Programmation Système VII
Boucles à événements

Juliusz Chroboczek

19 mars 2016

1 Introduction

Beaucoup de programmes ont la structure suivante :

```
while(1) {
    attendre un événement
    réagir à l'événement
}
```

Par exemple, un programme à interface graphique attend une interaction de l'utilisateur (un clic de souris etc.), détermine à quel objet de l'interface utilisateur cette interaction s'adresse, et réagit en invoquant la bonne méthode de cet objet. De même, un serveur réseau attend l'arrivée d'une requête, puis réagit en répondant à cette requête.

Une telle structure s'appelle une *boucle à événements (event loop)*.

2 L'appel système `select`

Une boucle à événements est normalement construite sur l'appel système `select`[1]. Cet appel système a le prototype suivant :

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

Un appel à `select` bloque jusqu'à ce qu'un des descripteurs dans `readfds` soit prêt en lecture, ou qu'un des descripteurs dans `writefds` soit prêt en écriture, ou qu'un des descripteurs dans `exceptfds` ait une exception, ou que l'intervalle de temps spécifié par `timeout` soit écoulé. Le paramètre `nfds` spécifie le numéro du plus grand descripteur intéressant plus un. L'appel système `select` retourne le nombre de descripteurs prêts en cas de succès, -1 en cas d'erreur. Comme tout

1. Ou l'appel système équivalent `poll`. Sous Windows, `select` est terriblement inefficace, il faut utiliser d'autres techniques, comme les *ports de complétion* qui, eux, passent à l'échelle d'une façon merveilleuse.

appel système bloquant, `select` est interrompu par la livraison d'un signal (et alors il retourne -1 avec `errno` valant `EINTR`).

La figure 1 montre une copie bidirectionnelle entre deux descripteurs de fichiers effectuée en style boucle à événements à l'aide de `select`. Vous remarquerez notamment la gestion de `EINTR` (lors de `select`, `read` et `write`) ainsi que la gestion des écritures partielles (le cas de `write` retournant moins que le nombre d'octets demandé).

3 Boucles à événements et signaux

Une boucle à événements a souvent besoin de gérer des signaux, par exemple pour faire le ménage avant de terminer le programme, pour afficher des informations (traditionnellement dédié à `SIGUSR1`) ou pour relire les fichiers de configuration (traditionnellement dédié à `SIGUSR2`). Une approche naïve à la gestion de `SIGINT` pourrait se faire comme dans la figure 2, en utilisant la technique de conversion en notification synchrone vue au cours précédent.

On remarque que ce programme souffre d'une *race condition* : si le gestionnaire de signal est exécuté entre le test de `please_quit` et l'entrée dans `select`, ce dernier bloque indéfiniment.

4 Déblocage atomique des signaux

La solution au problème souligné ci-dessous consiste à bloquer les signaux d'intérêt avant de tester la variable globale. Mais comment peut-on les débloquer ?

- Si on les débloque avant d'entrer dans `select`, la *race condition* existe encore ;
- si on les débloque après `select`, ce dernier ne sera plus interrompu en cas d'arrivée du signal.

Il est donc nécessaire de débloquer le signal et d'entrer dans `select` de façon atomique, ce qui est fait par l'appel système `pselect` :

```
int pselect(int nfds, fd_set *readfds, fd_set *writefds,
            fd_set *exceptfds, const struct timespec *timeout,
            const sigset_t *sigmask);
```

Cet appel système est analogue à `select`, sauf qu'il effectue l'équivalent de `sigprocmask(SIG_SETMASK)` avant de bloquer ; si le signal débloqué était en attente, `pselect` retourne immédiatement -1 avec `errno` valant `EINTR`.

Le fragment de code de la figure 3 résoud la *race condition* du programme précédent.

5 Descripteurs de fichiers non-bloquants

Les programmes des paragraphes précédents utilisent des descripteurs de fichiers bloquants. En pratique, une boucle à événements utilise des descripteurs de fichiers non-bloquants, ce qui permet d'éviter des *deadlocks* au cas où `select` ou `pselect` retourne une notification incorrecte, ce qui peut notamment être le cas avec des *sockets* UDP. Voyez la partie 4 du cours de L3 pour plus d'informations.

```
while(1) {
    fd_set fds;
    int rc;
    FD_ZERO(&fds);
    FD_SET(fd1, &fds);
    FD_SET(fd2, &fds);
    rc = select(max(fd1, fd2) + 1, &fds, NULL, NULL, NULL);
    if(rc < 0) {
        if(errno == EINTR)
            continue;
        abort();
    }
    if(FD_ISSET(fd1, &fds) {
        int offset, rc2;
        unsigned char buf[512];
        rc = read(fd1, buf, 512);
        if(rc <= 0) {
             if(rc < 0 && errno == EINTR)
                 continue;
             break;
        }

        offset = 0;
        while(offset < rc) {
            rc2 = write(fd2, buf + offset, rc - offset);
            if(rc2 < 0) {
                if(errno == EINTR)
                    continue
                /* La flemme de gérer l'erreur. */
                abort();
            }
            offset += rc2;
        }
    }

    if(FD_ISSET(fd1, &fds) {
        ...
    }
}
```

Figure 1 — Une copie entre deux descripteurs en style boucle à événements

```
volatile sig_atomic_t please_quit = 0;

void
quit_handler(int signo)
{
    please_quit = 1;
}

...
memset(&sa, 0, sizeof(sa));
sa.sa_handler = quit_handler;
sa.sa_flags = 0;
rc = sigaction(SIGINT, &sa, NULL);
...
while(1) {
    fd_set fds;
    int rc;

    if(please_quit)
        break;

    /* Si le signal arrive ici, on bloque dans select. */

    FD_ZERO(&fds);
    FD_SET(fd1, &fds);
    FD_SET(fd2, &fds);
    rc = select(max(fd1, fd2) + 1, &fds, NULL, NULL, NULL);
    ...
}
```

Figure 2 — Boucle à événement avec gestion de signal. Ce programme souffre d'une *race condition* qui peut causer un blocage de durée non-bornée.

```
while(1) {
    fd_set fds;
    sigset_t old, new;
    int rc;

    sigemptyset(&new);
    sigaddset(&new, SIGINT);
    rc = sigprocmask(SIG_BLOCK, &new, &old);
    if(rc < 0) abort();

    if(please_quit)
        break;

    FD_ZERO(&fds);
    FD_SET(fd1, &fds);
    FD_SET(fd2, &fds);
    rc = pselect(max(fd1, fd2) + 1, &old,
                 &fds, NULL, NULL, NULL);
    ...
}
```

Figure 3 — Boucle à événement avec gestion de signal sans *race condition*.

6 Boucles à événements en pratique

En pratique, une boucle à événements manipule plusieurs structures de données :

- l'ensemble des descripteurs de fichiers « intéressants » et les gestionnaires d'événements associés ;
- une file de priorité *(priority queue)* des *timeouts* et les gestionnaires de fichiers associés, souvent représentée par un tas *(heap)*.

Les paramètres de `select` et `pselect` sont calculés à partir de ces structures de données, et peuvent changer à chaque itération. Le code exécuté après la sortie de `select` est déterminé dynamiquement.

S'il est possible d'écrire sa propre boucle à événements, on se sert souvent de bibliothèques déjà faites. Si on utilise une interface graphique, la boucle à événements est généralement imposée : par exemple, un programme Java qui a créé des composants graphiques invoque automatiquement la boucle à événements de *AWT* dans un *thread* séparé, tandis qu'un programme *Gtk+* invoque une boucle à événements à l'aide de la fonction `gtk_main`.

Pour les applications réseau, il existe deux boucles à événements dont j'ai entendu dire du bien : `libevent` et `libev`.