

Sviluppo di software di rete con l'utilizzo di sistemi UNIX

IO con multiplexing

Claudio Vicari

Definizioni

- ➔ modelli di I/O in Unix:
 - bloccante
 - non bloccante
 - con multiplexing (select, poll)
 - con segnali (SIGIO)
 - asincrono

Definizioni

⇒ I/O bloccante:

- è il modello più comune, che abbiamo usato sinora. Quando si esegue una chiamata di sistema bloccante, il flusso di esecuzione si interrompe, fino al completamento dell'operazione

⇒ I/O non bloccante:

- le primitive di I/O ritornano subito. In questo caso, per ricevere dati bisogna ciclare per controllarne l'arrivo (*polling*)

Definizioni

⇒ I/O con segnali:

- istruiamo il kernel ad avvisarci con un SIGIO quando i dati da un particolare dispositivo sono pronti per una lettura. In questo modo, il processo non si blocca

⇒ I/O asincrono:

- è un nuovo sistema, descritto nelle estensioni cosiddette *realtime* di Posix. Si usa la funzione *aio_read* per istruire il kernel sull'operazione da eseguire e sul tipo di notifica da ricevere. La notifica avviene **dopo la ricezione** dei dati (a differenza di SIGIO)

IO con multiplexing: usi possibili

- ➔ anche questo è un sistema con cui il kernel notifica un processo, quando alcune condizioni specificate di I/O sono pronte
 - es: un dispositivo pronto per la lettura (read), oppure che è in grado di ricevere nuovi dati (con write)
- ➔ nelle applicazioni di rete, si usa per:
 - un client che gestisce più di un file descriptor (per esempio, da console e su socket)
 - un client che gestisce più di un socket (raro)
 - un server che gestisce i socket in listen nonché i socket relativi ai client
 - un server che usa sia UDP che TCP
 - un server che gestisce più servizi, come **inetd** (su Linux, provare `strace xinetd -dontfork`)

IO con multiplexing: funzionamento

- ➔ normalmente, dovendo leggere dei dati, ci si blocca in attesa che siano disponibili
- ➔ possiamo usare le system call *select* o *poll*. Queste si bloccano, in attesa che sia possibile leggere da almeno uno dei socket a disposizione
- ➔ quindi, ci si ferma su *select*, poi quando questa esce possiamo fare una *read*
 - lo svantaggio, è che dobbiamo fare due chiamate di sistema invece che una sola...

La funzione select

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

- ➡ con questa funzione, il processo chiamante fornisce al kernel un elenco di possibili eventi di I/O, in modo che:
 - alla chiamata di select, il processo si sospenda
 - il processo venga svegliato al verificarsi di uno o più di questi eventi, o dopo la scadenza di un timeout prefissato
- ➡ il valore di ritorno è 0 se non è accaduto nulla, altrimenti ci indica il numero di eventi avvenuti
- ➡ attenzione: normalmente timeout è const, ma non per Linux...

La funzione select: specificare gli eventi

- ➡ gli eventi possibili sono:
 - qualcuno dei descrittori nell'insieme `readfds` è pronto per una operazione di lettura (`read`)
 - qualcuno di quelli in `writfds` è pronto per un'operazione di scrittura (`write`)
 - qualcuno di quelli in `exceptfds` ha una condizione di *eccezione* in attesa
 - il timeout specificato da `timeout` scade

Specificare il timeout

- ➡ si utilizza una struttura di tipo timeval:

```
struct timeval {  
    long    tv_sec;    /* seconds */  
    long    tv_usec; /* microseconds */  
};
```

- ➡ se usiamo NULL nella chiamata a funzione, non viene impostato alcun timer e la select attende per sempre
- ➡ se specifichiamo valori validi, la select esce comunque allo scadere del timeout
- ➡ se i valori nella struttura sono pari a 0, la select esce subito in ogni caso, ma solo dopo aver controllato lo stato dei descrittori

La funzione select: manipolare gli insiemi

- ➔ il tipo di dati fdset è tipicamente implementato come una mappa di bit, ma abbiamo a disposizione delle macro per utilizzarlo senza preoccuparsi dei dettagli...
- ➔ macro disponibili:
 - `FD_ZERO(fd_set *set);` *// pulisce l'insieme*
 - `FD_SET(int fd, fd_set *set);` *// aggiunge fd*
 - `FD_CLR(int fd, fd_set *set);` *// rimuove fd*
 - `FD_ISSET(int fd, fd_set *set);` *// fd è in set?*
- ➔ i tre insiemi passati a select, dopo la chiamata di select conterranno solamente i file descriptor su cui è avvenuto un evento. Dobbiamo quindi salvare gli insiemi prima della chiamata. L'operazione di assegnamento (`a=b`) fra insiemi di tipo `fd_set` è valida

Ancora su select

- ➔ `select` può essere interrotta da un segnale! Comunque *errno* lo indica...
- ➔ gli insiemi sono tipicamente rappresentati come mappe di bit, cioè l' n -esimo bit è pari ad 1 se il fd $n-1$ è nell'insieme
- ➔ il kernel, quindi, esamina tutti i file descriptor da 0 fino ad un limite, controllando se sono nell'insieme
- ➔ il primo argomento della funzione (n) indica proprio il valore del massimo file descriptor che la funzione deve esaminare, +1. Cioè, il numero di file descriptor che il kernel deve esaminare
- ➔ la costante `FD_SETSIZE` indica qual è il numero di descriptor per il tipo `fd_set` – generalmente è 1024

Condizioni per gli eventi di lettura

- ➔ i file regolari sono poco problematici, vediamo le condizioni per i socket
- ➔ un socket è pronto per la lettura se una di queste condizioni è vera:
 - ci sono abbastanza byte da leggere ($>$ del valore di low-water-mark, generalmente 1 per TCP e UDP)
 - la semiconnessione in lettura è chiusa (in TCP, accade quando si riceve un FIN). In questo caso, una read leggerà 0 byte (cioè EOF)
 - il socket è un listening socket, e ci sono nuove connessioni pronte (quindi accept non blocca)
 - c'è un errore sul socket. In questo caso, read restituirà un valore di -1

Condizioni per gli eventi di scrittura

- ➔ un socket è pronto per la scrittura se:
 - il buffer ci permette di scrivere (per protocolli connessi), oppure il socket usa un protocollo non connesso
 - la semiconnessione in scrittura è chiusa. In questo caso, una write genererebbe SIGPIPE
 - c'è un errore sul socket. In questo caso, una write ci restituirà immediatamente -1 (con *errno*)

Condizioni per gli eventi generati da eccezioni

- ➔ un socket è in condizione di eccezione in attesa, se:
 - ci sono dati fuori banda in attesa (particolarità di TCP)
 - c'è presenza di informazioni di controllo per uno pseudo terminale (che non abbiamo trattato)

Esempio: client

- ➔ questo client fa multiplexing fra la lettura dell'input da tastiera, ed un file descriptor passato come parametro (il main gli fornisce un socket TCP)
- ➔ in questo modo, non si blocca inutilmente ad aspettare input da uno dei descriptor

`select/strclselect01.c`

Problema sul client

- ➔ questo client funziona correttamente se l'input avviene da tastiera... ma cosa accade se l'input avviene con una redirectione?
 - `echo ciao | ./tcpcli01 127.0.0.1` oppure
 - `cat /tmp/input | ./tcpcli01 127.0.0.1 > /tmp/output`
- ➔ il problema è che, in `str_cli`, quando incontriamo un EOF su `stdin`, torniamo al main e quindi usciamo! Alcuni dati, quindi, possono non essere stati ancora ricevuti, e quindi non verranno emessi in output

Soluzione: utilizzo di EOF e shutdown

- ➔ la seconda versione del client controlla se lo standard input o il socket hanno ricevuto EOF
- ➔ inoltre, usa shutdown per imporre l'invio di un FIN verso il server, quando troviamo EOF sull'input
- ➔ attende comunque che il server abbia finito di rispondere, prima di terminare
- ➔ adesso, gli esempi della pagina precedente funzionano correttamente

`select/strcliselect02.c`

La funzione pselect

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

- ➔ è identica a select, ma usa una struttura diversa per descrivere il tempo (con risoluzione al nanosecondo!)
- ➔ se ci sono dei segnali bloccati, pselect temporaneamente li sblocca (quindi i *signal handler* possono gestirli mentre pselect attende un evento)
- ➔ all'uscita, pone la maschera dei segnali a sigmask, e restituisce il controllo come per select

Esempio per pselect

- ➔ pensiamo ad un processo che attende un segnale specifico, per un'azione... il signal handler, dunque, attiva un flag apposito:

```
// ( blocca il segnale )  
1. if( intr_flag ) handle_intr();  
2. if( n_ready == select() ) < 0 ) {  
3.     if( errno == EINTR ) {  
4.         if( intr_flag )  
5.             handle_intr();  
6.     }  
7. }
```

- ➔ in questo caso, se il segnale occorresse fra la riga (1) e la (2), ci si bloccherebbe in select! Con pselect, bloccando il segnale dove indicato, non avremmo problemi

La funzione poll

```
int poll( struct pollfd *fdarray,  
          unsigned long nfd,  
          int timeout );
```

- ➔ funziona similmente a select
- ➔ la struttura pollfd permette di specificare, per ogni *fd*, quali eventi controllare. Al ritorno, in un altro campo troveremo anche gli eventi occorsi
- ➔ nfd è il numero di strutture nell'array
- ➔ timeout è espresso in millisecondi
- ➔ per un esempio, vedere tcpcliserv/tcpservpoll01.c