

Concorrenza: Mutua Esclusione e Sincronizzazione

Esistono tre tipi di modi per gestire processi e thread:

1. MULTIPROGRAMMAZIONE: gestioni di processi multipli in un sistema a singolo processore
2. MULTIPROCESSING: gestione di processi multipli in un multiprocessore
3. PROCESSI DISTRIBUITI: gestione di processi multipli eseguiti su sistemi distribuiti

Il concetto di concorrenza è basilare per questi approcci. I contesti in cui appare sono:

- Applicazioni Multiple: la multiprogrammazione fu inventata per consentire di dividere dinamicamente il tempo tra le varie applicazioni attive
- Applicazioni Strutturate: estensioni dei principi di progettazione modulare e programmazione strutturata.
- Struttura dei SO: anche i SO sono implementati come insiemi di processi o thread

Termini utili legati alla concorrenza:

- i. Operazione Atomica: sequenza di una o più istruzioni che appaiono indivisibili, quindi o il processo viene bloccato dopo queste o prima ma non durante la loro esecuzione
- ii. Sezione Critica: sezione di codice che richiede l'accesso a risorse condivise. Solo un processo alla volta può entrarci
- iii. Deadlock: situazione in cui due, o più processi, non possono procedere in quanto ognuno è in attesa che uno degli altri faccia qualcosa
- iv. Livelock: situazione in cui due o più processi continuamente cambiano i loro stati in risposta alla variazione degli altri quindi sono attivi ma non fanno nessun lavoro utile
- v. Mutua Esclusione: è richiesta quando un processo entra nella sezione critica e quindi gli altri non devono accedervi
- vi. **Race Condition**: → problema che si verifica quando più processi o thread leggono e scrivono dei dati
 - il risultato finale dipende dall'ordine di esecuzione
 - il perdente della gara è il processo che aggiorna per ultimo (il primo lavora con i valori originari), e che determinerà i valori finali
- vii. Starvation: situazione in cui un processo in stato ready ma non viene scelto per un tempo indefinito dallo scheduler, quindi non riesce ad evolvere

Multiprogrammazione → i processi vengono alternati nel tempo (interleaving) → illusione di un'esecuzione contemporanea

Multiprocessing → alternanza + sovrapposizione (overlapping).

I problemi delle diverse tecniche sono sostanzialmente le stesse.

Problema multiprogrammazione: impossibile prevedere la velocità di esecuzione dei processi → dipende dall'attività degli altri processi, dalla gestione degli interrupt del SO e dalla politica di schedulazione → 3 difficoltà:

1. Condivisione delle risorse globali
2. Difficoltà del SO di gestire l'assegnazione di risorse in maniera ottimale
3. Difficoltà nel trovare errori di programmazione poiché i risultati non sono riproducibili

Problemi di progettazione del SO legati alla concorrenza:

- ◊ SO deve tenere traccia dei processi attivi
- ◊ SO deve allocare e deallocare varie risorse per ogni processo attivo
- ◊ SO deve proteggere i dati e le risorse fisiche di ogni processo da interferenze di altri processi
- ◊ SO deve garantire che il processo e il risultato siano indipendenti dalla velocità di esecuzione

Interazione tra processi in relazione all'accesso a risorse condivise → classificazione in base al grado di conoscenza che hanno i processi dell'esistenza di altri:

1. Processi che non si vedono tra loro → relazione: COMPETIZIONE → il risultato di un processo è indipendente dall'azione degli altri ma l'esecuzione di uno può rallentare, e quindi influenzare, quella degli altri → problemi di controllo: USO MUTUA ESCLUSIÓN, ELUSIÓN DEADLOCK, STARVATION
2. Processi che ne vedono altri indirettamente → rel: COOPERAZIONE TRAMITE CONDIVISIONE → i processi sanno che le modifiche ai dati condivisi possono influenzare altri quindi cooperano per garantire integrità → prob: USO MUTUA ESCLUSIÓN (solo in caso di scrittura), ELUSIÓN DEADLOCK, STARVATION e COERENZA DEI DATI
3. Processi che ne vedono altri direttamente → rel: COOPERAZIONE TRAMITE COMUNICAZIONE → avviene una comunicazione diretta tra i processi quindi la sincronizzazione viene semplificata → ELUSIÓN DEADLOCK e STARVATION

Requisiti per la mutua esclusione:

- A. La mutua esclusione deve essere garantita: un solo processo alla volta può entrare in zona critica
- B. Un processo in attesa e quindi fuori dalla sezione critica non deve interferire con gli altri

- C. Non devono esserci stalli o starvation
- D. Se la zona critica è libera, un processo può entrarci senza attese
- E. Non si devono fare supposizioni sulla velocità relativa ai processi o al loro numero
- F. Il tempo di permanenza nella sezione critica deve essere finito

Approcci Hardware per garantire la Mutua Esclusione (m. e.):

- DISABILITARE L'INTERRUZIONE DEI PROCESSI (INTERRUPT DISABLING) → realizzato con apposite primitive del kernel → svantaggi: l'efficienza potrebbe peggiorare, approccio che non funziona nei sistemi con multiprocessori
- ISTRUZIONI MACHINA SPECIALI → effettuano due azioni in modo atomico, quindi non interferiscono con altre istruzioni, (lettura + scrittura o lettura + test) su una locazione di memoria → più comuni:

Compare&Swap

```
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;
}
```

Compare: compara valore di memoria (reg) con il valore del test (oldval)

se è lo stesso → swap: cambia il valore in memoria con il nuovo valore (newval)

In questa versione ritorna il vecchio valore del registro, altrimenti potrebbe tornare un valore booleano (true se è avvenuto lo swap, false altrimenti).

La figura di sotto riporta un protocollo per la mutua esclusione → solo un processo troverà la variabile condivisa bolt=0 e sarà quello ad entrare nella sezione critica. Gli altri entrano in un ciclo che li mette in attesa → entrano in BUSY WAITING (o SPIN WAITING) MODE, cioè continuano a girare sulla CPU ma in pratica non fanno nulla.

```
/* program mutual exclusion */
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));
}
```

Exchange Instruction

- Exchange instruction

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

Cambia il contesto del registro con quello di una locazione di memoria.

La figura di sotto mostra il protocollo di mutua esclusione usando questa istruzione → la variabile condivisa bolt è settata a 0 invece quella locale key ad 1 → il processo che entra in zona critica è quello che trova bolt=0 → per fermare gli altri scambia i valori tra key e bolt → quando ha finito risetta bolt a 0

```

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

```

Vantaggi delle Istruzioni Macchina:

- Applicabile ad un numero indefinito di processi sia su macchine con singolo processore sia con multiprocessore
- Facile da verificare
- Può essere usato per più sezioni critiche, ognuna delle quali ha una propria variabile

Svantaggi:

- È implementata la Busy Waiting → quando un processo è in attesa comunque consuma il tempo del processore
- È sempre possibile la Starvation
- È sempre possibile il Deadlock

Semafori

- Meccanismo per gestire la concorrenza
- Principio fondamentale: due o più processi possono cooperare attraverso semplici segnali, in modo che uno si fermi ad una locazione di memoria ben precisa fino a che non riceva un segnale specifico.
- Il semaforo è una variabile che ha un valore interno e tre operazioni associate (non esiste altro modo per leggere o modificare i semafori):
 1. Inizializzazione con un valore non negativo
 2. Ricezione segnale semWait(s) che decrementa il valore del semaforo → se valore diventa negativo il processo che sta eseguendo questa operazione viene bloccato
 3. Trasmissione grazie a semSignal(s) che incrementa il valore del semaforo → se il valore non è positivo allora uno dei processi bloccati sulla semWait(s) viene sbloccato

```

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 */;
    }
}

```

- semWait e semSignal sono operazioni atomiche
- Per la loro implementazione si usa una coda che contiene i processi in attesa. Per sbloccarli se si usa una politica FIFO allora parleremo di semafori forti (viene evitata la starvation), altrimenti, se non è specificato, si definiranno semafori deboli
- Conseguenze: 1. Non c'è modo di sapere a priori se il processo che decrementerà il semaforo sarà bloccato oppure no
2. Non si sa quale processo continua immediatamente su un sistema con processore singolo se sono presenti due processi che concorrono
3. Il numero dei processi sbloccati, a seguito di un segnale, potrebbe essere 0 o 1

- Esistono dei particolari semafori, chiamati Binari, che settano il valore del semaforo o a 0 o a 1

```

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 {
        /* removes a process P from s.queue */;
        /* places process P on ready list */;
    }
}

```

- Utilizzo dei semafori per risolvere il problema della mutua esclusione (evitando Busy Waiting):

```

/* program mutual_exclusion */
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));
}

```

1. Viene eseguita la semWait prima della sezione critica (s.c.) → se il valore s è negativo allora il processo è sospeso → se è 1 allora diventa 0, quindi il processo successivo verrà bloccato, s diventa -1 → non esiste un numero limite di processi che prova ad entrare nella s. c.

2. Quando il processo in s.c. esce, esegue un semSignal che incrementa s di 1 e uno di quelli in attesa viene sbloccato, sarà lui ad entrare in s.c.

- Problema Produttore/Consumatore

Formulazione generale:

- Ci sono uno o più produttori che generano dati e li inseriscono in un buffer
- C'è un singolo consumatore che prende un elemento alla volta dal buffer
- Un solo produttore o consumatore può accedere al buffer
- bisogna assicurarsi che un produttore non aggiunga dati in un buffer pieno e un consumatore non li rimuova in uno vuoto

Si assume di avere un buffer di dimensione infinita

Soluzione incorretta del problema usando i semafori binari:

```

/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
    while (true) {
        produce();
        semWaitB(s);
        append();
        n++;
        if (n==1) semSignalB(delay);
        semSignalB(s);
    }
}
void consumer()
{
    semWaitB(delay);
    while (true) {
        semWaitB(s);
        take();
        n--;
        semSignalB(s);
        consume();
        if (n==0) semWaitB(delay);
    }
}
void main()
{
    n = 0;
    parbegin (producer, consumer);
}

```

- > Si tiene traccia degli elementi nel buffer tramite n=in-out
- > Vengono usati due semafori: s (per mutua esclusione) e delay (per forzare il consumatore ad attendere se il buffer è vuoto)
- > Il produttore aggiunge elementi in qualsiasi momento, solitamente è più veloce quindi non blocca il consumatore tramite il semaforo delay, poiché n sarà positivo
- > Difetto: se il consumatore svuota il buffer, deve aspettare il produttore che aggiunga dati
- Soluzione Corretta del problema usando semafori binari:
- > Viene usata una variabile ausiliaria assegnata nella s.c. del consumatore ed usata dopo

```

/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
    while (true) {
        produce();
        semWaitB(s);
        append();
        n++;
        if (n==1) semSignalB(delay);
        semSignalB(s);
    }
}
void consumer()
{
    int m; /* a local variable */
    semWaitB(delay);
    while (true) {
        semWaitB(s);
        take();
        n--;
        m = n;
        semSignalB(s);
        consume();
        if (m==0) semWaitB(delay);
    }
}
void main()
{
    n = 0;
    parbegin (producer, consumer);
}

```

> In questa situazione non avviene il deadlock

Soluzione Corretta del problema usando semafori generali (o semafori a contatore):

> n viene visto non come una variabile ma come un semaforo

```
/* 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();
    }
}
void main()
{
    parbegin (producer, consumer);
}
```

> Come si nota se vengono scambiate le semSignal non succede nulla, ma se le semWait vengono eseguite in ordine diverso, il consumatore accede al buffer anche se vuoto, creando lo stallo

Soluzione del problema con un BUFFER DI DIMENSIONE FINITA:

> Il buffer è un sistema di memorizzazione circolare e i valori dei puntatori sono espressi modulo la dimensione del buffer

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n = 0, e = sizeofbuffer;
void producer()
{
    while (true) {
        produce();
        semWait(e);
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
void consumer()
{
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        semSignal(e);
        consume();
    }
}
void main()
{
    parbegin (producer, consumer);
}
```

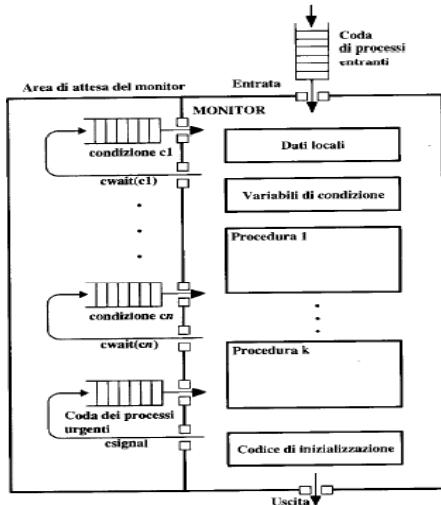
- Implementazione dei Semafori:

- semWait e semSignal devono essere implementate come operazioni atomiche
- possono essere implementate in hw o firmware
- si potrebbero usare alcuni algoritmi sw come quello di Dekker o Peterson
- può essere usato uno schema hw per la mutua esclusione (compare&switch)

Monitor

- Costrutto dei linguaggi di programmazione (Java, Concurrent-Pascal, Pascal Plus) che ha le medesime funzionalità dei semafori ma è più semplice implementarli e individuare eventuali errori
- Contiene una o più procedure, una sequenza di inizializzazione e dati locali → caratteristiche:

- Le variabili locali sono accessibili solo dalle procedure del monitor e non da quelle esterne
 - Un processo entra nel monitor chiamando una delle sue procedure
 - Solo un processo alla volta può entrare nel monitor, gli altri vengono sospesi (\rightarrow garantisce la mutua esclusione)
- Essendo garantita la mutua esclusione, le risorse critiche, quindi quelle da proteggere, vengono inserite semplicemente nei monitor
 - Tramite VARIABILI DI CONDIZIONE viene fornita la sincronizzazione: un processo nel monitor che si blocca perché in attesa di una condizione, lo rilascia per far entrare altri, ma quando ritorna disponibile viene fatto rientrare nello stesso punto in cui era stato sospeso \rightarrow funzioni su queste variabili:
 - cwait(c): sospende l'esecuzione del processo chiamante sulla condizione c \rightarrow il monitor torna disponibile
 - csignal(c): riattiva il processo sospeso sulla condizione c \rightarrow se ce ne sono molti, ne sceglie uno, se non ce ne è nessuno, non fa nulla
 - Struttura di un monitor:



- Soluzione Problema Produttore/ Consumatore con buffer limitato usando i monitor:

```

/* 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 */

void append (char x)
{
    if (count == N) cwait(notfull);           /* buffer is full; avoid overflow */
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;
    /* one more item in buffer */
    csignal(notempty);                      /* resume any waiting consumer */
}
void take (char x)
{
    if (count == 0) cwait(notempty);          /* buffer is empty; avoid underflow */
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;
    /* one fewer item in buffer */
    csignal(notfull);                      /* resume any waiting producer */
}
{
    nextin = 0; nextout = 0; count = 0;        /* monitor body */
    /* buffer initially empty */
}

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

```

- > variabili condizione: notfull (vera quando nel buffer c'è spazio), notempty (se è contenuto almeno un carattere)
- > il produttore non può accedere al buffer direttamente, ma solo tramite la funzione del monitor append, che per prima cosa verifica se c'è spazio per aggiungere elementi, altrimenti viene messo in attesa. Quando aggiunge un dato viene segnalato notempty → il consumatore si comporta similmente
- I monitor garantiscono di per sé la mutua esclusione, anche se il programmatore deve decidere dove mettere le cwait e csignal; nei semafori, invece, sia mutua esclusione che sincronizzazione, dipendono da chi scrive il codice

Scambio di messaggi

L'interazione tra processi implica due requisiti:

1. Sincronizzazione → per garantire la mutua esclusione
2. Scambio di informazioni → affinché cooperino

Il passaggio di messaggi li soddisfa

Supporto di scambio dato da due primitive: send(destinazione, messaggio) e receive(sorgente, messaggio)

Sincronizzazione: il ricevente non può acquisire un messaggio prima che il mittente lo abbia inviato

Send e Receive possono essere rispettivamente bloccanti o non; le loro combinazioni più usate sono:

1. Send Bloccante e Receive Bloccante: sia mittente che ricevente sono bloccati fino al completamento dell'operazione → sincronizzazione stretta tra processi
2. Send Non Bloccante e Receive Bloccante: il ricevente è bloccato fino all'arrivo del messaggio → più utile perché mittente può inviare vari messaggi a diversi destinatari
3. Send Non Bloccante e Receive Non Bloccante: nessuno dei due deve aspettare

Per la programmazione concorrente la più naturale è la send non bloccante → svantaggi: se un processo continua a generare messaggi, potrebbe consumare risorse di sistema (spazio dei buffer e tempo del processore) a danno degli altri e del SO; il programmatore si deve preoccupare di controllare che i messaggi arrivino: i processi devono inviare una conferma ricezione.

Invece per la receive la più naturale è la versione bloccante → infatti un processo, che richiede un messaggio, necessita di avere quelle informazioni per progredire → svantaggi: se il messaggio viene perso o il processo fallisce prima di rispondere, esso rimane bloccato per sempre → se si sceglie, però, la versione non bloccante, il processo continua l'esecuzione e quindi ogni messaggio inviato dopo la receive, verrà perso → altro modo è di far controllore al processo se c'è un messaggio in arrivo prima di effettuare la receive o specificare più mittenti, utile soprattutto se è sufficiente una sola informazione per continuare l'esecuzione

Indirizzamento: esistono due tipi di indirizzamento:

1. Diretto: la send contiene un identificatore specifico del processo destinatario. La receive può essere gestita chiedendo di specificare esplicitamente il mittente oppure viene usato l'indirizzamento implicito cioè il parametro sorgente della receive serve per rispondere al mittente del messaggio quando l'operazione è compiuta
2. Indiretto: i messaggi sono mandati ad una struttura condivisa che si compone di code, chiamate caselle di posta → il processo invia il messaggio ad una casella e da lì il destinatario la preleva → maggiore flessibilità

Le relazioni tra mittenti e riceventi può essere:

- a. Uno – a – uno → comunicazione diretta per evitare interferenza con altri processi
- b. Molti – a – uno → struttura client/server → comunicazione indiretta ma la casella di posta viene detta porta
- c. Uno – a – molti e Molti – a – molti → comunicazione indiretta

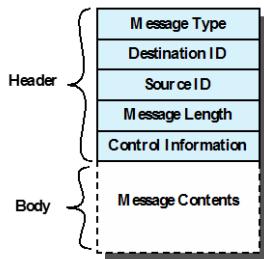
L'associazione dei processi alle caselle di posta può essere:

- ◊ Statica: una porta è associata permanentemente ad un processo
- ◊ Dinamica: scelta soprattutto nel caso ci siano più mittenti → utilizzate primitive connect e disconnect

Solitamente la porta appartiene al processo ricevente, che è anche quello che lo crea, quindi quando il processo termina, essa deve essere distrutta

Formato Messaggi:

- Per alcuni SO sono stati scelti messaggi corti di lunghezza fissata → si evita il sovraccarico e lo spazio richiesto
- Se è necessario inviare molti dati, basta salvarli in un file ed inviare solo il nome
- L'uso di messaggi di dimensione variabile permettono un approccio più flessibile
- Struttura del messaggio



Mutua esclusione

```

/* 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));
}

```

→ Viene utilizzata una receive bloccante e una send non bloccante, in più viene condivisa la casella di posta (box)

→ Box viene inizializzata con un messaggio vuoto

→ Un processo, prima di entrare in s. c., cerca di ricevere un messaggio, se vuoto viene bloccato → appena arriva il messaggio accede in s.c e al termine invia qualcosa alla casella → il messaggio diventa un permesso di entrata, un token

→ Se più processi eseguono la receive:

- se è presente un messaggio, solo uno eseguirà la receive e gli altri saranno bloccati

- se la coda dei messaggi è vuota, tutti i processi sono bloccati, appena arriva ne verrà attivato solo uno

Soluzione Problema Produttore/Consumatore con buffer limitato usando il passaggio di messaggi:

> Si usano due caselle:

1. Il produttore, quando genera i dati, li invia a mayconsumer → finché ci sono messaggi, il consumatore li può prendere → mayconsumer funziona da buffer e gli elementi sono memorizzati come una coda → capacità data dalla variabile capacity

2. L'altra casella è mayproducer → inizializzata con tanti messaggi vuoti quanto è grande → il numero diminuisce per ogni dato prodotto ed aumenta per ogni dato consumato

```

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);
}

```

> Approccio flessibile → ci possono essere molti produttori e consumatori ma tutti devono poter accedere ad entrambe le caselle → possibilità di utilizzo anche su sistemi distribuiti

Problema Lettore/Consumatore

- Formulazione: esiste un'area dati condivisa (file, blocco di memoria o banco di registri del processore) che i processi possono leggere (lettori) o scrivere (scrittori) → condizioni da soddisfare:
 1. Più lettori possono leggere i dati contemporaneamente
 2. Solo uno scrittore alla volta può modificare i file
 3. Se uno scrittore sta scrivendo, allora nessun lettore può leggerlo
- Caso generale: tutti i processi possono sia leggere che scrivere, ma è più interessante studiare quello specifico: gli scrittori non possono essere anche lettori e viceversa → si possono trovare soluzioni efficienti
- Il caso specifico non può essere paragonato al problema Consumatore/Produttore, poiché un produttore oltre a scrivere, deve leggere i puntatori della cosa per stabilire dove mettere il prossimo elemento e per verificare se il buffer è pieno (stessa cosa per il consumatore)
- Soluzione usando i semafori: priorità ai lettori

```
/* program readersandwriters */
int readcount;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true) {
        semWait (x);
        readcount++;
        if (readcount == 1) semWait (wsem);
        semSignal (x);
        READUNIT();
        semWait (x);
        readcount--;
        if (readcount == 0) semSignal (wsem);
        semSignal (x);
    }
}
void writer()
{
    while (true) {
        semWait (wsem);
        WRITEUNIT();
        semSignal (wsem);
    }
}
void main()
{
    readcount = 0;
    parbegin (reader, writer);
}
```

> Usa un solo lettore e uno scrittore, ma non cambia se ne sono presenti di più

> Il semaforo wsem serve per garantire la mutua esclusione invece x serve per assicurarsi che il readcount (numero di lettori che sta leggendo i dati) sia correttamente aggiornata

- Soluzione usando i semafori: priorità agli scrittori

```

/* program readersandwriters */
int readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true) {
        semWait (z);
        semWait (rsem);
        semWait (x);
        readcount++;
        if (readcount == 1) semWait (wsem);
        semSignal (x);
        semSignal (rsem);
        semSignal (z);
        READUNIT();
        semWait (x);
        readcount--;
        if (readcount == 0) semSignal (wsem);
        semSignal (x);
    }
}
void writer ()
{
    while (true) {
        semWait (y);
        writecount++;
        if (writecount == 1) semWait (rsem);
        semSignal (y);
        semWait (wsem);
        WRITEUNIT();
        semSignal (wsem);
        semWait (y);
        writecount--;
        if (writecount == 0) semSignal (rsem);
        semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}

```

> Nella soluzione precedente, un lettore, quando inizia ad accedere ai dati, può mantenere il controllo dell'area fino a che non si disattiva → gli scrittori potrebbero aspettare indefinitivamente

> In questa, un lettore non può accedere all'area dati se uno scrittore ha chiesto di effettuare una modifica

> Sono stati aggiunti:

- semaforo rsem → blocca i lettori se c'è uno scrittore che si è prenotato
- variabile writecount → controlla il numero di scrittori prenotati
- semaforo y → permette di aggiornare correttamente writecount
- semaforo z → inserito per non creare una lunga coda su rsem

- Soluzione usando scambio di messaggi: priorità agli scrittori

> C'è un processo controllore che ha accesso ai dati condivisi

> Procedura dei processi per accedere in s.c.: inviano richiesta al controllore → aspettare un messaggio di risposta di via libera → spedire un messaggio "ho finito" quando hanno completato l'accesso

> Il controllore serve prima le richieste di scrittura, poi di lettura

> Mutua esclusione garantita dalla variabile count:

- se >0 → nessun scrittore è in attesa, potrebbero esserci lettori attivi
- se =0 → si può far procedere uno scrittore
- se <0 → uno scrittore ha fatto richiesta di accesso → controllore accetta dai lettori solo messaggi "ho finito"

```

void reader(int i)
{
    message rmsg;
    while (true) {
        rmsg = i;
        send (readrequest, rmsg);
        receive (mbox[i], rmsg);
        READUNIT ();
        rmsg = i;
        send (finished, rmsg);
    }
}
void writer(int j)
{
    message rmsg;
    while(true) {
        rmsg = j;
        send (writerequest, rmsg);
        receive (mbox[j], rmsg);
        WRITEUNIT ();
        rmsg = j;
        send (finished, rmsg);
    }
}

void controller()
{
    while (true)
    {
        if (count > 0) {
            if (!empty (finished)) {
                receive (finished, msg);
                count++;
            }
            else if (!empty (writerequest)) {
                receive (writerequest, msg);
                writer id = msg.id;
                count = count - 100;
            }
            else if (!empty (readrequest)) {
                receive (readrequest, msg);
                count--;
                send (msg.id, "OK");
            }
        }
        if (count == 0) {
            send (writer id, "OK");
            receive (finished, msg);
            count = 100;
        }
        while (count < 0) {
            receive (finished, msg);
            count++;
        }
    }
}

```