Modulo 10
System call per I/O avanzato

Laboratorio di Sistemi Operativi I
Anno Accademico 2008-2009

Non blocking I/O e I/O Multiplexing: Introduzione

- Le system call Unix/Linux che riguardano le operazioni di I/O possono essere del tipo "**slow system calls**" se il processo chiamante può essere bloccato su tali chiamate.
- Esempi:
  - `read`: il processo chiamante può bloccarsi su un'operazione di read se i dati non sono presenti (pipe, terminal device, network device);
  - `write`: il processo chiamante può bloccarsi su un'operazione di write se i dati non possono essere accettati (pipe, terminal device, network device);
  - `open`: il processo può bloccarsi in fase di apertura se certe condizioni non si sono verificate
    - Chiamata open su FIFO in sola lettura (senza flag `O_NONBLOCK`) quando nessun altro processo ha aperto il FIFO in scrittura

Slow system call e Segnali

- Un processo bloccato su una **slow system call** può essere interrotto dalla consegna di un segnale al processo stesso
- La system call termina con un errore e la variabile `errno` assume il valore `EINTR`
- In queste situazioni la gestione della slow system call deve essere fatta in modo tale da ripeterla.

```
do {
  again = 0;
  if ((n = read(fd, buf, SIZE)) < 0) {
    if (errno == EINTR) { again = 1; }
    else { ... }
  }
  else { /* no error */ }
} while (again);
```

Non blocking I/O

- E' possibile impostare le modalità di I/O per un file descriptor in modo che non siano bloccanti
  - Nella chiamata `open()` si può impostare il flag `O_NONBLOCK`

    open(fifoname, O_RDONLY | O_NONBLOCK);
  - Nel caso di file descriptor già aperti si può impiegare la funzione `fcntl()` per impostare il flag `O_NONBLOCK`

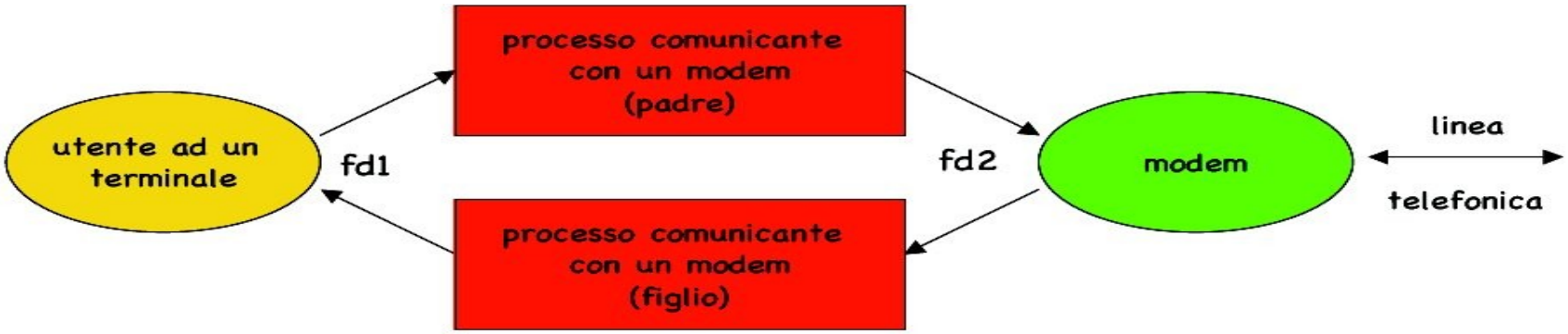
    fcntl(fd, F_SETFL, O_NONBLOCK)
- Con file descriptor in modalità non bloccante:
  - Se l'I/O e' pronto i dati vengono letti o scritti
  - Se l'I/O non e' pronto la chiamata non blocca il processo ma ritorna immediatamente
    - return status **-1** ed `errno == EAGAIN`

I/O Multiplexing (1)

- Supponiamo che un processo debba leggere da 2 file descriptor, in questo caso l'impiego di operazioni bloccanti in lettura rischierebbe, ad esempio, di bloccare il processo sul *fd1* perché non ci sono dati disponibili e impedire così la lettura da *fd2* sul quale possono esserci dati disponibili.
- Esempio:
  - Il processo legge da *fd1* e scrive su *fd2*
  - Il processo legge da *fd2* e scrive su *fd1*
  - Il processo può rimanere bloccato sulla prima `read()` (*fd1*) impedendo la lettura da *fd2*

I/O Multiplexing (2)

- Una possibile soluzione è di utilizzare due processi (`fork()`)



- Ognuno dei due processi gestisce la comunicazione in una sola direzione
- Se il figlio termina perché il processo remoto disconnette la linea, il processo padre ne viene a conoscenza perché riceve un segnale `SIGCHLD`
- Se il padre termina perché l'utente disconnette il terminale, il padre deve comunicarlo al figlio (es. invia segnale `SIGUSR1`)
- La struttura del programma diventa più complicata

I/O Multiplexing (3)

- Soluzione alternativa con un solo processo:

  (1) Imposto i file descriptor in modalità non-bloccante;

  (2) Eseguo una `read(fd1,...)`. Se ci sono dati li leggo e li scrivo sul descrittore *fd2*. Se non ci sono dati la chiamata ritorna immediatamente.

  (3) Eseguo una `read(fd2,...)`. Se ci sono dati li leggo e li scrivo sul descrittore *fd1*. Se non ci sono dati la chiamata ritorna immediatamente.

  (4) Attendo un po' di tempo e riprendo dal punto (2)

- Questa tecnica è nota come **polling**
  - Spreco del tempo di CPU (*CPU time*) a scapito dell'efficienza totale del sistema (utilizzo non ottimale delle risorsa CPU).

La system call select()

- In molti casi la soluzione migliore per l'I/O multiplexing è fornita dalla system call `select`, che permette il monitoraggio simultaneo di un set di file descriptor, rimanendo in attesa finché uno o più di essi diventano pronti per l'operazione di I/O prevista.

- **Argomenti della system call `select()`:**
  - Quali file descriptor ci interessano
  - Quali condizioni si devono verificare per ciascun descrittore
    - Vogliamo leggere dal file descriptor
    - Vogliamo scrivere sul file descriptor
    - Siamo interessati ad una condizione di eccezione di un file descriptor
  - Quanto tempo vogliamo attendere
    - Infinito (chiamata bloccante)
    - Per un certo tempo (timeout)
    - Polling (chiamata non bloccante)

La system call `select()`

- **La system call `select()` restituisce:**
  - Il numero totale di descrittori che sono pronti per l'I/O
  - Quali descrittori sono pronti per ciascuna delle tre condizioni
    - *read*: la chiamata `read` sul descrittore pronto non bloccherà il processo
    - *write*: la chiamata `write` sul descrittore pronto non bloccherà più il processo
    - *exception condition*: si è verificata una condizione di eccezione sul descrittore (es. out-of-band)
  - Queste informazioni ci permettono di eseguire le appropriate funzioni di I/O senza bloccare il processo

La system call select()

- **Valore di ritorno della system call**
  - *n* > 0 : *n* descrittori sono pronti per l'I/O
  - 0 : Timeout scaduto
  - -1 : Si e' verificato un errore. La variabile *errno* può assumere i seguenti valori:
    - EBADF: C'è un descrittore non valido tra gli argomenti
    - EINTR: La chiamata e' stata interrotta da un segnale
    - EINVAL: Il numero di descrittori da monitorare e' negativo o il timeout non è valido
    - ENOMEM: Il sistema non può allocare memoria per le strutture interne

Prototipo della system call select()

```
#include <sys/select.h>

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


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

- **nfds** : è il valore del descrittore massimo più 1. Va calcolato considerando il massimo valore del file descriptor a cui siamo interessati aggiungendoci 1.

Impostazione del tempo di attesa

- `timeout == NULL`
  - Tempo di attesa infinito (chiamata bloccante)
  - La funzione ritorna se almeno un descrittore è pronto per l'I/O oppure se si è verificato un errore
- `timeout->tv_sec == 0 && timeout->tv_usec == 0`
  - Polling (chiamata non bloccante)
  - Tutti i descrittori vengono controllati e la funzione ritorna immediatamente
- `timeout->tv_sec != 0 || timeout->tv_usec != 0`
  - Tempo di attesa limitato
  - La funzione ritorna se almeno un descrittore è pronto per l'I/O, se il timeout è trascorso oppure se si è verificato un errore

Insiemi dei descrittori (fd_set)

- **L'insieme dei file descriptor in lettura, scrittura ed eccezione è descritto tramite gli argomenti:**
  - readfds
  - writefds
  - exceptfds
- **Il tipo di questi argomenti è fd_set**
  - Tipicamente è necessario un bit per ciascun file
- **Procedura**
  - Allocazione di una variabile di tipo fd_set
  - Utilizzo di apposite MACRO per modificare la variabile di tipo fd_set

Operazioni sull'insiemi dei descrittori (*fd_set*)

- **Svuota l'insieme dei descrittori *fdset***
  - void FD_ZERO(fd_set *fdset)
- **Aggiunge il descrittore *fd* all'insieme *fdset***
  - void FD_SET(int fd, fd_set *fdset)
- **Rimuove il descrittore *fd* dall'insieme *fdset***
  - void FD_CLR(int fd, fd_set *fdset)
- **Verifica se il descrittore *fd* appartiene all'insieme *fdset***
  - int FD_ISSET(int fd, fd_set *fdset)
  - La MACRO ritorna non zero se il descrittore appartiene all'insieme, zero altrimenti

Esempio I (1)

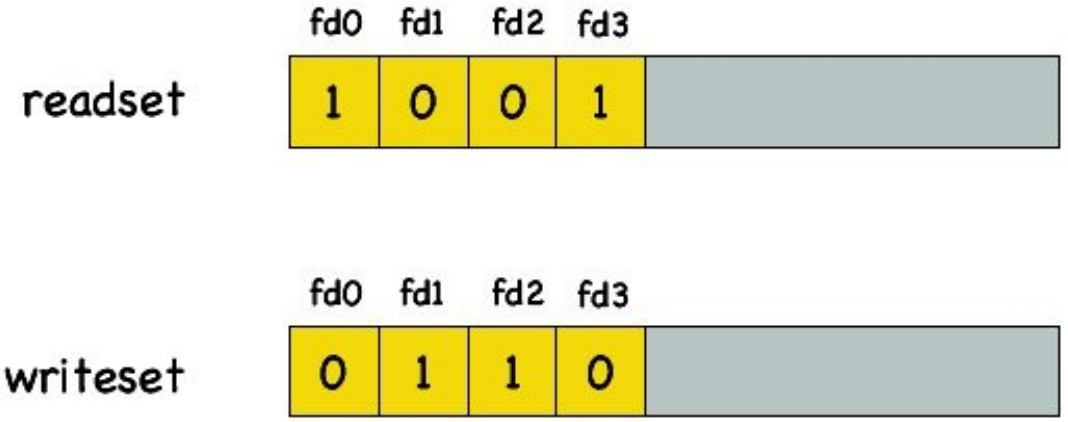
```
fd_set readset, writeset;

/* Clear the sets */
FD_ZERO(&readset);
FD_ZERO(&writeset);

/* Add the given fds */
FD_SET( 0, &readset);
FD_SET( 3, &readset);
FD_SET( 1, &writeset);
FD_SET( 2, &writeset);

/* Call the system call */
n = select(3+1, &readset, &writeset, NULL, NULL);
```

max file descriptor + 1



Esempio I (2)

- Per individuare quali sono i descrittori pronti dopo la chiamata `select()` si deve eseguire un test bit sulla variabile di tipo `fd_set` utilizzata, che viene aggiornata dalla `select()`
- Questa operazione deve essere eseguita solo se il valore restitutito da select() è positivo

```
n = select(3+1, &readset, &writeset, NULL, NULL);
if (n > 0) {
    if (FD_ISSET(0, &readset))  { /* non-blocking read on fd 0 */ }
    if (FD_ISSET(3, &readset))  { /* non-blocking read on fd 3 */ }
    if (FD_ISSET(1, &writeset)) { /* non-blocking write on fd 1 */ }
    if (FD_ISSET(2, &writeset)) { /* non-blocking write on fd 2 */ }
}
else {
  /* error */
}
```

Esempio II: Polling

```
fd_set rset; int n;
struct timeval timeout;
FD_ZERO(&rset);
/* Read set: {0,3} */
FD_SET(0, &rset); FD_SET(3, &rset);
/* Set timeout to zero */
memset(&timeout, 0, sizeof(struct timeout));
switch (select(3+1, &rset, NULL, NULL, &timeout)) {
  case -1: /* an error has occurred */
    break;
  case 0: /* no data available */
    break;
  default: /* data available */
    if (FD_ISSET(0, &rset)) { /* non-blocking read on fd 0 */ }
    if (FD_ISSET(3, &rset)) { /* non-blocking read on fd 3 */ }
}
```

Esempio III: Timeout

```
fd_set rset; int n;
struct timeval timeout;
FD_ZERO(&rset);
/* Read set: {0,3} */
FD_SET(0, &rset); FD_SET(3, &rset);
/* Set timeout to five seconds */
timeout.tv_sec  = 5; timeout.tv_usec = 0;
switch (select(3+1, &rset, NULL, NULL, &timeout)) {
  case -1: /* an error has occurred */
    break;
  case 0: /* no data within five seconds */
    break;
  default: /* data available */
    if (FD_ISSET(0, &rset)) { /* non-blocking read on fd 0 */ }
    if (FD_ISSET(3, &rset)) { /* non-blocking read on fd 3 */ }
}
```

Record/File Lock: Introduzione (1)

- **Problema: Cosa succede se un processo scrive un file mentre altri lo leggono?**
  - I processi che leggono possono leggere informazioni scritte in maniera parziale o incompleta
- **Problema: Cosa succede se più processi scrivono sullo stesso file contemporaneamente?**
  - L'output sul file viene mescolato in maniera imprevedibile
  - Un processo può sovrascrivere l'output di un altro processo
- **Soluzione: Record/File Lock**
  - Permette di bloccare l'accesso di altri processi ad un file (o parti di esso), così da evitare le sovrapposizioni e garantire l'atomicità dell'operazione di scrittura

Record/File Lock: Introduzione (2)

- **Tipologie di File Locking**
  - Read Lock / Shared Lock
    - Lock condiviso: permette l'accesso in lettura (al file) ad uno o più processi
  - Write Lock / Exclusive Lock
    - Lock esclusivo: permette l'accesso in scrittura (al file) ad un singolo processo
- **Modalità di File Locking**
  - Advisory Locking
    - Controllo degli accessi a carico dei singoli processi
  - Mandatory Locking
    - Controllo degli accessi a carico del kernel

Advisory Locking

- E' la prima modalità di *file locking* implementata nei sistemi unix-like
- I singoli processi sono incaricati di asserire e verificare se esistono delle condizioni che impediscono l'accesso al file
- Le funzioni `read()` o `write()` vengono eseguite comunque e non risentono della presenza di un eventuale *lock*
- I processi devono controllare esplicitamente lo stato dei file prima di accedervi utilizzando le relative funzioni

Mandatory Locking

- Introdotto in SVR4
- Il sistema impedisce ad un processo l'accesso al file (o parti di esso) se un altro processo ne detiene il lock
- Problemi:
  - Neanche *root* può accedere ad un file con lock, pertanto un processo che blocca un file cruciale (rendendolo completamente inaccessibile), può rendere inutilizzabile il sistema
- Di default il *Mandatory Locking* è disabilitato, per abilitarlo il filesystem deve essere montato con l'opzione mand

```
mount -t <vfstype> <device> <dir> -o mand
```

Record Lock: Operazioni

- **Richiesta di un Lock su una regione di un file**

	Read Lock	Write Lock
No Locks	OK	OK
n read locks (n >= 1)	OK	denied
One write lock	denied	denied

- **Operazioni eseguite da un processo**
  - Richiesta del lock (bloccante)
  - Se il processo riceve il lock:
    - Operazioni sul file (lettura o scrittura)
    - Rilascio del lock acquisito

Unix/Linux Record Lock: fcntl

```
#include <unistd.h>
#include <fcntl.h>

/* fcntl returns 0 on success, -1 on error */
int fcntl(int fd, int cmd, struct flock *lock);

struct flock {
  short l_type;   /* F_RDLCK, F_WRLCK, F_UNLCK */
  short l_whence; /* How to interpret l_start: SEEK_SET,
                     SEEK_CUR, SEEK_END */
  off_t l_start;  /* Starting offset for lock */
  off_t l_len;    /* Number of bytes to lock */
  pid_t l_pid;    /* PID of process blocking our lock
                    (returned by F_GETLK only) */
}
```

Man page: `man 2 fcntl`

La system call fcntl()

- La system call **fcntl** è utilizzata per varie operazioni su descrittori di file.
- Le operazioni più importanti sono:
  - Duplicare un descrittore esistente
    - cmd: **F_DUPFD**
  - Impostare o ottenere i flag associati al file descriptor (es. il flag close-on-exec)
    - cmd: **F_GETFD, F_SETFD**
  - Impostare o ottenere i flag stato di file (es. **O_RDONLY, O_WRONLY, O_APPEND, O_ASYNC, ...**)
    - cmd: **F_GETFL, F_SETFL**
  - Gestione dei record lock
    - cmd: **F_GETLK, F_SETLK, F_SETLKW**

Struttura flock (1)

- **Tipo di lock (l_type)**
  - F_RDLCK: lock in lettura (shared lock)
    - Il file deve essere aperto in lettura o lettura/scrittura
  - F_WRLCK: lock in scrittura (exclusive lock)
    - Il file deve essere aperto in scrittura o lettura/scrittura
  - F_UNLCK: rilascia il lock
    - Il lock viene rilasciato automaticamente quando:
      - Il processo termina
      - Il processo chiude il descrittore del file su cui si riferisce il lock
    - Nota: è buona norma rilasciare manualmente tutti i lock ottenuti

Struttura flock (2)

- **Offset da cui inizia la richiesta di lock (l_start): l'offset è un numero di byte aggiunti ad una data posizione del file specificata nel campo l_whence**
  - SEEK_SET: inizio del file
  - SEEK_CUR: posizione corrente
  - SEEK_END: fine del file
  - L'offset può essere negativo
- **Numero di byte su cui richiedere il lock (l_len)**
  - Valore non negativo
  - l_len == 0 : valore speciale per richiedere il lock dalla locazione specificata nei campi l_whence e l_start fino alla fine del file

fcntl: comandi di gestione del lock (1)

- F_SETLK
  - Richiede il lock quando l_type è F_RDLCK o F_WRLCK
  - Rilascia il lock quando l_type è F_UNLCK
  - L'operazione di lock/unlock interessa un certo numero di bytes specificati nei campi l_whence, l_start e l_len della struttura struct flock.
  - Se la richiesta di lock non può essere soddisfatta (va in conflitto con il lock acquisito da un altro processo) la funzione ritorna -1 e la variabile errno assume il valore EACCES o EAGAIN

fcntl: comandi di gestione del lock (2)

- **F_SETLKW**
  - Simile al comando F_SETLK
  - Differenza: se il lock non può essere concesso, il processo viene bloccato fino a quando non acquisisce il lock richiesto
  - La funzione diventa bloccante
  - Se il processo cattura un segnale, la chiamata viene interrotta e dopo la gestione del segnale la system call ritorna -1 e la variabile errno assume il valore EINTR

fcntl: comandi di gestione del lock (3)

- F_GETLK
  - Verifica se il lock può essere concesso oppure no
  - Se il lock può essere concesso, il campo `l_type` assume il valore F_UNLCK mentre gli altri campi della struttura `struct flock` non vengono modificati
    - Nota bene: il processo non acquisisce il lock
  - Se il lock non può essere concesso, la funzione ritorna il *Process ID*, nel campo `l_pid` della struttura `struct flock`, del processo che detiene il lock incompatibile

Esempio (1)

- **Lock in lettura sull'intero file**

```
struct flock lock;
lock.l_type = F_RDLCK; /* Read Lock */
lock.l_whence = SEEK_SET; /* Start of the file */
lock.l_start = 0; /* Offset at start of the file */
lock.l_len = 0; /* Special value: lock all bytes starting
                   at the location specified by l_whence
                   and l_start through the end of file */
/* Lock file descriptor fd (blocking mode) */
if (!fcntl(fd, F_SETLKW, &lock)) {
  /* Read operations on file descriptor fd ... */
  /* Release the lock */
  lock.l_type = F_UNLCK; /* Unlock */
  if (fcntl(fd, F_SETLK, &lock) == -1) { /* error */ }
}
```

Esempio (2)

- **Lock in scrittura sugli ultimi 10 bytes del file**

```
struct flock lock;
lock.l_type = F_WRLCK;    /* Write Lock */
lock.l_whence = SEEK_END; /* End of the file */
lock.l_start = -10;  /* Offset starting 10 bytes before the
                        end of the file */
lock.l_len = 0;      /* Special value */
/* Lock file descriptor fd (blocking mode) */
if (!fcntl(fd, F_SETLKW, &lock)) {
  /* Write operations on the locked region ... */
  /* Release the lock */
  lock.l_type = F_UNLCK; /* Unlock */
  if (fcntl(fd, F_SETLK, &lock) == -1) { /* error */ }
}
```

Esempio (3)

- **Lock in scrittura su 10 bytes dalla posizione corrente del file (in modalità non bloccante)**

```
struct flock lock;
lock.l_type = F_WRLCK;    /* Write Lock */
lock.l_whence = SEEK_CUR; /* Current file offset */
lock.l_start = 0;         /* Current file offset */
lock.l_len = 10;          /* Lock 10 bytes */
/* Lock file descriptor fd (non-blocking mode) */
if (!fcntl(fd, F_SETLK, &lock)) {
  /* Write operations on the locked region ... */
  /* Release the lock */
  lock.l_type = F_UNLCK; /* Unlock */
  if (fcntl(fd, F_SETLK, &lock) == -1) { /* error */ }
}
```

Record/File Lock: Informazioni aggiuntive

- I lock non vengono ereditati dal processo figlio creato con la system call `fork()`
- I lock vengono preservati dopo una chiamata `execve`
  - Se il flag *close-on-exec* è true, i lock vengono rilasciati ed i file vengono chiusi
- Evitare di utilizzare le funzioni della libreria **stdio**(3) per accedere alla regione di file su cui si è ottenuto il lock, poiché queste funzioni utilizzano un buffer nello spazio utente. In alternativa:
  - Forzare la scrittura con la funzione `fflush()`
  - Togliere il buffer dallo stream interessato con la funzione `setbuf()`
  - Utilizzare le system call `read()` e `write()`

Record Lock: Funzioni aggiuntive

- `lockf()`
  - Standard POSIX.1-2001
  - Permette la gestione del lock su una regione del file (solo dalla posizione corrente)
  - E' un'interfaccia verso la system call `fcntl()`
  - man 3 lockf
- `flock()`
  - Permette la gestione del lock solo sull'intero file
  - man 2 flock
- Queste funzioni sono più semplici rispetto alla `fcntl()` ma meno versatili e complete