

Programmazione C: Esercizi Avanzati

FIFO e Segnali

Scheda di Esercitazione

1 Esercizi Avanzati sulle FIFO

6. Server "Login" con Canali Dedicati e fork()

Simulare un server che gestisce la "connessione" di client multipli creando canali di comunicazione privati (FIFO dedicate) per ciascuno, utilizzando la `fork()` per gestire ogni client.

- **Server (server.c):**

- Crea una FIFO "ben nota" (es. `"/tmp/server_login"`).
- Si mette in attesa di richieste su questa FIFO. Una richiesta è una struttura contenente il PID del client: `struct login_req { pid_t pid; }`.
- Quando il server legge una richiesta:
 - (a) Crea un **processo figlio** (`fork()`) per gestire quel client.
 - (b) Il **Padre** torna immediatamente ad ascoltare sulla FIFO `server_login`.
 - (c) Il **Figlio** (gestore client):
 - * Costruisce i nomi di due FIFO dedicate (es. `"/tmp/[PID]_in"` e `"/tmp/[PID]_out"`).
 - * Le crea entrambe con `mkfifo()`.
 - * Implementa un servizio (es. "eco": legge da `[PID]_in` e riscrive su `[PID]_out`) finché il client non invia il comando "exit".

- * All'uscita, il figlio rimuove (`unlink()`) le due FIFO dedicate e termina.

- **Client (client.c):**

- Ottiene il proprio PID (`getpid()`).
- Invia la struttura `login_req` al server sulla FIFO `server_login`.
- Apre le *proprie* FIFO dedicate (invertendo input/output rispetto al server-figlio) e inizia a comunicare (es. invia stringhe e attende l'eco).

- **Bonus:** Il server padre deve gestire `SIGCHLD` per raccogliere i figli gestori terminati ed evitare processi zombie.

7. Trasferimento File via FIFO (Gestione EOF)

Testare la capacità di gestire flussi di dati di dimensioni sconosciute (il contenuto di un file) e la corretta segnalazione di fine trasmissione (EOF) su una pipe.

- **Mittente (file_sender.c):**

- Chiede all'utente il nome di un file da inviare (es. `testo.txt`).
- Crea una FIFO `"/tmp/file_transfer"`.
- Apre la FIFO in scrittura (`O_WRONLY`).
- Apre il file locale (es. `testo.txt`) in lettura.
- Legge il file a blocchi (es. buffer di 1024 byte) e scrive ogni blocco sulla FIFO.
- **Importante:** Quando il file locale è finito (`read()` dal file restituisce 0), il mittente **chiude la FIFO** (`close(fd_fifo)`).

- **Destinatario (file_receiver.c):**

- Apre la FIFO `"/tmp/file_transfer"` in lettura (`O_RDONLY`). (Si bloccherà finché il sender non la apre in scrittura).
- Apre un file di output (es. `copia.txt`) in scrittura.
- Entra in un ciclo: legge dati dalla FIFO (a blocchi) e li scrive sul file di output.
- **Gestione EOF:** Quando `read()` dalla FIFO restituisce 0, significa che il mittente ha chiuso la sua estremità di scrittura

(fine del file). Il destinatario deve uscire dal ciclo, chiudere il file di output e chiudere la FIFO.

- Il destinatario rimuove la FIFO con `unlink()`.

8. Server Multiplexato con `select()`

Introdurre l'I/O multiplexing. Il server deve ascoltare contemporaneamente due (o più) FIFO senza bloccarsi su una sola.

- **Prerequisito:** Spiegare l'uso di `select()` con `FD_ZERO`, `FD_SET`, `FD_ISSET`.
- **Programma (`multiplexer.c`):**
 - Crea due FIFO: `"/tmp/canale_A"` e `"/tmp/canale_B"`.
 - Apre entrambe le FIFO in lettura **non bloccante** (`O_RDONLY` | `O_NONBLOCK`).
 - Definisce un `fd_set` (set di file descriptor).
 - Entra in un ciclo infinito:
 - (a) Azzera il set (`FD_ZERO`).
 - (b) Aggiunge i due file descriptor delle FIFO al set (`FD_SET`).
 - (c) Chiama `select()` per attendere che almeno uno dei due FD diventi leggibile.
 - (d) Dopo `select()`, controlla quale FD è pronto:
 - (e) Se `FD_ISSET(fd_A)` è vero: legge i dati da `canale_A` e li stampa (es. "A: [dati]").
 - (f) Se `FD_ISSET(fd_B)` è vero: legge i dati da `canale_B` e li stampa (es. "B: [dati]").
- **Test (per l'alunno):** Aprire due terminali. In uno, scrivere su `canale_A` (es. `echo "ciao" > /tmp/canale_A`). Nell'altro, scrivere su `canale_B`. Verificare che il server gestisca entrambi gli input non appena arrivano.

2 Esercizi Avanzati sui Segnali

6. Coordinatore di Shutdown (SIGTERM e Processi Figli)

Combinare la gestione dei segnali esterni (SIGTERM) con la gestione dei processi figli (SIGCHLD) per un'uscita pulita (graceful shutdown).

- **Programma (coordinator.c):**

- Il processo padre (Main) crea $N = 3$ processi figli. Salva i loro PID in un array globale.
 - I figli entrano in un ciclo infinito (es. `while(1) { printf("Figlio %d vivo", getpid()); sleep(2); }`).
 - Il padre installa un handler per SIGCHLD e uno per SIGTERM.
 - **Handler SIGCHLD:** Deve usare `waitpid(-1, NULL, WNOHANG)` in un ciclo `while` per raccogliere *tutti* i figli terminati (evitando zombie) e decrementare un contatore globale dei figli attivi.
 - **Handler SIGTERM:** (Questo è il nucleo dell'esercizio)
 - (a) Stampa "Ricevuto SIGTERM: avvio shutdown..."
 - (b) Itera sull'array dei PID figli e invia SIGTERM a *tutti* i figli ancora vivi usando `kill()`.
 - Il padre, nel `main()`, attende la terminazione di tutti i figli (es. `while(contatore_figli > 0) { pause(); }`) prima di terminare sé stesso.
- **Test:** Eseguire il programma e, da un altro terminale, inviare `kill -TERM [PID_PADRE]`. Verificare che il padre inoltri il segnale ai figli, che questi terminino, e che il padre li raccolga correttamente prima di uscire.

7. Sezione Critica e Mascheramento (sigprocmask)

Dimostrare come i segnali possano interrompere operazioni non atomiche (creando *race condition*) e come evitarlo mascherando i segnali durante la sezione critica.

- **Obiettivo:** Proteggere un'operazione non atomica (es. aggiornamento di una struttura dati complessa) dall'interruzione di un segnale.

- **Programma (critical_section.c):**

- Definire due contatori globali: `volatile long contatoreA = 0;` e `volatile long contatoreB = 0;`. L'invariante del programma è che `contatoreA` deve essere sempre uguale a `contatoreB`.
- Installare un handler per `SIGINT` (`CTRL+C`) che stampa i valori di `contatoreA` e `contatoreB` e poi termina (`exit(1)`).
- Nel `main()`, entrare in un ciclo infinito `while(1)`.
- **Sezione Critica (errata):** All'interno del ciclo:

```

1 contatoreA++;
2 // Simula un calcolo lungo o un context switch
3 usleep(100); // 100 microsecondi
4 contatoreB++;

```

- **Test (Problema):** Eseguire il programma e premere `CTRL+C`. Molto probabilmente, il segnale arriverà *tra* i due incrementi (è la race condition), e l'handler stamperà valori diversi per `contatoreA` e `contatoreB`, violando l'invariante.

- **Soluzione (Esercizio):**

- Modificare il ciclo `while` usando `sigprocmask()`.
- Definire una maschera `sigset_t mask, oldmask;` e aggiungere `SIGINT` a `mask`.
- **Prima** della sezione critica: bloccare `SIGINT` → `sigprocmask(SIG_BLOCK, &mask, &oldmask);`
- Eseguire la sezione critica (i due incrementi).
- **Dopo** la sezione critica: ripristinare la vecchia maschera → `sigprocmask(SIG_SETMASK, &oldmask, NULL);`
- **Test (Soluzione):** Verificare che ora, premendo `CTRL+C` in qualsiasi momento, l'handler stamperà sempre valori uguali per i due contatori.

8. Gestione Sincrona dei Segnali (sigwait)

Introdurre un approccio alternativo alla gestione dei segnali, trattandoli in modo sincrono (come un evento da "leggere") anziché asincrono (con un handler), utile per i thread.

- **Obiettivo:** Creare un programma che attende specifici segnali in un loop principale, senza usare handler, per gestire lo stato del programma in modo controllato.
- **Programma (sigwaiter.c):**
 - **Mascheramento Totale:** All'inizio del `main()`, definire una maschera `sigset_t mask`; e aggiungervi i segnali `SIGINT`, `SIGUSR1` e `SIGUSR2`.
 - Usare `sigprocmask(SIG_BLOCK, &mask, NULL)`; per bloccare questi segnali per *tutto* il processo. (Questo è fondamentale, altrimenti verrebbe eseguita l'azione di default, es. terminazione per `SIGINT`).
 - Stampare il PID e le istruzioni (es. "Usa `kill -USR1 [PID]` per incrementare, `-USR2` per decrementare, `-INT` per uscire").
 - Definire un contatore `int contatore = 0;`.
 - Entrare in un ciclo `while(1)`.
 - All'interno del ciclo, chiamare `sigwait(&mask, &sig_received);`.
 - `sigwait()` sospende il programma finché uno dei segnali in `mask` non viene ricevuto, e scrive il segnale ricevuto in `sig_received`.
 - Usare uno `switch` sulla variabile `sig_received`:
 - * `case SIGUSR1: contatore++; stampa "Contatore: [valore]".`
 - * `case SIGUSR2: contatore--; stampa "Contatore: [valore]".`
 - * `case SIGINT: stampa "Ricevuto SIGINT. Uscita." e fa break dal ciclo.`
- **Discussione (per gli studenti):** Quali sono i vantaggi di questo approccio rispetto a un handler? (Risposta: Si evitano i problemi di *race condition* e di funzioni *async-signal-safe*, perché il segnale viene gestito nel flusso principale e controllato del programma, non in modo asincrono).