

Università di Roma Tor Vergata  
Corso di Laurea triennale in Informatica  
**Sistemi operativi e reti**  
A.A. 2017-18

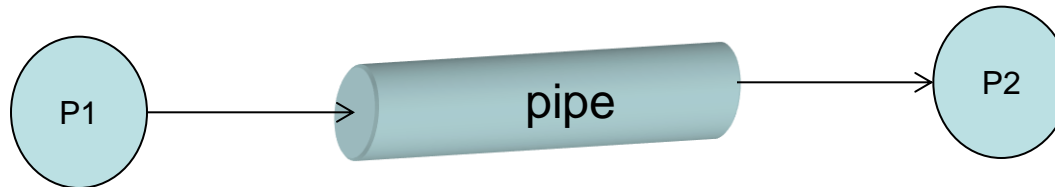
Pietro Frasca

## Lezione 9

Giovedì 2-11-2017

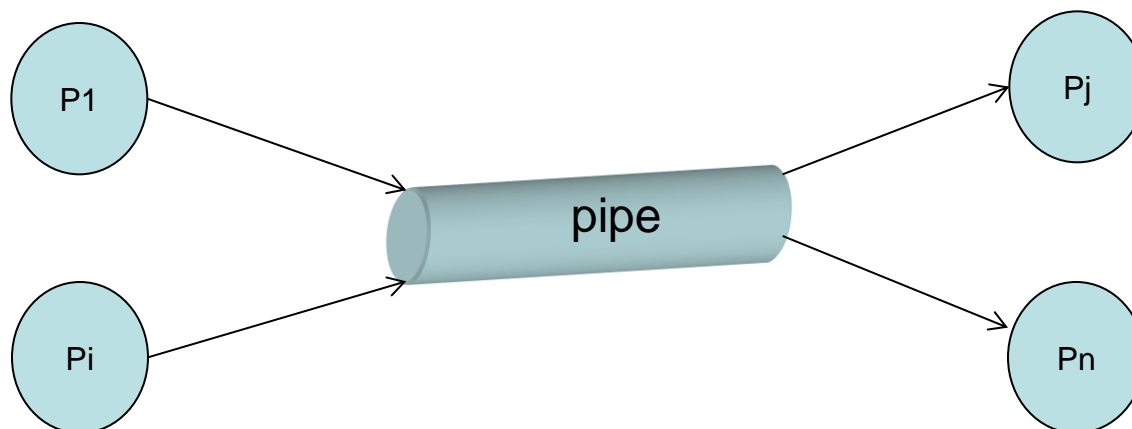
# Comunicazione con pipe

- Oltre che con la memoria condivisa e lo scambio di messaggi, i processi possono comunicare mediante un'altra tecnica di comunicazione che si basa sulle **pipe (tubo)**.
- Una **pipe** è un'astrazione di un canale per consentire la comunicazione tra processi.



- Due tipi comuni di pipe utilizzate su vari sistemi operativi, compresi Linux e Windows sono le **pipe senza nome (unnamed pipe)** e le **pipe con nome (named pipe)**.
- 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 dimensione di una pipe è limitata ed è stabilita da una costante di sistema (ad esempio in Linux `BUFSIZ` generalmente è di 4KB).
- 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.

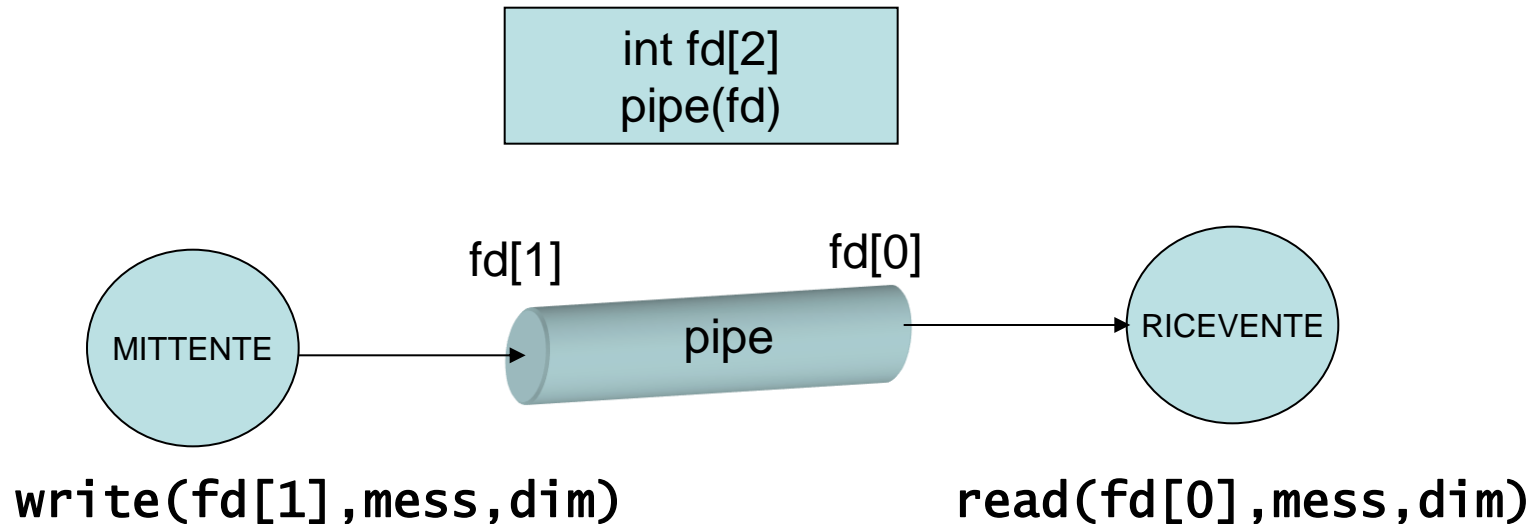


- Le pipe sono gestite allo stesso modo dei file. In particolare, ogni lato di accesso alla pipe è rappresentato da *un file descriptor*.
- I processi utilizzano le funzioni di lettura e scrittura di file **read** e **write** rispettivamente per ricevere o inviare messaggi dalla o alla pipe.

- In POSIX, 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 stessa funzione 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.

## 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, fd[2];
    int pid;
    char messaggio[DIM];

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

```

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

```



# Comunicazione con 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 sono cancellate dal sistema operativo.
- Le ***named pipe*** forniscono uno strumento di comunicazione più potente. La comunicazione può avvenire anche se i processi non hanno alcuna relazione parentale. 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 like*. Una volta create, possono essere visualizzate come tipici file nel file system.

- Un file FIFO si crea con la chiamata di sistema `mknod` (o con `mkfifo`) ed è gestito con le funzioni ordinarie `open`, `read`, `write`, e `close`.
- Un file FIFO continuerà ad esistere finché non è esplicitamente eliminato dal file system. Come la pipe senza nome, anche un file FIFO consente la comunicazione half-duplex. Se i processi devono scambiarsi dati in entrambe le direzioni, si utilizzano due FIFO. Inoltre, i processi in comunicazione devono risiedere sulla stessa macchina.

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
struct message_t {
    int id;
    char text[64];
} msg;
int main(int argc, char *argv[]) {
    int fd, i=0, n=10;
    int size=sizeof(msg);
    mknod("mia_pipe", S_IFIFO|0666, 0); // crea file FIFO
    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);  
    }  
}
```

# Thread

- La comunicazione tra processi può richiedere un alto tempo di esecuzione (*overhead*) dato che è necessario ricorrere a chiamate di sistema del kernel, come descritto.
- Anche la creazione e la terminazione di un processo risultano costose in termini di overhead.
- La *separazione* degli spazi di indirizzamento dei processi, consente da una parte la protezione dei dati dei singoli processi, ma dall'altra rende complesso l'accesso a strutture dati comuni, che si realizza attraverso **memoria comune** o mediante **scambio di messaggi**.
- Molte componenti di un sistema operativo e molte applicazioni possono essere sviluppate in moduli che girano in parallelo. Generalmente, ciascun modulo condivide dati e risorse comuni. Un esempio è dato da applicazioni in tempo reale per il controllo di impianti fisici in cui si possono individuare attività come il controllo di dispositivi di I/O dedicati a prelevare dati dall'ambiente fisico o a inviare comandi verso di esso. Ogni attività è implementata da un modulo che deve poter accedere a strutture dati comuni che rappresentano lo stato complessivo del sistema da controllare.

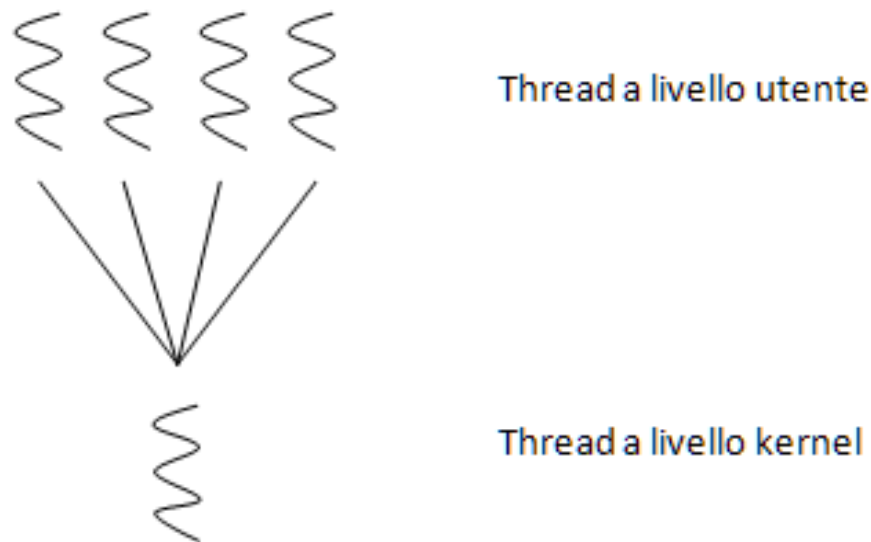
- Un altro esempio di applicazione che prevede l'esecuzione in parallelo di varie attività è l'editor di testi.
- Per ottenere una soluzione efficiente per le applicazioni che presentano un tipico grado di parallelismo, sono stati introdotti, nei sistemi operativi, i **thread**.
- Un thread è un **flusso di esecuzione** all'interno di uno stesso processo.
- All'interno di un processo è possibile definire più thread, ognuno dei quali condivide le risorse del processo, appartiene allo stesso spazio di indirizzamento e accede agli stessi dati, definiti con visibilità globale.
- Non possedendo risorse i thread possono essere creati e distrutti più facilmente rispetto ai processi; il cambio di contesto è più efficiente.
- Il termine **multithreading** significa che un processo possiede più thread.

- Ad ogni thread sono associati un descrittore, uno stato: esecuzione, pronto e bloccato, uno spazio di memoria per le variabili locali, uno stack, un contesto rappresentato dai valori di registri del processore utilizzati dal thread.
- Ad un thread non appartengono le risorse, che invece appartengono al processo che lo contiene.
- Le informazioni relative ai thread sono ridotte rispetto a quelle dei processi, quindi le operazioni di cambio di contesto, di creazione e terminazione sono molto semplificate rispetto a quelle dei processi.
- La gestione dei thread può avvenire sia a **livello utente** che a **livello kernel**.

# Thread a livello utente

## Modello da molti a uno

- Nel caso di thread a livello utente si utilizzano librerie realizzate a livello utente che forniscono tutto il supporto per la gestione dei thread: creazione, terminazione, sincronizzazione nell'accesso di variabili globali del processo, per lo scheduling, etc.; tutte queste funzioni sono realizzate nello spazio utente, il SO non vede l'esistenza dei thread e considera solo il processo che li contiene.





- I thread possono chiamare le system call, ad esempio per operazione di I/O; in questo caso interviene il SO che blocca il processo e di conseguenza tutti i thread in esso contenuti.
- I thread a livello utente hanno il vantaggio che possono essere utilizzati anche in SO che non supportano direttamente i thread; è possibile usare politiche di scheduling indipendenti dal kernel del SO.
- Tuttavia, non è possibile, con i thread a livello utente, sfruttare il parallelismo delle architetture multiprocessore, dato che quando un processo è assegnato ad uno dei processori, tutti i suoi thread sono eseguiti, uno alla volta, su quel solo processore.