Processus sous UNIX (rappels du L2)

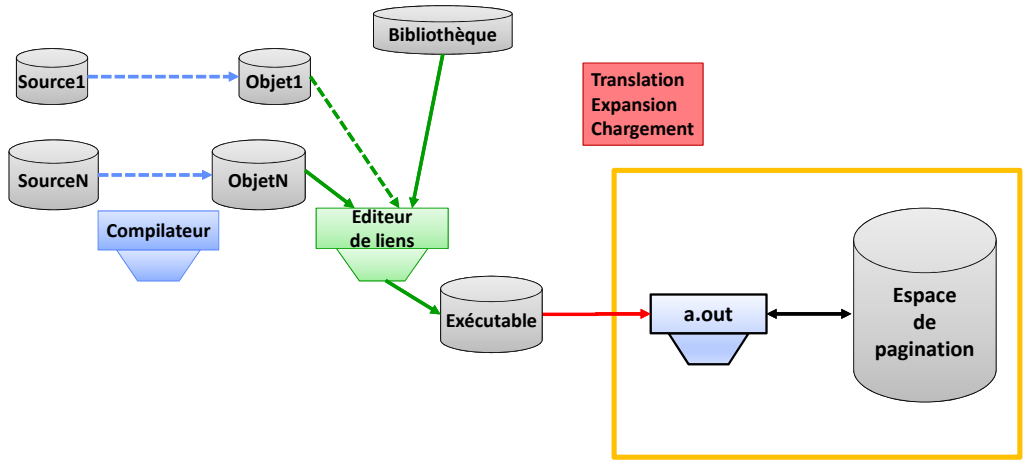
Rappel : classes de variables en langage C

- **Variable statique**
  - Variable ayant la durée de vie du processus
  - Les variables définies hors des fonctions sont toujours statiques
  - Les variables locales à une fonction peuvent comporter l'attribut static
- **Variable automatique**
  - Variables créées lorsque le bloc dans lequel elles sont déclarées est exécuté et supprimées à la sortie du même bloc
  - Variables allouées dans la pile d'exécution (permet la récursivité des fonctions: plusieurs instances de la même variable peuvent coexister)
- **Variable dynamique**
  - Variables créées à la demande explicite du programmeur (fonction malloc)
  - Variables allouées dans une zone spécifique : le tas (heap)
  - La variable existe jusqu'à la destruction explicite (fonction free)

Classes de variables : Exemple

```c
#include <stdlib.h>

/* Zone des variable globales */

int V_globale;                /* statique */
int V_globale_init = 12;      /* statique */

main()
{
  int V_locale;               /* automatique */
  static int V_local_static;  /* statique */

  char * PTR;                 /* automatique */

  PTR = malloc(13);           /* dynamique */

}
```


Production d'un exécutable



Exécutable sur disque

- Formats a.out ou ELF
- En-tête, contenant notamment
  - Le « Magic Number » : fichier directement exécutable ou interprétable (+ indique la méthode d'interprétation)
  - Le « sticky bit » : fichier résident en mémoire centrale
  - Indication de code partageable
- Code
- Données
  - Les variables non initialisées ne sont pas présentes, seule leur taille totale est spécifiée
- Table des symboles
  - Liens utiles pour une éventuelle future édition de liens (dynamique) et débogage

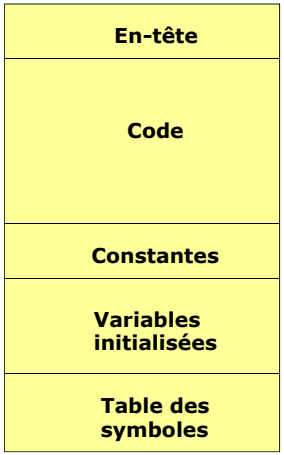
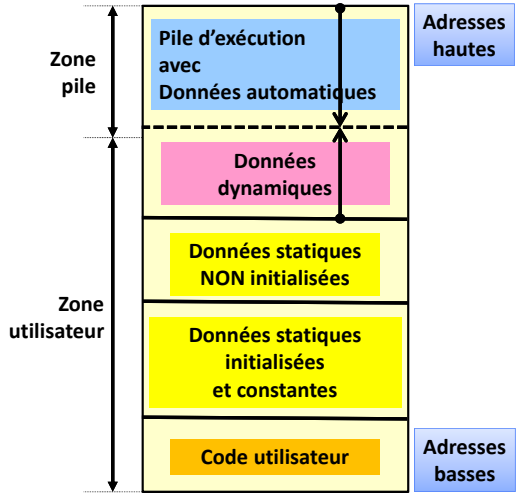


Image d'un processus

- A tout processus est associé un ensemble d'informations appelé image
- Lorsqu'un programme est "lancé", le système crée un processus et construit son image, contenant :
  - Un descripteur contenant les informations générales au processus (PCB)
  - Un segment de données privé au processus (constantes, données statiques et la zone dynamique)
  - Un segment pile privé au processus qui permet de gérer les appels de sous-programmes
  - Un segment de code partageable entre plusieurs processus qui contient le code à exécuter par le processus
  - Code réentrant : peut être exécuté en parallèle par plusieurs processus en produisant le même résultat que s'il avait été exécuté successivement par chacun de ces processus
  - Un code réentrant est donc partageable. S'il ne l'est pas, le programmeur doit utiliser des mécanismes de synchronisation pour assurer la cohérence des ressources critiques partagées

Image d'un processus en mémoire virtuelle



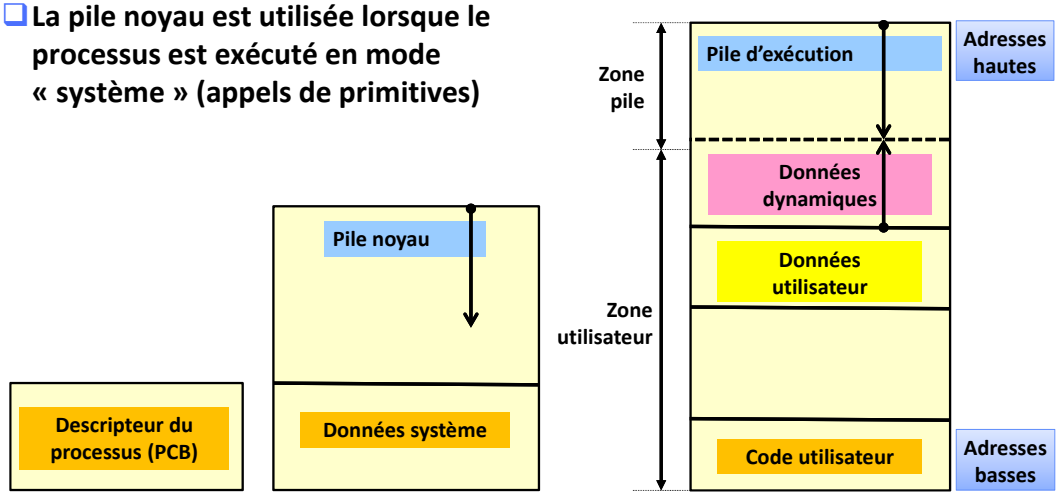
La zone « Données statiques NON initialisées » est remplie avec des zéros

PCB (Process Control Block)

- Descripteur du processus, contenant :
  - Le numéro interne du processus
  - Le numéro du processus père
  - Les variables d'environnement du processus
  - D'autres informations
    - Propriétaire, groupe du propriétaire
    - Priorité
    - Descriptifs des fichiers ouverts
    - Informations pour des statistiques ou comptabilité

Processus comme objet du système

- La pile noyau est utilisée lorsque le processus est exécuté en mode « système » (appels de primitives)



Caractéristiques générales des processus UNIX

- Identification
  - Tout processus possède un identificateur habituellement appelé PID
- Création dynamique, hiérarchie
  - Le processus qui exécute l'action de création est appelé processus parent, le processus créé est appelé processus fils

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

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


Les propriétaires d'un processus

- UNIX associe à tout processus, un utilisateur propriétaire (le véritable)
- Le propriétaire est désigné par son numéro interne : le real user_ID
- UNIX associe un propriétaire effectif lors de l'exécution d'un programme :
  - Effective user id, utilisé pour évaluer effectivement les droits d'accès du processus aux ressources (fichiers, sémaphores, etc.)
  - Permet d'attribuer temporairement les droits d'accès du propriétaire du fichier exécutable à un processus créé par un autre utilisateur, exécutant ce programme
  - Par défaut, le effective user id est égal au real user id. La commande chmod permet de positionner le setuid_bit d'un exécutable, pour qu'un processus P ait les droits d'accès du propriétaire de cet exécutable pendant que P exécute ce code
  - Exemple : la commande passwd

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

uid_t getuid(void);
uid_t geteuid(void);
```

Véritable propriétaire (real user id)

Propriétaire temporaire (effective user id)

Effective user id = Real user id

```
#include <stdio.h>
#include <unistd.h>

int main()
{
  printf("*** Debut du travail\n");
  printf("Processus courant: %d\n", (int)getpid());
  printf("Processus pere:    %d\n", (int)getppid());
  printf("***\n");
  printf("Proprietaire reel:       %ld\n", (long)getuid());
  printf("Proprietaire effectif:   %ld\n", (long)geteuid());
  printf("***\n");
  system("ps -al");
  printf("*** Fin du travail \n");
}
```

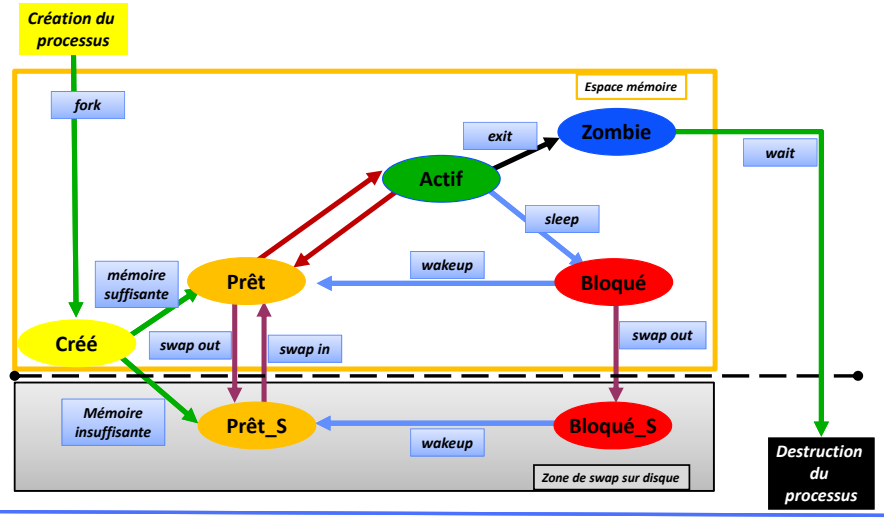
```
luciole %./getpid
*** Debut du travail
Processus courant: 312
Processus pere:    239
***
Proprietaire reel:       501
Proprietaire effectif:   501
***
  UID   PID  PPID CPU PRI NI      VSZ    RSS WCHAN  STAT  TT       TIME COMMAND
  501   239   237   0  31  0    27812    864 -      Ss    p1    0:00.17 -bash
  501   312   239   0  31  0    36644    772 -      S+    p1    0:00.01 ./getpid
    0   313   312   0  29  0    27308    404 -      R+    p1    0:00.01 ps -al
  501   247   237   0  31  0    27812    892 -      Ss+   p2    0:00.05 -bash
*** Fin du travail
luciole %
```

Effective user id != Real user id

```
#include <stdio.h>
#include <unistd.h>

int main()
{
  printf("*** Debut du travail\n");
  printf("Processus courant: %d\n", (int)getpid());
  printf("Processus pere:    %d\n", (int)getppid());
  printf("***\n");
  printf("Proprietaire reel:       %ld\n", (long)getuid());
  printf("Proprietaire effectif:   %ld\n", (long)geteuid());
  printf("***\n");
  system("ps -al");
  printf("*** Fin du travail \n");
}
```

```
luciole %./getpid
*** Debut du travail
Processus courant: 312
Processus pere:    239
***
Proprietaire reel:       501
Proprietaire effectif:   501
***
  UID   PID  PPID CPU PRI NI      VSZ    RSS WCHAN  STAT  TT       TIME COMMAND
  501   239   237   0  31  0    27812    864 -      Ss    p1    0:00.17 -bash
  501   312   239   0  31  0    36644    772 -      S+    p1    0:00.01 ./getpid
    0   313   312   0  29  0    27308    404 -      R+    p1    0:00.01 ps -al
  501   247   237   0  31  0    27812    892 -      Ss+   p2    0:00.05 -bash
*** Fin du travail
luciole %
```


États d'un processus – Niveau physique



La commande ps

- La commande ps donne des informations sur les processus du système
- Les informations fournies varient selon les constructeurs
- Voir le man pour plus de détails
  - UID : numéro propriétaire
  - PID : numéro processus
  - PPID : numéro du processus père
  - PRI : priorité du processus
  - RSS : mémoire occupée (en Koctets)
  - STAT : état (S: sleeping < 20 s, I: Idle > 20 s, R: running, Z: zombie)

```
luciole %ps -al
  UID   PID  PPID CPU PRI NI      VSZ    RSS WCHAN  STAT  TT       TIME COMMAND
  501   239   237   0  31  0    27812    864 -      Ss    p1    0:00.17 -bash
    0   320   239   0  31  0    27292    404 -      R+    p1    0:00.01 ps -al
  501   247   237   0  31  0    27812    892 -      Ss+   p2    0:00.05 -bash
luciole %
```


Création d'un processus UNIX

- Un processus UNIX est créé par un autre processus sur un appel de la primitive fork
- Seuls deux processus sont créés statiquement à l'initialisation du système
  - Le processus 0 (swapper, scheduler, cpu_idle)
  - Le processus 1, son fils init, qui active les processus prévus dans le fichier d'initialisation du système
- Le nombre de processus évoluant dans un système est variable et peut croître jusqu'à atteindre une valeur limite (dans l'ensemble du système ou pour un utilisateur)
- Le type pid_t est un type entier (int ou long)

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

pid_t fork(void);
```

Création d'un processus fils à l'image du père

- **Très important**
  - Copie de la plupart des attributs, à l'exception
    - Du pid
    - Du pid du père
    - De al localisation des segments donnée et pile
    - De quelques autres informations « personnelles »
  - Copie du segment de données
  - Copie du segment pile
  - Partage du segment de code

Sémantique de la primitive fork

- **Très important**
- Les deux processus père et fils repartent en parallèle
  - La prochaine instruction exécutée par le père et par le fils est l'instruction qui suit l'appel de fork
- Le père revient de fork avec comme valeur de retour le numéro du fils qu'il vient de créer
- Le fils revient de fork avec comme valeur de retour 0
- Dans le cas où la création n'a pas pu être effectuée correctement, le père revient de fork avec comme valeur de retour -1
- Pour mieux comprendre : voir l'animation disponible sous Moodle

Exemple type de création d'un processus

```
pid_t pid;

switch (pid = fork()) {
  case -1 : /* Cas d'erreur */
    printf("Bug lors de création \n");
    ...

  case 0 :  /* Code du processus FILS */
    printf("Je suis le fils\n");
    ...

  default : /* Code du processus PERE */
    printf("Je suis le père:
                    fils créé %d\n, pid");
    ...
}
```

La création du fils n'a pas pu être effectuée
Le retour de fork est -1
C'est le processus père qui est exécuté

La création du fils a été effectuée
Le retour de fork est 0
C'est le processus fils qui est exécuté

La création du fils a été effectuée
Le retour de fork est positif
C'est le processus père
qui est exécuté

Copy-on-Write (COW) – Copie sur écriture

- **Stratégie d'optimisation**
  - Le processus créé dispose de pointeurs sur les données du processus père
  - Cette liaison est conservée jusqu'à ce qu'un des deux processus modifie la donnée; dans ce cas, une copie privée est créée
  - Dans les cas où il n'y a pas de modification, la copie n'est pas créée
  - Utilisation fréquente avec la mémoire virtuelle

Exemple de création de processus

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
  printf("*** Debut du travail processus principal\n");
  printf("Processus principal: %d\n", (int)getpid());
  switch (fork())
  {
    case -1 :
    {
      printf("ERREUR: Creation processus fils %d\n", (int)getpid());
      exit(99);
    }
    case 0 :
    {
      printf("FILS demarre \n");
      printf("FILS %d en cours d'execution \n", (int)getpid());
      printf("FILS se termine\n");
      exit(50);
    }
    default :
    {
      printf("PERE %d en cours d'execution \n", (int)getpid());
    }
  }
  printf("*** Fin du travail du pere \n");
  exit(1);
}
```

La création du fils n'a pas pu être effectuée
Le retour de fork est -1
C'est le processus père qui est exécuté

La création du fils a été effectuée
Le retour de fork est 0
C'est le processus fils qui est exécuté

La création du fils a été effectuée
Le retour de fork est positif
C'est le processus père qui est exécuté

Terminaison d'un processus

- **Terminaison**
  - **Normale et prévue**
    - Le processus décide lui-même de se terminer
    - Il s'agit de la fin de l'algorithme du processus
    - Le processus informe le système de cette décision
  - **Anormale ou imprévue**
    - Destruction par un autre processus
    - Destruction par le système
      - Suite à une anomalie de fonctionnement du processus
      - Dans le cadre d'une politique de gestion des processus

Terminaison d'un processus – Actions réalisées

- Le système récupère les ressources détenues par le processus et met à jour les statistiques
- Tous les fichiers ouverts par le processus sont automatiquement fermés
- Réveille et informe le processus père si celui-ci est bloqué en attente de la terminaison d'un fils
- Si le processus père n'est pas en attente, le processus devient un zombie
- Rattachement des processus fils (orphelins) au processus d'accueil : le processus init de pid égal à 1
- Le processus d'accueil (init) attend périodiquement les orphelins afin de les faire disparaître

Terminaison normale d'un processus

- Le processus rend le contrôle au système qui réalise la terminaison
- Émission d'un compte-rendu au processus père si le père qui attend a demandé ce compte-rendu
- Solutions
  - Appel à la fonction return dans le corps de la fonction main
  - Rencontre de la fin du corps de main
  - Appel à la fonction exit ou à la primitive _exit

```c
#include <stdlib.h>
#include <unistd.h>

void exit(int status);
void _exit(int status);
```

- Sémantique
  - La fonction exit assure la fermeture des fichiers encore ouverts
  - La fonction exit permet l'exécution des handlers installés avant la terminaison
  - L'installation de handlers est faite par la fonction atexit
  - L'appel à la primitive _exit est effectué à la suite de l'exécution de exit

Exemple de terminaison de processus

```
pid_t pid;
switch (pid = fork()) {
  case -1 :
    perror("Bug lors de création \n");
    exit(99);

  case 0 :
    printf("Je suis le fils\n");
    exit(1);

  default :
    printf("Je suis le père de %d\n" , pid);
    ...

}
```

Terminaison du processus père et compte-rendu de 99 au processus grand-père (pas de création de fils)

Terminaison du processus fils et compte-rendu de 1 au processus père

Exemple de terminaison de processus

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
  printf("*** Debut du travail processus principal\n");
  printf("Processus principal: %d\n", (int)getpid());
  switch (fork())
  {
    case -1 :
    {
      printf("ERREUR: Creation processus fils %d\n", (int)getpid());
      exit(99);
    }
    case 0 :
    {
      printf("FILS demarre \n");
      printf("FILS %d en cours d'execution \n", (int)getpid());
      printf("FILS se termine\n");
      exit(50);
    }
    default :
    {
      printf("PERE %d en cours d'execution \n", (int)getpid());
    }
  }
  printf("*** Fin du travail du pere \n");
  exit(1);
}
```

Terminaison du processus père et compte-rendu de 99 au processus grand-père (pas de création de fils)

Terminaison du processus fils et compte-rendu de 50 au processus père

Terminaison du processus père et compte-rendu de 1 au processus grand-père

Exemple de terminaison de processus

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
  printf("*** Debut du travail processus principal\n");
  printf("Processus principal: %d\n", (int)getpid());
  switch (fork())
  {
    case -1 :
    {
      printf("ERREUR: Creation processus fils %d\n", (int)getpid());
      exit(99);
    }
    case 0 :
    {
      printf("FILS demarre \n");
      printf("FILS %d en cours d'execution \n", (int)getpid());
      printf("FILS se termine\n");
      exit(50);
    }
    default :
    {
      printf("PERE %d en cours d'execution \n", (int)getpid());
    }
  }
  printf("*** Fin du travail du pere \n");
  exit(1);
}
```

```
luciole %./fork
*** Debut du travail processus principal
Processus principal: 335
PERE 335 en cours d'execution
*** Fin du travail du pere
FILS demarre
FILS 336 en cours d'execution
FILS se termine
luciole %
```


Attente de la terminaison d'un processus fils

- **Principe**
  - Après avoir créé un processus fils, processus parent et processus fils progressent en parallèle à partir du point d'appel de fork
  - Si le processus parent souhaite attendre la fin de l'exécution du processus fils, il va appeler la primitive wait ou waitpid

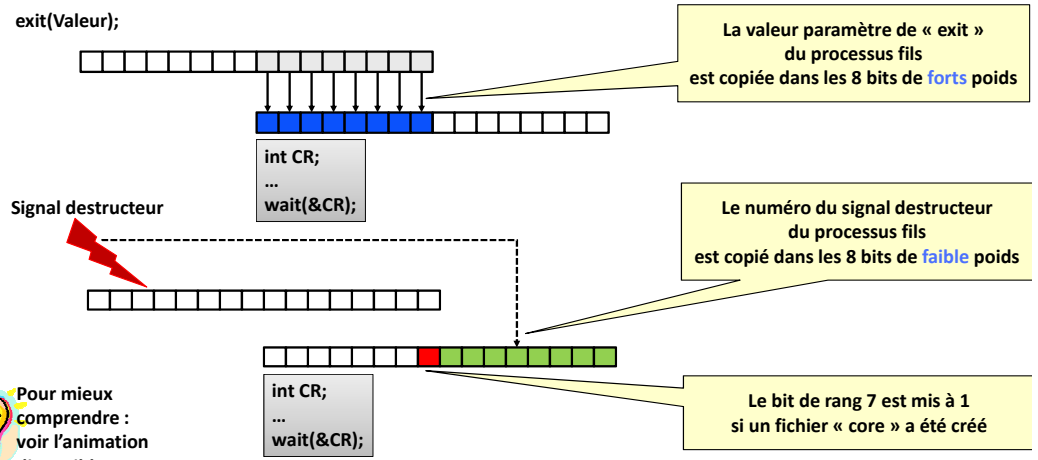
Attente de la terminaison d'un fils – Primitive wait

- Suspension de l'exécution du processus parent jusqu'à l'un des deux événements suivants
  - Terminaison d'un fils
  - Réception d'un signal
- Il n'y a pas de suspension si
  - Un fils est déjà terminé et n'a pas été attendu
  - Il n'y a plus de processus fils à attendre
- La valeur retournée par la primitive wait est
  - le pid du fils terminé
  - -1 sinon, et la variable errno contient alors :
    - ECHILD si il n'y a plus de processus fils actif
    - EINTR si l'appel a été interrompu par un signal

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

pid_t wait(int *CR);
```

- Le paramètre CR est un résultat
  - si le pointeur est différent de NULL, la variable pointée reçoit le compte-rendu de la terminaison
  - sinon, aucun compte-rendu n'est transmis
- Le compte-rendu peut être analysé par les macros POSIX

Mécanisme du wait/exit



Pour mieux comprendre : voir l'animation disponible sous Moodle

Attente de la terminaison d'un fils – Primitive waitpid

- Un processus peut attendre un fils particulier
- Le paramètre pid précise le processus fils à attendre
  Si le paramètre pid vaut -1, tout processus fils est attendu
- L'utilisation de WNOHANG comme paramètre options permet une attente non bloquante. Dans ce cas :
  - si waitpid retourne 0, cela signifie que des fils existent encore mais aucun fils n'est actuellement terminé et prêt à disparaître (zombie)
  - si waitpid retourne -1, cela signifie probablement qu'il n'existe plus de fils vivants ou zombies (errno précise la raison de ce retour en erreur)

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

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


Les macros POSIX

- Syntaxe : **macro (compteRendu)**

WIFEXITED	Produit une valeur non nulle si le compte-rendu correspond à un fils terminé par exit
WIFSIGNALED	Produit une valeur non nulle si le compte-rendu correspond à un fils détruit par un signal
WEXITSTATUS	Valeur de type byte spécifiée dans le paramètre du exit par le processus fils terminé
WTERMSIG	Numéro du signal ayant détruit le processus associé au compte-rendu

Exemple de récupération du compte-rendu

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main()
{
  int CR;
  pid_t Fils;
  printf("*** Debut du travail processus principal\n");
  printf("Processus principal: %d\n", (int)getpid());
  switch (fork())
  {
    case -1 :
    {
      printf("ERREUR: Creation processus fils %d\n", (int)getpid());
      exit(99);
    }
    case 0 :
    {
      printf("FILS demarre \n");
      printf("FILS %d en cours d'execution \n", (int)getpi
      printf("FILS se termine - CR = 50\n");
      exit(50);
    }
    default :
    {
      printf("PERE %d en cours  execution \n", (int)getpid());
    }
  }
  Fils = wait(&CR);
  printf("PERE attend \n");
  printf("PERE: Compte-rendu du fils %d - CR1: %d - CR2: %d \n", (int)Fils, CR, WEXITSTATUS(CR));
  printf("*** Fin du travail du pere \n");
  exit(1);
}
```

Attente de la terminaison du processus fils :
Le numéro du fils sera récupéré dans la variable Fils.
Le compte-rendu de sa terminaison
sera récupéré dans la variable CR

```
l  wait
 ebut du travail processus principal
rocessus principal: 351
PERE 351 en cours d'execution
FILS demarre
FILS 352 en cours d'execution
FILS se termine - CR = 50
PERE attend
PERE: Compte-rendu du fils 352 - CR1: 12800 - CR2: 50
*** Fin du travail du pere
luciole %
```


Exercices 1 & 2

- **Exercice 1**
  - Écrire une commande dans laquelle le processus père crée 3 processus fils et mémorise leur identifiant dans un tableau. Veuillez pour chaque processus donner le contenu de ce tableau et la valeur de toutes les variables utilisées.
- **Exercice 2**
  - Écrire une commande qui permet à un processus fils de saisir une information au clavier et au processus père d'afficher cette même information.
  - Peut-on transmettre des informations de type quelconque ?

Paramètres de la fonction main

```
int main (int Argc, char *Argv[ ], char *Env[ ])
```

- Argc : nombre d'éléments du tableau Argv
- Argc : tableau contenant des pointeurs vers des chaînes de caractères définissant les paramètres de l'application
- Par convention, le premier argument Argv[0] désigne le nom de la commande (nom du fichier programme)
- Env[] : tableau précisant l'environnement dans lequel doit être exécutée l'application

Commutation d'images

- **Principe**
  - Un processus peut dynamiquement modifier l'image qu'il exécute
  - Cette modification permet d'associer les segments de données et de code définis dans un fichier exécutable
  - Les autres attributs du processus restent inchangés (pid, groupe, priorité…)
  - La pile est vidée
- **Primitives de commutation d'images**

```
#include <unistd.h>

int execl
  (const char *cheminAcces,
   const char *arg0,
   …          *arg1, …,
   const char *argn,
   const char * /*NULL*/);
```

```
#include <unistd.h>

int execv
  (const char *cheminAcces,
   char *const tabArg[]);
```

Commutation d'images – Sémantique

- L'exécution commence dans le nouveau code à la première instruction exécutable de la fonction main
- Si la commutation d'image a correctement fonctionné, l'instruction qui suit l'appel de la primitive exec n'est pas exécutée
  - Il n'y a jamais de retour d'un exec
- La fonction main récupère les paramètres suivant l'appel de la primitive exec en utilisant argc et argv
- Les paramètres qui seront transmis à la fonction peuvent être spécifiés
  - Sous forme de liste (execl)
  - Sous la forme d'un tableau (execv)
  - Dans les 2 cas, le dernier paramètre est le pointeur NULL

Commutation d'images – Primitives (suite)

- Prise en compte de la variable d'environnement path

```
#include <unistd.h>

int execlp
  (const char *cheminAcces,
   const char *arg0,
   const char *arg1,
                ... ,
   const char *argn,
   const char * /*NULL*/);
```

```
#include <unistd.h>

int execvp
  (const char *cheminAcces,
   char *const tabArg[]);
```


Exemple de commutation d'image

```
/* Lecture et analyse de la ligne de commande        */
/*  le texte de la commande est dans la variable COM */
/* Deux paramètres sont dans P1 et P2                */
int CR;

/* Création d'un processus fils pour exécuter la commande */
switch (fork()) {
  case -1 : perror("Problème de création du fils \n");
            exit(12);

  case  0 : /* Processus fils chargé de la commande */
            execlp (FIC, COM, P1, P2, NULL);
            printf(" Commande inconnue ou syntaxe appel exec incorrecte \n");
            exit(99);

  default : break;
}

wait (&CR);  /* Attente dans le cas d'un commande en direct */
             /* Aucune attente dans le cas du batch  */
```

Commutation d'image du processus fils vers un fichier exécutable appelé FIC

Pour mieux comprendre : voir l'animation disponible sous Moodle

Exemple de commutation d'image

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
char * V_pere[] = {"ps", "-au", NULL};
int main()
{
  int CR;
  printf("*** Debut du travail processus principal\n");
  switch (fork())
  {
    case -1 :
    {
      perror("ERREUR: Creation processus fils");
      exit(99);
    }
    case 0 :
    {
      printf("FILS demarre et execute ls -al\n");
      execlp("ls", "ls", "-al", NULL);
      perror("Erreur exec processus fils");
      exit(89);
    }
    default :
    {
      break;
    }
  }
  wait(&CR);
  printf("PERE attend puis execute ps -au\n");
  execvp("ps", V_pere);
  perror("Erreur exec processus pere");;
  exit(98);
}
```

Commutation d'image du processus fils vers le fichier exécutable ls

Commutation d'image du processus père vers le fichier exécutable ps

```
luciole %./exec
*** Debut du travail processus principal
FILS demarre et execute ls -al
total 256
drwxr-xr-x   13 jean-mar  jean-mar    442 Oct 18 17:16 .
drwxr-xr-x    5 jean-mar  jean-mar    170 Oct 18 11:58 ..
-rw-r--r--    1 jean-mar  jean-mar   6148 Oct 18 16:34 .DS_Store
-rwxr-xr-x    1 jean-mar  jean-mar  17020 Oct 18 15:23 Variables_en_C
-rw-r--r--    1 jean-mar  wheel       348 Oct 18 15:23 Variables_en_C.c
-rwxr-xr-x    1 jean-mar  jean-mar  17380 Oct 18 17:16 exec
-rw-r--r--    1 jean-mar  jean-mar    669 Oct 18 17:13 exec.c
-rwxr-xr-x    1 jean-mar  jean-mar  17268 Oct 18 16:35 fork
-rw-r--r--    1 jean-mar  wheel       640 Oct 18 16:34 fork.c
-rwxr-xr-x    1 jean-mar  jean-mar  17368 Oct 18 15:43 getpid
-rw-r--r--    1 jean-mar  jean-mar    415 Apr  6  2006 getpid.c
-rwxr-xr-x    1 jean-mar  jean-mar  17296 Oct 18 16:56 wait
-rw-r--r--    1 jean-mar  jean-mar    841 Oct 18 16:53 wait.c
PERE attend puis execute ps -au
USER       PID %CPU %MEM      VSZ    RSS  TT  STAT STARTED      TIME COMMAND
root       364   2.6  0.1    27292    404  p1  R+    5:16PM   0:00.01 ps -au
jean-mar   239   0.3  0.2    27812    864  p1  Ss    2:44PM   0:00.21 -bash
jean-mar   247   0.0  0.2    27812    892  p2  Ss+   3:04PM   0:00.05 -bash
luciole %
```

L'environnement d'un processus

- Mécanisme permettant de préciser les valeurs de défaut de certaines informations spécifiques au système ou à l'utilisateur dans un processus
- Ensemble de chaînes de caractères de la forme
  Nom=Valeur
  - Nom désigne une variable d'environnement (TERM, HOME, PATH…)
  - Val désigne la valeur de la variable
- L'environnement est accessible par la variable externe environ
- La primitive getenv permet de connaître la valeur d'une variable

```
#include <stdlib.h>

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

```
extern char **environ;
```


Commutation d'image – Primitives (suite)

- Prise en compte de l'environnement

```
#include <unistd.h>

int execle
  (const char *Chemin_Acces,
   const char *Arg0,
   const char *Arg1,
                ...,
   const char *Argn,
   const char * /*NULL*/
   char *const Tab_Env[]);
```

```
#include <unistd.h>

int execve
  (const char *Chemin_Acces,
   char *const Tab_Arg[],
   char *const Tab_Env[]);
```


Exemple de commutation d'image

getenv.c

```
#include <stdio.h>
#include <unistd.h>

extern char ** environ;
int K;

int main()
{
  printf("*** Debut du programme\n");
  printf("Processus courant: %d\n", (int)getpid());
  printf("*** Variables d'environnement\n");
  for(K=0; environ[K]!=NULL; K++)
    printf("%s\n", environ[K]);
  printf("*** Fin du programme \n");
}
```

Terminal — bash — 75x23

```
luciole %./getenv
*** Debut du programme
Processus courant: 369
*** Variables d'environnement
TERM_PROGRAM=Apple_Terminal
TERM=xterm-color
SHELL=/bin/bash
TERM_PROGRAM_VERSION=133
OLDPWD=/Users/jean-marierigaud/Enseignements/Cours/Cours_UNIX/Cours_PPU_2008/Cours_PPU_2008_B-Processus_etc
USER=jean-marierigaud
__CF_USER_TEXT_ENCODING=0x1F5:0:91

PATH=/Applications/vim:/bin:/sbin:/usr/bin:/usr/sbin
PWD=/Users/jean-marierigaud/Enseignements/Cours/Cours_UNIX/Cours_PPU_2008/Cours_PPU_2008_B-Processus_etc/PPU_2008_B_Processus_Exemples
SHLVL=1
HOME=/Users/jean-marierigaud
prompt=lucio.e
SECURITYSESSIONID=591500
_=./getenv
*** Fin du programme
luciole %
```


Exercice 3

- Écrire une commande qui effectue la combinaison ci-dessous en respectant la séquence :

  date; ls –al

Exercice 4

- On se propose de paralléliser le traitement d'une matrice de L lignes et de C colonnes, en confiant le traitement de chaque ligne à une application existante. Les valeurs de la matrice sont saisies au clavier de l'utilisateur. Après avoir lancé les L traitements pour chacune des lignes de la matrice, le processus attend leurs terminaisons avant de se terminer lui-même. Le processus doit alors afficher à l'écran de l'utilisateur, un résultat d'exécution.
- On suppose existante la fonction traiterDonnees, qui effectue un traitement sur une liste d'entiers en nombre quelconque et retourne un compte-rendu d'exécution (1 en cas de succès, 0 sinon).
- Écrire le programme C réalisant ce travail selon les trois politiques suivantes :
- a – Le résultat affiché est le nombre de lignes ayant obtenu un résultat positif
- b – Idem, mais les valeurs de L et C sont des paramètres du programme
- c – On reprend le a), mais les résultats d'exécution de chaque ligne sont affichés dans l'ordre des lignes