Programmation système et reseau cours 2 : les signaux et les semaphores

Besoins de communication entre processus

- Deux processus peuvent éventuellement partager le même code (ceci est sous le contrôle du système) mais les espaces mémoires réservés aux variables sont complètement étanches : un processus n'a **aucun accès** aux **variables d'un autre processus**

- On peut néanmoins créer une application multitâche et avoir besoin de **synchroniser ces tâches** ou **d'échanger des informations entre ces tâches.**
- Les systèmes d'exploitations sont donc munis de mécanismes appropriés:

  - Synchronisation par **signaux** ou **sémaphores**
  - Communication de données par **fichiers**, **tubes** (pipes) files de **messages** (queues ou boîtes à lettres) variables implantées dans un **segment de mémoire partagée**

Les signaux

- Un signal est une **interruption logicielle** asynchrone d'un processus. Le signal peut être envoyé par un **processus** ou par **le noyau**.

- Les signaux sont identifiés par un **numéro entier** et un **nom symbolique** décrit dans **signal.h**.

- Comme pour une interruption matérielle, la réception d'un signal interrompt le traitement en cours et exécute automatiquement la fonction associée au signal (**programmation événementielle**).

- En langage C, l'association entre le numéro du signal et la fonction est réalisé par un appel à la fonction système **signal().**

- la fonction **sigaction()** peut être utilisée à la place de signal().

Les signaux

- **Un signal est:**
  - ✓ Envoyé par un processus
  - ✓ Reçu par un autre processus (éventuellement le même)
  - ✓ Véhiculé par le noyau

- **Comment réagit un processus qui reçoit un signal ?**
  - ✓ Il interrompt le traitement en cours
  - ✓ il exécute la fonction de traitement du signal
  - ✓ il reprend l'exécution du traitement interrompu

Remarques: Si le processus était endormi, il est **réveillé par le signal**. Après l'exécution de la fonction associée au signal, dans le cas où il est réveillé avant la fin d'une temporisation il **ne se rendort pas**; par contre s'il attendait la fin d'une **entrée-sortie**, il **continue à attendre**

Les signaux

- **Comportement associée à la réception d'un signal :**

  - ✓ En général un comportement par défaut est défini : Terminaison anormale du processus (ex: violation de mémoire, division par zéro)

  - ✓ Le processus peut ignorer le signal

  - ✓ Le processus peut redéfinir, par une fonction spécifique, son comportement à la réception d'un signal

Origine des signaux

- **Frappe de caractères**

touche	signal
CTRL-C	SIGINT
CTRL-\	SIGQUIT
CTRL-Z	SIGSTP

- **Erreurs du programme**
  - ✓ Transmis par le noyau
  - ✓ Violation de mémoire, SIGSEGV
  - ✓ Division par zéro, SIGFPE
  - ✓ ...
- **Primitive système :** kill(), alarm().

$ kill –KILL 2400

La commande ci-dessus envoie le signal **SIGKILL** au **processus** d'identité **2400**.

$ kill –9 2400

Liste des signaux

$ kill -l

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL
5) SIGTRAP	6) SIGABRT	7) SIGBUS	8) SIGFPE
9) SIGKILL	10) SIGUSR1	11) SIGSEGV	12) SIGUSR2
13) SIGPIPE	14) SIGALRM	15) SIGTERM	17) SIGCHLD
18) SIGCONT	19) SIGSTOP	20) SIGTSTP	21) SIGTTIN
22) SIGTTOU	23) SIGURG	24) SIGXCPU	25) SIGXFSZ
26) SIGVTALRM	27) SIGPROF	28) SIGWINCH	29) SIGIO
30) SIGPWR	31) SIGSYS	33) SIGRTMIN	34) SIGRTMIN+1
35) SIGRTMIN+2	36) SIGRTMIN+3	37) SIGRTMIN+4	38) SIGRTMIN+5
39) SIGRTMIN+6	40) SIGRTMIN+7	41) SIGRTMIN+8	42) SIGRTMIN+9
43) SIGRTMIN+10	44) SIGRTMIN+11	45) SIGRTMIN+12	46) SIGRTMIN+13
47) SIGRTMIN+14	48) SIGRTMIN+15	49) SIGRTMAX-15	50) SIGRTMAX-14
51) SIGRTMAX-13	52) SIGRTMAX-12	53) SIGRTMAX-11	54) SIGRTMAX-10
55) SIGRTMAX-9	56) SIGRTMAX-8	57) SIGRTMAX-7	58) SIGRTMAX-6
59) SIGRTMAX-5	60) SIGRTMAX-4	61) SIGRTMAX-3	62) SIGRTMAX-2
63) SIGRTMAX-1	64) SIGRTMAX		

Signaux les plus fréquemment utilisés

Signal	
1 SIGHUP	terminaison du processus leader
2 SIGINT	frappe d'interruption (CTRL-C)
3 SIGQUIT	frappe de quit (CTRL-\)
4 SIGILL	instruction illégale
6 SIGABRT	problème matériel ou appelle de la function **abort()**
7 SIGBUS	une erreur liée à l'accès à la mémoire
8 SIGFPE	erreur arithmétique
9 SIGKILL	**signal de terminaison** **la réaction à ce signal ne peut être redéfinie**
10 SIGUSR1	signal utilisateur
11 SIGSEGV	violation écriture mémoire
12 SIGUSR2	signal utilisateur
13 SIGPIPE	écriture sur un tube non ouvert en lecture
14 SIGALRM	fin de temporisateur

Signaux les plus fréquemment utilisés

15 SIGTERM	Terminaison normale d'un processus
17 SIGCHLD	Terminaison (arrêt) d'un fils
18 SIGCONT	Continuation d'un processus arrêté par SIGSTOP
19 SIGSTOP	Signal de suspension d'exécution de processus la réaction à ce signal ne peut être redéfinie
20 SIGTSTP	Frappe du caractère de suspension sur le clavier (CTRL-Z)

Envoi de signaux en langage C

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid,int sig)
```

La primitive kill permet d'émettre le signal **sig** (désigné par son **nom symbolique** ou **son numéro**) à :

- processus d'identité **pid**, si **pid > 0**,
- tous les processus du même groupe que le processus appelant, **si pid = 0**
- tous les processus du **groupe dont le gid** est la valeur **absolue** de **pid**, **si pid < −1**

Les processus émetteur et destinataire(s) doivent être du **même propriétaire**, ou que le **premier** soit **le super-utilisateur (root)**

Cette primitive renvoie **0** en cas de **succès** et **-1 sinon**

Réponse à un signal

Il existe trois manières de répondre à un signal :

1. **Exécution de l'action par défaut** pour le signal. Cinq traitements par défaut sont possibles:

   - **exit** : provoque la terminaison du processus (Ex. SIGINT, SIGKILL, SIGALRM)
   - **core** : sauvegarde l'état de la mémoire et termine le processus (Ex. SIGFPE, SIGBUS, SIGSEGV)
   - **stop** : suspend l'exécution du processus (Ex. SIGSTOP)
   - **ignore** : le signal est ignoré (Ex. SIGCHLD)
   - **continue** : le processus suspendu reprend son exécution ou le signal est ignoré (Ex. SIGCONT)

Réponse à un signal

2. **Ignorance du signal**. On peut ignorer un certain nombre de signaux (sauf par exemple **SIGKILL**, **SIGSTOP**)

3. **Interception puis invocation d'une fonction en réaction à l'événement**.

   Le signal peut être rattrapé par le processus destinataire, ce qui provoque un déroutement et le lancement d'une routine spécifique de traitement **(handler).** Après le traitement, le processus reprend où il a été interrompu.

Remarque

Un signal peut être dérouté soit à travers les appels systèmes **signal()** et **sigaction()** à partir d'un programme **C/C++** ou en utilisant la commande **trap** à partir du **shell**.

Exemple: **trap** 'commande' signal1 signal2 ...

Préparer la réception des signaux en C

```
#include <signal.h>
void (*signal(int signum, void (*handler)(int)))(int);
```

L'appel système signal() permet d'installer **handler** comme fonction du traitement lors de la réception du signal **signum**. Cette fonction ne pourra prendre qu'un argument de type entier : le numéro du signal qui l'aura appelée. La fonction renvoie **SIG_ERR** en cas d'échec.

signum : le signal à intercepter

handler : pointeur sur une fonction qui gère le signal c'est simplement en C l'identificateur de cette fonction ou constante correspondant à un comportement prédéfini:

- SIG_DFL : comportement par défaut (terminaison)
- SIG_IGN : ignorer le signal

Préparer la réception des signaux en C

- La fonction de déroutement (handler) permet d'afficher le numéro et le nom symbolique du signal dérouté. S'il s'agit du signal **SIGINT**, on **arrête** le programme.

- La fonction **strsignal** permet de renvoyer le nom symbolique du signal correspondant au numéro **sig**.

```
#include <string.h>
char *strsignal (int sig)
```

Préparer la réception des signaux en C

```
#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <stdlib.h>
//fonction de déroutement
void handler (int sig) {
printf("Bien reçu %d %s\n", sig, strsignal(sig));
if (sig == SIGINT) {
printf ("Fin volontaire\n");
exit (1);
}
}
void main () {
signal (SIGINT, handler); //Ctrl-c : signal 2
signal (SIGQUIT, handler); //Ctrl-\ : signal 3
signal (SIGTSTP, handler); //Ctrl-z : signal 20
//SIGKILL est non déroutable
if (signal (SIGKILL, handler) == SIG_ERR) perror("SIGKILL");
for (;;);
}
```

Préparer la réception des signaux en C

La fonction de traitement peut être remplacée par une des constantes suivantes :

SIG_IGN : qui indique que le signal doit être ignoré.

```
/*Rendre Ctrl-C (SIGINT) inopérant*/
/*l'arrêt de ce programme doit se faire avec kill -9 pid*/
#include <signal.h>
void main () {
signal (SIGINT, SIG_IGN);
for (;;);
}
```

SIG_DFL : qui indique de rétablir l'action par défaut pour le signal

Préparer la réception des signaux en C

L’appel système sigaction()

La primitive **sigaction()** associe un signal à un contexte de déroutement, représenté par une structure sigaction.

```
struct sigaction{
void (*sa_handler)(); /*SIG_DFL ou SIG_IGN ou ptr sur handler*/
sigset_t sa_mask; /*signaux supplémentaires à bloquer pendant l’execution de gestionnaire*/
int sa_flags; /*indicateurs optionnels pour mofifier le comportement du signal*/
}
```

Remarque

Le champ **sa_handler** est le seul obligatoire ; c'est un pointeur sur la fonction qui servira de **handler**

Préparer la réception des signaux en C

```
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <stdlib.h>
void handler (int sig) {
printf ("Bien reçu %d %s\n", sig,
strsignal(sig));
if (sig == SIGINT) {
printf ("Fin volontaire\n");
exit (1);
}
}
```

```
void main () {
struct sigaction act;
act.sa_handler = handler;
sigaction (SIGINT, &act, NULL); //Ctrl-c : signal 2
sigaction (SIGQUIT, &act, NULL); //Ctrl-\ : signal 3
sigaction (SIGTSTP, &act, NULL); //Ctrl-z:signal 20
//SIGKILL est non déroutable
if (sigaction(SIGKILL, &act, NULL) == -1)
perror ("SIGKILL");
for (;;);
}
```

Les sémaphores

Outils de coordination et synchronisation

But : protéger l'accès à des variables partagées

Exemple : gestion d'un stock

- Un processus (p1) lit dans un registre la valeur courante **V** du stock et veut la décrémenter.

- Un autre processus (p2) préempte le premier avant, lit la valeur courante (toujours **V**) , la décrémente et met à jour le registre qui contient alors **V-1**

- le premier processus reprend son exécution et met à jour le registre en écrivant **V-1**, au lieu de **V-2**

Outils de coordination

Problème : incohérence des données

Solution

utilisation des mécanismes de synchronisation entre les processus pour protéger les variables partagées comme **les sémaphores**

Pourquoi utiliser un sémaphore ?

- Un sémaphore sert à **la synchronisation** entre les processus.

- Un sémaphore utilisé pour l'exclusion mutuelle (garantir qu'un seul processus accède à un moment donné à une variable partagée), utilisera 2 primitives:

  - ✓ **Attendre l'accès exclusif** à la ressource partagée : si la ressource est déjà utilisée par un processus, le processus qui fait cette demande sera endormi et mis dans une file d'attente de processus.

  - ✓ **Libérer l'accès exclusif** à la ressource partagée : si la file d'attente des processus attendant cette ressource est non vide, le processus qui a dormi le plus longtemps est réveillé.

- La section de programme qui se trouve entre les 2 primitives **attendre** la ressource et **libérer** la ressource s'appelle **une section critique**

- Aux sémaphores sont associés des droits d'accès de type Unix.

Le "mécanisme" des sémaphores

- **Sémaphore** = objet composé :
  - ✓ Une variable (la valeur du sémaphore)
  - ✓ Une file d'attente (les processus bloqués attendant la ressource)

- **Primitives associées :**
  - ✓ **Initialisation** (avec une valeur **positive** ou **nulle**)
  - ✓ **Manipulation**
    1. **Prendre** (**P** ou Wait) = demande d'autorisation
    2. **Vendre** (**V** ou Signal) = fin d'utilisation

- **Principe** : sémaphore associé à une ressource
  1. **Prendre** = demande d'autorisation (puis-je utiliser la ressource?)
     Si valeur **> 0 accord**, sinon **blocage**
  2. **Vendre =** restitution d'autorisation (je n'ai plus besoin de la ressource) éventuellement déblocage d'un processus qui se trouve dans le file d'attente.

Sémaphores : algorithmes des primitives P et V

- **Initialisation(sémaphore, n)**
  valeur[sémaphore] = n
- **P(sémaphore)**

```
si (valeur[sémaphore] > 0) alors
     valeur[sémaphore] = valeur[sémaphore] - 1
sinon
          si (valeur[sémaphore] == 0)
                    étatProcessus = Bloqué
                    mettre processus en file d'attente
          fsi
finSi
 invoquer l'ordonnanceur
```

Sémaphores : algorithmes des primitives P et V

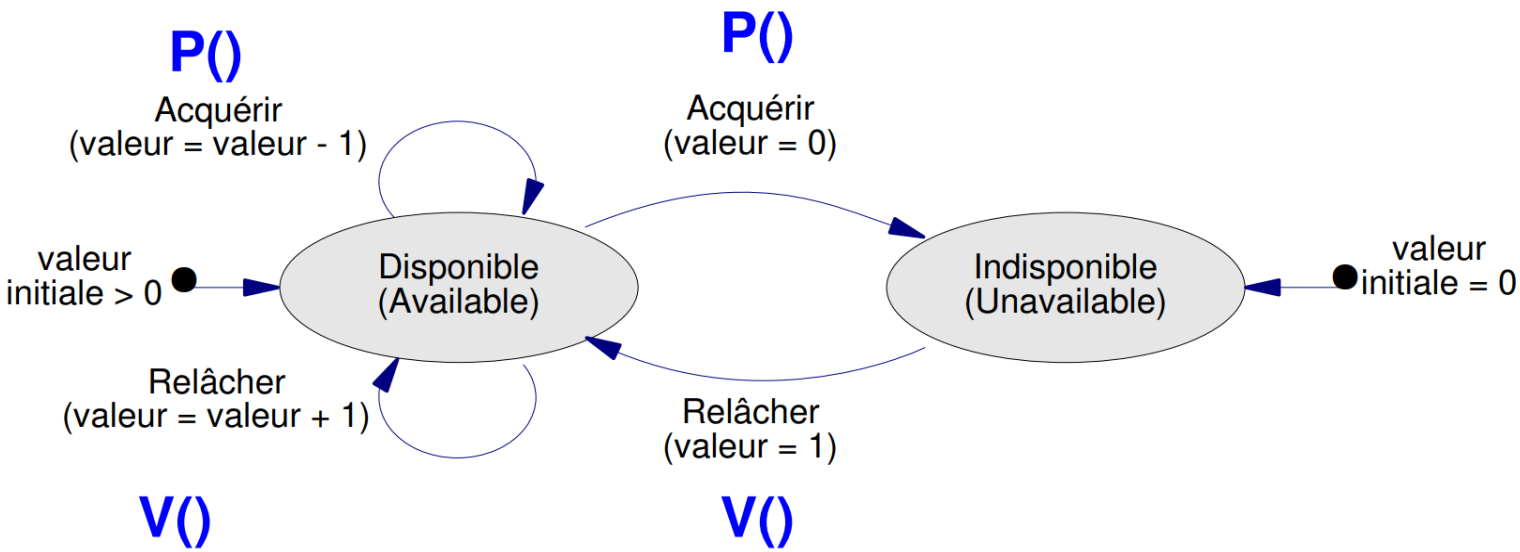
- **V(sémaphore)**

```
valeur[sémaphore] =valeur[sémaphore] + 1
si (valeur[sémaphore] > 0) alors
    extraire processus de file d'attente
    étatProcessus = Prêt
finSi
    invoquer l'ordonnanceur
```

Les différents types de sémaphores

sémaphores à compte

- valeur initiale positive ou nulle
- décrémentée à chaque fois que l'acquisition est accordée,
- incrémentée quand le sémaphore est relâché quand la valeur devient nulle, la demande d'acquisition est bloquante
- ressource globale



Sémaphores POSIX.4

Existent sous deux formes

- **sémaphores nommés**

  mécanisme de gestion analogue à celui d'un système de fichiers : accessible par tous les processus (selon les droits)

- **sémaphores anonymes**

  Basés sur la mémoire

Sémaphores nommés

Création de sémaphore

sem_t *sem_open(const char *sem_name, int oflags, mode_t creation_mode, unsigned int initial_val);

1. **sem_name : nom du semaphore à créer,** il est recommandé de fournir un nom "compatible « avec un nom de fichier, et commençant par un **« / » barre oblique** et peut contenir jusqu'à **255 caractères. (exemple:** "/semaphore1"**)**
    1. **oflags :**
        - ✓ **O_CREAT** pour créer un sémaphore.
        - ✓ **O_CREAT et O_EXCL** erreur si existe déjà.
    2. **initial_val :** valeur initiale.
    3. **creation_mode** : Ce paramètre définit les permissions (mode de création) du sémaphore
    4. **Retour :**
        - adresse du nouveau sémaphore
        - si echec SEM_FAILED + errno

Sémaphores nommés

sem_t *sem_open(const char *sem_name, int oflags);

- accède à un sémaphore déjà créé.
- **Sem_name**: nom du semaphore.
- **oflags**
  - ✓ **O_CREAT** : Crée un sémaphore nommé s'il n'existe pas déjà.
  - ✓ **O_EXCL** : Si O_CREAT est spécifié et que le sémaphore existe déjà, l'appel échoue.
  - ✓ **O_RDWR** : Ouvre le sémaphore en mode lecture/écriture (habituellement non utilisé pour les sémaphores).

int sem_close (sem_t *sem_id);

- ferme le sémaphore

Retour :

- renvoie 0.
- Si erreur : -1 + errno

Sémaphores nommés

int sem_unlink(const char *sem_name);

supprime le sémaphore.

- permet de supprimer un sémaphore nommé du système. Elle est utilisée pour supprimer un sémaphore qui a été précédemment créé avec sem_open.
- sem_unlink ne supprime pas immédiatement le sémaphore si des processus utilisent encore ce dernier. Le sémaphore sera effectivement supprimé lorsque toutes les références à celui-ci seront fermées

Retour :

- renvoie 0.
- Si erreur : -1 + errno

Sémaphores anonymes

sémaphores anonymes:

- Sont des sémaphores qui n'ont pas de nom associé dans le système de fichiers. Contrairement aux sémaphores nommés.
- Sont basés sur la mémoire

Création le sémaphore

int sem_init(sem_t *sem_location, int pshared, unsigned int initial_value);

- **sem_location:** est une zone mémoire, éventuellement en zone partagée.
- **Pshared:** indique la visibilité des sémaphores (un ou plusieurs processus) :
  - si le sémaphore est local au processus courant (pshared==0)
  - partagé entre plusieurs processus (pshared!=0)

Retour :

- renvoie 0.
- Si erreur : -1 + errno

Sémaphores anonymes

int sem_destroy(sem_t sem_location);

détruit le sémaphore

Retour :

- renvoie 0.
- Si erreur : -1 + errno

Opérations sur les sémaphores

P() : int sem_wait(sem_t *sem_id);

décrémente (verrouille) le sémaphore pointé par **sem_id**.

Si sem_id> 0 : décrémentation et retour immédiat.

Si sem_id=0: le processus appelant est mis en attente jusqu'à ce qu'une autre entité (procesus) incrémente la valeur du sémaphore (généralement avec sem_post), ce qui libère la ressource et réveille le processus en attente.

Retour :

- renvoie 0.
- Si erreur : -1 + errno

Opérations sur les sémaphores

V(): int sem_post(sem_t *sem_id);

- Incrémente le compteur et s'il devient positif réveille une tâche en attente

Retour :

- renvoie 0.
- Si erreur : -1 + errno