



---

# Cours de système

## E/S multiplexées et sockets

### Sébastien Paumier



# Multiplexage

---

- par défaut, les opérations d'E/S sont bloquantes
- problème: comment faire quand on doit attendre simultanément plusieurs E/S différentes ?
- solution n°1: le mode non bloquant
- solution n°2: les sélecteurs



# Mode non bloquant

---

- en étant configurées correctement, certaines opérations habituellement bloquantes peuvent devenir non bloquantes
- ouverture de fichier:
  - utile pour une ouverture d'un tube nommé, quand on ne sait pas encore s'il y a quelqu'un de l'autre côté

`monoinstance2.cpp`



# Mode non bloquant

---

- on utilise **open** avec l'attribut **O\_NONBLOCK**:
  - **open(PIPENAME, ... | O\_NONBLOCK);**
- en écriture, si l'ouverture n'est pas possible sans bloquer, **open** renvoie -1 et positionne errno à **ENXIO**
- l'ouverture non bloquante en lecture marche toujours sur une FIFO



# Mode non bloquant

---

- pour **read** et **write**, on doit passer le descripteur utilisé en mode non bloquant avec **`fcntl(fd, F_SETFD, O_NONBLOCK)`**
- utile sur les descripteurs susceptibles de bloquer: les tubes et les sockets
- exemple: un père qui écoute simultanément plusieurs fils lui parlant chacun par un tube propre

`no_select.cpp`



# Les sélecteurs

---

- problème: on fait de l'attente active :(
- solution: ne faire des E/S que quand on est sûr qu'elles ne bloqueront pas, grâce aux sélecteurs
- un sélecteur permet de surveiller un ensemble de descripteurs:
  - bloque tant qu'il n'y a rien à faire
  - rend la main dès qu'il y a au moins une opération possible sans bloquer



# Les sélecteurs

---

- `int select(int nfd, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);`
- les `fd_set` désignent des ensembles de descripteurs à surveiller:
  - `readfds` = en attente pour une lecture
  - `writefds` = en attente pour une écriture
  - `exceptfds` = en attente d'"événements exceptionnels" (man select *dixit*)



# Les sélecteurs

---

- on manipule les `fd_set` avec les fonctions suivantes:
  - `void FD_CLR(int fd, fd_set *set);`
  - `int FD_ISSET(int fd, fd_set *set);`
  - `void FD_SET(int fd, fd_set *set);`
  - `void FD_ZERO(fd_set *set);`
- on crée donc des ensembles avec les descripteurs à surveiller (NULL=ensemble vide)



# Les sélecteurs

---

- **nfds** doit correspondre au numéro du plus grand descripteur surveillé + 1
- si **timeout** est non NULL, il définit un délai au bout duquel `select` retournera, même en l'absence d'opération possible sur un des descripteurs surveillés
- avant chaque appel à **select**, on doit initialiser l'ensemble à surveiller
- après chaque appel, on teste chaque descripteur avec **FD\_ISSET**



# Les sélecteurs

---

- plus besoin de configurer les descripteurs en mode non bloquant
- pas d'attente active
- retourne le nombre de descripteurs pour lesquelles des opérations sont possibles
- version du programme précédent avec un sélecteur: `select.cpp`
- problème: comment attendre soit une E/S, soit un signal ?



# Sélecteurs et signaux

---

- imaginons le code suivant, censé attendre soit une E/S, soit un signal:

```
void sig_handler(int n) {  
    finished=1;  
}
```

...

```
while (1) {  
    if (finished) break;  
  
    int ret=select(...);  
    if (ret==-1 && errno=EINTR) continue;  
  
}
```

si le signal arrive ici,  
**select** va bloquer quand même!



# Sélecteurs et signaux

---

- solution:
  - 1) bloquer tous les signaux
  - 2) tester la condition
  - 3) appeler **select**, en laissant passer les signaux qui nous intéressent pour qu'ils puissent débloquent **select**
  - 4) débloquent tous les signaux
- pour que ça fonctionne sans problème de concurrence, l'étape 3 doit être atomique, et c'est le rôle de **pselect**



# Sélecteurs et signaux

---

- version sécurisée:

```
sigset_t new_set,old_set;
sigfillmask(&new_set);
sigprocmask(SIG_SETMASK,NULL,&old_set);

while (1) {
    sigprocmask(SIG_SETMASK,&new_set,NULL); // on bloque tout
    if (finished) break;

    int ret=pselect(.....,&old_set); // pselect ne va laisser
                                    // passer que les signaux
                                    // voulus
    if (ret==-1 && errno==EINTR) continue;

    sigprocmask(SIG_SETMASK,&old_set,NULL); // on débloque tout
}
```



# poll/ppoll

---

- **poll/ppoll** est une variante de **select/pselect**
- **int poll(struct pollfd \*fds, nfds\_t nfds, int timeout);**
- même valeur de retour que **select**
- **fds** est un tableau de taille **nfds**
- **timeout** est une valeur en millisecondes; attente infinie si elle est négative



# poll/ppoll

---

- la structure **struct pollfd** est la suivante:

```
struct pollfd {  
    int    fd;          /* Descripteur de fichier */  
    short  events;     /* Événements attendus   */  
    short  revents;    /* Événements détectés  */  
};
```

- **events** est un masque binaire dans lequel on indique ce qu'on attend
- après l'appel, on teste les résultats dans **revents**



# poll/ppoll

---

- les principales valeurs possibles pour **events** sont les suivantes:
  - POLLIN: attente en lecture (ou attente d'une connexion dans le cas d'une socket serveur)
  - POLLOUT: attente en écriture
- pour **revents**, on a aussi POLLUP et POLLRDHUP pour détecter une fin de connexion

*poll.cpp*



# Les sockets

---

- `int socket(int domain, int type, int protocol);`
- permet de créer des points de communication, essentiellement pour des E/S via le réseau
- **domain** définit la nature du protocole:
  - AF\_INET=IPv4
  - AF\_INET6=IPv6
  - etc.



# Les sockets

---

- **type** définit le type de socket:
  - SOCK\_STREAM=support pour les flux de données avec garantie d'intégrité (connexion TCP)
  - SOCK\_DGRAM=transmission de datagrammes de longueur fixe sans garantie (communication UDP)
  - SOCK\_RAW=accès aux trames ethernet brutes
  - etc
- **protocol** vaut toujours 0



# connect

---

- en mode client, une socket TCP doit être connectée à un hôte
- une socket UDP peut l'être aussi pour être en mode pseudo-connecté, mais c'est facultatif
- `int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);`
- `addr` désigne l'adresse de l'hôte



# connect

---

- on utilise le type `struct sockaddr_in` qui a 3 champs importants:
  - `sin_family`: famille de protocole; presque toujours `AF_INET`
  - `sin_port`: le numéro de port de l'hôte, avec l'endianness du réseau
  - `sin_addr`: l'adresse de l'hôte
- l'endianness du réseau se gère avec les fonctions `ntohs`, `ntohl`, `htons` et `htonl`



# connect

---

- on obtient une adresse de type **struct hostent\*** avec **gethostbyname**
- on copie le champ **h\_addr** dans le champ **sin\_addr.s\_addr** de l'adresse:

```
struct hostent* host=gethostbyname(hostname);
if (host==NULL) {
    switch(h_errno) {
        ...
    }
}
memcpy(&addr.sin_addr.s_addr,host->h_addr,host->h_length);
```



# Adresses et interfaces

---

- on peut passer d'une adresse IP à une représentation sous forme de chaîne avec `inet_ntoa` et `inet_aton`
- on peut obtenir la liste des interfaces réseau utilisables avec `ioctl`:

`netint.cpp`



# E/S

---

- en mode connecté/pseudo-connecté, on peut alors lire et écrire avec **read** et **write**

*udp.cpp*

*my\_wget.cpp*

- pour une socket UDP non connectée, on doit utiliser **sendto** et **recvfrom**

*udp2.cpp*



# Sockets serveur

---

- pour des sockets serveur, on doit s'attacher au port qu'on veut écouter avec **bind**
- on utilise l'adresse spéciale **INADDR\_ANY**
- on se met ensuite en écoute des connexions entrantes avec **listen**:
- **int listen(int sockfd, int backlog);**
- **backlog**=taille de la file d'attente pour les demandes de connexion



# Sockets serveur

---

- le serveur entre ensuite dans une boucle infinie dans laquelle il attend les connexions entrantes avec **accept**
- **accept** bloque jusqu'à une connexion, et retourne un descripteur de socket pour communiquer avec le client
- exemple: serveur TCP de caractères aléatoires (à tester avec telnet ou nc)

*rand\_server.cpp*



# Sockets serveur

---

- pour gérer plus d'un client à la fois, on peut utiliser du multithread:

```
rand_server2_multithread.cpp
```

- on peut aussi utiliser un sélecteur, ce qui est plus élégant:

```
rand_server2_poll.cpp
```



# Maîtriser la force

---

- "Maître, TCP et UDP, c'est pour les petits joueurs. Comment devenir un vrai guerrier du réseau pour sniffer les trames ethernet, voir les requêtes ARP, tout ça ?"
- "Des sockets en mode raw, en utilisant, jeune padawan."





# Les sockets raw

---

- réservées à root
- on les crée avec `socket(AF_PACKET, SOCK_RAW, protocol)` où `protocol` désigne la nature des trames qu'on veut capturer:
  - ETH\_P\_ALL: absolument tout
  - ETH\_P\_IP: les paquets IP (mais seulement les entrants, à cause d'un bug du noyau)
  - etc (cf. `linux/if_ether.h`)



# Les sockets raw

---

- une socket raw n'a pas besoin d'être bindée ni connectée
- par défaut, elle voit passer tout le trafic entrant et sortant concernant la machine locale
- mais, on peut avoir tout le trafic qui circule sur le câble en passant en *mode promiscuité*
- exemple:

`raw_socket.cpp`



# libpcap

---

- pour ne pas capturer à la main les différentes sortes de trames ethernet, on peut utiliser la bibliothèque libpcap
- **pcap\_lookupdev**: récupération de l'interface par défaut
- **pcap\_open\_live**: ouverture d'un flux de capture
- **pcap\_next**: capture d'un paquet

`pcap_sniffer.cpp`



# libpcap

---

- on peut éviter de faire une boucle moche en demandant à `pcap_loop` de le faire pour nous
- utilise une fonction de *callback* pour savoir quoi faire d'un paquet reçu

`pcap_sniffer2.cpp`



# Filtrage de paquet

---

- grâce à `pcap_compile` et `pcap_setfilter`, on peut définir un filtre de paquets
- utilise la syntaxe des BSD packet filters, comme `tcpdump`
- exemples:
  - "arp"
  - "udp port 600"

`pcap_filter.cpp`



# Injection de paquets

---

- "Et avec les sockets raw, je peux injecter des paquets forgés à la main, comme par exemple pour faire de l'ARP poisoning ?"
- "Oui, mais vers le côté obscur de <http://www.backtrack-linux.org> cela t'entraînerait..."

