ld)
    S'appuie sur le runtime d'un langage
  - Script : C'est l'interprète
    C'est lui qui exécute le script

Communication inter-processus
Des solutions adaptées

- Partie Système
  - Mécanisme de redirection (***dup***)
  - Comunication par tubes (***dup***, ***pipe***)
  - envoi/traitement de signaux (***signal***, ***kill***)
- Partie Réseau
  - Bibliothèque *IPC** (même machine)
    - Mémoire partagée, Sémaphores
    - Files de messages
  - Bibliothèque des sockets (tout le réseau)

* *Inter Process Communication*