Inter Process Communication (IPC)

Meccanismi, API POSIX e esempi (Pipe, FIFO, Memoria Condivisa)

Corso: Sistemi di Calcolo 2
Docente: Riccardo Lazzeretti
Adattato da: M. Kerrisk, *The Linux Programming Interface*

Indice

1 Spazio di indirizzamento di un processo

- Un processo può accedere solo al **proprio** spazio di indirizzamento.
- Ogni processo ha il proprio spazio di indirizzamento separato.
- Il **kernel** può accedere a tutto.

2 Processi cooperanti e necessità di IPC

Processi indipendenti: non possono influenzare né essere influenzati dall'esecuzione di altri processi.
Processi cooperanti: possono influenzare ed essere influenzati dall'esecuzione altrui.
Ragioni per cooperare:

- Condivisione di informazioni.
- Aumento della velocità di calcolo (parallelismo).
- Modularità (suddivisione in moduli/sotto-task).
- Comodità (più processi semplificano alcune applicazioni).

Figura (descrizione): un'applicazione suddivisa in più processi cooperanti.

3 Che cos'è l'IPC

Meccanismo per far comunicare processi e per **sincronizzare** le loro azioni.

4 Modelli di comunicazione IPC

Due modelli fondamentali (forniti dalla maggior parte dei sistemi):

1. **Memoria condivisa**: i processi usano una regione di memoria condivisa per scambiarsi dati.
2. **Passaggio di messaggi**: i processi si inviano messaggi tramite il kernel.

Figura (descrizione): confronto tra memoria condivisa e coda di messaggi nel kernel.

5 Memoria condivisa: concetti e produttore/consumatore

5.1 Meccanismo di memoria condivisa

- Una regione di memoria è stabilita fra due o più processi *con l'aiuto del kernel* (chiamate di sistema).
- I processi possono leggere/scrivere la regione come normale memoria (tramite puntatori).
- Una volta mappata, il kernel non è coinvolto sugli accessi: **veloce**.

Figura (descrizione): Processi A e B con una regione condivisa nel mezzo.

5.2 Bounded buffer con semafori (schema dalle slide)

Stato del buffer in memoria condivisa:

```
1 int out;
2 int in;
3 item buffer[BUFFER_SIZE];  // in shared memory
```

Buffer pieno/vuoto (intuizione dalle slide). *Figure descritte*: due casi con indici `in` e `out`.

- **Buffer pieno**: `in == out`; `sem_empty.val == 0`; `sem_filled.val == BUFFER_SIZE`.
- **Buffer vuoto**: `in == out`; `sem_empty.val == BUFFER_SIZE`; `sem_filled.val == 0`.

5.3 Codice produttore/consumatore (con semafri nominati)

Listing 1: Produttore (memoria condivisa + semafori)

```
while (true) {
  /* produce ITEM */
  sem_wait(sem_empty);   // numero di slot liberi
  sem_wait(sem_cs);      // mutua esclusione
  writeToBuffer(ITEM);
  in = (in + 1) % BUFFER_SIZE;
  sem_post(sem_cs);
  sem_post(sem_filled);  // elemento disponibile
}
```

Listing 2: Consumatore (memoria condivisa + semafori)

```
while (true) {
  sem_wait(sem_filled);  // attendi elemento
  sem_wait(sem_cs);      // mutua esclusione
  readFromBuffer(ITEM);
  out = (out + 1) % BUFFER_SIZE;
  sem_post(sem_cs);
  sem_post(sem_empty);   // libera uno slot
  /* consume ITEM */
}
```

5.4 API POSIX per memoria condivisa

- `shm_open()` — crea/apre un oggetto di memoria condivisa, restituisce un *file descriptor*.
- `ftruncate()` — imposta la dimensione dell'oggetto.
- `mmap()` — mappa la pagina nello spazio di indirizzi del processo (poi si accede via puntatore).
- `close()` — chiude il descrittore.
- `shm_unlink()` — rimuove l'oggetto (i processi con riferimenti aperti possono continuare ad accedervi).

Nota di accuratezza: nelle slide compare anche `ltrunc()`, ma in POSIX l'interfaccia corretta è `ftruncate()`.[1]

5.5 Esempio: scrivere su memoria condivisa

```
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main(void) {
  const int SIZE = 4096;
  const char *name = "MY_PAGE";
  const char *msg  = "Hello World!";
```

[1] Allo stesso modo, `shm_unlink()` prende il *nome* e non un file descriptor.

```
  int shm_fd;
  char *ptr;

  shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
  ftruncate(shm_fd, SIZE);
  ptr = (char*) mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);
  sprintf(ptr, "%s", msg);
  close(shm_fd);
  return 0;
}
```

5.6 Esempio: leggere da memoria condivisa

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

int main(void) {
  const int SIZE = 4096;
  const char *name = "MY_PAGE";
  int shm_fd;
  char *ptr;

  shm_fd = shm_open(name, O_RDONLY, 0666);
  ptr = (char*) mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);
  printf("%s\n", ptr);
  close(shm_fd);
  shm_unlink(name);  // attenzione: prende il nome
  return 0;
}
```

6 Passaggio di messaggi

6.1 Primitive e formato

Primitive: `send(dest, msg)` e `receive(src, msg)` (o versioni senza specifica di src/dest).
Formato messaggio: header + body. In UNIX classico: nessun ID, solo *message type*.
Controllo: gestione buffer pieno, numeri di sequenza, priorità.
Accodamento: tipicamente FIFO, ma possono essere supportate priorità.

7 Pipe (anonime) e FIFO (named pipe)

7.1 Concetti di base sulle pipe

- Consentono a più processi di comunicare come file sequenziali.
- Comunicazione **monodirezionale**.
- I dati letti **scompaiono** dalla pipe (a meno di nuova scrittura).
- Implementate come **buffer** in kernel-space (spesso 4096 byte o multipli).
- Un processo che legge da pipe vuota **si blocca** finché non arrivano dati.
- Un processo che scrive su pipe piena **si blocca** finché non c'è spazio.

7.2 Tipi di pipe

Unnamed/anonime Esistono tra un processo padre e i suoi figli (create con `pipe()`); tipiche per IPC a breve vita.

Named/FIFO Oggetti con un nome nel file system (`mkfifo`); non richiedono relazione padre-figlio; adatte a IPC di lunga durata; possono essere usate in modo bidirezionale (con due FIFO o protocolli).

7.3 API `pipe()` e semantica

```
int pipe(int fd[2]);
/* fd[0]: lettura; fd[1]: scrittura. Ritorna 0/-1. */
```

Dimensione: fino a 16 pagine ($\approx$ 65536 byte con pagine da 4096 B) su Linux.
Atomicità scritture: una `write()` di $n \leq PIPE\_BUF$ byte è atomica ($PIPE\_BUF = 4096$ B). Se $n > PIPE\_BUF$ la scrittura può interlecciarsi con altre `write()` e il chiamante resta bloccato finché non scrive tutti gli n byte (modalità bloccante).
EOF e SIGPIPE:

- `read()` ritorna 0 quando *tutti* gli scrittori (`fd[1]`) sono chiusi.
- Scrivere quando non ci sono lettori (`fd[0]` chiusi) genera **SIGPIPE** (*Broken pipe*).

7.4 Evitare deadlock con le pipe

- Ogni processo deve **chiudere** i descrittori di pipe che non usa con `close()`.
- Un lettore che eredita `fd[0]` e `fd[1]` deve chiudere la propria copia di `fd[1]` **prima** di leggere da `fd[0]` (altrimenti l'EOF non arriva mai $\Rightarrow$ rischio di stallo).

7.5 Esempio: trasferimento stringhe via pipe

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

#define ERRORE(x) do { perror(x); exit(1); } while (0)

int main(void) {
  char msg[30];
  int fd[2];
  if (pipe(fd) == -1) ERRORE("pipe");
  pid_t pid = fork();
  if (pid == -1) ERRORE("fork");

  if (pid == 0) {            // figlio: lettore
    close(fd[1]);
    ssize_t n;
    while ((n = read(fd[0], msg, sizeof msg)) > 0) {
      fwrite(msg, 1, n, stdout);
    }
    close(fd[0]);
    _exit(0);
  } else {                   // padre: scrittore
    close(fd[0]);
```

```
    puts("digita testo (quit per terminare):");
    do {
      if (!fgets(msg, sizeof msg, stdin)) break;
      write(fd[1], msg, strlen(msg));
    } while (strcmp(msg, "quit\n") != 0);
    close(fd[1]);
    int status; wait(&status);
  }
  return 0;
}
```

7.6 FIFO (named pipe): creazione e uso

```
#include <sys/stat.h>
int mkfifo(const char *name, mode_t mode);
/* Restituisce 0/-1; rimozione con unlink(name). */
```

Apertura bloccante: aprire in sola lettura (scrittura) blocca finché un altro processo non apre in scrittura (lettura).
O_NONBLOCK: possibile apertura non bloccante via flag a `open()`.
Assenza lettori: scrivere su FIFO senza lettori genera **SIGPIPE**.

7.7 Esempio server (FIFO `/serv`)

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

typedef struct {
  long type;
  char fifo_response[20];
} request;

int main(void){
  const char *SERV = "/serv";
  if (mkfifo(SERV, 0666) == -1) { perror("mkfifo"); /* potrebbe esistere */ }
  int fd = open(SERV, O_RDONLY);
  if (fd == -1) { perror("open SERV"); exit(1); }
  while (1) {
    request r;
    ssize_t ret = read(fd, &r, sizeof r);
    if (ret <= 0) break;
    printf("Richiesto servizio (fifo risposta = %s)\n", r.fifo_response);
    /* ... dispatch per tipo ... */
    sleep(10); // simula ritardo
    int fdc = open(r.fifo_response, O_WRONLY);
    const char *response = "fatto";
    write(fdc, response, 20);
    close(fdc);
  }
  close(fd);
  unlink(SERV);
  return 0;
}
```

7.8 Esempio client (FIFO `/serv` + FIFO risposta)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

typedef struct { long type; char fifo_response[20]; } request;

int main(void){
  request r; char response[20];
  printf("Seleziona una lettera minuscola: ");
  if (scanf("%19s", r.fifo_response) != 1) return 1;
  if (r.fifo_response[0] < 'a' || r.fifo_response[0] > 'z') {
    printf("Carattere non valido\n"); return 1;
  }
  r.fifo_response[1] = '\0';
  if (mkfifo(r.fifo_response, 0666) == -1) {
    printf("servente sovraccarico - riprovare\n"); return 1;
  }
  int fd = open("/serv", O_WRONLY);
  if (fd == -1) { perror("open /serv"); unlink(r.fifo_response); return 1; }
  write(fd, &r, sizeof r);
  close(fd);

  int fdc = open(r.fifo_response, O_RDONLY);
  read(fdc, response, sizeof response);
  printf("risposta = %s\n", response);
  close(fdc);
  unlink(r.fifo_response);
  return 0;
}
```

7.9 Confronto pipe vs FIFO

- Aprire una **pipe** restituisce *due* descrittori (lettura e scrittura); occorre chiudere quello non usato.
- Aprire una **FIFO** avviene *in lettura* oppure *in scrittura*; gestione simile a pipe dopo l'apertura.
- Le FIFO persistono nel file system finché non vengono `unlink()`-ate; le pipe anonime sono effimere (legate ai processi).