IX - La surveillance de descripteurs

La surveillance de descripteurs

Quelles sont les solutions lorsqu'une application doit effectuer, sans ordre prédéfini, plusieurs opérations susceptibles de bloquer ?

- créer un processus (appel à `fork()`) par opération bloquante.

  Problème : limite sur le nombre de processus en parallèle possibles et le passage d'un processus à l'autre prend du temps, synchronisation difficile.

- créer un processus léger (appel à `pthread_create()`) par opération bloquante.

  Problème : limite sur le nombre de processus légers en parallèle possibles et nécessite une gestion de la mémoire lors des appels concurrents.

- surveiller les descripteurs correspondants et réagir lorsqu'une opération se débloque.

La surveillance de descripteurs

Comment surveiller une ensemble de descripteurs ?

- on peut passer en boucle sur les différentes opérations mises en mode non bloquant

  Problème : attente active qui consomme de la ressource souvent inutilement

- déléguer au système la surveillance des descripteurs

La surveillance de descripteurs

appel bloquant

Les opérations bloquantes :

- `accept`, `read`, `recv`, `recvfrom` sont des opérations bloquantes en lecture
- `connect`, `write`, `send` et `sendto` sont des opérations bloquantes en écriture

  en mode `UDP` :

  - `sendto` commence par recopier **toutes** les données à envoyer dans un tampon du système,
  - si il n'y a pas la place pour tout copier dans ce tampon au moment d'un appel à `sendto`, l'appel bloque jusqu'à ce que le tampon ait été suffisamment vidé pour recevoir toutes les données de l'appel.

La surveillance de descripteurs

appel bloquant

en mode TCP :

- send (ou write) commence par recopier les données à envoyer dans un tampon du système,
- si ce tampon est plein au moment d'un appel à send, l'appel
  - bloque jusqu'à ce que le tampon ait été suffisamment vidé pour recevoir les données de l'appel,
  - ou bloque puis retourne en ayant envoyé une partie des données, par exemple si il a reçu un signal du système.
    → Le retour de send donne le nombre d'octets envoyés et si celui-ci est inférieur à la taille des données, il faut décider si l'envoi doit être complété.

En pratique, il est plus prudent, comme pour la réception, d'utiliser une boucle sur send pour s'assurer que tous les octets ont été envoyés

La surveillance de descripteurs

select

La fonction `select` permet :

- étant donné un ensemble de descripteurs, de bloquer jusqu'à ce qu'un des descripteurs de l'ensemble pointe sur une socket prête en lecture,
- étant donné un ensemble de descripteurs, de bloquer jusqu'à ce qu'un des descripteurs de l'ensemble pointe sur une socket prête en écriture,
- de bloquer jusqu'à ce qu'un temps soit écoulé.

On dit que la socket est **prête en lecture** (respectivement **en écriture**) si l'appel à `accept`, `read`, `recv`, `recvfrom` (respectivement à `connect`, `write`, `send`, `sendto`) sur cette socket n'est pas bloquant.

La surveillance de descripteurs

select

```
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set
*exceptfds, struct timeval *timeout);
```

- `ndfs` est strictement supérieur au plus grand descripteur à surveiller,
- `readfds` est un pointeur sur un ensemble de descripteurs en lecture - peut être `NULL`
- `writefds` est un pointeur sur un ensemble de descripteurs en écriture - peut être `NULL`
- `timeout` est un pointeur sur une variable représentant le temps d'attente maximal ou `NULL` pour un temps infini. (deux champs, `tv_sec` et `tv_usec`, en secondes et microsecondes),
- `exceptfds` sera toujours `NULL`

Les ensembles sont initialisés et modifiés avec :

- `void FD_ZERO(fd_set *set);` initialise un ensemble vide
- `void FD_SET(int fd, fd_set *set);` ajoute `fd` à `set`
- `void FD_CLR(int fd, fd_set *set);` enlève `fd` de `set`

La surveillance de descripteurs

select

L'appel à `select` bloque jusqu'à ce que l'un des situations suivantes survienne :

- un des descripteurs de readfds ou de writefds est « prêt », *i.e.* l'appel à `accept`, `read`, `recv`, `recvfrom` ou à `connect`, `write`, `send`, `sendto` n'est pas bloquant,
- le temps donné par `timeout` est écoulé,
- le processus a reçu un signal.

`select` retourne :

- le nombre de descripteurs prêts,
- `0` si le temps est écoulé,
- `-1` si erreur.

La surveillance de descripteurs

select

- Après l'appel à `select`, les ensembles `readfds` et `writefds` sont modifiés et décrivent les ensembles de descripteurs prêts.

  $\Rightarrow$ nécessité de redéfinir les ensembles `readfds` et `writefds` avant chaque appel à `select`

- `int FD_ISSET(int fd, fd_set *set);` permet de tester si `fd` est dans `set`

- Attention, `select` impose une limite de 1024 (sous linux) sur le nombre maximal de descripteurs à surveiller (consulter la valeur de la variable `FD_SETSIZE`).

  $\Rightarrow$ utiliser `poll` pour ôter cette limite.

La surveillance de descripteurs

checksum

Un message TCP ou UDP est encapsulé dans un paquet IP :

- en IPv4, le paquet IP contient un champs `checksum` permettant de vérifier l'intégrité de l'entête IP
- en IPv6, le paquet IP ne contient pas de champs checksum associé à son entête.

Un message TCP ou UDP contient un champs `checksum` permettant de vérifier l'intégrité de l'entête TCP ou UDP et des données.

La surveillance de descripteurs

lecture sur socket

Lorsque le système reçoit un paquet IP contenant un message TCP ou UDP :

- Il active en lecture le descripteur de la socket `sock` associée
- si l'application bloque sur `select` ou `poll` en attente (entre autre) de la réception d'un message sur `sock` , elle est débloquée et peut aller lire le message,
- pendant ce temps, le système vérifie l'intégrité du message à l'aide du champs `checksum` et si le message est corrompu, il le jette.

La surveillance de descripteurs

lecture sur socket

Conséquence : l'application peut se retrouver bloquée sur la lecture (`read`, `recv`, `recvfrom`) :

- en TCP, cela entrainera un délai car le message sera réémis par l'hôte distant,
- en UDP, l'application peut bloquer indéfiniment ou recevoir un message ne correspondant pas à celui attendu.

Solution : passer la socket en mode non bloquant avec `fcntl`

$\Rightarrow$ `read`, `recv` ou `recvfrom` retourne `-1` si finalement il n'y a rien à lire sur la socket et `errno` vaut `EAGAIN` ou `EWOULDBLOCK`.

La surveillance de descripteurs

mode non bloquant

Pour passer un descripteur en mode non bloquant on utilise la fonction

```
int fcntl(int fd, int cmd, ... /* arg */ );
```

- `fd` : descripteur sur lequel on veut agir, ici notre descripteur de socket
- `cmd` détermine l'opération à effectuer.
  - `F_SETFL` positionne les statuts (le mode d'accès ou les attributs (flags)) du fichier (ici la socket) à la valeur donnée par `arg`.
  - `F_GETFL` pemet que la fonction retourne une valeur donnant les statuts du fichier (ici la socket). `arg` est ignoré.
  - ...
- `arg` est optionnel et `cmd` détermine si une valeur est requise. Valeurs possibles si `cmd` = `F_SETFL` : `O_APPEND`, `O_NONBLOCK`, ...
- retourne une valeur dépendant de la valeur de `cmd`
  - `-1` si erreur,
  - `0` si succès et `cmd` = `F_SETFL`
  - valeur donnant les statuts du fichier si succès et `cmd` = `F_GETFL`

La surveillance de descripteurs

mode non bloquant

En pratique, pour passer une socket `sock` en mode non bloquant,

```
int flags = fcntl (sock, F_GETFL, 0);
if(flags == -1)
  //erreur
else
  if(fcntl(sock, F_SETFL, flags | O_NONBLOCK) == -1)
    //erreur
```

Et pour passer de nouveau la socket en mode bloquant :

```
if(fcntl(sock, F_SETFL, flags) == -1)
  //erreur
```

La surveillance de descripteurs

select

Comment programmer deux applications qui s'échangent simultanément, en mode TCP, une grande quantité de données ?

- TCP est un protocole full-duplex ⇒ on peut indépendemment lire et écrire sur la même socket
- Il faut donc profiter de cela pour éviter les situations de blocage : *l'application A envoie une grande quantité de données à l'application B et simultanément, B envoie une grande quantité de données à A ⇒ appel à send bloque lorsque le buffer d'envoi est plein ⇒ blocage*
- solution : A et B doivent alterner les `send` et `recv` → utiliser `select` pour surveiller le descripteur de la socket de communication en lecture et en écriture et lorsque la socket est prête pour
  - l'écriture, aller faire un send
  - la lecture, aller faire un recv

La surveillance de descripteurs

select

```
fd_set rset, wset, rcpy, wcpy;
FD_ZERO(&rset);
FD_ZERO(&wset);
FD_SET(sock, &rset);  // pour surveillance en lecture de sock
FD_SET(sock, &wset);  // pour surveillance en écriture de sock

int ecrits = 0, cont_lect = 1, cont_ecrit = 1;
while(cont_lect || cont_ecrit){
  if(cont_lect) memcpy(&rcpy, &rset, sizeof(rset)); else FD_ZERO(&rcpy);
  if(cont_ecrit) memcpy(&wcpy, &wset, sizeof(wset)); else FD_ZERO(&wcpy);

  if(select(sock+1, &rcpy, &wcpy, 0, NULL) < 0) exit(1);

  if (FD_ISSET(sock, &wcpy)) {
    r = send(sock, buf_send + ecrits, taille - ecrits, 0);
    if (r < 0) return 1;
    if (r != taille - ecrits) ecrits += r;
    else cont_ecrit = 0;
  }
  if (FD_ISSET(sock, &rcpy)) {
    r = recv(sock, buf_recv, taille, 0);
    if (r < 0) exit(1); else if (r == 0) {cont_lect = 0; continue;}
    // récupérer les données de buf_recv...
  }
}
```

La surveillance de descripteurs

select

Attention à bien réinitialiser les ensembles de descripteurs `rcpy` et `wcpy` avant chaque appel à `select`.

Cette méthode peut également s'appliquer pour gérer plusieurs sockets de communication en même temps :

- si l'application est un serveur, ajouter la socket serveur à l'ensemble de descripteurs à surveiller en lecture,
- ajouter chaque socket de communication aux ensembles de descripteurs à surveiller en lecture et en écriture.
- l'application doit garder en mémoire le nombre d'octets restants à envoyer pour chaque socket de communication.

La surveillance de descripteurs

select

- On peut donc utiliser `select` pour éviter de bloquer sur la lecture et l'écriture sur socket,
- Mais `select` peut bloquer indéfiniment si aucun descripteur ne devient prêt (par exemple si on surveille des socket en lecture et que rien n'arrive)
  $\longrightarrow$ on utilise le 5ème argument de `select`

```
struct timeval timeout;
timeout.tv_sec = 5;
timeout.tv_usec = 0;

ret = select(sock+1, &rfd, NULL, NULL, &timeout);
if(ret > 0) // lire sur les descripteurs de rfd
else if (ret == 0) // 5 secondes ont passées, aucun descripteur prêt,
                   // rien à lire
```

Attention, select peut modifier la variable `timeout`, il faut donc la réinitialiser avant chaque appel à `select`.

La surveillance de descripteurs

select

- On peut donc utiliser `select` pour éviter de bloquer sur la lecture et l'écriture sur socket,
- Mais comment faire lorsque `connect` bloque dans l'attente d'une réponse du serveur ?
  - on peut prendre son mal en patience et attendre jusqu'à ce que le système décide que la connexion n'est pas possible. `connect` retourne alors avec la valeur -1 et `errno` est positionné.
  - ou on décide de stopper l'attente du connect lorsque celle-ci dépasse un certain temps.
    Mais comment fait-on exactement ?
    $\rightarrow$ passer en mode non bloquant et utiliser le timeout de `select`.

La surveillance de descripteurs

interrompre une demande de connexion

→ permet d'interrompre une demande de connexion en attente trop longue.

1. Initialiser l'adresse du serveur, créer la socket de communication et la passer en mode non bloquant

```
struct sockaddr_in6 address_sock;
address_sock.sin6_family = AF_INET6;
address_sock.sin6_port = htons(atoi(argv[2]));
inet_pton(AF_INET6, argv[1], &address_sock.sin6_addr);

int sock = socket(PF_INET6, SOCK_STREAM, 0);

int flags = fcntl(sock, F_GETFL, 0);
if(flags == -1) //erreur
if(fcntl(sock, F_SETFL, flags | O_NONBLOCK) == -1){
  close(sock);
  exit(1);
}
```

La surveillance de descripteurs

interrompre une demande de connexion

2. lancer la demande de connexion

```
int ret = connect(sock, (struct sockaddr *) &address_sock, sizeof(address_sock));
```

3. Si la valeur de retour de `connect` est `0`, la connexion a réussi
   $\rightarrow$ mode non-bloquant, donc peu de chance pour que cela arrive

4. Si la valeur de retour de `connect` est `-1` et `errno` vaut `EINPROGRESS`, on attend maximum `5` secondes que la connexion aboutisse

5. On rétablit le mode bloquant pour la socket

```
fcntl(sock, F_SETFL, flags);
```

La surveillance de descripteurs

interrompre une demande de connexion : étape 4

4. Si la valeur de retour de `connect` est `-1` et `errno` vaut `EINPROGRESS`, on attend maximum `5` secondes que la connexion aboutisse

```
if (ret < 0){
  if (errno == EINPROGRESS){
    fd_set wfd;
    FD_ZERO(&wfd);
    FD_SET(sock, &wfd);

    struct timeval timeout;
    timeout.tv_sec = 5;
    timeout.tv_usec = 0;

    ret = select(sock+1, NULL, &wfd, NULL, &timeout);
    if (ret > 0)                // sûrement connecté - à tester...
    else if (ret == 0)          // échec connexion : ETIMEDOUT
    else                        // échec select
  }
}
```

La surveillance de descripteurs

poll

`int poll(struct pollfd *fds, nfds_t nfds, int timeout);`
se comporte de manière analogue à `select`, mais les paramètres sont passés de manière différente :

- `fds` : pointeur sur un tableau de structure contenant les descripteurs à surveiller
- `nfds` : nombre de descripteurs à surveiller contenus dans `fds`,
- `timeout` : temps d'attente de poll avant de retourner, exprimé en millisecondes. Si vaut `0` ne bloque pas et si vaut `-1` bloque indéfinimment,
- retourne :
  - le nombre d'éléments de `fds` dont le champs `revents` n'est pas nul,
  - ou `0` si le temps a dépassé `timeout` sans événement observé,
  - ou `-1` si erreur.

La surveillance de descripteurs

poll

```
struct pollfd {
  int   fd;
  short events;
  short revents;
};
```

- `fd` : descripteur → si négatif ignoré par `poll`
- `events` : événements à surveiller → `POLLIN`, `POLLOUT`, disjonction
- `revents` : événements observés → `POLLIN`, `POLLOUT`, `POLLHUP`, disjonction

`poll` ne détruit pas ses paramètres ⇒ on peut les réutiliser

`poll` n'a pas de limite sur le nombre de descripteurs qu'il peut gérer. Mais il y a, sous linux, une limite par défaut à 1024 descripteurs ouverts simultanément (cf. `ulimit -a`). Cette limite peut être augmentée si on a les droits (cf. `RLIMIT_NOFILE` et `setrlimit`).

La surveillance de descripteurs

fork vs thread vs select vs poll

`fork`	thread
changement de contexte lourd processus indépendants - robustesse processus pere + 1 processus par client ordonnanceur : système	changement de contexte léger gestion de la concurrence 1 processus multi-threads ordonnanceur : système

`select`	`poll`
pas de changement de contexte gestion du parallèlisme 1 processus pour au plus 1023 clients ordonnanceur : programmeur	pas de changement de contexte gestion du parallèlisme 1 processus pour un grand nombre de clients ordonnanceur : programmeur

utiliser la commande `ulimit -a` pour déterminer le nombre maximal de fichiers ouverts par processus, le nombre maximal de processus par utilisateur, etc...

Les fichiers `/proc/sys/kernel/threads-max` et `/proc/sys/kernel/pid_max` donnent les nombres de processus et de threads que peut gérer, en parallèle, le noyau.