

Systèmes d'exploitation pour l'embarqué

UV 5.2 - Exécution et Concurrency

Paul Blottière

ENSTA Bretagne

24 Novembre 2015

<https://github.com/pblottiere>

Amélioration continue

Contributions



- Dépôt du cours : <https://github.com/pblottiere/embsys>

Amélioration continue

Contributions



- ▶ Dépôt du cours : <https://github.com/pblottiere/embsys>
- ▶ Souhaits d'amélioration, erreurs, idées de TP, ... :
ouverture d'Issues (avec le bon label!)
- ▶ Apports de corrections : Pull Request

IPC et programmation réseau

Plan

1. Descripteur de fichier
2. IPC : qu'est-ce?
3. Pipe
4. File de messages
5. Mémoire partagée
6. Sémaphore
7. Programmation réseau
8. Multiplexage d'entrées

Descripteur de fichier (1)

Généralités

File descriptor : entier compris entre 0 et OPEN_MAX.

Descripteur de fichier (1)

Généralités

File descriptor : entier compris entre 0 et OPEN_MAX.

Trois descripteurs réservés :

- ▶ 0 : STDIN_FILENO
- ▶ 1 : STDOUT_FILENO
- ▶ 2 : STDERR_FILENO

Descripteur de fichier (1)

Généralités

File descriptor : entier compris entre 0 et OPEN_MAX.

Trois descripteurs réservés :

- ▶ 0 : STDIN_FILENO
- ▶ 1 : STDOUT_FILENO
- ▶ 2 : STDERR_FILENO

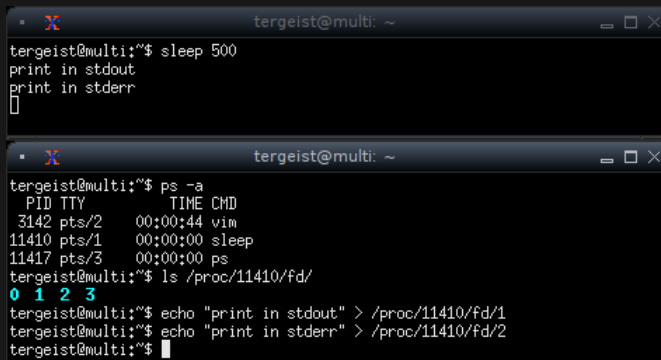
Un flux de type **FILE** * est associé aux descripteurs de fichier par défaut :

- ▶ stdin
- ▶ stdout
- ▶ stderr

Descripteur de fichier (2)

`/proc/<PID>/fd/`

Une liste de tous les file descriptors d'un processus est disponible dans le **/proc** :



```
tergeist@multi: ~  
tergeist@multi:~$ sleep 500  
print in stdout  
print in stderr  
█  
  
tergeist@multi: ~  
tergeist@multi:~$ ps -a  
  PID TTY          TIME CMD  
  3142 pts/2    00:00:44 vim  
 11410 pts/1    00:00:00 sleep  
 11417 pts/3    00:00:00 ps  
tergeist@multi:~$ ls /proc/11410/fd/  
0 1 2 3  
tergeist@multi:~$ echo "print in stdout" > /proc/11410/fd/1  
tergeist@multi:~$ echo "print in stderr" > /proc/11410/fd/2  
tergeist@multi:~$ █
```

Descripteur de fichier (3)

Exemple (fd.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <string.h>

int main ()
{
    int fd = open("/tmp/test.txt", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
    if (fd < 0)
    {
        fprintf(stderr, "Error on open(): %s\n", strerror(errno));
        return EXIT_FAILURE;
    }

    write(fd, "HEY YOU!!!!\n", 12);
    sleep(30); // just a sleep to observe some thing

    close(fd);

    return EXIT_SUCCESS;
}
```

```
tergeist@multi:~$
tergeist@multi:~$
tergeist@multi:~$
tergeist@multi:~$
tergeist@multi:~$
tergeist@multi:~$
tergeist@multi:~$
tergeist@multi:~$ ./fd
[]
```

```
tergeist@multi:~$ ps -a
  PID TTY          TIME CMD
 2250 pts/0    00:00:03 evince
 2735 pts/1    00:00:13 vim
13841 pts/3    00:00:00 fd
13846 pts/4    00:00:00 ps
tergeist@multi:~$ cat /proc/13841/fd/4
HEY YOU!!!!
tergeist@multi:~$ []
```

IPC : qu'est-ce?

Principe

Entre threads, la communication est simple : données partagées au sein du même espace mémoire.

IPC : qu'est-ce?

Principe

Entre threads, la communication est simple : données partagées au sein du même espace mémoire.

Dans le cas de communication entre processus, des mécanismes sont nécessaires!

Ce sont les Inter Process Communication (définis par SUSv4) :

- ▶ pipe
- ▶ file de messages
- ▶ mémoire partagée (et sémaphore pour la synchronisation)

Pipe (1)

Définition

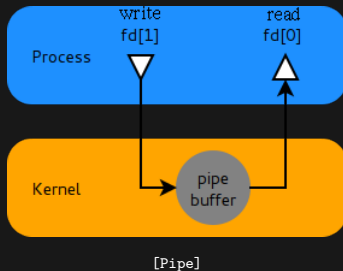
Moyen de communication unidirectionnel.

Pipe (1)

Définition

Moyen de communication unidirectionnel.

Deux extrémités représentées par des file descriptors :

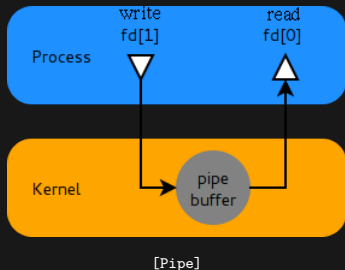


Pipe (1)

Définition

Moyen de communication unidirectionnel.

Deux extrémités représentées par des file descriptors :



=> comme les fd doivent être connus pour l'écriture et la lecture, les pipe sont utilisables avec les threads ou les processus dupliqués (fork)!

Pipe (2)

Comment?

Cinq fonctions :

- ▶ **pipe** : création du tube
- ▶ **write** : écriture dans le tube
- ▶ **read** : lecture des données
- ▶ **close** : fermeture du tube

Pipe (2)

Comment?

Cinq fonctions :

- ▶ **pipe** : création du tube
- ▶ **write** : écriture dans le tube
- ▶ **read** : lecture des données
- ▶ **close** : fermeture du tube

=> une écriture dans un pipe dont l'extrémité est fermée échoue!

Pipe (3)

pipe.c

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

int main()
{
    int pipe_fds[2];
    char * msg = "WATER";

    if (pipe(pipe_fds) != 0)
    {
        perror("pipe");
        return EXIT_FAILURE;
    }

    close(pipe_fds[0]);

    size_t bytes = write(pipe_fds[1], msg, strlen(msg));
    printf("Bytes: %d\n", bytes);
    close(pipe_fds[1]);

    return EXIT_SUCCESS;
}
```

Pipe (3)

pipe.c

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

int main()
{
    int pipe_fds[2];
    char * msg = "WATER";

    if (pipe(pipe_fds) != 0)
    {
        perror("pipe");
        return EXIT_FAILURE;
    }

    close(pipe_fds[0]);

    size_t bytes = write(pipe_fds[1], msg, strlen(msg));
    printf("Bytes: %d\n", bytes);
    close(pipe_fds[1]);

    return EXIT_SUCCESS;
}
```



tergeist@multi: ~

```
tergeist@multi:~$ ./pipe
tergeist@multi:~$ echo $?
141
tergeist@multi:~$ kill -1 13
PIPE
tergeist@multi:~$
```



Pipe (4)

Named pipe

Pour faire communiquer des processus distincts (n'ayant pas accès au file descriptor du tube), il existe des **tubes nommés**.

Pipe (4)

Named pipe

Pour faire communiquer des processus distincts (n'ayant pas accès au file descriptor du tube), il existe des **tubes nommés**.

Contrairement au tube simple résidant en mémoire, un tube nommé possède une représentation au sein du système de fichiers.



Pipe (5)

Named pipe

Six fonctions :

- ▶ **mkfifo** : création du fichier représentant le tube
- ▶ **open** : ouverture du tube
- ▶ **write** : écriture de données dans le tube
- ▶ **read** : lecture des données
- ▶ **close** : fermeture du tube
- ▶ **unlink** : suppression du fichier représentant le tube

```
#include <sys/stat.h>

int mkfifo(const char * pipe_name, mode_t mode);
```

Pipe (6)

named_pipe.c

```
#include <stdlib.h>
#include <sys/stat.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    if (mkfifo("MY_PIPE_NAME", 0644) != 0)
    {
        perror("mkfifo");
        return EXIT_FAILURE;
    }

    // unlink("MY_PIPE_NAME");

    return EXIT_SUCCESS;
}
```

Pipe (6)

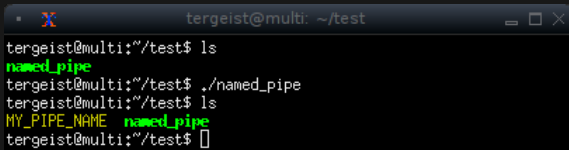
named_pipe.c

```
#include <stdlib.h>
#include <sys/stat.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    if (mkfifo("MY_PIPE_NAME", 0644) != 0)
    {
        perror("mkfifo");
        return EXIT_FAILURE;
    }

    // unlink("MY_PIPE_NAME");

    return EXIT_SUCCESS;
}
```



A terminal window titled "tergeist@multi: ~/test" showing the execution of the program. The user runs "ls" and sees "named_pipe". Then they run "./named_pipe" and run "ls" again, seeing "MY_PIPE_NAME" and "named_pipe".

```
tergeist@multi:~/test$ ls
named_pipe
tergeist@multi:~/test$ ./named_pipe
tergeist@multi:~/test$ ls
MY_PIPE_NAME named_pipe
tergeist@multi:~/test$
```


File de messages (1)

Différence avec les tubes

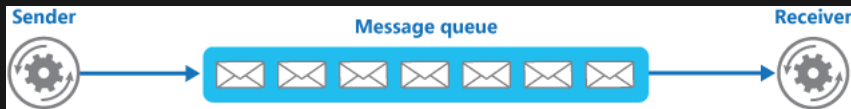
La communication par tubes se fait par l'intermédiaire de flux : la taille des données peut être variable entre chaque écriture/lecture.

File de messages (1)

Différence avec les tubes

La communication par tubes se fait par l'intermédiaire de flux : la taille des données peut être variable entre chaque écriture/lecture.

La communication par file de messages se fait par passage de messages de taille fixe avec des niveaux de priorité!



[Message Queue]

File de messages (2)

Comment?

Les cinq appels système principaux :

- ▶ **mq_open** : ouverture d'une file
- ▶ **mq_close** : fermeture de la file
- ▶ **mq_unlink** : destruction de la file
- ▶ **mq_send** : envoie d'un message dans la file
- ▶ **mq_receive** : réception d'un message

```
#include <mqueue.h>

mqd_t mq_send(mqd_t mq, const char * msg,
              size_t size_msg,
              unsigned int priority);
```

File de messages (3)

mq_server.c / mq_client.c

```
#include <stdlib.h>
#include <mqueue.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>

void send(mqd_t mq, char *msg, int priority)
{
    if (mq_send(mq, msg, strlen(msg), priority) != 0)
    {
        perror("mq_send");
        exit(EXIT_FAILURE);
    }
}

int main()
{
    mqd_t mq = mq_open("/MQ_NAME", O_WRONLY|O_CREAT, 0644, NULL);
    if (mq == (mqd_t) -1)
    {
        perror("mq_open");
        return EXIT_FAILURE;
    }

    send(mq, "MSG1", 0);
    send(mq, "MSG2", 3);
    send(mq, "MSG3", 2);

    return EXIT_SUCCESS;
}
```



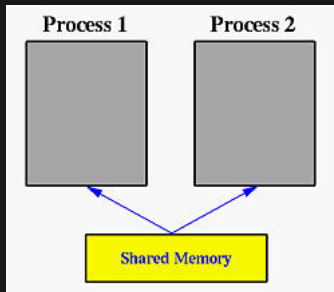
A terminal window titled "tergeist@multi: ~/test" showing the execution of the mq_server and mq_client programs. The user runs ./mq_server, then ./mq_client, which prints "msg 'MSG2' with priority '3'", then ./mq_client again, which prints "msg 'MSG3' with priority '2'", and finally ./mq_client a third time, which prints "msg 'MSG1' with priority '0'".

```
tergeist@multi:~/test$ ./mq_server
tergeist@multi:~/test$ ./mq_client
msg 'MSG2' with priority '3'
tergeist@multi:~/test$ ./mq_client
msg 'MSG3' with priority '2'
tergeist@multi:~/test$ ./mq_client
msg 'MSG1' with priority '0'
tergeist@multi:~/test$ ./mq_client
```

Mémoire partagée (1)

Principe

Shared memory : segment de mémoire accessible en lecture / écriture par plusieurs processus et persistant jusqu'au **reboot** de la machine.



[Shared memory]

Mémoire partagée (2)

Les appels système

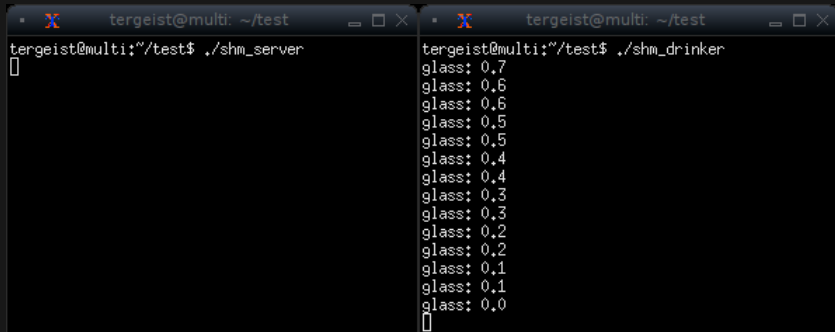
Quatre appels système :

- ▶ **shm_open** : ouverture du segment mémoire
- ▶ **ftruncate** : dimensionnement du segment
- ▶ **mmap** : projection de la structure de données sur le segment
- ▶ **shm_unlink** : destruction du segment

```
int shm_open(const char * name, int flags,  
             mode_t mode);
```

Mémoire partagée (3)

shm_server.c / shm_drinker.c



The image shows two terminal windows side-by-side. The left window has the title 'tergeist@multi: ~/test' and shows the command `./shm_server` being executed, with a single empty line below it. The right window also has the title 'tergeist@multi: ~/test' and shows the command `./shm_drinker` being executed, followed by a list of 14 lines: `glass: 0.7`, `glass: 0.6`, `glass: 0.6`, `glass: 0.5`, `glass: 0.5`, `glass: 0.4`, `glass: 0.4`, `glass: 0.3`, `glass: 0.3`, `glass: 0.2`, `glass: 0.2`, `glass: 0.1`, `glass: 0.1`, and `glass: 0.0`, followed by an empty line.

```
tergeist@multi: ~/test
tergeist@multi:~/test$ ./shm_server
[]

tergeist@multi: ~/test
tergeist@multi:~/test$ ./shm_drinker
glass: 0.7
glass: 0.6
glass: 0.6
glass: 0.5
glass: 0.5
glass: 0.4
glass: 0.4
glass: 0.3
glass: 0.3
glass: 0.2
glass: 0.2
glass: 0.1
glass: 0.1
glass: 0.0
[]
```

=> mais peut avoir des problèmes d'accès concurrent!

Sémaphore (1)

Principe

Les communications multiprocessus à travers la mémoire partagée peut conduire à des incohérences dans les données si les accès ne sont pas synchronisés.

=> contrôle d'accès aux ressources critiques via les sémaphores!

Sémaphore (1)

Principe

Les communications multiprocessus à travers la mémoire partagée peut conduire à des incohérences dans les données si les accès ne sont pas synchronisés.

=> contrôle d'accès aux ressources critiques via les sémaphores!

On peut aussi gérer le nombre d'accès simultanés maximum autorisé.

Sémaphore (2)

flags levés / baissés

L'accès à la donnée :

1. un processus attend que le flag soit levé (ressource disponible)
2. le processus baisse le flag
3. le processus utilise la ressource protégée
4. le processus lève le flag pour indiquer qu'elle est disponible
5. le kernel réveille les autres processus en attente

Sémaphore (3)

versus mutex

Mutex : surtout utilisé dans les applications multithreads (protection d'accès d'une section de code ne pouvant pas être exécutée par plus d'un thread).

Sémaphore : surtout utilisé pour les communications multiprocessus.

Mutex = Sémaphore local avec un nombre de clé de 1 (sémaphore binaire)

```
sem_init(&mutex, 0, 1);
```

Sémaphore (4)

Sémaphore nommé

De même que pour les tubes, un nom peut être associé au sémaphore pour permettre à d'autres processus de l'utiliser!

Sémaphore (4)

Sémaphore nommé

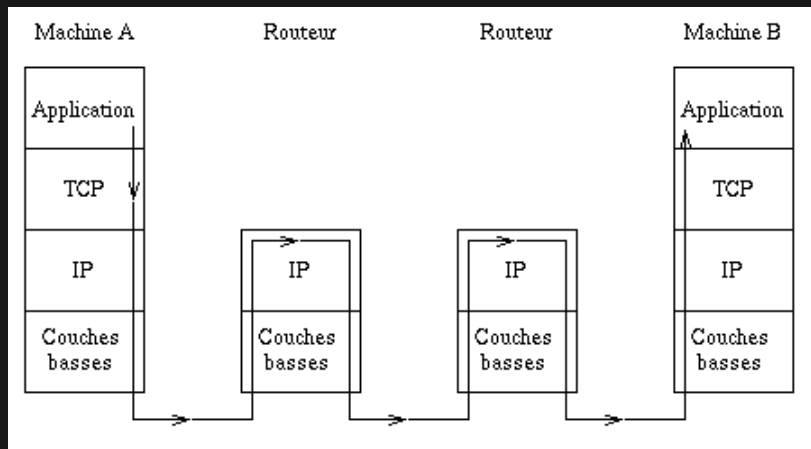
De même que pour les tubes, un nom peut être associé au sémaphore pour permettre à d'autres processus de l'utiliser!

Les sept appels système :

- ▶ **sem_init** : initialisation du sémaphore
- ▶ **sem_open** x 2 : deux prototypes d'ouverture (créateur/utilisateur)
- ▶ **sem_close** : fermeture du sémaphore
- ▶ **sem_unlink** : destruction du sémaphore
- ▶ **sem_wait** : attente bloquante du sémaphore
- ▶ **sem_post** : relache le sémaphore

Programmation réseau (1)

Les couches



[Les couches pour TCP / UDP]

Programmation réseau (2)

Matériels et MAC

Les principaux éléments d'un réseau :

- ▶ **hub** : envoie toutes les données à toutes les machines
- ▶ **switch** : distribue les données aux machines destinataires
- ▶ **routeur** : liaison entre sous réseaux

Programmation réseau (2)

Matériels et MAC

Les principaux éléments d'un réseau :

- ▶ **hub** : envoie toutes les données à toutes les machines
- ▶ **switch** : distribue les données aux machines destinataires
- ▶ **routeur** : liaison entre sous réseaux

Les cartes Ethernet sont représentées par un identificateur unique appelé adresse MAC (Medium Access Control) :

- ▶ 24 premiers bits : numéro constructeur
- ▶ 24 derniers bits : numéro attribué par le constructeur

Programmation réseau (3)

Internet Protocol

Pour permettre la communication entre machines de sous réseaux différents, le protocole IP est utilisé!

Les adresses IP sont représentées sur 4 octets.

Programmation réseau (3)

Internet Protocol

Pour permettre la communication entre machines de sous réseaux différents, le protocole IP est utilisé!

Les adresses IP sont représentées sur 4 octets.

Address Resolution Protocol : trouver l'adresse MAC d'une machine distante dont on ne connaît que l'adresse IP.



Programmation réseau (4)

TCP / UDP et diffusion

Transmission Control Protocol (TCP) :

- ▶ mode connecté : contrôle de flux, acquittement
- ▶ fiable : les données arrivent dans l'ordre (répétées si perdues)

Programmation réseau (4)

TCP / UDP et diffusion

Transmission Control Protocol (TCP) :

- ▶ mode connecté : contrôle de flux, acquittement
- ▶ fiable : les données arrivent dans l'ordre (répétées si perdues)

User Datagram Protocol (UDP) :

- ▶ non connecté : pas d'acquittement
- ▶ non fiable : données non répétées si perdues

Programmation réseau (4)

TCP / UDP et diffusion

Transmission Control Protocol (TCP) :

- ▶ mode connecté : contrôle de flux, acquittement
- ▶ fiable : les données arrivent dans l'ordre (répétées si perdues)

User Datagram Protocol (UDP) :

- ▶ non connecté : pas d'acquittement
- ▶ non fiable : données non répétées si perdues

Diffusion :

- ▶ unicast : envoie d'un message à une IP de classe A, B ou C spécifique
- ▶ broadcast : envoie d'un message à toutes les IP
- ▶ multicast : envoie d'un message à toutes les IP de classe D d'un groupe multicast

Programmation réseau (5)

Port

Plusieurs applications sont disponibles sur un seul serveur. Pour discuter de manière particulière avec une application, un numéro de port est donc nécessaire en plus de l'adresse IP.

Par exemple :

- ▶ ftp : 21
- ▶ ssh : 22
- ▶ http : 80



Programmation réseau (6)

Les commandes utiles

- ▶ **ifconfig** : configuration d'une interface réseau
- ▶ **netstat** : affiche les connexions réseaux, les ports ouverts, ... (machine locale)
- ▶ **nmap** : scanneur de port, exploration du réseau
- ▶ **ssh** : connexion sur machine distante
- ▶ **wireshark** : analyseur réseau
- ▶ ftp, telnet, tcpdump, route, traceroute, ...

<https://debian-handbook.info/browse/fr-FR/stable/sect.network-diagnosis-tools.html>

Programmation réseau (7)

Les sockets

Tube permettant de faire dialoguer des machines distantes via le réseau IP.

Programmation réseau (7)

Les sockets

Tube permettant de faire dialoguer des machines distantes via le réseau IP.

Pour chaque socket, on doit indiquer :

- ▶ Le domaine : IPv4, IPv6 ou local
- ▶ Le protocole : UDP / TCP
- ▶ Le port : int

Programmation réseau (7)

Les sockets

Tube permettant de faire dialoguer des machines distantes via le réseau IP.

Pour chaque socket, on doit indiquer :

- ▶ Le domaine : IPv4, IPv6 ou local
- ▶ Le protocole : UDP / TCP
- ▶ Le port : int

Un socket est manipulé via un file descriptor.

Programmation réseau (8)

Appels système

Côté client TCP :

- ▶ **socket** : création du socket
- ▶ **connect** : connexion à un socket (adresse / port)
- ▶ **send / recv** : discussion
- ▶ **close** : fermeture du socket

Programmation réseau (8)

Appels système

Côté client TCP :

- ▶ **socket** : création du socket
- ▶ **connect** : connexion à un socket (adresse / port)
- ▶ **send / recv** : discussion
- ▶ **close** : fermeture du socket

Côté serveur TCP :

- ▶ **socket** : création du socket
- ▶ **bind** : assigne adresse/port au socket
- ▶ **listen** : attend des connexions
- ▶ **accept** : accepte une connexion (bloquant)
- ▶ **send / recv** : discussion
- ▶ **close** : fermeture du socket

Programmation réseau (9)

Appels système

Côté client UDP :

- ▶ **socket** : création du socket
- ▶ **sendto / recvfrom** : discussion
- ▶ **close** : fermeture du socket

Programmation réseau (9)

Appels système

Côté client UDP :

- ▶ **socket** : création du socket
- ▶ **sendto** / **recvfrom** : discussion
- ▶ **close** : fermeture du socket

Côté serveur UDP :

- ▶ **socket** : création du socket
- ▶ **bind** : assigne adresse/port au socket
- ▶ **recvfrom** / **sendto** : discussion
- ▶ **close** : fermeture du socket

Multiplexage d'entrées (1)

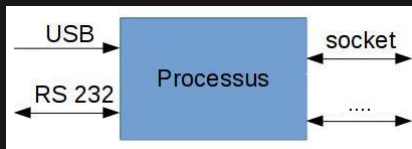
Pourquoi?

Dans un système communiquant, un même processus doit généralement gérer l'arrivée et l'envoi d'informations à travers plusieurs canaux de type différents.

Multiplexage d'entrées (1)

Pourquoi?

Dans un système communiquant, un même processus doit généralement gérer l'arrivée et l'envoi d'informations à travers plusieurs canaux de type différents.



=> l'appel système **select** est utilisé pour le multiplexage!

Multiplexage d'entrées (2)

Concept

Étapes du multiplexage :

1. on indique au kernel les fd que je souhaite écouter
2. tant que rien ne se passe, attente passive (pas de consommation CPU)
3. quand une donnée est disponible, le kernel réveille le processus
4. le processus regarde quel fd est prêt
5. les données sont lues
6. puis on retourne dans l'état d'attente passive jusqu'à la prochaine fois

Multiplexage d'entrées (3)

Les appels systèmes

Cinq appels système principaux :

- ▶ **select** : indique le set de fd à écouter
- ▶ **FD_ZERO** : initialise un set de fd vide
- ▶ **FD_SET** : ajoute un fd dans un set
- ▶ **FD_CLR** : enlève un fd d'un set
- ▶ **FD_ISSET** : vérifie si un fd est présent dans un set

Multiplexage d'entrées (4)

mult.c

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
    fd_set fdset;
    char buff[255];

    int fd = fileno(stdin);
    if (fd < 0)
    {
        perror("open");
        return EXIT_FAILURE;
    }

    while (1)
    {
        FD_ZERO(&fdset);
        FD_SET(fd, &fdset);
        select(fd+1, &fdset, NULL, NULL, NULL);

        if (FD_ISSET(fd, &fdset))
            fgets(buff, sizeof(buff), stdin);
            printf("EVENT: %s", buff);
    }

    return EXIT_SUCCESS;
}
```

Conclusion

Vivement les travaux pratiques
pour apprendre à maîtriser la
force!



Références

- ▶ Linux Embarqué - Pierre Ficheux
- ▶ Développement système sous Linux - Christophe Blaess
- ▶ Modern Operating Systems - Andrew Tanenbaum