Concorrenza

Concorrenza

I sistemi operativi devono lavorare con più processi:

- Multiprogrammazione → gestione del sistema operativo di più processi/thread in un singolo core
- Multiprocessi → più processi in più core
- Gestione distribuita → più processi in più macchine

Questo può portare ad avere concorrenza, che avviene:

- in applicazioni dove i processi accedono alle stesse risorse (es. scrittura su file)
- in applicazioni strutturate (es. applicazioni modulari)
- per la struttura stessa del sistema operativo, che può essere implementato come una serie di thread/processi

Termini relativi alla concorrenza:

1. Operazione atomica → operazione che non può essere interrotta durante la sua esecuzione; alcune possono essere intrinsecamente atomiche (es. assegnazione di una variabile), altre sono più complesse ma sempre date da 1 istruzione macchina (es. i++ non è istruzione atomica perché è formata da più istruzioni assembly)
2. **Sezione critica** → parte del codice nel quale viene utilizzata una risorsa condivisa (es. variabile globale ⇨ il cambiamento della variabile può cambiare il codice se acceduta da più processi)
3. **Mutua esclusione** → requisito per cui se un processo è in sezione critica per una risorsa, nessun altro processo può essere in sezione critica per la stessa risorsa condivisa
4. Race condition (corsa alla risorsa) → situazione in cui più thread o processi cercano di accedere ad uno stesso dato; l'ultimo che arriva è quello che scrive l'ultimo valore nella variabile
5. **Deadlock** → situazione in cui più processi risultano bloccati perché si aspettano a vicenda per compiere un'azione (es. per errori di programmazione; non posso uscirne ⇨ dead = morte)
6. **Livelock** → situazione in cui più processi cambiano continuamente il loro stato in risposta ai cambiamenti di stato negli altri processi senza fare lavoro utile (situazione di stallo ⇨ posso uscirne)

7. **Starvation** → situazione in cui un processo potrebbe terminare, andare avanti e accedere alla risorsa di cui ha bisogno ma non riesce a farlo (es. processo a bassa priorità con processi ad alta priorità in livelock)

L'output di un processo deve essere indipendente dalla velocità di esecuzione di altri processi concorrenti

Nel momento in cui i processi concorrono per una risorsa bisogna considerare:

1. **Interleaving** ⇨ i processi/thread si intervallano nello stesso processore
2. Overlapping ⇨ i processi lavorano in parallelo (su più core)

➔ possono sembrare due cose distinte ma in realtà hanno lo stesso problema

Le difficoltà principali della concorrenza sono:

- condivisione di risorse globali
- difficoltà per il sistema operativo di gestire l'allocazione di risorse in maniera ottimale
- difficoltà nel localizzare errori di programmazione in quanto i risultati non sono deterministici e riproducibili

Un sistema operativo deve gestire questi problemi sapendo tutti i processi che sono in esecuzione, occuparsi di allocare/deallocare le risorse per i processi, proteggere i dati e le risorse fisiche di ogni processo contro l'interferenza di altri processi e assicurarsi che i processi e gli output siano indipendenti dalla velocità del processo

Interazione tra processi:

Degree of Awareness	Relationship	Influence that One Process Has on the Other	Potential Control Problems
Processes unaware of each other	Competition	• Results of one process independent of the action of others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation • Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Deadlock (consumable resource) • Starvation

Mutua esclusione

```
PROCESS 1 */

void P1
{
   while (true) {
      /* preceding code */;
      entercritical (Ra);
      /* critical section */;
      exitcritical (Ra);
      /* following code */;
   }
}
```

```
/* PROCESS 2 */

void P2
{
   while (true) {
      /* preceding code */;
      entercritical (Ra);
      /* critical section */;
      exitcritical (Ra);
      /* following code */;
   }
}
```

. . .

```
/* PROCESS n */

void Pn
{
   while (true) {
      /* preceding code */;
      entercritical (Ra);
      /* critical section */;
      exitcritical (Ra);
      /* following code */;
   }
}
```

dati più processi, essi rientrano ciclicamente nella sezione critica (ma alcuni processi possono semplicemente terminare)

→ esiste un comando per entrare in sezione critica e uno per uscirne
→ se uno dei processi è in sezione critica tutti gli altri si bloccano in *entercritical(Ra)*
→ quando il processo esce dalla sezione critica, avvisa gli altri per farne entrare un altro

i requisiti per la gestione della mutua esclusione sono:

- un processo che si ferma in una sezione non critica deve farlo senza interferire con gli altri processi
- i processi non devono andare in deadlock/livelock
- se la sezione critica è libera (non c'è nessun processo), un processo deve potervi accedere → solo il sistema operativo può negare l'accesso a una sezione critica libera per evitare deadlock (rallenta pesantemente l'esecuzione)
- non bisogna programmare pensando alla velocità di processo o al n° di processi
- un processo rimane nella sezione critica per un tempo finito

Per gestire la mutua esclusione tramite hardware si possono utilizzare:

1. interrupt disabling → disabilitando gli interrupt prima di entrare in sezione critica è garantito che anche se il tempo concesso al processo dallo scheduler è terminato, il processo non può essere interrotto; minore efficienza e non funziona in architetture multiprocessore (perché se ci sono più processi che possono lavorano su più core, non è sicuro che la sezione critica sia protetta)
2. compare&swap → "if" a livello hardware (istruzione atomica); viene confrontato il valore in memoria con un valore di test; se il valore è lo stesso del test, viene fatta la swap con il nuovo valore (che va a finire nel registro del vecchio valore) e restituisco il vecchio valore
   ⇨ si usa la compare&swap per controllare se la sezione critica è occupata(1) o meno(0); se non è occupata, il processo entra

```
int compare_and_swap(int* reg, int oldval, int newval)
{
  ATOMIC();
  int old_reg_val = *reg;
  if (old_reg_val == oldval)
     *reg = newval;
  END_ATOMIC();
  return old_reg_val;
}
```

N.B. non è un vero e proprio codice ma un riferimento per capire

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
  while (true) {
    while (compare_and_swap(&bolt, 0, 1) == 1)
        /* do nothing */;
    /* critical section */;
    bolt = 0;
    /* remainder */;
  }
}
void main()
{
  bolt = 0;
  parbegin (P(1), P(2), . . . ,P(n));

}
```

3. exchange instruction → simile a compare&swap per funzionamento (uso lo zero come fosse un pass/token); scambia il contenuto di un registro con quello di una locazione di memoria (classico scambio)

```
void exchange (int *register, int *memory)
{
  int temp;
  temp = *memory;
  *memory = *register;
  *register = temp;
}
```

```
/* program mutualexclusion */
int const n = /* number of processes*/;
int bolt;
void P(int i)
{
  while (true) {
    int keyi = 1;
    do exchange (&keyi, &bolt) while (keyi != 0);
    /* critical section */;
    bolt = 0;
    /* remainder */;
  }
}
void main()
{
  bolt = 0;
  parbegin (P(1), P(2), . . ., P(n));
}
```

vantaggi: non bisogna programmare in assembly; non importa il n° di processi; funziona sia su uniprocessori che su multiprocessori; garantisce il supporto a più sezioni critiche

svantaggi: busy-waiting (→ il processo che aspetta l'accesso a una sezione critica occupata continua a perdere tempo nel processore); possibile starvation quando un processo lascia una sezione critica e più di un processo sta aspettando; possibili deadlock; non disponibile su ogni architettura

Per gestire la mutua esclusione tramite software si possono utilizzare:

1. **semaforo** → costrutto basato su un valore intero usato per segnalazioni tra i processi; solo tre operazioni possono essere fatte da un semaforo (tutte atomiche):
   - initialize ⇨ inizializza il valore (non negativo) al n° di risorse disponibili
   - semWait ⇨ può risultare in un blocco; decremento il valore e se è negativo blocco il processo tramite il sistema operativo (running → blocked)
   - semSignal ⇨ può risultare in uno sblocco del processo; incremento la variabile del semaforo
2. **semaforo binario** → semaforo con 0 e 1

3. mutex → simile al semaforo binario in cui il processo che blocca il mutex (⇨ setta il valore a 0) deve essere lo stesso che lo sblocca (⇨setta il valore a 1)
4. variabile condizionale → un tipo di dato che viene utilizzato per bloccare un processo o un thread finché una particolare condizione è verificata
5. **monitor** → costrutto di un linguaggio di programmazione che incapsula variabili, procedure di accesso e codice di inizializzazione all'interno di un tipo di dato astratto; solo un processo alla volta può accedere attivamente al monitor
6. event flag → parole di memoria utilizzate come meccanismo di sincronizzazione; un processo aspetta uno o più eventi tramite i bit di interesse; il processo si blocca fino a che tutti i bit non saranno settati nella flag o almeno uno dei bit non sia settato
7. **mailbox/messaggi** → mezzo con cui due processi possono scambiare informazioni e può essere utilizzato per la sincronizzazione
8. spinlocks → un processo esegue un loop infinito aspettando una variabile di lock per indicare la disponibilità (busy-waiting)

Semaforo

conseguenze dell'uso di un semaforo:

1. non c'è modo per sapere a priori se il decremento porta un blocco o meno
2. non c'è modo su un sistema a singolo processore di sapere quale processo tra due concorrenziali verrà eseguito dopo una semWait
3. non c'è modo di sapere se dopo una SemSignal sblocco i processi o meno

primitive di un semaforo

```
struct semaphore {
     int count;
     queueType queue;
};
void semWait(semaphore s)
{
     s.count--;
     if (s.count < 0) {
          /* place this process in s.queue */;
          /* block this process */;
     }
}
void semSignal(semaphore s)
{
     s.count++;
     if (s.count <= 0) {
          /* remove a process P from s.queue */;
          /* place process P on ready list */;
     }
}
```

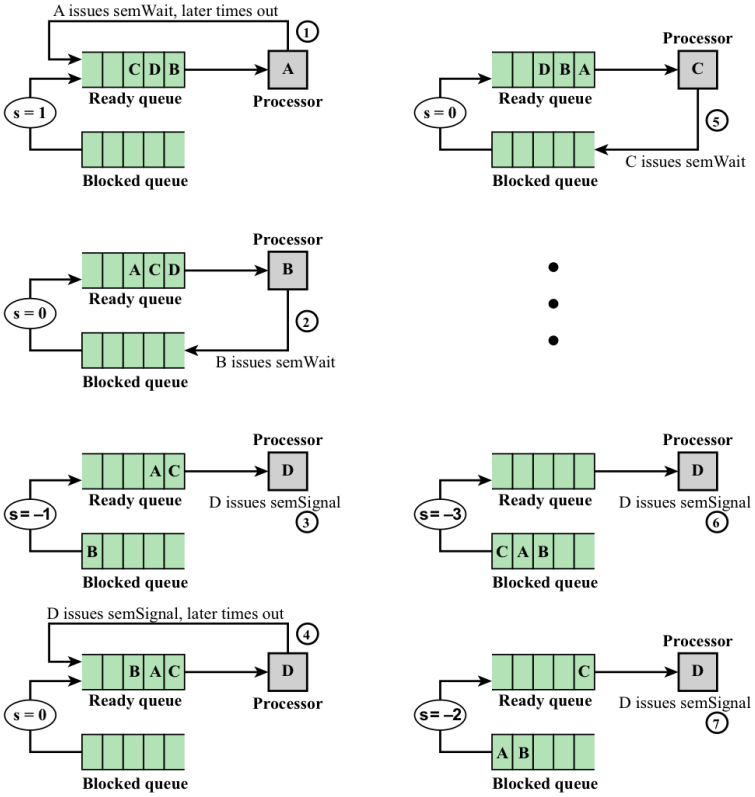
primitive di un semaforo binario

```
struct binary_semaphore {
     enum {zero, one} value;
     queueType queue;
};
void semWaitB(binary_semaphore s)
{
     if (s.value == one)
          s.value = zero;
     else {
          /* place this process in s.queue */;
          /* block this process */;
     }
}
void semSignalB(semaphore s)
{
     if (s.queue is empty())
          s.value = one;
     else {
          /* remove a process P from s.queue */;
          /* place process P on ready list */;
     }
}
```

La politica della gestione della coda differenzia i semafori in:

- semafori strong → seguono la politica FIFO
- semafori weak → posso definire politiche arbitrarie (es. priorità)

es. funzionamento di un semaforo strong



una coda ready ∀ sistema operativo
una coda blocked ∀ semaforo

Ogni volta che chiamo signal un processo bloccato va in ready

Ogni volta che chiamo wait se semaforo diventa 0 procede altrimenti viene messo nella coda del semaforo

mentre D chiama signal, B viene rimesso in coda Ready

Mutua esclusione con messaggi:

```
/* program mutualexclusion */
const int n = /* number of processes  */;
semaphore s = 1;
void P(int i)
{
     while (true) {
          semWait(s);
          /* critical section   */;
          semSignal(s);
          /* remainder   */;
     }
}
void main()
{
     parbegin (P(1), P(2), . . ., P(n));
}
```

SemWait e SemSignal devono essere implementate come operazioni atomiche ⇨ non posso bloccare una delle due operazioni in corsa

→ i semafori possono essere implementati tramite hardware o firmware; posso inoltre usare degli schemi software come quelli di Dekker o l'algoritmo di Peterson

es. due possibili implementazioni

compare&swap

```
semWait(s)
{
   while (compare_and_swap(s.flag, 0 , 1) == 1)
      /* do nothing */;
   s.count--;
   if (s.count < 0) {
      /* place this process in s.queue*/;
      /* block this process (must also set s.flag to 0) */;
   }
   s.flag = 0;
}

semSignal(s)
{
   while (compare_and_swap(s.flag, 0 , 1) == 1)
          /* do nothing */;
      s.count++;
   if (s.count <= 0) {
      /* remove a process P from s.queue */;
      /* place process P on ready list */;
      }
      s.flag = 0;
}
```

uso di interrupts

```
semWait(s)
{
   inhibit interrupts;   // se uni-processore OK altrimenti problemi
   s.count--;
   if (s.count < 0) {
      /* place this process in s.queue */;
      /* block this process and allow interrupts */;
   }
   else
     allow interrupts;
}

semSignal(s)
{
   inhibit interrupts;
   s.count++;
   if (s.count <= 0) {
      /* remove a process P from s.queue */;
      /* place process P on ready list */;
   }
   allow interrupts;
}
```

Semafori Posix

int sem_init(sem_t * sem, int pshared, unsigned value) → inizializza il semaforo

- sem: puntatore al semaforo
- pshared: se vale 0, il semaforo è condiviso tra i thread del processo; altrimenti il semaforo è condiviso tra i processi che si trovano nell'area di memoria condivisa con il semaforo
- value: valore del semaforo (n° risorse)

int sem_wait(sem_t *sem) → semWait sul semaforo sem

int sem_post(sem_t *sem) → semSignal sul semaforo sem

int sem_destroy(sem_t *sem) → distrugge il semaforo sem

➔ ritornano -1 in caso di errore, 0 altrimenti

Semafori Java

Semaphore(int value) //initialization

Semaphore(int value, boolean how) //initialization

acquire() //wait

release() //signal

Semafori Python

threading.Semaphore([value]) //initialization

acquire([blocking]) //wait

release() //signal

Parametri:

- value: valore iniziale del semaforo (default 1)
- blocking: permette al codice di impedire che il semaforo blocchi il

Named Semaphore

Un semaforo named è identificato univocamente dal suo nome; il nome è una stringa preceduta da uno slash ("/semaforo") → processi diversi accedono tramite il nome

sem_t *sem_open(const char *name, int oflag)

sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value)

→ crea e inizializza il semaforo; le due segnature che dipendono dai flag

- name: nome del named semaphore
- oflag: flag che controllano la open (definiti in fcntl.h); possibili flag:
  - O_CREAT il semaforo viene creato se non esiste già
  - O_CREAT |O_EXCL il semaforo viene creato se non esiste già; se esiste già viene lanciato un errore

  ➔ quando creato, l'user e il group ID sono quelli del processo chiamante

  ➔ se O_CREAT compare nel flag devono essere specificati gli altri 2 parametri
- mode: specifica i permessi del semaforo; maschera ottale $0xyz$ con
  - x → permessi proprietario
  - y → permessi gruppo
  - z → permessi altri utenti

  x, y, z sono costruiti sommando i valori:
  - 0 → nessun permesso
  - 1 → permesso di esecuzione
  - 2 → permesso di scrittura
  - 4 → permesso di lettura
- value: valore iniziale del semaforo

➔ ritornano il puntatore al semaforo in caso di successo, SEM_FAILED altrimenti (e viene settato errno)

int sem_wait(sem_t *sem) → semWait sul semaforo sem

int sem_post(sem_t *sem) → semSignal sul semaforo sem

int sem_close(sem_t *sem) → chiude il semaforo; deve essere fatta da ogni processo terminato il lavoro; se un thread chiude un semaforo su cui erano bloccati altri thread, il loro comportamento è indefinito

int sem_unlink(const char *name) → distrugge il semaforo dal sistema operativo

- name: nome del semaforo

➔ il semaforo verrà distrutto non appena tutti i processi che lo hanno aperto in precedenza lo avranno chiuso

int sem_getvalue(sem_t *sem, int *sval) → consente di leggere il valore corrente del semaforo

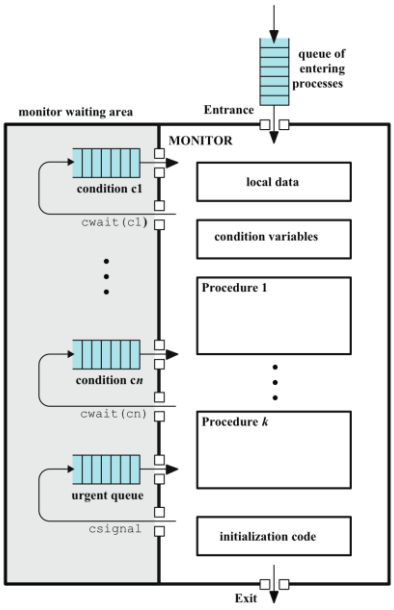
- sem: puntatore al semaforo
- sval: puntatore ad un intero che verrà settato al valore del semaforo

➔ su Linux se ci sono processi in coda al semaforo, sval è settato a 0

Monitors

Costrutti con funzionalità simili ai semafori ma più semplici da controllare; è come se fosse una classe

- implementati in altri linguaggi rispetto a C (non c'è standard)
- consiste in una o più procedure, una sequenza di inizializzazione e memoria locale



il vantaggio dell'utilizzo di un monitor deriva dal fatto che non bisogna codificare nessun meccanismo che realizza la mutua esclusione

- le variabili locali possono essere accedute soltanto dalle procedure del monitor (sono private)
- il processo entra nel monitor invocando una delle sue procedure
- nel monitor posso eseguire solo un processo alla volta

La sincronizzazione nel monitor avviene attraverso l'uso di variabili condizionali; modifico le variabili tramite *cwait(c)* [sospende esecuzione del processo chiamante sulla condizione c] e *csignal(c)* [ripristina l'esecuzione di alcuni processi bloccati dopo la cwait sulla stessa condizione]

L'operazione *cwait(c)*, applicata ad una variabile condizionale c, permette di sospendere un processo che occupa il monitor, facendo in modo che il processo sparisca temporaneamente dal monitor e venga posto in una coda d'attesa per quella variabile condizionale, dando così via libera ad un nuovo processo che desidera entrare nel monitor oppure ad un altro processo pronto a riprendere l'esecuzione. L'operazione *csignal*(*c*) risveglia esattamente un processo sospeso sulla variabile condizionale c per cui è chiamata; questo processo riprende la propria esecuzione appena ha via libera. In ogni caso, quando non ci sono processi in attesa sulla variabile condizionale per cui è chiamata la notifica, non accade nulla

Message Passing

Approccio per la sincronizzazione e comunicazione tra processi; scambio di messaggi che crea un dialogo tra 2 processi (ma anche più); lavora anche su sistemi distribuiti

2 primitive:

- send (destination, message) → un processo invia informazioni in forma di messaggio ad un altro processo (destinatario)
- receive (source, message) → un processo riceve informazioni eseguendo la primitiva receive, indicando la sorgente e il messaggio

Caratteristiche di sistemi basati su message passing:

Sincronizzazione
send
- bloccante
- non bloccante

receive
- bloccante
- non bloccante
- test di messaggi in arrivo

Indirizzamento
Diretto
- send
- receive
  - esplicito
  - implicito

Indiretto
- statico
- dinamico
- possesso

Formato
Contenuto
Lunghezza
- Fissa
- Variabile

Tipo di Coda
FIFO
A priorità

Sincronizzazione:

send	*receive*	*caratteristiche*
bloccante	bloccante	sia il mittente sia il ricevente sono bloccati fino al completamento dell'operazione (rendez-vous) → sincronizzazione stretta tra processi a questa tipologia di scambio di messaggi appartiene anche la chiamata di procedura remota (RPC – Remote Procedure Call) detta anche "rendez-vous esteso"
non bloccante	bloccante	il mittente può continuare con il suo codice, il ricevente deve aspettare l'arrivo del messaggio → il sender può inviare vari messaggi a vari destinatari molto velocemente
non bloccante	non bloccante	né il mittente né il destinatario devono aspettare

→ un pericolo potenziale della send non bloccante è che un errore potrebbe portare ad una situazione in cui un processo continua a generare messaggi; siccome non c'è modo di bloccare il mittente, i messaggi possono consumare le risorse di sistema, in particolare il tempo di processore e lo spazio dei buffer, a danno degli altri processi e del sistema operativo

→ inoltre, la send non bloccante lascia al programmatore il compito di controllare che un messaggio arrivi a destinazione: i processi devono mandare messaggi di risposta per confermare la ricezione di un messaggio

→ in caso di receive bloccante, un processo che richiede un messaggio avrà bisogno, per procedere, delle informazioni in esso contenute; tuttavia, se un messaggio è perso, cosa che può accadere nei sistemi distribuiti, o se un processo fallisce prima di poter mandare il messaggio, il processo ricevente rimarrà bloccato per sempre

→ il problema si può risolvere utilizzando una receive non bloccante; in questo secondo caso il pericolo è che ogni messaggio, inviato dopo che il processo ha già effettuato la receive corrispondente, andrà perso

→ un altro approccio possibile è dare al processo la possibilità di controllare se c'è un messaggio in arrivo prima di effettuare la receive; oppure specificare più di un mittente in una receive. Quest'ultima possibilità è utile quando un processo aspetta messaggi provenienti da processi diversi, e se solo un messaggio è sufficiente per continuare l'esecuzione

Addressing

Per instaurare una comunicazione fra processi è necessario che fra i parametri delle specifiche send e receive vi siano il destinatario e il mittente dei messaggi; 2 tipi di indirizzamento:

- indirizzamento diretto → ogni processo che intenda comunicare deve nominare esplicitamente il ricevente e il trasmittente della comunicazione
  in questo caso la primitiva *send()* richiede un identificatore del processo di destinazione
  la primitiva *receive()* può essere gestita in 2 modi:
  - addressing esplicito → richiede che il processo esplicitamente designi un processo mittente; utile per i processi concorrenti che cooperano
  - addressing implicito → soltanto il trasmittente nomina il ricevente, mentre il ricevente non deve nominare il trasmittente (es. un processo server per la stampante deve accettare richieste di stampa da qualunque processo)
- indirizzamento indiretto → i messaggi si inviano a delle mailbox (o porte), che li ricevono; una porta si può considerare in modo astratto come un oggetto nel quale i processi possono introdurre e prelevare messaggi ed è identificata in modo unico

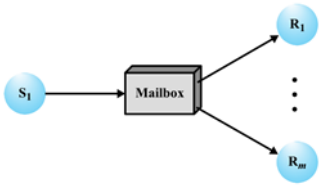
  un processo invia un messaggio a una mailbox e l'altro processo lo prende da essa

Relazione mittente-ricevente:

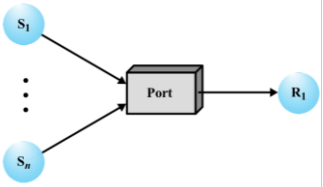
Lo scambio di messaggi consente di instaurare una comunicazione di tipo uno-a-uno, uno-a-molti, molti-a-uno, molti-a- molti

a) uno-a-uno → permettono connessioni private fra due processi; in questo modo si isolano le interazioni fra i due da possibili interferenze causate da errori degli altri processi; questa tipologia di relazione instaura un canale di tipo simmetrico, spesso chiamato semplicemente canale o link

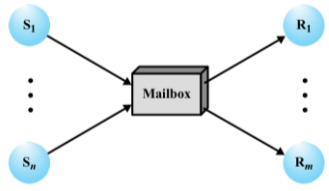
b) uno-a-molti → permettono di avere un mittente e molti destinatari: è utile per applicazioni dove un messaggio o qualche informazione deve essere mandata ad un insieme di processi



c) molti-a-uno → sono tipiche delle interazioni client/server, dove un processo fornisce un servizio per molti processi; la mailbox in questo caso è detta porta



d) molti-a-molti → i processi client inviano richieste non ad un particolare server, ma ad uno qualunque scelto tra un insieme di server equivalenti



Questo caso comporta problemi di natura realizzativa, infatti, il supporto a tempo di esecuzione del linguaggio deve garantire che un messaggio di richiesta sia inviato a tutti i processi server e deve assicurare che, non appena il messaggio è ricevuto da uno di essi, lo stesso non sia più disponibile per tutti gli altri server

Formato del messaggio:

Il formato dei messaggi dipende dagli obiettivi del sistema di scambio di messaggi e dall'uso di un singolo computer o di un sistema distribuito.

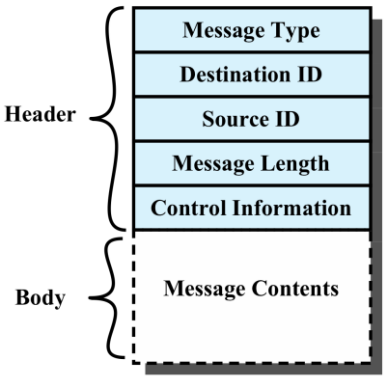
Per alcuni sistemi operativi i progettisti hanno scelto messaggi corti di lunghezza fissata, in modo da minimizzare il sovraccarico e lo spazio richiesto, inoltre la realizzazione a livello di sistema è semplice anche se i limiti imposti rendono più difficile il compito della programmazione.

Se è necessario passare una gran quantità di dati, si possono mettere i dati in un file e indicare semplicemente il nome del file nel messaggio.

L'uso di messaggi di dimensione variabile costituisce un approccio più flessibile e anche se la scelta di messaggi a dimensione variabili richiede una realizzazione più complessa a livello del sistema, il lavoro di programmazione risulta semplificato.

Il messaggio p è formato da due parti:

- un'intestazione, che contiene informazioni riguardo al messaggio
- un corpo, che contiene il messaggio vero e proprio



L'intestazione può contenere:

- un identificatore del mittente e del destinatario del messaggio
- un campo per la lunghezza
- un campo che indica il tipo di messaggio
- eventualmente ci possono essere delle informazioni di controllo, come un puntatore per creare una lista di messaggi, un numero sequenziale per tenere traccia dei messaggi

Mutua esclusione con messaggi:

```
/* program mutualexclusion */
const int n = /* number of processes  */;
void P(int i)
{
    message msg;
    while (true) {
     receive (box, msg);
     /* critical section   */;
     send (box, msg);
     /* remainder   */;
    }
}
void main()
{
    create_mailbox (box);
    send (box, null);
    parbegin (P(1), P(2), . . ., P(n));
}
```

Producer/Consumer Problem

- uno o più produttori generano degli elementi e li mettono in un buffer
- uno o più consumatori prendono gli elementi dal buffer
- solo 1 produttore o consumatore alla volta può accedere al buffer

➔ bisogna assicurarsi che il produttore non possa mettere ulteriori elementi in un buffer pieno e il consumatore non possa rimuovere elementi da un buffer vuoto

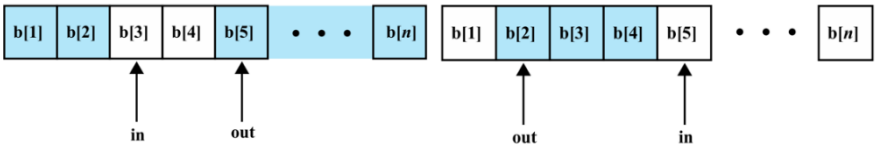
```
/* program producerconsumer */
semaphore n = 0, s = 1;

void producer()
{
   while (true) {
        produce();
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
   }
}

void consumer()
{
   while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        consume();
   }
}
```

n = risorse da consumare
s = semaforo
(buffer dimensione infinita)

Se però il buffer ha dimensione finita, bisogna gestirlo circolarmente



in = prima posizione in cui posso aggiungere un elemento

out = prima posizione in cui posso prendere un elemento

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n= 0, e= sizeofbuffer;
```

e = semaforo empty ; mi indica n° posizioni libere

n = semaforo che indica n° posizioni occupate

```
void producer()
{
     while (true) {
          produce();
          semWait(e);   → diminuisco posizioni libere
          semWait(s);   [se e<0 mi blocco e devo aspettare la semSignal di un consumatore]
          append();     ← entrata e uscita dalla sezione critica
          semSignal(s);
          semSignal(n);
     }
}
```

```
void consumer()
{
     while (true) {
          semWait(n);   → diminuisco n
          semWait(s);   ← entrata e uscita dalla sezione critica
          take();
          semSignal(s);
          semSignal(e);
          consume();
     }
}
```

➔ è necessario che il consumatore si blocchi se il produttore sta lavorando in un'altra sezione dell'array? no, quindi si può usare un semaforo per i produttori (s1) e uno per i consumatori (s2) che bloccano la possibilità che 2+ produttori/consumatori lavorino sulla stessa sezione di array

➔ in questo caso può capitare che un produttore e un consumatore lavorino contemporaneamente sulla stessa casella? in teoria si, in pratica no perché se sono nella stessa posizione significa che o il buffer è pieno (produttore bloccato) o il buffer è vuoto (consumatore bloccato)

➔ se ci sono solo un produttore e solo un consumatore? non è necessaria la presenza dei semafori per la sezione critica (bastano e e n)

```
void producer()
{
     char x;
     while (true) {
     produce(x);
     append(x);
     }
}
void consumer()
{
     char x;
     while (true) {
       take(x);
       consume(x);
     }
}
```

P/C usando monitor

produttore → produce un valore x e lo aggiunge al buffer con append (append è parte di un monitor)

consumatore → toglie dal buffer con take (take è parte di un monitor) e lo consuma

➔ l'utilizzo di un monitor unico per tutti i produttori e per tutti i consumatori implica che se un produttore sta scrivendo un consumatore non può consumare

prox posizione dove scrivere/leggere — n° elementi nel buffer

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                    /* space for N items */
int nextin, nextout;                /* buffer pointers */
int count;                          /* number of items in buffer */
cond notfull, notempty;  /* condition variables for synchronization */
```

Init: `nextin = nextout = count = 0;`

condizioni binarie

blocco produttore finché qualcuno non consuma; viene messo nella coda di attesa legata alla variabile not full

```
void append (char x)
{
     if (count == N) cwait(notfull);
     buffer[nextin] = x;
     nextin = (nextin + 1) % N;
     count++;
     /* one more item in buffer */
     csignal(notempty);
}
```

segnalo che il buffer non è sicuramente più vuoto perché il produttore ha appena messo un elemento

blocco consumatore finché qualcuno non produce; viene messo nella coda di attesa legata alla variabile notempty

```
void take (char x)
{
     if (count == 0) cwait(notempty);
     x = buffer[nextout];
     nextout = (nextout + 1) % N;
     count--;
     csignal(notfull);
}
```

segnalo che il buffer non è sicuramente più pieno perché il consumatore ha appena preso un elemento

P/C usando message passing

```
const int
    capacity = /* buffering capacity */ ;
    null =/* empty message */ ;
int i;
void producer()
{   message pmsg;
    while (true) {
     receive (mayproduce, pmsg);
     pmsg = produce();
     send (mayconsume, pmsg);
    }
}
void consumer()
{   message cmsg;
    while (true) {
     receive (mayconsume, cmsg);
     consume (cmsg);
     send (mayproduce, null);
    }
}

void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1; i <= capacity; i++) send (mayproduce,
null);
    parbegin (producer, consumer);
}
```

non ho un buffer ma una coda di messaggi che riportano il valore prodotto/consumato (message passing indiretto)

2 mailbox: *mayproduce* produce, *mayconsume* consuma (perché la mailbox può contenere un numero di messaggi limitato)

mando un numero di messaggi = capacity nella *mayproduce* con messaggio null

➔ un produttore che dopo aver prodotto vuole mandare un messaggio lo può fare solo se può produrre (→ se trova un messaggio in *mayproduce* c'è uno spazio libero nella capacità della mailbox)

Readers/Writers Problem

alternativa al P/C; l'area di memoria viene condivisa tra i processi → non produco e consumo ma leggo e scrivo

condizioni da verificare:

1. più lettori possono leggere simultaneamente
2. solo uno scrittore alla volta può scrivere
3. se uno scrittore sta scrivendo i lettori non possono leggere → se qualcuno lo ha già aperto in lettura lo scrittore non può accedere

```
/* program readersandwriters */
int readcount = 0;
semaphore x = 1, wsem = 1;
```

gestione readcount ↵ ↳ semaforo scrittura

writer()

```
while (true) {
  semWait (wsem);
  WRITEUNIT();
  semSignal (wsem);
}
```

reader()

```
while (true) {
  semWait (x);
  readcount++;
  if (readcount == 1) semWait (wsem);
  semSignal (x);
  READUNIT();
  semWait (x);
  readcount--;
  if (readcount == 0) semSignal (wsem);
  semSignal (x);
}
```

↗ blocco scrittori

↗ sblocco scrittori

Problema? I lettori hanno forte priorità; se avessi tanti lettori che arrivano al secondo, gli scrittori finirebbero per non scrivere più (readcount mai = a 0)

→ sicuramente non ho deadlock ma posso avere starvation dei writers

➔ dovrò bloccare i nuovi arrivi dei reader