Systèmes avancées

Wieslaw Zielonka
zielonka@irif.fr

Synchronisation avec la terminaison d'un fils : wait() et waitpid()

Terminaison du processus

Comment le processus peut terminer ?

Processus à l'état zombie :

ps axl : pour voir l'état de tous les processus (Z - zombie)

Pourquoi le système laisse un processus à l'état zombie au lieu de le supprimer complètement après la terminaison ?

Est-ce que le processus à l'état zombie consume les resources de systèmes (fichiers ouverts? mémoire?, temps du processeur ?)

synchronisation avec la terminaison d'un processus fils

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

Le processus père peut synchroniser avec la terminaison d'un processus fils en exécutant `wait()`.

- Si le processus père possède des fils vivants et aucun fils n'a terminé depuis le dernier `wait()` le processus père reste bloqué sur `wait()` jusqu'à la terminaison d'un fils.
- Si *`p_status`* n'est pas NULL alors l'information sur la terminaison du fils est mise à l'adresse donnée par *`p_status`*.
- A la terminaison d'un fils le noyau ajoute les statistiques sur l'utilisation de ressources et de temps du processeurs aux statistiques concernant l'ensemble de fils.
- `wait()` retourne le pid du fils qui a terminé, ce fils est définitivement supprimé.
- wait() retourne -1 en cas d'erreur.
- Notez que wait() peut terminer à la réception d'un signal.

Une des erreurs possibles de wait() : errno == ECHILD si le processus n'a pas de fils.

attendre la terminaison d'un processus fils

```
/* attente de terminaison de tous les
enfants*/
pid_t pidEnfant;
while( (pidEnfant = wait(NULL) ) != -1 )
    continue;
  if( errno != ECHILD ){
    //traiter erreur non prévue
    exit(1);
  }
```

Processus père termine laissant les enfants orphelins

Quand le processus père termine, tous ses fils (vivants ou zombies) acquièrent un nouveau père : le processus *init* dont le pid est 1.

Le processus *init* supprime automatiquement tous ses fils zombies.

comment éviter d'avoir les enfants zombies

Astuce de double fork() :

- faire *`fork()`*
- le processus père attend la terminaison du fils (attente courte puisque le fils termine presque tout de suite),
- le processus fils fait *`fork()`* à son tour, crée un petit fils, et fait tout de suite *`exit(0)`* pour terminer.

- le petit fils, qui devient orphelin, est adopté par le processus *`init`* (processus 1), le petit fils exécute la tâche. A la terminaison du petit fils c'est *`init`* qui se chargera de sa suppression.

faiblesses de wait

- Impossible de spécifier le fils dont la terminaison on attend.
- Attente bloquante.
- Avec wait() on trouve les fils qui ont terminé mais pas les fils arrêtés par un signal (SIGSTOP) ou les fils qui reprennent l'activité après l'arrêt en recevant le SIGCONT.

waitpid()

```
#include <sys/wait.h>
pid_t waitpid( pid_t pid, int *p_status,
               int options)
```

pid : −1 attendre la terminaison d'un fils quelconque

>0 attendre la terminaison du fils avec le pid donné

p_status : comme dans wait()

options : soit 0 soit WNOHANG. Si WNOHANG est spécifié alors sans attente, s'il n'y a pas de fils qui a déjà terminé waitpid() retourne tout de suite 0.

D'autres valeurs de paramètres pid et options sont possibles.

l'utilisation du paramètre p_status de wait() et waitpid()

```
int status;
pid_t pidEnfant;
while( ( pidEnfant = wait( &status ) ) == -1
&& errno !=  ECHILD )
   continue;
```

après cette boucle la variable **`status`** contient les information sur la terminaison de fils dont le pid est dans pidEnfants.

l'utilisation de wait()

Les macro-fonctions pour examiner l'état de terminaison de l'enfant :

`WIFEXITED(status)`	VRAI si l'enfant a terminé par exit(), _exit() ou _Exit(). Dans ce cas nous pouvons obtenir le 8 bits de poids faible de valeur exit grâce à `WEXITSTATUS(status)`
`WIFSIGNALED(status)`	VRAI si l'enfant a terminé à cause d'un signal. Dans ce cas `WTERMSIG(status)` permet de récupérer le numéro du signal qui a provoqué la terminaison du processus fils.

l'utilisation de wait()

```
int status;
pid_t pidEnfant;
pidEnfant = wait( &status );
if( pidEnfant  == -1 ){
   if( errno == ECHILD ){
       // il n'y a plus d'enfant
    }else{
      // traiter l'erreur de wait()
}
else if( WIFEXITED( status ) ){
   /* enfant a terminé par exit() */
   printf("fils %ld termine par exit %d\n",
          (long) pid,
          WEXITSTATUS( status ) );
   .....
}else if( WIFSIGNALED( status ) ){
   /* enfant  a terminé à cause d'un signal */
      printf("fils %ld termine par signal %d\n",
          (long) pid,
          WTERMSIG( status ) );
   ....
}
```

comment se débarrasser de fils zombies ?

Comment se débarrasser de fils zombies sans être bloqué sur `wait()`?

Utiliser un compteur `nb_fils` de fils incrémenté à chaque `fork()`.

Faire exécuter de temps en temps :

```
if( nb_fils > 0 && waitpid(-1, NULL, WNOHANG) > 0 )
   nb_fils--;
```

recouvrement de processus la famille exec

chargement d'un exécutable

Recouvrement de processus permet de démarrer l'exécution d'un nouveau programme par le processus. Le code exécuté par le processus est remplacé par le code d'un autre programme.

C'est le processus lui-même qui doit indiquer quel programme il veut exécuter à la place du programme courant.

Les étapes :

1. le remplacement du code du processus par le nouveau programme à exécuter
2. l'initialisation des arguments de main() du nouveau programme
3. l'initialisation de variables d'environnement du nouveau programme
4. l'appel à la fonction main() du nouveau programme
5. la pile, le tas, le code de l'ancien programme sont supprimés

la famille de fonctions exec...()

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

l'appel système pour exec. Toutes les autres formes d'exec sont dérivées de celle-ci.

path - le chemin vers un exécutable. path peut être un chemin absolu (commence par /) ou relatif par rapport au répertoire courant du processus.

argv[0] contient le nom de la commande (de l'exécutable).

la famille de fonctions exec...()

```
#include <unistd.h>
int execl( const char *path, const char *arg0, ... /* (char *) 0 */ )
int execv(  const char *path, char *const argv[ ])
int execle( const char*path, const char *argv0, ...
        /* (char *) 0 , char *const envp[ ]  */ )
int execve( const char *path, char *const argv[ ],
                    char *const envp[ ] )
int execlp( const char *path, const char *argv0, ...
       /* (char *) 0 */ )
int execvp(  const char *path, char *const argv[ ])
```

v - vecteur : les paramètres de main dans un vecteur

l - liste : les paramètres de main dans un liste

la famille de fonctions exec...()

```
#include <unistd.h>
int execl(const char *path, const char *arg0, ... /* (char *) 0 */ )
int execv(  const char *path, char *const argv[ ])
int execlp( const char *path, const char *argv0, ...
      /* (char *) 0 */ )
int execvp(  const char *path, char *const argv[ ])
```

l - (list) si les arguments de main() sous la forme d'une liste

v - (vector) si les arguments de main() sous la forme d'un vecteur

p - l'exécutable recherché dans les répertoires spécifiés dans la variable PATH, sans p il faut donner le chemin absolu complet

exemples

le processus courant doit exécuter la commande

`ls -l -a`

```
char *argv[] = { "ls" , "-l", "-a", (char *) 0 };
```

```
execvp( "ls", argv);
```

ls recherché dans un des répertoires de la variable d'environnement PATH.

Il n'y a jamais de retour d'un appel réussi d'une fonction de la famille exec.

exemples

le processus courant doit exécuter la commande

`ls -l -a`

```
char *argv[] = { "ls" , "-l", "-a", (char *) 0 };
```

```
execv( "/bin/ls", argv);
```

(char *) 0 est toujours le dernier élément du vecteur argv

`sans p :` faut donner le chemin absolu /bin/ls vers `ls`

exemples

le processus courant doit exécuter la commande

`ls -l -a`

```
execlp( "ls", "ls" , "-l", "-a", (char *) 0);
```

- Pourquoi `"ls"` apparaît deux fois?
- Pourquoi on ne vérifie jamais la valeur de retour de `exec` pour voir si `exec` a réussi ou non ?

exemples

Exécuter le programme

`monprog -a -f fic`

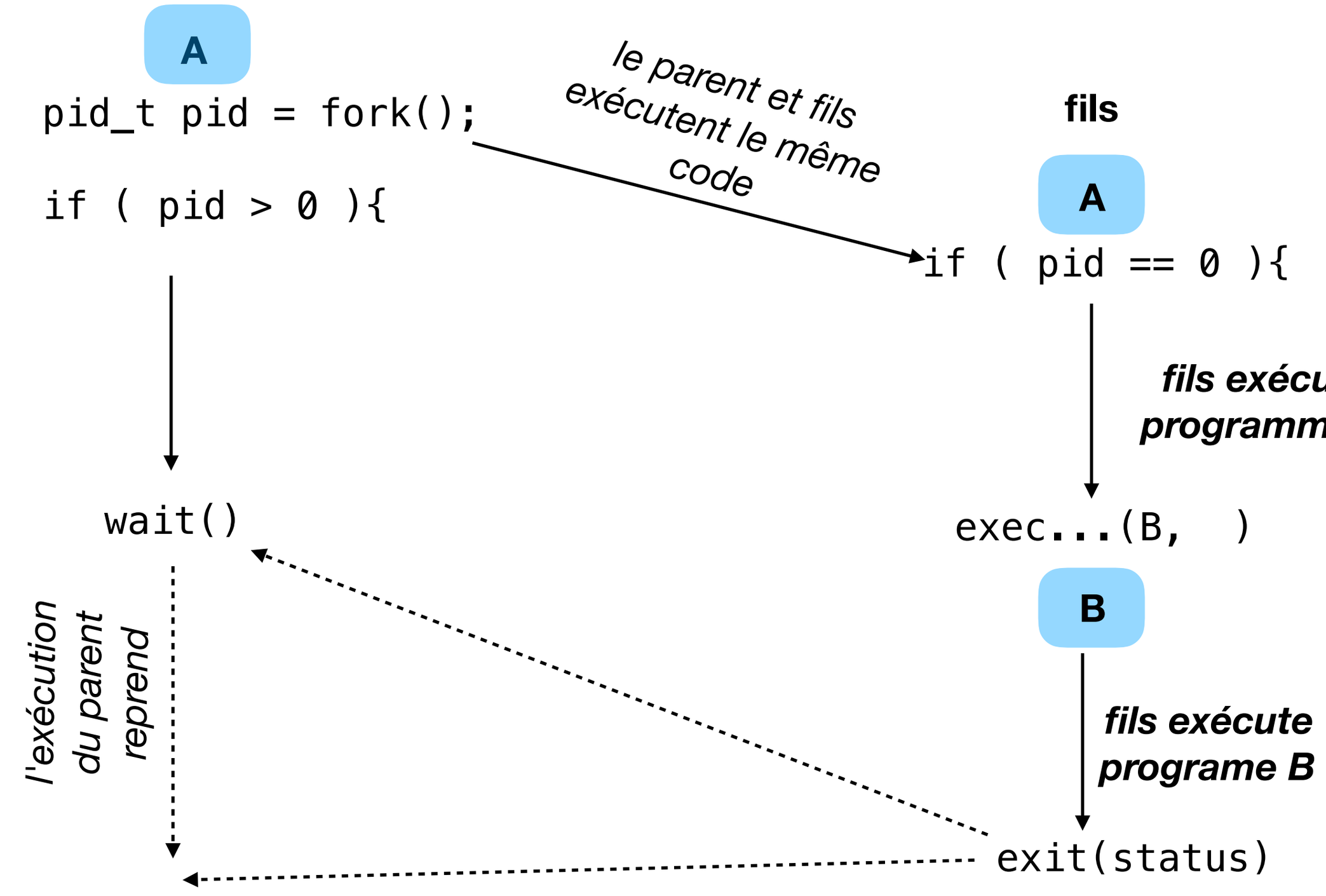
qui se trouve dans le répertoire $HOME/bin:

```
#define SIZE 1024
char chemin[SIZE] ;
/* préparer le chemin $HOME/bin/monprog */
snprintf( chemin, SIZE, "%s/bin/monprog",
getenv("HOME"));
execl( chemin, "monprog", "-a", "-f" , "fic",
        (char *) 0);
/* traiter erreur de exec */
```

argv[0] argv[1] argv[2] argv[3]

fork() --> exec()

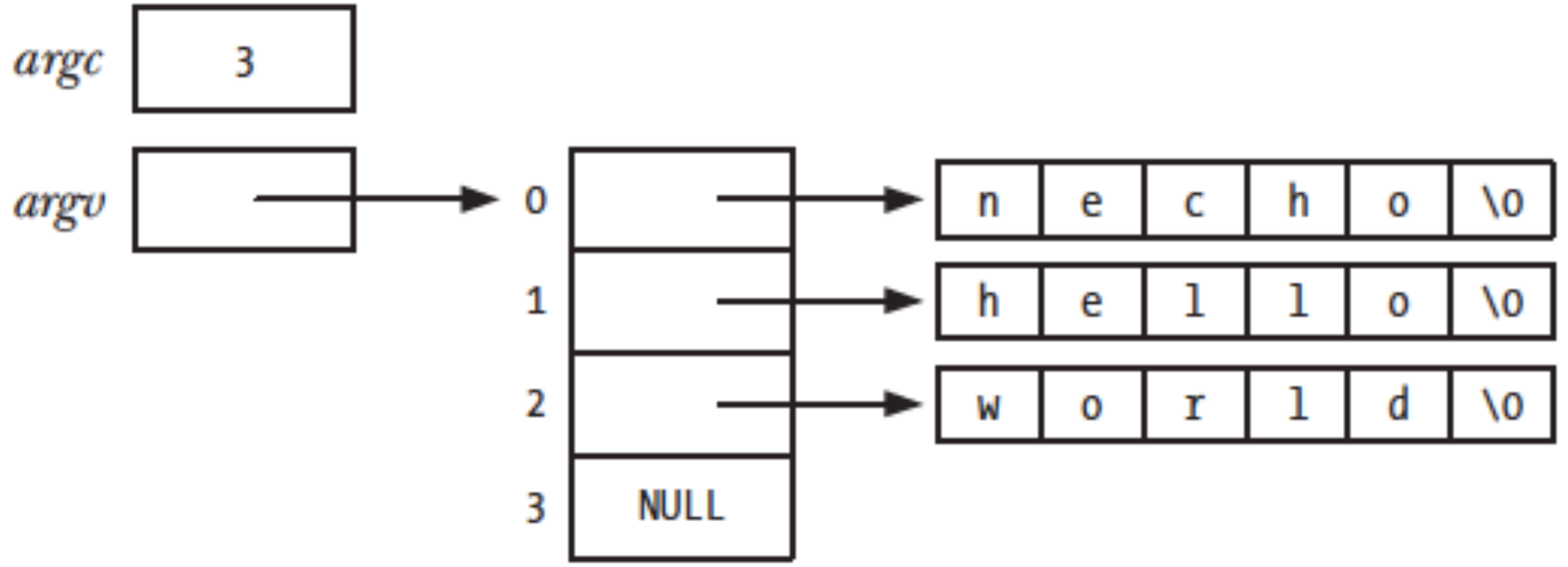
père exécute programme A



le parent reprend l'exécution

les paramètres de main

```
int main( int arc, char **argv, char **environ)
```



les variables d'environnement

`printenv` *affiche la liste de variables d'environnement avec leur valeurs.*

SHELL=/bin/bash *créer une variable locale SHELL*

export SHELL *la variable SHELL devient globale donc héritée par les enfants*

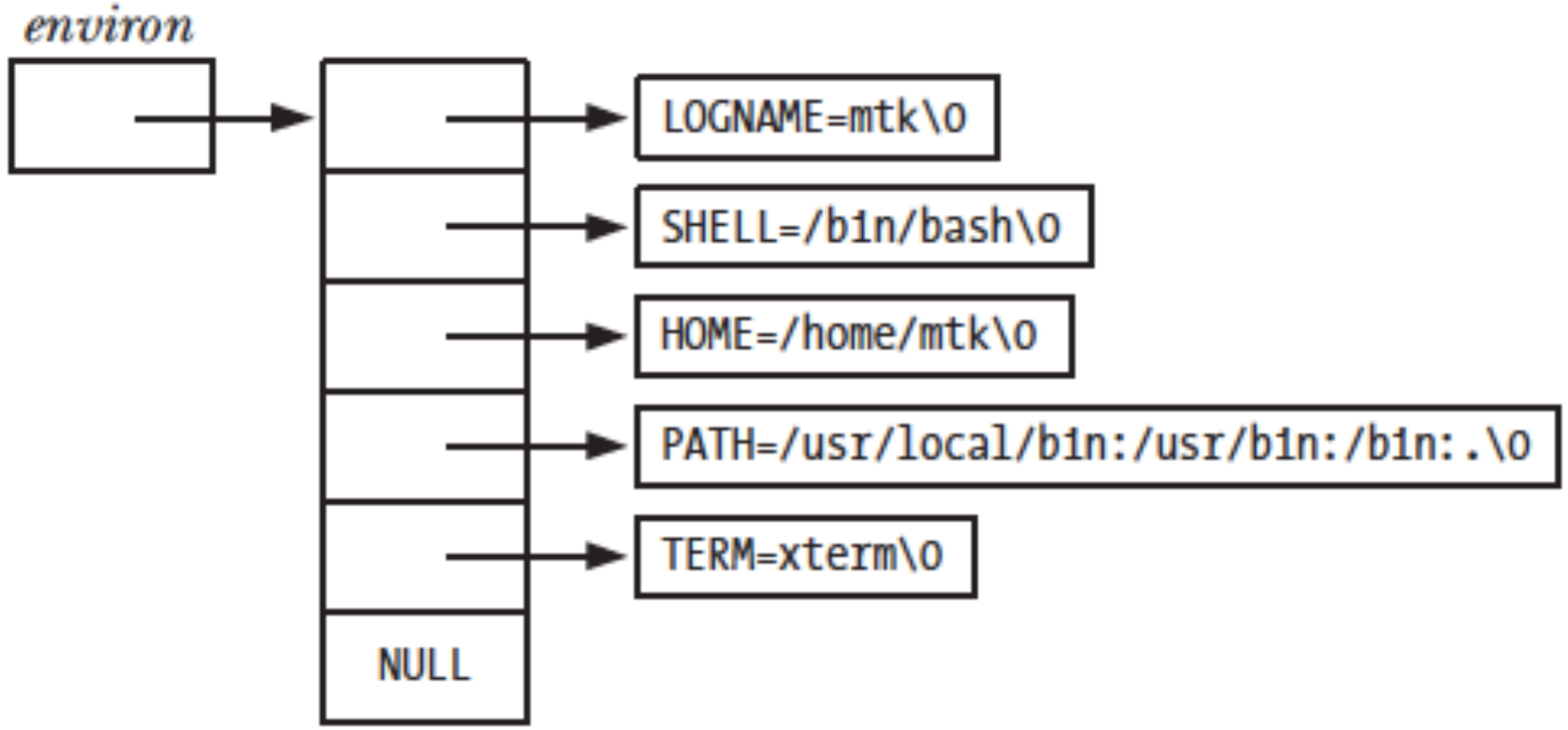
echo $SHELL *afficher la valeur de la variable SHELL*

`NOM=valeur ./programme` *ajoute la variable environnement* NOM *sur la liste de variables d'environnement du nouveau processus qui exécute* `./programme`

les variables d'environnement

```
int main(int arc, char **argv, char **environ)
```

Le troisième paramètre de main() – le tableau de variables d'environnement avec les valeurs



les variables d'environnement

```
#include <stdlib.h>

char *getenv(const char *nom)
```

RETOURNE : la valeur de la variable d'environnement nom ou NULL si la variable n'existe pas. (Vous ne devez pas modifier la chaîne retournée par `getenv()`)

Pour ajouter ou modifier une variable d'environnement:

```
putenv("USER=Britta");
```

Les descripteurs de fichiers ouverts après fork() et exec()

`fork()` le tableau de descripteur de fichier du processus fils est la copie de tableau de descripteurs du processus père

`exec()` si on ne fait rien alors `exec` ne change rien dans le tableau de descripteurs, les descripteurs ouverts restent ouverts.

Si on ne veut pas qu'un descripteur soit ouvert après exec changer la valeur de flag FD_CLOEXEC:

```
int flags;
flags = fcntl(fd, F_GETFD);  // fd descripteur ouvert
if( flags ==  -1 ){ /* traiter erreur */ exit(1); }
flags |= FD_CLOEXEC; /*ajouter un nouveau flag*/ l
if( fcntl( fd, F_SETFD, flags) == -1 ){
    /*traiter erreur*/ exit(1);
}
```

Propriétaire de processus

A quoi sert le propriétaire de processus?

Apres fork(), qui est le propriétaire du processus fils?

Après exec(), est-ce que le propriétaire du processus change?

Propriétaire de processus

A quoi sert le propriétaire de processus?

Pour déterminer les droits d'accès du processus aux fichiers.

Apres fork() qui est le propriétaire du processus fils?

Le processus enfant hérite le propriétaire de son père.

Après exec est-ce que le propriétaire du processus change?

Le processus dont le propriétaire est **michel** fait **exec** pour exécuter **/bin/ls** dont le propriétaire est **root.** /bin/ls exécuté avec quel droit? De michel ou de root?

Le processus dont le propriétaire est michel fait exec pour exécuter /usr/bin/passwd dont le propriétaire est root. /usr/bin/passwd exécuté avec quel droit? De michel ou de root?

les bits Set-user-ID et Set-group-ID