Contents

1 Intro

Traduzione a cura di Giovanni Mu.

Sapete cos'è facile? `fork()` è facile. Potete forkare nuovi processi tutto il giorno e fargli gestire singoli pezzi di un problema in parallelo. Naturalmente, è più semplice se i processi non devono comunicare tra loro mentre sono in esecuzione e possono semplicemente stare lì a fare le loro cose.

Tuttavia, quando si inizia a fare `fork()` sui processi, si inizia subito a pensare alle fantastiche funzionalità multiutente che si potrebbero realizzare se i processi potessero comunicare tra loro facilmente. Quindi si prova a creare un array globale e poi si esegue `fork()` per vedere se è condiviso. (Ovvero, si verifica se sia il processo figlio che quello padre usano lo stesso array.) Presto, naturalmente, si scopre che il processo figlio ha la sua copia dell'array e il padre è ignaro di qualsiasi modifica apportata dal figlio.

Come si fa a far sì che questi processi comunichino tra loro, condividano strutture dati e siano generalmente amichevoli? Questo documento illustra diversi metodi di *Comunicazione Interprocesso (IPC)* che possono raggiungere questo obiettivo, alcuni di questi saranno indicati per determinati contesti, mentre altri non lo saranno.

1.1 Utenza

Se conoscete il C o il C++ e siete abbastanza abili nell'uso di un ambiente Unix (o di un altro ambiente POSIX che supporti queste chiamate di sistema), questi documenti fanno al caso vostro. Se non siete così bravi, beh, non preoccupatevi, ce la farete. Presumo, tuttavia, che abbiate una discreta esperienza di programmazione in C.

Come la Guida di Beej alla programmazione di rete tramite Internet Sockets[1], questi documenti hanno lo scopo di introdurre l'utente sopra menzionato nel mondo dell'IPC, offrendo una panoramica concisa delle varie tecniche IPC. Non si tratta di un insieme di documenti definitivo sull'argomento. Come ho detto, è progettato semplicemente per fornire un punto di partenza nell'entusiasmante mondo dell'IPC.

[1] `https://beej.us/guide/bgipc/source/examples/fork1.c`

1.2 Piattaforma e compilatore

Gli esempi in questo documento sono stati compilati sotto Linux usando `gcc`. Dovrebbero essere compilabili ovunque sia disponibile un buon compilatore Unix.

1.3 Pagina ufficiale

Il sito ufficiale di questo documento è `https://beej.us/guide/bgipc/`.[fn: 2]

1.4 Politica delle email

In genere sono disponibile ad aiutare con le domande via email, quindi sentiti libero di scrivermi, ma non posso garantire una risposta. Conduco una vita piuttosto impegnata e ci sono momenti in cui non riesco proprio a rispondere a una domanda. In tal caso, di solito cancello il messaggio. Non è niente di personale; semplicemente non avrò mai il tempo di darti la risposta dettagliata di cui hai bisogno.

Di norma, più complessa è la domanda, meno probabilità ho di ricevere una risposta. Se riesci a circoscrivere la tua domanda prima di inviarla e assicurati di includere tutte le informazioni pertinenti (come piattaforma, compilatore, messaggi di errore che ricevi e qualsiasi altra cosa che ritieni possa aiutarmi a risolvere il problema), è molto più probabile che tu riceva una risposta.

Se non ricevi risposta, continua a lavorarci sopra, prova a trovare la risposta e, se ancora non la trovi, scrivimi di nuovo con le informazioni che hai trovato e spero che mi siano sufficienti per aiutarti.

Ora che vi ho assillato su come scrivere e non scrivere a me, vorrei solo farvi sapere che apprezzo *profondamente* tutti gli elogi che la guida ha ricevuto nel corso degli anni. È un vero toccasana per il morale, e mi fa piacere sapere che viene usata per fare del bene! `:-)` Grazie!

1.5 Mirroring

Siete più che benvenuti a replicare questo sito, sia pubblicamente che privatamente. Se desiderate che il sito venga replicato pubblicamente e che io vi colleghi la pagina principale, scrivetemi a `beej@beej.us` .

1.6 Nota per i traduttori

Se desiderate tradurre la guida in un'altra lingua, scrivetemi a [`beej@beej.us`] e inserirò un link alla vostra traduzione dalla pagina principale. Sentitevi liberi di aggiungere il vostro nome e le vostre informazioni di contatto alla traduzione.

Si prega di notare le restrizioni di licenza nella sezione Copyright e distribuzione, qui sotto.

1.7 Copyright and Distribution

2 Introduzione a `fork()`

"Fork", oltre a essere una di quelle parole che iniziano a sembrare molto strane dopo essere state digitate ripetutamente, si riferisce al modo in cui Unix crea nuovi processi. Questo documento fornisce una rapida e semplice introduzione a `fork()`, poiché l'utilizzo di questa chiamata di sistema apparirà in altri documenti IPC. Se conoscete già tutto su `fork()`, potreste anche saltare questo documento.

2.1 2.1 "Cercate la Gola del Pericolo Eterno"

`fork()` può essere considerato un biglietto per il potere. Il potere a volte può essere considerato un biglietto per la distruzione. Pertanto, dovreste fare attenzione quando usate `fork()` sul vostro sistema, soprattutto quando le persone stanno lavorando ai loro progetti di fine semestre e sono pronte a distruggere il primo organismo che blocca il sistema. Non è che non dovreste mai usare `fork()` , dovete solo essere cauti. È un po' come ingoiare una spada: se state attenti, non vi sventrerete.

Visto che sei ancora qui, suppongo sia meglio che ti spieghi tutto. Come ho detto, `fork()` è il modo in cui Unix avvia nuovi processi. In pratica, funziona così: il processo padre (quello già esistente) esegue `fork()` su un processo figlio (quello nuovo). Il processo figlio ottiene una copia dei dati del padre. Voilà! Hai due processi dove ce n'era solo uno!

Naturalmente, ci sono un sacco di problemi da affrontare quando si esegue `fork()` sui processi, altrimenti il tuo amministratore di sistema si arrabbierà con te quando riempi la tabella dei processi di sistema e dovrà premere il pulsante di reset sulla macchina.

```
main()
{
     signal(SIGCHLD, SIG_IGN); /* Ora non devo avere wait()! */
     .
     .
     fork();fork();fork(); /* Conigli, conigli, conigli!*/
```

Ora, quando un processo figlio muore e non è stato `wait()` su di esso, di solito viene visualizzato in un elenco `ps` come `"<defunct>"` . Rimarrà in questo stato finché il processo padre non `wait()` su di esso, oppure finché non viene gestito come indicato di seguito.

Ora c'è un'altra regola che devi imparare: quando il genitore muore prima di usare `wait()` per il figlio (supponendo che non stia ignorando `SIGCHLD`), il figlio viene riassegnato al processo init (PID 1). Questo non è un problema se il figlio è ancora attivo e sotto controllo. Tuttavia, se il figlio è già defunto, siamo in una situazione difficile. Vedi, il genitore originale non può più usare wait(), dato che è morto. Quindi come fa `init` a sapere di dover usare `wait()` per questi *processi zombie* ?

La risposta: è magia! Beh, su alcuni sistemi, `init` distrugge periodicamente tutti i processi inattivi di sua proprietà. Su altri sistemi, si rifiuta categoricamente di diventare il padre di qualsiasi processo inattivo, distruggendolo invece immediatamente. Se si utilizza uno dei sistemi precedenti, si

potrebbe facilmente scrivere un ciclo che riempia la tabella dei processi con i processi inattivi di proprietà di `init` . Non renderebbe felice il vostro amministratore di sistema?

La tua missione: assicurarti che il tuo processo padre ignori `SIGCHLD` o `wait()` tutti i processi figlio che `fork` a. Beh, non è *sempre* necessario farlo (ad esempio quando si avvia un demone o qualcosa del genere), ma se sei un principiante con `fork()` , scrivi codice con cautela. Altrimenti, sentiti libero di lanciarti nella stratosfera.

In sintesi: i processi figlio diventano inattivi finché il processo padre non `wait()` , a meno che il padre non ignori `SIGCHLD` . Inoltre, i processi figlio (viventi o meno) i cui genitori muoiono senza attendere (sempre che il padre non ignori `SIGCHLD`) diventano figli del processo `init` , che li gestisce in modo pesante.

2.2 "Sono mentalmente preparato! Dammi il pulsante!"

Esatto! Ecco un esempio[??] di come usare `fork()`:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>main()

inti main(void)
{
     pid_t pid;
     int rv;

     switch(pid = fork()) {
     case -1:
           perror("fork"); /* something went wrong */
           exit(1);        /* parent exits */
    case 0:
           printf(" CHILD: This is the child process!\n");
           printf(" CHILD: My PID is %d\n", getpid());
           printf(" CHILD: My parent's PID is %d\n", getppid());
           printf(" CHILD: Enter my exit status (make it small): ");
           scanf(" %d", &rv);
```

```
            printf(" CHILD: I'm outta here!\n");
            exit(rv);
    default:
            printf("PARENT: This is the parent process!\n");
            printf("PARENT: My PID is %d\n", getpid());
            printf("PARENT: My child's PID is %d\n", pid);
            printf("PARENT: I'm now waiting for my child to exit()...\n");
            wait(&rv);
            printf("PARENT: My child's exit status is: %d\n", WEXITSTATUS(rv));
            printf("PARENT: I'm outta here!\n");
    }
    return 0;
}
```

Ci sono un sacco di cose da notare in questo esempio, quindi partiamo dall'inizio, va bene?

`pid_t` è il tipo di processo generico. In Unix, questo è un `short` . Quindi, chiamo `fork()` e salvo il valore di ritorno nella variabile `pid` . `fork()` è semplice, poiché può restituire solo tre cose:

Valore di ritorno	descrizione
`0`	Se restituisce `0` , sei il processo figlio. Puoi ottenere il PID del processo padre chia
`-1`	Se restituisce `-1`, qualcosa è andato storto e non è stato creato alcun figlio. Usa `p`
ogni altra cosa	Qualsiasi altro valore restituito da `fork()` indica che sei il genitore e il valore rest

Quando il figlio chiama `exit()` , il valore di ritorno passato arriverà al padre quando `wait()` . Come potete vedere dalla chiamata `wait()` , c'è una certa stranezza che entra in gioco quando stampiamo il valore di ritorno. Cos'è questa roba `WEXITSTATUS()` , comunque? Beh, è una macro che estrae il valore di ritorno effettivo del figlio dal valore restituito da `wait()` . Sì, ci sono altre informazioni nascoste in quell'int. Vi lascio cercare da soli.

"Come", vi chiederete, "`wait()` sa quale processo attendere? Voglio dire, dato che il padre può avere più figli, quale aspetta effettivamente wait()?" La risposta è semplice, amici miei: attende quello che esce per primo. Se proprio dovete, potete specificare esattamente quale figlio attendere chiamando `waitpid()` con il PID del vostro figlio come argomento.

Un'altra cosa interessante da notare dall'esempio precedente è che sia il padre che il figlio usano la variabile rv. Questo significa che è condivisa tra i processi? *NO!* Se lo fosse, non avrei scritto tutta questa roba IPC. *Ogni*

processo ha la sua copia di tutte le variabili. Ci sono anche molte altre cose che vengono copiate, ma dovrete leggere la pagina `man` per vedere cosa.

Un'ultima nota sul programma sopra: ho usato un'istruzione switch per gestire `fork()` , e non è proprio tipico. Il più delle volte vedrete un'istruzione if; a volte è più breve di:

```
if (!fork()) {
        printf("Sono il figlio!\n");
        exit(0);
    } else {
        printf("Sono il genitore!\n");
        wait(NULL);
    }
```

Oh sì, l'esempio precedente dimostra anche come usare `wait()` se non ti interessa quale sia il valore restituito dal figlio: basta chiamarlo con `NULL` come argomento.

2.3 Riepilogo

Ora sai tutto sulla potente funzione `fork()` ! È più utile di un sacco di vermi bagnato nella maggior parte delle situazioni computazionalmente intensive, e puoi stupire i tuoi amici alle feste. Lo giuro. Provala.

3 Segnali

A volte, un processo può usare un metodo utile per interferire un altro: i segnali. In pratica, un processo può "lanciare" un segnale e inviarlo a un altro processo. Viene invocato il gestore dei segnali del processo di destinazione (una semplice funzione) e il processo può gestirlo.

Il diavolo si nasconde nei dettagli, ovviamente, e in realtà ciò che è consentito fare in sicurezza all'interno del proprio gestore dei segnali è piuttosto limitato. Ciononostante, i segnali forniscono un servizio utile.

Ad esempio, un processo potrebbe volerne interrompere un altro, e questo può essere fatto inviando il segnale `SIGSTOP` a quel processo. Per continuare, il processo deve ricevere il segnale `SIGCONT` . Come fa il processo a sapere di doverlo fare quando riceve un certo segnale? Beh, molti segnali sono predefiniti e il processo ha un gestore dei segnali predefinito per gestirli.

Un gestore predefinito? Sì. Prendiamo ad esempio `SIGINT` . Questo è il segnale di interrupt che un processo riceve quando l'utente preme `^C` . Il

gestore dei segnali predefinito per `SIGINT` provoca l'uscita del processo! Vi suona familiare? Bene, come puoi immaginare, puoi sovrascrivere `SIGINT` per fare quello che vuoi (o niente!). Potresti far sì che il tuo processo stampi "Interrupt?! No way, Jose!" e continui a fare i suoi affari.

Ora sai che puoi far sì che il tuo processo risponda a qualsiasi segnale nel modo che preferisci. Naturalmente, ci sono delle eccezioni, perché altrimenti sarebbe troppo facile da capire. Prendi il sempre popolare `SIGKILL` , il segnale n. #9. Hai mai digitato "`kill -9 nnnn`" per terminare un processo in fuga numero `nnnn` ? Stavi inviandogli un `SIGKILL` . Ora potresti anche ricordare che nessun processo può uscire da un "kill -9", e avresti ragione. `SIGKILL` è uno dei segnali per i quali `non è possibile` aggiungere un proprio gestore di segnali. Anche il già citato `SIGSTOP` rientra in questa categoria.

(A parte: spesso si usa il comando Unix "`kill`" senza specificare un segnale da inviare... quindi di che segnale si tratta? La risposta: `SIGTERM` . È possibile scrivere un gestore personalizzato per `SIGTERM` in modo che il processo non risponda a un normale "`kill`", e l'utente debba quindi usare "`kill -9`" per distruggere il processo.)

Tutti i segnali sono predefiniti? Cosa succede se si desidera inviare a un processo un segnale con un significato che solo l'utente comprende? Ci sono due segnali che non sono riservati: `SIGUSR1` e `SIGUSR2` . È possibile utilizzarli come si desidera e gestirli come si preferisce. (Ad esempio, il programma del mio lettore CD potrebbe rispondere a SIGUSR1 avanzando alla traccia successiva. In questo modo, potrei controllarlo dalla riga di comando digitando "`kill -SIGUSR1 nnnn`".)

3.1 Catturare segnali per divertimento!

Come puoi immaginare, il comando Unix "kill" è un modo per inviare segnali a un processo. Per pura e semplice coincidenza, esiste una chiamata di sistema chiamata `kill()` che fa la stessa cosa. Accetta come argomento un numero di segnale (come definito in `signal.h`) e un ID di processo. Inoltre, esiste una routine di libreria chiamata `raise()` che può essere utilizzata per generare un segnale all'interno dello stesso processo.

La domanda scottante rimane: come si cattura un `SIGTERM` in eccesso di velocità? Bisogna chiamare `sigaction()` e fornirgli tutti i dettagli su quale segnale si desidera catturare e quale funzione si desidera chiamare per gestirlo.

Ecco la scomposizione di `sigaction()` :

```
int sigaction(int sig, const struct sigaction *act,
```

```
                struct sigaction *oact);
```

Il primo parametro, `sig` , indica quale segnale catturare. Può essere (probabilmente "dovrebbe" essere) un nome simbolico da `signal.h` , simile a `SIGINT` . Questa è la parte facile.

Il campo successivo, act, è un puntatore a una `struct sigaction` che contiene una serie di campi che è possibile compilare per controllare il comportamento del gestore del segnale. (Un puntatore alla funzione del gestore del segnale stessa, inclusa nella `struct` .)

Infine, `oact` può essere `NULL` , ma in caso contrario restituisce le informazioni del *vecchio* gestore del segnale che erano presenti in precedenza. Questo è utile se si desidera ripristinare il precedente gestore del segnale in un secondo momento.

Ci concentreremo su questi tre campi nella `struct sigaction` :

Signal	Descrizione
`sa_handler`	La funzione di gestione del segnale (o `SIG_IGN` per ignorare il segnale)
`sa_mask`	Un insieme di segnali da bloccare mentre questo viene gestito
`sa_flags`	Flag per modificare il comportamento del gestore, oppure `0`

E il campo `sa_mask`? Quando si gestisce un segnale, si potrebbe voler bloccare la trasmissione di altri segnali, e si può fare aggiungendoli a `sa_mask` . È un "set", il che significa che è possibile eseguire normali operazioni di set per manipolarli: `sigemptyset()` , `sigfillset()` , `sigaddset()` , `sigdelset()` e `sigismember()` . In questo esempio, cancelleremo semplicemente il set e non bloccheremo altri segnali.

Gli esempi aiutano sempre! Eccone uno che gestisce `SIGINT` , che può essere trasmesso premendo `^C` , chiamato `sigint.c4` :[2]

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>

void sigint_handler(int sig)
{
        /* using a char[] so that sizeof will work */
    const char msg[] = "Ahhh! SIGINT!\n";
```

[2] `https://beej.us/guide/bgipc/source/examples/sigint.c`

```
    write(0, msg, sizeof(msg));
}

int main(void)
{
    char s[200];
        struct sigaction sa = {
            .sa_handler = sigint_handler,
            .sa_flags = 0, // or SA_RESTART
            .sa_mask = 0,
        };

    if (sigaction(SIGINT, &sa, NULL) == -1) {
        perror("sigaction");
        exit(1);
    }

    printf("Enter a string:\n");

    if (fgets(s, sizeof s, stdin) == NULL)
        perror("fgets");
    else
        printf("You entered: %s\n", s);

    return 0;
}
```

Questo programma ha due funzioni: `main()` , che imposta il gestore del segnale (utilizzando la chiamata `sigaction()`), e `sigint_handler()` , che è il gestore del segnale stesso.

Cosa succede quando lo si esegue? Se si sta inserendo una stringa e si preme `^C` , la chiamata a `gets()` fallisce e imposta la variabile globale `errno` a `EINTR` . Inoltre, `sigint_handler()` viene chiamata ed esegue la sua routine, quindi si vede effettivamente:

```
Enter a string: the quick brown fox jum^CAhhh!  SIGINT! fgets:
Interrupted system call
```

E poi esce. Ehi, che tipo di gestore è questo, se esce comunque?

Beh, abbiamo un paio di cose in gioco, qui. Innanzitutto, noterete che il gestore del segnale è stato chiamato, perché ha stampato "Ahhh! SIGINT!". Ma poi `fgets()` restituisce un errore, ovvero `EINTR` , ovvero "Chiamata di

sistema interrotta". Alcune chiamate di sistema possono essere interrotte dai segnali e, quando ciò accade, restituiscono un errore. Potreste vedere codice come questo (a volte citato come un uso scusabile di `goto`):

```
restart:
    if (some_system_call() == -1) {
        if (errno == EINTR) goto restart;
        perror("some_system_call");
        exit(1);
    }
```

Invece di usare `goto` in questo modo, potresti impostare `sa_flags` in modo che includa `SA_RESTART` . Ad esempio, se modifichiamo il codice del gestore SIGINT in questo modo:

```
sa.sa_flags = SA_RESTART;
```

Il nostro processo assomiglia più a questa:

```
Enter a string: Hello^CAhhh!  SIGINT! Er, hello!^CAhhh!  SIGINT!
This time fer sure! You entered:  This time fer sure!
```

Alcune chiamate di sistema sono interrompibili, mentre altre possono essere riavviate. Dipende dal sistema.

3.2 Il gestore non è onnipotente

Bisogna fare attenzione quando si effettuano chiamate di funzione nel gestore di segnale. Queste funzioni devono essere "async-safe", in modo che possano essere chiamate senza invocare comportamenti indefiniti.

Potresti essere curioso, ad esempio, del perché il mio gestore di segnale, sopra, abbia chiamato `write()` per visualizzare il messaggio invece di `printf()` . Beh, la risposta è che POSIX afferma che `write()` è async-safe (quindi è sicuro chiamarlo dall'interno del gestore), mentre `printf()` non lo è.

Le funzioni di libreria e le chiamate di sistema async-safe che possono essere chiamate dall'interno dei gestori di segnale sono (respira):

`_Exit()`, `_exit()`, `abort()`, `accept()`, `access()`, `aio_error()`, `aio_return()`, `aio_suspend()`, `alarm()`, `bind()`, `cfgetispeed()`, `cfgetospeed()`, `cfsetispeed()`, `cfsetospeed()`, `chdir()`, `chmod()`, `chown()`, `clock_gettime()`, `close()`, `connect()`, `creat()`, `dup()`, `dup2()`, `execle()`, `execve()`, `fchmod()`, `fchown()`, `fcntl()`, `fdatasync()`, `fork()`, `fpathconf()`, `fstat()`, `fsync()`, `ftruncate()`, `getegid()`, `geteuid()`, `getgid()`, `getgroups()`, `getpeername()`, `getpgrp()`,

`getpid()`, `getppid()`, `getsockname()`, `getsockopt()`, `getuid()`, `kill()`, `link()`, `listen()`, `lseek()`, `lstat()`, `mkdir()`, `mkfifo()`, `open()`, `pathconf()`, `pause()`, `pipe()`, `poll()`, `posix_trace_event()`, `pselect()`, `raise()`, `read()`, `readlink()`, `recv()`, `recvfrom()`, `recvmsg()`, `rename()`, `rmdir()`, `select()`, `sem_post()`, `send()`, `sendmsg()`, `sendto()`, `setgid()`, `setpgid()`, `setsid()`, `setsockopt()`, `setuid()`, `shutdown()`, `sigaction()`, `sigaddset()`, `sigdelset()`, `sigemptyset()`, `sigfillset()`, `sigismember()`, `sleep()`, `signal()`, `sigpause()`, `sigpending()`, `sigprocmask()`, `sigqueue()`, `sigset()`, `sigsuspend()`, `sockatmark()`, `socket()`, `socketpair()`, `stat()`, `symlink()`, `sysconf()`, `tcdrain()`, `tcflow()`, `tcflush()`, `tcgetattr()`, `tcgetpgrp()`, `tcsendbreak()`, `tcsetattr()`, `tcsetpgrp()`, `time()`, `timer_getoverrun()`, `timer_gettime()`, `timer_settime()`, `times()`, `umask()`, `uname()`, `unlink()`, `utime()`, `wait()`, `waitpid()`, and `write()`.

Naturalmente, puoi chiamare le tue funzioni dall'interno del tuo gestore di segnali (purché non chiami funzioni non async-safe).

Ma aspetta, c'è di più!

Non puoi nemmeno modificare in modo sicuro i dati condivisi (ad esempio globali), con una notevole eccezione: le variabili dichiarate di classe di archiviazione e tipo `volatile sig_atomic_t` .

Ecco un esempio che gestisce `SIGUSR1` impostando un flag globale, che viene poi esaminato nel ciclo principale per verificare se il gestore è stato chiamato. Questo è `sigusr.c` :[3]

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>

volatile sig_atomic_t got_usr1;

void sigusr1_handler(int sig)
{
    got_usr1 = 1;
}

int main(void)
{
        struct sigaction sa = {
```

[3] `https://beej.us/guide/bgipc/source/examples/sigusr.c`

```
            .sa_handler = sigusr1_handler,
            .sa_flags = 0, // or SA_RESTART
            .sa_mask = 0,
        };

    got_usr1 = 0;

    if (sigaction(SIGUSR1, &sa, NULL) == -1) {
        perror("sigaction");
        exit(1);
    }

    while (!got_usr1) {
        printf("PID %d: working hard...\n", getpid());
        sleep(1);
    }

    printf("Done in by SIGUSR1!\n");

    return 0;
}
```

Avvialo in una finestra, quindi usa `kill -USR1` in un'altra finestra per terminarlo. Il programma `sigusr` stampa comodamente il suo ID di processo, così puoi passarlo a `kill` :

`$ sigusr PID 5023: working hard... PID 5023: working hard... PID 5023: working hard...`

Quindi nell'altra finestra, invia il segnale `SIGUSR1` :

`$ kill -USR1 5023`

E il programma dovrebbe rispondere:

`PID 5023: working hard... PID 5023: working hard... Done in by SIGUSR1!`

(E la risposta dovrebbe essere immediata anche se è appena stata chiamata `sleep()` : `sleep()` viene interrotta dai segnali.)

3.3 Che dire di `signal()` ?

ANSI C definisce una funzione chiamata `signal()` che può essere utilizzata per catturare segnali. Non è affidabile o completa come `sigaction()` , quindi l'uso di `signal()` è generalmente sconsigliato.

Alcuni segnali per renderti popolare

Ecco un elenco dei segnali che (molto probabilmente) hai a disposizione:

Segnale	Descrizione
SIGABRT	Segnale di interruzione del processo.
SIGALRM	Sveglia.
SIGFPE	Operazione aritmetica errata.
SIGHUP	Sospendere.
SIGILL	Istruzione illegale.
SIGINT	Segnale di interruzione terminale.
SIGKILL	Uccidi (non può essere catturato o ignorato).
SIGPIPE	Scrivi su una pipe senza che nessuno lo legga.
SIGQUIT	Segnale di chiusura del terminale.
SIGSEGV	Riferimento di memoria non valido.
SIGTERM	Segnale di terminazione.
SIGUSR1	Segnale definito dall'utente 1.
SIGUSR2	Segnale definito dall'utente 2.
SIGCHLD	Processo figlio terminato o arrestato.
SIGCONT	Continua l'esecuzione, se interrotta.
SIGSTOP	Interrompe l'esecuzione (non può essere intercettato o ignorato).
SIGTSTP	Segnale di arresto terminale.
SIGTTIN	Processo in background che tenta di leggere.
SIGTTOU	Processo in background che tenta di scrivere.
SIGBUS	Bus error.
SIGPOLL	Evento interrogabile.
SIGPROF	Timer di profilazione scaduto.
SIGSYS	Chiamata di sistema errata.
SIGTRAP	Trappola di traccia/punto di interruzione.
SIGURG	I dati ad alta larghezza di banda sono disponibili su una presa.
SIGVTALRM	Timer virtuale scaduto.
SIGXCPU	Limite di tempo della CPU superato.
SIGXFSZ	Limite di dimensione del file superato.

Ogni segnale ha il suo gestore di segnali predefinito, il cui comportamento è definito nelle pagine man locali.

Cosa ho tralasciato

Quasi tutto. Ci sono tonnellate di flag, segnali in tempo reale, mix di segnali con thread, mascheramento dei segnali, `longjmp()` e segnali, e altro ancora.

Naturalmente, questa è solo una guida introduttiva, ma in un ultimo disperato tentativo di fornirvi maggiori informazioni, ecco un elenco di pagine man con maggiori dettagli:

Gestione dei segnali:

- sigaction()[4]
- sigwait()[5]
- sigwaitinfo()[6]
- sigtimedwait()[7]
- sigsuspend()[8]
- sigpending()[9]

Delivering signals:

- kill()[10]
- raise()[11]
- sigqueue()[12]

Set operations:

- sigemptyset()[13]
- sigfillset()[14]
- sigaddset()[15]

[4] `https://man.archlinux.org/man/sigaction.2`
[5] `https://man.archlinux.org/man/sigwait.3`
[6] `https://man.archlinux.org/man/sigwaitinfo.2`
[7] `https://man.archlinux.org/man/sigtimedwait.2`
[8] `https://man.archlinux.org/man/sigsuspend.2`
[9] `https://man.archlinux.org/man/sigpending.2`
[10] `https://man.archlinux.org/man/kill.2`
[11] `https://man.archlinux.org/man/raise.3`
[12] `https://man.archlinux.org/man/sigqueue.3`
[13] `https://man.archlinux.org/man/sigemptyset.3`
[14] `https://man.archlinux.org/man/sigfillset.3`
[15] `https://man.archlinux.org/man/sigaddset.3`

- sigdelset()[16]
- sigismember()[17]

Altri:

- sigprocmask()[18]
- sigaltstack()[19]
- siginterrupt()[20]
- sigsetjmp()[21]
- siglongjmp()[22]
- signal()[23]

4 Pipe

Non esiste una forma di IPC più semplice delle pipe. Implementate su ogni versione di Unix, `pipe()` e `fork()` costituiscono la funzionalità alla base del carattere "`|`" in "`ls | more`". Sono marginalmente utili per scopi interessanti, ma rappresentano un buon modo per apprendere i metodi base di IPC.

Dato che sono così semplici, non ci dedicherò molto tempo. Ci limiteremo ad alcuni esempi e altro.

4.1 "Queste pipe sono pulite!"

Aspetta! Non così in fretta. A questo punto potrei dover definire un "descrittore di file". Mettiamola così: conosci "`FILE*`" da `stdio.h`, vero? Sai che ci sono tutte quelle belle funzioni come `fopen()`, `fclose()`, `fwrite()` e così via? Beh, quelle sono in realtà funzioni di alto livello implementate tramite *descrittori di file* , che utilizzano chiamate di sistema come `open()`,

[16] `https://man.archlinux.org/man/sigdelset.3`
[17] `https://man.archlinux.org/man/sigismember.3`
[18] `https://man.archlinux.org/man/sigprocmask.2`
[19] `https://man.archlinux.org/man/sigaltstack.2`
[20] `https://man.archlinux.org/man/siginterrupt.3`
[21] `https://man.archlinux.org/man/sigsetjmp.3`
[22] `https://man.archlinux.org/man/siglongjmp.3`
[23] `https://man.archlinux.org/man/signal.2`

`creat()`, `close()` e `write()` . I descrittori di file sono semplicemente interi analoghi a `FILE*` in `stdio.h` .

Ad esempio, stdin è il descrittore di file "0", `stdout` è "1" e `stderr` è "2". Allo stesso modo, qualsiasi file aperto tramite `fopen()` ottiene il proprio descrittore di file, sebbene questo dettaglio ti sia nascosto. (Questo descrittore di file può essere recuperato da `FILE*` utilizzando la macro `fileno()` da `stdio.h` .)

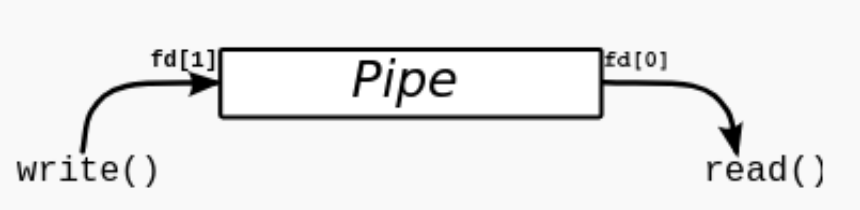

Figura 4.1: Come una pipe è organizzata

In pratica, una chiamata alla funzione `pipe()` restituisce una coppia di descrittori di file. Uno di questi descrittori è connesso all'estremità di scrittura della pipe, mentre l'altro è connesso all'estremità di lettura. Qualsiasi dato può essere scritto nella pipe e letto dall'altra estremità nell'ordine in cui è arrivato. Su molti sistemi, le pipe si riempiono dopo aver scritto circa 10K senza leggere nulla.

Come esempio inutile[24], il programma seguente crea, scrive e legge da una pipe.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>

int main(void)
{
    int pfds[2];
    char buf[30];

    if (pipe(pfds) == -1) {
        perror("pipe");
        exit(1);
    }
```

[24] `https://beej.us/guide/bgipc/source/examples/pipe1.c`

```
    printf("writing to file descriptor #%d\n", pfds[1]);
    write(pfds[1], "test", 5);
    printf("reading from file descriptor #%d\n", pfds[0]);
    read(pfds[0], buf, 5);
    printf("read \"%s\"\n", buf);

    return 0;
}
```

Come potete vedere, `pipe()` accetta un array di due int come argomento. Supponendo che non ci siano errori, connette due descrittori di file e li restituisce nell'array. Il primo elemento dell'array è il terminale di lettura della pipe, il secondo è il terminale di scrittura.

4.2 `fork()` e `pipe()` : hai il potere!

Dall'esempio precedente, è piuttosto difficile capire come possano essere utili. Bene, dato che questo è un documento IPC, inseriamo `fork()` nel mix e vediamo cosa succede. Immagina di essere un agente federale di alto livello incaricato di far sì che un processo figlio invii la parola "test" al padre. Non molto affascinante, ma nessuno ha mai detto che l'informatica sarebbe stata X-Files, Mulder.

Per prima cosa, faremo in modo che il padre crei una pipe. In secondo luogo, eseguiremo `fork()` . Ora, la pagina man di fork() ci dice che il figlio riceverà una copia di tutti i descrittori di file del padre, inclusa una copia dei descrittori di file della pipe. Quindi, il figlio potrà inviare dati al lato scrittura della pipe, e il padre li riceverà dal lato lettura. In questo modo: [25]

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    int pfds[2];
```

[25] `https://beej.us/guide/bgipc/source/examples/pipe2.c`

```
    char buf[30];

    pipe(pfds);

    if (!fork()) {
        printf(" CHILD: writing to the pipe\n");
        write(pfds[1], "test", 5);
        printf(" CHILD: exiting\n");
        exit(0);
    } else {
        printf("PARENT: reading from pipe\n");
        read(pfds[0], buf, 5);
        printf("PARENT: read \"%s\"\n", buf);
        wait(NULL);
    }

    return 0;
}
```

Tieni presente che i tuoi programmi dovrebbero avere un controllo degli errori molto più approfondito del mio. A volte lo ometto per mantenere le cose più chiare.

In ogni caso, questo esempio è identico al precedente, solo che ora eseguiamo il `fork()` di un nuovo processo e gli facciamo scrivere nella pipe, mentre il processo padre legge da essa. L'output risultante sarà simile al seguente:

```
PARENT: reading from pipe
 CHILD: writing to the pipe
 CHILD: exiting
PARENT: read "test"
```

In questo caso, il genitore ha tentato di leggere dalla pipe prima che il figlio vi scrivesse. Quando ciò accade, si dice che il genitore si *blocca* , o dorme, finché non arrivano dati da leggere. Sembra che il genitore abbia tentato di leggere, sia andato a dormire, il figlio abbia scritto ed è uscito, e il genitore si sia risvegliato e abbia letto i dati.

Evviva!! Hai appena fatto una comunicazione interprocesso! Era terribilmente semplice, eh? Scommetto che stai ancora pensando che `pipe()` non abbia molti usi e, beh, probabilmente hai ragione. Le altre forme di IPC sono generalmente più utili e spesso più esotiche.

4.3 La ricerca di Pipe come la conosciamo

Nel tentativo di farvi credere che le pipe siano in realtà bestie ragionevoli, vi farò un esempio di utilizzo di `pipe()` in una situazione più familiare. La sfida: implementare "`ls | wc -l`" in C.

Questo richiede l'utilizzo di un paio di funzioni in più di cui potreste non aver mai sentito parlare: `exec()` e `dup()` . La famiglia di funzioni exec() sostituisce il processo attualmente in esecuzione con quello passato a `exec()` . Questa è la funzione che useremo per eseguire `ls` e `wc -l` . `dup()` prende un descrittore di file aperto e ne crea un clone (un duplicato). Ecco come collegheremo l'output standard di ls allo standard input di `wc` . Notate, lo stdout di `ls` fluisce nella pipe, e lo stdin di `wc` fluisce dalla pipe. La pipe si adatta proprio lì, nel mezzo!

Comunque, ecco il codice:[26]

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    int pfds[2];

    pipe(pfds);

    if (!fork()) {
        close(1);       /* close normal stdout */
        dup(pfds[1]);   /* make stdout same as pfds[1] */
        close(pfds[0]); /* we don't need this */
        execlp("ls", "ls", NULL);
    } else {
        close(0);       /* close normal stdin */
        dup(pfds[0]);   /* make stdin same as pfds[0] */
        close(pfds[1]); /* we don't need this */
        execlp("wc", "wc", "-l", NULL);
    }

    return 0;
}
```

[26] `https://beej.us/guide/bgipc/source/examples/pipe3.c`

Vorrei fare un'altra annotazione sulla combinazione `close()/dup()`, perché è piuttosto strana. `close(1)` libera il descrittore di file 1 (standard output). `dup(pfds[1])` crea una copia del lato scrittura della pipe nel primo descrittore di file disponibile, che è "1", dato che lo abbiamo appena chiuso. In questo modo, tutto ciò che `ls` scrive sullo standard output (descrittore di file 1) andrà invece a `pfds[1]` (il lato scrittura della pipe). La sezione `wc` del codice funziona allo stesso modo, ma al contrario.

4.4 Riepilogo

Non ce ne sono molte per un argomento così semplice. In effetti, non ce ne sono quasi nessuna. Probabilmente l'uso migliore per le pipe è quello a cui siete più abituati: inviare lo standard output di un comando allo standard input di un altro. Per altri usi, è piuttosto limitante e spesso ci sono altre tecniche IPC che funzionano meglio.

5 FIFO

Una FIFO ("First In, First Out", pronunciato "Fy-Foh") è talvolta nota come *chiamata pipe* . In altre parole, è come una pipe , solo che ha un nome! In questo caso, il nome è quello di un file che più processi possono aprire (`open()`) e leggere e scrivere.

Quest'ultimo aspetto delle FIFO è progettato per aggirare uno dei limiti delle pipe normali: non è possibile afferrare un'estremità di una pipe normale creata da un processo non correlato. Ad esempio, se eseguo due copie individuali di un programma, entrambe possono chiamare `pipe()` quanto vogliono e non essere comunque in grado di comunicare tra loro. (Questo perché è necessario eseguire `pipe()` e poi `fork()` per ottenere un processo figlio che possa comunicare con il padre tramite la pipe.) Con le FIFO, invece, ogni processo non correlato può semplicemente aprire (`open()`) la pipe e trasferire dati attraverso di essa.

5.1 Nasce una nuova FIFO

Dato che la FIFO è in realtà un file su disco, è necessario fare qualche operazione un po' più elaborata per crearla. Non è poi così difficile. Basta chiamare `mkfifo()` con gli argomenti appropriati. Ecco una chiamata a `mkfifo()` che crea una FIFO:

```
mkfifo("myfifo", 0644);
```

Nell'esempio precedente, il file FIFO si chiamerà "myfifo". Il secondo argomento è la modalità di creazione, che viene utilizzata per indicare a `mkfifo()` di creare un FIFO (la parte `S_IFIFO` dell'OR) e impostare i permessi di accesso a quel file (ottale 644, o `rw-r--r--`), che possono essere impostati anche combinando le macro tramite OR da `sys/stat.h` . Questo permesso è identico a quello che si imposta con il comando `chmod` . Infine, viene passato un numero di dispositivo. Questo viene ignorato durante la creazione di un FIFO, quindi è possibile inserirvi qualsiasi elemento desiderato.

(Nota: un FIFO può essere creato anche dalla riga di comando utilizzando il comando Unix `mkfifo` .)

5.2 Una nota storica: `mknod`

Il modo originale per creare una FIFO era usare `mknod()` , ma questa funzione è deprecata. Per ora, queste due chiamate sono equivalenti:

```
mknod("myfifo", S_IFIFO | 0644, 0);   // old way
mkfifo("myfifo", 0644);               // new way
```

Tuttavia, se il sistema lo supporta, dovresti usare `mkfifo()` per creare FIFO.

5.3 Produttori e consumatori

Una volta creata la FIFO, un processo può avviarsi e aprirla in lettura o scrittura utilizzando la chiamata di sistema standard `open()` .

Poiché il processo è più facile da comprendere una volta che si ha un po' di codice in mano, presenterò qui due programmi che inviano dati tramite una FIFO. Uno è `speak.c` , che invia dati tramite la FIFO, e l'altro si chiama `tick.c` , poiché estrae i dati dalla FIFO.

Ecco `speak.c` [27]:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
```

[27] `https://beej.us/guide/bgipc/source/examples/speak.c`

```
#include <sys/stat.h>
#include <unistd.h>

#define FIFO_NAME "american_maid"

int main(void)
{
    char s[300];
    int num, fd;

    mkfifo(FIFO_NAME, 0644);

    printf("waiting for readers...\n");
    fd = open(FIFO_NAME, O_WRONLY);
    printf("got a reader--type some stuff\n");

    while (gets(s), !feof(stdin)) {
        if ((num = write(fd, s, strlen(s))) == -1)
            perror("write");
        else
            printf("speak: wrote %d bytes\n", num);
    }

    return 0;
}
```

Ciò che `speak` fa è creare la FIFO, provando ad aprirla con `open()`. Ora, quello che succederà è che la chiamata `open()` si bloccherà finché un altro processo non aprirà l'altra estremità della pipe per la lettura. (C'è un modo per aggirare questo problema: vedi `O_NDELAY`, più avanti.) Quel processo è `tick.c`[28], mostrato qui:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
```

[28] `https://beej.us/guide/bgipc/source/examples/tick.c`

```
#include <unistd.h>

#define FIFO_NAME "american_maid"

int main(void)
{
    char s[300];
    int num, fd;

    mkfifo(FIFO_NAME, 0644);

    printf("waiting for writers...\n");
    fd = open(FIFO_NAME, O_RDONLY);
    printf("got a writer\n");

    do {
        if ((num = read(fd, s, 300)) == -1)
            perror("read");
        else {
            s[num] = '\0';
            printf("tick: read %d bytes: \"%s\"\n", num, s);
        }
    } while (num > 0);

    return 0;
}
```

Come `speak.c`, tick si bloccherà su `open()` se nessuno sta scrivendo nella FIFO. Non appena qualcuno apre la FIFO per scrivere, tick si attiverà.

Provalo! Avvia speak e si bloccherà finché non avvierai `tick` in un'altra finestra. (Al contrario, se avvii `tick`, si bloccherà finché non avvierai speak in un'altra finestra.) Digita nella finestra di `speak` e tick assorbirà tutto.

Ora, esci da `speak`. Nota cosa succede: `read()` in tick restituisce 0, a indicare EOF. In questo modo, il lettore può sapere quando tutti gli scrittori hanno chiuso la loro connessione alla FIFO. "Cosa?" ti chiederai "Possono esserci più scrittori nella stessa pipe?" Certo! Può essere molto utile, sai. Forse più avanti nel documento ti mostrerò come sfruttare questo problema.

Ma per ora, concludiamo questo argomento vedendo cosa succede quando si esce da `tick` mentre `speak` è in esecuzione. "Pipe rotta"! Cosa significa? Bene, quello che è successo è che quando tutti i lettori di una FIFO si chi-

udono e il writer è ancora aperto, il writer riceverà il segnale SIGPIPE la prossima volta che tenterà di scrivere (`write()`). Il gestore di segnale predefinito per questo segnale stampa "Broken Pipe" ed esce. Naturalmente, è possibile gestire la situazione in modo più efficiente intercettando SIGPIPE tramite la chiamata `signal()`.

Infine, cosa succede se si hanno più lettori? Beh, succedono cose strane. A volte uno dei lettori riceve tutto. A volte si alterna tra i lettori. Perché mai si vogliono avere più lettori?

5.4 `O_NDELAY`! SONO INARRESTABILE!

In precedenza, ho accennato alla possibilità di aggirare la chiamata bloccante `open()` se non esiste un reader o writer corrispondente. Il modo per farlo è chiamare `open()` con il flag `O_NDELAY` impostato nell'argomento mode:

```
fd = open(FIFO_NAME, O_WRONLY | O_NDELAY);
```

Questo farà sì che `open()` restituisca `-1` se non ci sono processi che hanno il file aperto in lettura.

Analogamente, è possibile aprire il processo di lettura utilizzando il flag `O_NDELAY`, ma questo ha un effetto diverso: tutti i tentativi di `read()` dalla pipe restituiranno semplicemente 0 byte letti se non ci sono dati nella pipe. (Ovvero, `read()` non si bloccherà più finché non ci saranno dati nella pipe.) Si noti che non è più possibile sapere se `read()` restituisce 0 perché non ci sono dati nella pipe o perché il processo di scrittura è terminato. Questo è il prezzo da pagare per il potere, ma il mio consiglio è di provare a mantenere il blocco quando possibile.

5.5 Note conclusive

Avere il nome della pipe direttamente sul disco semplifica sicuramente le cose, non è vero? Processi non correlati possono comunicare tramite pipe! (Questa è una possibilità che desidererete avere se utilizzate pipe normali per troppo tempo.) Tuttavia, la funzionalità delle pipe potrebbe non essere esattamente quella di cui avete bisogno per le vostre applicazioni. Le code di messaggi potrebbero essere più adatte alle vostre esigenze, se il vostro sistema le supporta.

6 Blocco dei file

Il blocco dei file fornisce un meccanismo molto semplice ma incredibilmente utile per coordinare gli accessi ai file. Prima di iniziare a illustrare i dettagli, lasciatemi svelarvi alcuni segreti del blocco dei file:

Esistono due tipi di meccanismi di blocco: obbligatorio e consultivo. I sistemi obbligatori impediscono effettivamente le operazioni di lettura e scrittura (`read()`) e scrittura (`write()`) sui file. Diversi sistemi Unix li supportano. Tuttavia, li ignorerò in questo documento, preferendo invece parlare esclusivamente dei blocchi consultivi. Con un sistema di blocco consultivo, i processi possono comunque leggere e scrivere da un file mentre è bloccato. Inutile? Non proprio, dato che esiste un modo per un processo di verificare l'esistenza di un blocco prima di una lettura o una scrittura. Vedete, è una sorta di sistema di blocco cooperativo. Questo è ampiamente sufficiente per quasi tutti i casi in cui è necessario il blocco dei file.

Dato che questo è chiaro, ogni volta che d'ora in poi mi riferirò a un blocco in questo documento, mi riferirò ai blocchi consultivi. Ecco fatto.

Ora, lasciatemi analizzare un po' meglio il concetto di blocco. Esistono due tipi di blocchi (di consulenza!): blocchi di lettura e blocchi di scrittura (chiamati anche blocchi condivisi e blocchi esclusivi, rispettivamente). I blocchi di lettura funzionano in modo tale da non interferire con altri blocchi di lettura. Ad esempio, più processi possono avere un file bloccato contemporaneamente in lettura. Tuttavia, quando un processo ha un blocco di scrittura su un file, nessun altro processo può attivare un blocco di lettura o scrittura finché non viene rilasciato. Un modo semplice per spiegarlo è che possono esserci più lettori contemporaneamente, ma solo un solo scrittore alla volta.

Un'ultima cosa prima di iniziare: ci sono molti modi per bloccare i file nei sistemi Unix. A System V piace `lockf()`, che, personalmente, trovo pessimo. Sistemi migliori supportano `flock()`, che offre un controllo migliore sul blocco, ma presenta ancora alcune lacune. Per portabilità e completezza, parlerò di come bloccare i file usando `fcntl()`. Vi incoraggio, tuttavia, a utilizzare una delle funzioni di livello superiore in stile `flock()` se si adatta alle vostre esigenze, ma voglio dimostrarvi in modo portabile l'intera gamma di potenza che avete a portata di mano. (Se il vostro System V Unix non supporta `fcntl()` POSIX-y, dovrete confrontare le seguenti informazioni con la pagina man di `lockf()`.)

6.1 Impostare un blocco

La funzione `fcntl()` fa praticamente tutto, ma la useremo solo per il blocco dei file. Impostare il blocco consiste nel compilare una struttura flock (dichiarata in `fcntl.h`) che descrive il tipo di blocco richiesto, aprire il file con la modalità corrispondente e chiamare `fcntl()` con gli argomenti appropriati, come ad esempio:

```
struct flock fl = {
    .l_type   = F_WRLCK,  /* F_RDLCK, F_WRLCK, F_UNLCK      */
    .l_whence = SEEK_SET, /* SEEK_SET, SEEK_CUR, SEEK_END   */
    .l_start  = 0,        /* Offset from l_whence           */
    .l_len    = 0,        /* length, 0 = to EOF             */
    // .l_pid             /* PID holding lock; F_RDLCK only */
};
int fd;

fd = open("filename", O_WRONLY);

fcntl(fd, F_SETLKW, &fl);  /* F_GETLK, F_SETLK, F_SETLKW */
```

Cosa è appena successo? Iniziamo con la `struct flock`, poiché i campi al suo interno vengono utilizzati per *descrivere* l'azione di blocco in corso. Ecco alcune definizioni di campo:

Campo	Descrizione
`l_type`	Qui è possibile specificare il tipo di blocco che si desidera impostare. `F_RDLCK`, `F_WRLCK` o
`l_whence`	Questo campo determina da dove inizia il campo `l_start` (è come un offset per l'offset).
`l_start`	Questo è l'offset iniziale in byte del blocco, relativo a `l_whence`.
`l_len`	Questa è la lunghezza della regione di blocco in byte (che inizia da `l_start` che è relativo
`l_pid`	ID del processo che detiene il blocco. Viene impostato dal kernel quando si utilizza il con

Nel nostro esempio, abbiamo chiesto di creare un lock di tipo `F_WRLCK` (un lock in scrittura), a partire da `SEEK_SET` (l'inizio del file), offset `0`, lunghezza `0` (un valore zero significa "lock a fine file"), con il PID impostato su `getpid()`.

Il passo successivo è aprire il file con `open()`, poiché `flock()` necessita di un descrittore di file per il file che viene bloccato. Si noti che quando si apre il file, è necessario aprirlo nella stessa modalità specificata nel lock, come mostrato nella tabella seguente. Se si apre il file nella modalità errata per un dato tipo di lock, `fcntl()` restituirà `-1` e `errno` verrà impostato su `EBADF`.

l_type	Mode
F_RDLCK	O_RDONLY o O_RDWR
F_WRLCK	O_RDONLY o O_RDWR

Infine, la chiamata a `fcntl()` imposta, cancella o ottiene effettivamente il blocco. Il secondo argomento (il comando) di `fcntl()` indica cosa fare con i dati passati nella struttura flock. L'elenco seguente riassume le funzioni di ciascun comando di `fcntl()`:

cmd	Descrizione
F_SETLKW	Questo argomento indica a `fcntl()` di tentare di ottenere il lock richiesto nella struttura
F_SETLK	Questa funzione è quasi identica a F_SETLKW. L'unica differenza è che questa non attende
F_GETLK	Se si desidera solo verificare la presenza di un blocco, ma non impostarne uno, è possibile

Nell'esempio precedente, chiamiamo `fcntl()` con F_SETLKW come argomento, quindi si blocca finché non riesce a impostare il blocco, quindi lo imposta e continua.

6.2 Pulire un blocco

Uffa! Dopo tutto quello che ho detto finora sul blocco, è il momento di qualcosa di semplice: sbloccare! In realtà, questo è un gioco da ragazzi in confronto. Riutilizzerò semplicemente il primo esempio e aggiungerò il codice per sbloccarlo alla fine:

```
struct flock fl = {
    .l_type   = F_WRLCK,  /* F_RDLCK, F_WRLCK, F_UNLCK      */
    .l_whence = SEEK_SET, /* SEEK_SET, SEEK_CUR, SEEK_END   */
    .l_start  = 0,        /* Offset from l_whence           */
    .l_len    = 0,        /* length, 0 = to EOF             */
    // .l_pid             /* PID holding lock; F_RDLCK only */
};
int fd;

fd = open("filename", O_WRONLY);

fcntl(fd, F_SETLKW, &fl);  /* F_GETLK, F_SETLK, F_SETLKW */
```

Ora, ho lasciato il vecchio codice di blocco lì per un contrasto elevato, ma puoi vedere che ho appena cambiato il campo `l_type` in F_UNLCK (lasciando gli altri completamente invariati!) e ho chiamato `fcntl()` con F_SETLK come comando. Facile!

6.3 Un programma demo

Qui includerò un programma demo, `lockdemo.c`, che attende che l'utente prema Invio, quindi blocca il proprio codice sorgente, attende un altro Invio e infine lo sblocca. Eseguendo questo programma in due (o più) finestre, è possibile osservare come i programmi interagiscono in attesa di blocchi.

In pratica, l'utilizzo è questo: se si esegue lockdemo senza argomenti da riga di comando, il programma tenta di ottenere un blocco in scrittura (`F_WRLCK`) sul suo codice sorgente (`lockdemo.c`). Se lo si avvia con qualsiasi argomento da riga di comando, tenta di ottenere un blocco in lettura (`F_RDLCK`).

Ecco il codice sorgente[29]:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
        struct flock fl = {
            .l_type = F_WRLCK,
            .l_whence = SEEK_SET,
            .l_start = 0,
            .l_len = 0,
        };
    int fd;

    if (argc > 1)
        fl.l_type = F_RDLCK;

    if ((fd = open("lockdemo.c", O_RDWR)) == -1) {
        perror("open");
        exit(1);
    }

    printf("Press <RETURN> to try to get lock: ");
    getchar();
```

[29] https://beej.us/guide/bgipc/source/examples/lockdemo.c

```
    printf("Trying to get lock...");

    if (fcntl(fd, F_SETLKW, &fl) == -1) {
        perror("fcntl");
        exit(1);
    }

    printf("got lock\n");
    printf("Press <RETURN> to release lock: ");
    getchar();

    fl.l_type = F_UNLCK;  /* set to unlock same region */

    if (fcntl(fd, F_SETLK, &fl) == -1) {
        perror("fcntl");
        exit(1);
    }

    printf("Unlocked.\n");

    close(fd);

    return 0;
}
```

Compila quel cucciolo e inizia a lavorarci in un paio di finestre. Nota che quando un lockdemo ha un blocco in lettura, altre istanze del programma possono ottenere i propri blocchi in lettura senza problemi. È solo quando viene ottenuto un blocco in scrittura che gli altri processi non possono ottenere alcun tipo di blocco.

Un'altra cosa da notare è che non è possibile ottenere un blocco in scrittura se sono presenti blocchi in lettura nella stessa area del file. Il processo in attesa di ottenere il blocco in scrittura attenderà che tutti i blocchi in lettura siano stati rimossi. Un risultato di questo è che puoi continuare ad accumulare blocchi in lettura (perché un blocco in lettura non impedisce ad altri processi di ottenere blocchi in lettura) e tutti i processi in attesa di un blocco in scrittura rimarranno lì a morire. Non esiste una regola che ti impedisca di aggiungere altri blocchi in lettura se c'è un processo in attesa di un blocco in scrittura. Devi fare attenzione.

In pratica, però, probabilmente userai i blocchi in scrittura principal-

mente per garantire l'accesso esclusivo a un file per un breve periodo di tempo durante l'aggiornamento; Questo è l'uso più comune dei blocchi, per quanto ne so. E li ho visti tutti... beh, ne ho visto uno... uno piccolo... un'immagine... beh, ne ho sentito parlare.

6.4 Riepilogo

I blocchi sono fondamentali. A volte, però, potrebbe essere necessario un maggiore controllo sui processi in una situazione produttore-consumatore. Per questo motivo, se non altro, dovresti consultare il documento sui semafori System V (o POSIX, se è per questo; non sono identici) se il tuo sistema supporta una bestia del genere. Forniscono una funzionalità più estesa e almeno equivalente ai blocchi di file.

7 Code di messaggi

Coloro che ci hanno portato System V hanno ritenuto opportuno includere alcune chicche IPC implementate su diverse piattaforme (incluso Linux, ovviamente). Questo documento descrive l'utilizzo e le funzionalità delle fantastiche code di messaggi di System V! Linux supporta anche una versione POSIX di ciascuna di esse; vedere / `mq_overview`, `sem_overview` e `shm_overview` / nelle pagine man.

Come al solito, vorrei darvi una panoramica prima di entrare nel vivo dell'argomento. Una coda di messaggi funziona in un certo senso come una FIFO, ma supporta alcune funzionalità aggiuntive. In genere, i messaggi vengono rimossi dalla coda nell'ordine in cui vengono inseriti. Nello specifico, tuttavia, ci sono modi per estrarre determinati messaggi dalla coda prima che raggiungano la parte iniziale. È come saltare la fila. (A proposito, non cercate di saltare la fila mentre visitate il parco divertimenti Great America nella Silicon Valley, perché potreste essere arrestati. Lì prendono il taglio molto seriamente.)

In termini di utilizzo, un processo può creare una nuova coda di messaggi o connettersi a una esistente. In quest'ultimo caso, due processi possono scambiarsi informazioni attraverso la stessa coda di messaggi. Punto.

Un'altra cosa su System V IPC: quando si crea una coda di messaggi, questa non scompare finché non la si elimina, proprio come i file non scompaiono finché non vengono rimossi esplicitamente. Tutti i processi che l'hanno utilizzata possono uscire, ma la coda continuerà a esistere. Una buona pratica è usare il comando ipcs per verificare se alcune delle code di messaggi inutilizzate sono in giro. È possibile eliminarle con il comando ipcrm, che è

preferibile a ricevere una visita dall'amministratore di sistema che vi informa che avete preso tutte le code di messaggi disponibili sul sistema.

7.1 Dov'è la mia coda?

Cominciamo! Prima di tutto, vuoi connetterti a una coda, o crearla se non esiste. La chiamata per farlo è la chiamata di sistema `msgget()`:

```
int msgget(key_t key, int msgflg);
```

`msgget()` restituisce l'ID della coda messaggi in caso di successo, o `-1` in caso di fallimento (e imposta `errno`, ovviamente).

Gli argomenti sono un po' strani, ma possono essere compresi con un po' di intimidazione. Il primo, `key`, è un identificatore univoco a livello di sistema che descrive la coda a cui vuoi connetterti (o creare). Ogni altro processo che vuole connettersi a questa coda dovrà usare la stessa `key`.

L'altro argomento, `msgflg`, indica a `msgget()` cosa fare con la coda in questione. Per creare una coda, questo campo deve essere impostato uguale a `IPC_CREAT` tramite `OR` bit a bit con i permessi per questa coda. (I permessi della coda sono gli stessi dei permessi standard dei file: le code assumono l'ID utente e l'ID gruppo del programma che le ha create.)

Nella sezione seguente viene fornito un esempio di chiamata.

7.2 "Sei il Key Master?"

Che dire di questa assurdità sulla chiave? Come si crea una chiave? Beh, dato che il tipo `key_t` è in realtà solo un `long`, puoi usare qualsiasi numero tu voglia. Ma cosa succede se codifichi il numero in modo fisso e un altro programma non correlato codifica lo stesso numero in modo fisso ma vuole un'altra coda? La soluzione è usare la funzione `ftok()` che genera una chiave da due argomenti:

```
key_t ftok(const char *‘path‘, int ‘id‘);
```

Ok, la situazione si fa strana. In pratica, path deve essere semplicemente un percorso verso un file che identifichi in modo univoco questa applicazione; il percorso verso il file di configurazione dell'applicazione è una stringa comune da utilizzare (quali sono le probabilità che due applicazioni utilizzino lo stesso file di configurazione?). L'altro argomento, `id`, è solitamente impostato su un carattere arbitrario, come "A". La funzione `ftok()` utilizza le informazioni sul file indicato (come il numero di inode, ecc.) e l'id per

generare una `key` probabilmente univoca per `msgget()`. I programmi che vogliono utilizzare la stessa coda devono generare la stessa chiave, quindi devono passare gli stessi parametri a `Ftok()`.

Infine, è il momento di effettuare la chiamata:

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

key = ftok("/home/beej/somefile", 'b');
msqid = msgget(key, 0666 | IPC_CREAT);
```

Nell'esempio precedente, ho impostato i permessi sulla coda a 666 (o rw-rw-rw-, se vi sembra più chiaro). Ora abbiamo `msqid` che verrà utilizzato per inviare e ricevere messaggi dalla coda.

7.3 Invio alla coda

Una volta connessi alla coda dei messaggi tramite `msgget()`, si è pronti per inviare e ricevere messaggi. Innanzitutto, l'invio:

Ogni messaggio è composto da due parti, definite nella struttura del template `struct msgbuf`, come definito in `sys/msg.h`:

```
struct msgbuf {
    long mtype;
    char mtext[1];
};
```

Il campo `mtype` viene utilizzato in seguito durante il recupero dei messaggi dalla coda e può essere impostato su qualsiasi numero positivo. `mtext` è il dato che verrà aggiunto alla coda.

"Cosa?! Puoi inserire solo array di un byte in una coda di messaggi?! Inutile!!" Beh, non esattamente. Puoi usare qualsiasi struttura per inserire i messaggi nella coda, purché il primo elemento sia un long. Ad esempio, potremmo usare questa struttura per memorizzare ogni genere di "chicche":

```
struct pirate_msgbuf {
    long mtype;  /* must be positive */
    struct pirate_info {
        char name[30];
        char ship_type;
        int notoriety;
        int cruelty;
```

```
        int booty_value;
    } info;
};
```

Ok, quindi come passiamo queste informazioni a una coda di messaggi? La risposta è semplice, amici miei: basta usare `msgsnd()`:

```
int msgsnd(int msqid, const void *msgp,
           size_t msgsz, int msgflg);
```

`msqid` è l'identificativo della coda dei messaggi restituito da `msgget()`. Il puntatore `msgp` è un puntatore ai dati che si desidera aggiungere alla coda. `msgsz` è la dimensione in byte dei dati da aggiungere alla coda (senza contare la dimensione del membro mtype). Infine, `msgflg` consente di impostare alcuni parametri flag opzionali, che per ora ignoreremo impostandoli a `0`.

Il modo migliore per ottenere la dimensione dei dati da inviare è impostarla correttamente fin dall'inizio. Il primo campo della struttura dovrebbe essere di tipo `long`, come abbiamo visto. Per sicurezza e portabilità, dovrebbe esserci un solo campo aggiuntivo. Se ne serve più di uno, racchiudetelo in una struttura come nel caso di struct `pirate_msgbuf`, sopra.

Per ottenere la dimensione dei dati da inviare, basta prendere la dimensione del secondo campo:

```
struct cheese_msgbuf {
    long mtype;
    char name[20];
};

/* calculate the size of the data to send: */

struct cheese_msgbuf mbuf;
int size;

size = sizeof mbuf.name;

/* Or, without a declared variable: */

size = sizeof ((struct cheese_msgbuf*)0)->name;
```

Oppure, se hai molti campi diversi, inseriscili in una `struct` e usa l'operatore sizeof su di essa. Può essere particolarmente comodo, perché ora la sottostruttura può avere un nome a cui fare riferimento. Ecco un frammento

di codice che mostra una delle nostre strutture pirata aggiunta alla coda dei messaggi:

```
#include <sys/msg.h>
#include <stddef.h>

key_t key;
int msqid;
struct pirate_msgbuf pmb = {2, { "L'Olonais", 'S', 80, 10, 12035 } };

key = ftok("/home/beej/somefile", 'b');
msqid = msgget(key, 0666 | IPC_CREAT);

/* stick him on the queue */
/* struct pirate_info is the sub-structure */
msgsnd(msqid, &pmb, sizeof(struct pirate_info), 0);
```

A parte ricordarsi di controllare gli errori nei valori di ritorno di tutte queste funzioni, questo è tutto. Ah, sì: nota che ho impostato arbitrariamente il campo `mtype` a 2 lassù. Questo sarà importante nella prossima sezione.

7.4 Ricezione dalla coda

Ora che abbiamo il temuto pirata Francis L'Olonais bloccato nella nostra coda di messaggi, come facciamo a liberarlo? Come potete immaginare, esiste una controparte di `msgsnd()`: `msgrcv()`. Che fantasia!

Una chiamata a `msgrcv()` che lo farebbe sarebbe più o meno questa:

```
#include <sys/msg.h>
#include <stddef.h>

key_t key;
int msqid;
struct pirate_msgbuf pmb; /* where L'Olonais is to be kept */

key = ftok("/home/beej/somefile", 'b');
msqid = msgget(key, 0666 | IPC_CREAT);

/* get him off the queue! */
msgrcv(msqid, &pmb, sizeof(struct pirate_info), 2, 0);
```

C'è una novità da notare nella chiamata `msgrcv()`: il `2`! Cosa significa? Ecco la sinossi della chiamata:

```
int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

Il valore `2` specificato nella chiamata è il `msgtyp` richiesto. Ricordiamo che abbiamo impostato arbitrariamente `mtype` a `2` nella sezione `msgsnd()` di questo documento, quindi sarà quello che verrà recuperato dalla coda.

In realtà, il comportamento di `msgrcv()` può essere modificato drasticamente scegliendo un `msgtyp` positivo, negativo o zero:

msgtyp	Effetto su `msgrcv()`
Zero	Recupera il messaggio successivo nella coda, indipendentemente dal suo `mtype` .
Positivo	Recupera il messaggio successivo con un `mtype` *uguale* al `msgtyp` specificato.
Negativo	Recupera il primo messaggio nella coda il cui campo `mtype` è minore o uguale al valore as

Quindi, spesso si desidera semplicemente il messaggio successivo in coda, indipendentemente dal suo `mtype`. Pertanto, si imposta il parametro `msgtyp` a `0`.

7.5 Distruzione di una coda di messaggi

Arriva il momento in cui è necessario eliminare una coda di messaggi. Come ho detto prima, rimarranno in memoria finché non verranno rimossi esplicitamente; è importante farlo per non sprecare risorse di sistema. Ok, hai usato questa coda di messaggi tutto il giorno e sta diventando vecchia. Vuoi eliminarla. Ci sono due modi:

1. Usa il comando Unix ipcs per ottenere un elenco delle code di messaggi definite, quindi usa il comando `ipcrm` per eliminare la coda.

2. Scrivi un programma che lo faccia per te.

Spesso, quest'ultima scelta è la più appropriata, poiché potresti voler che il tuo programma ripulisca la coda prima o poi. Per farlo è necessario introdurre un'altra funzione: `msgctl()`.

La sinossi di `msgctl()` è:

```
int msgctl(int msqid, int cmd,
           struct msqid_ds *buf);
```

Naturalmente, `msqid` è l'identificatore di coda ottenuto da `msgget()`. L'argomento importante è `cmd`, che indica a `msgctl()` come comportarsi. Può essere una varietà di cose, ma parleremo solo di `IPC_RMID`, che viene utilizzato per rimuovere la coda di messaggi. L'argomento `buf` può essere impostato su `NULL` ai fini di `IPC_RMID`.

Supponiamo di avere la coda creata sopra per contenere i pirati. È possibile eliminare quella coda eseguendo la seguente chiamata:

```
#include <sys/msg.h>
.
.
msgctl(msqid, IPC_RMID, NULL);
```

E la coda dei messaggi non esiste più. (Naturalmente, il controllo degli errori di questi valori di ritorno è sempre opportuno!)

7.6 Qualcuno ha qualche esempio di programmi?

Per completezza, includerò un paio di programmi che comunicheranno tramite code di messaggi. Il primo, `kirk.c`, aggiunge messaggi alla coda e `spock.c` li recupera.

Ecco il codice sorgente di kirk.c[30]:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct my_msgbuf {
    long mtype;
    char mtext[200];
};

int main(void)
{
    struct my_msgbuf buf;
```

[30] `https://beej.us/guide/bgipc/source/examples/kirk.c`

```
    int msqid;
    key_t key;

    if ((key = ftok("kirk.c", 'B')) == -1) {
        perror("ftok");
        exit(1);
    }

    if ((msqid = msgget(key, 0644 | IPC_CREAT)) == -1) {
        perror("msgget");
        exit(1);
    }

    printf("Enter lines of text, ^D to quit:\n");

    buf.mtype = 1; /* we don't really care in this case */

    while(fgets(buf.mtext, sizeof buf.mtext, stdin) != NULL) {
        int len = strlen(buf.mtext);

        /* ditch newline at end, if it exists */
        if (buf.mtext[len-1] == '\n') buf.mtext[len-1] = '\0';

        if (msgsnd(msqid, &buf, len, 0) == -1)
            perror("msgsnd");
    }

    if (msgctl(msqid, IPC_RMID, NULL) == -1) {
        perror("msgctl");
        exit(1);
    }

    return 0;
}
```

`kirk` funziona in questo modo: consente di inserire righe di testo. Ogni riga viene raggruppata in un messaggio e aggiunta alla coda dei messaggi. La coda dei messaggi viene quindi letta da `spock`.

Ecco il codice sorgente di `spock.c`[31]:

[31] `https://beej.us/guide/bgipc/source/examples/spock.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct my_msgbuf {
    long mtype;
    char mtext[200];
};

int main(void)
{
    struct my_msgbuf buf;
    int msqid;
    key_t key;

    if ((key = ftok("kirk.c", 'B')) == -1) {  /* same key as kirk.c */
        perror("ftok");
        exit(1);
    }

    if ((msqid = msgget(key, 0644)) == -1) { /* connect to the queue */
        perror("msgget");
        exit(1);
    }

    printf("spock: ready to receive messages, captain.\n");

    for(;;) { /* Spock never quits! */
        if (msgrcv(msqid, &buf, sizeof buf.mtext, 0, 0) == -1) {
            perror("msgrcv");
            exit(1);
        }
        printf("spock: \"%s\"\n", buf.mtext);
    }

    return 0;
}
```

Si noti che `spock`, nella chiamata a `msgget()`, non include l'opzione `IPC_CREAT`. Abbiamo lasciato a `kirk` il compito di creare la coda dei messaggi, e spock restituirà un errore se non lo ha fatto.

Si noti cosa succede quando si eseguono entrambi in finestre separate e si termina uno o l'altro. Provate anche a eseguire due copie di `kirk` o due copie di `spock` per farvi un'idea di cosa succede quando si hanno due lettori o due scrittori. Un'altra dimostrazione interessante è eseguire `kirk`, inserire una serie di messaggi, quindi eseguire `spock` e vederlo recuperare tutti i messaggi in un colpo solo. Anche solo armeggiare con questi programmi giocattolo vi aiuterà a capire cosa sta realmente succedendo.

7.7 Riepilogo

Le code di messaggi offrono molto più di quanto questo breve tutorial possa presentare. Assicuratevi di consultare le pagine man per scoprire cos'altro potete fare, soprattutto nell'ambito di `msgctl()`. Inoltre, ci sono altre opzioni che potete passare ad altre funzioni per controllare come `msgsnd()` e `msgrcv()` gestiscono rispettivamente se la coda è piena o vuota.

8 Semafori

Ricordate il blocco dei file? Beh, i semafori possono essere considerati meccanismi di blocco consultivo molto generici. Potete usarli per controllare l'accesso ai file, alla memoria condivisa e, beh, praticamente a qualsiasi cosa vogliate. La funzionalità di base di un semaforo è che potete impostarlo, verificarlo o attendere che si azzeri e poi impostarlo ("test-n-set"). Non importa quanto complesso possa essere il procedimento che segue, ricordate queste tre operazioni.

Questo documento fornirà una panoramica delle funzionalità dei semafori e si concluderà con un programma che utilizza i semafori per controllare l'accesso a un file. (Questo compito, è vero, potrebbe essere facilmente gestito con il blocco dei file, ma è un buon esempio poiché è più facile da comprendere rispetto, ad esempio, alla memoria condivisa.)

8.1 Acquisizione di semafori

Con System V IPC, non si acquisiscono singoli semafori, ma insiemi di semafori. Naturalmente, è possibile acquisire un insieme di semafori che ne contenga uno solo, ma il punto è che è possibile ottenere un'intera serie di semafori semplicemente creando un singolo insieme di semafori.

Come si crea l'insieme di semafori? Si fa con una chiamata a `semget()`, che restituisce l'id del semaforo (di seguito denominato `semid`):

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

int semget(key_t key, int nsems, int semflg);
```

Qual è la `key`? È un identificatore univoco utilizzato da diversi processi per identificare questo set di semafori. (Questa `key` verrà generata utilizzando `ftok()`, descritto nella sezione Code di messaggi.)

L'argomento successivo, `nsems`, è (avete indovinato!) il numero di semafori in questo set di semafori. Il numero massimo dipende dal sistema, ma probabilmente si aggira intorno ai 32000. Se ne servono di più (avido miserabile!), basta procurarsi un altro set di semafori. È possibile passare `0` se ci si connette a un set di semafori esistente, ma è necessario specificare un numero positivo se si crea un nuovo set di semafori.

Infine, c'è l'argomento `semflg`. Questo indica a `semget()` quali permessi devono essere assegnati al nuovo set di semafori, se si sta creando un nuovo set o se si desidera semplicemente connettersi a uno esistente, e altre informazioni che è possibile consultare. Per creare un nuovo set, i permessi possono essere sottoposti a OR bit a bit con `IPC_CREAT`.

Ecco un esempio di chiamata che genera la chiave con `ftok()` e crea un set di 10 semafori, con 666 permessi (`rw-rw-rw-`):

```
#include <sys/ipc.h>
#include <sys/sem.h>

key_t key;
int semid;

key = ftok("/home/beej/somefile", 'E');
semid = semget(key, 10, 0666 | IPC_CREAT);
```

Congratulazioni! Hai creato un nuovo set di semafori! Dopo aver eseguito il programma, puoi verificarlo con il comando `ipcs`. (Non dimenticare di rimuoverlo quando hai finito con `ipcrm`!)

Aspetta! Attenzione! *¡Advertencia! ¡No pongas las manos en la tolva!* (È l'unica parola in spagnolo che ho imparato mentre lavoravo da Pizza Hut nel 1990. Era stampata sulla macchina per impastare.) Guarda qui:

Quando crei dei semafori per la prima volta, sono tutti non inizializzati; è necessaria un'altra chiamata per contrassegnarli come liberi (in particolare

a `semop()` o `semctl()` - vedi le sezioni seguenti). Cosa significa questo? Beh, significa che la creazione di un semaforo non è *atomica* (in altre parole, non è un processo in un'unica fase). Se due processi tentano di creare, inizializzare e utilizzare un semaforo contemporaneamente, potrebbe verificarsi una race condition.

Un modo per aggirare questa difficoltà è avere un singolo processo init che crei e inizializzi il semaforo molto prima che i processi principali inizino a funzionare. Il processo principale vi accede semplicemente, ma non lo crea né lo distrugge mai.

Stevens definisce questo problema "difetto fatale" del semaforo. Lo risolve creando il set di semafori con il flag `IPC_EXCL`. Se il processo 1 lo crea per primo, il processo 2 restituirà un errore durante la chiamata (con `errno` impostato su `EEXIST`). A quel punto, il processo 2 dovrà attendere che il semaforo venga inizializzato dal processo 1. Come fa a saperlo? Può chiamare ripetutamente `semctl()` con il flag `IPC_STAT` e analizzare il membro `sem_otime` della struttura `struct semid_ds` restituita. Se questo è diverso da zero, significa che il processo 1 ha eseguito un'operazione sul semaforo con `semop()`, presumibilmente per inizializzarlo.

Per un esempio, vedere il programma dimostrativo `semdemo.c`[32], qui sotto, in cui in genere reimplemento il codice di Stevens.

Nel frattempo, passiamo alla sezione successiva e diamo un'occhiata a come inizializzare i nostri semafori appena creati.

8.2 Controllo dei semafori con `semctl()`

Una volta creati i set di semafori, è necessario inizializzarli a un valore positivo per indicare che la risorsa è disponibile per l'uso. La funzione `semctl()` consente di apportare modifiche atomiche ai valori di singoli semafori o di set completi di semafori.

```
int semctl(int semid, int semnum, int cmd, ... /*arg*/);
```

`semid` è l'ID del set di semafori ottenuto dalla chiamata a `semget()`, in precedenza. `semnum` è l'ID del semaforo di cui si desidera manipolare il valore. `cmd` è l'operazione che si desidera eseguire con il semaforo in questione. L'ultimo "argomento", "`arg`", se richiesto, deve essere un `union semun`, che verrà definito nel codice come uno di questi:

```
union semun {
```

[32] `https://beej.us/guide/bgipc/source/examples/semdemo.c`

```
    int val;               /* used for SETVAL only */
    struct semid_ds *buf;  /* used for IPC_STAT and IPC_SET */
    ushort *array;         /* used for GETALL and SETALL */
};
```

(Si noti che l'`union semun` è ora definita nei file di intestazione dei moderni sistemi Linux. Tuttavia, non so quale macro di test delle funzionalità utilizzare per determinarlo, quindi definite questa unione solo se il vostro sistema non la definisce già. Leggete la documentazione di `semctl()` per maggiori informazioni.)

I vari campi nell'`union semun` vengono utilizzati a seconda del valore del parametro `cmd` di `semctl()` (segue un elenco parziale; consultate la vostra pagina man locale per maggiori informazioni):

I vari campi nell'`union semun` vengono utilizzati a seconda del valore del parametro `cmd` di `semctl()` (segue un elenco parziale: per maggiori informazioni, consultare la pagina man locale):

`cmd`	Effetto
`GETVAL`	Restituisce il valore del semaforo specificato.
`SETVAL`	Imposta il valore del semaforo specificato al valore nel membro val dell'unione passata se
`SETALL`	Imposta i valori di tutti i semafori nell'insieme ai valori nell'array puntato dal membro ar
`GETALL`	Ottiene i valori di tutti i semafori nell'insieme e li memorizza nell'array puntato dal mem
`IPC_RMID`	Rimuove l'insieme di semafori specificato dal sistema. Il parametro semnum viene ignora
`IPC_STAT`	Carica le informazioni di stato sull'insieme di semafori nella struttura struct `semid_ds` pu

Per i curiosi, ecco il contenuto (abbreviato) della `struct semid_ds` utilizzata nell'`union semun`:

```
struct semid_ds {
    struct ipc_perm sem_perm;  /* Ownership and permissions
    time_t          sem_otime; /* Last semop time */
    time_t          sem_ctime; /* Last change time */
    unsigned short  sem_nsems; /* No. of semaphores in set */
};
```

Utilizzeremo il membro `sem_otime` più avanti, quando scriveremo il nostro `initsem()` nel codice di esempio riportato di seguito.

8.3 `semop()`: potenza atomica!

Tutte le operazioni che impostano, ottengono o testano e impostano un semaforo utilizzano la chiamata di sistema `semop()`. Questa chiamata di sistema è di uso generale e la sua funzionalità è dettata da una struttura che le viene passata, `struct sembuf`:

```
/* Warning! Members might not be in this order! */

struct sembuf {
    ushort sem_num;
    short sem_op;
    short sem_flg;
};
```

Naturalmente, `sem_num` è il numero del semaforo nell'insieme che si desidera manipolare. Quindi, `sem_op` è ciò che si desidera fare con quel semaforo. Questo assume significati diversi a seconda che `sem_op` sia positivo, negativo o zero, come mostrato nella tabella seguente:

`sem_op`	Cosa succede
Negativo	Alloca risorse. Blocca il processo chiamante finché il valore del semaforo non è maggiore o
Positivo	Rilascia risorse. Il valore di `sem_op` viene aggiunto al valore del semaforo.
Zero	Questo processo attenderà che il semaforo in questione raggiunga 0.

Quindi, in pratica, quello che fai è caricare una `struct sembuf` con tutti i valori che vuoi, quindi chiamare `semop()`, in questo modo:

```
int semop(int semid, struct sembuf *sops,
          unsigned int nsops);
```

L'argomento `semid` è il numero ottenuto dalla chiamata a `semget()`. Il successivo è `sops`, che è un puntatore alla `struct sembuf` che hai riempito con i tuoi comandi per semafori. Se vuoi, però, puoi creare un array di `struct sembuf` per eseguire un sacco di operazioni semaforiche contemporaneamente. Il modo in cui `semop()` sa che stai facendo questo è l'argomento `nsop`, che indica quante `struct sembuf` stai inviando. Se ne hai solo una, beh, metti `1` come argomento.

Un campo nella `struct sembuf` che non ho menzionato è il campo `sem_flg`, che consente al programma di specificare flag per modificare ulteriormente gli effetti della chiamata a `semop()`.

Uno di questi flag è `IPC_NOWAIT` che, come suggerisce il nome, fa sì che la chiamata a `semop()` restituisca l'errore `EAGAIN` se incontra una situazione in cui normalmente si bloccherebbe. Questo è utile per le situazioni in cui si desidera effettuare un "poll" per verificare se è possibile allocare una risorsa.

Un altro flag molto utile è il flag `SEM_UNDO`. Questo fa sì che `semop()` registri, in un certo senso, la modifica apportata al semaforo. All'uscita dal programma, il kernel annullerà automaticamente tutte le modifiche contrassegnate con il flag `SEM_UNDO`. Naturalmente, il programma dovrebbe fare del suo meglio per deallocare tutte le risorse contrassegnate tramite il semaforo, ma a volte questo non è possibile quando il programma riceve un `SIGKILL` o si verifica qualche altro crash.

8.4 Eliminare un semaforo

Esistono due modi per eliminare un semaforo: uno è usare il comando Unix `ipcrm`. L'altro è tramite una chiamata a `semctl()` con il `cmd` impostato su `IPC_RMID`.

In pratica, si chiama `semctl()` e si imposta semid sull'ID del semaforo che si desidera eliminare. Il `cmd` deve essere impostato su `IPC_RMID`, che indica a `semctl()` di rimuovere questo set di semafori. Il parametro `semnum` non ha alcun significato nel contesto `IPC_RMID` e può essere impostato a zero.

Ecco un esempio di chiamata per eliminare un set di semafori:

```
int semid;
.
.
semid = semget(...);
.
.
semctl(semid, 0, IPC_RMID);
```

Easy peasy.

8.5 Programmi di esempio

Ce ne sono due. Il primo, `semdemo.c`, crea il semaforo se necessario ed esegue un finto blocco dei file su di esso in una demo molto simile a quella presente nel documento sul blocco dei file. Il secondo programma, `semrm.c`, viene utilizzato per distruggere il semaforo (anche in questo caso, `ipcrm` potrebbe essere utilizzato per questo scopo).

L'idea è di eseguire run `semdemo.c` in alcune finestre e vedere come interagiscono tutti i processi. Al termine, utilizzare `semrm.c` per rimuovere il semaforo. Si può anche provare a rimuovere il semaforo durante l'esecuzione di `semdemo.c`, solo per vedere che tipo di errori vengono generati.

Ecco semdemo.c[33], che include una funzione denominata `initsem()` che aggira le condizioni di competizione del semaforo, in stile Stevens:

```
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define MAX_RETRIES 10

#ifdef NEED_SEMUN
/* Defined in sys/sem.h as required by POSIX now */
union semun {
    int val;
    struct semid_ds *buf;
    ushort *array;
};
#endif

 /*
 ** initsem() -- more-than-inspired by W. Richard Stevens' UNIX Network
 ** Programming 2nd edition, volume 2, lockvsem.c, page 295.
 */
int initsem(key_t key, int nsems)  /* key from ftok() */
{
    int i;
    union semun arg;
    struct semid_ds buf;
    struct sembuf sb;
    int semid;

    semid = semget(key, nsems, IPC_CREAT | IPC_EXCL | 0666);
```

[33] https://beej.us/guide/bgipc/source/examples/semdemo.c

```
if (semid >= 0) { /* we got it first */
        sb.sem_op = 1; sb.sem_flg = 0;
        arg.val = 1;

        printf("press return\n"); getchar();

        for(sb.sem_num = 0; sb.sem_num < nsems; sb.sem_num++) {
                /* do a semop() to "free" the semaphores. */
                /* this sets the sem_otime field, as needed below. */
                if (semop(semid, &sb, 1) == -1) {
                        int e = errno;
                        semctl(semid, 0, IPC_RMID); /* clean up */
                        errno = e;
                        return -1; /* error, check errno */
                }
        }

} else if (errno == EEXIST) { /* someone else got it first */
        int ready = 0;

        semid = semget(key, nsems, 0); /* get the id */
        if (semid < 0) return semid; /* error, check errno */

        /* wait for other process to initialize the semaphore: */
        arg.buf = &buf;
        for(i = 0; i < MAX_RETRIES && !ready; i++) {
                semctl(semid, nsems-1, IPC_STAT, arg);
                if (arg.buf->sem_otime != 0) {
                        ready = 1;
                } else {
                        sleep(1);
                }
        }
        if (!ready) {
                errno = ETIME;
                return -1;
        }
} else {
        return semid; /* error, check errno */
```

```
    }

    return semid;
}

int main(void)
{
    key_t key;
    int semid;
    struct sembuf sb;

    sb.sem_num = 0;
    sb.sem_op = -1;  /* set to allocate resource */
    sb.sem_flg = SEM_UNDO;

    if ((key = ftok("semdemo.c", 'J')) == -1) {
            perror("ftok");
            exit(1);
    }

    /* grab the semaphore set created by seminit.c: */
    if ((semid = initsem(key, 1)) == -1) {
            perror("initsem");
            exit(1);
    }

    printf("Press return to lock: ");
    getchar();
    printf("Trying to lock...\n");

    if (semop(semid, &sb, 1) == -1) {
            perror("semop");
            exit(1);
    }

    printf("Locked.\n");
    printf("Press return to unlock: ");
    getchar();

    sb.sem_op = 1; /* free resource */
```

```
    if (semop(semid, &sb, 1) == -1) {
            perror("semop");
            exit(1);
    }

    printf("Unlocked\n");

    return 0;
}
```

Ecco `semrm.c`[34] per rimuovere il semaforo una volta terminato:

```
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int main(void)
{
    key_t key;
    int semid;
    union semun arg;

    if ((key = ftok("semdemo.c", 'J')) == -1) {
            perror("ftok");
            exit(1);
    }

    /* grab the semaphore set created by seminit.c: */
    if ((semid = semget(key, 1, 0)) == -1) {
            perror("semget");
            exit(1);
    }

    /* remove it: */
    if (semctl(semid, 0, IPC_RMID, 0) == -1) {
            perror("semctl");
            exit(1);
```

[34] https://beej.us/guide/bgipc/source/examples/semrm.c

```
    }

    return 0;
}
```

Non è divertente? Sono sicuro che abbandoneresti Quake[35] solo per giocare con questa roba dei semafori tutto il giorno!

8.6 Riepilogo

Potrei aver sottovalutato l'utilità dei semafori. Ti assicuro che sono molto, molto, molto utili in una situazione di concorrenza. Spesso sono anche più veloci dei normali blocchi di file. Inoltre, puoi usarli su altre cose che non sono file, come i segmenti di memoria condivisa! In effetti, a volte è difficile farne a meno, francamente.

Ogni volta che hai più processi che eseguono una sezione critica di codice, amico, hai bisogno di semafori. Ne hai miliardi, tanto vale usarli.

9 Segmenti di memoria condivisa

La cosa interessante dei segmenti di memoria condivisa è che sono esattamente ciò che sembrano: un segmento di memoria condiviso tra i processi. Insomma, pensate al potenziale di tutto questo! Potreste allocare un blocco di informazioni sui giocatori per una partita multigiocatore e far sì che ogni processo vi acceda a piacimento! Divertente, divertente, divertente. (Naturalmente, i file mappati in memoria svolgono la stessa funzione e hanno il vantaggio aggiuntivo della persistenza, sebbene con le stesse avvertenze che si applicano alla memoria condivisa.)

Come al solito, ci sono altri inconvenienti a cui fare attenzione, ma a lungo termine è tutto abbastanza semplice. Vedete, basta connettersi al segmento di memoria condivisa e ottenere un puntatore alla memoria. Potete leggere e scrivere su questo puntatore e tutte le modifiche apportate saranno visibili a tutti gli altri connessi al segmento. Non c'è niente di più semplice. Beh, in realtà c'è, ma stavo solo cercando di mettervi più a vostro agio.

La cosa interessante dei segmenti di memoria condivisa è che sono esattamente ciò che sembrano: un segmento di memoria condiviso tra i processi. Insomma, pensate al potenziale di tutto questo! Potreste allocare un blocco di informazioni sui giocatori per una partita multigiocatore e far sì che ogni

[35] O qualunque sia l'attuale gioco FPS avvincente di questi tempi.

processo vi acceda a piacimento! Divertente, divertente, divertente. (Naturalmente, i file mappati in memoria svolgono la stessa funzione e hanno il vantaggio aggiuntivo della persistenza, sebbene con le stesse avvertenze che si applicano alla memoria condivisa.)

Come al solito, ci sono altri inconvenienti a cui fare attenzione, ma a lungo termine è tutto abbastanza semplice. Vedete, basta connettersi al segmento di memoria condivisa e ottenere un puntatore alla memoria. Potete leggere e scrivere su questo puntatore e tutte le modifiche apportate saranno visibili a tutti gli altri connessi al segmento. Non c'è niente di più semplice. Beh, in realtà c'è, ma stavo solo cercando di mettervi più a vostro agio.

9.1 Creazione del segmento e connessione

Analogamente ad altre forme di IPC System V, un segmento di memoria condivisa viene creato e connesso tramite la chiamata `shmget()`:

```
int shmget(key_t key, size_t size, int shmflg);
```

Al termine, `shmget()` restituisce un identificatore per il segmento di memoria condivisa. L'argomento `key` deve essere creato come mostrato nel documento Message Queues, utilizzando `ftok()`. L'argomento successivo, `size`, è la dimensione in byte del segmento di memoria condivisa. Infine, `shmflg` deve essere impostato sui permessi del segmento, tramite OR bit a bit con `IPC_CREAT`, se si desidera creare il segmento, ma può essere impostato su `0` in caso contrario. (Non è male specificare `IPC_CREAT` ogni volta: si connetterà semplicemente se il segmento esiste già.)

Ecco un esempio di chiamata che crea un segmento da 1K con `644` permessi (`rw-r--r--`):

```
key_t key;
int shmid;

key = ftok("/home/beej/somefile3", 'R');
shmid = shmget(key, 1024, 0644 | IPC_CREAT);
```

(Potrebbe non essere possibile creare effettivamente un segmento da 1K, poiché il sistema operativo può aumentarne le dimensioni per adattarsi a eventuali vincoli interni. Ad esempio, su un sistema con 4K di pagine virtuali, è probabile che la dimensione venga aumentata a 4K. Naturalmente, il programma non lo saprà né gli importerà; questo è solo un dettaglio implementativo.)

Ma come si ottiene un puntatore a quei dati dall'handle shmid? La risposta è nella chiamata `shmat()`, nella sezione seguente.

9.2 Collegami: ottenere un puntatore al segmento

Prima di poter utilizzare un segmento di memoria condivisa, è necessario collegarsi ad esso tramite la chiamata `shmat()`:

```
void *shmat(int ‘shmid‘, void *‘shmaddr‘, int ‘shmflg‘);
```

Cosa significa tutto questo? Beh, `shmid` è l'ID della memoria condivisa ottenuto dalla chiamata a `shmget()`. Il successivo è `shmaddr`, che puoi usare per indicare a `shmat()` quale indirizzo specifico usare, ma dovresti semplicemente impostarlo a 0 e lasciare che sia il sistema operativo a scegliere l'indirizzo per te. Infine, `shmflg` può essere impostato a `SHM_RDONLY` se vuoi solo leggere da esso, 0 altrimenti. (Consulta le pagine man per altri flag utili che possono essere inclusi.)

Ecco un esempio più completo di come ottenere un puntatore a un segmento di memoria condivisa:

```
key_t key;
int shmid;
char *data;

key = ftok("/home/beej/somefile3", 'R');
shmid = shmget(key, 1024, 0644 | IPC_CREAT);
data = shmat(shmid, (void *)0, 0);
```

E *bammo*! Hai il puntatore al segmento di memoria condivisa! Nota che `shmat()` restituisce un puntatore `void` e, in questo caso, lo stiamo trattando come un puntatore `char`. Puoi trattarlo come preferisci, a seconda del tipo di dati che contiene. I puntatori ad array di strutture sono accettabili tanto quanto qualsiasi altro.

Inoltre, è interessante notare che `shmat()` restituisce `-1` in caso di errore (così come `mmap()`). Ma come si ottiene `-1` in un puntatore `void`? Basta eseguire un cast durante il confronto per verificare la presenza di errori:

```
data = shmat(shmid, (void *)0, 0);
if (data == MAP_FAILED)
    perror("shmat");
```

(È importante notare che l'intero viene convertito in un puntatore, e non il valore di ritorno del puntatore. È una differenza sottile, ma quest'ultimo non è sempre portabile tra architetture. Si noti inoltre che il cast è in `void*` e non in `char*`, come ci si potrebbe aspettare. Poiché il linguaggio garantisce che i cast impliciti da `void*` a qualsiasi altro tipo di puntatore siano sempre sicuri e affidabili, è meglio usare `void*` e lasciare che sia il compilatore a fare il resto.)

Tutto ciò che devi fare ora è modificare i dati a cui punta in stile puntatore normale. Nella prossima sezione troverai alcuni esempi.

9.3 Lettura e scrittura

Supponiamo di avere il puntatore `data` dell'esempio precedente. È un puntatore a `char`, quindi leggeremo e scriveremo caratteri da esso. Inoltre, per semplicità, supponiamo che il segmento di memoria condivisa da 1K contenga una stringa terminata da null.

Non potrebbe essere più semplice. Dato che si tratta solo di una stringa, possiamo stamparla in questo modo:

```
printf("shared contents: %s\n", data);
```

E potremmo memorizzarci qualcosa di semplice come questo:

```
printf("Enter a string: ");
fgets(data, 1024, stdin);
```

Naturalmente, come ho detto prima, puoi inserire altri dati oltre ai soli caratteri. Li sto solo usando come esempio. Darò per scontato che tu abbia abbastanza familiarità con i puntatori in C da essere in grado di gestire qualsiasi tipo di dato tu inserisca.

9.4 Scollegamento ed eliminazione dei segmenti

Una volta terminato il segmento di memoria condivisa, il programma dovrebbe scollegarsi da esso tramite la chiamata `shmdt()` (in caso contrario, ciò avverrà automaticamente al termine del processo):

`{.c{ int shmdt(void *‘shmaddr‘);`

L'unico argomento, `shmaddr`, è l'indirizzo ottenuto da `shmat()`. La funzione restituisce `-1` in caso di errore, `0` in caso di successo.

Quando ci si stacca dal segmento, questo non viene distrutto. Né viene rimosso quando *tutti* si staccano. È necessario distruggerlo specificamente

tramite una chiamata a `shmctl()`, simile alle chiamate di controllo per le altre funzioni IPC di System V:

```
shmctl(shmid, IPC_RMID, NULL);
```

La chiamata precedente elimina il segmento di memoria condivisa, supponendo che nessun altro vi sia collegato. La funzione `shmctl()` fa molto di più, tuttavia, e vale la pena di approfondire l'argomento. (Naturalmente, da soli, dato che questa è solo una panoramica!)

Come sempre, è possibile eliminare il segmento di memoria condivisa dalla riga di comando utilizzando il comando Unix `ipcrm`. Inoltre, assicuratevi di non lasciare segmenti di memoria condivisa inutilizzati, sprecando risorse di sistema. Tutti gli oggetti IPC System V di cui siete in possesso possono essere visualizzati utilizzando il comando `ipcs`.

9.5 Concorrenza

Cosa sono i problemi di concorrenza? Beh, poiché più processi modificano il segmento di memoria condivisa, è possibile che si verifichino determinati errori quando gli aggiornamenti del segmento avvengono simultaneamente. Questo accesso *simultaneo* è quasi sempre un problema quando più processi di scrittura si trovano su un oggetto condiviso.

Un modo per aggirare questo problema è usare i semafori per bloccare il segmento di memoria condivisa mentre un processo vi sta scrivendo. (A volte il blocco comprende sia la lettura che la scrittura sulla memoria condivisa, a seconda di cosa si sta facendo.)

Una vera e propria discussione sulla concorrenza esula dallo scopo di questo articolo, e potrebbe essere utile consultare l'articolo di Wikipedia sull'argomento[36]. Mi limiterò a questo: se si iniziano a riscontrare strane incongruenze nei dati condivisi quando si collegano due o più processi, si potrebbe benissimo avere un problema di concorrenza.

9.6 Codice di esempio

Ora che vi ho spiegato tutti i pericoli dell'accesso simultaneo a un segmento di memoria condivisa senza l'utilizzo di semafori, vi mostrerò una demo che fa proprio questo. Poiché non si tratta di un'applicazione mission-critical ed è improbabile che accediate ai dati condivisi contemporaneamente a qualsiasi altro processo, ometterò i semafori per semplicità.

[36] `https://en.wikipedia.org/wiki/Concurrency`

Questo programma fa una di queste due cose: se lo eseguite senza parametri da riga di comando, stampa il contenuto del segmento di memoria condivisa. Se gli fornite un parametro da riga di comando, lo memorizza nel segmento di memoria condivisa.

Ecco il codice per `shmdemo.c`[37]:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_SIZE 1024  /* make it a 1K shared memory segment */

int main(int argc, char *argv[])
{
    key_t key;
    int shmid;
    char *data;
    int mode;

    if (argc > 2) {
            fprintf(stderr, "usage: shmdemo [data_to_write]\n");
            exit(1);
    }

    /* make the key: */
    if ((key = ftok("shmdemo.c", 'R')) == -1) {
            perror("ftok");
            exit(1);
    }

    /* connect to (and possibly create) the segment: */
    if ((shmid = shmget(key, SHM_SIZE, 0644 | IPC_CREAT)) == -1) {
            perror("shmget");
            exit(1);
    }
```

[37] https://beej.us/guide/bgipc/source/examples/shmdemo.c

```
    /* attach to the segment to get a pointer to it: */
    data = shmat(shmid, (void *)0, 0);

    /* we _could_ use MAP_FAILED, but technically that's not */
    /* the defined return value. System V failed on this one! */
    if (data == (void *)(-1)) {
            perror("shmat");
            exit(1);
    }

    /* read or modify the segment, based on the command line: */
    if (argc == 2) {
            printf("writing to segment: \"%s\"\n", argv[1]);
            strncpy(data, argv[1], SHM_SIZE);
            data[SHM_SIZE-1] = '\0';
    } else
            printf("segment contains: \"%s\"\n", data);

    /* detach from the segment: */
    if (shmdt(data) == -1) {
            perror("shmdt");
            exit(1);
    }

    return 0;
}
```

Più comunemente, un processo si collega al segmento e rimane in esecuzione per un po' mentre altri programmi modificano e leggono il segmento condiviso. È interessante osservare un processo che aggiorna il segmento e vedere le modifiche apparire agli altri processi. Anche in questo caso, per semplicità, il codice di esempio non lo fa, ma è possibile vedere come i dati vengono condivisi tra processi indipendenti.

Inoltre, non c'è codice per rimuovere il segmento: assicuratevi di farlo quando avete finito di modificarlo.

10 File mappati in memoria

Arriva il momento in cui si desidera leggere e scrivere da e verso file in modo che le informazioni siano condivise tra i processi. Pensatela in questo modo: due processi aprono lo stesso file e lo leggono e lo scrivono, condividendo così le informazioni. Il problema è che a volte è complicato usare tutti quei `fseek()` e altre cose del genere. Non sarebbe più semplice se si potesse semplicemente mappare una sezione del file in memoria e ottenere un puntatore ad essa? In questo caso, si potrebbe semplicemente usare l'aritmetica dei puntatori per ottenere (e impostare) i dati nel file.

Beh, questo è esattamente ciò che è un file mappato in memoria. Ed è anche molto facile da usare. Poche semplici chiamate, unite ad alcune semplici regole, e si mappa come un matto.

10.1 Mapmake

Prima di mappare un file in memoria, è necessario ottenere un descrittore di file per esso utilizzando la chiamata di sistema `open()`:

```
int fd;

fd = open("mapdemofile", O_RDWR);
```

In questo esempio, abbiamo aperto il file in lettura/scrittura. È possibile aprirlo in qualsiasi modalità, ma deve corrispondere alla modalità specificata nel parametro `prot` della chiamata `mmap()`, riportata di seguito.

Per mappare in memoria un file, si utilizza la chiamata di sistema `mmap()`, definita come segue:

```
void *mmap(void *addr, size_t len, int prot,
           int flags, int fildes, off_t off);
```

Che sfilza di parametri! Eccoli, uno alla volta:

Parametro	Descrizione
`addr`	Questo è l'indirizzo in cui vogliamo che il file venga mappato. Il modo migliore per utili
`len`	Questo parametro è la lunghezza dei dati che vogliamo mappare in memoria. Può essere
`prot`	L'argomento "protection" consente di specificare il tipo di accesso di questo processo all
`flags`	Questi sono solo vari flag che possono essere impostati per la chiamata di sistema. Impo
`fildes`	Qui è dove si inserisce il descrittore di file aperto in precedenza.
`off`	Questo è l'offset nel file da cui si desidera iniziare la mappatura. Una restrizione: *deve* e

Per quanto riguarda i valori di ritorno, come avrete intuito, `mmap()` restituisce `MAP_FAILED` in caso di errore (il valore `-1` opportunamente convertito per il confronto) e imposta `errno`. In caso contrario, restituisce un puntatore all'inizio dei dati mappati.

In ogni caso, senza ulteriori indugi, faremo una breve demo che mappa la seconda "pagina" di un file in memoria. Per prima cosa useremo `open()` per ottenere il descrittore di file, poi useremo `getpagesize()` per ottenere la dimensione di una pagina di memoria virtuale e useremo questo valore sia per `len` che per `off`. In questo modo, inizieremo la mappatura dalla seconda pagina e mapperemo per la lunghezza di una pagina. (Sul mio sistema Linux, la dimensione della pagina è 4K.)

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

int fd, pagesize;
char *data;

fd = open("foo", O_RDONLY);
pagesize = getpagesize();
data = mmap((void*)0, pagesize, PROT_READ, MAP_SHARED, fd, pagesize);
```

Una volta eseguito questo codice stretch, è possibile accedere al primo byte della sezione mappata del file utilizzando `data[0]`. Si noti che qui avviene una notevole conversione di tipo. Ad esempio, `mmap()` restituisce `void*`, ma noi lo trattiamo come `char*`.

Si noti inoltre che abbiamo mappato il file `PROT_READ`, quindi abbiamo accesso in sola lettura. Qualsiasi tentativo di scrittura sui dati (`data[0] = 'B'`, ad esempio) causerà una violazione di segmentazione. Aprire il file `O_RDWR` con `prot` impostato su `PROT_READ|PROT_WRITE` se si desidera l'accesso in lettura e scrittura ai dati.

10.2 Annullamento della mappatura del file

Esiste, ovviamente, una funzione `munmap()` per annullare la mappatura in memoria di un file:

```
int munmap(void *addr, size_t len);
```

Questo semplicemente rimuove la mappatura della regione puntata da `addr` (restituita da `mmap()`) con lunghezza `len` (la stessa della lunghezza passata a `mmap()`). `munmap()` restituisce `-1` in caso di errore e imposta la variabile `errno`.

Una volta rimosso il mapping di un file, qualsiasi tentativo di accedere ai dati tramite il vecchio puntatore causerà un segmentation fault. Siete stati avvisati!

Un'ultima nota: il file verrà rimosso automaticamente se il programma termina, ovviamente.

10.3 Ancora concorrenza?!

Se più processi manipolano contemporaneamente i dati nello stesso file, potreste trovarvi nei guai. Potrebbe essere necessario bloccare il file o utilizzare semafori per regolarne l'accesso mentre un processo lo manipola. Consultate il documento sulla Memoria Condivisa per ulteriori informazioni (anche solo marginali) sulla concorrenza.

10.4 Esempio semplice

Bene, è di nuovo il momento del codice. Ho qui un programma demo che mappa la propria sorgente in memoria e stampa il byte trovato a qualsiasi offset specificato sulla riga di comando.

Il programma limita gli offset specificabili all'intervallo da 0 alla lunghezza del file. La lunghezza del file si ottiene tramite una chiamata a `stat()` che potresti non aver mai visto prima. Restituisce una struttura piena di informazioni sul file, un campo delle quali è la dimensione in byte. Abbastanza semplice.

Ecco il codice sorgente per `mmapdemo.c`[38]:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <errno.h>
```

[38] `https://beej.us/guide/bgipc/source/examples/mmapdemo.c`

```
int main(int argc, char *argv[])
{
    int fd;
    off_t offset;
    char *data;
    struct stat sbuf;

    if (argc != 2) {
            fprintf(stderr, "usage: mmapdemo offset\n");
            exit(1);
    }

    if ((fd = open("mmapdemo.c", O_RDONLY)) == -1) {
            perror("open");
            exit(1);
    }

    if (stat("mmapdemo.c", &sbuf) == -1) {
            perror("stat");
            exit(1);
    }


    offset = atoi(argv[1]);
    if (offset < 0 || offset > sbuf.st_size-1) {
        fprintf(stderr, "mmapdemo: offset must be in the range 0-%d\n", \
                                                            sbuf.st_size-1);
        exit(1);
    }

    data = mmap((caddr_t)0, sbuf.st_size, PROT_READ, MAP_SHARED, fd, 0);
    if (data == MAP_FAILED) {
        perror("mmap");
        exit(1);
    }

    printf("byte at offset %ld is '%c'\n", offset, data[offset]);

    return 0;
}
```

Questo è tutto. Compila quel programma ed eseguilo con una riga di comando tipo:

```
$ mmapdemo 30
byte at offset 30 is 'e'
```

Lascio a te il compito di scrivere programmi davvero interessanti usando questa chiamata di sistema.

10.5 Osservazioni sulla mappatura della memoria

Sarei negligente se non sottolineassi alcuni aspetti interessanti dell'utilizzo di file mappati su Linux. Innanzitutto, la memoria che il sistema operativo alloca come storage per i dati dei file mappati è *la stessa memoria* utilizzata per eseguire operazioni di buffering dei file quando altri processi eseguono operazioni `read()` e `write()`! Mentre `read()` e `write()` sono garantite atomiche da POSIX fino a una certa dimensione, questo non vale quando alcuni processi ignorano completamente le funzioni POSIX!

In secondo luogo, poiché stiamo ignorando queste funzioni POSIX, possiamo leggere e scrivere il contenuto del buffer senza considerare il blocco dei record che potrebbe essere applicato al descrittore di file (come discusso in una sezione precedente). Normalmente, questo non è un grosso problema: chi userebbe file mappati in memoria in un'applicazione mentre usa il blocco dei record in un'altra, quando entrambe accedono allo stesso file? Se il file è documentato per richiedere il blocco dei record, allora tutte le applicazioni dovrebbero utilizzarlo. Detto questo, nulla impedisce a un'applicazione di utilizzare il blocco di lettura e scrittura di cui abbiamo parlato in precedenza, immediatamente prima di aggiornare la memoria che appartiene al file mappato.

In terzo luogo, poiché stiamo bypassando quelle funzioni POSIX (sembro un disco rotto?), il sistema non è in grado di fornire strategie di readahead o writebehind significative. Al momento in cui scrivo, le versioni 4.x e successive del kernel Linux implementano un algoritmo che rileva quando si verificano due page fault adiacenti all'interno di un file mappato in memoria, ed esegue una quantità minima di readahead (solo due pagine, rispetto al readahead configurabile a livello di file system, che può superare i 256 KB). Non esiste alcun writebehind, poiché non esiste un modo pratico per rilevare quando vengono scritte pagine adiacenti con le attuali configurazioni hardware.

Infine, considerato tutto quanto sopra, ci sono ancora motivi molto validi per utilizzare i file mappati in memoria. Il principale è che tali file sono, per

definizione, "archiviazione persistente", il che significa che le applicazioni non devono creare lunghe funzioni `load()/save()` per i propri dati se utilizzano file mappati in memoria. Tuttavia, qualsiasi dato binario verrà scritto in modo dipendente dalla piattaforma (ad esempio, endianness), quindi è probabile che tali file non siano portabili.

10.6 Riepilogo

I file mappati in memoria possono essere molto utili, soprattutto su sistemi che non supportano segmenti di memoria condivisa. In effetti, i due sono molto simili sotto molti aspetti. (Anche i file mappati in memoria vengono salvati su disco, quindi questo potrebbe persino essere un vantaggio, no?) Con il file locking o i semafori, i dati in un file mappato in memoria possono essere facilmente condivisi tra più processi.

11 Socket Unix

Ricordate i FIFO? Ricordate come possono inviare dati solo in una direzione, proprio come i Pipe? Non sarebbe fantastico se poteste inviare dati in entrambe le direzioni come con un socket?

Beh, non sperate più, perché la risposta è qui: Socket di Dominio Unix! Nel caso vi stiate ancora chiedendo cos'è un socket, beh, è un canale di comunicazione bidirezionale, che può essere utilizzato per comunicare in un'ampia varietà di *domini*. Uno dei domini più comuni su cui i socket comunicano è Internet, ma non ne parleremo qui. Parleremo, tuttavia, dei socket nel dominio Unix, ovvero dei socket che possono essere utilizzati tra processi sullo stesso sistema Unix.

I socket Unix utilizzano molte delle stesse chiamate di funzione dei socket Internet, e non descriverò in dettaglio tutte le chiamate che utilizzo in questo documento. Se la descrizione di una determinata chiamata è troppo vaga (o se si desidera semplicemente saperne di più sui socket Internet), consiglio arbitrariamente la Guida di Beej alla programmazione di rete con i socket Internet[39]. Conosco personalmente l'autore.

11.1 Panoramica

Come ho detto prima, i socket Unix sono come le FIFO bidirezionali. Tuttavia, tutte le comunicazioni dati avverranno tramite l'interfaccia socket,

[39] `https://beej.us/guide/bgnet`

anziché tramite l'interfaccia file. Sebbene i socket Unix siano un file speciale nel file system (proprio come le FIFO), non si utilizzeranno `open()` e `read()`, ma `socket()`, `bind()`, `recv()`, ecc.

Quando si programma con i socket, di solito si creano programmi server e client. Il server rimane in ascolto delle connessioni in arrivo dai client e le gestisce. Questa situazione è molto simile a quella che si verifica con i socket Internet, ma con alcune piccole differenze.

Ad esempio, quando si descrive quale socket Unix si desidera utilizzare (ovvero, il percorso del file speciale che rappresenta il socket), si utilizza una `struct sockaddr_un`, che contiene i seguenti campi:

```
struct sockaddr_un {
    unsigned short sun_family;  /* AF_UNIX */
    char sun_path[108];
}
```

Questa è la struttura che passerete alla funzione `bind()`, che associa un descrittore di socket (un descrittore di file) a un determinato file (il cui nome è specificato nel campo `sun_path`).

11.2 Come fare per essere un server

Senza entrare troppo nei dettagli, descriverò i passaggi che un programma server deve solitamente seguire per svolgere il suo compito. Già che ci sono, proverò a implementare un "echo server" che riproduce tutto ciò che riceve sul socket.

Ecco i passaggi del server:

1. Chiamata `socket()`: Una chiamata a `socket()` con gli argomenti appropriati crea il socket Unix:

```
unsigned int s, s2;

struct sockaddr_un remote, local = {
    .sun_family = AF_UNIX,
    // .sun_path = SOCK_PATH,   // Can't do assignment to an array
};

int len;

s = socket(_AF_UNIX_, SOCK_STREAM, 0);
```

Il secondo argomento, `SOCK_STREAM`, indica a `socket()` di creare un socket di flusso. Sì, i socket di datagramma (`SOCK_DGRAM`) sono supportati nel dominio Unix, ma qui tratterò solo i socket di flusso. Per i curiosi, consultare la Guida alla Programmazione di Rete di Beej[40] per una buona descrizione dei socket di datagramma non connessi, che si applica perfettamente ai socket Unix. L'unica cosa che cambia è che ora si utilizza una `struct sockaddr_un` invece di una `struct sockaddr_in`.

Un'altra nota: tutte queste chiamate restituiscono `-1` in caso di errore e impostano la variabile globale errno per riflettere l'errore. Assicuratevi di eseguire il controllo degli errori.

1. Chiamata `bind()`: avete ottenuto un descrittore di socket dalla chiamata a `socket()`, ora volete associarlo a un indirizzo nel dominio Unix. (Quell'indirizzo, come ho detto prima, è un file speciale su disco.)

```
strcpy(local.sun_path, "/home/beej/mysocket");
unlink(local.sun_path);
len = strlen(local.sun_path) + sizeof(local.sun_family);

bind(s, (struct sockaddr *)&local, len);
```

Questo associa il descrittore di socket `"s"` all'indirizzo socket Unix `"/home/beej/mysocket"`. Si noti che abbiamo chiamato `unlink()` prima di `bind()` per rimuovere il socket se già esiste. Si otterrà un errore `EINVAL` se il file è già presente.

1. Chiamata `listen()`: questa istruzione al socket di ascoltare le connessioni in ingresso dai programmi client:

```
listen(s, 5);
```

Il secondo argomento, `5`, è il numero di connessioni in ingresso che possono essere messe in coda prima di chiamare `accept()`, come indicato di seguito. Se ci sono così tante connessioni in attesa di essere accettate, i client aggiuntivi genereranno l'errore `ECONNREFUSED`.

1. Chiamata `accept()`: questa funzione accetterà una connessione da un client. Questa funzione restituisce *un altro descrittore di socket*! Il vecchio descrittore è ancora in ascolto per nuove connessioni, ma questo nuovo è connesso al client:

[40] `https://beej.us/guide/bgnet`

```
len = sizeof(remote);
s2 = accept(s, &remote, &len);
```

Al termine di `accept()`, la variabile `remote` verrà riempita con la `struct sockaddr_un` del lato remoto e `len` verrà impostato alla sua lunghezza. Il descrittore `s2` è connesso al client ed è pronto per `send()` e `recv()`, come descritto nella Guida alla Programmazione di Rete[41].

1. Gestire la connessione e tornare ad `accept()`: in genere, in questo caso si desidera comunicare con il client (ci limiteremo a ripetere tutto ciò che ci invia), chiudere la connessione e quindi accettarne una nuova.

```
while (len = recv(s2, &buf, 100, 0), len > 0)
    send(s2, &buf, len, 0);

 /* loop back to accept() from here */
```

1. Chiudere la connessione: è possibile chiudere la connessione chiamando `close()` o shutdown[42].

Detto questo, ecco il codice sorgente per un server echo, echos.c[43]. Il suo compito è semplicemente attendere una connessione su un socket Unix (chiamato, in questo caso, `"echo_socket"`).

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

#define SOCK_PATH "echo_socket"

int main(void)
{
    int s, s2, len;
```

[41] `https://beej.us/guide/bgnet`
[42] `https://man.archlinux.org/man/shutdown.2`
[43] `https://beej.us/guide/bgipc/source/examples/echos.c`

```
struct sockaddr_un remote, local = {
        .sun_family = AF_UNIX,
        // .sun_path = SOCK_PATH,   // Can't do assignment to an array
};
char str[100];

if ((s = socket(AF_UNIX, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
}

strcpy(local.sun_path, SOCK_PATH);
unlink(local.sun_path);
len = strlen(local.sun_path) + sizeof(local.sun_family);
if (bind(s, (struct sockaddr *)&local, len) == -1) {
        perror("bind");
        exit(1);
}

if (listen(s, 5) == -1) {
        perror("listen");
        exit(1);
}

for(;;) {
        int done, n;
        printf("Waiting for a connection...\n");
        socklen_t slen = sizeof(remote);
        if ((s2 = accept(s, (struct sockaddr *)&remote, &slen)) == -1) {
                perror("accept");
                exit(1);
        }

        printf("Connected.\n");

        done = 0;
        do {
                n = recv(s2, str, sizeof(str), 0);
                if (n <= 0) {
                        if (n < 0) perror("recv");
```

```
                        done = 1;
                }

                if (!done)
                        if (send(s2, str, n, 0) < 0) {
                                perror("send");
                                done = 1;
                        }
        } while (!done);

        close(s2);
    }

    return 0;
}
```

Come potete vedere, tutti i passaggi sopra menzionati sono inclusi in questo programma: chiamata `socket()`, chiamata `bind()`, chiamata `listen()`, chiamata `accept()` ed eseguire alcuni `send()` e `recv()` di rete.

11.3 Cosa fare per essere un client

Ci vuole un programma per comunicare con il server di cui sopra, giusto? Solo che con il client è molto più semplice perché non è necessario eseguire fastidiosi `listen()` o `accept()`. Ecco i passaggi:

1. Chiama `socket()` per ottenere un socket di dominio Unix attraverso cui comunicare.
2. Imposta una `struct sockaddr_un` con l'indirizzo remoto (dove il server è in ascolto) e chiama `connect()` con questo come argomento.
3. Supponendo che non ci siano errori, sei connesso al lato remoto! Usa `send()` e `recv()` a tuo piacimento!

Che ne dici di un codice per comunicare con il server echo di cui sopra? Niente paura, ecco echoc.c[44]:

```
#include <stdio.h>
#include <stdlib.h>
```

[44] `https://beej.us/guide/bgipc/source/examples/echoc.c`

```
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

#define SOCK_PATH "echo_socket"

int main(void)
{
    int s, len;
    struct sockaddr_un remote = {
        .sun_family = AF_UNIX,
        // .sun_path = SOCK_PATH,   // Can't do assignment to an array
    };
    char str[100];

    if ((s = socket(AF_UNIX, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    printf("Trying to connect...\n");

    strcpy(remote.sun_path, SOCK_PATH);
    len = strlen(remote.sun_path) + sizeof(remote.sun_family);
    if (connect(s, (struct sockaddr *)&remote, len) == -1) {
        perror("connect");
        exit(1);
    }

    printf("Connected.\n");

    /* size in fgets() includes the null byte */
    while(printf("> "), fgets(str, sizeof(str), stdin), !feof(stdin)) {
        if (send(s, str, strlen(str)+1, 0) == -1) {
            perror("send");
            exit(1);
        }
```

```
        if ((len=recv(s, str, sizeof(str)-1, 0)) > 0) {
            str[len] = '\0';
            printf("echo> %s", str);
        } else {
            if (len < 0) perror("recv");
            else printf("Server closed connection\n");
            exit(1);
        }
    }

    close(s);

    return 0;
}
```

Nel codice client, ovviamente, noterete che ci sono solo poche chiamate di sistema utilizzate per impostare le cose: `socket()` e `connect()`. Dato che il client non accetterà alcuna connessione in ingresso, non c'è bisogno che `listen()`. Naturalmente, il client utilizza ancora `send()` e `recv()` per trasferire i dati. Questo è tutto.

11.4 `socketpair()`: pipe full-duplex veloci

Cosa succederebbe se si volesse usare una `pipe()`, ma si volesse usare una singola pipe per inviare e ricevere dati da *entrambi i lati*? Dato che le pipe sono unidirezionali (con eccezioni in SYSV), non è possibile! C'è una soluzione, però: usare un socket di dominio Unix, dato che possono gestire dati bidirezionali.

Che seccatura, però! Impostare tutto quel codice con `listen()` e `connect()` e tutto il resto solo per passare dati in entrambe le direzioni! Ma indovinate un po'! Non è necessario!

Esatto, esiste una bella chiamata di sistema chiamata `socketpair()`, che è abbastanza utile da restituirvi una coppia di *socket già connessi*! Non è necessario alcun lavoro aggiuntivo da parte vostra; potete usare immediatamente questi descrittori di socket per la comunicazione interprocesso.

Ad esempio, impostiamo due processi. Il primo invia un `char` al secondo, e il secondo converte il carattere in maiuscolo e lo restituisce. Ecco un semplice codice per fare proprio questo, chiamato spair.c[45] (senza controllo

[45] `https://beej.us/guide/bgipc/source/examples/spair.c`

degli errori per chiarezza):

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>

int main(void)
{
    int sv[2]; /* the pair of socket descriptors */
    char buf; /* for data exchange between processes */

    if (socketpair(AF_UNIX, SOCK_STREAM, 0, sv) == -1) {
            perror("socketpair");
            exit(1);
    }

    if (!fork()) {  /* child */
            read(sv[1], &buf, 1);
            printf("child: read '%c'\n", buf);
            buf = toupper(buf);  /* make it uppercase */
            write(sv[1], &buf, 1);
            printf("child: sent '%c'\n", buf);

    } else { /* parent */
            write(sv[0], "b", 1);
            printf("parent: sent 'b'\n");
            read(sv[0], &buf, 1);
            printf("parent: read '%c'\n", buf);
            wait(NULL); /* wait for child to die */
    }

    return 0;
}
```

Certo, è un modo costoso per convertire un carattere in maiuscolo, ma è il fatto che si stia verificando una comunicazione semplice che conta davvero.

Un'altra cosa da notare è che `socketpair()` accetta sia un dominio (`AF_UNIX`) che un tipo di socket (`SOCK_STREAM`). Questi possono essere qualsiasi valore consentito, a seconda delle routine del kernel che si desidera gestire e se si desidera usare socket stream o datagram. Ho scelto i socket `AF_UNIX` perché questo è un documento sui socket Unix e, a quanto pare, sono un po' più veloci dei socket `AF_INET`.

Infine, potresti essere curioso di sapere perché uso `write()` e `read()` invece di `send()` e `recv()`. Beh, in breve, ero pigro. Vedi, usando queste chiamate di sistema, non devo inserire l'argomento flags usato da `send()` e `recv()`, e comunque lo imposto sempre a zero. Naturalmente, i descrittori di socket sono semplicemente descrittori di file come tutti gli altri, quindi rispondono bene a molte chiamate di sistema per la manipolazione dei file.

12 Ulteriori risorse IPC

12.1 Libri

Ecco alcuni libri che descrivono alcune delle procedure che ho trattato in questa guida, oltre a dettagli specifici su Unix:

Bach, Maurice J. *The Design of the UNIX Operating System.* Pubblicato da Prentice-Hall, 1986. ISBN 0132017997[46].

W. Richard Stevens. *Unix Network Programming, volumi 1-2.* Pubblicato da Prentice Hall. ISBN per i volumi 1-2: 013141155149, 0130810819[47].

W. Richard Stevens. *Advanced Programming in the UNIX Environment.* Pubblicato da Addison Wesley. ISBN 0201433079[48].

12.2 Altra documentazione online

UNIX Network Programming Volume 2 home page[49]—include il codice sorgente tratto dal super raffinato libro di Stevens.

The Linux Programmer's Guide[50]—sezione approfondita su IPC.

UNIX System Calls and Subroutines using C[51]—contiene informazioni modeste su IPC.

The Linux Kernel[52]—come il kernel Linux implementa IPC.

[46] `https://beej.us/guide/url/unixdesign`
[47] `https://beej.us/guide/url/unixnet2`
[48] `https://beej.us/guide/url/advunix`
[49] `http://www.kohala.com/start/unpv22e/unpv22e.html`
[50] `http://tldp.org/LDP/lpg/node7.html`
[51] `https://users.cs.cf.ac.uk/Dave.Marshall/C/`
[52] `https://tldp.org/LDP/tlk/ipc/ipc.html`

12.3 Pagine man di Linux

Esistono pagine man di Linux. Se utilizzi una versione di Unix, consulta le tue pagine man, poiché potrebbero avere comporta,menti differenti sul tuo sistema.

- `accept()`[53],
- `bind()`[54],
- `connect()`[55],
- `dup()`[56],
- `exec()`[57],
- `exit()`[58],
- `fcntl()`[59],
- `fileno()`[60],
- `fork()`[61],
- `ftok()`[62],
- `getpagesize()`[63],
- `ipcrm`[64],
- `ipcs`[65],
- `kill`[66],
- `kill()`[67],

[53] https://man.archlinux.org/man/accept.2
[54] https://man.archlinux.org/man/bind.2
[55] https://man.archlinux.org/man/connect.2
[56] https://man.archlinux.org/man/dup.2
[57] https://man.archlinux.org/man/exec.2
[58] https://man.archlinux.org/man/exit.2
[59] https://man.archlinux.org/man/fcntl.2
[60] https://man.archlinux.org/man/fileno.3
[61] https://man.archlinux.org/man/fork.2
[62] https://man.archlinux.org/man/ftok.3
[63] https://man.archlinux.org/man/getpagesize.2
[64] https://man.archlinux.org/man/ipcrm.8
[65] https://man.archlinux.org/man/ipcs.8
[66] https://man.archlinux.org/man/kill.1
[67] https://man.archlinux.org/man/kill.2

- `listen()`[68],
- `lockf()`[69],
- `lseek()`[70] (per il campo `l_whence` nella `struct flock`),
- `mknod`[71],
- `mknod()`[72],
- `mmap()`[73],
- `msgctl()`[74],
- `msgget()`[75],
- `msgsnd()`[76],
- `munmap()`[77],
- `open()`[78],
- `pipe()`[79],
- `ps`[80],
- `raise()`[81],
- `read()`[82],
- `recv()`[83],

[68] `https://man.archlinux.org/man/listen.2`
[69] `https://man.archlinux.org/man/lockf.2`
[70] `https://man.archlinux.org/man/lseek.2`
[71] `https://man.archlinux.org/man/mknod.1`
[72] `https://man.archlinux.org/man/mknod.2`
[73] `https://man.archlinux.org/man/mmap.2`
[74] `https://man.archlinux.org/man/msgctl.2`
[75] `https://man.archlinux.org/man/msgget.2`
[76] `https://man.archlinux.org/man/msgsnd.2`
[77] `https://man.archlinux.org/man/munmap.2`
[78] `https://man.archlinux.org/man/open.2`
[79] `https://man.archlinux.org/man/pipe.2`
[80] `https://man.archlinux.org/man/ps.1`
[81] `https://man.archlinux.org/man/raise.3`
[82] `https://man.archlinux.org/man/read.2`
[83] `https://man.archlinux.org/man/recv.2`

- `semctl()`[84],
- `semget()`[85],
- `semop()`[86],
- `send()`[87],
- `shmat()`[88],
- `shmctl()`[89],
- `shmdt()`[90],
- `shmget()`[91],
- `sigaction()`[92],
- `signal()`[93],
- `signals`[94],
- `sigpending()`[95],
- `sigprocmask()`[96],
- `sigsetops`[97],
- `sigsuspend()`[98],
- `socket()`[99],

[84] https://man.archlinux.org/man/semctl.2
[85] https://man.archlinux.org/man/semget.2
[86] https://man.archlinux.org/man/semop.2
[87] https://man.archlinux.org/man/send.2
[88] https://man.archlinux.org/man/shmat.2
[89] https://man.archlinux.org/man/shmctl.2
[90] https://man.archlinux.org/man/shmdt.2
[91] https://man.archlinux.org/man/shmget.2
[92] https://man.archlinux.org/man/sigaction.2
[93] https://man.archlinux.org/man/signal.2
[94] https://man.archlinux.org/man/signal.7
[95] https://man.archlinux.org/man/sigpending.2
[96] https://man.archlinux.org/man/sigprocmask.2
[97] https://man.archlinux.org/man/sigsetopts.2
[98] https://man.archlinux.org/man/sigsuspend.2
[99] https://man.archlinux.org/man/socket.2

- `socketpair()`[100],
- `stat()`[101],
- `wait()`[102],
- `waitpid()`[103],
- `write()`[104].

[100] `https://man.archlinux.org/man/socketpair.2`
[101] `https://man.archlinux.org/man/stat.2`
[102] `https://man.archlinux.org/man/wait.2`
[103] `https://man.archlinux.org/man/waitpid.2`
[104] `https://man.archlinux.org/man/write.2`