Chapitre 15

Multiplexer des entrées-sorties

15.1 Gerer plusieurs cannaux d'entrée sortie

Dans ce chapitre, nous voulons présenter le problème des attentes actives sur plusieurs descripteurs.

L'exemple le plus fréquent est celui d'un serveur web, le serveur doit gérer simultanément un très grand nombre de flux d'entrée et de flux de sortie et de flux de contrôle (les information de contrôle des sockets).

15.1.1 Solution avec le mode non bloquant

Il est possible d'utiliser des entrée-sorties non bloquantes mais c'est loing d'être la solution optimal car notre processus vas réaliser de nombreux appels système inutile d'autant plus si dans le cas d'un serveur avec des comportements de clients très alléatoires. Le coût en ressources de cette attente active est extrêmement cher, et doit être évité dans le cas d'une machine en temps partagé.

15.1.2 Utiliser les mécanismes asynchrones

On peut utiliser des entrées-sorties asynchrones et demander au noyau de nous prévenir par un signal qui informe de l'arrivée de données sur un descripteur. Ce signal est `SIGIO`, mais ce n'est valable que sur les descripteurs qui sont des périphériques. De plus ce mécanisme ne désigne pas le descripteur sur lequel s'est faite l'arrivée de caractères, d'où de nouvelles pertes de temps dûes aux appels réalisés inutilement en mode non bloquant.

15.2 Les outils de sélection

La solution la plus efficase vient de systèmes de sélection qui prend un paramètre un ensemble de descripteurs, et qui permet tester si l'un de ses descripteurs est près à satisfaire un appel système `read` ou `write`. Cet appel est bloquant jusqu'à l'arrivée de caractères sur un des descripteurs de l'ensemble. Ainsi il n'y pas de consomation de ressource processus inutile, le travail est fait à un niveau plus bas (dans le noyau) de façon plus économique en ressources.

15.2.1 La primitive `select`

La première implémentation d'un outil de selection sous Unix est l'appel système select, malheureusement sa syntaxe est devenu inadapté pour cituations ou le nombre de descipteur utilisé par le programme est très grand ce qui peut arriver facilement avec un serveur de fichier. Nous fournissont à la primitive `select` :

- Les descripteurs que nous voulons scruter. (l'indice du plus grand descripteur qui nous intéresse dans la table des descripteurs du processus)
- Les conditions de réveil sur chaque descripteur (en attente de lecture, écriture, évènement ?)
- Combien de temps nous voulons attendre.

La fonction retourne pour chaque descripteur s'il est prêt en lecture, écriture, ou si l'évènement a eu lieu, et aussi le nombre de descripteur prêts. Cette information nous permet ensuite d'appeler `read` ou `write` sur le(s) bon(s) descripteur(s).

```
#include  <sys/types.h>
#include  <sys/time.h>
#include  <unistd.h>

int select(int maxfd,
           fd_set *readfds,
           fd_set *writefds,
           fd_set *exceptfds
           struct timeval *delai);
```

Retourne le nombre de descripteurs prêts, 0 en cas d'expiration du délai.

Paramétrage du délai :

```
struct timeval {
    long tv_sec;
    long tv_usec;
};
```

`delai == NULL` Bloquant, attente infinie

`delai->tv_sec == 0 && delai->tv_usec == 0` Non bloquant, retour immédiat.

`delai->tv_sec` $>$ `0 && delai->tv_usec` $>$`0` Semi bloquant, attente jusqu'à ce qu'un descripteur soit prêt ou que le délai en secondes plus microsecondes soit écoulé.

Les trois pointeurs (`readfds`, `writefds`, et `exceptfds`) sur des ensembles de descripteurs sont utilisés pour indiquer en entrée les situations qui nous intéressent. C'est à priori (cela peut varier avec l'implémentation) des tableaux de bits avec un bit pour chaque descripteur du tableau de descripteurs du processus. L'entier `maxfd` est la position du dernier bit significatif de ce tableau de bits.

Les seules façons de manipuler ces ensembles de descripteurs sont :

- Allocation :`fd\_set *fd=(fd\_set*)malloc(sizeof(fd\_set));`
- Création
- Affectation
- Utilisation d'une des quatre macros suivantes :

  `FD_ZERO(fd_set fdset)` RAZ de l'ensemble.

  `FD_SET(int fd, fd_set *fdset)` Positionne le bit fd a 1.

  `FD_CLR(int fd, fd_set *fdset)` Positionne le bit fd à 0

  `FD_ISSET(int fd, fd_set *fdset)` vrai si le bit fd est à 1 dans l'ensemble.

Un descripteur est considéré comme prêt en lecture si un appel `read` dessus ne sera pas bloquant. De même, un descripteur est considéré comme prêt en écriture si un appel `write` ne sera pas bloquant. Les exceptions / évènements sont définis pour les lignes de communication qui acceptent les *messages hors bande* comme les `sockets` en mode datagramme.

15.2.2 La primitive `poll`

La primitive `poll` fournit un service proche de `select` avec une autre forme d'interface. Cette interface est adaptée quand le nombre de descripteurs ouvert par le processsus est très grand mais que l'on ne s'intérèsse qu'a un petit nombre de ceux-ci.

```
#include <stropts.h>
#include <poll.h>
int poll(struct pollfd fdarray[],
         unsigned long nfds,
         int           timeout
  );

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

Ici on spécifie la liste de descripteurs (dans un tableau) et ce que l'on veut gèter sur chacun d'eux.

La valeur de retour est -1 en cas d'erreur, 0 si le temps d'attente `timeout` est écoulé, ou un entier positif indiquant le nombre de descripteurs pour lesquels la valeur du champ `revents` a été modifiée.

Les évènements sont ici :
Pour les évènements de `events` :

POLLIN Données non prioritaire peuvent être lues.

POLLPRI Données prioritaire peuvent être lues.

POLLOUT Données non prioritaire peuvent être écrites, les messages de haute priorité peuvent toujours êtres écrits.

Pour les `revents` (valeurs de retour de la primitive `poll`) :

POLLIN,POLLPRI les données sont là.

POLLOUT l'écriture est possible

POLLERR Une erreur a eu lieu.

POLLHUP La ligne a été coupée.

POLLNVAL Descripteur invalide.

Le mode de blocage de la primitive `poll` dépend du paramètre `timeout`

timeout == INFTIM Bloquant, INFTIM est défini dans `stropts.h`.

timeout == 0 Non bloquant.

timeout > 0 Semi bloquant, attente de `timeout` micro secondes.

Un Exemple Attente de données sur ifd1 et ifd2, de place pour écrire sur ofd, avec un délai maximum de 10 seconds :

```
#include <poll.h>
struct pollfd fds[3];
int ifd1, ifd2, ofd, count;

fds[0].fd = ifd1;
fds[0].events = POLLIN;
fds[1].fd = ifd2;
fds[1].events = POLLIN;
fds[2].fd = ofd;
```

```
fds[2].events = POLLOUT;
count = poll(fds, 3, 10000);
 if (count == -1) {
        perror("poll failed");
         exit(1);
 }
 if (count==0)
       printf("Rien \n");
 if (fds[0].revents & POLLIN)
       printf("Données a lire sur ifd%d\n", fds[0].fd);
 if (fds[1].revents & POLLIN)
       printf("Données a lire sur ifd%d\n", fds[1].fd);
 if (fds[2].revents & POLLOUT)
       printf("De la place sur fd%d\n", fds[2].fd);
```

15.2.3 Le périphérique poll

Dans le cas de serveur travaillant avec un très grand nombre de descripteurs (plueirus dizaine de milliers de descripteurs) les deux syntaxes `poll` et `select` sont inéfficaces. Soit dans le cas de`select` car le nombre de descripteurs scrutés par le noyau est très grand alors qu'un très faible par d'entre eux sont inutilisé. Soit dans le cas de `poll` car il faut manipuler avant chaque appel un très grand tableau et que le système doit relire ce tableau a chaque appel.

Pour résoudre ce problème une nouvelle interface a été mise au point `/dev/poll`. Cette interface permet de créer un périphérique poll dans lequel il suffit d'écrire pour ajouter un descripteur a la liste des descipteurs que le noyau doit scruter. Et il suffit d'écrire de nouveau pour retirer un descripteur.

Epoll est un patch du noyau ![1]

1. Il faut verifier la présence d'epoll sur votre système, ouvrer le périphérique `/dev/epoll` en mode O_RDWR, sinon retour a `select` et `poll kdpfd = open("/dev/epoll",O_RDWR);`
2. Définiser le nombre maximal `maxfd` de descipteurs scrutables `#include <linux/eventpoll.h> \ldots ioctl(epo`
3. Allouer un segment de mémoire partagé avec `char *map = (char *)mmap(NULL, EP_MAP_SIZE(maxfds, PROT_READ | PROT_WRITE, MAP_PRIVATE, epoll_fd, 0))`
4. Maintenant vous pouvez ajouter des descipteurs

```
struct pollfd pfd;
pfd.fd = fd;
pfd.events = POLLIN | POLLOUT | POLLERR | POLLHUP;
pfd.revents = 0;
if (write(kdpfd, &pfd, sizeof(pfd)) != sizeof(pfd)) {
/* gestion d'erreur */
}
```

5. Récupere les évenements

```
struct pollfd *pfds;
struct evpoll evp;

for (;;) {
        evp.ep_timeout = STD_SCHED_TIMEOUT;
        evp.ep_resoff = 0;

        nfds = ioctl(kdpfd, EP_POLL, &evp);
```

[1] Il existe deux version une version /Dev/poll et une version /dev/epoll qui faut utiliser car plus efficasse.

```
                pfds = (struct pollfd *) (map + evp.ep_resoff);
                for (ii = 0; ii < nfds; ii++, pfds++) {
                        traitmement(pfds[ii].fd, pfds[ii].revents);
                }
        }
```

6. Retirer des descripteurs

```
pfd.fd = fd;
pfd.events = POLLREMOVE;
pfd.revents = 0;
if (write(kdpfd, &pfd, sizeof(pfd)) != sizeof(pfd)) {
/* gestion d'erreur */
}
```

Le petit détail technique génial de cette interface est le fait que pendant que vous récupere des évenements le système continu a travailler pour vous dans le segment de mémoire fournis par `mmap` ce qui fait que votre programme s'exécute en parallèle de la récupération d'information sur les périphériques et que l'appel `ioctl` est ainsi très rapide.

15.2.4 Les extensions de `read` et `write`

Une extension `readv, writev` de `read` et `write` permet en un seul appel système de réaliser l'écriture de plusieurs zones mémoire non contiguës, ce qui permet d'accélerer certaines entrées-sorties structurées. Mais aussi de mieux organiser les appels système dans notre cas.

```
#include <sys/types.h>
#include <sys/uio.h>

ssize_t readv(int fd, const struct iovec iov[], int iovl);
ssize_t writev(int fd, const struct iovec iov[], int iovl);

struct iovec {
    void *iov_base ;
    int   iov_len;
};
```

15.3 une solution multi-activités

L'utilisation de plusieurs activités (threads, voir chapitre 16) permet de réaliser plusieurs appels de `read` en simultané, le premier `read` qui se débloque entraine l'exécution de l'activité le réalisant, ainsi le coût d'attente sur les descripteurs est minimal le système signalant imédiatement à la thread l'évenement. Le seul problème est d'avoir a gérer cette multiplicité d'activités, ce qui est dans le cas d'un simple echange bidirectionel est raisonable il suffit de deux activités indépendantes.

Pour un serveur de fichier cette solutions multi-activité peut ne pas supporter la monté en charge le nombre de threads étant limité plus rapidement que celui des descripteurs (a ma connaissance ont peut créer au plus 10000 threads sous linux et xp)

Pour une situation plus complexe comme un serveur de partage de données, des mécanismes d'exclusion mutuelle entre activités devront être mis en oeuvre, ce qui peut compliquer inutilement le problème. La solution avec un seul processus gérant rapidement des requêtes multiples sur plusieurs flux étant plus simple a réaliser.