S3-Programmation Système

IUT informatique Bordeaux

Published with GitBook

Table of Contents

S3-Programmation Système

Vous trouverez dans les pages qui suivent l'ensemble du cours de programmation système du S3.

Les bases de processus et les signaux

Quelques rappels

Swapper : processus initial, pid = 0 pid : numéro unique identifiant un processus, reçus lors de la création

- getpid() : accéder au pid d'un processus
- getppid() : accéder au pid du père du processus
- ps -aux (ps -axu), ps -el : affiche la table des processus

Interagir avec le Système:

Il est possible d'appeler une commande système en c en usant de la méthode int system(const char* command) qui renvoie 0 si la commande a été exécutée correctement et -1 sinon.

Le programme suivant lance la commande système ps:

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

int main()
{
  system("ps -l");
  exit(0);
}
```

Signaux

Les signaux permettent de synchroniser des processus sous UNIX

A propos

En nombre finis : 32 avec Linux Le signal véhicule une information : l'identité (aka le numéro) du signal Si un processus reçoit un signal, cela provoque une interruption dans le programme. Le programme s'arrête alors, puis reprendra ensuite à l'endroit où il c'est arrêté. kill -l permet d'afficher la liste de tous les signaux disponible sur le système.

Exemple de signaux visant à ‘terminer’ un processus : SIGHUP : lors de la fermeture du terminal, ce signa est envoyé à tous les processus lancés/excutés par ce même terminal SIGINT : c’est une interruption. Elle peut être généré au clavier via Ctrl-C SIGQUIT : c’est également une interruption, obtenue par Ctrl-. Son but est de générer un fichier core SIGTERM : terminaison. Peut être réalisé par la commande kill (kill pid) SIGKILL : idem que SIGTERM (kill -9 pid). Ce signal ne peux pas être incercepté par le processus.

Exemple de signaux visant à stopper/reprendre un processus : SIGSTOP : stopper un processus SIGTSTP : stopper un processus, généré par Ctrl-Z → Redemmarage du processus par >fg ou >bg SIGCONT : reprise du processus

Générer un signal

Il est possible d’envoyer un signal donné à un programme spécifique en usant de la méthode système kill présente dans signal.h Ainsi le programme suivant :

```
#include<stdio.h>
#include<sys/types.h>
#include<signal.h>

int main(int argc, char **argv)
{
    if (argc < 2)
    {
        printf("usage: ./kill PID");
        return -1;
    }
    kill(atoi(argv[1]), SIGKILL);
    return 0;
}
```

Permet d'arrêter le programme possédant l’id passé en paramètre en lui envoyant le signal SIGKILL (voir plus haut).

Réception et gestion d’un signal

Chaque signal possède sa propre signification, qui est associé par le système à un comportement prédéfinie. Cependant, il est parfaitement possible de modifier ce comportement par défaut en créant une fonction de réception de signal.

On définie une telle fonction à l’aide de la méthode système signal comme suit:

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

```
void catch_function(int signo) {
    puts("Interactive attention signal caught.");
}

int main(void) {
       signal(SIGINT, catch_function);
    ...
}
```

Ainsi à chaque fois que ce programme recevra le signal SIGINT, il appellera la méthode catch_function, qu'on appelle ici handler. Un handler de signal doit toujours prendre la forme suivante:

void nom(int).

Ce handler vient modifier la table de traitement des signaux qui correspond à un tableau associant un signal à une fonction pour le processus, celle-ci est créer lors du lancement du processus.

Cas particulier: effectuer une action à intervalle régulier

Il est possible d'envoyer un signal SIGALARM à un programme à l'aide de la fonction système alarm(int délai). Cette méthode enverra de manière régulière tout les "délai" secondes le système SIGALARM au programme, ce qui, associé à un bon usage de la méthode signal, permet d'effectuer une action à intervalle régulier.

Le programme suivant exécute sauve toutes les secondes:

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#define DELAI 1

void initialise(), calcule(), sauve(), onalrm(int);
 unsigned long i;
 main ()
{
 initialise();
signal(SIGALRM, onalrm);
alarm(DELAI);
calcule();
fprintf(stdout,"calcul termin´e\n");
 exit(0);
}

void onalrm (int k)
{
sauve();
 signal(SIGALRM,onalrm);
alarm(DELAI);
}
```

```
void initialise ()
{
 i=0;
}

void sauve()
{
fprintf(stderr,"sauvegarde de i : %lu\n",i);
}

void calcule()
{
while (i += 2);
}
```

Processus sous UNIX

Fork() :

Fork permet de dupliquer un processus de façon dynamique, en créant un nouveau processus, identique en tout point au processus père.

Le processus fils hérite du processus père il possède donc:

- Le même code
- Une copie de la zone de donnée
- Une copie de l'environnement
- Les différent propriétaires
- Une copie de la table des descripteurs de fichiers
- Une copie de la table de traitement des signaux

La seule chose qui distingue un processus père d'un processus fils après un fork(), c'est leur PID. De plus, le processus fils retourne 0 en valeur de retour, alors que pour le processus père, c'est le pid du fils. Les deux processus s'éxécutent en parallèle.

En cas de soucis lors de la création du fils, fork() retourne -1.

Wait(int* status) :

Wait provoque la suspension (aka la mise en attente) du processus père jusqu'à ce que l'un des processus fils se termine.

Cela permet également à un processus d'attendre un évènement.

La fonction wait(int* status) retourne le pid du fils qui c'est terminé, et en cas d'erreur, retourne -1

Il est possible de passer une adresse d'entier à wait en paramètre, dans ce cas, la variable transmise contiendra le code de retour donné par le processus fils lors de sa mort.

Processus Zombie :

Lorsqu'un processus fils est terminé, il deviens un processus zombie jusq'à ce que :

- Il soit rattrapé par un wait dans le processus père
- Que le processus père meurt

Les processus zombies peuvent empêcher la création de nouveaux processus : Importance du WAIT !!

exec(l,v) : // Duplication et recouvrement

Permet de lancer l'éxécution d'un code, qui recouvrera alors l'ancien. L correspond ici au chemin de l'application à lancer, tandis que v est une liste de paramètre qui doit être terminée par (char*)0.

Le processus garde les mêmes caractéristiques (aka le même contexte) :

- Même pid
- Même père
- Même priorité
- Même propriétaire
- Même répertoire de travail
- Même descripteurs de fichiers ouverts

Pipes et descripteurs de fichiers

La table des descripteurs:

La table des descripteurs est un tableau d'entier comportant des informations permettant au programme c d'accéder à certains flux de données que sont l'entrée/sortie standard, d'erreur, des fichiers ou d'autres éléments en mémoire (pipes).

Primairement la table des descripteurs prends cette forme:

Nombre entier	description
0	Entrée standard
1	Sortie standard
2	Sortie d'erreur

Notez qu'il est possible de modifier cette table. (voir plus bas).

Pipes

Pipe() crée un tube de communication pour permettre à des processus affiliés de communiquer entre eux.

```
#include <unistd.h>
  int pipe(int fd[2])
```

Cet appel système va permettre de placer dans le tableau fd (voir ci-dessus) deux descripteurs de fichier qui permettront à un couple de processus de communiquer entre eux. Ces deux descripteurs de fichier viennent prendre place dans la table des descripteurs de fichiers du processus.

La valeur de retour d'un appel réussis est 0, et -1 en cas d'échec.

D'une manière générale on fait usage de ces descripteurs de la manière suivante:
Le descripteur fd[1] permet d'écrire des données qui seront transmises à un ou plusieurs processus, qui seront en mesure de les lire grâce à fd[0].

Ainsi on a:

- fd[1]: écriture
- fd[0]: lecture

Il est a noter que les processus voulant communiquer entre eux, doivent tous avoir ces descripteurs de fichier en mémoire.

L'écriture dans le pipe est réalisée à l'aide de la fonction void write(int fd[1], char c, sizeof(c)). Celle-ci écrira le contenu de la chaine de caractèrs c dans le tube fd[1].

La lecture du pipe se fait à l'aide de la fonction read(int fd[0]; char &c, sizeof(c)). Celle-ci lira le contenu du pipre fd[0] et le placera dans la chaine de caractères c.

Le programme suivant permet à un processus fils de communiquer avec son père:

```
main()
  {
    int fd[2];
    char p,c;
    pipe(fd);
    if(fork()) { /* processus p`ere */ ...
      write(fd[1], &c, 1);
    }
    else { /* processus fils */ ...
      read(fd[0], &p, 1); …
   }
  }
```

Ces deux fonctions sont bloquantes, c'est à dire que:

- La fonction read se bloque tant qu'aucune donnée n'est présente dans le tube et attends.
- La fonction write se bloque tant que le tube est plein, le processus se bloque en attendant que les tube est suffisamment d'espace disponible

Fermeture des descripteurs de fichiers:

Il est possible de fermer un descripteur à l'aide de la méthode close(int descripteur).

Duplication des descripteurs de fichiers:

La fonction système dup(int descripteur_a_copier) permet de dupliquer un descripteur de fichier passé en paramètre, en utilisant le plus petit numéro de descripteur disponible dans la table de descripteurs pour l'enregistrer (dans la première entrée libre dans la table des descripteurs).

Les Threads

Les processus classiques (lourds) d'UNIX possèdent des ressources séparées (espaces mémoire, table de fichiers ouverts …). Lorsqu'un nouveau processus est crée par un fork(), il se voit attribuer une copies des ressources du processus père :

- Problèmes de performances, car la duplication est un mécanisme coûteux
- Problèmes de communication entre processus, qui ont des variables séparées.

→ La solution : les threads

Les threads permettent d'avoir plusieurs fils d'éxécution dans un même espace de ressources non dupliquées → Notion de processus légers

La communication donc entre deux threads est une opération plus économique !

Il existe plusieurs mécanismes de synchronisation : exclusion mutuelle (mutex), sémaphores et conditions …

A RETENIR :

- Processus : indépendants, difficile pour le partage de ressources
- Threads : infos partagées, mais accès concurrent à gérer

Pour accéder aux méthodes associés aux threads, il est nécessaire d'inclure pthread.h et d'ajouter -pthread dans la ligne de compilation.

Pthread t

Pthread_t est la structure de données représentant un thread.

Pthread_create

pthread_create lance un nouveau processus léger, avec les attributs pointés par attr(NULL = attributs par défauts)

```
int pthread create( pthread_t *thread, pthread attr t *attr, void * (*start routine) (voi
```

Ce processus léer exécutera la fonction start_routine, en lui donnant le pointeur arg en

paramètre. L'identifiant du processus léger est rangé à l'endroit pointé par Thread

La valeur de retour de pthread_creat est 0 si tout est ok !

Pthread_join

Permet au processus père d'attendre d'attendre la fin d'un processus léger, et de récupérer éventuellement son code de retour.

```
int pthread join( pthread t th, void **thread return);
```

Remarque : Le fonctionnement des processus légers peut être modifié (priorités, algorithme d'ordonnance …) en manipulant les attributs qui lui sont associés

Concurrence et comment la gérer

Des processus sont concurrent s'ils existent en même temps.

Deux modélisations possibles de traitements parallèles :

- Graphes de processus
- Structure parbein/parend

Graphes de processus

Il peut être utile lors de la conception d'un programme de visionner l'organisation des opérations atomiques afin de gérer au mieux le fonctionnement des threads. Ainsi, on divise le programme en un ensemble d'opération de base et on essaie d'optimiser leur organisation dans le temps/nombre de threads en les organisant tel que présenté ci-dessous:

Soit ils s'exécutent les un à la suite des autres : instr1 → instr2 →

Soit ils s'exécutent en parallèle :

- instr1 →
- instr2 →

Parbegin/parend

De même, on peut utiliser une structure de contrôle présentée ci dessous permettant de

représenter les opération pouavnt être réalisées parallélement.

```
parbegin;
{
instruction1;
instruction2;
}
Parend;
```

Ainsi, les instructions instruction1 et instruction2 peuvent être réalisées simultanément. Les instructions suivantes seront représentées dans un autre ensemble parbegin/parend.

Processus indépendants ou coopérants

Les processus concurrents peuvent être indépendants ou coopérants pour l'exécution d'une tâches commune.

Etre coopérants:

Intérêts	Inconvénient
optimise l'utilisation des ressources partagées	gérer les accès concurrent à ces ressources
permet également l'échange d'informations entre processus	

Les sections critiques

Une section critique est une zone de code que l'on veut protéger pour qu'elle ne soit accessible que par un seul processus à la fois, car ses données peuvent être accéder par un ou plusieurs threads en même temps. → On parle d'exclusion mutuelle

Interblocage

Un ensemble de processus (thread) est en interblocage si chaque processus de l'ensemble attend qu'un évènement que seul un autre processus de l'ensemble peut engender.

Mutex (aka verrous d'exclusion mutuelle)

→ Permet de gérer l'accès concurrent à une ressource

Un processus souhaitant alors utiliser cette ressource :

- Peut demander à utliser le verrou

- Si le verrou est déjà utilisé, le verrou reste en attente (attente passive)
- Dès que le verrou devient inutilisé, un processus en attente obtient le verrou et redevient actif

pthread_mutex_init crée un verrou d'exclusion mutuelle (mutex).

```
int pthread mutex init( pthread mutex t *mutex, const pthread mutexattr t *mutexattr);
```

Il en existe de différents types (attributs pointés par le paramètres mutexattr (par défaut NULL)).
L'identificateur du verrou est placé dans la variable pointée par mutex.

ptread_mutex_destroy détruit le verrou.

```
int pthread mutex destroy(pthread mutex t *mutex);
```

La fonction pthread mutex lock tente de bloquer le verrou (met le thread en attente s'il est d ´ej`a bloqu´e),

```
int pthread_mutex_lock(pthread mutex t *mutex);
```

La fonction pthread mutex unlock le d´ebloque,

```
int pthread_mutex_unlock(pthread mutex t *mutex);
```

La fonction pthread mutex trylock tente de bloquer le verrou, et ´echoue s'il est d´ej`a bloqu ´e.

```
int pthread_mutex_trylock(pthread mutex t *mutex)
```

Néanmoins, le problèmes posé par les mutex est le suivant: un mutex ne peut gérer qu'une et une seule section critique en même temps, il est donc nécessaire de créer autant de mutex que de section critiques.

Sémaphores

Un sémaphore peut être :

- Initialisé à n'importe quelle valeur, mais il ne peux, par la suite, être décrémenter ou incrémenter que de -1 ou +1.

- Si un thread décrémente le sémaphore et que la valeur de cedernier deviens négative, le thread se bloque.
- Si un thread incrémente un sémaphore et que certains threads sont bloqués, alors l'un d'eux est débloqué.

Ce que represente la valeur n d'un semaphore :

- n > 0 n ressources disponibles, donc les threads peuvent decrementer n fois sans bloquer
- n == 0 pas de thread bloque, mais si un thread decremente, il sera bloque
- n < 0 n represente le nombre de threads bloques.

Un sémaphore initialisé à 0 reviens à un échange de signaux

Un sémaphore initialisé à 1 est un mutex

sem_init crée un sémaphore et place l'identificateur du sémaphore à l'endroit pointé par sem, et l'initialise à value. Si pshared est nul, le sémaphore est local au processus lourd.

```
#include <semaphore.h>
int sem init(sem t *sem,
int pshared,
unsigned int value);
int sem destroy(sem t *sem);
```

Les fonctions sem wait et sem post sont les primitives :

```
int sem wait(sem t *sem); // d´ecr´emente int sem post(sem t *sem); // incr´emente
```

La fonction sem trywait ´echoue (au lieu de bloquer) si la valeur du s´emaphore est nulle.

```
int sem trywait(sem t *sem);
```

La fonction sem getvalue consulte la valeur courante du s´emaphore.

```
int sem getvalue(sem t *sem, int *sval);
```