

Università di Ferrara
Laurea Triennale in Informatica
A.A. 2021-2022
Sistemi Operativi e Laboratorio

5. Interazione fra processi: meccanismi di sincronizzazione e comunicazione

Prof. Carlo Giannelli

`http://www.unife.it/scienze/informatica/insegnamenti/
sistemi-operativi-laboratorio`

`http://docente.unife.it/carlo.giannelli`

`https://ds.unife.it/people/carlo.giannelli/`

Processi interagenti

Classificazione

- *processi indipendenti*

due processi sono indipendenti se l'esecuzione di ognuno non è in alcun modo influenzata dall'altro

- *processi interagenti*

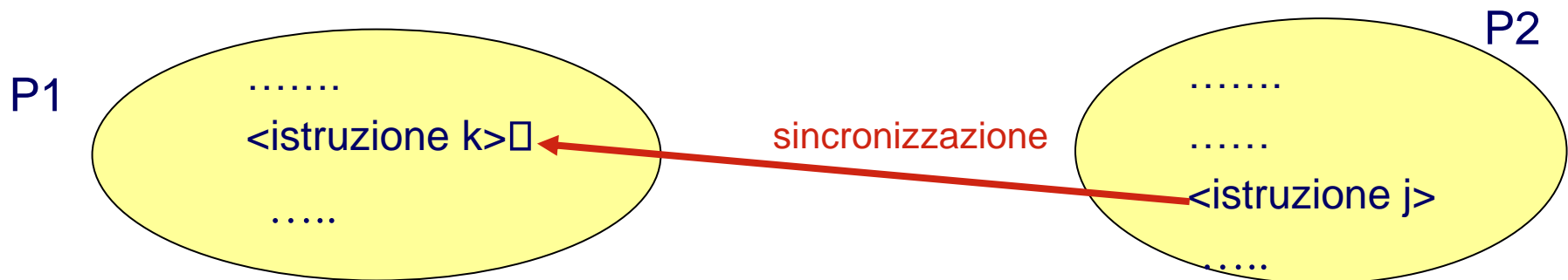
- **cooperanti**: i processi interagiscono *volontariamente* per raggiungere **obiettivi comuni** (fanno parte della stessa applicazione)
- **in competizione**: i processi, in generale, non fanno parte della stessa applicazione, ma *interagiscono indirettamente per l'acquisizione di risorse comuni*

Processi interagenti

L'interazione può avvenire mediante due meccanismi:

- **Comunicazione:** scambio di informazioni tra i processi interagenti
- **Sincronizzazione:** imposizione di *vincoli temporali, assoluti o relativi*, sull'esecuzione dei processi

Ad esempio, l'istruzione k del processo P1 può essere eseguita **soltanto dopo** l'istruzione j del processo P2



Processi interagenti

Realizzazione dell'interazione: dipende dal modello di esecuzione per i processi

- **modello ad ambiente locale**: non c'è condivisione di variabili (processo pesante)
 - **comunicazione** avviene attraverso **scambio di messaggi**
 - **sincronizzazione** avviene mediante **scambio di eventi (segnali)**
- **modello ad ambiente globale**: più processi possono condividere lo stesso spazio di indirizzamento → possibilità di condividere variabili (come nei *thread*)
 - **variabili condivise e relativi strumenti di sincronizzazione** (ad esempio, **lock** e **semafori**)

Processi interagenti mediante scambio di messaggi

Facciamo riferimento al **modello ad ambiente locale**:

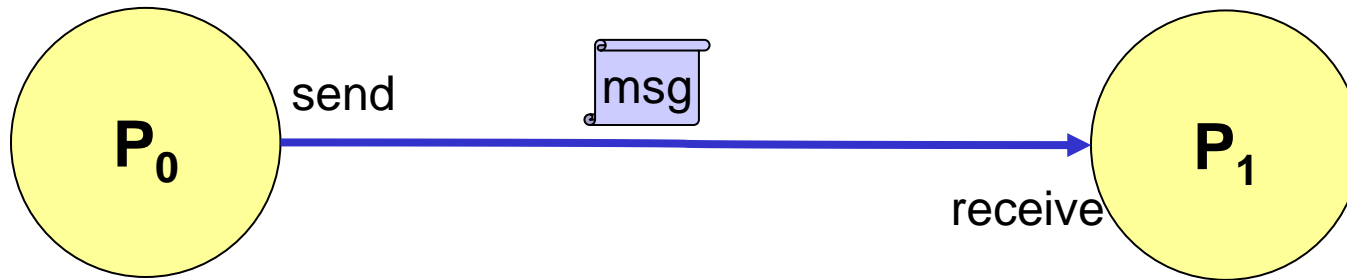
- ***non vi è memoria condivisa***
- i processi possono interagire mediante ***scambio di messaggi: comunicazione***

Spesso SO offre meccanismi a supporto della comunicazione tra processi (Inter Process Communication - IPC)

Operazioni Necessarie

- ***send***: spedizione di messaggi da un processo ad altri
- ***receive***: ricezione di messaggi

Scambio di messaggi



Lo scambio di messaggi avviene mediante un **canale di comunicazione** tra i due processi

Caratteristiche del canale:

- monodirezionale, bidirezionale
- uno-a-uno, uno-a-molti, molti-a-uno, molti-a-molti
- capacità
- modalità di creazione: automatica, non automatica

Naming

In che modo viene specificata la destinazione di un messaggio?

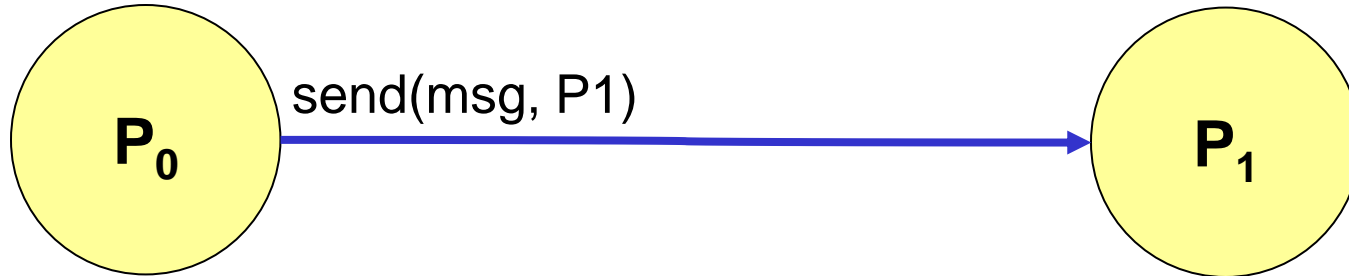
- **Comunicazione diretta** - al messaggio viene associato *l'identificatore del processo destinatario* (naming esplicito)

send(Proc, msg)

- **Comunicazione indiretta** - il messaggio viene indirizzato a una mailbox (contenitore di messaggi) dalla quale il destinatario preleverà il messaggio

send(Mailbox, msg)

Comunicazione diretta



Il canale è creato automaticamente tra i due processi che devono ***conoscersi reciprocamente***:

- canale punto-a-punto
- canale bidirezionale:
 - p_0 : `send(query, P1);` p_1 : `send(answ, P0);`
- per ogni coppia di processi esiste un solo canale ($\langle P_0, P_1 \rangle$)

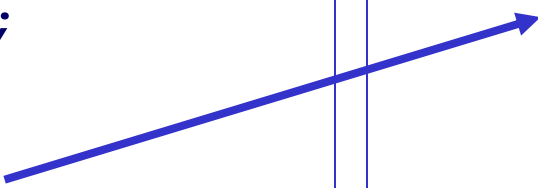
Esempio: produttore & consumatore

Processo produttore P:

```
pid C =....;
main() {
    msg M;
    do{
        produco (&M) ;
        ...
        send(C, M) ;
    }while(!fine) ;
}
```

Processo consumatore C:

```
pid P=....;
main() {
    msg M;
    do{
        receive(P, &M) ;
        ...
        consumo (M) ;
    }while(!fine) ;
}
```



Comunicazione simmetrica: destinatario deve fare *naming* esplicito del mittente

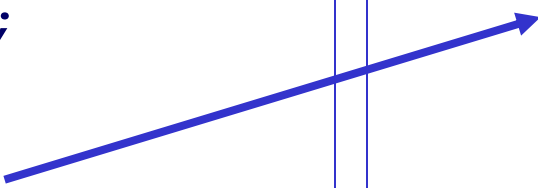
Comunicazione asimmetrica

Processo produttore P:

```
....  
main() {  
    msg M;  
    do{  
        produco (&M) ;  
        ...  
        send(C, M) ;  
    }while(!fine) ;  
}
```

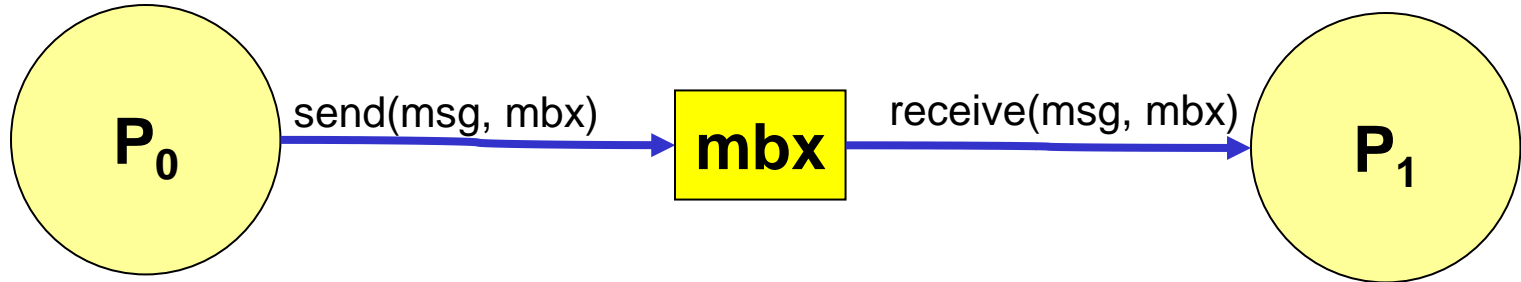
Processo consumatore C:

```
....  
main() {  
    msg M; pid id;  
    do{  
        receive(&id, &M) ;  
        ...  
        consumo (M) ;  
    }while(!fine) ;  
}
```



Comunicazione asimmetrica: destinatario *non è obbligato a conoscere* l'identificatore del *mittente*

Comunicazione indiretta



I processi cooperanti ***non sono tenuti a conoscersi*** reciprocamente e si scambiano messaggi depositandoli/prelevandoli da una ***mailbox condivisa***

- ***mailbox (o porta)*** come ***risorsa astratta condivisibile*** da più processi, che funge da contenitore dei messaggi

Comunicazione indiretta

Proprietà

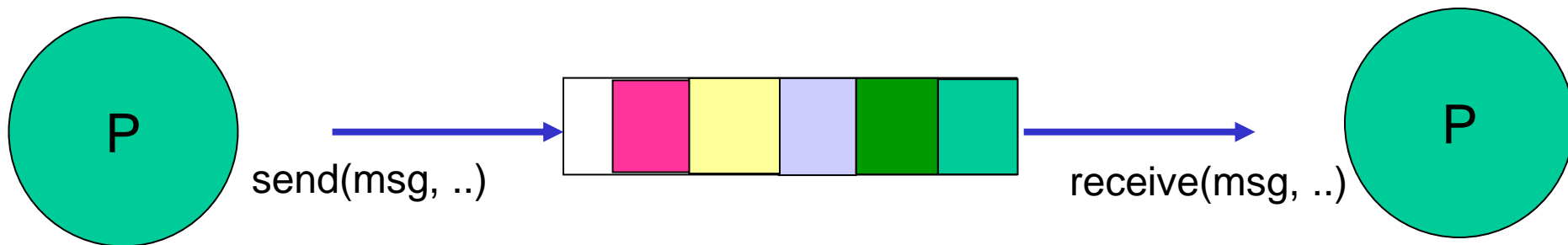
- il canale di comunicazione è rappresentato dalla mailbox (non viene creato automaticamente)
- il canale ***può essere associato a più di 2 processi***:
 - ***mailbox di sistema: multi-a-molti*** (come individuare il processo destinatario di un messaggio?)
 - ***mailbox del processo destinatario: multi-a-uno***
- canale bidirezionale:
 - p0: send(query, mbx)
 - p1: send(answ, mbx)
- per ogni coppia di processi possono esistere più canali (uno per ogni mailbox condivisa)

Buffering del canale

Ogni canale di comunicazione è caratterizzato da una **capacità**: numero dei messaggi che è in grado di gestire contemporaneamente

Gestione usuale secondo politica **FIFO**:

- i messaggi vengono posti in una coda in attesa di essere ricevuti
- la lunghezza massima della coda rappresenta la capacità del canale



Buffering del canale

- Caso semplificato con capacità nulla:** non vi è accodamento perché il canale non è in grado di gestire messaggi in attesa
- processo mittente e destinatario devono **sincronizzarsi all'atto di spedire (send)/ricevere (receive)** il messaggio: **comunicazione sincrona o rendez vous**
 - **send e receive** possono essere (solitamente sono) **sospensive**



Buffering del canale

- **Capacità limitata:** esiste un limite N alla dimensione della coda
 - se la coda ***non è piena***, un nuovo messaggio viene posto in fondo
 - se la coda ***è piena: send è sospensiva***
 - se la coda ***è vuota: receive può essere sospensiva***
- **Capacità illimitata:** lunghezza della coda teoricamente infinita. L'invio sul canale non è sospensivo

Sincronizzazione tra processi

Si è visto che due processi possono interagire per

- **cooperare**: i processi interagiscono allo scopo di perseguire un **obiettivo comune**
- **competere**:
 - i processi possono essere logicamente indipendenti,
ma
 - necessitano della stessa **risorsa** (dispositivo, file, variabile, ...) per la quale sono stati imposti dei vincoli di accesso. Ad esempio:
 - gli accessi di due processi a una risorsa **devono escludersi mutuamente nel tempo**

In entrambi i casi è necessario disporre di **strumenti di sincronizzazione**

Sincronizzazione tra processi

Sincronizzazione permette di imporre vincoli temporali sulle operazioni dei processi interagenti

Ad esempio

nella cooperazione

- per imporre un particolare ***ordine cronologico alle azioni eseguite*** dai processi interagenti
- per garantire che le ***operazioni di comunicazione*** avvengano secondo un ***ordine prefissato***

nella competizione

- per garantire la ***mutua esclusione*** dei processi nell'accesso alla risorsa condivisa

Sincronizzazione tra processi nel modello ad ambiente locale

Mancando la possibilità di condividere memoria:

- Gli accessi alle risorse "condivise" vengono controllati e coordinati da SO
- La sincronizzazione avviene mediante meccanismi offerti da SO che consentono la ***notifica di “eventi” asincroni*** (di solito privi di contenuto informativo o con contenuto minimale) tra un processo ed altri
 - **segnali UNIX**

Sincronizzazione tra processi nel modello ad ambiente globale

Facciamo riferimento a processi che possono condividere variabili (***modello ad ambiente globale***, o a memoria condivisa) per descrivere alcuni strumenti di sincronizzazione tra processi

- **cooperazione**: lo ***scambio di messaggi*** avviene attraverso ***strutture dati condivise*** (ad es., ***mailbox***)
- **competizione**: le risorse sono rappresentate da ***variabili condivise*** (ad esempio, puntatori a file)

In entrambi i casi è necessario ***sincronizzare i processi per coordinarli nell'accesso alla memoria condivisa***:

problema della mutua esclusione

Esempio: comunicazione in ambiente globale con mailbox di capacità MAX

```
typedef struct {  
    coda mbx;  
    int num_msg; } mailbox;
```

Processo mittente:

```
shared mailbox M;  
  
...  
main() {  
    <crea messaggio m>  
    if (M.num_msg < MAX) {  
        /* send: */  
        inserisci(M.mbx,m);  
        M.num_msg++;  
    } /* fine send*/  
  
    ...  
}
```

Processo destinatario:

```
shared mailbox M;  
  
...  
main() {  
    if (M.num_msg > 0) {  
        /* receive: */  
        estrai(M.mbx,m);  
        M.num_msg--;  
    } /* fine receive*/  
    <consumo messaggio m>  
  
    ...  
}
```

Esempio: esecuzione

HP: a T_0 $M.num_msg=1$;

Processo mittente:

```
 $T_0$ : <crea messaggio m>  
 $T_1$ : if (M.num_msg < MAX)  
 $T_2$ : inserisci (M.mbx, m) ;
```

Processo destinatario:

```
.  
. .  
 $T_3$ : if (M.num_msg > 0)  
 $T_4$ : estrai (M.mbx, m) ;  
 $T_5$ : M.num_msg-- ;  
...
```

Sbagliato!

$M.num_msg=0$

- La correttezza della gestione della mailbox dipende dall'ordine di esecuzione dei processi
- È necessario imporre la **mutua esclusione** dei processi **nell'accesso alla variabile M**

Il problema della mutua esclusione

In caso di **condivisione di risorse (variabili)** può essere necessario **impedire accessi concorrenti** alla stessa risorsa

Sezione critica

sequenza di istruzioni mediante la quale un processo accede e può aggiornare variabili condivise

Mutua esclusione

ogni processo esegue le **proprie sezioni critiche** in modo **mutuamente esclusivo** rispetto agli altri processi

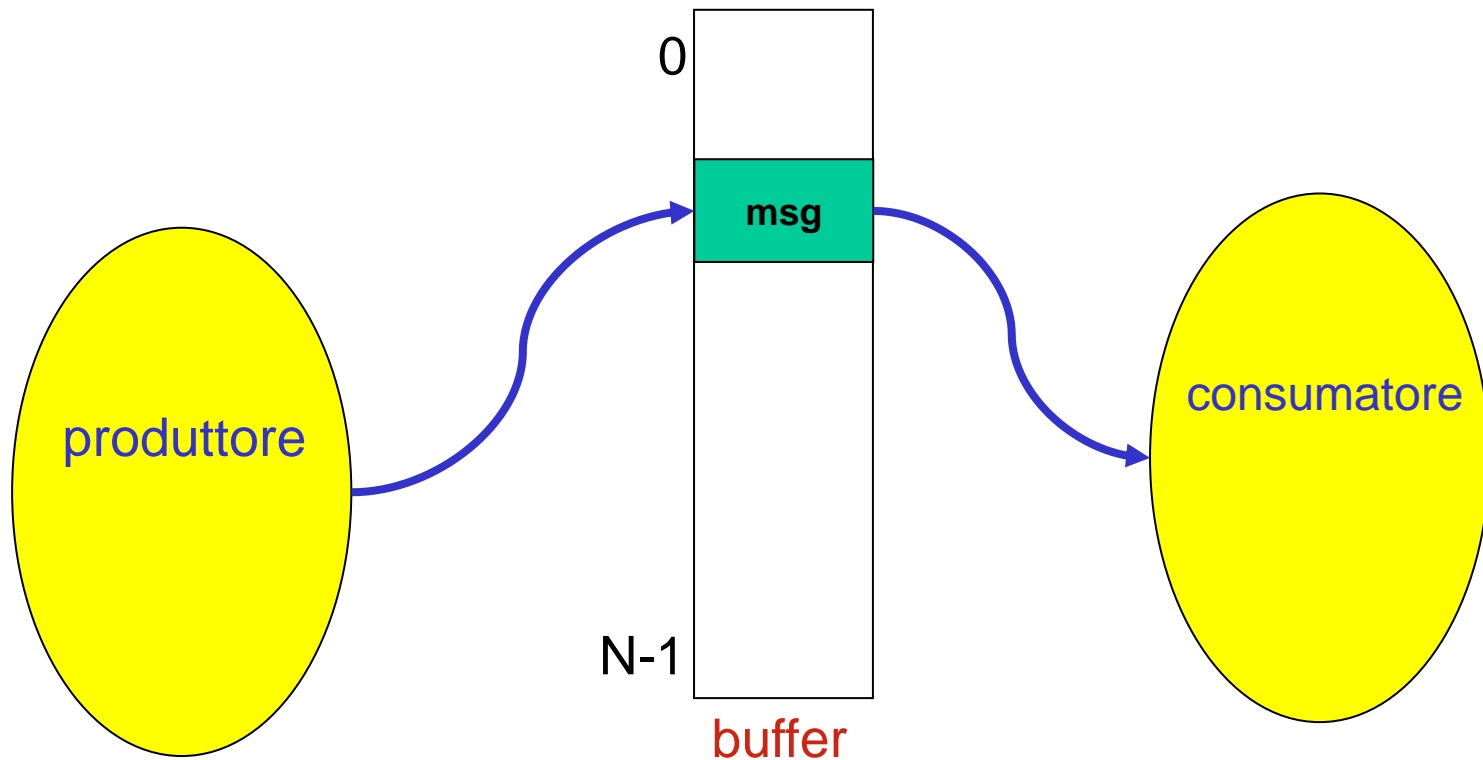
Mutua esclusione

In generale, per garantire la mutua esclusione nell'accesso a variabili condivise, ogni sezione critica è:

- ***preceduta da un prologo (entry section)***, mediante il quale il processo ottiene ***l'autorizzazione all'accesso in modo esclusivo***
- ***seguita da un epilogo (exit section)***, mediante il quale il processo ***rilascia la risorsa***

```
<entry section>  
<sezione critica>  
<exit section>
```

Esempio: produttore & consumatore



HP: *buffer (mailbox) limitato* di dimensione N

Esempio: produttore & consumatore

- Necessità di ***garantire la mutua esclusione*** nell'esecuzione delle ***sezioni critiche*** (accesso e aggiornamento del buffer)
- Necessità di ***sincronizzare i processi***:
 - quando il ***buffer è vuoto***, il consumatore non può prelevare messaggi
 - quando il ***buffer è pieno***, il produttore non può depositare messaggi

Produttore & consumatore: prima soluzione (attesa attiva)

Processo produttore:

....

shared int cont=0;

shared msg Buff [N];

main() {

msg M;

do{

 produco (&M) ;

while (cont==N) ;

 inserisco (M, Buff) ;

 cont=cont+1;

 }while(true) ;

}

Processo consumatore:

....

shared int cont=0;

shared msg Buff [N];

main() {

msg M;

do{

while (cont==0) ;

 prelievo (&M, Buff) ;

 cont=cont-1;

 consumo (M) ;

 }while(true) ;

}

Produttore & consumatore

Problema: finché non si creano le condizioni per effettuare l'operazione di inserimento/prelievo, ogni processo rimane in esecuzione all'interno di un ciclo

```
while (cont==N) ;
```

```
while (cont==0) ;
```

attesa attiva

Per migliorare l'efficienza del sistema, in alcuni SO è possibile utilizzare system call del tipo:

- **dormo()** per **sospendere** il processo che la chiama (stato di waiting e spreco di CPU evitato)
- **sveglia(P)** per **riattivare un processo P sospeso** (se P non è sospeso, non ha effetto e il segnale di risveglio viene perso)

Produttore & Consumatore: seconda soluzione

Processo produttore P:

....

shared msg Buff [N];

shared int cont=0;

main() {

msg M;

do{

 produco (&M) ;

 if (cont==N) *sleep()* ;

 inserisco (M, Buff) ;

 cont = cont + 1;

 if (cont==1)

wakeup (C) ;

 } while (true) ;

} ...

Processo consumatore C:

....

shared msg Buff [N];

shared int cont=0;

main() {

msg M;

do{

 if (cont==0) *sleep()* ;

 prelievo (&M, Buff) ;

 cont=cont-1;

 if (cont==N-1)

wakeup (P) ;

 consumo (M) ;

 }while (true) ;

} ...

Produttore & Consumatore: seconda soluzione

Possibilità di blocco dei processi: ad esempio, consideriamo la sequenza temporale:

1. `cont=0` (buffer vuoto)
2. **C** legge `cont`, poi viene *deschedulato prima di sleep* (stato pronto)
3. **P** inserisce un messaggio, `cont++` (`cont=1`)
4. **P** esegue una `wakeup(C)`: C non è bloccato (è pronto), **il segnale è perso**
5. **C** verifica `cont` e si blocca
6. **P** continua a inserire Messaggi, fino a riempire il buffer

 **blocco di entrambi i processi (deadlock)**

Soluzione: garantire la *mutua esclusione* dei processi nell'esecuzione delle *sezioni critiche (accesso a `cont`, `inserisco` e `prelievo`)*

Possibile soluzione: semafori (Dijkstra, 1965)

Definizione di semaforo

- ***Tipo di dato astratto*** condiviso fra più processi al quale sono applicabili solo due operazioni (***system call a esecuzione non interrompibile***):
 - ***wait (s)***
 - ***signal (s)***
- A una variabile *s* di tipo *semaforo* sono associate:
 - una **variabile intera *s.value*** non negativa con valore iniziale ≥ 0
 - una **coda di processi *s.queue***

Semaforo può essere condiviso da 2 o più processi per risolvere problemi di sincronizzazione (es. mutua esclusione)

System call sui semafori: definizione

```
void wait(s){  
    if (s.value == 0)  
        <processo viene sospeso e descrittore  
        inserito in s.queue>  
    else s.value = s.value-1;  
}
```

```
void signal(s){  
    if (<esiste un processo in s.queue>)  
        <descrittore viene estratto da s.queue  
        e stato modificato in pronto>  
    else s.value = s.value+1;  
}
```

`wait()` / `signal()`

- `wait()`

- in caso di `s.value=0`, implica la **sospensione del processo che la esegue** (stato running→waiting) nella coda `s.queue` associata al semaforo

- `signal()`

- non comporta concettualmente nessuna modifica nello stato del processo che l'ha eseguita, ma **può causare il risveglio di un processo waiting nella coda s.queue**
- la scelta del processo da risvegliare avviene secondo una politica FIFO (il primo processo della coda)

`wait()` e `signal()` agiscono su variabili condivise e pertanto sono a loro volta **sezioni critiche!**

Atomicità di `wait()` e `signal()`

Affinché sia rispettato il vincolo di mutua esclusione dei processi nell'accesso al semaforo (mediante `wait/signal`), ***`wait()` e `signal()` devono essere operazioni indivisibili (azioni atomiche):***

- durante un'operazione sul semaforo (**`wait()`** o **`signal()`**) nessun altro processo può accedere al semaforo fino a che l'operazione non è **completa** o **bloccata** (sospensione nella coda)
- SO che mette a disposizione le primitive di Dijkstra deve realizzare **`wait()` e `signal()`** come operazioni non interrompibili (*system call*)

Esempio di mutua esclusione con semafori

Consideriamo due processi P1 e P2 che condividono una struttura dati D sulla quale vogliamo quindi imporre il ***vincolo di mutua esclusione***:

shared data D;

```
P1 :  
...  
/*sezione critica: */  
Aggiorna1 (&D) ;  
/*fine sez.critica: */  
...
```

```
P2 :  
...  
/*sezione critica: */  
Aggiorna2 (&D) ;  
/*fine sez.critica: */  
...
```

➤ ***Aggiorna1 e Aggiorna2 sono sezioni critiche e devono essere eseguite in modo mutuamente esclusivo***

Esempio di mutua esclusione con semafori

Soluzione: uso di *un semaforo (binario) mutex*, il cui valore è inizializzato a 1 (e può assumere soltanto due valori: 0 e 1)

```
shared data D;  
semaphore mutex;  
mutex.value=1;
```

```
P1 :  
...  
wait(mutex) ;  
Aggiorna1 (&D) ;  
signal(mutex) ;  
...
```

```
P2 :  
...  
wait(mutex)  
Aggiorna2 (&D) ;  
signal(mutex) ;  
...
```

- la soluzione è sempre **corretta**, indipendentemente dalla sequenza di esecuzione dei processi (e dallo scheduling della CPU)

Mutua esclusione con semafori: esecuzione

Ad esempio, verifichiamo la seguente sequenza di esecuzione:

P1:

T₃: wait(mutex) → P1 sospeso
sulla coda; Cambio di Contesto
P1→P2 [P1 waiting, P2 running]

T₇: <esecuzione di Aggiorna2>

T₈: signal(mutex) → mutex.value=1;

...

P2:

T₀: wait(mutex) → mutex.value=0;

T₁: <inizio di Aggiorna2>

T₂: Cambio di Contesto P2→P1
[P2 ready, P1 running]

T₄: <conclus. di Aggiorna2>

T₅: signal(mutex) → risveglio
di P1 [P1 ready, P2 running]

T₆: P2 termina: Cambio di Contesto
P2→P1 [P2 terminated, P1
running]

...

Sincronizzazione di processi cooperanti

Mediante semafori possiamo anche imporre **vincoli temporali** sull'esecuzione di processi cooperanti.

Ad esempio:

P1 :

...

/*fase A : */

faseA (...) ;

/*fine fase A */ ...

P2 :

...

/*fase B: */

faseB (...) ;

/*fine fase B */ ...

Obiettivo: vogliamo imporre che l'esecuzione della fase A (in P1) preceda sempre l'esecuzione della fase B (in P2)

Sincronizzazione di processi cooperanti

Soluzione: si introduce un **semaforo** *sync*, *inizializzato a 0*

```
semaphore sync;
```

```
sync.value=0
```

P1 :

...

```
faseA (...);
```

```
signal(sync);
```

...

P2 :

...

```
wait(sync);
```

```
faseB (...);
```

...

- se P2 esegue la `wait()` prima della terminazione della fase A, P2 viene sospeso
- quando P1 termina la fase A, può sbloccare P2, oppure portare il valore del semaforo a 1 (se P2 non è ancora arrivato alla `wait`)

Produttore & consumatore con semafori

- Problema di **mutua esclusione**
 - produttore e consumatore non possono accedere contemporaneamente al buffer
 - semaforo **binario mutex**, con valore iniziale a 1
- Problema di **sincronizzazione**
 - produttore non può scrivere nel buffer se pieno
 - semaforo **vuoto**, con valore iniziale a **N**;
valore dell'intero associato a **vuoto** rappresenta il numero di elementi liberi nel buffer
 - consumatore non può leggere dal buffer se vuoto
 - semaforo **pieno**, con valore iniziale a **0**;
valore dell'intero associato a **pieno** rappresenta il numero di elementi occupati nel buffer

Produttore & consumatore con semafori

```
shared msg Buff [N];  
shared semaforo mutex; mutex.value=1;  
shared semaforo pieno; pieno.value=0;  
shared semaforo vuoto; vuoto.value=N;
```

```
/* Processo produttore P:*/
```

```
main() {  
    msg M;  
    do{  
        produco (&M) ;  
        wait(vuoto) ;  
        wait(mutex) ;  
        inserisco (M, Buff) ;  
        signal(mutex) ;  
        signal(pieno) ;  
    } while(true) ;  
}
```

```
/* Processo consumatore C:*/
```

```
main() {  
    msg M;  
    do{  
        wait(pieno) ;  
        wait(mutex) ;  
        prelievo (&M, Buff) ;  
        signal(mutex) ;  
        signal(vuoto) ;  
        consumo (M) ;  
    } while(true) ;  
}
```


Strumenti di sincronizzazione

Semafori:

- consentono una **efficiente realizzazione di politiche di sincronizzazione**, anche complesse, tra processi
- correttezza della realizzazione **completamente a carico del programmatore**

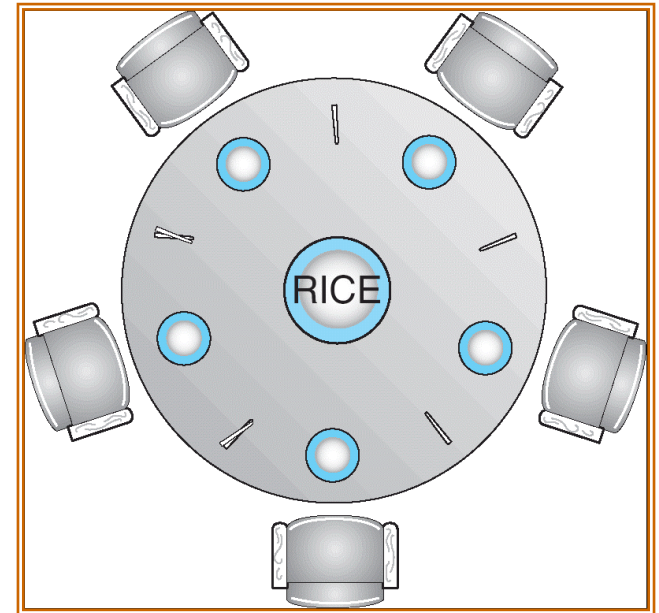
Alternative: esistono **strumenti di più alto livello** (costrutti di linguaggi di programmazione) che eliminano a priori il problema della mutua esclusione sulle variabili condivise

- Variabili condizione
- Regioni critiche
- Monitor
- **synchronized** in Java
- ...

Problema dei “dining-philosophers” (1)

Problema molto noto in letteratura

- Ci sono 5 filosofi seduti a una tavola rotonda
- Ognuno ha davanti a sé un piatto e tra ogni piatto c'è una bacchetta
- Per mangiare **un filosofo usa due bacchette**, quella alla sua sinistra e quella alla sua destra che però sono condivise con i filosofi vicini
- Di conseguenza **due filosofi vicini non possono mangiare contemporaneamente**
- I filosofi oltre a mangiare, naturalmente, pensano ma questa attività avviene in modo indipendente: **l'unico momento in cui si devono sincronizzare è quindi quando mangiano**

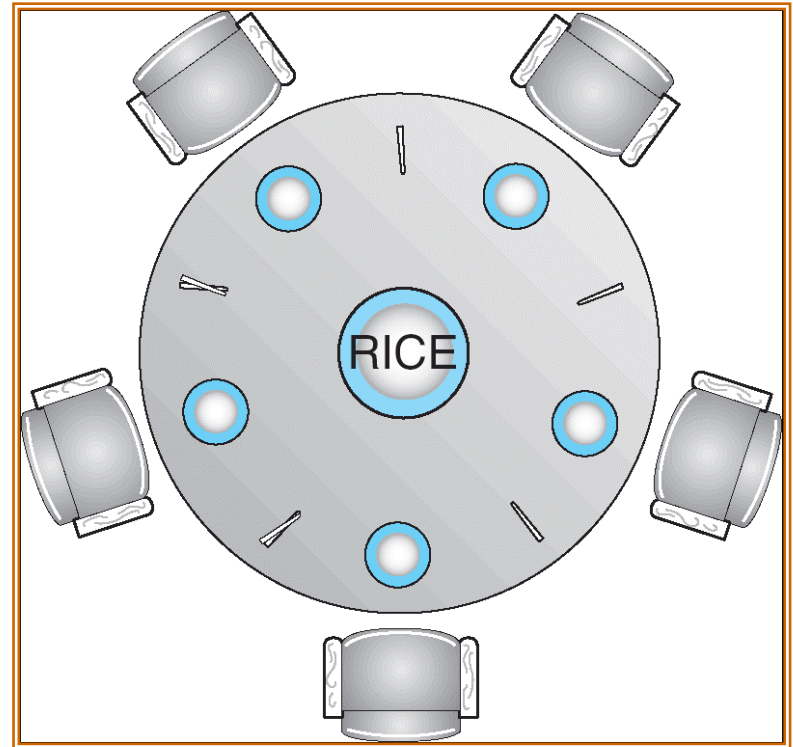


Problema dei “dining-philosophers” (2)

Risorse condivise:

- Ciotola di riso (data set)
- Semafori **bastoncini**[5] inizializzati a 1

Provare a pensare a soluzioni di sincronizzazione mediante il solo uso di semafori e possibili problemi

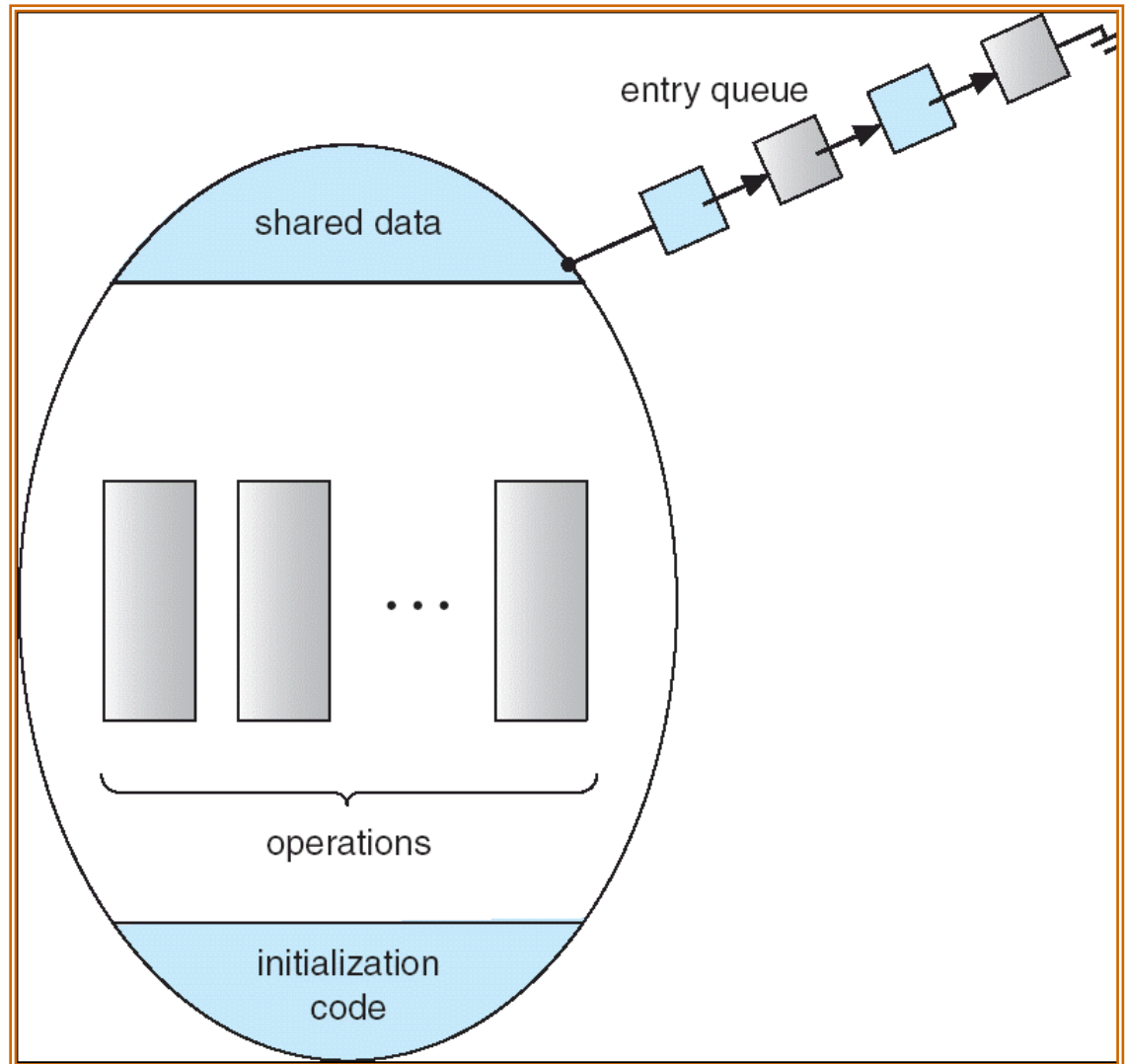


Si possono verificare situazioni di blocco indefinito?

Meccanismi alternativi di sincronizzazione: monitor

Ve ne occuperete soprattutto in corsi successivi (in questo corso solo esercizi d'esame piuttosto semplici...)

Coda di accesso **regolata e disciplinata** verso i dati condivisi, magari con priorità differenziate



Sincronizzazione tra processi UNIX: i segnali

Sincronizzazione tra processi

Processi interagenti possono avere bisogno di meccanismi di **sincronizzazione**

Ad esempio, abbiamo visto e rivedremo diffusamente il caso di ***processi pesanti UNIX*** che vogliano accedere allo ***stesso file in lettura/scrittura*** (***sincronizzazione di produttore e consumatore***)

UNIX: non c'è condivisione alcuna di spazio di indirizzamento tra processi. Serve un meccanismo di sincronizzazione per modello ad ambiente locale



Sincronizzazione tra processi

Segnali

Sono ***interruzioni software*** a un processo, che notifica un evento asincrono. Ad esempio segnali:

- generati da ***terminale*** (es. CTRL+C)
- generati da ***altri processi***
- generati dal ***kernel SO*** in seguito ad ***eccezioni HW*** (violazione dei limiti di memoria, divisioni per 0, ...)
- generati dal ***kernel SO*** in seguito a ***condizioni SW*** (time-out, scrittura su pipe chiusa come vedremo in seguito, ...)

Segnali UNIX

Un segnale può essere inviato

- ❑ dal kernel SO a un processo
- ❑ da un processo utente ad altri processi utente (es. comando **kill**)

Quando un processo riceve un segnale, può comportarsi in ***tre modi diversi***

1. ***gestire il segnale*** con una funzione ***handler*** definita dal programmatore
2. ***eseguire un'azione predefinita*** dal SO (azione di ***default***)
3. ***ignorare il segnale*** (nessuna reazione)

Nei primi due casi, processo ***reagisce in modo asincrono*** al segnale

1. interruzione dell'esecuzione
2. esecuzione dell'azione associata (***handler*** o ***default***)
3. ritorno alla prossima istruzione del codice del processo interrotto

Segnali UNIX

- Per ogni versione di UNIX esistono vari tipi di segnale (in Linux, 32 segnali), ognuno identificato da un intero
- Ogni segnale è ***associato a un particolare evento*** e prevede una ***specifica azione di default***
- È possibile riferire i segnali con identificatori simbolici (**SIGxxxx**):

SIGKILL, SIGSTOP, SIGUSR1, ...
- L'associazione tra nome simbolico e intero corrispondente (che dipende dalla versione di UNIX) è specificata nell'header file **<signal.h>**

Segnali UNIX (Linux): signal.h

```
#define SIGHUP      1      /* Hangup (POSIX).  Action: exit */
#define SIGINT      2      /* Interrupt (ANSI).  CTRL+C.  Action: exit */
#define SIGQUIT     3      /* Quit (POSIX).  Action: exit, core dump */
#define SIGILL      4      /* Illegal instr (ANSI).  Action: exit,
                           core dump */
...
#define SIGKILL     9      /* Kill, unblockable (POSIX).  Action: exit */
#define SIGUSR1     10     /* User-def sig1 (POSIX).  Action: exit */

#define SIGSEGV     11     /* Segm. violation (ANSI).  Action: exit, core
                           dump */
#define SIGUSR2     12     /* User-def sig2 (POSIX).  Action: exit */
#define SIGPIPE     13     /* Broken pipe (POSIX).  Action: exit */
#define SIGALRM     14     /* Alarm clock (POSIX).  Action: exit */
#define SIGTERM     15     /* Termination (ANSI).  Action: exit */
...
#define SIGCHLD     17     /* Chld stat changed (POSIX).  Action: ignore */
#define SIGCONT     18     /* Continue (POSIX).  Action: ignore */
#define SIGSTOP     19     /* Stop, unblockable (POSIX).  Action: stop */
...
```

Gestione dei segnali UNIX

Quando un processo riceve un segnale, può gestirlo in 3 modi diversi:

- gestire il segnale con una ***funzione handler definita dal programmatore***
- eseguire ***un'azione predefinita*** dal SO (azione di ***default***)
- ***ignorare*** il segnale

NB: non tutti i segnali possono essere gestiti in modalità scelta esplicitamente dai processi: **SIGKILL** e **SIGSTOP** non sono ***né intercettabili, né ignorabili***

- qualunque processo, alla ricezione di **SIGKILL** o **SIGSTOP** esegue sempre ***l'azione di default***

System call signal

Ogni processo può *gestire esplicitamente un segnale utilizzando la system call signal()*:

```
typedef void (*handler_t)(int);  
handler_t signal(int sig, handler_t handler);
```

- **sig** è l'intero (o il nome simbolico) che individua il segnale da gestire
- il parametro **handler** è un puntatore a una funzione che indica l'azione da associare al segnale. **handler()** può:
 - puntare alla routine di gestione dell'interruzione (**handler**)
 - valere **SIG_IGN** (nel caso di segnale ignorato)
 - valere **SIG_DFL** (nel caso di azione di default)
- ritorna un puntatore a funzione:
 - al precedente gestore del segnale
 - **SIG_ERR(-1)**, nel caso di errore

Esempi di uso di `signal()`

```
#include <signal.h>
void gestore(int) ;
...
int main() {
...
signal(SIGUSR1, gestore); /*SIGUSR1 gestito */
...
signal(SIGUSR1, SIG_DFL); /*USR1 torna a default */
signal(SIGKILL, SIG_IGN); /*errore! SIGKILL non è
                             ignorabile */
...
}
```

Routine di gestione del segnale (*handler*)

Caratteristiche:

- *handler* prevede sempre ***un parametro formale*** di tipo *int* che rappresenta il ***numero del segnale effettivamente ricevuto***
- *handler* non restituisce alcun risultato

```
void handler(int signum) {  
    . . . .  
    . . . .  
    return;  
}
```

Gestione di segnali con handler

Non sempre *l'associazione segnale/handler* è *durevole*:

- alcune implementazioni di UNIX (BSD, SystemV r3 e seguenti), prevedono che *l'azione rimanga installata anche dopo la ricezione del segnale*
- in alcune realizzazioni (SystemV prime release), dopo l'attivazione, *handler ripristina automaticamente l'azione di default*. In questi casi, occorre riagganciare il segnale all'handler:

```
int main() {  
    ...  
    signal(SIGUSR1, f);  
    ...}
```

```
void f(int s) {  
    signal(SIGUSR1, f);  
    ....  
}
```

Esempio: sfruttamento del parametro di handler

```
/* file segnal1.c */
#include <signal.h>
void handler(int);

int main(){
    if (signal(SIGUSR1, handler)== SIG_ERR)
        perror("prima signal non riuscita\n");
    if (signal(SIGUSR2, handler)== SIG_ERR)
        perror("seconda signal non riuscita\n");
    for (;;) ;
}

void handler (int signum){
    if (signum == SIGUSR1) printf("ricevuto sigusr1\n");
    else if (signum == SIGUSR2) printf("ricevuto sigusr2\n");
}
```


Esempio: esecuzione & comando kill

```
carlo@info1-linux:~/esercizi$ vi segnal1.c
carlo@info1-linux:~/esercizi$ gcc segnal1.c
carlo@info1-linux:~/esercizi$ ./a.out&
[1] 313
carlo@info1-linux:~/esercizi$ kill -SIGUSR1 313
carlo@info1-linux:~/esercizi$ ricevuto sigusr1

carlo@info1-linux:~/esercizi$ kill -SIGUSR2 313
carlo@info1-linux:~/esercizi$ ricevuto sigusr2

carlo@info1-linux:~/esercizi$ kill -9 313
carlo@info1-linux:~/esercizi$
[1]+  Killed                  a.out
carlo@info1-linux:~/esercizi$
```

Esempio: gestore del SIGCHLD

***SIGCHLD** è il segnale che il **kernel SO** invia a un processo padre quando uno dei suoi figli termina*

Tramite l'uso di segnali è possibile svincolare il padre da un'attesa esplicita della terminazione del figlio, mediante un'apposita funzione **handler** per la gestione di **SIGCHLD**:

- la funzione **handler** verrà attivata in modo asincrono alla ricezione del segnale
- **handler chiamerà *wait()*** con cui il padre potrà raccogliere ed eventualmente gestire lo stato di terminazione del figlio

Esempio: gestore del SIGCHLD

```
#include <signal.h>
void handler(int);

int main(){
    int PID, i;
    PID=fork();
    if (PID>0){ /* padre */
        signal(SIGCHLD, handler);
        for (i=0; i<10000000; i++); /* attività del padre..*/
        exit(0); }
    else{ /* figlio */
        for (i=0; i<1000; i++); /* attività del figlio..*/
        exit(1); }
}

void handler (int signum){
    int status;
    wait(&status);
    printf("stato figlio:%d\n", status>>8);}
```

Segnali & `fork()`

Le *associazioni segnali-azioni* vengono registrate in *User Structure del processo*

Siccome:

- `fork()` copia *User Structure* del padre in quella del figlio
 - padre e figlio condividono lo stesso codice
- quindi
- il figlio eredita dal padre le informazioni relative alla gestione dei segnali:
 - *ignora gli stessi segnali ignorati* dal padre
 - *gestisce con le stesse funzioni gli stessi segnali gestiti* dal padre
 - *segnali a default del figlio sono gli stessi del padre*
- ovviamente `signal()` del figlio successive alla `fork()` non hanno effetto sulla gestione dei segnali del padre

Segnali & `exec()`

Sappiamo che

- `exec()` sostituisce codice e dati del processo invocante
- ***User Structure*** viene mantenuta, tranne le informazioni legate al codice del processo (ad esempio, le funzioni di gestione dei segnali, che dopo `exec()` non sono più visibili)

quindi

- dopo `exec()`, un processo:
 - ignora gli stessi segnali ignorati prima di `exec()`
 - i segnali a default rimangono a default

ma

- i segnali che prima erano gestiti, vengono riportati a default

Esempio

```
/* file segnali2.c */
#include <signal.h>

int main(){
    signal(SIGINT, SIG_IGN);
    execl("/bin/sleep", "sleep", "30", (char *)0);
}
```

NB: il comando shell `sleep N` sospende il processo invocante per N secondi

Esempio: esecuzione

```
carlo@info1-linux:~/esercizi$ gcc segnali2.c
carlo@info1-linux:~/esercizi$ a.out&
[1] 500
carlo@info1-linux:~/esercizi$ kill -SIGINT 500
carlo@info1-linux:~/esercizi$ kill -9 500
carlo@info1-linux:~/esercizi$
[1]+  Killed                  a.out
carlo@info1-linux:~/esercizi$
```

System call `kill()`

I processi possono inviare segnali ad altri processi invocando la system call `kill()`

```
int kill(int pid, int sig);
```

- `sig` è l'intero (o il nome simbolico) che individua il segnale da inviare
- il parametro `pid` specifica il destinatario del segnale:
 - `pid > 0`: l'intero è il ***pid*** dell'unico processo ***destinatario***
 - `pid = 0`: il segnale è spedito a tutti i processi appartenenti al ***gruppo del mittente***
 - `pid < -1`: il segnale è spedito a tutti i processi con ***groupid*** uguale al valore assoluto di `pid`
 - `pid == -1`: vari comportamenti possibili (Posix non specifica)

Esempio di uso di `kill()`

```
#include <stdio.h>
#include <signal.h>

int cont=0;

void handler(int signo){
    printf ("Proc. %d: ricevuti n. %d segnali %d\n",
        getpid(),cont++, signo);
}

int main (){
    int pid;
    signal(SIGUSR1, handler);
    pid = fork();
    if (pid == 0) /* figlio */
        for (;;) ;
    else /* padre */
        for(;;) kill(pid, SIGUSR1);
}
```

Segnali: altre system call

`sleep()`

```
unsigned int sleep(unsigned int N)
```

- provoca la sospensione del processo per N secondi (al massimo)
- se il ***processo riceve un segnale durante il periodo di sospensione, viene risvegliato prematuramente***
 - restituisce 0 se la sospensione non è stata interrotta da segnali
 - se il risveglio è stato causato da un segnale al tempo ***x***, `sleep()` restituisce il numero di secondi non utilizzati dell'intervallo di sospensione (***N-x***)

Esempio d'uso di `sleep()`

```
/* provasleep.c*/
#include <signal.h>

void stampa(int signo){
    printf("sono stato risvegliato!\n");
}

int main(){
    int k;
    signal(SIGUSR1, stampa);
    k=sleep(1000);
    printf("Valore di k: %d\n", k);
    exit(0);
}
```

Esempio d'uso di `sleep()`

```
bash-2.05$ gcc -o pr provasleep.c
bash-2.05$ pr&
[1] 2839
bash-2.05$ kill -SIGUSR1 2839
bash-2.05$ sono stato risvegliato!!
Valore di k: 987

[1]+  Done          pr
bash-2.05$
```

Segnali: altre system call

`alarm()`

```
unsigned int alarm(unsigned int N)
```

- imposta un timer che ***dopo N secondi invierà allo stesso processo il segnale SIGALRM***
- ritorna:
 - 0, se non vi erano time-out impostati in precedenza
 - il numero di secondi mancante allo scadere del time-out precedente

NB: ***comportamento di default*** associato a ricezione di SIGALRM è la terminazione

Segnali: altre system call

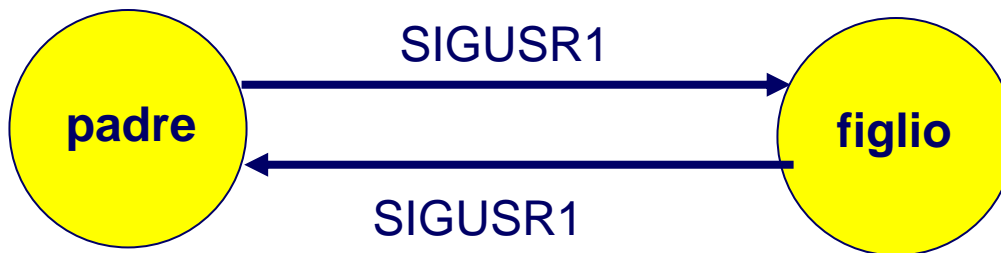
`pause()`

```
int pause(void)
```

- *sospende il processo fino alla ricezione di un qualunque segnale*
- ritorna -1 (`errno = EINTR`)

Esempio

Due processi (padre e figlio) si sincronizzano alternativamente mediante il segnale SIGUSR1 (gestito da entrambi con la funzione *handler*):



```
int ntimes = 0;
void handler(int signo){
    printf ("Processo %d ricevuto #%d volte il segnale
%d\n", getpid(), ++ntimes, signo);
}
```

```
int main () {
    int pid, ppid;
    signal(SIGUSR1, handler);
    if ((pid = fork()) < 0) /* fork fallita */
        exit(1);
    else if (pid == 0) { /* figlio */
        ppid = getppid(); /* PID del padre */
        for (;;) {
            printf("FIGLIO %d\n", getpid());
            sleep(1);
            kill(ppid, SIGUSR1);
            pause();
        }
    }
    else /* padre */
        for(;;) { /* ciclo infinito */
            printf("PADRE %d\n", getpid());
            pause();
            sleep(1);
            kill(pid, SIGUSR1);
        }
}
```


Modello affidabile dei segnali

Aspetti:

- il gestore rimane **installato**? In caso negativo, è possibile comunque reinstallare il gestore all'interno dell'handler

```
void handler(int s) {  
    signal(SIGUSR1, handler);  
    printf("Processo %d: segnale %d\n", getpid(), s);  
    ...}
```

Che cosa succede
se qui arriva un
nuovo segnale?

- che cosa succede se arriva il segnale durante l'esecuzione dell'handler?
 - innestamento delle routine di gestione?*
 - perdita del segnale?*
 - accodamento dei segnali* (segnali *reliable*, *BSD 4.2*)

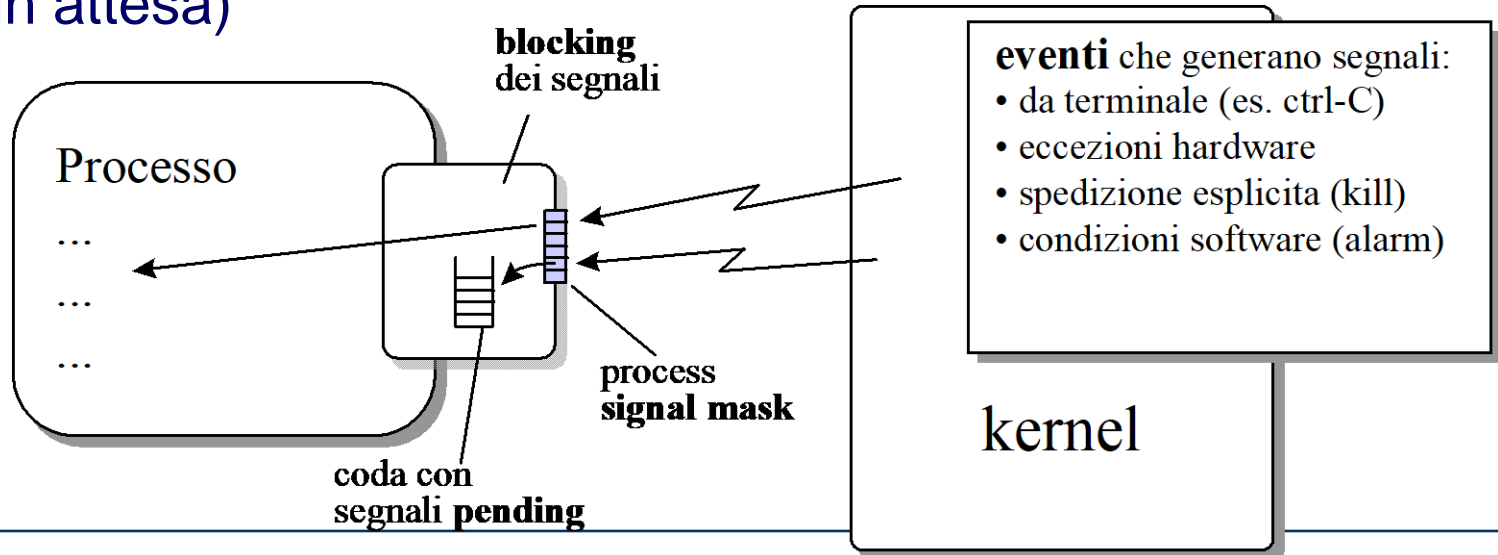
Interrompibilità di system call

System call possono essere classificate in

- ***slow*** system call: possono richiedere ***tempi di esecuzione non trascurabili*** dovuti a periodi di attesa (es. lettura da un dispositivo di I/O lento)
- ***non-slow*** system call
- ***solo slow system call*** sono ***interrompibili da parte di segnali***. In caso di interruzione:
 - ritorna -1
 - **`errno`** vale EINTR
- possibilità di ri-esecuzione della system call:
 - automatica (BSD 4.3)
 - non automatica, ma comandata dal processo (in base al valore di **`errno`** e al valore restituito)

Modello "reliable" segnali

- Occorrenze "contemporanee" di uno stesso segnale dovrebbero essere memorizzate:
 - ▣ processo le riceve tutte (1 contatore per ogni segnale)
 - ▣ non si perdono segnali
- Blocco dei segnali diretti a un processo P e gestione successiva
→ regioni critiche
- I segnali in attesa sono detti pending: la process signal mask definisce quali segnali bloccare per un processo P (i segnali sono messi in attesa)



Attenzione

1) Il codice dei gestori dei segnali non dovrebbe mai contenere chiamate a primitive di I/O, che sono lente e potenzialmente bloccanti. Negli esempi di codice visti in questi lucidi, le chiamate a `printf` sono usate esclusivamente a fini didattici.

2) Il modello di comunicazione tra il codice di gestione dei segnali e il codice principale dell'applicazione attraverso l'uso di una variabile statica rappresenta invece una “best practice”. Tuttavia, in questi casi è importante che la variabile utilizzata sia di tipo **`sig_atomic_t`** e che essa venga dichiarata con la keyword **`volatile`**.

- ❑ <https://en.cppreference.com/w/c/program/signal> → "On return from a signal handler, the value of any object modified by the signal handler that is not **`volatile sig_atomic_t`** [...] is undefined."

Dettagli su `sig_atomic_t` e `volatile`

- Infatti, lo standard ISO C definisce **`sig_atomic_t` come un tipo di dato che può essere acceduto senza interruzioni**. Ovverosia, nessuna lettura da o scrittura su una variabile di tipo `sig_atomic_t` sarà interrotta, per esempio dall'occorrenza di un nuovo segnale.
- La keyword **`volatile`** invece dà istruzione al compilatore di non ottimizzare l'accesso alla variabile corrispondente, che in questo caso produrrebbe un comportamento non corretto del nostro programma.
- Se non usassimo `sig_atomic_t` e `volatile` nella definizione della variabile, sarebbe molto probabile che alcuni cambiamenti allo stato di una variabile nel codice del gestore del segnale non venissero notati dal codice principale dell'applicazione a causa della ricezione di ulteriori segnali o di ottimizzazioni del compilatore.

sigaction()

- Come abbiamo visto, la primitiva **signal()** non è **portabile** perché ha una semantica diversa in diverse versioni di Unix. Per ovviare a questo problema, POSIX.1 introduce la sigaction()
- La **sigaction()** permette di esaminare e/o modificare l'azione associata con un particolare segnale. Si noti che POSIX.1 richiede che un segnale rimanga installato (fino a una modifica esplicita del comportamento).
- Con la sigaction() è anche possibile specificare il restart automatico delle system call interrotte da un segnale.

- **SIGACTION**

```
#include <signal.h>
int sigaction(int sig,
               const struct sigaction *act,
               struct sigaction *oldact)
```

// restituisce 0 in caso di successo, -1 in caso di errore

sigaction(): esempio

- Il gestore del SIGCHLD:

```
struct sigaction sa;

sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
sa.sa_handler = sigchldHandler;
if (sigaction(SIGCHLD, &sa, NULL) == -1) {
    perror("sigaction");
    exit(1);
}
```

- Dove sigchldHandler è il nome della funzione che verrà eseguita al ricevimento del segnale SIGCHLD
- sigchldHandler() tipicamente conterrà la wait per non lasciare figli zombie.

signal() vs. sigaction() in sintesi

- **signal() con semantica variabile reliable/unreliable**
 - **unreliable in alcune versioni** di Unix/Linux
 - **segnali da reinstallare** ogni volta, race critica tra inizio handler e reinstallazione handler come prima istruzione dell'handler
 - possibile **esecuzione innestata dell'handler** se ricezione dello stesso segnale quando siamo ancora nell'handler
- **sigaction() invece è sempre reliable**
 - semantica ben definita, identica in ogni versione di Unix/Linux
 - non c'è bisogno di reinstallare l'handler
 - non perdiamo segnali: il segnale che ha causato l'attivazione dell'handler è automaticamente bloccato fino alla fine dell'esecuzione dell'handler stesso

Comunicazione tra processi UNIX

Interazione tra processi UNIX

Processi UNIX non possono condividere memoria
(*modello ad ambiente locale*)

Interazione tra processi può avvenire

- mediante la **condivisione di file**
 - complessità: realizzazione della sincronizzazione tra i processi
- attraverso **specifici strumenti di Inter Process Communication (IPC)**:
 - tra processi **sulla stessa macchina**
 - **pipe** (tra processi della stessa gerarchia)
 - **fifo** (qualunque insieme di processi)
 - tra processi in nodi diversi della stessa **rete**:
 - **socket**

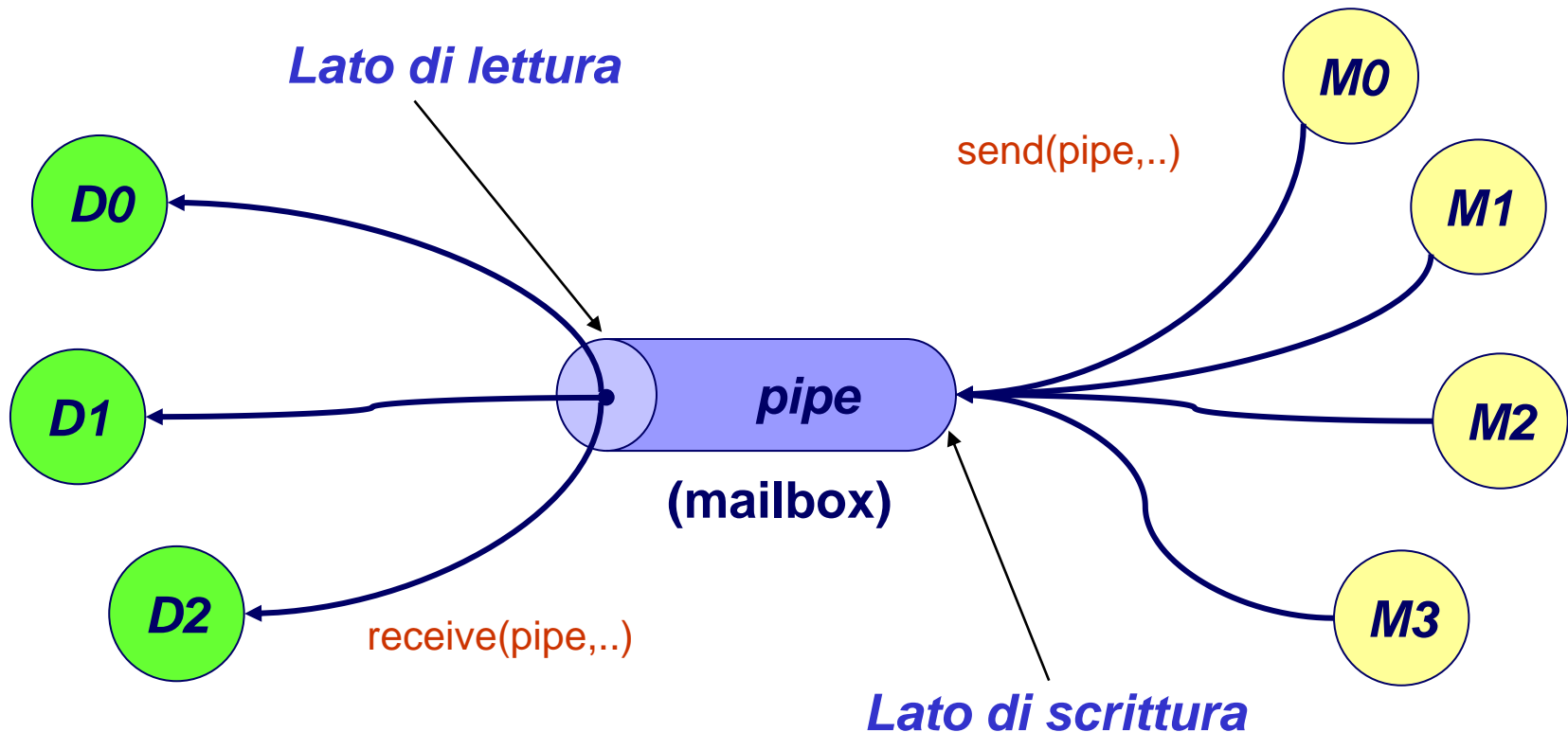
pipe

La pipe è un **canale di comunicazione** tra processi:

- **unidirezionale**: accessibile mediante due estremi distinti, uno di lettura e uno di scrittura
- (teoricamente) **molti-a-molti**:
 - più processi possono **spedire messaggi** attraverso la stessa pipe
 - più processi possono **ricevere messaggi** attraverso la stessa pipe
- **capacità limitata**:
 - in grado di gestire **l'accodamento di un numero limitato di messaggi**, gestiti in modo FIFO. Limite stabilito dalla **dimensione della pipe** (es. 4096B)

Comunicazione attraverso pipe

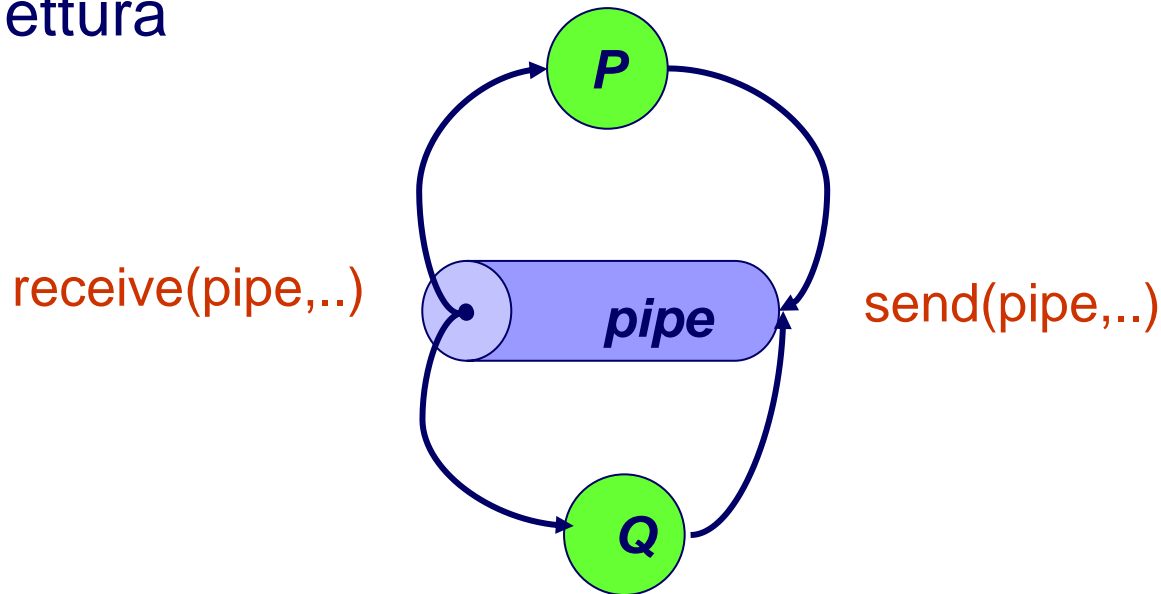
Mediante la pipe, la comunicazione tra processi è **indiretta** (senza naming esplicito): **modello mailbox**



Pipe: unidirezionalità/bidirezionalità

Uno stesso processo può:

- sia **depositare messaggi nella pipe** (*send*), mediante il lato di scrittura
- sia **prelevare messaggi dalla pipe** (*receive*), mediante il lato di lettura



la pipe può anche consentire una **comunicazione** “**bidirezionale**” tra P e Q (ma il programmatore deve rigidamente disciplinarne l’uso per l’utilizzo corretto)

System call `pipe()`

Per creare una pipe:

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

fd è un **vettore di 2 file descriptor**, che verranno inizializzati dalla system call in caso di successo:

- *fd[0]* rappresenta il lato di **lettura** della pipe
- *fd[1]* è il lato di **scrittura** della pipe

la system call `pipe()` restituisce:

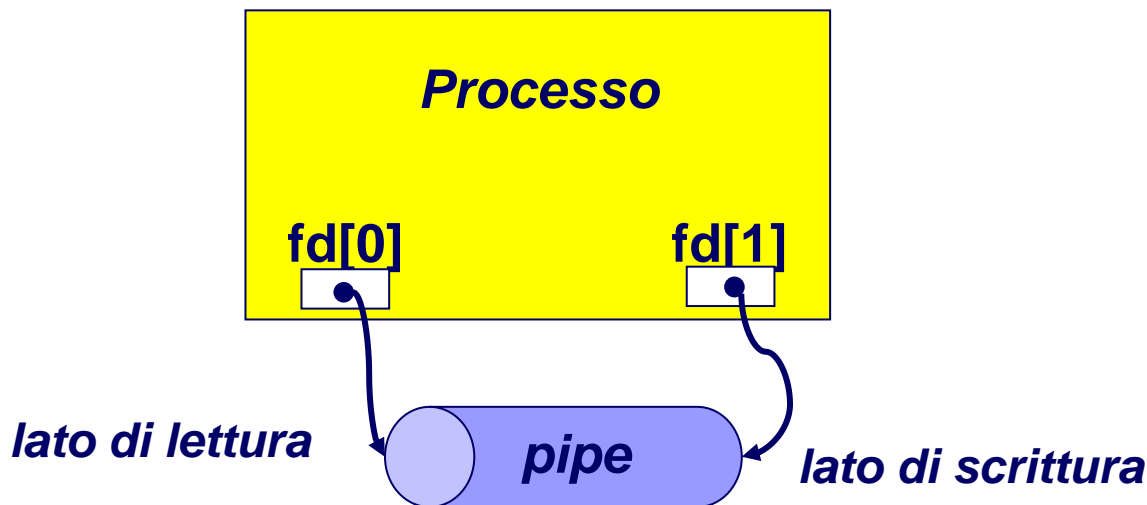
- un valore negativo, in caso di fallimento
- 0, se ha successo

Creazione di una pipe

Se `pipe (fd)` ha successo:

vengono allocati due nuovi elementi nella tabella dei file aperti del processo e i rispettivi file descriptor vengono assegnati a `fd[0]` e `fd[1]`

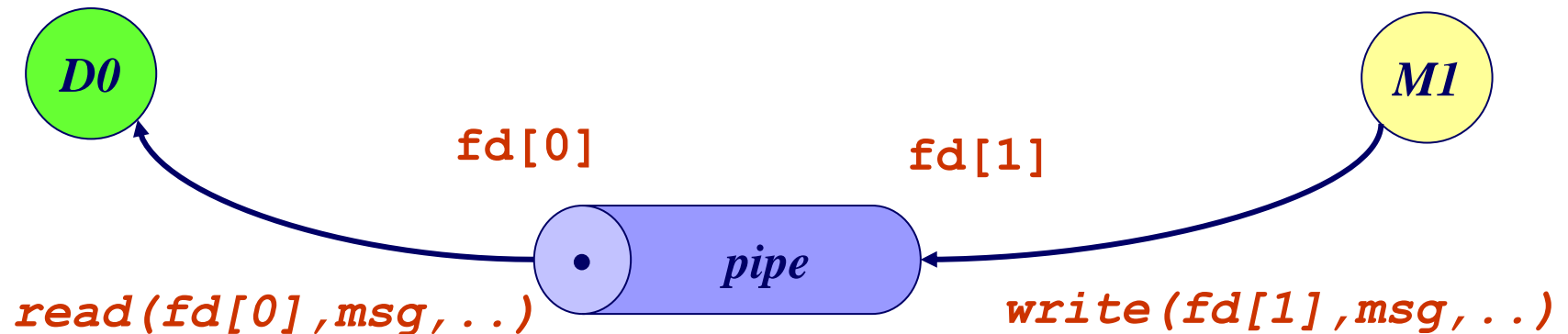
- `fd[0]`: lato di lettura (*receive*) della pipe
- `fd[1]`: lato di scrittura (*send*) della pipe



Omogeneità con i file

Ogni lato di accesso alla pipe è visto dal processo in ***modo omogeneo a qualunque altro file*** (file descriptor)

- si può accedere alla pipe mediante le system call di lettura/scrittura su file **`read()`**, **`write()`**



- **`read()`** : operazione di ricezione
- **`write()`** : operazione di invio

Sincronizzazione automatica delle pipe

Il canale (*pipe*) ha **capacità limitata**. Come nel caso di produttore/consumatore è necessario sincronizzare i processi. **Sincronizzazione automatica** in UNIX:

- se la *pipe* è **vuota**: un processo che **legge si blocca**
- se la *pipe* è **piena**: un processo che **scrive si blocca**

□ **Sincronizzazione automatica:** `read()` e `write()` sono implementate **in modo sospensivo** dal SO UNIX

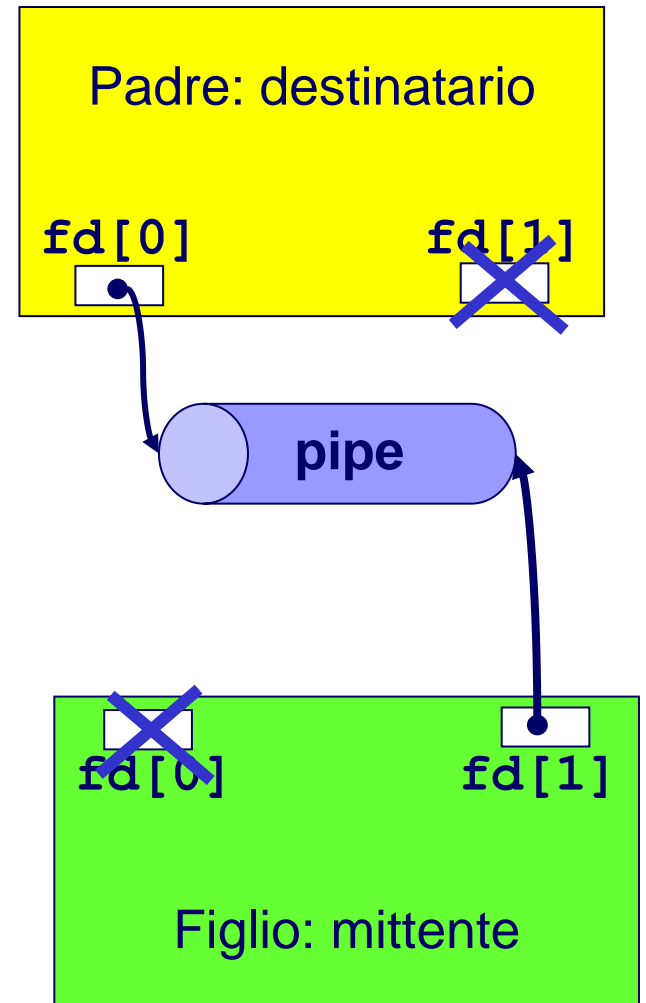
Quali processi possono comunicare mediante pipe?

Per mittente e destinatario *il riferimento al canale di comunicazione è un array di file descriptor:*

- ❑ soltanto i **processi appartenenti a una stessa gerarchia** (cioè, che hanno un **antenato** in comune) possono scambiarsi messaggi mediante pipe. Ad esempio, possibilità di comunicazione:
 - tra **processi fratelli** (che **ereditano** la pipe dal processo padre)
 - tra un processo **padre** e un processo **figlio**
 - tra **nonno** e **nipote**
 - ...

Esempio: comunicazione tra padre e figlio

```
int main() {  
    int pid;  
    char msg[]="ciao babbo";  
    int fd[2];  
    pipe(fd);  
    pid=fork();  
    if (pid==0) {  
        /* figlio */  
        close(fd[0]);  
        write(fd[1], msg, 10);  
        ...}  
    else {  
        /* padre */  
        close(fd[1]);  
        read(fd[0], msg, 10);  
        ...  
    }  
}
```



Ogni processo chiude il lato
pipe che non usa

Chiusura di pipe

Ogni processo può chiudere *un estremo della pipe* con la system call `close()`

- la comunicazione non è più possibile su di un estremo della pipe *quando tutti i processi che avevano visibilità di quell'estremo* hanno compiuto una `close()`

Se un processo P tenta:

- lettura da una pipe vuota il cui lato di scrittura è effettivamente chiuso: `read` ritorna 0
- scrittura da una pipe il cui lato di lettura è effettivamente chiuso: `write` ritorna -1, e il segnale **SIGPIPE** viene inviato a P (*broken pipe*)

Esempio (1)

```
/* Sintassi: progr N
padre(destinatario) e figlio(mittente) si scambiano una
   sequenza di messaggi di dimensione (DIM) costante; la
   lunghezza della sequenza non è nota a priori;
   destinatario interrompe sequenza di scambi di messaggi
   dopo N secondi */

#include <stdio.h>
#include <signal.h>
#define DIM 10

int fd[2];
void fine(int signo);
void timeout(int signo);
```

Esempio (2)

```
int main(int argc, char **argv){
    int pid, N; char messaggio[DIM]="ciao ciao ";
    if (argc!=2){
        printf("Errore di sintassi\n");
        exit(1);
    }
    N=atoi(argv[1]);
    pipe(fd);
    pid=fork();
    if (pid==0){ /* figlio */
        signal(SIGPIPE, fine);
        close(fd[0]);
        for(;;)
            write(fd[1], messaggio, DIM);
    }
```

Esempio (3)

```
else if (pid>0){
    /* padre */
    signal(SIGALRM, timeout);
    close(fd[1]);
    alarm(N);
    for(;;) {
        read(fd[0], messaggio, DIM);
        write(1, messaggio, DIM);
    }
} /* fine else if */
}/* fine main */
```

Esempio (4)

```
/* definizione degli handler dei segnali */
void timeout(int signo){
    int stato;
    close(fd[0]); /* chiusura effettiva del lato di lettura*/
    wait(&stato);
    if ((char)stato!=0)
        printf("Termin invol figlio (segnale %d)\n",
               (char)stato);
    else printf("Termin volont Figlio (stato %d)\n",
               stato>>8);

    exit(0);
}

void fine(int signo){
    close(fd[1]);
    exit(0);
}
```


System call dup

Per *duplicare un elemento della tabella dei file aperti di processo*:

```
int dup(int fd)
```

- **fd** è il file descriptor del file da duplicare
- L'effetto di **dup ()** è copiare l'elemento **fd** della tabella dei file aperti nella **prima posizione libera** (quella con l'indice minimo tra quelle disponibili)
- Restituisce il **nuovo file descriptor** (del file aperto copiato), oppure -1 (in caso di errore)

Stdin, stdout, stderr

Per convenzione, per ogni processo vengono **aperti automaticamente 3 descrittori di file** associati ai primi tre elementi della tabella

- stdin (fd 0) → tastiera
- stdout (fd 1) → video
- stderr (fd 2) → video

Esempio: mediante dup () ridirigere stdout su pipe

```
int main() {
    int pid, fd[2]; char msg[3]="bye";
    pipe(fd);
    pid=fork();
    if (!pid){ /* processo figlio */
        close(fd[0]); close(1);
        dup(fd[1]); /* ridirigo stdout sulla pipe */
        close(fd[1]);
        write(1,msg, sizeof(msg)); /*scrivo su pipe*/
        close(1);
    }else { /* processo padre */
        close(fd[1]);
        read(fd[0], msg, 3);
        close(fd[0]);
    } /* fine else */
} /* fine main */
```

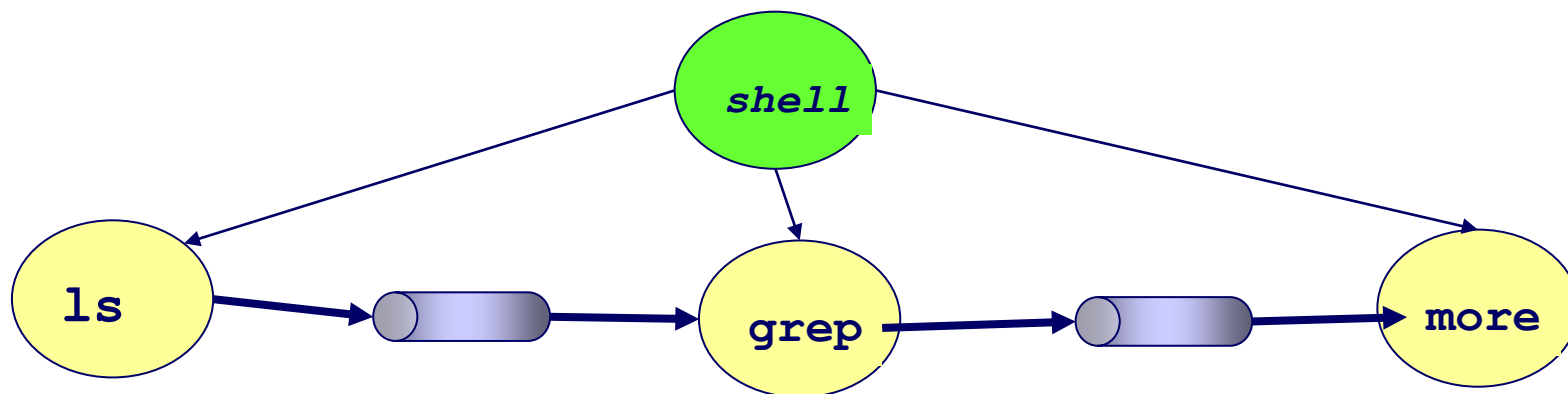
dup () & piping

Tramite **dup ()** si può realizzare il piping di comandi.

Ad esempio:

```
ls -lR | grep Jun | more
```

- Vengono creati 3 processi (uno per ogni comando), in modo che:
 - stdout di **ls** sia ridiretto nello stdin di **grep**
 - stdout di **grep** sia ridiretto nello stdin di **more**



Esempio: piping di 2 comandi senza argomenti

```
/* sintassi: programma com1 com2  significa:  
   com1 | com2  */
```

```
int main(int argc, char **argv){  
    int pid1, pid2, fd[2], i, status;  
    pipe(fd);  
    pid1=fork();  
    if (!pid1){ /* primo processo figlio: com2 */  
        close(fd[1]);  
        close(0);  
        dup(fd[0]); /* ridirigo stdin sulla pipe */  
        close(fd[0]);  
        execlp(argv[2], argv[2], (char *)0);  
        exit(-1);  
    }
```

```

else{ /* processo padre */
    pid2=fork() ;
    if (!pid2){ /* secondo figlio: com1 */
        close(fd[0]) ;
        close(1) ;
        dup(fd[1]) ;
        close(fd[1]) ;
        execlp(argv[1], argv[1], (char *)0) ;
        exit(-1) ;
    }
    for (i=0; i<2;i++){
        wait(&status) ;
        if((char)status!=0)
            printf("figlio terminato per segnale%d\n",
                (char)status) ;
    }
    exit(0) ;
} /* fine main */

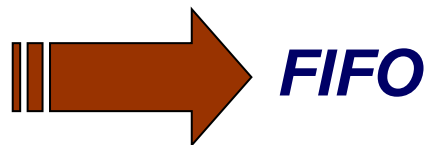
```

Pipe: possibili svantaggi

Il meccanismo delle pipe ha **due svantaggi**:

- consente la comunicazione **solo tra processi in relazione di parentela**
- **non è persistente**: pipe viene distrutta quando terminano tutti i processi che hanno accesso ai suoi estremi

Per realizzare la comunicazione persistente tra una coppia di **processi non appartenenti alla stessa gerarchia**?



fifo

È una **pipe con nome** nel file system:

- Esattamente come le pipe normali, canale **unidirezionale** del tipo **first-in-first-out**
- è **rappresentata da un file** nel file system: **persistenza**, **visibilità** potenzialmente globale
- ha un proprietario, un insieme di diritti e una lunghezza
- è creata dalla system call **mkfifo()**
- è aperta e acceduta con le stesse system call dei file

Per creare una fifo (pipe con nome):

```
int mkfifo(char* pathname, int mode);
```

- **pathname** è il nome della fifo
- **mode** esprime i permessi

restituisce 0, in caso di successo, un valore negativo, in caso contrario

Apertura/chiusura di fifo

Una volta creata, ***fifo può essere aperta*** (come tutti i file) mediante `open()`. Ad esempio, un processo destinatario di messaggi:

```
int fd;  
fd=open("myfifo", O_RDONLY);
```

Per chiudere una fifo, si usa `close()`:

```
close(fd);
```

Per eliminare una fifo, si usa `unlink()`:

```
unlink("myfifo");
```

Accesso a fifo

Una volta aperta, ***fifo può essere acceduta*** (come tutti i file) mediante `read()` / `write()`. Ad esempio, un processo destinatario di messaggi:

```
int fd;  
char msg[10];  
fd=open("myfifo", O_RDONLY);  
read(fd, msg, 10);
```


Slide aggiuntive sulla tabella dei file/pipe/socket descriptor

- Nelle seguenti slide alcune anticipazioni per meglio comprendere i meccanismi di redirectione e piping

Tabella file/pipe/socket descriptor

- Tabella associata ad ogni processo utente e costituita da un elemento (riga) per ogni file aperto dal processo
- Indice della tabella = descrittore del file (**fd**)
- Per convenzione, per ogni processo vengono aperti automaticamente **3 descrittori di file** associati ai primi tre elementi della tabella **stdin (0)**, **stdout (1)**, **stderr (2)** associati rispettivamente alla tastiera (0) e al video (1 e 2)
- Ogni entry (riga) della tabella contiene un puntatore o indice della riga della tabella globale dei file aperti relativa al file

0	stdin
1	stdout
2	stderr
fd1 3	...
...	...



Un processo apre due file

Tabella dei descrittori di
file aperti per processo

P0

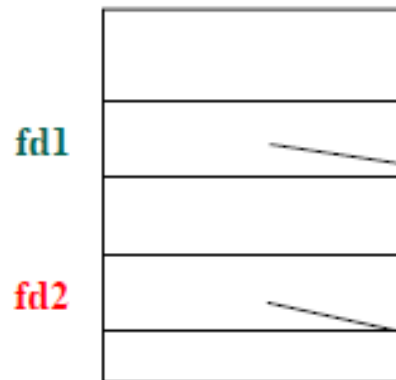


Tabella globale
dei file aperti

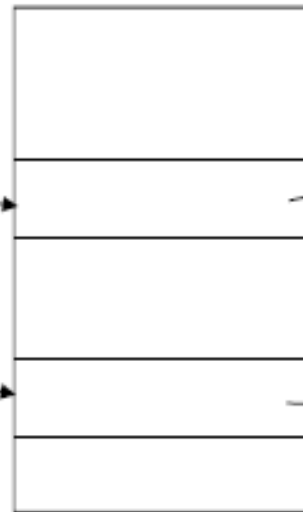
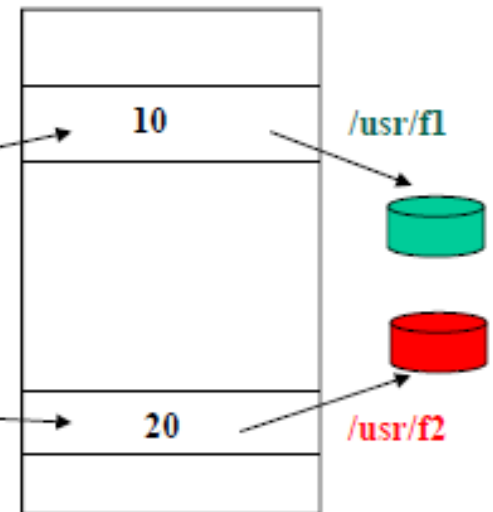


Tabella degli i-node



Processo P0:

```
fd1 = open("/usr/f1");
```

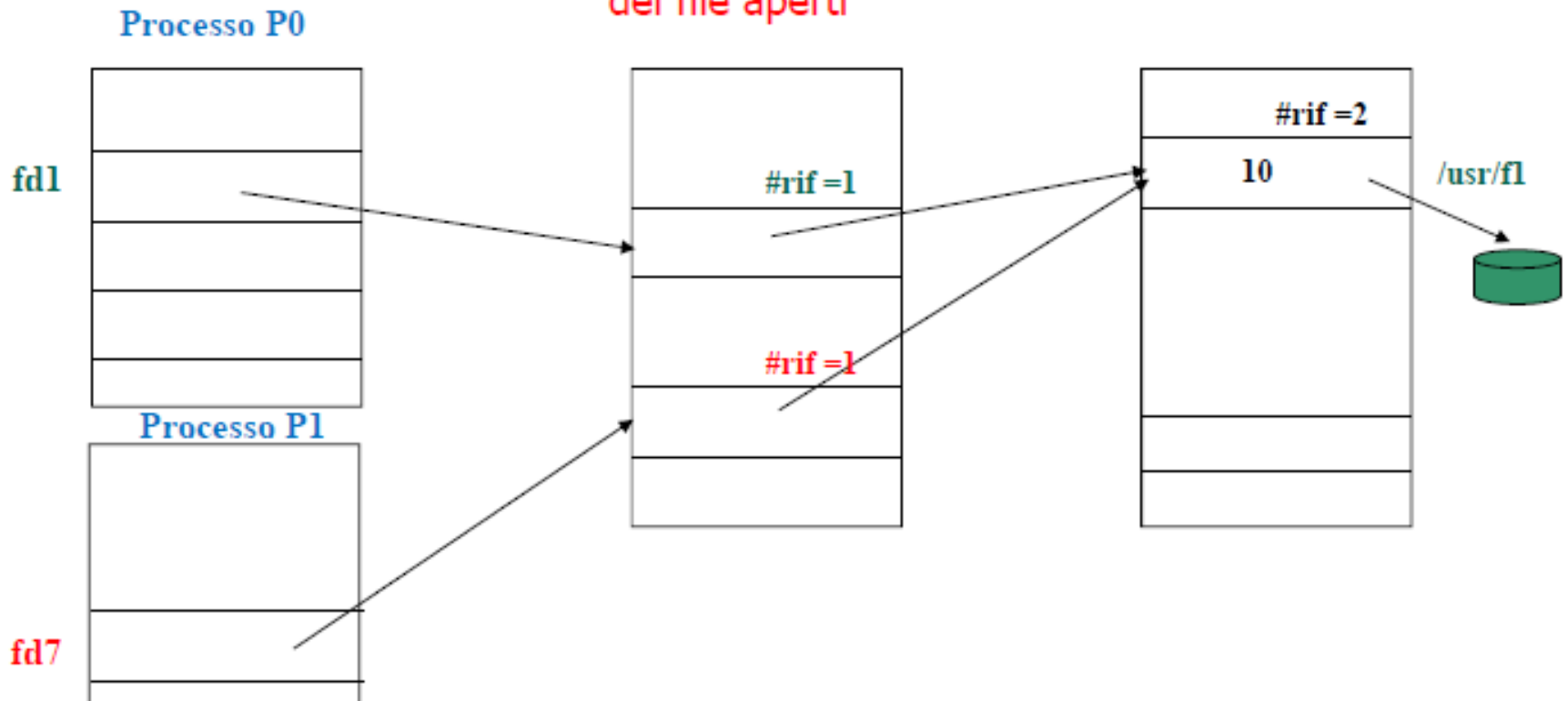
```
fd2 = open("/usr/f2");
```

Due processi aprono stesso file

Tabella dei descrittori di file aperti per processo

Tabella globale dei file aperti

Tabella degli i-node



Un processo duplica un file/pipe/socket descriptor

Tabella dei descrittori di file aperti per processo

Processo P0


fd1	
fd3	

Tabella globale dei file aperti

#rif = 2	

Tabella degli i-node

#rif = 1	
10	/usr/fl



int dup(int oldfd);

The dup() system call creates a copy of the file descriptor oldfd, using the **lowest-numbered unused file descriptor** for the new descriptor.

Fork di un processo

