POLITECNICO DI MILANO





Inter Process Communication

Laboratorio Software 2008-2009
C. Brandolese

Introduzione

Più processi o thread

- Concorrono alla relaizzazione di una funzione applicativa
- Devono poter realizzare
  - Sincronizzazione
  - Comunicazione

Memodia condivisa o shared memory

- Adatta a piccoli sistemi operativi e supporti runtime
- Adatta in caso di multi-threading

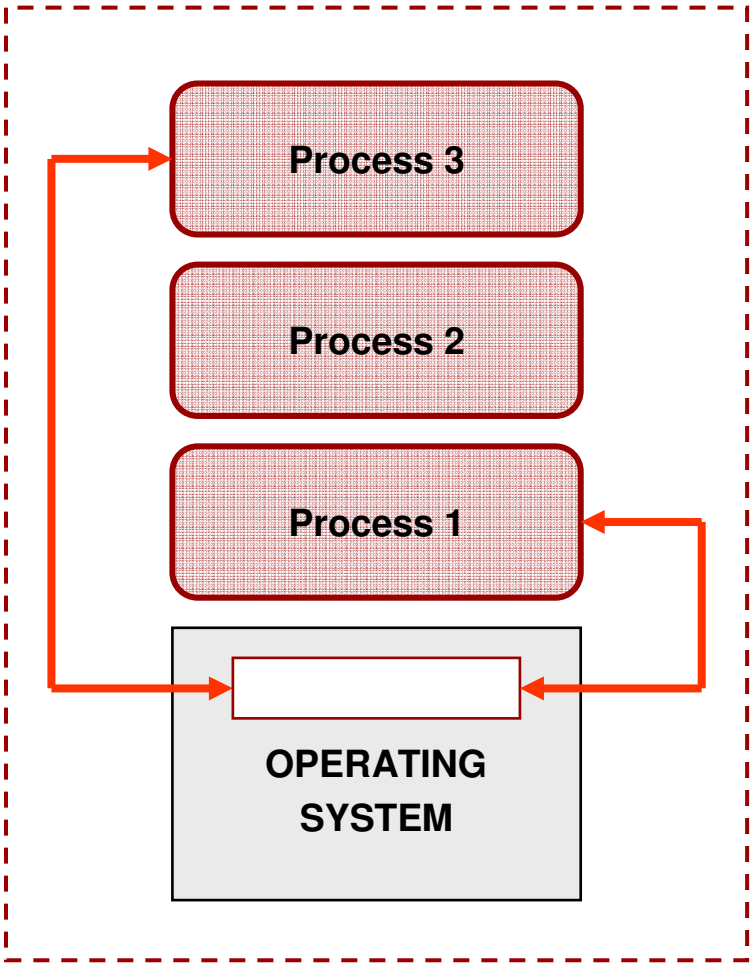
IPC o Inter Process Communication

- Processi in che risiedono in spazi di memoria separati
- Applicabile anche in ambienti
  - Distribuiti
  - Multiprocessore
- Flessibile rispetto alla ‘posizione dei processi’

Introduzione

Diversi tipi di IPC

- Messaggi
- Segnali
- Pipe
- Named pipe o FIFO
- Socket
- Remote Procedure Call o RPC



Messaggi

Si tratta di un metodo molto generale di IPC che fornisce

- Sincronizzazione
- Comunicazione
- Mutua esclusione

Si applica bene a

- Processi in esecuzione su una singola macchina
- Processi in un sistema distribuito

Richiede due primitive di base

- `send( destination, message )`
- `received ( source, message )`

Entrambe possono essere

- Bloccanti
- Non bloccanti

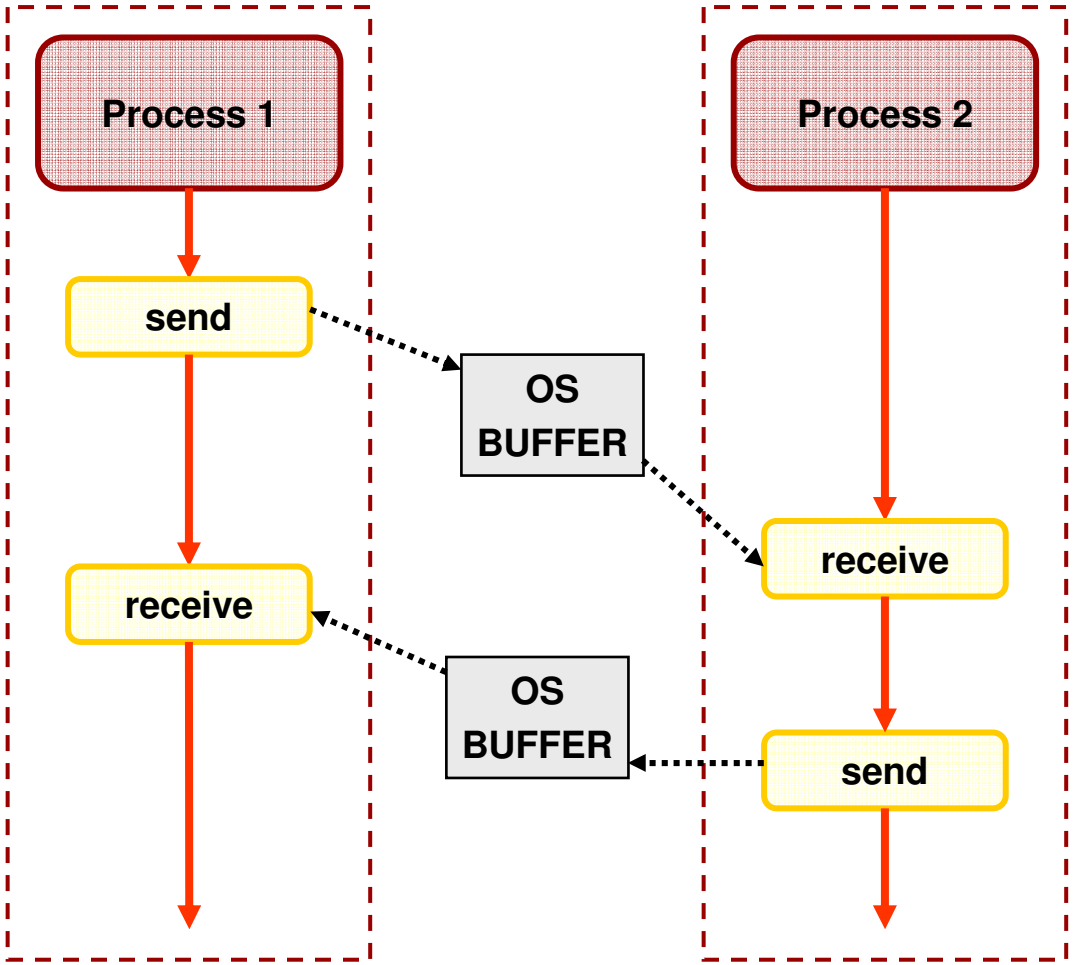
Messaggi – Schema non bloccante

Richiede

- Buffer di sistema
- Allocazione di memoria
- Gestione di risorse

Fornisce

- Sincronizzazione minima a livello applicativo



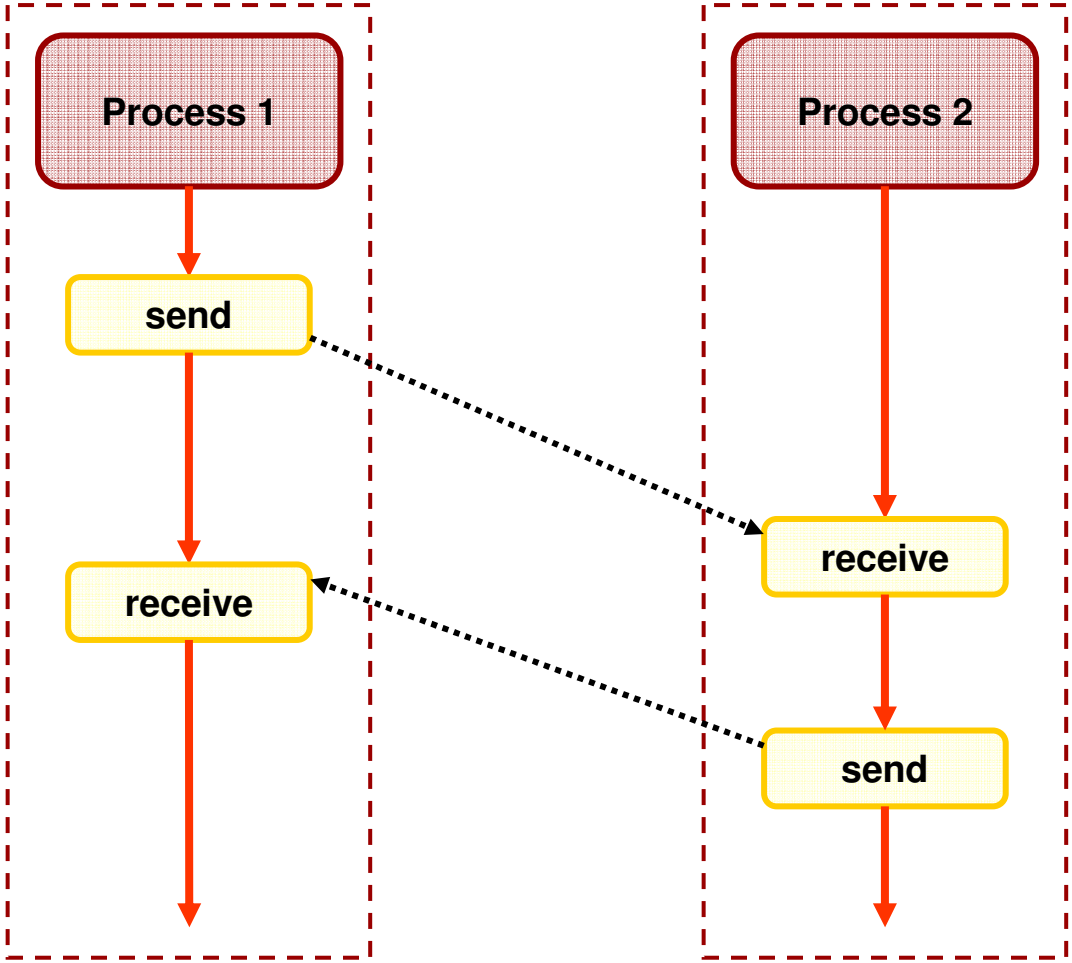
Messaggi – Schema bloccante

Richiede

- Nessun buffer di sistema
- Nessuna allocazione di memoria
- Nessuna risorsa aggiuntiva

Fornisce

- Sincronizzazione stretta a livello applicativo



Messaggi – Schema bloccante

Ad ogni stante, rispetto alla comunicazione, si distinguano

- Un sender
- Un receiver

Sender

- È più naturale che non si blocchi dopo una send()
  - Può mandare più messaggi in sequenza a destinatari diversi
  - Normalmente si attende un messaggio di conferma

Receiver

- Si sospende dopo una receive() in attesa dell'arrivo di un messaggio
  - Generalmente necessita del contenuto del messaggio per procedere
  - Potrebbe rimanere bloccato indefinitamente se la send() fallisce

Messaggi – Schema bloccante

Caso 1: Sincronizzazione

- send() non bloccante
- receive() bloccante

Caso 2: Sincronizzazione stretta o rendez-vous

- send() bloccante
- receive() bloccante

Messaggi – Indirizzamento

Un messaggio deve specificare

- Un indirizzo del sender
- Un indirizzo del receiver
- Dati

Si hanno due possibilità per gestire l'indirizzamento

- Indirizzamento dieretto
  - Indirizzi del sender e del receiver strettamente associati ai processi
  - Per sempio, PID
  - Problema: potrebbero non essere noti entrambi a priori
- Indirizzamento indiretto
  - I messaggi vengono inviati ad una mailbox condivisa
  - La mailbox impementa una coda di messaggi
  - I processi interessati prelevano i messagi a loro destinati

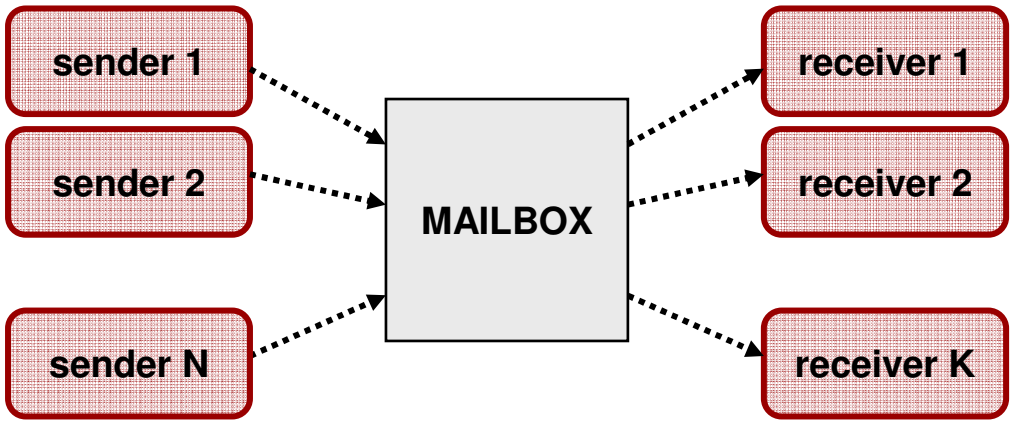
Messaggi – Mailbox

Può essere

- Privata ad una coppia sender/receiver
- Condivisa
  - In genere ha più sender e più receiver

Gestione

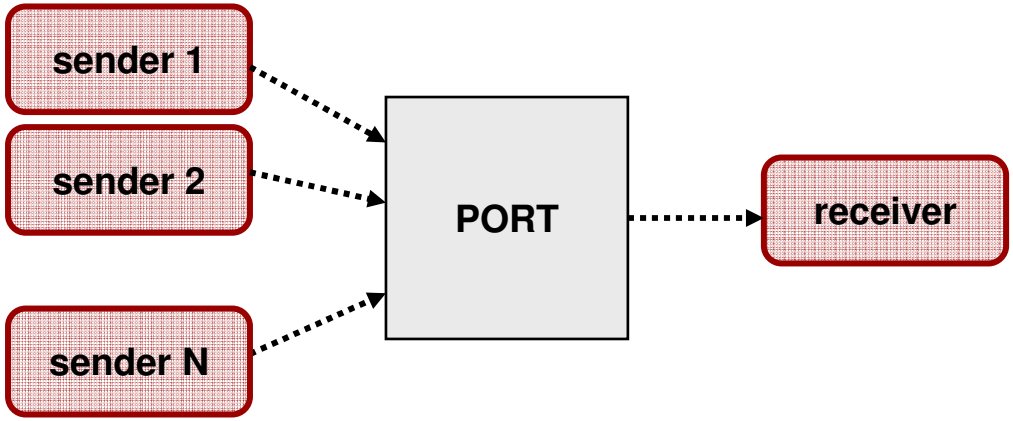
- Creata dal SO su richiesta di un processo
  - Il processo ne diviene il proprietario
- Distrutta
  - Su richiesta del proprietario
  - Quando il proprietario termina



Messaggi – Port

Port

- È una mailbox limitata
- Realizza il modello client/server



Può avere

- Più sender
- Un solo receiver

Gestione

- Creata dal receiver
  - Ne diviene il proprietario
- Distrutta
  - Quando il proprietario termina

Messaggi – Struttura

Si divide in

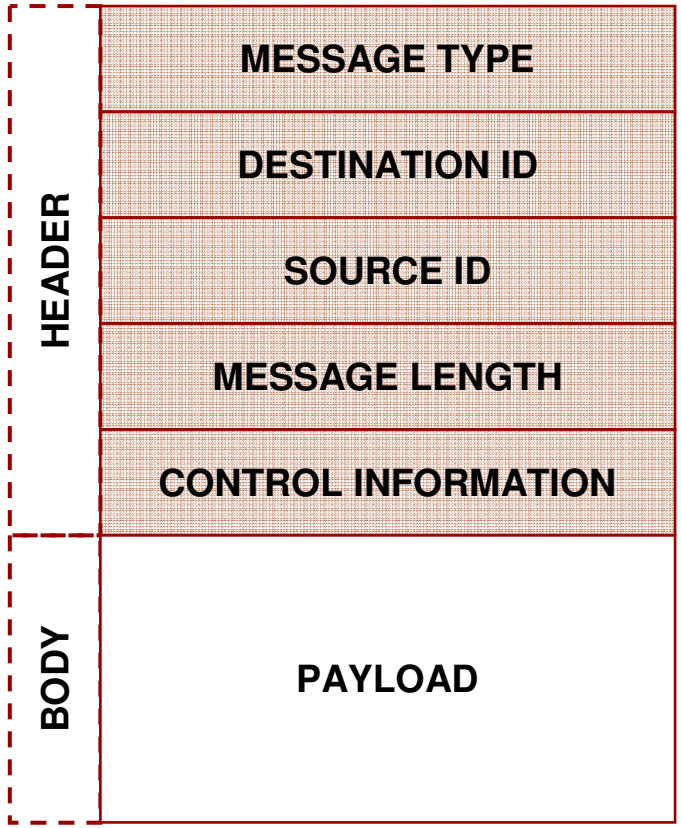
- Header, intestazione
- Body, corpo

Contiene

- Source/destination ID
  - Es. In UNIX, nessun ID
- Informazioni di controllo
  - Cosa fare con buffer pieni?
  - Numeri di sequenza
  - Priorità
  - ...

Politica di accodamento

- Generalmente FIFO
- Può supportare anche priorità



Messaggi e mutua esclusione

Si crea una mailbox: mutex

- Condivisa da più processi

Chiamate

- `send()` non bloccante
- `receive()` bloccante
  - Si sospende se mutex è vuota

Inizializzazione

- `send(mutex,"go");`

Ciclo

- Il primo processo che esegue la receive
  - Entra nella critical section
  - Svuota la mailbox
  - Al termine, rimette un messaggio nella mailbox
- Gli altri processi
  - Attendono che un messaggio sia disponibile

```
msg: message;
for(;;)
{
  receive( mutex, msg );
  /* CS */
  send( mutex, msg );
  /* RS */
}
```

Segnali

Un seganle

- È un messaggio di un singolo bit
- È un modo per notifiacre un evento
  - In modo asincrono

Alcuni tipici eventi

- Timer
- Completamento di una operazione di I/O
- Eccezioni di programma
- Altri eventi definiti dall'utente

Non si ha sincronizzazione

- Sender asincrono
- Receiver asncrono

Segnali – UNIX

Ogni segnale è indicato da un valore nuemrico intero

- 2 **`SIGINT`** Interrompe un processo
- 9 **`SIGKILL`** Termina un processo
- 3 **`SIGQUIT`** Abort
- 14 **`SIGALRM`** Interrupt del timer
- 18 **`SIGCLD`** Terminazione di un processo figlio

Ogni segnale

- Memorizzato come un singolo bit nel descrittore del proceso ricevente
- Il bit viene posto ad 1 quando il segnale arriva
  - Nessuna coda
- Il segnale viene elaborato non appena il processo diviene running
  - Normalmente si procede con l'azione di default, cioè la terminazione

Pipe

Realizzate come

- Una coda condivisa e limitata
  - Scritta da un processo, letta da un altro
- Si bassa sul modello produttore/consumatore

Mutua esclusione

- Garantita dal sistema operativo
  - Solo un processo alla volta può accedervi

Sincronizzazione

- Se la coda è piena si blocca il sender
- Se la coda è vuota si blocca il receiver
  - Anche se si cerca di leggere più dati di quanti disponibili

Mette in comunicazione solo pardri/figli

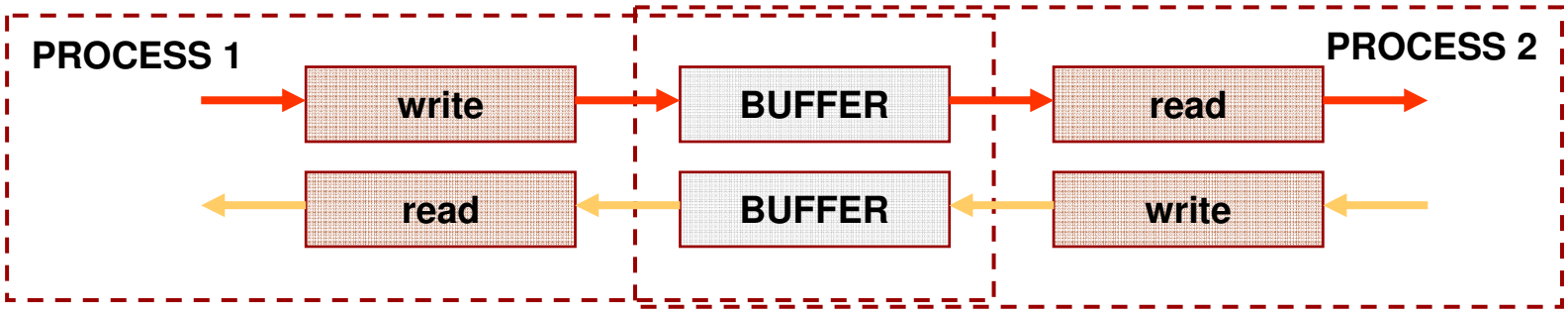
Pipe – UNIX

La creazione di una pipe per la comunicazione

- Avviene nel processo padre
- I descrittori di file relativi alla pipe vengono copiati nel figlio

Un buffer di sistema operativo implementa la pipe

- Si tratta di un canale di comunicazione bidirezionale
  - Implementato come due canali monodirezionali
- Accessibile mediante le stesse funzioni di I/O usate per i file
  - Descritto da un i-node
  - Si usano solo i blocchi diretti gestiti come una coda circolare

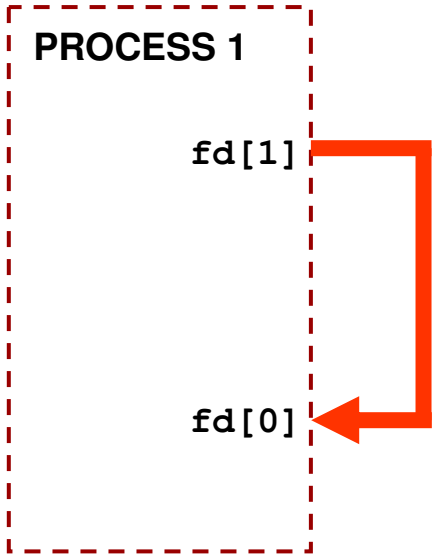


Pipe – UNIX

Esempio di implementazione: UNIX

- Creazione della pipe: `pipe( fd[2] )`
- Lettura/scrittura: `read(), write()`

```
main()
{
   int   fd[2];
   char buf[MAX_DIM]
   pipe( fd );
   write( fd[1], "Hello!", 6 );
   read( fd[0], buf, 6 );
   /* buf = "Hello!" */
}
```



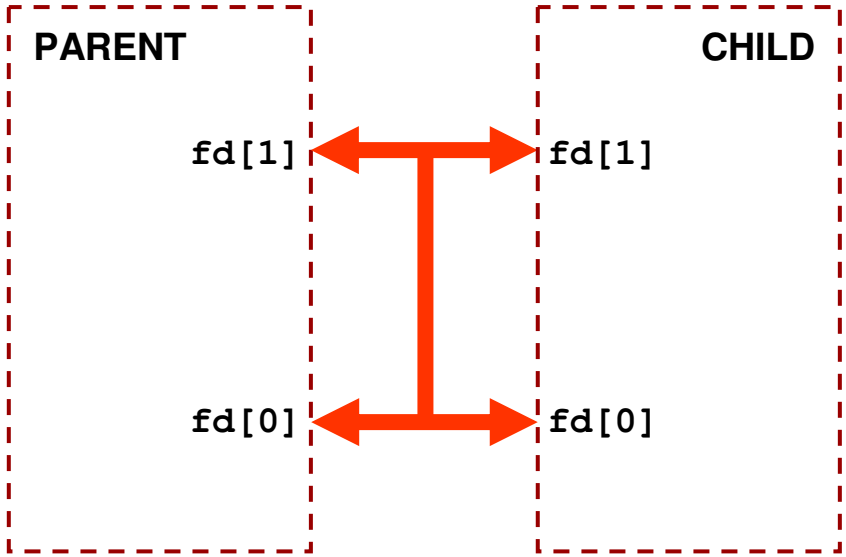
Pipe – UNIX

Comunicazione tra processi

- Il padre crea la pipe
- Il figlio la eredita

```
main()
{
  int  fd[2], status;
  pipe( fd );
  if( fork() == 0 ) {
    /* Child  */
  }
  /* Parent */;
  ...

  close( fd[0] );
  close( fd[1] );
  wait( &status );
}
```



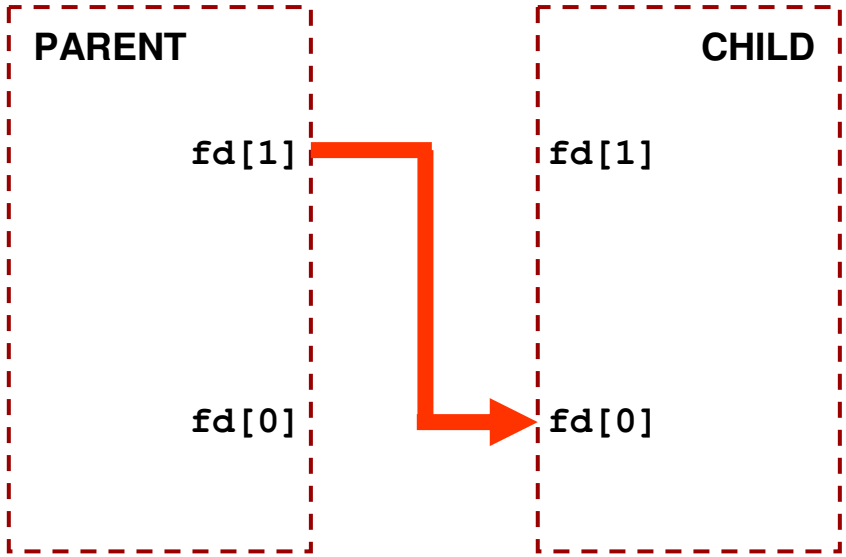
Pipe – UNIX

Comunicazione tra processi

- I processi padre e figlio dell'esempio precedente condividono le pipe
- Le pipe si riferiscono alle stesse entità (buffer) di I/O

```
main()
{
  int  fd[2], status;
  pipe( fd );
  if( fork() == 0 ) {
    /* Child  */
    close( fd[1] );
  }
  /* Parent */;
  close( fd[0] );
  ...

  wait( &status );
}
```



Named Pipe o FIFO

Simile ad una pipe

- Stesso meccanismo bassato su una coda condivisa

Una volta creata

- Esiste nel file system
- Vi si accede come ad un normale file

Sincronizzazione

- L'apertura in lettura di una FIFO si sospende
  - Fino a quando qualcuno non la apre in scrittura
- Se la coda è piena si blocca il sender
- Se la coda è vuota si blocca il receiver
  - Anche se si cerca di leggere più dati di quanti disponibili

Mette in comunicazione processi generici

```
mkfifo fifo1
prog3 < fifo1 &
prog1 < input_file | tee fifo1 | prog2
```

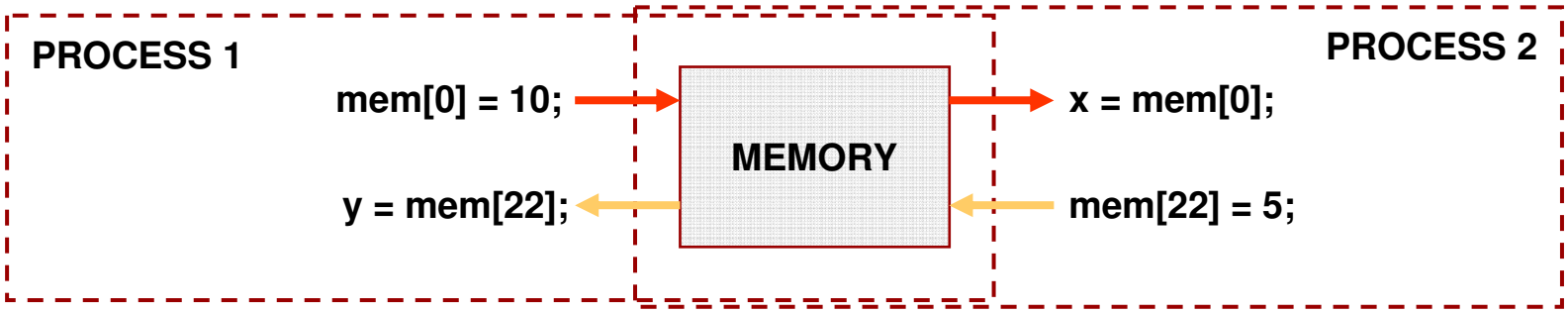
Shared memory

La memoria condivisa

- È uno dei meccanismi di comunicazione più efficienti
- Richiede il supporto del sistema operativo solo all'atto dell'inizializzazione
- Deve essere utilizzato con cautela poiché elimina (parzialmente) l'indipendenza e la separazione tra i processi

Per un corretto utilizzo

- Si deve garantire la mutua esclusione



Shared memory

Operazioni di base

Creazione: `shmget()`

- Crea una zona dimemoria condivisa
- Ritorna un identificatore numerico univoco

Controllo: `shmctl()`

- Controlla e configura una zona di memoria condivisa
- Legge informazioni, prepara per la rimozione, cambia i diritti, ...

Attach: `shmat()`

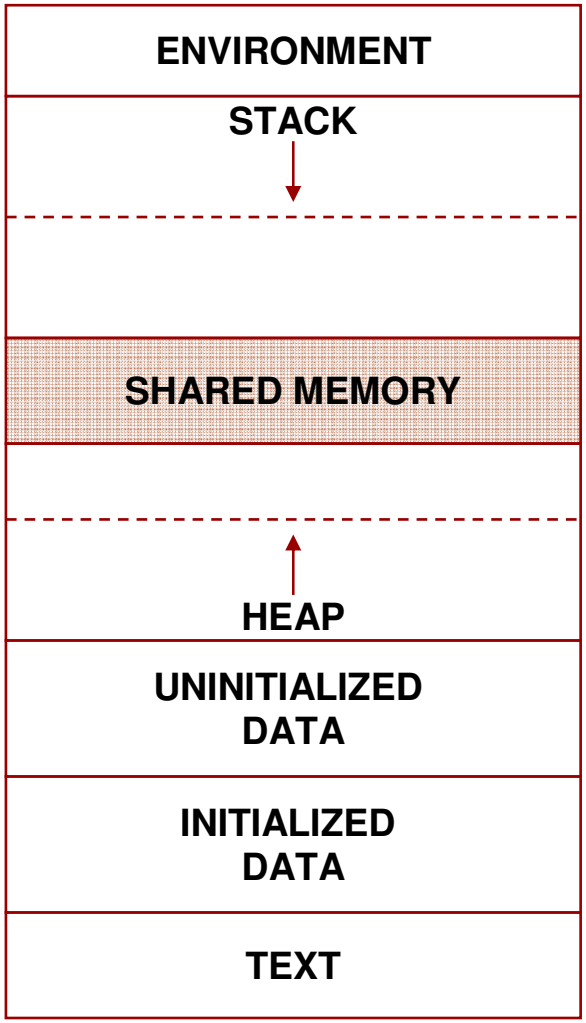
- Associa una zona di memoria condivisa al processo
- Ritorna un puntatore alla memoria

Detach: `shmdt()`

- Sgancia una zona di memoria condivisa
- Se non vi sono più processi associati, una zona può essere rimossa

Shared memory

Dopo la creazione di un segmento di memoria condivisa la struttura logica della memoria di un processo è mostrata a lato



Chiamata `fork()`

- La memoria è ereditata dal figlio

Chiamata `exec()`

- La memoria condivisa è rilasciata ma non distrutta

Chiamata `exit()`

- La memoria condivisa è rilasciata ma non distrutta