Comunicazione Inter-Processo

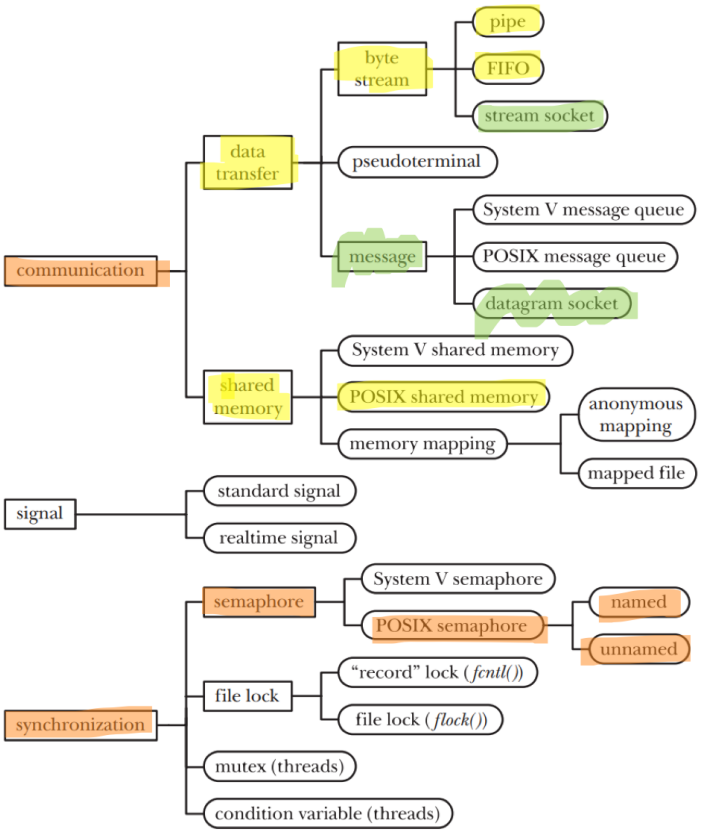
Nei thread le variabili globali possono essere utilizzate per la comunicazione inter-thread (con giusti meccanismi di sincronizzazione)

Invece nei processi, dato che ogni processo ha un proprio spazio di indirizzi di memoria (e può accedere solo a quelli), non si possono usare variabili globali ma bisogna sfruttare il kernel del sistema operativo (che può accedere a tutti gli indirizzi)

I processi in un sistema possono essere:

- indipendenti → non sono influenzati tra di loro
- cooperanti → sono influenzati tra di loro; devo implementare un meccanismo di comunicazione

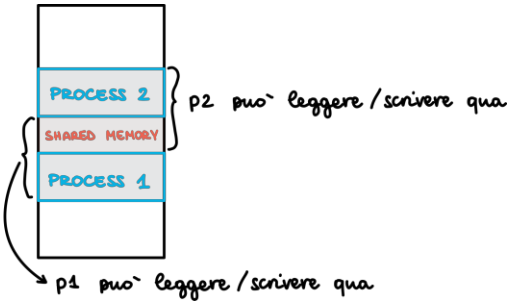
➔ la comunicazione viene usata per scambiare informazioni, per modularità (fare più task insieme per applicazioni complesse), per velocità di computazione



2 modelli di IPC:

- **shared memory** → c'è una memoria condivisa per scambiare dati tra processi (N.B. no sincronizzazione "build-in" ossia bisogna aggiungere meccanismi di sincronizzazione)
- **message passing** → i processi si mandano messaggi tramite il kernel

Shared Memory



I processi possono leggere/scrivere liberamente sulla memoria condivisa tramite puntatori

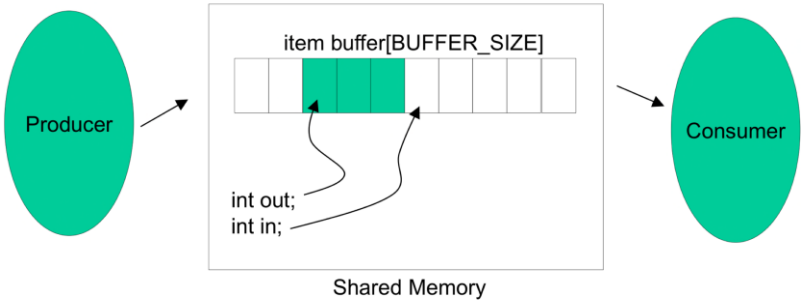
Il kernel è interpellato solo per l'allocazione della memoria condivisa

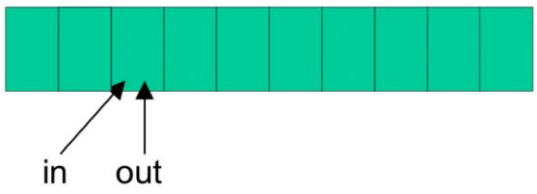
➔ la memoria condivisa permette ai processi di essere indipendenti dal kernel solo formalmente perché dovrà essere utilizzato il kernel per gestire la sincronizzazione

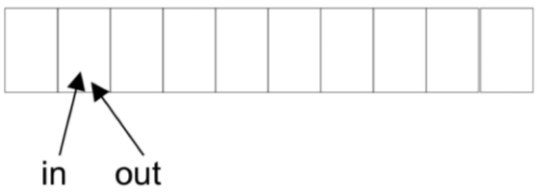
I processi comunicheranno tra loro tramite il modello produttore/consumatore

→ c'è un buffer che appartiene all'area di memoria condivisa in cui il produttore produce e il consumatore consuma; si accede al buffer con i 2 puntatori *int out* e *int in* che indicano l'inizio e la fine del buffer

→ per accedere al buffer si usano i semafori *sem_empty* e *sem_filled* che indicano rispettivamente il numero di posizioni libere e il numero di posizioni occupate







In caso di buffer pieno:

in==out

sem_empty.val==0

In caso di buffer vuoto:

in==out

sem_empty.val==BUFFER_SIZE

```
while (true) {
        /* Produce an item */
        sem_wait(sem_empty); %named semaphores
        sem_wait(sem_cs);
        writeToBuffer (ITEM);
        in = (in+1) % BUFFER_size;
        sem_signal(sem_cs);
        sem_signal(sem_filled);
}
```

Producer

Consumer

Buffer (an array)
in, out integer variable

Shared Memory

```
while (true) {
        sem_wait(sem_filled);
        sem_wait(sem_cs);
        readFromBuffer (ITEM);
        out = (out +1) % BUFFER_size;
        sem_signal(sem_cs);
        sem_signal(sem_empty);
        /* Consume the item */
    }
```

Posix API Shared Memory

shm_open( ) → crea e lo apre una pagina di memoria condivisa; i figli possono accedere alla pagina

ltrunc( ) o ftruncate( ) → limita la dimensione della pagina di memoria condivisa

mmap( ) → restituisce il puntatore all'area di memoria

close( ) → chiude il descriptor dell'area di memoria

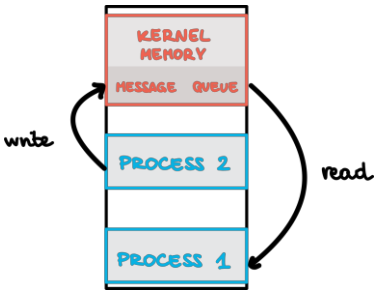
shm_unlink( ) → rimuove la pagina di memoria condivisa

es. utilizzo di una pagina di memoria

```
//scrittura
int main() {
        const int SIZE = 4096;

        const char * name = "MY_PAGE";
        const char * msg = "Hello World!";
        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;
}
```

```
//lettura
int main() {
        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);
        shm_unlink(shm_fd);
        return 0;
}
```

Message Passing

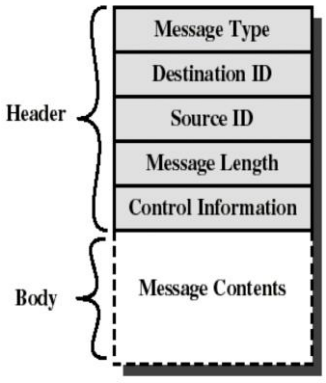


Sistema di comunicazione che usa 2 primitive [send, receive]

Dati 2 processi, essi utilizzano una message queue che appartiene alla memoria del kernel; c'è un overhead (perché ogni volta viene invocato il sistema operativo) ma la sincronizzazione è "build-in"; la politica della message queue è FIFO

Sia il mittente che il ricevente dovranno utilizzare un modulo dedicato al message passing

Il messaggio dovrà rispettare uno specifico formato (creato direttamente dal modulo)



➔ utilizzo **pipes** e **named-pipes (FIFOs)** [tubi]

Pipe e FIFO

Pipe → abilita un canale di comunicazione a senso unico tra processo padre e processo figlio

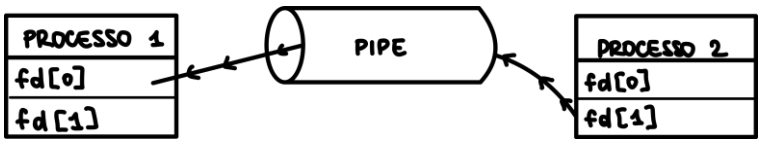
- quando tutti i processi che utilizzano la pipe terminano, la pipe è rimossa automaticamente
- la pipe viene istanziata con la system call *pipe()*
- le pipe permettono a più processi di comunicare come se stessero accedendo a dei file sequenziali; una volta lette le informazioni, esse spariscono dalla pipe e non possono più ripresentarsi
- i processi che usano le pipe devono essere relazionati (processi padre-figlio) come conseguenza di operazioni di *fork()*
- a livello di sistema operativo, le pipe non sono altro che buffer di dimensione più o meno grande (solitamente 4096 byte)

Named Pipe (FIFO) → come nei semafori named, qualunque processo conosca il nome della FIFO può utilizzarla; non necessitano di una relazione padre-figlio e la comunicazione è bidirezionale

➔ in sistemi UNIX l'uso di pipe avviene attraverso dei descrittori

ogni processo ha una coppia di descrittori

- 1° descrittore → lettura
- 2° descrittore → scrittura



Al momento dell'apertura si può definire chi legge e chi scrive attraverso i permessi

Le Pipe possono essere utilizzate in:

- full duplex (entrambi scrivono e ricevono) → può generare errori; tipicamente è meglio inizializzare 2 pipe unidirezionali
- half duplex (solo un processo scrive e solo un processo riceve)

Posix Pipe

int pipe (int fd[2]) → crea una pipe dove fd è puntatore ad un buffer di due interi

- fd[0] : descrittore di lettura dalla PIPE
- fd[1]: descrittore di scrittura sulla PIPE

➔ fd[0] è un canale aperto in lettura che consente ad un processo di leggere dati da una PIPE

➔ fd[1] è un canale aperto in scrittura che consente ad un processo di immettere dati sulla PIPE

➔ fd[0] e fd[1] possono essere usati come normali descrittori di file tramite le chiamate *read()* e *write()*

Un processo che provi a leggere da una pipe vuota rimane bloccato finché non ci sono dati disponibili

Un processo che provi a scrivere su una pipe piena rimane bloccato finché un altro processo non ha letto (e rimosso) abbastanza dati da permettere la scrittura → la *write()* di per sé non è bloccante; lo diventa solo se la pipe è piena

Una pipe ha una dimensione massima equivalente a 16 pagine di memoria in Linux [65,536 byte in un sistema con una page size di 4096 byte (standard POSIX)]

→ è possibile rendere la lettura e la scrittura non bloccanti (non in questo corso)

→ è possibile cambiare la dimensione della pipe

La scrittura di n byte con n ≤ dimensione pagina è atomica (non viene interrotta ma potrebbe fallire); se n > dimensione pagina può essere inframezzata con altre *write()*

Nel caso di write non bloccanti:

- Se n ≤ PIPE BUF ma non ci sono n byte disponibili write() fallisce e setta errno
- Se n > PIPE BUF potrebbe risultare in una scrittura parziale (ma non in un errore)

➔ dato che le pipe non sono dispositivi fisici ma logici, come dovrebbe fare un processo per "vedere" la fine di un file su una pipe? controlla se tutti i processi scrittori che condividevano il descrittore fd [1] lo hanno chiuso

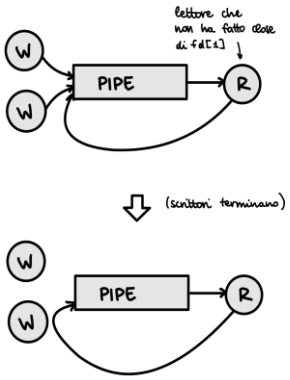
N.B. quando un processo fa la system call *pipe( )* riceve una copia del descrittore fd[0](lettura) e una copia del descrittore fd[1] (scrittura)

→ se un processo prova a chiamare *read()* su una pipe che non ha più scrittori, riceve 0 (notifica dell'evento che gli scrittori hanno finito)

→ se invece prova a scrivere su una pipa senza lettori, riceve il segnale SIGPIPE Io "broken pipe")

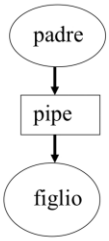
Quando un processo chiama la funzione *pipe()*, viene creata una pipe fornendo due descrittori ⇨ il processo è sia lettore che scrittore; se fa una *read()* troverà uno scrittore che è lui stesso, se fa una *write()* la troverà un lettore che è lui stesso e questo può portare a deadlock

es. un processo lettore R eredita tutti e due i descrittori ma non chiude il descrittore in scrittura ⇨ quando i processi scrittori terminano, R facendo la *read()* non avrà mai output uguale a zero perché R stesso è l'unico processo in scrittura e non potra mai scrivere perché è bloccato sulla *read()*



➔ per evitare deadlock, tutti i processi devono chiudere i descrittori non utilizzati con una *close()*

```c
#include <stdio.h>
#define Errore_(x) { puts(x); exit(1); }

int main(int argc, char *argv[]) {
    char messaggio[30];   int  pid, status, fd[2];

    ret = pipe(fd); /* crea una PIPE */
    if ( ret == -1 )
        Errore_("Errore nella creazione pipe");

    pid = fork(); /* crea un processo figlio */
    if ( pid == -1 ) Errore_("Errore nella fork");

    if ( pid == 0 ) { /* processo figlio: lettore */
        close(fd[1]); /* il lettore chiude fd[1] */
        while( read(fd[0], messaggio, 30) > 0 )
            printf("letto messaggio: %s", messaggio);
        close(fd[0]);
        _exit();
    }
```




```c
    /* processo padre: scrittore */
    else {

        close(fd[0]);
        puts("digitare testo da trasferire (quit per
terminare):");

        do {
            fgets(messaggio,30,stdin);
            write(fd[1], messaggio, 30);
            printf("scritto messaggio: %s", messaggio);
        } while( strcmp(messaggio,"quit\n") != 0 );

        close(fd[1]);
        wait(&status);
    }

}
```

Posix FIFO

int mkfifo (char* name, int mode) → invoca la creazione di una FIFO

- name: nome FIFO da creare
- mode: specifica i permessi di accesso alla FIFO

➔ ritorna 0 in caso di successo, -1 altrimenti

int unlink (char* name) → rimuove la FIFO

Normalmente l'apertura di una fifo è bloccante ⇨ il processo che tenta di aprire la FIFO in lettura (scrittura) viene bloccato fino a quando un'altro processo non la apre in scrittura (lettura)

➔ se si vuole inibire questo comportamento è possibile aggiungere il flag O_NONBLOCK al valore del parametro mode passato alla system call *open()* sulla FIFO

ogni FIFO deve avere sia un lettore che uno scrittore; se un processo tenta di scrivere su una FIFO che non ha un lettore esso riceve il segnale “SIGPIPE” da parte del sistema operativo

L’apertura di una pipe comporta ottenere sia il descrittore per la lettura che quello per la scrittura (bisogna chiudere quello che non viene usato)

L’apertura di una FIFO avviene in lettura o in scrittura (non serve chiudere descrittori)

→ nella gestione le pipe e le FIFO si differenziano solo nella fase di apertura e chiusura; lettura e scrittura avvengono nello stesso modo

es. client-server con FIFO

Server

```c
#include <stdio.h>
#include <fcntl.h>

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

int main(int argc, char *argv[]){
    char *response = "fatto";
    int pid, fd, fdc, ret;
    request r;

    ret = mkfifo("/serv", 0666);
    if ( ret == -1 ) {
        printf("Errore nella chiamata mkfifo\n");
        exit(1);
    }

    fd = open("/serv",O_RDONLY);
    while(1) {
        ret = read(fd, &r, sizeof(request));
        if (ret != 0) {
            printf("Richiesto un servizio (fifo di restituzione = %s)\n", r.fifo_response);
            sleep(10); /* emulazione di ritardo per il servizio */
            fdc = open(r.fifo_response,O_WRONLY);
            write(fdc, response, 20);
            close(fdc);
            exit(0);
        }
    }
}
```

Client

```c
#include <stdio.h>
#include <fcntl.h>

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

int main(int argc, char *argv[]) {
    int pid, fd, fdc, ret; request r; char response[20];

    printf("Selezionare un carattere alfabetico minuscolo: ");
    scanf("%s",r.fifo_response);

    if (r.fifo_response[0] > 'z' || r.fifo_response[0] < 'a' ) {
        printf("carattere selezionato non valido, ricominciare operazione\n");
        exit(1);
    }
    r.fifo_response[1] = '\0';
    ret = mkfifo(r.fifo_response, 0666);
    if (ret == -1) {
        printf("\n servente sovraccarico - riprovare \n");
        exit(1);
    }

    fd = open("/serv",O_WRONLY);
    if ( fd == -1 ) {
        printf("\n servizio non disponibile \n");
        ret = unlink(r.fifo_response);
        exit(1);
    }

    write(fd, &r, sizeof(request));
    close(fd);

    fdc = open(r.fifo_response,O_RDONLY);
    read(fdc, response, 20);
    printf("risposta = %s\n", response);
```

```
    close(fdc);
    unlink(r.fifo_response);
}
```