Università di Roma Tor Vergata
Corso di Laurea triennale in Informatica

Sistemi operativi e reti

A.A. 2016-17

Pietro Frasca

Lezione 20

Giovedì 22-12-2016

Comunicazione: pipe

- In Unix, processi possono comunicare utilizzando la memoria condivisa e lo scambio di messaggi. Un'altra tecnica di comunicazione si basa sulle **pipe (tubo)**.
- Due tipi comuni di pipe utilizzate su vari sistemi operativi, compresi UNIX e Windows sono le ***pipe senza nome (unnamed pipe***) e le ***pipe con nome (named pipe)*.**
- La chiamata di sistema **pipe (tubo**) implementa l'astrazione di un canale per consentire la comunicazione tra processi.
- La dimensione di una pipe è limitata ed è stabilita dalla costante di sistema **BUFSIZ** che generalmente è di 4KB.
- L'accodamento dei messaggi nella pipe avviene in modalità FIFO.
- La comunicazione mediante pipe è **unidirezionale** dato che si può accedere ad essa in lettura (o ricezione) da un solo estremo e in scrittura (o trasmissione) dall'altro estremo.

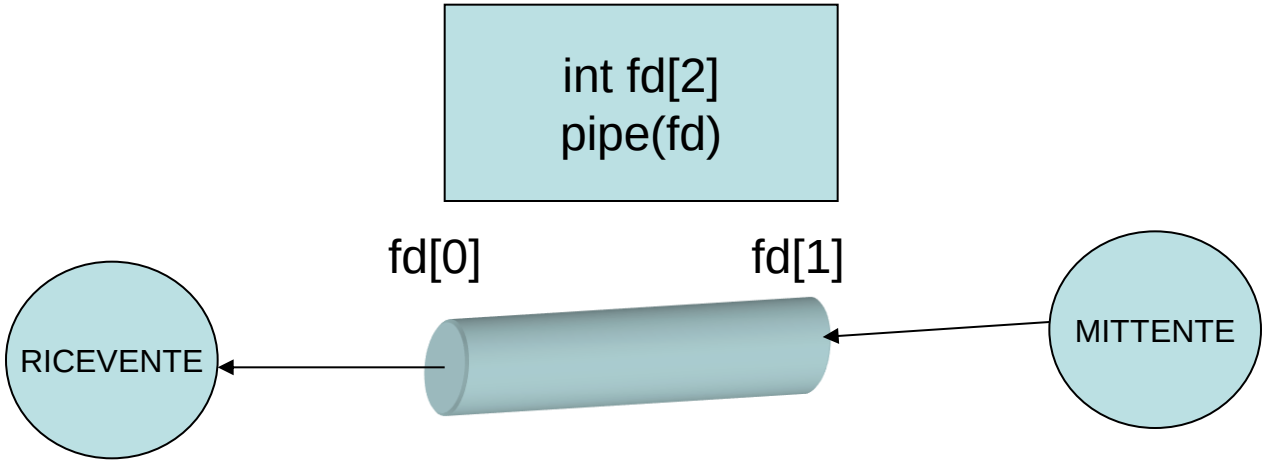
- La pipe è un canale di comunicazione di tipo ***da-molti-a-molti***, in quanto mediante la stessa pipe più processi possono inviare messaggi e più processi possono riceverli.
- Le pipe vengono gestite allo stesso modo dei file. In particolare, ogni lato di accesso alla pipe è rappresentato da *un file descriptor*. I processi utilizzano le funzioni di lettura e scrittura di file **read** e **write** rispettivamente per ricevere o inviare messaggi dalla o alla pipe.
- Per creare una pipe si utilizza la system call:

```
int pipe (int fd[2]);
```

- in cui il parametro **fd** è un vettore di 2 **file descriptor**, che sono inizializzati dalla pipe. In caso di successo, l'intero **fd[0]** rappresenta il lato di lettura della pipe e **fd[1]** il lato di scrittura della pipe.
- La pipe restituisce zero se è eseguita con successo o un valore negativo, in caso di fallimento.

- Ogni lato di accesso alla pipe, quindi è rappresentato da un file descriptor. In particolare, un processo mittente, per inviare messaggi utilizza la system call **write** sul file descriptor **fd[1]**; analogamente un processo destinatario può ricevere messaggi mediante la system call **read** sul file descriptor **fd[0]** .
- **I processi che possono comunicare attraverso una stessa pipe sono il processo che l'ha chiamata e tutti i suoi discendenti**. Infatti, poichè la pipe è identificata dalla coppia di file descriptor (fd[0] , fd[1]) appartenenti allo spazio di indirizzamento del processo padre, ogni processo discendente dal padre eredita una copia di (fd[0], fd[1]) e una copia della tabella dei file aperti del processo.

- Poiché la pipe ha capacità limitata, come nel problema produttore/consumatore, è necessario sincronizzare i processi in caso di canale pieno e/o vuoto.
- Con la pipe **la sincronizzazione è implicitamente fornita dalle funzioni read e write** che funzionano **in modalità bloccante.** Pertanto, nel caso di pipe vuota, un processo destinatario chiamando la read attende fino all'arrivo del prossimo messaggio; analogamente, in caso di pipe piena un processo mittente chiamando la write si sospende in attesa di spazio libero.



`read (fd[0],mess,dim)` `write (fd[1],mess,dim)`

Esempio uso di pipe

Un processo crea tramite fork() un processo figlio. I due processi, padre e figlio, comunicano attraverso pipe, usando le write e read.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define DIM 256
#define LEGGI 0
#define SCRIVI 1
int main() {
  int  n, pd[2];
  int  pid;
  char messaggio[DIM];

  if (pipe(pd) < 0) {
   printf ("errore pipe");
       exit(1);
  }
```

```
if ( (pid = fork()) < 0) {
    printf("errore fork");
    exit(1);
  } else if (pid > 0) {
    /* padre */
    close(pd[LEGGI]); // chiude il canale che non usa
    write(pd[SCRIVI], "Ciao, figlio", DIM);
  } else {
    /* figlio */
    close(pd[SCRIVI]);// chiude il canale che non usa
    n = read(pd[LEGGI], messaggio, DIM);
    printf("%s \n", messaggio);
  }
}
```

Comunicazione: pipe con nome

- Le pipe senza nome forniscono un meccanismo semplice per consentire ai processi di comunicare. Queste pipe esistono solo quando i processi comunicano tra loro. Una volta che i processi hanno terminato la comunicazione e terminano, le pipe cessano di esistere.
- Le *named pipe* forniscono uno strumento di comunicazione più potente. La comunicazione può avvenire anche se i processi non hanno alcuna relazione genitore-figlio. Una volta creata una pipe con nome, più processi possono usarla per comunicare. Inoltre, le pipe con nome continuano ad esistere anche dopo che i processi comunicanti hanno terminato la loro esecuzione.
- Le named pipe sono chiamate FIFO nei sistemi UNIX. Una volta create, vengono visualizzate come tipici file nel file system.

- Un file FIFO si crea con la chiamata di sistema mknod (o con mkfifo) e sono gestiti con le funzioni ordinarie open, read, write, e close.
- Il file FIFO continuerà ad esistere finché non viene esplicitamente eliminato dal file system. Anche se le FIFO consentono la comunicazione bidirezionale, è consentita solo la trasmissione half-duplex. Se i dati devono viaggiare in entrambe le direzioni, si utilizzano due FIFO. Inoltre, i processi in comunicazione devono risiedere sulla stessa macchina.
- Se è necessaria la comunicazione tra host remoti, devono essere utilizzate le socket.

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
struct message_t {
  int id;
  char text[32];
} msg;
int main(int argc,char *argv[]) {
  int fd, i=0, n=10;
  int size=sizeof(msg);
  mknod("mia_pipe",S_IFIFO|0666,0);
  if (argc == 2)
    fd = open("mia_pipe",O_WRONLY);
  else
    fd = open(«mia_pipe", O_RDONLY);
```

```
  strcpy(msg.text,"Auguri");
  for (i=0;i<n;i++)
    if (argc == 2){
      msg.id=rand()%100+1;
      printf("produttore: %s %d \n",msg.text,msg.id);
      write(fd, &msg, size);
      sleep(1);
    } else {
      read(fd,&msg,size);
      printf("consumatore: %s %d\n",msg.text,msg.id);
      sleep(1);
    }
}
```

Sospensione di processi

La SC pause

```
int pause (void);
```

- sospende il processo chiamante fino a quando esso riceve un segnale (che non deve essere ignorato)
- restituisce -1 in caso di errore

La SC sleep

```
unsigned int sleep (unsigned int seconds);
```

- sospende il processo chiamante per un intervallo di tempo, espresso in secondi, specificato nell'argomento.
- restituisce il numero di secondi mancanti a soddisfare la richiesta: la sospensione programmata con sleep può essere infatti più breve di quanto richiesto.

Ad esempio nel caso il processo sia abilitato a gestire segnali, l'intervallo di tempo di attesa programmato con sleep viene annullato e la funzione ritorna la quantità di tempo che mancava alla scadenza.

usleep

- Sospende il processo chiamante per un intervallo di tempo, espresso in microsecondi, specificato nel parametro della funzione.

#include <unistd.h>

int usleep(unsigned long usec)

I Threads nei sistemi Linux e Unix

- Linux e le recenti versioni di Unix supportano i *thread a livello kernel* e pertanto il thread è l'unità di scheduling.
- Come già descritto, il processi hanno uno spazio di indirizzamento privato e quindi non possono condividere dati tra loro. I processi possono scambiarsi dati mediante messaggi o allocando segmenti di memoria condivisa che dovranno essere gestiti in mutua esclusione (mediante opportune system call).
- Il *thread,* invece, possono condividere tra loro lo spazio di indirizzi cui essi appartengono.
- Linux per la generazione di thread dispone della chiamata di sistema **clone**.

Gestione di thread nello standard POSIX: la libreria pthreads

- La SC **clone** è presente solo in Linux e non in altri sistemi Unix e quindi non è portabile.
- Per realizzare applicazioni con i thread sia in Linux che in Unix e ottenere la portabilità delle applicazioni, si può utilizzare la libreria POSIX ***pthreads***.

Caratteristiche dei threads POSIX/Linux

- La libreria pthread definisce il tipo **pthread_t** per definire i thread all'interno di programmi concorrenti (la definizione è contenuta nel file header **pthread.h**).
- Lo standard POSIX prevede che i thread siano creati all'interno di un processo.

- In particolare, al codice della function main (nel caso del C) corrisponde il thread iniziale.
- Il thread iniziale può generare, attraverso chiamate di sistema, nuovi thread che possono condividere uno spazio di indirizzi (ad esempio variabili globali e function).

Creazione di thread

- La creazione di un *thread* si ottiene mediante la chiamata di sistema **pthread_create**:
- ```
  # include <pthread.h>
  int pthread_create (pthread_t* T,
                      pthread_attr_t* attr,
                      void *codice (void* arg);
  ```

  dove:
  - **T**: è il puntatore alla variabile di tipo **pthread_t** del nuovo thread;
  - **attr**: può essere usato per specificare eventuali attributi da associare al thread come ad esempio la priorità del thread, oppure NULL (valori di default).
  - **codice**: è il puntatore alla funzione che contiene il codice del thread creato;
  - **arg**: è il puntatore all'eventuale vettore contenente i valori dei parametri da passare alla funzione **codice**.
  ```

- La chiamata **pthread_create** restituisce
  - **0 in caso di successo**
  - **oppure un codice di errore**.

- Ogni nuovo *thread* va in esecuzione in modo concorrente con il padre e **condivide** con esso lo stesso spazio di indirizzamento del programma nel quale è definito.

Terminazione di un thread

- Un thread può terminare chiamando:

```
void pthread_exit (void * stato) ;
```

  - **stato** è il puntatore alla variabile che contiene il valore eventualmente restituito dal thread.

- Un thread padre può sospendersi in attesa della terminazione di un thread figlio con:

```
int pthread_join(pthread_t th, void *stato);
```

  - **th** è il **TID** del particolare thread da attendere e
  - **stato** è il puntatore alla variabile dove verrà memorizzato il valore eventualmente restituito dal thread (con pthread_exit).

```
1.   #include <stdio.h>
2.   #include <stdlib.h>
3.   #include <pthread.h>
4.   //variabili condivise:
5.   char MSG [] ="Ciao!";
6.   void *codice_T (void *arg){
7.     int i;
8.     for (i=0; i<5;i++) {
9.       printf ("Thread %s: %s\n“, (char*)arg, MSG);
10.      sleep(1) ; /* sospensione per l secondo. */
11.    }
12.    pthread_exit (0) ;
13.  }
```

```
1.   int main() {
2.     pthread_t th1, th2;
3.     int ret;
4.     /* creazione primo thread: */
5.     if (pthread_create(&th1,NULL,codice_T, "1") < 0) {
6.       fprintf (stderr, "Errore di creazione thread 1\n");
7.       exit(1);
8.     }
9.     /* creazione secondo thread: */
10.    if (pthread_create(&th2,NULL,codice_T, "2") < 0) {
11.      fprintf (stderr, "Errore di creazione thread 2\n");
12.      exit(1);
13.    }
14.    ret = pthread_join(th1,NULL);
15.    if (ret != 0)
16.      fprintf (stderr, "join fallito %d \n“, ret);
17.    else printf ("terminato il thread 1 \n) ;
18.    ret = pthread_join(th2,NULL);
19.    if (ret != 0)
20.      fprintf (stderr, "join fallito %d\n“, ret) ;
21.    else printf("terminato il thread 2\n);
22.    return 0;
23.  }
```