Università di Udine — Facoltà di Scienze MM.FF.NN.

Laurea in Informatica — A.A. 2003/04

Trasparenze del Corso di *Sistemi Operativi*

Marino Miculan

Università di Udine

Cooperazione tra Processi

- Principi
- Il problema della sezione critica: le *race condition*
- Supporto hardware
- Semafori
- Monitor
- Scambio di messaggi
- Barriere
- Problemi classici di sincronizzazione

Processi (e Thread) Cooperanti

- Processi *indipendenti* non possono modificare o essere modificati dall'esecuzione di un altro processo.
- I processi *cooperanti* possono modificare o essere modificati dall'esecuzione di altri processi.
- Vantaggi della cooperazione tra processi:
  - Condivisione delle informazioni
  - Aumento della computazione (parallelismo)
  - Modularità
  - Praticità implementativa/di utilizzo

IPC: InterProcess Communication

Meccanismi di comunicazione e interazione tra processi (e thread)

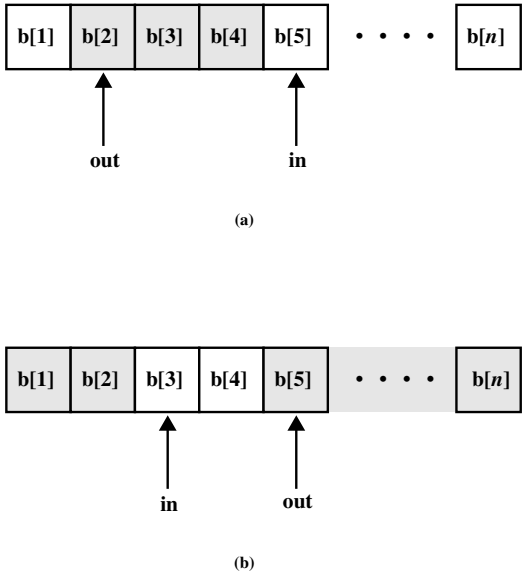
Questioni da considerare:

- Come può un processo passare informazioni ad un altro?
- Come evitare accessi inconsistenti a risorse condivise?
- Come sequenzializzare gli accessi alle risorse secondo la causalità?

Mantenere la consistenza dei dati richiede dei meccanismi per assicurare l'esecuzione ordinata dei processi cooperanti.

Esempio: Problema del produttore-consumatore

- Tipico paradigma dei processi cooperanti: il processo *produttore* produce informazione che viene consumata da un processo *consumatore*
- Soluzione a memoria condivisa: tra i due processi si pone un buffer di comunicazione di dimensione fissata.



Produttore-consumatore con buffer limitato

- Dati condivisi tra i processi

type *item* = ... ;
var *buffer*: **array** [0..*n*-1] **of** *item*;
in, *out*: 0..*n*-1;
counter: 0..*n*;
in, *out*, *counter* := 0;

Processo produttore

repeat
  ...
  produce un item in *nextp*
  ...
  while *counter* = *n* **do** *no-op*;
  buffer[*in*] := *nextp*;
  in := *in* + 1 **mod** *n*;
  counter := *counter* + 1;
until *false*;

Processo consumatore

repeat
  while *counter* = 0 **do** *no-op*;
  nextc := *buffer*[*out*];
  out := *out* + 1 **mod** *n*;
  counter := *counter* − 1;
  ...
  consuma l'item in *nextc*
  ...
until *false*;

- Le istruzioni
  - *counter* := *counter* + 1;
  - *counter* := *counter* − 1;

  devono essere eseguite *atomicamente*: se eseguite in parallelo non atomicamente, possono portare ad inconsistenze.

Race conditions

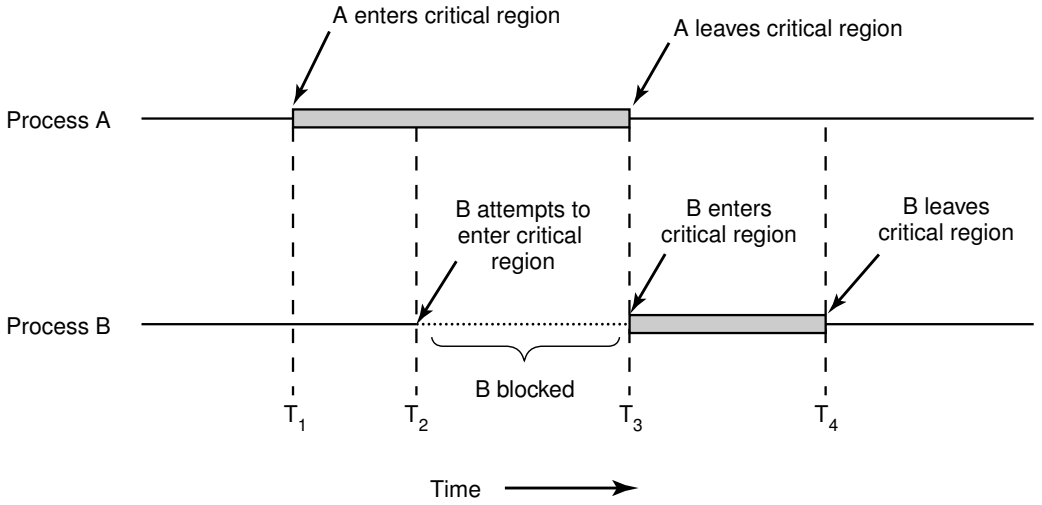
Race condition: più processi accedono concorrentemente agli stessi dati, e il risultato dipende dall'ordine di interleaving dei processi.

- Frequenti nei sistemi operativi multitasking, sia per dati in user space sia per strutture in kernel.
- Estremamente pericolose: portano al malfunzionamento dei processi cooperanti, o anche (nel caso delle strutture in kernel space) dell'intero sistema
- difficili da individuare e riprodurre: dipendono da informazioni astratte dai processi (decisioni dello scheduler, carico del sistema, utilizzo della memoria, numero di processori, . . .)

Problema della Sezione Critica

- n processi che competono per usare dati condivisi
- Ogni processo ha un segmento di codice, detto *sezione critica* in cui si accede ai dati condivisi.
- Problema: assicurare che quando un processo esegue la sua sezione critica, nessun altro processo possa entrare nella propria sezione critica.
- Bisogna proteggere la sezione critica con apposito *codice di controllo*

```
while (TRUE) {
    entry section
        sezione critica
    exit section
        sezione non critica
};
```

Criteri per una Soluzione del Problema della Sezione Critica

1. **Mutua esclusione:** se il processo P_i sta eseguendo la sua sezione critica, allora nessun altro processo può eseguire la propria sezione critica.
2. **Progresso:** se nessun processo è nella sezione critica e esiste un processo che desidera entrare nella propria sezione critica, allora l'esecuzione di tale processo non può essere posposta indefinitamente.
3. **Attesa limitata:** se un processo P ha richiesto di entrare nella propria sezione critica, allora il numero di volte che si concede agli altri processi di accedere alla propria sezione critica prima del processo P deve essere limitato.

- Si suppone che ogni processo venga eseguito ad una velocità non nulla.
- Non si suppone niente sulla velocità *relativa* dei processi (e quindi sul numero e tipo di CPU)

Soluzioni hardware: controllo degli interrupt

- Il processo può disabilitare TUTTI gli interrupt hw all'ingresso della sezione critica, e riabilitarli all'uscita
  - Soluzione semplice; garantisce la mutua esclusione
  - ma pericolosa: il processo può non riabilitare più gli interrupt, acquisendosi la macchina
  - può allungare di molto i tempi di latenza
  - non scala a macchine multiprocessore (a meno di non bloccare tutte le altre CPU)
- Inadatto come meccanismo di mutua esclusione tra processi utente
- Adatto per brevi(ssimi) segmenti di codice affidabile (es: in kernel, quando si accede a strutture condivise)

Soluzioni software

- Supponiamo che ci siano solo 2 processi, P_0 e P_1
- Struttura del processo P_i (l'altro sia P_j)

```
while (TRUE) {
    entry section
        sezione critica
    exit section
        sezione non critica
}
```

- Supponiamo che i processi possano condividere alcune variabili (dette *di lock*) per sincronizzare le proprie azioni

Tentativo sbagliato

- Variabili condivise
  - **var** *occupato*: (0..1);
    inizialmente *occupato* = 0
  - *occupato* = 0 $\Rightarrow$ un processo può entrare nella propria sezione critica
- Processo P_i

```
while (TRUE) {                    ⇓
    while (occupato ≠ 0);   occupato := 1;
        sezione critica
    occupato := 0;
        sezione non critica
};
```

- Non funziona: lo scheduler può agire dopo il ciclo, nel punto indicato.

Alternanza stretta

- Variabili condivise
  - **var** *turn*: (0..1);
    inizialmente *turn* = 0
  - *turn* = $i \Rightarrow P_i$ può entrare nella propria sezione critica
- Processo P_i

```
while (TRUE) {
    while (turn ≠ i) no-op;
        sezione critica
    turn := j;
        sezione non critica
};
```


Alternanza stretta (cont.)

- Soddisfa il requisito di mutua esclusione, ma non di progresso (richiede l'alternanza stretta) $\Rightarrow$ inadatto per processi con differenze di velocità
- È un esempio di *busy wait*: attesa *attiva* di un evento (es: testare il valore di una variabile).
  - Semplice da implementare
  - Porta a consumi inaccettabili di CPU
  - In genere, da evitare, ma a volte è preferibile (es. in caso di attese molto brevi)
- Un processo che attende attivamente su una variabile esegue uno *spin lock*.

Algoritmo di Peterson

```
#define FALSE  0
#define TRUE   1
#define N      2                    /* number of processes */

int turn;                           /* whose turn is it? */
int interested[N];                  /* all values initially 0 (FALSE) */

void enter_region(int process);     /* process is 0 or 1 */
{
    int other;                      /* number of the other process */

    other = 1 – process;            /* the opposite of process */
    interested[process] = TRUE;     /* show that you are interested */
    turn = process;                 /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)      /* process: who is leaving */
{
    interested[process] = FALSE;    /* indicate departure from critical region */
}
```


Algoritmo di Peterson (cont)

- Basato su una combinazione di *richiesta* e *accesso*
- Soddisfa tutti i requisiti; risolve il problema della sezione critica per 2 processi
- Si può generalizzare a N processi
- È ancora basato su spinlock

Algoritmo del Fornaio

Risolve la sezione critica per n processi, generalizzando l'idea vista precedentemente.

- Prima di entrare nella sezione critica, ogni processo riceve un numero. Chi ha il numero più basso entra nella sezione critica.
- Se i processi P_i and P_j ricevono lo stesso numero: se $i < j$, allora P_i è servito per primo; altrimenti P_j è servito per primo.
- Lo schema di numerazione genera numeri in ordine crescente, i.e., 1,2,3,3,3,3,4

Istruzioni di Test&Set

- Istruzioni di Test-and-Set-Lock: testano e modificano il contenuto di una parola atomicamente

```
function Test-and-Set (var target: boolean): boolean;
    begin
        Test-and-Set := target;
        target := true;
    end;
```

- Questi due passi devono essere implementati come atomici in assembler. (Es: le istruzioni BTC, BTR, BTS su Intel). Ipoteticamente:

```
TSL RX,LOCK
```

Copia il contenuto della cella `LOCK` nel registro `RX`, e poi imposta la cella `LOCK` ad un valore $\neq 0$. Il tutto atomicamente (viene bloccato il bus di memoria).

Istruzioni di Test&Set (cont.)

```
enter_region:
    TSL REGISTER,LOCK          | copy lock to register and set lock to 1
    CMP REGISTER,#0            | was lock zero?
    JNE enter_region           | if it was non zero, lock was set, so loop
    RET | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0               | store a 0 in lock
    RET | return to caller
```

- corretto e semplice
- è uno spinlock — quindi busy wait
- Problematico per macchine parallele

Evitare il busy wait

- Le soluzioni basate su spinlock portano a
  - busy wait: alto consumo di CPU
  - inversione di priorità: un processo a bassa priorità che blocca una risorsa viene ostacolato nella sua esecuzione da un processo ad alta priorità in busy wait sulla stessa risorsa.
- Idea migliore: quando un processo deve attendere un evento, che venga posto in *wait*; quando l'evento avviene, che venga posto in *ready*
- Servono specifiche syscall o funzioni di kernel. Esempio:
  - *sleep()*: il processo si autosospende (si mette in *wait*)
  - *wakeup(pid)*: il processo *pid* viene posto in *ready*, se era in *wait*.

  Ci sono molte varianti. Molto comune: con *evento* esplicito.

Produttore-consumatore con sleep e wakeup

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item();                /* generate next item */
          if (count == N) sleep();              /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep();              /* if buffer is empty, got to sleep */
          item = remove_item();                 /* take item out of buffer */
          count = count - 1;                    /* decrement count of items in buffer */
          if (count == N - 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```


Produttore-consumatore con sleep e wakeup (cont.)

- Risolve il problema del busy wait
- Non risolve la corsa critica sulla variabile *count*
- I segnali possono andare *perduti,* con conseguenti *deadlock*
- Soluzione: salvare i segnali "in attesa" in un contatore

Semafori

Strumento di sincronizzazione generale (Dijkstra '65)

- Semaforo *S*: variabile intera.
- Vi si può accedere solo attraverso 2 operazioni **atomiche:**
  - *up(S)*: incrementa *S*
  - *down(S)*: attendi finché *S* è maggiore di 0; quindi decrementa *S*
- Normalmente, l'attesa è implementata spostando il processo in stato di *wait*, mentre la *up(S)* mette uno dei processi eventualmente in attesa nello stato di *ready*.
- I nomi originali erano P (*proberen*, testare) e V (*verhogen*, incrementare)

Esempio: Sezione Critica per n processi

- Variabili condivise:
  - **var** *mutex* : *semaphore*
  - inizialmente *mutex* = 1
- Processo P_i

```
while (TRUE) {
    down(mutex);
        sezione critica
    up(mutex);
        sezione non critica
}
```


Esempio: Produttore-Consumatore con semafori

```
#define N 100                       /* number of slots in the buffer */
typedef int semaphore;              /* semaphores are a special kind of int */
semaphore mutex = 1;                /* controls access to critical region */
semaphore empty = N;                /* counts empty buffer slots */
semaphore full = 0;                 /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {                  /* TRUE is the constant 1 */
        item = produce_item();      /* generate something to put in buffer */
        down(&empty);               /* decrement empty count */
        down(&mutex);               /* enter critical region */
        insert_item(item);          /* put new item in buffer */
        up(&mutex);                 /* leave critical region */
        up(&full);                  /* increment count of full slots */
    }
}
```



```
void consumer(void)
{
    int item;

    while (TRUE) {                  /* infinite loop */
        down(&full);                /* decrement full count */
        down(&mutex);               /* enter critical region */
        item = remove_item();       /* take item from buffer */
        up(&mutex);                 /* leave critical region */
        up(&empty);                 /* increment count of empty slots */
        consume_item(item);         /* do something with the item */
    }
}
```

Esempio: Sincronizzazione tra due processi

- Variabili condivise:
  - **var** *sync* : *semaphore*
  - inizialmente *sync* = 0
- Processo P_1 Processo P_2

⋮	⋮
S_1;	down(sync);
up(sync);	S_2;
⋮	⋮

- S_2 viene eseguito solo dopo S_1.

Implementazione dei semafori

- La definizione classica usava uno *spinlock* per la *down:* facile implementazione (specialmente su macchine parallele), ma inefficiente
- Alternativa: il processo in attesa viene messo in stato di *wait*
- In generale, un semaforo è un record

```
type semaphore = record
                     value: integer;
                     L: list of process;
                 end;
```

- Assumiamo due operazioni fornite dal sistema operativo:
  - *sleep()*: sospende il processo che la chiama (rilascia la CPU)
  - *wakeup*(*P*): pone in stato di *ready* il processo *P*.

Implementazione dei semafori (Cont.)

- Le operazioni sui semafori sono definite come segue:

```
down(S): S.value := S.value − 1;
         if S.value < 0
           then begin
                    aggiungi questo processo a S.L;
                    sleep();
                end;
up(S):   S.value := S.value + 1;
         if S.value ≤ 0
           then begin
                    togli un processo P da S.L;
                    wakeup(P);
                end;
```


Implementazione dei semafori (Cont.)

- *value* può avere valori negativi: indica quanti processi sono in attesa su quel semaforo
- le due operazioni *wait* e *signal* devono essere *atomiche* fino a prima della *sleep* e *wakeup:* problema di sezione critica, da risolvere come visto prima:
  - disabilitazione degli interrupt: semplice, ma inadatto a sistemi con molti processori
  - uso di istruzioni speciali (test-and-set)
  - ciclo busy-wait (spinlock): generale, e sufficientemente efficiente (le due sezioni critiche sono molto brevi)

Mutex

- I mutex sono semafori con due soli possibili valori: *bloccato* o *non bloccato*
- Utili per implementare mutua esclusione, sincronizzazione, . . .
- due primitive: *mutex_lock* e *mutex_unlock*.
- Semplici da implementare, anche in user space (p.e. per thread). Esempio:

```
mutex_lock:
    TSL REGISTER,MUTEX          | copy mutex to register and set mutex to 1
    CMP REGISTER,#0             | was mutex zero?
    JZE ok                      | if it was zero, mutex was unlocked, so return
    CALL thread_yield           | mutex is busy; schedule another thread
    JMP mutex_lock              | try again later
ok: RET | return to caller; critical region entered

mutex_unlock:
    MOVE MUTEX,#0               | store a 0 in mutex
    RET | return to caller
```


Memoria condivisa?

Implementare queste funzioni richiede una qualche memoria condivisa.

- A livello kernel: strutture come quelle usate dai semafori possono essere mantenute nel kernel, e quindi accessibili da tutti i processi (via le apposite system call)
- A livello utente:
  - all'interno dello stesso processo: adatto per i thread
  - tra processi diversi: spesso i S.O. offrono la possibilità di condividere segmenti di memoria tra processi diversi (*shared memory*)
  - alla peggio: file su disco

Deadlock con Semafori

- **Deadlock (stallo):** due o più processi sono in attesa indefinita di eventi che possono essere causati solo dai processi stessi in attesa.
- L'uso dei semafori può portare a deadlock. Esempio: siano S e Q due semafori inizializzati a 1

P_0	P_1
down(S);	*down*(Q);
down(Q);	*down*(S);
⋮	⋮
up(S);	*up*(Q);
up(Q);	*up*(S);

- Programmare con i semafori è molto delicato e prono ad errori, difficilissimi da debuggare. Come in assembler, solo peggio, perché qui gli errori sono race condition e malfunzionamenti non riproducibili.

Monitor

- Un *monitor* è un tipo di dato astratto che fornisce funzionalità di mutua esclusione
  - collezione di dati privati e funzioni/procedure per accedervi.
  - i processi possono chiamare le procedure ma non accedere alle variabili locali.
  - *un solo* processo alla volta può eseguire codice di un monitor
- Il programmatore raccoglie quindi i dati condivisi *e tutte le sezioni critiche relative* in un monitor; questo risolve il problema della mutua esclusione
- Implementati dal compilatore con dei costrutti per mutua esclusione (p.e.: inserisce automaticamente `lock_mutex` e `unlock_mutex` all'inizio e fine di ogni procedura)

monitor *example*
    integer *i*;
    condition *c*;

    procedure *producer*( );
    .
    .
    .
    end;

    procedure *consumer*( );
    .
    .
    .
    end;
end monitor;

Monitor: Controllo del flusso di controllo

Per sospendere e riprendere i processi, ci sono le variabili *condition*, simili agli eventi, con le operazioni

- *wait(c):* il processo che la esegue si blocca sulla condizione c.
- *signal(c):* uno dei processi in attesa su c viene risvegliato.

  A questo punto, chi va in esecuzione nel monitor? Due varianti:

  - chi esegue la *signal(c)* si sospende automaticamente (*monitor di Hoare*)
  - la *signal(c)* deve essere l'ultima istruzione di una procedura (così il processo lascia il monitor) (*monitor di Brinch-Hansen*)
  - i processi risvegliati possono provare ad entrare nel monitor solo dopo che il processo attuale lo ha lasciato

  Il successivo processo ad entrare viene scelto dallo scheduler di sistema

- i *signal* su una condizione senza processi in attesa vengono persi

Produttore-consumatore con monitor

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count – 1;
        if count = N – 1 then signal(full)
    end;
    count := 0;
end monitor;
```

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```


Monitor (cont.)

- I monitor semplificano molto la gestione della mutua esclusione (meno possibilità di errori)
- Veri costrutti, non funzioni di libreria $\Rightarrow$ bisogna modificare i compilatori.
- Implementati (in certe misure) in veri linguaggi. Esempio: i metodi `synchronize` di Java.
  - solo un metodo `synchronized` di una classe può essere eseguito alla volta.
  - Java non ha variabili *condition*, ma ha *wait* and *notify* (+ o − come *sleep* e *wakeup*).
- Un problema che rimane (sia con i monitor che con i semafori): è necessario avere *memoria condivisa* $\Rightarrow$ questi costrutti non sono applicabili a sistemi distribuiti (reti di calcolatori) senza memoria fisica condivisa.

Passaggio di messaggi

- Comunicazione non basato su memoria condivisa con controllo di accesso.
- Basato su due primitive (chiamate di sistema o funzioni di libreria)
  - `send(destinazione, messaggio)`: spedisce un messaggio ad una certa destinazione; solitamente non bloccante.
  - `receive(sorgente, &messaggio)`: riceve un messaggio da una sorgente; solitamente bloccante (fino a che il messaggio non è disponibile).
- Meccanismo più astratto e generale della memoria condivisa e semafori
- Si presta ad una implementazione su macchine distribuite

Problematiche dello scambio di messaggi

- Affidabilità: i canali possono essere inaffidabili (es: reti). Bisogna implementare appositi protocolli fault-tolerant (basati su acknowledgment e timestamping).
- Autenticazione: come autenticare i due partner?
- Sicurezza: i canali utilizzati possono essere intercettati
- Efficienza: se prende luogo sulla stessa macchina, il passaggio di messaggi è sempre più lento della memoria condivisa e semafori.

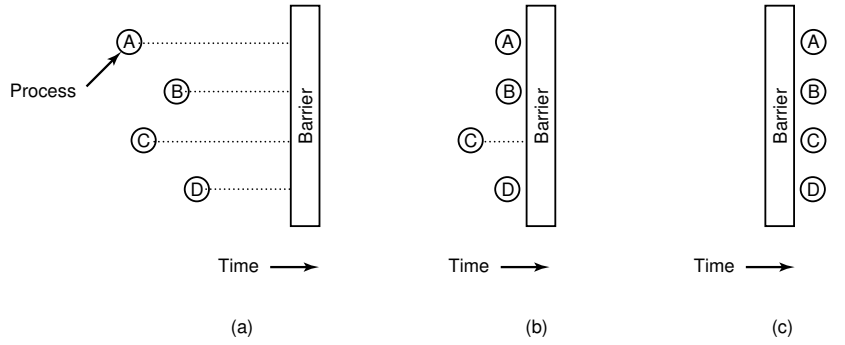
Produttore-consumatore con scambio di messaggi

- Comunicazione *asincrona*
  - I messaggi spediti ma non ancora consumati vengono automaticamente bufferizzati in una *mailbox* (mantenuto in kernel o dalle librerie)
  - L'oggetto delle `send` e `receive` sono le mailbox
  - La `send` si blocca se la mailbox è piena; la `receive` si blocca se la mailbox è vuota.
- Comunicazione *sincrona*
  - I messaggi vengono spediti direttamente al processo destinazione
  - L'oggetto delle `send` e `receive` sono i processi
  - Le `send` e `receive` si bloccano fino a che la controparte non esegue la chiamata duale (*rendez-vous*).


```
#define N 100                           /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                         /* message buffer */

     while (TRUE) {
          item = produce_item( );       /* generate something to put in buffer */
          receive(consumer, &m);        /* wait for an empty to arrive */
          build_message(&m, item);      /* construct a message to send */
          send(consumer, &m);           /* send item to consumer */
     }
}

void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
     while (TRUE) {
          receive(producer, &m);        /* get message containing item */
          item = extract_item(&m);      /* extract item from message */
          send(producer, &m);           /* send back empty reply */
          consume_item(item);           /* do something with the item */
     }
}
```

Barriere

- Meccanismo di sincronizzazione per *gruppi* di processi, specialmente per calcolo parallelo a memoria condivisa (es. SMP, NUMA)
  - Ogni processo alla fine della sua computazione, chiama la funzione `barrier` e si sospende.
  - Quando tutti i processi hanno raggiunto la barriera, la superano *tutti assieme* (si sbloccano).



I Grandi Classici

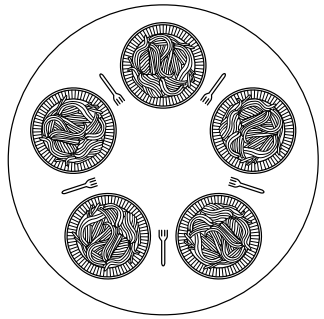
Esempi paradigmatici di programmazione concorrente. Presi come testbed per ogni primitiva di programmazione e comunicazione.

(E buoni esempi didattici!)

- Produttore-Consumatore a buffer limitato (già visto)
- I Filosofi a Cena
- Lettori-Scrittori
- Il Barbiere che Dorme

I Classici: I Filosofi a Cena (Dijkstra, 1965)

n filosofi seduti attorno ad un tavolo rotondo con n piatti di spaghetti e n forchette (bastoncini). (nell'esempio, $n = 5$)



- Mentre pensa, un filosofo non interagisce con nessuno
- Quando gli viene fame, cerca di prendere le bacchette più vicine, una alla volta.
- Quando ha due bacchette, un filosofo mangia senza fermarsi.
- Terminato il pasto, lascia le bacchette e torna a pensare.

Problema: programmare i filosofi in modo da garantire

- assenza di deadlock: non si verificano mai blocchi
- assenza di starvation: un filosofo che vuole mangiare, prima o poi mangia.

I Filosofi a Cena—Una non-soluzione

```
#define N 5                          /* number of philosophers */

void philosopher(int i)              /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think();                     /* philosopher is thinking */
        take_fork(i);                /* take left fork */
        take_fork((i+1) % N);        /* take right fork; % is modulo operator */
        eat();                       /* yum-yum, spaghetti */
        put_fork(i);                 /* put left fork back on the table */
        put_fork((i+1) % N);         /* put right fork back on the table */
    }
}
```

Possibilità di deadlock: se tutti i processi prendono contemporaneamente la forchetta alla loro sinistra. . .

I Filosofi a Cena—Tentativi di correzione

- Come prima, ma controllare se la forchetta dx è disponibile prima di prelevarla, altrimenti rilasciare la forchetta sx e riprovare daccapo.
  - Non c'è deadlock, ma possibilità di starvation.
- Come sopra, ma introdurre un ritardo casuale prima della ripetizione del tentativo.
  - Non c'è deadlock, la possibilità di starvation viene ridotta ma non azzerata. Applicato in molti protocolli di accesso (CSMA/CD, es. Ethernet). Inadatto in situazione mission-critical o real-time.

I Filosofi a Cena—Soluzioni

- Introdurre un semaforo `mutex` per proteggere la sezione critica (dalla prima `take_fork` all'ultima `put_fork`):
  - Funziona, ma solo un filosofo per volta può mangiare, mentre in teoria $\lfloor n/2 \rfloor$ possono mangiare contemporaneamente.
- Tenere traccia dell'*intenzione* di un filosofo di mangiare. Un filosofo ha tre stati (THINKING, HUNGRY, EATING), mantenuto in un vettore `state`. Un filosofo può entrare nello stato EATING solo è HUNGRY e i vicini non sono EATING.
  - Funziona, e consente il massimo parallelismo.


```
#define N              5              /* number of philosophers */
#define LEFT           (i+N–1)%N      /* number of i's left neighbor */
#define RIGHT          (i+1)%N        /* number of i's right neighbor */
#define THINKING       0              /* philosopher is thinking */
#define HUNGRY         1              /* philosopher is trying to get forks */
#define EATING         2              /* philosopher is eating */
typedef int semaphore;                /* semaphores are a special kind of int */
int state[N];                         /* array to keep track of everyone's state */
semaphore mutex = 1;                  /* mutual exclusion for critical regions */
semaphore s[N];                       /* one semaphore per philosopher */

void philosopher(int i)               /* i: philosopher number, from 0 to N–1 */
{
    while (TRUE) {                    /* repeat forever */
        think();                      /* philosopher is thinking */
        take_forks(i);                /* acquire two forks or block */
        eat();                        /* yum-yum, spaghetti */
        put_forks(i);                 /* put both forks back on table */
    }
}
```

```
void take_forks(int i)                /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);                     /* enter critical region */
    state[i] = HUNGRY;                /* record fact that philosopher i is hungry */
    test(i);                          /* try to acquire 2 forks */
    up(&mutex);                       /* exit critical region */
    down(&s[i]);                      /* block if forks were not acquired */
}

void put_forks(i)                     /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);                     /* enter critical region */
    state[i] = THINKING;              /* philosopher has finished eating */
    test(LEFT);                       /* see if left neighbor can now eat */
    test(RIGHT);                      /* see if right neighbor can now eat */
    up(&mutex);                       /* exit critical region */
}

void test(i)                          /* i: philosopher number, from 0 to N–1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

I Classici: Lettori-Scrittori

Un insieme di dati (es. un file, un database, dei record), deve essere condiviso da processi *lettori* e *scrittori*

- Due o più lettori possono accedere contemporaneamente ai dati
- Ogni scrittore deve accedere ai dati in modo esclusivo.

Implementazione con i semafori:

- Tenere conto dei lettori in una variabile condivisa, e fino a che ci sono lettori, gli scrittori non possono accedere.
- Dà maggiore priorità ai lettori che agli scrittori.

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to 'rc' */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to 'rc' */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to 'rc' */
        read_data_base();               /* access the data */
        down(&mutex);                   /* get exclusive access to 'rc' */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to 'rc' */
        use_data_read();                /* noncritical region */
    }
}


void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data();                /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base();              /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

I Classici: Il Barbiere che Dorme

In un negozio c'è un solo barbiere, una sedia da barbiere e n sedie per l'attesa.

- Quando non ci sono clienti, il barbiere dorme sulla sedia.
- Quando arriva un cliente, questo sveglia il barbiere se sta dormendo.
- Se la sedia è libera e ci sono clienti, il barbiere fa sedere un cliente e lo serve.
- Se un cliente arriva e il barbiere sta già servendo un cliente, si siede su una sedia di attesa se ce ne sono di libere, altrimenti se ne va.

Problema: programmare il barbiere e i clienti filosofi in modo da garantire assenza di deadlock e di starvation.

Il Barbiere—Soluzione

- Tre semafori:
  - `customers`: i clienti in attesa (contati anche da una variabile `waiting`)
  - `barbers`: conta i barbieri in attesa
  - `mutex`: per mutua esclusione
- Ogni barbiere (uno) esegue una procedura che lo blocca se non ci sono clienti; quando si sveglia, serve un cliente e ripete.
- Ogni cliente prima di entrare nel negozio controlla se ci sono sedie libere; altrimenti se ne va.
- Un cliente, quando entra nel negozio, sveglia il barbiere se sta dormendo.


```
#define CHAIRS 5                        /* # chairs for waiting customers */

typedef int semaphore;                  /* use your imagination */

semaphore customers = 0;                /* # of customers waiting for service */
semaphore barbers = 0;                  /* # of barbers waiting for customers */
semaphore mutex = 1;                    /* for mutual exclusion */
int waiting = 0;                        /* customers are waiting (not being cut) */

void barber(void)
{
    while (TRUE) {
        down(&customers);               /* go to sleep if # of customers is 0 */
        down(&mutex);                   /* acquire access to 'waiting' */
        waiting = waiting – 1;          /* decrement count of waiting customers */
        up(&barbers);                   /* one barber is now ready to cut hair */
        up(&mutex);                     /* release 'waiting' */
        cut_hair();                     /* cut hair (outside critical region) */
    }
}
```

```
void customer(void)
{
     down(&mutex);                  /* enter critical region */
     if (waiting < CHAIRS) {        /* if there are no free chairs, leave */
          waiting = waiting + 1;    /* increment count of waiting customers */
          up(&customers);           /* wake up barber if necessary */
          up(&mutex);               /* release access to 'waiting' */
          down(&barbers);           /* go to sleep if # of free barbers is 0 */
          get_haircut( );           /* be seated and be serviced */
     } else {
          up(&mutex);               /* shop is full; do not wait */
     }
}
```

Primitive di comunicazione e sincronizzazione in UNIX

- Tradizionali:
  - pipe: canali monodirezionali per produttori/consumatori
  - segnali: interrupt software asincroni
- *InterProcess Communication di SysV:*
  - semafori
  - memoria condivisa
  - code di messaggi
- Per la rete: *socket*

Semafori in UNIX

Caratteristiche dei semafori di Unix:

- i semafori sono mantenuti dal kernel, e identificati con un numero
- possiedono modalità di accesso simili a quelli dei file
- si possono creare *array* di semafori con singole operazioni
- si può operare su più semafori (dello stesso array) simultaneamente

Creazione dei semafori: semget(2)

```
int semget ( key_t key, int nsems, int semflg )
```

- `key`: intero identificante l'array di semafori
- `nsems`: numero di semafori da allocare
- `semflg`: flag di creazione, con modalità di accesso. Es: `IPC_CREAT | 0644` per la creazione di un nuovo insieme, con modalità di accesso `rw-r--r--` `0` per usare un insieme già definito.

Il risultato è una "handle" al set di semafori, oppure `NULL` su fallimento.

Operazione sui semafori: semop(2)

```
int semop(int semid, struct sembuf *sops,
      unsigned nsops)
```

- `semid`: puntatore all'array di semafori
- `sops`: puntatore ad array di operazioni; ogni operazione è una `struct sembuf` come segue:

```
struct sembuf {
  short int sem_num; /* semaphore number */
  short int sem_op;  /* semaphore operation */
  short int sem_flg; /* operation flag */
};
```

- `nsops`: numero di operazioni

Il risultato è `NULL` se la chiamata ha successo, -1 se è fallita.

Operazione sui semafori: semop(2)

Ogni singola operazione:

- `sem_num` indica su quale semaforo operare
- `sem_flg` può essere 0 (bloccante), `IPC_NOWAIT` (non bloccante), ...
- `sem_op` è un intero da sommare algebricamente al semaforo indicato da `sem_num`.
  - se il valore del semaforo andrebbe negativo, e `sem_flg`=0, il processo viene sospeso finché il valore non è sufficiente
  - valori positivi incrementano il semaforo (non sono bloccanti)

Tutte le operazioni vengono svolte atomicamente.

Controllo dei semafori: semctl(2)

```
int semctl(int semid, int semnum, int cmd, union semun arg)
```

Alcune possibili operazioni sono:

- `cmd=GETALL`: leggi i valori dei semafori
- `cmd=SETALL`: imposta i valori dei semafori
- `cmd=IPC_RMID`: cancella il set di semafori
- `cmd=GETPID`: restituisce il PID del processo che ha fatto l'ultima `semop`

Tutte le operazioni vengono svolte atomicamente.

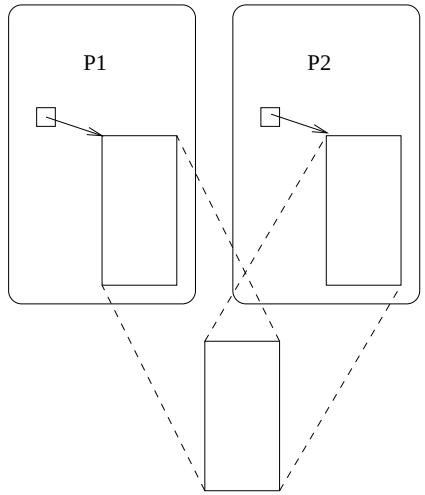
Queste operazioni di controllo non sono bloccanti.

Memoria condivisa

Due o più processi possono condividere parte dello spazio di indirizzamento.

Caratteristiche della memoria condivisa

- non è sincronizzante
- è allocata e deallocata dal kernel, ma accessibile direttamente da user space, senza necessità di system call.
- i segmenti condivisi possiedono modalità di accesso simili a quelli dei file
- è la più veloce forma di comunicazione tra processi



Creazione della memoria condivisa: shmget(2)

```
int shmget ( key_t key, int size, int shmflg )
```

- `key`: intero identificante il segmento di memoria
- `size`: dimensione (in byte)
- `shmflg`: flag di creazione, con modalità di accesso. Es: `IPC_CREAT | 0644` per la creazione di un nuovo segmento, con modalità di accesso `rw-r--r--` `0` per usare un segmento già definito.

Il risultato è una "handle" al segmento, oppure `NULL` su fallimento.

Operazioni sulla memoria condivisa

```
ptr = shmat(int shmid, const void *shmaddr, int shmflg)
```

- `shmid`: handle del segmento
- `shmaddr`: indirizzo dove appicicare il segmento condiviso. 0=lascia scegliere al sistema.
- `shmflg`: modo di attacco. Es: `SHM_RDONLY` = read-only. 0 = read-write.

Il risultato è il puntatore alla zona di memoria attaccata se la chiamata ha successo, 0 se è fallita.

`int shmdt(const void *shmaddr)`
Distacca la zona di memoria condivisa puntata da `shmaddr`

Esempio di uso della memoria condivisa

Problema: un processo "demone" che stampa una riga di caratteri ogni 4 secondi; un altro che controlla il carattere e la lunghezza della riga.

Soluzione: con memoria condivisa. File di definizione comuni:

```
/* line.h - definizioni comuni */
struct info {
  char c;
  int  length;
};

#define KEY      ((key_t)(1243))
#define SEGSIZE  sizeof(struct info)
```



```
/* pline.c - print a line of char - con shared memory */
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "line.h"

void main()
{
  int              i, id;
  struct info     *ctrl;

  /* Creazione dell'area condivisa */
  id = shmget(KEY, SEGSIZE, IPC_CREAT | 0666);
  if (id < 0) {
    perror("pline: shmget failed");
    exit(1);
  }
  /* Attaccamento all'area */
  ctrl = (struct info *)shmat(id, 0, 0);
  if (ctrl <= (struct info *)(0)) {
    perror("pline: shmat failed");
    exit(2);
  }
  /* Impostazione valori di default */
  ctrl->c      = 'a';
  ctrl->length = 10;

  /* Loop principale */
  while (ctrl->length > 0) {
    for (i=0; i<ctrl->length; i++) {
      putchar(ctrl->c);
      sleep(1);
    }
    putchar('\n');
    sleep(4);
  }
  exit(0);
}
```

```
/* cline.c - set the line to print - con shared memory */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "line.h"

void main(unsigned argc, char **argv)
{
  int              id;
  struct info     *ctrl;

  if (argc != 3) {
    fprintf(stderr, "usage: cline <char> <length>\n");
    exit(3);
  }

  /* Dichiarazione dell'area condivisa */
  id = shmget(KEY, SEGSIZE, 0);
  if (id < 0) {
    perror("cline: shmget failed");
    exit(1);
  }

  /* Attaccamento all'area */
  ctrl = (struct info *)shmat(id, 0, 0);
  if (ctrl <= (struct info *)(0)) {
    perror("cline: shmat failed");
    exit(2);
  }

  /* Copia dei valori nell'area condivis
  ctrl->c      = argv[1][0];
  ctrl->length = atoi(argv[2]);

  exit(0);
}
```


Esecuzione di pline/cline

```
miculan@coltrane:Shared_Memory$ pline
aaaaaaaaaa
aaaaaaaaaa
bbbbb
bbbbb
aaaaaaa
eeeeeeeeeeeeeeee
miculan@coltrane:Shared_Memory$
```

```
miculan@coltrane:Shared_Memory$ ./cline b 5
miculan@coltrane:Shared_Memory$ ./cline a 7
miculan@coltrane:Shared_Memory$ ./cline e 30
miculan@coltrane:Shared_Memory$ ./cline a 0
miculan@coltrane:Shared_Memory$
```


pline con semafori

Proteggiamo la sezione condivisa con un semaforo:

```
/* line.h - definizioni comuni */
struct info {
  char c;
  int  length;
};

#define KEY      ((key_t)(1243))
#define SEGSIZE  sizeof(struct info)
#define SKEY     ((key_t)(101))
```


pline con semaforo

```
/* sempline.c - print a line of char
 * con shared memory e semafori */
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include "line.h"

void main()
{
  int              i, j, id, sem;
  struct info     *ctrl;
  struct sembuf    lock, unlock;

  /* Creazione dell'area condivisa */
```

```
  id = shmget(KEY, SEGSIZE, IPC_CREAT | 0666);
  if (id < 0) {
    perror("pline: shmget failed");
    exit(1);
  }

  /* Attaccamento all'area */
  ctrl = (struct info *)shmat(id, 0, 0);
  if (ctrl <= (struct info *)(0)) {
    perror("sempline: shmat failed");
    exit(2);
  }

  /* Creazione del semaforo, se non c'e' gia' */
  sem = semget(SKEY, 1, IPC_CREAT | 0666);
  if (sem < 0) {
    perror("sempline: semget failed");
    exit(1);
  }
```

```
  /* Preparazione delle due operazioni sul semaforo */
  lock.sem_num = unlock.sem_num = 0;
  lock.sem_op  = -1;
  lock.sem_flg = unlock.sem_flg = 0;
  unlock.sem_op  = 1;

  /* Impostazione valori di default: */
  ctrl->c      = 'a';
  ctrl->length = 10;

  /* Loop principale (il semaforo e' ancora a 0) */
  while (ctrl->length > 0) {
    for (i=0; i<ctrl->length; i++) {
      putchar(ctrl->c);
      /* ciclo di rallentamento */
      for (j=0; j<1000000; j++);
    }
    putchar('\n');
```

```
    /* liberiamo l'area condivisa durante lo sleep... */
    semop(sem, &unlock, 1);
    sleep(4);
    /* ...ma ora ce la riprendiamo */
    semop(sem, &lock, 1);
  }

  /* Cancellazione del semaforo */
  semctl(sem, 0, IPC_RMID);

  exit(0);
}
```

cline con semaforo

```
/* semcline.c - set the line to print
 * con shared memory e semafori */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include "line.h"

void main(unsigned argc, char **argv)
{
  int              id, sem;
  struct info     *ctrl;
  struct sembuf    sop;

  if (argc != 3) {
```

```
    fprintf(stderr, "usage: semcline <char> <length>\n");
    exit(3);
  }

  /* Dichiarazione dell'area condivisa */
  id = shmget(KEY, SEGSIZE, 0);
  if (id < 0) {
    perror("semcline: shmget failed");
    exit(1);
  }

  /* Attaccamento all'area */
  ctrl = (struct info *)shmat(id, 0, 0);
  if (ctrl <= (struct info *)(0)) {
    perror("semcline: shmat failed");
    exit(2);
  }

  /* Dichiarazione del semaforo */
  sem = semget(SKEY, 1, 0);
  if (sem < 0) {
    perror("semcline: semget failed");
    exit(1);
  }

  /* Lock dell'area condivisa */
  sop.sem_num = 0;
  sop.sem_op  = -1;
  sop.sem_flg = 0;
  semop(sem, &sop, 1);

  /* Copia dei valori nell'area condivisa */
  ctrl->c      = argv[1][0];
  ctrl->length = atoi(argv[2]);

  /* Unlock dell'area condivisa */
  sop.sem_op  = 1;
  semop(sem, &sop, 1);

  exit(0);
}
```

Monitoraggio della memoria condivisa: ipcs e ipcrm

```
miculan@coltrane:Shared_Memory$ ipcs -m -s
------ Shared Memory Segments --------
key        shmid     owner     perms     bytes     nattch    status
0x00000000 923653    miculan   777       65536     2         dest
0x000004db 1017862   miculan   666       8         0

------ Semaphore Arrays --------
key        semid     owner     perms     nsems     status
0x00000065 0         miculan   666       1

miculan@coltrane:Shared_Memory$ ipcrm shm 1017862
resource deleted
miculan@coltrane:Shared_Memory$ ipcs -m
------ Shared Memory Segments --------
key        shmid     owner     perms     bytes     nattch    status
0x00000000 923653    miculan   777       65536     2         dest

miculan@coltrane:Shared_Memory$
```


Memoria condivisa con mmap(2)

Permette di accedere a file su disco come a zone di memoria.

`ptr = mmap(id *start, size_t length, int prot, int flags, int fd, off_t offset)`

- `start`: indirizzo suggerito per l'attacco. 0=lascia scegliere al sistema
- `length`: dimensione (in byte)
- `prot`: flag di protezione. Es: `PROT_READ | PROT_WRITE`
- `flags`: flag di mappatura. Esempi:
  `MAP_SHARED` : condiviso con altri processi; `MAP_PRIVATE` : mappatura privata
- `fd`: file descriptor del file da mappare in memoria
- `offset`: offset nel file da cui iniziare la mappatura

`ptr` punta all'inizio del segmento, oppure è `NULL` su fallimento.

Sincronizzazione di thread in Solaris 2

- Quattro primitive aggiuntive di sincronizzazione per supportare multitasking, multithreading (anche real-time) e macchine multiprocessore
- Vengono impiegate per i thread a livello kernel
- Accessibili anche per thread a livello utente.
- Quando un thread si blocca per una operazione su queste strutture, altri thread dello stesso processo possono procedere
- Sono implementate con operazioni di test-and-set

Sincronizzazione di thread in Solaris 2 (Cont.)

- **Semafori** di thread: singoli, incremento/decremento
- **Lock di mutua esclusione adattativi** (*adaptative mutex*), adatti per proteggere brevi sezioni di codice: sono spinlock che possono trasformarsi in veri block se il lock è tenuto da un processo in wait.
- **Variabili condition**, per segmenti di codice lunghi (associate a mutex)

```
mutex_enter(&m);
...
while(<condizione>) {
  cv_wait(&cv, &m);
}
...
mutex_exit(&m);
```

- **Lock lettura/scrittura**: più thread possono leggere contemporaneamente, ma solo un thread può avere accesso in scrittura.

Pipe e filtri

Le *pipe* sono la più comune forma di IPC

- canali unidirezionali FIFO, a buffer limitato, senza struttura di messaggio, tra due processi, accessibili attraverso dei file descriptor
- Le pipe sono al cuore della filosofia UNIX: le soluzioni a problemi complessi si ottengono componendo strumenti semplici ma generali ("distribuzione algoritmica a granularità grossa")
- Da shell, è possibile comporre singoli comandi in catene di pipe

```
$ ls | pr | lpr
```

- Classica soluzione per situazioni produttore/consumatore
- *Filtro:* un comando come *pr, awk, sed, sort*, che ricevono dati dallo standard input, lo processano e dànno il risultato sullo standard output.

Pipe: creazione, utilizzo

- Una pipe viene creata con *pipe(2)*:

```
#include <unistd.h>
int pipe(int filedes[2]);
```

- Se ha successo (risultato = 0)
  - `filedes[0]` è la coda della pipe (l'output)
  - `filedes[1]` è la testa della pipe (l'input)
- I due file descriptor possono essere usati per letture/scritture con le syscall *read(2), write(2)*

Pipe: creazione, utilizzo (cont.)

La creazione di una pipe viene sempre fatta da un processo padre, prima di uno o più fork. Ad esempio come segue:

1. Processo A crea una pipe, ottenendo i due file descriptor
2. A si forka due volte, creando i processi B, C, che ereditano i file descriptor
3. A chiude entrambi i file descriptor, B chiude quello di output, C quello di input della pipe
4. Ora B è il produttore e C è il consumatore; possono comunicare attraverso la pipe con delle `read/write`.

Pipe: creazione, utilizzo (cont.)

- Non viene creato nessun file su disco: viene solo allocato un buffer di memoria (tipicamente, 1 pagina (tipicamente, 4K))
- sincronizzazione "lasca" tra produttore/consumatore
  - una `read` da una pipe vuota blocca il processo finché il produttore non vi scrive
  - una `write` su una pipe piena blocca il processo finché il consumatore non legge
- La `read` restuisce EOF se non c'è nessun processo che ha aperto l'input della pipe in scrittura.

Pipe: esempio: `who | sort`

```
/* File whosort.c */
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

void main ()
{
  int fds[2];

  pipe(fds);    /* Crezione della pipe */

  /* Il 1o figlio attacca il suo stdin alla fine del pipe */
  if (fork() == 0) {
    dup2(fds[0], 0);


    close(fds[1]);   /* questo close e' essenziale */
```



```
    execlp("sort", "sort", 0);
  }

  /* Il 2o figlio attacca il suo stdout all'inizio del pipe */
  else if (fork() == 0) {
    dup2(fds[1], 1);
    close(fds[1]);
    execlp("who", "who", 0);
  }

  /* Il parent chiude la pipe, e aspetta i figli */
  else {
    close(fds[0]);
    close(fds[1]);  /* questo close e' essenziale */
    wait(0);
    wait(0);
  }

  exit(0);
}
```

Pipe: limiti

- Sono unidirezionali
- Non mantengono struttura del messaggio
- Un processo non può sapere chi è dall'altra parte del "tubo"
- Devono essere prearrangiate da un processo comune agli utilizzatori
- Non funzionano attraverso una rete (sono locali ad ogni macchina)
- Non sono permanenti

La soluzione a questi problemi saranno le *socket*.

Named pipe

- Sono pipe "residenti" su disco.
- Creazione: con la syscall *mknod(2)*, o con il comando `mknod`:

```
$ mknod tubo p
$ ls -l tubo
prw-r--r--  1 miculan ospiti  0 Jan 10 16:54 tubo
$
```

- Si usano come un normale file: si aprono con la *fopen*, e vi si legge/scrive come da qualunque file.
- In realtà i dati non vengono mai scritti su disco.
- Vantaggi: sono permanenti, hanno meccanismi di protezione, possono essere usate anche da processi non parenti
- Svantaggi: bisogna ricordarsi di "pulirle", alla fine.

Code di messaggi

Le *code di messaggi* sono una forma di IPC di SysV

- "mailbox" strutturate, in cui si possono riporre e ritirare messaggi
- preservano la struttura e il tipo dei messaggi
- accessibili da più processi
- non necessitano di un processo genitore comune per la creazione
- soggetti a controllo di accesso come i file, semafori,...
- sia le code, sia i singoli messaggi sono permanenti rispetto alla vita dei processi
- si possono monitorare e cancellare da shell con i comandi `ipcs`, `ipcrm`

Code di messaggi (cont.)

- consentono una sincronizzazione "lasca" tra processi
  - una lettura generica (senza tipo) da una queue vuota blocca il processo finché qualcuno non vi scrive un messaggio
  - una scrittura su una queue piena blocca il processo finché qualcuno non consuma un messaggio
- non sono permanenti su disco
- non permettono comunicazione tra macchine diverse

Adatte per situazioni produttore/consumatore locali, con messaggi strutturati.

Code di messaggi: creazione

```
id = msgget(key, flag)
```

- `key`: intero identificatore della coda di messaggi
- `flag`: modo di creazione:
  `IPC_CREAT | 0644` crea una nuova coda con modo `rw-r--r--`
  `0` si attacca ad una coda preesistente
- `id`: handle per successivo utilizzo della coda

Fallisce se si chiede di creare una coda che esiste già, o se si cerca di attaccarsi ad una coda per la quale non si hanno i permessi.

Code di messaggi: spedizione/ricezione

```
msgsnd(id, ptr, size, flag)
msgrcv(id, ptr, size, type, flag)
```

- `id`: handle restuita dalla `msgget` precedente
- `ptr`: puntatore ad una struct della forma

```
struct message {
   long mtype;
   char mtext[...];
}
```

- `size`: dimensione del messaggio in `mtext`
- `type`: tipo di messaggio richiesto; 0 = qualsiasi tipo
- `flag`: modo di spedizione/ricezione: `0` = comportamento normale (bloccante); `IPC_NOWAIT` = non bloccante

Segnali

- Strumenti per gestire eventi *asincroni* — sorta di interrupt software.
  Asincroni= non sono collegati o sincronizzati con una istruzione del processo che li riceve.
- Esempio: il segnale di interrupt SIGINT, è usato per fermare un comando prima della terminazione (CTRL-C).

  Un processo che esegue un'istruzione non valida riceve un SIGILL.
- I segnali sono impiegati anche per notificare eventi "normali"
  - iniziare/terminare dei sottoprocessi su richiesta
  - SIGWINCH informa il processo che la finestra in cui i dati sono mostrati è stata ridimensionata
- Ogni processo può cambiare la gestione di default dei segnali (tranne per SIGKILL)

Segnali: esempio

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
int n = 0;

void dispatcher(int sig)
{
  printf("--> SEGNALE %d: ", sig);

  switch (sig) {
  case SIGHUP:
    printf("incremento il contatore\n");
    n++;
    break;
  case SIGQUIT:
    printf("decremento il contatore\n");
    n--;
```

```
    break;
  case SIGINT:
    printf("me ne faccio un baffo!\n");
    break;
  }

  return;
}

void main(unsigned argc, char **argv)
{
  sigset(SIGINT, &dispatcher);
  sigset(SIGHUP, &dispatcher);
  sigset(SIGQUIT, &dispatcher);

  while (n < 5) {
    pause();
  }
  exit(0);
}
```

Segnali: esempio (cont.)

```
> a.out >log &
[1] 18293
> kill -INT 18293
> kill -INT 18293
> kill -HUP 18293
> kill -HUP 18293
> kill -HUP 18293
> kill -QUIT 18293
> kill -QUIT 18293
> kill -HUP 18293
> kill -HUP 18293
> kill -HUP 18293
> kill -HUP 18293
> kill -HUP 18293
18293: No such process
[1]    Done                    a.out > log
> more log
--> SEGNALE 2: me ne faccio un baffo!
--> SEGNALE 2: me ne faccio un baffo!
--> SEGNALE 1: incremento il contatore
--> SEGNALE 1: incremento il contatore
--> SEGNALE 1: incremento il contatore
--> SEGNALE 3: decremento il contatore
--> SEGNALE 3: decremento il contatore
--> SEGNALE 1: incremento il contatore
--> SEGNALE 1: incremento il contatore
--> SEGNALE 1: incremento il contatore
--> SEGNALE 1: incremento il contatore
>
```


Cosa fare quando arriva un segnale? Alcuni esempi

1. Ignorare il segnale!
2. Terminare il processo in modo “pulito”
3. Riconfigurare dinamicamente il processo
4. Dumping di tabelle e strutture dati interne
5. Attivare/Disattivare messaggi di debugging
6. Implementare un timeout sulle chiamate bloccanti