Programmation Système

Jean-Claude Bajard

IUT de Montpellier, Université Montpellier 2

Licence Génie Logiciel

Table des matières

Les Threads

Les Threads
Principe de fonctionnement

- Rappel : Les ressources d'un processus sont
  - Un espace d'adressage
  - Un propriétaire (User, Group)
  - Un répertoire de travail
  - Une table de descripteurs de fichiers
  - Des handlers de signaux

Les Threads
Principe de fonctionnement

- Principe du fork : copie des ressources au lancement, puis environnement propre.
- Principe du Thread:
  - Ressources communes avec le père (et donc les autres threads)
  - Par contre
    - Contexte d'exécution propre (registres, CO)
    - Attributs d'ordonnancement
    - Pile (sous pile du Père)
    - Signaux

Les Threads
Création de threads

Interface POSIX(Portable Operating System Interface for uniX)

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
```

thread : nom du thread créé

attr : attributs liés, si NULL alors attributs par défaut (modifiables après création)

start_routine : Fonction pointée, lancée lors de la création. La fin de cette fonction crée un pthread_exit().

arg : pointeur sur les arguments de la fonction.

Retour : 0 si tout est OK, un signal d'erreur sinon.

Les Threads

Exemple : Création de threads (1)

```
pthread_t tid[3]; // pour récupérer les identications des pthreads
char *nom[3] = {"ainee", "cadette", "benjamine"};

/* fonction exécutée au lancement de chaque pthread */
void *annexe(void *ordre) {
int boucle;
printf("pthread %s d'identite %d \n", ordre, (int)pthread_self());
for(boucle = 0; boucle < 10000000; boucle++) ;
printf("%s : apres la boucle 1\n", (char *)ordre);
for(boucle = 0; boucle < 10000000; boucle++) ;
printf("%s : apres la boucle 2\n", (char *)ordre);
}
```

Les Threads

Exemple : Création de threads (2)

```
main() {
int ind, rep,valeur;
char *val;
printf("Processus %d actif \n", getpid());
for(ind = 0; ind <3; ind++)
   if((rep = pthread_create(tid + ind, NULL, annexe, nom[ind])) == 0)
         printf("Thread %d cree\n", ind);
         else { fprintf(stderr, "%d : ", ind); perror("pthread_create"); }
sleep(20); // pour attendre la fin des threads fils
printf("Processus %d Fin \n", getpid());
}
```

Compilation : `gcc nom.c -o sortie -lpthread`

Attention : La fin du père entraîne la mort des threads fils

Les Threads

Exemple : Création de threads (3)

```
Processus 2969 actif
Thread 0 cree
Thread 1 cree
Thread 2 cree
Je suis la pthread ainee d'identite 41944576
Je suis la pthread cadette d'identite 41945600
Je suis la pthread benjamine d'identite 41946624
ainee : apres la boucle 1
benjamine : apres la boucle 1
cadette : apres la boucle 1
cadette : apres la boucle 2
ainee : apres la boucle 2
benjamine : apres la boucle 2
Processus 2969 Fin
```

Les Threads
Fin d'exécution des threads

```
#include <pthread.h>
void pthread_exit(void *value_ptr);
```

- Le thread termine et renvoie la variable pointée qui est récupérable si le thread est joignable
- Attention : la fonction `exit()` termine le processus père et tous les thread fils.
- L'ensemble est vu comme un processus unique.

Les Threads

Fin d'exécution des threads : attribut "detachstate"

- Un thread peut être déclaré joignable PTHREAD_CREATE_JOINABLE (attribut par défaut) dans ce cas lors de sa sortie avec pthread_exit, la valeur pointée est maintenue lisible
- Il est possible de modifier cet attribut avec la commande `int pthread_detach(pthread_t ind)` pour qu'il prenne la valeur PTHREAD_CREATE_DETACHED, dans ce cas le thread termine sans laisser de trace.

Les Threads

Fin d'exécution des threads : lecture de la valeur de sortie

```
#include <pthread.h>
int pthread_join(pthread_t tid, void **value_ptr);
```

- L'appel est bloquant : attente de la fin du thread tid
- Réception de la valeur pointée émise par le thread tid.
- Si la lecture a déjà été faite alors renvoie du message d'erreur ESRCH.

Les Threads

Fin d'exécution des threads : Exemple

Reprise de l'exemple précédent avec récupération des signaux

```
void *annexe(void *ordre) {
...
pthread_exit(ordre);}

main() {
char *val;
...
for(ind = 0; ind < 3; ind++)
   {pthread_join(tid[ind],(void **) &val);
   printf("Pthread %d valeur %s \n",(int) tid[ind],val);}
printf("Processus %d Fin \n", getpid());
}
```

Les Threads

Fin d'exécution des threads : Exemple

Reprise de l'exemple précédent avec récupération des signaux

```
Processus 3164 actif
Thread 0 cree
Thread 1 cree
Thread 2 cree
Je suis la pthread ainee d'identite 41944576
Je suis la pthread cadette d'identite 41945600
Je suis la pthread benjamine d'identite 41946624
...
Pthread 41944576 valeur ainee
Pthread 41945600 valeur cadette
Pthread 41946624 valeur benjamine
Processus 3164 Fin
```

Les Threads
Suppression d'un thread

```
#include <pthread.h>
int pthread_cancel(pthread_t tid);
```

- Lance une requête de suppression du thread tid
- Cette requête n'est prise en compte par tid que lorsque ce dernier arrive sur une primitive bloquante (`pause`, `msgrcv`,`pthread_cond_wait`), ou explicitement via l'appel à

```
#include <pthread.h>
int pthread_testcancel();
```

Threads : Gestion de la concurrence

Des effets de bord

Visualisation de la concurrence sur une variable globale (donc commune à tous les threads)

```
int varglob=0;
void *annexe(void *ordre) { ...
for(boucle = 0; boucle < 10000000; boucle++) varglob++;
...}

main() {
...
printf("Processus %d Fin valglob = %d \n", getpid(), varglob); }
```

Sortie : Processus 3208 Fin valglob = 21678781

Remarque : la somme cumulée aurait du être : 30000000

Threads : Gestion de la concurrence

Principe d'exclusion mutuelle (1)

- Principe de verrouillage et déverrouillage
- Déclaration et initialisation :

```
pthread_mutex_t varmutex = PTHREAD_MUTEX_INITIALIZER;
```

- Possibilité d'initialisation via une fonction :

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex, const
pthread_mutexattr_t *attr);
```

Si attr = NULL alors attributs par défaut sinon initialisation des attributs via pthread_mutexattr_init() mais complexe.

Threads : Gestion de la concurrence

Principe d'exclusion mutuelle (2)

- Verrouillage :
  `int pthread_mutex_lock(pthread_mutex_t *mutex);`
  ou non bloquant (si déjà un lock actif):
  `int pthread_mutex_trylock(pthread_mutex_t *mutex);`
- Déverrouillage :
  `int pthread_mutex_unlock(pthread_mutex_t *mutex);`

Threads : Gestion de la concurrence

Exemple d'exclusion mutuelle

```
int varglob=0;
pthread_mutex_t varmutex = PTHREAD_MUTEX_INITIALIZER;
void *annexe(void *ordre) { ...
for(boucle = 0; boucle < 10000000; boucle++)
    { pthread_mutex_lock(&varmutex);
    varglob++;
    pthread_mutex_unlock(&varmutex); }
...}
main() {
... printf("Processus %d Fin valglob = %d \n", getpid(), varglob); }
```

Sortie : Processus 3245 Fin valglob = 30000000

Remarque : ce contrôle ralenti énormément le programme. Par contre la somme cumulée est celle attendue.

Les sémaphores : contrôle des ressources

Principe de fonctionnement

- Déclaration et initialisation :

  ```
  #include <semaphore.h>
  int sem_init (sem_t *sempt, int portee, unsigned int valeur);
  ```

  - sempt: variable de type sémaphore
  - portee : nombre de threads partageant le sémaphore, si 0 alors tous partagent
  - valeur : valeur initiale du sémaphore
- Ce type de sémaphore est anonyme. Il n'est pas propre aux threads (voir fork)

Les sémaphores : contrôle des ressources

Principe de fonctionnement

(commandes POSIX)

- Attente de sémaphore :
  - int sem_wait (sem_t *sempt);
    Si valeur = 0 alors attente sinon valeur– et passage
  - int sem_trywait (sem_t *sempt);
    Ici pas d'attente.
- Dépot (ou relâchement)du sémaphore :
  int sem_post (sem_t *sempt);
- Destruction du sémaphore (libération des ressources) :
  int sem_destroy (sem_t *sempt);

Les sémaphores : contrôle des ressources

Exemple de fonctionnement

Problème des philosophes installés autour d'une table ronde avec un bol de riz devant chacun d'eux et une baguette entre chaque bol (n philosophes, n bols, n baguettes).
Un philosophe pense ou mange...
Pour manger un philosophe doit prendre les 2 baguettes disposées autour de son bol.

Comment faire pour qu'ils puissent manger (au moins un par un...)??
Une solution est de créer un sémaphore par baguette initialisé à un (baguette libre).
Tout philiosophe i prend d'abord la baguette i puis la i+1 mod n.

Enfin on crée un sémaphore veut-manger initialisé à n-1, pour éviter que les n désirent manger en même temps.

(Programme à écrire en TD)

Les I P C POSIX

Les Inter Process Communications
Principe de fonctionnement

Trois types de mécanismes:

- Les files de messages,
- Les sémaphores,
- Les segments de mémoire partagée.

Caractéristiques communes:

- Extérieurs au système de gestion des fichiers (pas de possibilité de redirection)
- Gestion d'une table spécifique par type d'objet

Option de compilation : En général appel à la bibliothèque `librt.a` d'où l'option `-lrt` nécessaire avec `gcc`.

Les Inter Process Communications

Principe de fonctionnement (suite)

Deux nommages :

- Identification interne avec un type de variable
  - `mqd_t` file de messages
  - `sem_t` sémaphore
  - `int` index de mémoire partagée
- Identification externe (nécessaire pour l'"inter process")
  - Un nom : chaîne de caractères commençant par /
  - Un lien entre la variable et l'objet externe nommé (lors de l'ouverture "open")

Les files de messages
Déclaration et typage

- Type opaque : `mqd_t` descripteur d'utilisation proche de celle du type `FILE`
- Les attributs sont (au moins) :
  - `long mq_flags` : Indicateur (0 ou O_NONBLOCK).
  - `long mq_maxmsg` : Nombre maximum de messages.
  - `long mq_msgsize` : Taille maximum d'un message.
  - `long mq_curmsgs` : Nombre de messages présents dans la file.
  - `int mq_pad[12]` : Zone de remplissage.

Les files de messages
Ouverture et initialisation

```
#include <mqueue.h>
mqd_t mq_open(const char *name, int oflag, ...);
```

`name` pointe sur une chaîne de caractères commençant par /.
Les options pour `oflag` sont :

- pour le mode :
  `O_RDONLY`, `O_WRONLY`, `O_RDWR`
- pour la création (en combinaison avec les précédentes options):
  `O_CREAT` impose la création
  `O_EXCL` crée une erreur si file déjà existante
  `O_NONBLOCK` envoi et reception non bloquants

Les files de messages
Ouverture et initialisation (suite)

Retour de `mq_open` : -1 si erreur sinon adresse d'un `mqd_t`
Gestion des attributs :

- Récupération d'un attribut :
```
int mq_getattr(mqd_t file, struct mqattr *pt );
```
- Modification d'un flag :
```
int mq_setattr(mqd_t file, struct mqattr *nouvel,
struct mqattr *ancien);
```
seul le flag est modifié, passage de `0` à `O_NONBLOCK`

Retour de ces fonctions : 0 si OK, -1 si erreur

Les files de messages

Ajout d'un message

```
int mq_send(mqd_t file, char *mess, size_t taille,
unsigned int prio);
```

Fonctionnement :

- Si non pleine et `taille` $<$ espace libre
  Alors Insère le message `mess` dans la file d'attente
  avec la priorité `prio` (de 0 à MQ_PRIO_MAX) (sysconf() permet de récupérer cette valeur max)
- Sinon,
  Si `O_NONBLOCK` passage à la suite avec envoi de -1
  Sinon blocage jusqu'à insertion possible...

Pour une même priorité principe FIFO

Les files de messages

Récupération d'un message

```
int mq_received(mqd_t file, char *mess, size_t taille,
unsigned int *prio);
```

Fonctionnement :

- Si non vide et taille `mess` $<$ `taille`
  Alors récupère le message `mess` (le plus ancien de plus grande priorité)
  avec la priorité `prio`
- Sinon si `mess` $>$ `taille`, alors échec
  Sinon Si `O_NONBLOCK` passage à la suite avec envoi de -1
  Sinon blocage jusqu'à message dans la file...

Les files de messages

Fin d'utilisation d'une file

```
int mq_close(mqd_t file);
```

Fermeture du descripteur local

```
int mq_unlink(const char *nom);
```

Fermeture du descripteur externe

Cet appel bloque toute tentative d'ouverture

La fermeture est effective après fermeture locale par tous les processus utilisateurs.

Mémoire partagée
Principe et création

Fonctionnement : sur le principe des fichiers avec des accès read/write

```
#include <sys/mman.h >
int shm_open(const char *nom, int option, int mode);
```

Les options sont : O_RDONLY, O_WRONLY, O_RDWR, O_CREAT, O_EXCL , O_NONBLOCK

mode : droit fichier en octal

Mémoire partagée

Lecture et écriture

```
#include <sys/types.h >
#include <sys/uio.h >
#include <unistd.h >
ssize_t read(int d, void *buf, size_t nbytes);
```

Lecture de la mémoire partagée d

```
ssize_t write(int d, const void *buf, size_t nbytes);
```

Ecriture dans la mémoire partagée d

Mémoire partagée
Fermeture

```
int close(int d);
```

Fermeture locale

```
int shm_unlink(const char *nom);
```

Suppression de l'objet externe dès tous les descripteurs locaux sont fermés

Mémoire partagée
Projection

```
#include <sys/types.h>  #include <sys/mman.h>
 void * mmap(void *addr, size_t len, int prot, int
flags, int fd, off_t offset);
```

`addr` : en général NULL
`len` : taille du segment mémoire
`prot` : protection : `PROT_EXEC`, `PROT_READ`, `PROT_WRITE`
`flags` : option (voir man) `MAP_PRIVATE`, `MAP_SHARED`, `MAP_FILE...`
`fd` : la mémoire partagée
`offset` : position dans fd
Retourne l'adresse où fd est placée en mémoire.

Les Sémaphores

Ouverture

```
#include <semaphore.h>
 sem_t * sem_open(const char *name, int flags);
 sem_t * sem_open(const char *name, int flags, mode_t
mode, unsigned int value);
```

`flags` : `O_CREAT`, `O_EXCL`

`mode` : droit fichier

`valeur` : valeur initiale du sémaphore

Les Sémaphores

Fonctionnement

```
int sem_wait(sem_t *sem;
```

Attente sémaphore non nul, sinon décrémente le sémaphore

```
int sem_trywait(sem_t *sem;
```

idem mais non bloquant.

```
int sem_post(sem_t *sem;
```

Libère le sémaphore en l'incrémentant

Les Sémaphores

Fermeture

```
sem_t * sem_close(sem_t *sem;
```

Fermeture locale

```
sem_t * sem_unlink(const char *name, int flags);
```

Fermeture du descripteur externe après fermeture de tous les locaux

Introduction aux Sockets

Les Sockets
Principe de fonctionnement

- Communications inter-processus, inter-machines via un réseau;
- Socket: plot de connexion
- Notion de client-serveur :
  - Le serveur possède un processus actif exécuté en tâche de fond répondant à des requêtes lancées par des clients (locaux ou distant),
  - Mise en place d'un protocole d'échanges (TCP-IP, UDP-IP...).

Les Sockets

Création d'un socket

```
#include <sys/types.h>
#include <ys/socket.h>
int socket(int domain, int type, int protocol);
```

- `socket` renvoie un numéro référençant le socket (descripteur)
- `domain` :
  `AF_UNIX` pour un domaine local
  `AF_INET` pour un domaine internet
  Autres domaines : `AF_ISO`, `AF_NS`,`AF_IMPLINK`.

Les Sockets
Création d'un socket (suite)

- `type` :
  `SOCK_STREAM` flux de données binaires en mode connecté, fiabilité maximale
  `SOCK_DGRAM` Transmission de datagrammes, mode non connecté, aucune garantie de fiabilité
  Autres types : `SOCK_RAW`, `SOCK_RDM`, `SOCK_SEQPACKET`

Les Sockets
Création d'un socket (suite)

- `protocol` : Ils sont visibles via `man 5 protocols`, plus précisément dans `/etc/protocols`
  On trouve entre autres :
  `ip 0 IP # internet protocol, pseudo protocol`
  `number` laisse le choix au système en fonction du type
  `tcp 6 TCP # transmission control protocol`
  `udp 17 UDP # user datagram protocol`

Les Sockets

Attachement d'un socket

- Une fois créé un socket doit être rendu accessible. Pour ceci, il est nommé via la primitive `bind()`:

```
int bind(int sock, const struct sockaddr *name,
socklen_t namelen);
```

- `sock` : représente le descripteur récupéré avec `socket()`
- `name` : est une structure d'enregistrement associée au socket dépendant du domaine.
- `namelen` : est la longueur de cette structure

Les Sockets

Attachement d'un socket (suite)

Le type `struct sockaddr` dépend du domaine :

- `AF_UNIX`

```
#include <sys/un.h>
struct sockaddr_un {
 unsigned char sun_len;   longueur
 sa_family_t sun_family;   AF_UNIX
 char sun_path[104];   path name
};
```

Les Sockets

Attachement d'un socket (suite)

Le type `struct sockaddr` dépend du domaine :

- `AF_INET`

```
#include <netinet/in.h>
struct sockaddr_in {
  __uint8_t sin_len; longueur
  sa_family_t sin_family; AF_INET
  in_port_t sin_port; numero de port
  struct in_addr sin_addr; adresse ip de la machine
  char sin_zero[8]; caracteres nuls };
```

Avec

```
struct in_addr { in_addr_t s_addr; };
typedef __uint32_t in_addr_t;  /* IPV4 */
```

Les Sockets

Attachement d'un socket (suite)

- Pour connaître les adresses `sin_addr;` utilisation de `gethostname()` et de `gethostbyname()` voir le manuel
- Pour prendre en compte toutes les adresses de la machine :

```
struct sockaddr_in adresse
s adresse.sin_addr.s_addr = htonl(INADDR_ANY )
```

Les Sockets

Attachement d'un socket (suite)

Le numéro de port :

- Essentiel côté serveur pour distinguer ses clients voir `/etc/services` :

```
# Well Known Ports are those from 0 through 1023.
# Registered Ports are those from 1024 through 49151
# Dynamic and/or Private from 49152 through 65535
```

- Côté client le plus simple mettre 0 laissant le système attribuer un numéro sinon un numéro quelconque non attribué

Remarque :

- `getsockname()` pour récupérer le nom d'un socket
- `adresse.sin_port = htons(port)` pour assurer le bon format

Les Sockets
Fermeture d'un socket

- En tant que descripteur un `socket` la fermeture se fait avec la fonction`close()`
- Attention : un socket étant lié à une communication la fermeture n'est effective qu'après délivrance de tous les messages.

Client-Serveur UDP
Principe de fonctionnement

C'est un mode non connecté

- Côté serveur : utilisation de `socket()` et `bind()` pour créer le socket en précisant un numéro de port
- Côté client : utilisation de `socket()` et `bind()` pour créer le socket avec 0 comme numéro de port

Lors de la requête faite par le client le serveur récupère adresse et numéro de port du client.

Client-Serveur UDP

Envoi de messages

`ssize_t sendto(int s,` descripteur de socket
`const void *msg,` message
`size_t len,` longueur du message (<9000)
`int flags,` 0 par défaut sinon voir manuel
`const struct sockaddr *to,` adresse et port serveur
`socklen_t tolen);` longueur du struct sockaddr

Remarque : `const struct sockaddr` est générique pour les adresses "in" et "un".

```
ssize_t sendmsg(int s, const struct msghdr *msg, int
flags);
```

Client-Serveur UDP

Envoi de messages (suite)

Pour des envois d'un message dispersé (fragmenté) en mémoire et rassemblé à l'envoi

```
struct msghdr {
 void *msg_name;  optional address
 socklen_t msg_namelen;  size of address
 struct iovec *msg_iov; scatter/gather array
 int msg_iovlen;  nb elements in msg_iov
 void *msg_control;  ancillary data, see below
 socklen_t msg_controllen;  ancillary data buffer len
 int msg_flags;  flags on received message };
```

Client-Serveur UDP

Réception de messages

Réception d'un message

```
ssize_t recvfrom(int s,
void *buf,
 size_t len,
 int flags,
 struct sockaddr *from,
 socklen_t *fromlen);
```

id du socket local

id de l'expéditeur

Réception d'un message fragmenté

```
ssize_t recvmsg(int s, struct msghdr *msg, int
flags);
```

Client-Serveur TCP-IP (mode connecté)

Architecture du serveur

- Création d'un socket : `socket()`
- Attachement à un port connu : `bind()`
- Ouverture du service (FIFO) : `listen()`
- Acceptation d'une connexion : `accept()`
- Création d'un processus (fork ou thread) pour la gestion du dialogue
- Retour à l'ouverture de service

Remarque: un nouveau descripteur est associé à la connexion.

Client-Serveur TCP-IP (mode connecté)

Architecture du serveur (suite)

```
int listen(int s, int backlog);
```

`int s` id du socket

`int backlog` nb maxi de connexions, voir SOMAXCONN dans /usr/include/sys/socket.h

Client-Serveur TCP-IP

Architecture du serveur (suite)

```
int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

`int s` id du socket

`struct sockaddr *addr` récupère l'adresse du client si non NULL

Attention : `accept()` retourne le nouveau descripteur à utiliser pour le dialogue

Attention : l'appel à `accept()` est bloquant si la file d'attente est vide

Remarque : `select()` teste l'existence de connexions en attente.

Client-Serveur TCP-IP (mode connecté)

Architecture du client

- Création d'un socket : `socket()`
- (optionnel Attachement à un port connu : `bind()`)
- Construction de l'adresse du serveur : IP + port
- Demande de connexion : `connect()`
- Dialogue

Client-Serveur TCP-IP (mode connecté)
Architecture du client (suite)

```
int connect(int sockfd, const struct sockaddr
*serv_addr, socklen_t addrlen);
```

`int sockfd` id du socket local du client

`const struct sockaddr *serv_addr` adresse du serveur

`socklen_t addrlen` longueur de l'adresse du serveur

Attention : l'appel à `connect()` est bloquant si le serveur ne répond pas

Client-Serveur TCP-IP (mode connecté)

Dialogue client-serveur : envoi de messages

Avec une fonction propre aux sockets :

```
ssize_t send(int s, const void *msg, size_t len, int
flags);
```

Ou encore comme pour les fichiers :

```
#include <unistd.h>
 ssize_t write(int fd, const void *buf, size_t count);
```

Client-Serveur TCP-IP (mode connecté)

Dialogue client-serveur : réception de messages

Avec une fonction propre aux sockets :

```
ssize_t recv(int s, void *buf, size_t len, int flags);
```

Ou encore comme pour les fichiers :

```
#include <unistd.h>
 ssize_t read(int fd, const void *buf, size_t count);
```