Processus

Généralités

Généralités

Définition

Un processus est l'image mémoire de l'exécution d'un programme ou fichier binaire.

Généralités

Définition

Un processus est l'image mémoire de l'exécution d'un programme ou fichier binaire.

- Programme ou fichier binaire :
  - objet statique ;
  - contenu identique dans le temps ;
  - les variables n'ont pas de valeur ;
  - les fichiers ne sont ni lus ni écrits.

Généralités

Définition

Un processus est l'image mémoire de l'exécution d'un programme ou fichier binaire.

- Programme ou fichier binaire :
  - objet statique ;
  - contenu identique dans le temps ;
  - les variables n'ont pas de valeur ;
  - les fichiers ne sont ni lus ni écrits.
- Processus :
  - chargement en mémoire du contenu d'un fichier binaire ;
  - variables initialisées ;
  - tableaux créés par allocation mémoire ;
  - fichiers lus, écrits.

- Un processus est un objet dynamique
- Son état varie tout au long de son exécution.

- Un processus est un objet dynamique
- Son état varie tout au long de son exécution.

Remarques

- Chaque système d'exploitation met en œuvre son propre modèle de processus.
- Pour des versions différentes du même système d'exploitation, le modèle de processus peut être différent
  - taille maximale de la mémoire utilisable
  - taille des pointeurs
  - taille des tableaux
  - propriétaire, droits
  - états du processus
  - ...

Modèles de processus UNIX

- Modèle conçu dans les années 1970
- Évolution avec les différentes versions du système UNIX puis GNU/Linux
- Caractéristiques :
  - Espace virtuel propre
  - Segmenté : plusieurs segments
  - Segments partageables
  - Identifiants de processus et de processus père uniques

Descripteur de processus

- Descripteur du processus : ensemble des informations caractérisant un processus
- Table des descripteurs de processus : ensemble des descripteurs
- Table utilisée à :
  - Création
  - Destruction
  - Allocation du processeur
  - Modification de l'état

États d'un processus

- Dans tous les systèmes d'exploitation un processus peut-être dans les trois états principaux suivants :
  - Actif : un processeur lui a été attribué et il exécute une partie de son code
  - Activable : il est prêt à être exécuté, il dispose de toutes les ressources nécessaires hormis un processeur
  - En attente ou bloqué : un événement extérieur, une ressource est/sont nécessaire(s) à son exécution
- Le nombre réels d'états varie suivant les système d'exploitation

Diagramme d'états

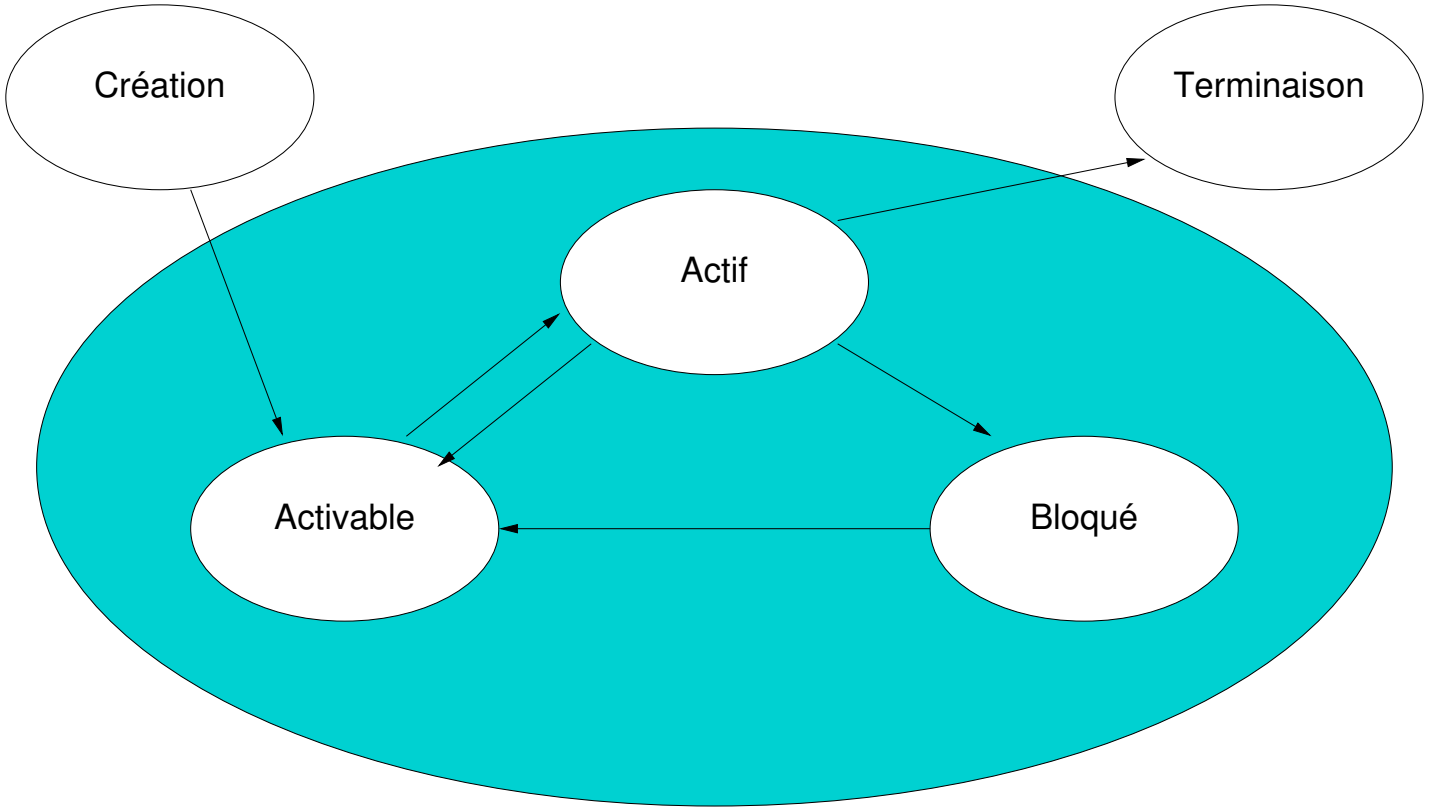
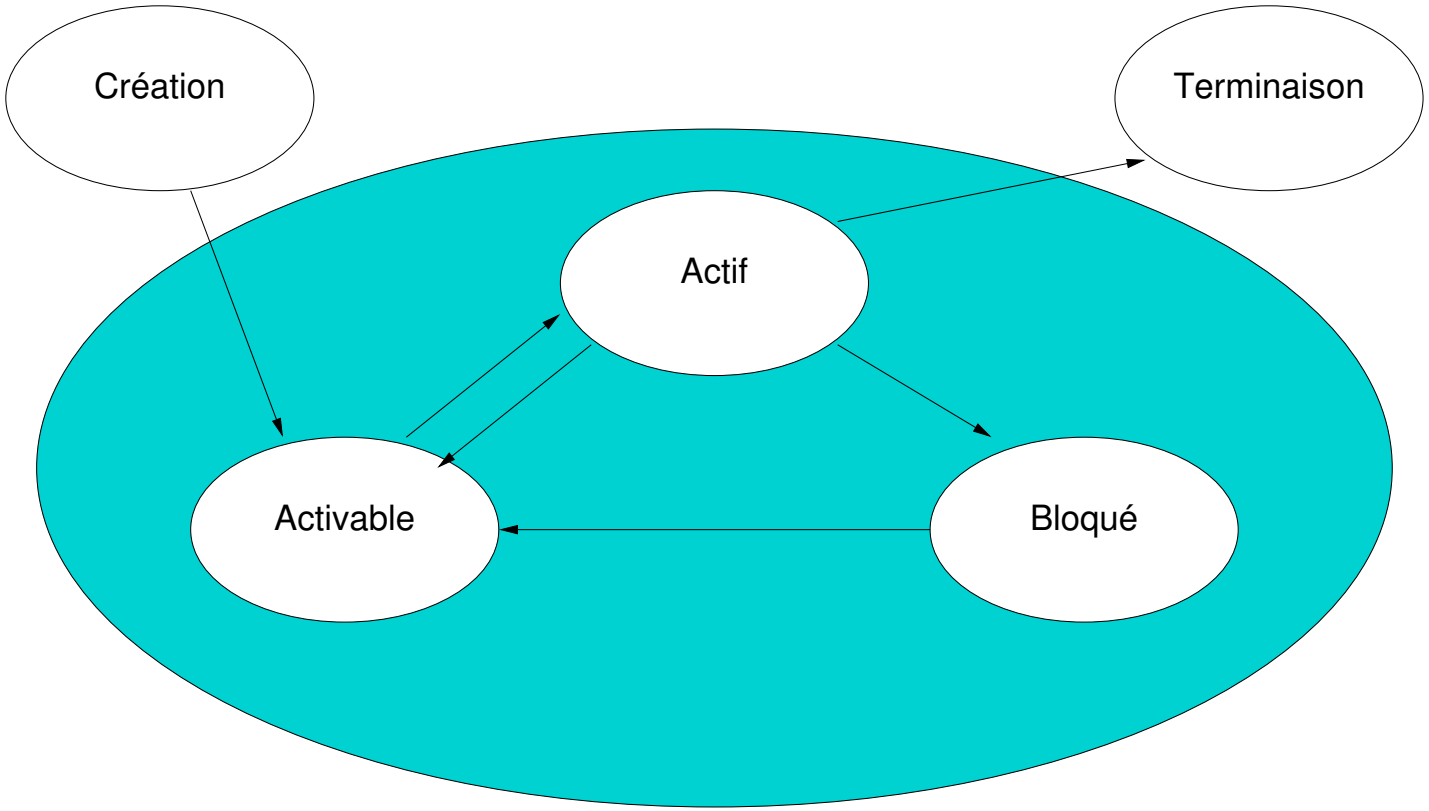
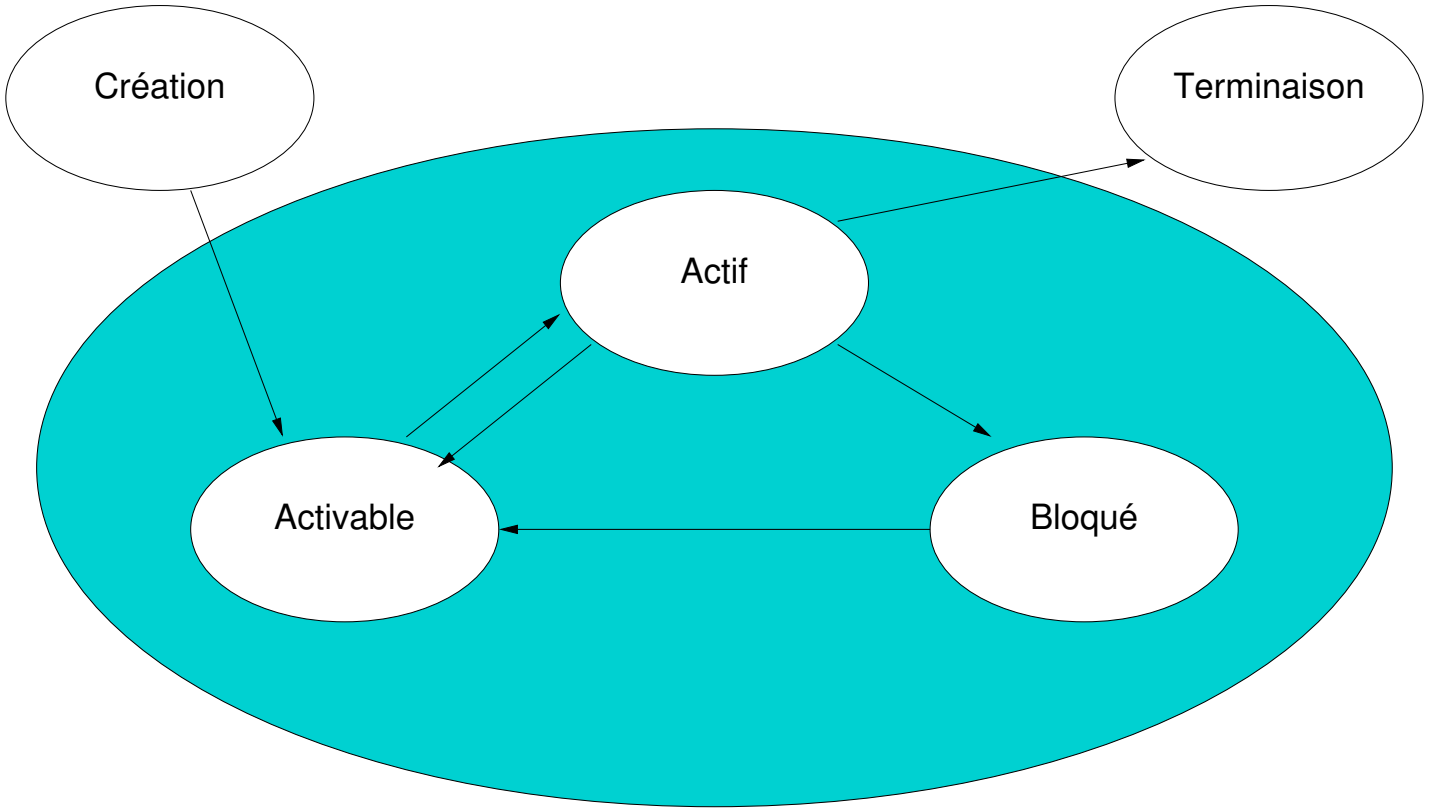


Diagramme d'états



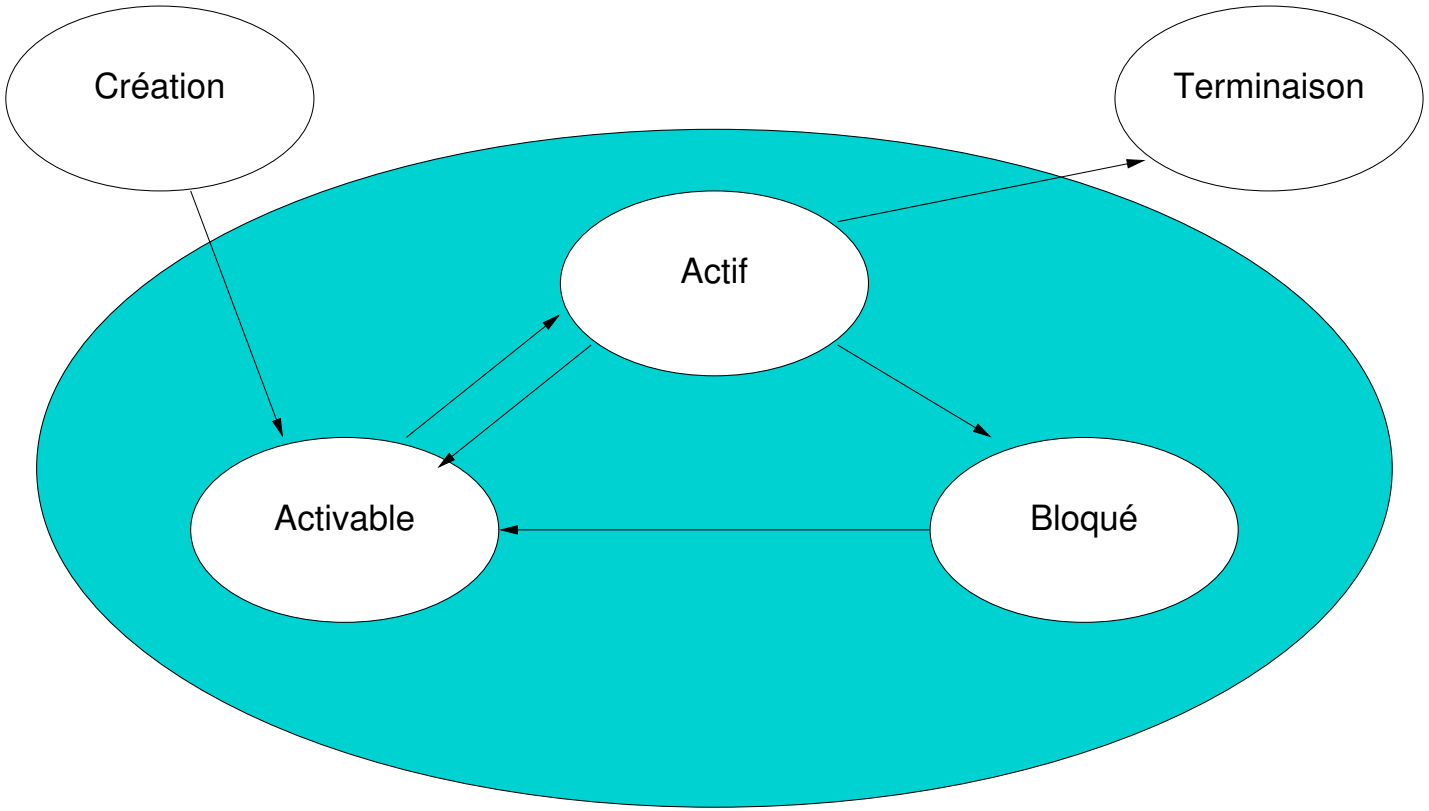
- Activable $\rightarrow$ Actif : attribution d'un processeur

Diagramme d'états



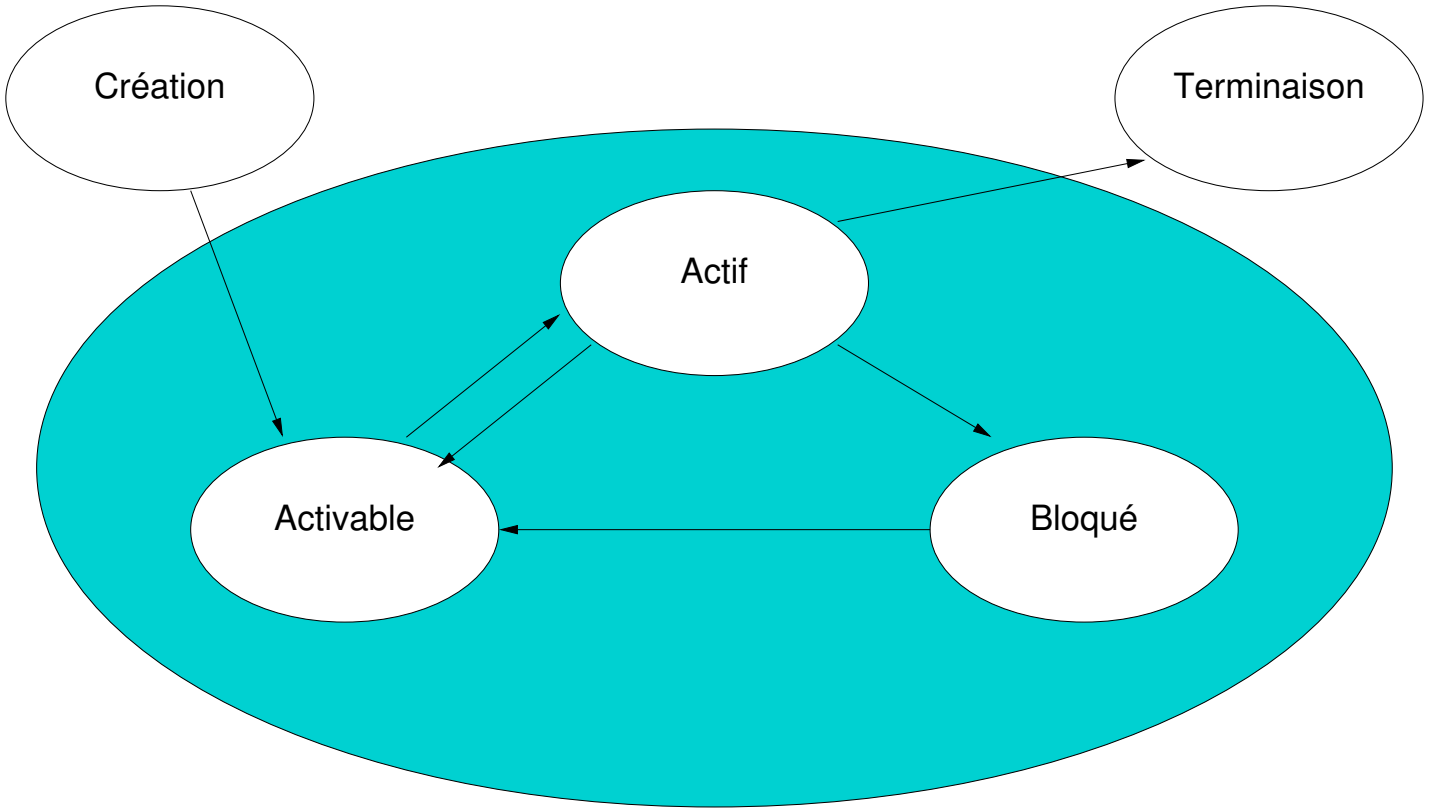
- Actif → Activable : perte du processeur, en temps partagé par exemple

Diagramme d'états



- Actif $\rightarrow$ Bloqué : le processus se met ou est mis en attente d'un événement extérieur
  - Fin d'une opération d'E/S, attente d'un signal, allocation de mémoire, ...

Diagramme d'états



- Bloqué →Activable : l'événement attendu s'est produit

Processus : création

- Dans les systèmes modernes, multi-tâches, la création de processus est dynamique
- Primitive système permettant de créer un processus
- À la création d'un processus le système d'exploitation doit
- Créer un nouveau descripteur
- Trouver les ressources nécessaires à son exécution : mémoire, périphériques,. . ., sauf le processeur
- Insérer le processus dans la liste des tâches à exécuter pour lui attribuer un processeur

Processus : terminaison

- Un processus peut se terminer
  - Il arrive à la fin de son code : terminaison normale
  - Par action du système d'exploitation : il a commis une faute
- Dans tous les cas le système d'exploitation doit :
  - Libérer les ressources qui avait été attribuées
  - Détruire le descripteur du processus

Linux

Arborescence des processus

- Structure arborescente des processus
- Un processus est créé par le noyau lorsqu'il a terminé son initialisation
- Processus 1 : init
- Chaque processus possède :
  - Un identifiant : PID ou *Processus IDentifier*
  - Celui de son père est le PPID ou *Parent PID*

Arborescence des processus

- Structure arborescente des processus
- Un processus est créé par le noyau lorsqu'il a terminé son initialisation
- Processus 1 : init
- Chaque processus possède :
  - Un identifiant : PID ou *Processus IDentifier*
  - Celui de son père est le PPID ou *Parent PID*
  - Un identifiant utilisateur réel : créateur du processus
  - Un identifiant utilisateur effectif : droits ou permissions

Arborescence des processus

- Structure arborescente des processus
- Un processus est créé par le noyau lorsqu'il a terminé son initialisation
- Processus 1 : init
- Chaque processus possède :
  - Un identifiant : PID ou *Processus IDentifier*
  - Celui de son père est le PPID ou *Parent PID*
  - Un identifiant utilisateur réel : créateur du processus
  - Un identifiant utilisateur effectif : droits ou permissions
  - Des identifiants de groupe réel, effectif
- En général, identifiant effectif = réel, mais c'est modifiable par utilisation de primitives ou du mécanisme de prise d'identité

Arborescence des processus

- Structure arborescente des processus
- Un processus est créé par le noyau lorsqu'il a terminé son initialisation
- Processus 1 : init
- Chaque processus possède :
  - Un identifiant : PID ou *Processus IDentifier*
  - Celui de son père est le PPID ou *Parent PID*
  - Un identifiant utilisateur réel : créateur du processus
  - Un identifiant utilisateur effectif : droits ou permissions
  - Des identifiants de groupe réel, effectif
- En général, identifiant effectif = réel, mais c'est modifiable par utilisation de primitives ou du mécanisme de prise d'identité
- Fonctions :
  - `getpid(2)`, `getppid(2)`
  - `getuid(2)`, `geteuid(2)`, `getgid(2)`, `getegid(2)`

Création (fork)

Création de processus : fork

- La création de processus se fait par duplication d'un processus existant
- Primitive :

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

Création de processus : fork

- La création de processus se fait par duplication d'un processus existant
- Primitive :

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

- L'exécution de cette primitive par un processus duplique le processus appelant qui devient le processus père, le processus créé étant le processus fils

Création de processus : fork

- La création de processus se fait par duplication d'un processus existant
- Primitive :

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

- L'exécution de cette primitive par un processus duplique le processus appelant qui devient le processus père, le processus créé étant le processus fils
- L'ensemble des segments hormis le segment de pile sont dupliqués et un nouveau segment de pile est créé

Création de processus : héritage

- Le processus fils hérite de son père la majorité des caractéristiques sauf le PID, le PPID et traitement des signaux
- L'héritage comprends l'ensemble des descripteurs ouverts ce qui permet la redirection des fichiers d'E/S standard

Création de processus : distinction père/fils

- A l'issue du fork il existe deux processus exécutant le même code
- La distinction processus père, processus fils est réalisée par la valeur retournée par fork qui est différente selon le processus
- Père :
  - -1 : indique que la primitive a échoué
  - > 0 : un processus fils a été créé et son PID est la valeur retournée
- Fils : 0

Exemple

```
#include <unistd.h>
#include <stdio.h>
int main(void)
{
  pid_t fils, processus, pere;
  fils = fork();
  switch (fils) {
  case –1 : fprintf(stderr,"Pere : echec du fork\n");
      break;
  case 0 : // processus fils
    processus = getpid(); pere = getppid();
    printf("Processus fils : mon no = %d mon pere = %d\n", processus, pere);
    return 0;
  }
  // processus pere
  processus = getpid(); pere = getppid();
  printf("Processus pere : mon no = %d mon pere = %d\n", processus, pere);
  return 0;
}
```

Terminaison

Terminaison d'un processus

- Deux cas possible : fin anormale et fin normale

Terminaison d'un processus

- Deux cas possible : fin anormale et fin normale
- Dans les deux cas :
  - l'ensemble des ressources allouées au processus sont libérées ;
  - le descripteur du processus est mis à jour avec son code de retour ou *status* ;
  - un signal *SIGCHLD* est émis vers son processus père.

Terminaison d'un processus

- Deux cas possible : fin anormale et fin normale
- Dans les deux cas :
  - l'ensemble des ressources allouées au processus sont libérées ;
  - le descripteur du processus est mis à jour avec son code de retour ou *status* ;
  - un signal *SIGCHLD* est émis vers son processus père.
- Si le processus a des enfants, ceux-ci sont *adoptés* par le processus *init(1)*

  . . . ou par le processus «*subreaper*» le plus proche, tel que défini par l'opération *prctl(PR_SET_CHILD_SUBREAPER)*

Terminaison d'un processus

- Deux cas possible : fin anormale et fin normale
- Dans les deux cas :
  - l'ensemble des ressources allouées au processus sont libérées ;
  - le descripteur du processus est mis à jour avec son code de retour ou *status* ;
  - un signal *SIGCHLD* est émis vers son processus père.
- Si le processus a des enfants, ceux-ci sont *adoptés* par le processus *init(1)*

  . . . ou par le processus «*subreaper*» le plus proche, tel que défini par l'opération *prctl(PR_SET_CHILD_SUBREAPER)*
- Le processus passe à l'état *zombie* jusqu'à l'acquisition de son code de retour par son père

Terminaison anormale

- Lorsque le processus reçoit un signal terminant son exécution, en particulier lorsqu'il tente d'effectuer une opération illégale :
  - accès mémoire à une partie de la mémoire non allouée
  - tentative d'écriture dans un segment en lecture seule
  - ...

Terminaison anormale

- Lorsque le processus reçoit un signal terminant son exécution, en particulier lorsqu'il tente d'effectuer une opération illégale :
  - accès mémoire à une partie de la mémoire non allouée
  - tentative d'écriture dans un segment en lecture seule
  - ...
- Le système arrête alors le processus immédiatement

Terminaison anormale

- Lorsque le processus reçoit un signal terminant son exécution, en particulier lorsqu'il tente d'effectuer une opération illégale :
  - accès mémoire à une partie de la mémoire non allouée
  - tentative d'écriture dans un segment en lecture seule
  - ...
- Le système arrête alors le processus immédiatement
- Le code de retour transmis au processus père indique la nature de la faute (numéro de signal)

Terminaison normale

- Durant son exécution le processus effectue un appel direct ou indirect à la primitive *_exit(2)*

```
#include <unistd.h>
void _exit(int status);
```

où *status* est le code de retour du processus

Terminaison normale

- Durant son exécution le processus effectue un appel direct ou indirect à la primitive *_exit(2)*

```
#include <unistd.h>
void _exit(int status);
```

où *status* est le code de retour du processus

- La fonction de bibliothèque *exit(3)* permet d'exécuter une partie de code associée au processus par la fonction *atexit(3)*

```
#include <stdlib.h>
void exit(int status);
int atexit(void (*function)(void));
```

Terminaison normale

- Durant son exécution le processus effectue un appel direct ou indirect à la primitive *_exit(2)*

```
#include <unistd.h>
void _exit(int status);
```

où *status* est le code de retour du processus

- La fonction de bibliothèque *exit(3)* permet d'exécuter une partie de code associée au processus par la fonction *atexit(3)*

```
#include <stdlib.h>
void exit(int status);
int atexit(void (*function)(void));
```

- Dans un programme C, terminer la fonction *main()* avec l'instruction **return** expression; est équivalent à appeler *exit()* avec la valeur de l'expression en argument.

Attente de la fin d'un processus fils

Attente de la fin d'un processus fils

- Il est souvent utile qu'un processus père ne reprenne son exécution qu'après la fin du processus fils
- Fonctionnement classique du shell lorsqu'une commande est exécutée en avant plan
- Le primitive *wait()* permet de mettre en attente un processus jusqu'à la fin d'un de ces processus fils
- Si un processus fils s'est terminé avant l'appel de wait() le père n'est pas mis en attente
- Synchronisation du père par la fin du fils

Attente de la fin d'un processus fils

- Lorsqu'un processus se termine toutes ses ressources sont libérées seul son descripteur est conservé car il contient son code de retour
- Le processus est dans l'état zombie jusqu'à ce que son père exécute wait() pour obtenir son code de retour
- Code de retour :

fin normale	valeur retournée par la fonction main ou passée en paramètre à la primitive exit()
fin anormale	numéro du signal ayant provoqué la fin du processus

Primitive `wait()`

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

Résultat retourné :

- -1 en cas d'erreur, ou si le processus n'a pas de fils
- le PID d'un processus fils qui s'est terminé
- si *status* n'est pas NULL, alors **status* contiendra le code de retour du processus

Primitive `wait()`

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

Résultat retourné :

- -1 en cas d'erreur, ou si le processus n'a pas de fils
- le PID d'un processus fils qui s'est terminé
- si *status* n'est pas NULL, alors **status* contiendra le code de retour du processus

Autres primitives

```
pid_t waitpid(pid_t pid, int *status, int options);
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

- Voir les pages du manuel en ligne : `man 2 wait`

Code de retour

Les macros suivantes permettent d'examiner la valeur du code de retour :

WIFEXITED(status) vrai si le processus s'est terminé normalement

WEXITSTATUS(status) code de retour du processus (8 bits de poids faible de *status)*

Code de retour

Les macros suivantes permettent d'examiner la valeur du code de retour :

WIFEXITED(status) vrai si le processus s'est terminé normalement

WEXITSTATUS(status) code de retour du processus (8 bits de poids faible de *status)*

WIFSIGNALED(status) vrai si le processus a été terminé par un signal

WTERMSIG(status) numéro du signal ayant terminé le processus

WCOREDUMP(status) vrai si une image mémoire du processus a été créée (fichier core)

Code de retour

Les macros suivantes permettent d'examiner la valeur du code de retour :

WIFEXITED(status) vrai si le processus s'est terminé normalement

WEXITSTATUS(status) code de retour du processus (8 bits de poids faible de *status)*

WIFSIGNALED(status) vrai si le processus a été terminé par un signal

WTERMSIG(status) numéro du signal ayant terminé le processus

WCOREDUMP(status) vrai si une image mémoire du processus a été créée (fichier core)

WIFSTOPPED(status) vrai si le processus a été arrêté par un signal

WSTOPSIG(status) numéro du signal ayant causé l'arrêt

WIFCONTINUED(status) vrai si le processus a été relancé par SIGCONT

Exemple `wait()` (1/3)

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

int main(void)
{
    pid_t fils, processus, pere;
    int i, status;
    i = 1;
    fils = fork();
    switch (fils) {
    case -1:
        fprintf(stderr,"Pere : echec du fork\n");
        return EXIT_FAILURE;
        break;
        /*... */
```

Exemple `wait()` (2/3)

```
    /*... */
case 0 : //processus fils
    processus = getpid();
    pere = getppid();
    printf("Processus fils : mon no = %d mon pere = %d\n", processus, pere);
    printf("Processus fils : i = %d\n", i);
    i = 10;
    printf("Processus fils : i = %d\n", i);
    return 0;
} // switch

/* ... */
```

Exemple `wait()` (3/3)

```
    /* ... */
    // Processus pere
    processus = getpid();
    pere = getppid();
    // Attente du fils
    fils = wait (&status);
    printf("Processus pere : mon no = %d mon pere = %d\n", processus, pere);
    printf("Processus pere : i = %d\n", i);
    printf("Processus pere : mon fils %d s'est terminé \n", fils);
    if (WIFEXITED(status))
        printf("Processus pere : mon fils s'est terminé normalement"\
              " avec le code %d\n", WEXITSTATUS(status));
    if (WIFSIGNALED(status))
        printf("Processus pere : mon fils s'est terminé anormalement"\
              " avec le signal %d\n", WTERMSIG(status));
    return EXIT_SUCCESS;
}
```

Exemple `wait()` : exécution

xterm

```
$ ./exemple_wait
Processus fils : mon no = 3954122 mon pere = 3954121
Processus fils : i = 1
Processus fils : i = 10
Processus pere : mon no = 3954121 mon pere = 3954113
Processus pere : i = 1
Processus pere : mon fils 3954122 s'est terminé
Processus pere : mon fils s'est terminé normalement avec le code 0
$
```

Recouvrement (exec)

Recouvrement de processus par le chargement d'un fichier exécutable

- *fork* : duplique un processus existant mais ne permet pas de créer un nouveau processus exécutant un nouveau segment de code
- Les primitives de la famille *exec* permettent de remplacer les segments de code, données et pile par le chargement en mémoire d'un fichier binaire
- Le segment système n'est pas modifié l'héritage du fork est donc préservé
- Les descripteurs de fichiers, sauf ceux ouverts avec O_CLOEXEC, sont toujours valides permettant les redirections

La primitive de recouvrement : `execve()`

- C'est la seule primitive (fonction exécutée par le système d'exploitation) permettant de charger un mémoire un fichier binaire
- Prototype :

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

- Cette primitive lorsqu'elle est exécutée par un processus provoque la modification des segments de code, données, pile par le contenu d'un fichier binaire

execve()

- Paramètres

  pathname chemin d'accès du fichier binaire

  argv tableau de chaînes de caractères qui seront transmises comme argument (le tableau doit être terminé par un pointeur NULL)

  envp tableau de chaînes de caractères de la forme `"NOM=valeur"`, constituant l'environnement du processus (le tableau doit être terminé par un pointeur NULL)

execve()

- Paramètres
  - pathname : chemin d'accès du fichier binaire
  - argv : tableau de chaînes de caractères qui seront transmises comme argument (le tableau doit être terminé par un pointeur NULL)
  - envp : tableau de chaînes de caractères de la forme `"NOM=valeur"`, constituant l'environnement du processus (le tableau doit être terminé par un pointeur NULL)
- Résultat retourné
  - En cas de succès cette primitive ne revient pas car le segment de code initial est détruit
  - En cas d'échec execve retourne -1 et errno contient un code d'erreur indiquant les raisons

`execve()`, remarques

- La variable PATH est une variable du SHELL, elle ne peut et n'est pas utilisée pour trouver le fichier binaire dans l'arborescence il est donc obligatoire d'utiliser un chemin d'accès pour le fichier
- Ce fichier doit être exécutable pour le propriétaire effectif du processus (droit x)
- Si ce fichier binaire possède un des bits de prise d'identité du propriétaire (*setuid*) ou du groupe (*setgid*) positionné les identificateurs de propriétaire ou de groupe effectifs deviennent le propriétaire ou le groupe du fichier

Caractéristiques du processus résultant

Tous les attributs du processus initial sont réservés, sauf :

- Les signaux pour lesquels le processus avait placé un gestionnaire sont réinitialisés à leur valeur par défaut (consultez signal(7))
- L'éventuelle pile spécifique pour les gestionnaires de signaux n'est pas conservée (sigaltstack(2))
- Les projections en mémoire ne sont pas conservées (mmap(2))
- Les segments de mémoire partagée System V sont détachés (shmat(2))
- Les objets de mémoire partagée POSIX sont supprimés (shm_open(3))
- Les descripteurs de files de messages POSIX ouverts sont fermés (mq_overview(7))

Caractéristiques du processus résultant

- Les sémaphores nommés POSIX ouverts sont fermés (sem_overview(7))
- Les temporisations POSIX ne sont pas conservées (timer_create(2))
- Les flux de répertoires ouverts sont fermés (opendir(3))
- Les verrouillages de mémoire ne sont pas préservés (mlock(2), mlockall(2))
- Les gestionnaires de terminaison ne sont pas préservés (atexit(3), on_exit(3))
- L'environnement de travail en virgule flottante est remis à zéro (consultez fenv(3))

Exemple : exécution de `ls`

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
    char *arg[4];
    char *env[] = {NULL};
    if (argc != 2) {
        fprintf(stderr, "Usage: %s répertoire\n", argv[0]);
        return EXIT_FAILURE;
    }
    arg[0] = "ls"; /* création du tableau d'arguments */
    arg[1] = "-l";
    arg[2] = argv[1];
    arg[3] = NULL;
    execve("/bin/ls", arg, env); /* exécution de /bin/ls */
    perror("execve"); /* execve() ne retourne qu'en cas d'erreur */
    return EXIT_FAILURE;
}
```

Les fonctions de bibliothèque de recouvrement

- Elles sont au nombre de 6 que l'on peut séparer en deux groupes
  - Les fonctions `execl...` admettant en paramètres une liste de paramètres de type `char*` terminées par NULL
  - Les fonctions `execv...` admettant en paramètres un tableau de pointeur sur des caractères
- Lorsqu'elles se terminent par p la variable PATH de l'environnement est utilisée pour trouver le fichier exécutable
- Lorsqu'elles se terminent par e un environnement est transmis

Les fonctions de bibliothèque de recouvrement

```
#include <unistd.h>

int execl(const char *pathname, const char *arg, ...
                  /*, (char *) NULL */);
int execlp(const char *file, const char *arg, ...
                  /*, (char *) NULL */);
int execle(const char *pathname, const char *arg, ...
                  /*, (char *) NULL, char *const envp[] */);

int execv(const char *pathname, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

Les fonctions de bibliothèque de recouvrement

- Comme execve toutes ces fonctions ne retournent pas en cas de succès puisque le segment de code original est remplacé par le segment de code issu du chargement du fichier binaire
- Elles retournent -1 en cas d'erreur
- errno contient le code d'erreur
- perror() peut donc être appelé pour afficher le message d'erreur système

Exemple : `execlp()`

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

int main (int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "Il manque le nom du fichier\n");
        return EXIT_FAILURE;
    }
    else {
        execlp("wc", "wc", "-c", "-m", "-l", "-w", argv[1], (char*)NULL);
        fprintf(stderr, "Echec de la primitive execlp\n");
        return EXIT_SUCCESS ;
    }
}
```

Exemple : `execv()`

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
    char *arg[4];
    if (argc != 2) {
        fprintf(stderr, "Usage: %s répertoire\n", argv[0]);
        return EXIT_FAILURE;
    }
    arg[0] = "ls"; /* création du tableau d'arguments */
    arg[1] = "-l";
    arg[2] = argv[1];
    arg[3] = NULL;
    execv("/bin/ls", arg); /* exécution de /bin/ls */
    perror("execv"); /* execv() ne retourne qu'en cas d'erreur */
    return EXIT_FAILURE;
}
```