Programmation Système sous Linux/Unix

Dr. Ing. Chiheb Ameur ABID

Contact: chiheb.abid@gmail.com

Mars 2023

Plan

Les signaux

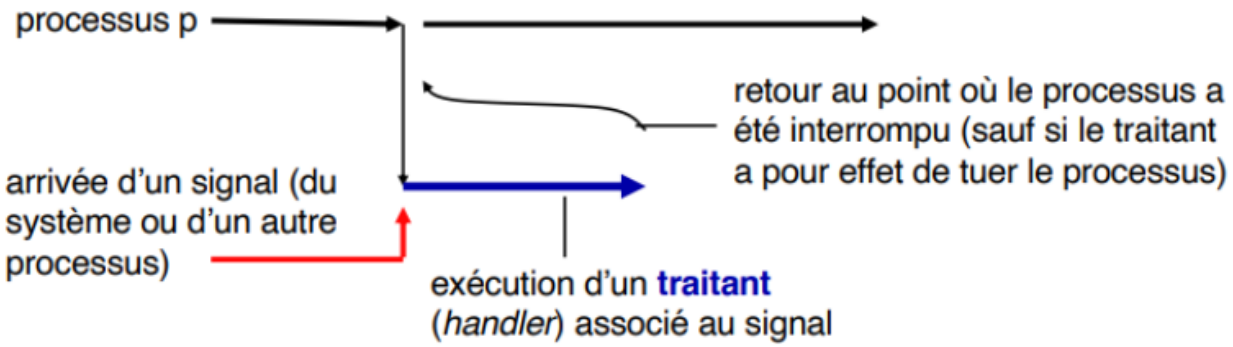
Préentation

- Un signal est un événement asynchrone destiné à un (ou plusieurs) processus.
  - Un signal peut être émis par un processus ou par le système d'exploitation.
- Un signal est analogue à une interruption
  - Un processus destinataire réagit à un signal en exécutant un programme de traitement, ou traitant (handler)
  - La différence est qu'une interruption s'adresse à un processeur alors qu'un signal s'adresse à un processus
  - Certains signaux traduisent d'ailleurs la réception d'une interruption
- Les signaux sont un mécanisme de bas niveau. Ils doivent être manipulés avec précaution car leur usage recèle des piège
  - Le risque de perte de signaux
  - Ils sont utiles lorsqu'on doit contrôler l'exécution d'un ensemble de processus ou que l'on traite des événements liés au temps.

Les signaux

Fonctionnement

- Il existe différents signaux, chacun étant identifié par un nom symbolique : ce nom représente un entier
- Chaque signal est associé à un traitant (handler) par défaut
- Un signal peut être ignoré (le traitant est vide)
- Le traitant d'un signal peut être changé (sauf pour 2 signaux particuliers)
- Un signal peut être bloqué (il n'aura d'effet que lorsqu'il sera débloqué)
- Les signaux ne sont pas mémorisés (
  - Risque de perte



Les signaux

États d'un signal

- Un signal est envoyé à un processus destinataire et reçu par ce processus
  - Tant qu'il n'a pas été pris en compte par le destinataire, le signal est **pendant**
  - Lorsqu'il est pris en compte (exécution du traitant), le signal est dit **traité**
- Qu'est-ce qui empêche que le signal soit immédiatement traité dès qu'il est reçu ?
  - Le signal peut être bloqué, ou masqué (c'est à dire retardé) par le destinataire. Il est délivré dès qu'il est débloqué
  - En particulier, un signal est bloqué pendant l'exécution du traitant d'un signal du même type ; il reste bloqué tant que ce traitant n'est pas terminé

Avertissement !

Il ne peut exister qu'un seul signal pendant d'un type donné (il n'y a qu'un bit par signal pour indiquer les signaux de ce type qui sont pendants). S'il arrive un autre signal du même type, il est perdu.

Les signaux

Types de signaux

- Linux supporte les 32 signaux standard ainsi que les 32 signaux temps réel (normes POSIX)
- Chaque signal est défini par un numéro unique auquel est associé un nom symbolique (`SIG*`);
  - Éviter d'utiliser les numéros, et préférer les noms explicites (compatibilité entre les implémentations et sûreté)

Valeur	Signal
0 ... 31	Signaux classiques (non temps-réel)

32
...
63
	`SIGRTMIN`
	(signaux temps-réel)
	`SIGRTMAX`

64	`NSIG`

Les signaux

Les signaux

- Récupérer la liste des signaux supportés par le système
  `kill -l`

```
chiheb@chiheb-82ht:~$ 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
16) SIGSTKFLT   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      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+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
```

Les signaux

Exemples de signaux

N°	Nom	Rôle
1	SIGHUP	Hang Up, envoyé par le père à tous ses fils lorsqu'il se termine.
2	SIGINT	Interruption du processus demandé (touche , [Ctrl]+C).
3	SIGQUIT	Idem SIGINT mais génération d'un Core Dump (fichier de débuggage).
9	SIGKILL	Signal ne pouvant être ignoré, force le processus à finir *brutalement*.
15	SIGTERM	Signal par défaut. Demande au processus de se terminer normalement.

Plan

Les signaux

Émission d'un signal

- Émission implicite
  - Division par 0 : engendre `SIGFPE` (Floating Point Exception)
  - Violation mémoire : engendree SIGSEGV (segmentation fault)
  - Tube sans lecteur reçu lors d'un `write()` dans un tube par un écrivain sans lecteur : engendre le signal `SIGPIPE`
  - Mort d'un fils envoyé par un fils à son père lors de sa terminaison exit : SIGCHLD

Les signaux

Émission d'un signal

- Émission explicite
  - Seuls les processus issus du même propriétaire peuvent échanger des signaux (mis à part root).
  - Appel système

```
int kill(pid_t pid,int sig);
```

  - Commande

```
kill [ -s signal | -p ] [ -a ] pid ...
```

Les signaux

Émission d'un signal

- Émettre un signal dans le processus

```
1 int raise(int sig);
```

  - Renvoie 0 si OK, sinon -1
  - `sig` désigne le signal à émettre

Les signaux

Émission d'un signal

- Planifier l'émission d'un signal `SIG_ALARM`

```
unsigned int alarm(unsigned int seconds);
```

  - Renvoie 0 si aucune alarme n'est définie
  - `seconds` spécifie la durée en secondes après laquelle le signal sera émis

Exemple d'émission d'un signal

```
#include<stdlib.h>
#include<unistd.h>
#include<signal.h>
void raisedAlarm(int sig);
int main() {
   alarm(5);
   signal(SIGALRM, raisedAlarm);
   while(1){
      printf("Hello!\n");
      sleep(1);
   }
   return 0;
}
void raisedAlarm(int sig) {
   printf("The Alarm has Raised.\n");
}
```

Les signaux

Émission d'un signal

- Émettre le signal `SIGABRT` ce qui entraine la terminaison du processus

```
1  void abort(void);
```

  - La terminaison du processus est considérée comme anormale
  - Les fichiers et les pointeurs ouverts ne seront pas fermés

Exemple d'émission d'un signal

```
1  #include<stdio.h>
2  #include<unistd.h>
3  #include<stdlib.h>
4  #include<signal.h>
5  int main(){
6     int status = fork();
7     if(status == 0){
8        printf("Child Process ID: %d\n", getpid());
9        while(1);
10    }else if(status > 0){
11       printf("Parent Process ID: %d\n", getpid());
12    }
13    abort(); // Child process will continue its execution
14    printf("Due to abnormal termination this line will not execute.\n");
15    return 0;
16 }
```

Les signaux

Mettre en pause un processus

➥ Mettre en pause le processus courant jusqu'à la réception d'un signal

```
int pause(void);
```

☞ Renvoie le signal intercepté si OK, sinon -1

Exemple

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
void SIGNAL_HANDLER(int);
int main(void){
   alarm(10);
   signal(SIGALRM, SIGNAL_HANDLER);
   if(alarm(7) > 0){
      printf("An alarm has been set already.\n");
   }
   pause();
   printf("You will not see this text.\n");
   return 1;
}
void SIGNAL_HANDLER(int signo){
   printf("Caught the signal with number: %d\n", signo);
   exit(0);
}
```

Les signaux

Mettre en pause un processus

- Mettre en pause le processus courant jusqu'à la réception d'un signal ou que la durée est écoulée

```
unsigned int sleep(unsigned int seconds);
```

  - Renvoie 0 si la durée est écoulée, sinon le signal intercepté si OK, sinon -1

Plan

Les signaux

Redéfinir le traitant d'un signal

- Un traitant doit être de type suivant :

```
1 typedef void (*sighandler_t)(int);
```

- Deux appels systèmes pour définir un traitant :

```
1 sighandler_t signal(int signum,sighandler_t handler);
2 int sigaction(int signum,const struct sigaction *act,struct sigaction *oldact);
```

  - `sigaction()` est plus complet et plus générique que `signal()`
  - `signal()` est plus simple à utiliser

Les signaux

Exemple de redéfinition d'un gestionnaire d'un signal avec `signal()`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <signal.h>
4  #include <unistd.h>
5  void gestionnaire(int numero_signal) {
6     fprintf(stdout,"\n␣%ld␣a␣recu␣le␣signal␣%d␣\n",(long) getpid(),
7     numero_signal);
8  }
9  int main (void) {
10    int i;
11
12    if (signal(SIGINT, gestionnaire) == SIG_ERR)
13    fprintf(stderr, "Signal␣%d␣non␣capturé \n", SIGINT);
14    while (1)
15    pause();
16 }
```

Plan

Communiquer sur le réseau

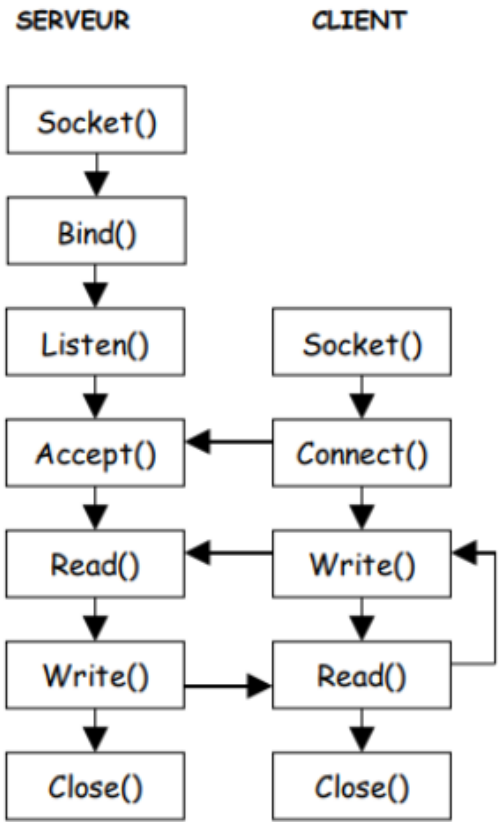
Introduction aux sockets

- La notion de sockets a été introduite dans les distributions de Berkeley (un fameux système de type UNIX)
- Il s'agit d'un modèle permettant la communication IPC
  - Communiquer sur la même machine
  - Ou en réseau à travers le protocole TCP/IP
- On distingue deux modes de communication
  1. Le mode connecté,
  2. Le mode non connecté

Communiquer sur le réseau

Schéma général d'une communication en mode connecté

- Utilise le protocole TCP.
- Une connexion durable est établie entre les deux processus, de telle façon que la socket de destination n'est pas nécessaire à chaque envoi de données.



Communiquer sur le réseau

Principe de la communication IPC avec les sockets

1. Chaque machine crée une socket,
2. Chaque socket sera associée (attachée) à un port de sa machine hôte,
3. Les deux sockets seront explicitement connectées si on utilise un protocole en mode connecté
4. Chaque machine lit et/ou écrit dans sa socket,
5. Les données vont d'une socket à une autre à travers le réseau,
6. Une fois terminé chaque machine ferme sa socket.

Communiquer sur le réseau

Créer une socket

- Créer une socket

```
1  int socket(int domain, int type, int protocol);
```

  - Retourne un descripteur de fichier si OK, sinon -1 et met le code d'erreur dans `errno`
  - `domain` spécifie le domaine de communication (familles de protocoles)
    `AF_INET` protocole fondé sur IP (IP, TCP, UDP et ICMP)
    `AF_INET6` protocole IPv6
  - `type` indique le type de service (orienté connexion ou non).
    `SOCK_STREAM` correspond à une communication en mode connectée, orientée flux d'octets
    `SOCK_DGRAM` correspond à une communication en mode datagramme, non connectée
    Il existe aussi d'autres types.
  - `protocol` définit le protocole de communication utilisé par la socket. Souvent, on le met à 0

Communiquer sur le réseau

Attacher une socket

- Une socket doit être attachée à une adresse et un port

```
int bind(int fd, struct sockaddr *addr, int addrlen);
```

  - `fd` est le descripteur du fichier identifiant le socket
  - `addr` est une structure qui spécifie l'adresse locale à travers laquelle le programme doit communiquer. Le format de l'adresse est fortement dépendant du protocole utilisé
  - `addrlen` indique la taille du champ `addr`. On utilise généralement `sizeof(addr)`.

Communiquer sur le réseau

Attente d'une connexion

- Mettre le serveur en attente d'une connexion

```
1 int listen(int socket,int backlog)
```

  - Renvoie 0 si OK, sinon `SOCKET_ERROR`
  - `socket` représente la socket ouverte
  - `backlog` représente le nombre maximal de connexions pouvant être mises en attente

Communiquer sur le réseau

Établir une connexion sur le serveur

- Se connecter à un serveur

```
1 int connect(int socket,struct sockaddr * addr,socklen_t addrlen)
```

  - Renvoie 0 si OK, sinon -1
  - `socket` représente la socket ouverte
  - `addr` est une structure qui spécifie l'adresse du serveur
  - `addrlen` indique la taille du champ `localaddr`. On utilise généralement `sizeof(localaddr)`.

Communiquer sur le réseau

Envoie/Réception

- Lire des données (réception)

```
int read(int socket,char * buffer,int len);
```

- Écrire des données (envoie)

```
int write(int socket,char * buffer,int len);
```

  - Renvoie le nombre d'octets lus (effectivement envoyés) si OK, sinon -1 (code d'erreur)
  - `socket` représente la socket ouverte
  - `buffer` est le tampon de données
  - `len` est la taille de données à lire (envoyer)

Plan

Protocole TCP

Schéma général d'une communication en mode non connecté

- Le protocole TCP sert à établir une communication fiable entre deux hôtes
  - Connexion : l'émetteur et le récepteur se mettent d'accord pour établir un connexion. La connexion reste ouverte jusqu'à ce qu'on la referme.
  - Fiabilité : Suite au transfert de données, des tests sont faits pour vérifier qu'il n'y a pas eu d'erreur dans la transmission. Ces tests utilisent la redondance des données, c'est à dire qu'un partie des données est envoyée plusieurs fois. De plus, les données arrivent dans l'ordre où elles ont été émises.
  - Possiblilité de communiquer sous forme de flot de données, comme dans un tube (par exemple avec les fonctions read et write). Les paquets arrivent à destination dans l'ordre où ils ont été envoyés.
- Pour une socket destinée à être utilisée avec un protocole TCP/IP (avec connexion TCP) fondé sur IP (`AF_INET`), on utilise

```
1  #include <arpa/inet.h>
2  int sock = socket(AF_INET, SOCK_STREAM, 0);
```

Communiquer sur le réseau

Exemple de client socket en mode connecté

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <arpa/inet.h>
5
6  int main() {
7     int sock = socket(AF_INET , SOCK_STREAM ,0);
8     struct sockaddr_in addr;
9     addr.sin_family = AF_INET;
10    addr.sin_port = htons(12345); // Port du serveur
11    addr.sin_addr.s_addr = inet_addr("192.168.1.29"); // Adresse IP du serveur (localhost)
12
13    int res = connect(sock, (struct sockaddr*)&addr, sizeof(addr));
14    if(res==-1) { perror("connect"); exit(1); }
15    write(sock, "Hello", 6);
16    char buf[6];
17    read(sock, buf, 6);
18    printf("reçu: %s\n", buf);
19    close(sock);
20    return EXIT_SUCCESS;
21 }
```

Communiquer sur le réseau

Exemple de serveur socket en mode connecté

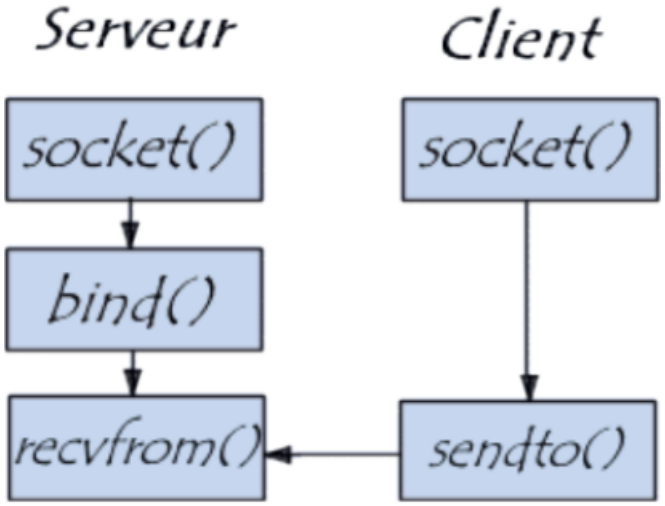
```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <arpa/inet.h>
6
7  int main() {
8     int sock = socket(AF_INET , SOCK_STREAM , 0);
9     struct sockaddr_in addr;
10    addr.sin_family = AF_INET;
11    addr.sin_port = htons(12345); // Port sur lequel le serveur écoute
12    addr.sin_addr.s_addr = INADDR_ANY; // Écoute sur toutes les interfaces disponibles
13
14    int res = bind(sock, (struct sockaddr *) &addr, sizeof(addr));
15    if (res == -1) { perror("bind"); exit(1); }
16    listen(sock, SOMAXCONN);
17    int cli = accept(sock, NULL, NULL);
18    write(cli, "World", 6);
19    char buf[6];
20    read(cli, buf, 6);
21    printf("reçu: %s\n", buf);
22    close(cli);
23    close(sock);
24    unlink("sock");
25    return EXIT_SUCCESS;
26 }
```

Plan

Communiquer sur le réseau

Schéma général d'une communication en mode non connecté

- Pas besoin d'établir une connexion
  - La socket est utilisée directement
- Les messages sont transmis par partis et de manière désordonnée



Protocole UDP

Schéma général d'une communication en mode non connecté

- Le protocole UDP permet seulement de transmettre les paquets sans assurer la fiabilité
  - Pas de connexion préalable ;
  - Pas de controle d'intégrité des données. Les données ne sont envoyées qu'un fois ;
  - Les paquets arrivent à destination dans le désordre.

Communiquer sur le réseau

Réception des données en mode non connecté

- Lire des données (réception) en mode non connecté

```
ssize_t recvfrom(int socket,char *buf,int len,int flags,sockaddr *from,socklen_t *frlen);
```

  - Renvoie le nombre d'octets lus (effectivement envoyés) si OK, sinon -1 et code d'erreur affecté dans `errno`
  - `socket` représente la socket ouverte
  - `buf` est le tampon de données
  - `len` est la taille de données à lire
  - `from` pointeur sur la structure contenant l'adresse de la destination définie par `recvfrom`
  - `frlen` est la taille de la structure adresse de la source

Communiquer sur le réseau

Exemple de réception des données en mode non connecté

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <arpa/inet.h>

int main() {
   int sock = socket(AF_INET , SOCK_DGRAM , 0);
   struct sockaddr_in addr;
   addr.sin_family = AF_INET;
   addr.sin_port = htons(12345); // Port sur lequel le serveur écoute
   addr.sin_addr.s_addr = INADDR_ANY; // Écoute sur toutes les interfaces
   int res = bind(sock, (struct sockaddr *) &addr, sizeof(addr));
   if (res == -1) { perror("bind"); exit(1); }
   printf("Serveur en attente de messages...\n");
   char buffer[1024];
   struct sockaddr_in clientAddr;
   socklen_t clientAddrLen = sizeof(clientAddr);
   int bytesRead = recvfrom(sock, buffer, sizeof(buffer), 0,
         (struct sockaddr*)&clientAddr, &clientAddrLen);
   if (bytesRead == -1) { perror("recvfrom");close(sock);exit(EXIT_FAILURE);}
   buffer[bytesRead] = '\0';
   printf("Message reçu du client : %s\n", buffer);
   close(sock);
   return EXIT_SUCCESS;
}
```

Communiquer sur le réseau

Envoie des données en mode non connecté

- Envoyer des données (réception) en mode non connecté

```
ssize_t sendto(int socket, const void *buf, size_t len,int flags,
      const struct sockaddr *to,socklen_t tolen);
```

  - Renvoie le nombre d'octets lus (effectivement envoyés) si OK, sinon -1 et le code d'erreur est affecté dans `errno`
  - `socket` représente la socket ouverte
  - `buf` est le tampon de données
  - `len` est la taille de données à envoyer
  - `flags` spécifie le type de transmission du message> Souvent, on utilise la valeur 0.
    `MSG_DONTROUTE` sert à déboguer les communications sur un réseau en négligeant les procédures de routage
    `MSG_PEEK` pour lire des données sur une socket TCP sans les extraire de la file d'attente
  - `to` pointeur sur la structure contenant l'adresse de la destination
  - `tolen` est la taille de la structure contenant l'adresse de la destination

Communiquer sur le réseau

Exemple d'envoie des données en mode non connecté

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>

int main() {
   int sock = socket(AF_INET , SOCK_DGRAM ,0);
   struct sockaddr_in addr;
   addr.sin_family = AF_INET;
   addr.sin_port = htons(12345); // Port du serveur
   addr.sin_addr.s_addr = inet_addr("127.0.0.1"); // Adresse IP du serveur (localhost)

   const char *message = "hello";
   if (sendto(sock, message, strlen(message), 0,
         (struct sockaddr*)&addr, sizeof(addr)) == -1) {
      perror("Erreur␣lors␣de␣l'envoi␣du␣message");
      close(sock);
      exit(EXIT_FAILURE);
   }
}
```

Merci pour votre attention

Questions ?