Sincronizzazione dei processi nei Sistemi Operativi

Dobbiamo trovare un modo per sincronizzare e far comunicare i processi fra di loro in maniera strutturata (Non usando gli interrupt).

> ⓘ **La comunicazione fra processi è detta anche IPC (InterProcess Comunication).**

Sono tre i problemi da affrontare:

1. Far **passare informazioni** da un processo all'altro.
2. Evitare che processi diversi si intralcino.
3. Esecuzione corretta quando ci sono dipendenze.

Questi problemi valgono anche per i thread (Il primo è più semplice, gli altri sono complessi).

Race Condition

E' un tipo di errore che avviene quando due o più processi cercano di accedere in modo concorrente ad una risorsa condivisa, e il risultato finale dipende dall'ordine i cui i processi sono eseguiti. I processi "gareggiano" tra loro e giungono a conclusioni errate.

Critical Region

Per evitare le race condition dobbiamo trovare un modo per proibire a più processi di leggere e scrivere dati contemporaneamente.

> ⓘ **Mutua Esclusione**
>
> Sistema che impedisca ad un processo A di accedere ad una determinata risorsa se un processo B sta accedendo alla stessa risorsa.

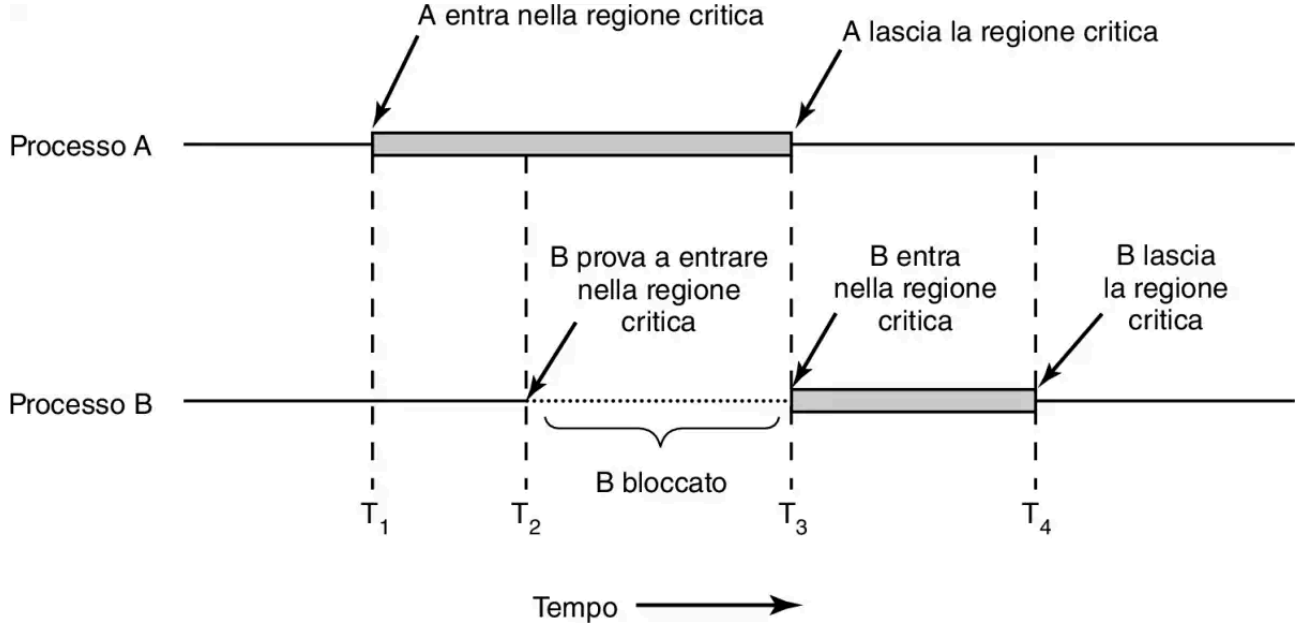
Per risolvere il problema dobbiamo fare in modo che più processi non entrino contemporaneamente nelle **regioni critiche**.

> ⚠ **Regione critiche**

Parte di programma che accede alla memoria condivisa

Per ottenere una buona soluzione dobbiamo mantenere vere quattro condizioni:

1. Non devono mai esserci due processi contemporaneamente nelle loro regioni critiche.
2. Non può essere fatta alcuna supposizione sulle velocità o sul numero delle CPU.
3. Nessun processo in esecuzione al di fuori della propria regione critica può bloccare altri processi.
4. Nessun processo deve aspettare all'infinito per entrare nella propria regione critica.



NON soluzioni:

- **Disabilitare gli interrupt:** Impedisce che la CPU possa essere riallocata, inoltre funziona solo per sistemi a CPU singola.
- **Bloccare le variabili:** Proteggere le regioni critiche con variabili 0/1. Le "corse" ora si verificano sulle variabili di blocco.

Mutua Esclusione

Ricorda

I Sistemi Operativi sono proncipalemente scritti in C perché è potente, efficiente e prevedibile.

Il primo approccio ad una soluzione è il seguente:

```
while (TRUE) {
  while (turn != 0) { }  /* ciclo */
  critical region( );
  turn = 1;
  noncritical region( );
}
(a)

while (TRUE) {
  while (turn != 1) { }  /* ciclo */
  critical region( );
  turn = 0;
  noncritical region( );
}
(b)
```

Prima di poter entrare una regione critica, la variabile turn, controlla il suo stato, se:

- 0 possiamo entrare nella regione critica del processo b;
- 1 possiamo entrare nella regione critica del processo a;

Questo approccio si chiama **busy waiting**, il che andrebbe evitato perché consuma tempo di CPU, ma si può usare se sappiamo che l'attesa dei processi è breve.

> 🖊 **Un lock che utilizza il busy waiting si chiama spinlock**

L'unico problema di questo approccio è che viola la terza regola che ci siamo imposti, infatti:

1. Non permette i processi di entrare nelle regioni critiche più volte.
2. **Un processo fuori dalla regione critica può bloccarne un altro.**

Algoritmo di Peterson

```
#define FALSE  0
#define TRUE  1
#define N  2      /* numero di processi */

int turn;        /* A chi tocca? */
int interested[N];      /* Tutti i valori inizialmente 0 (FALSE) */

void enter_region(int process);    /* process è 0 o 1 */
{
  int other;        /* numero dell'altro processo */
```

```
  other = 1 - process;    /* l'opposto del processo */
  interested[process] = TRUE;    /* mostra che si è interessati */
  turn = process;      /* imposta il flag */
  while (turn == process && interested[other] == TRUE) /* istruzione null */
;
}
void leave_region(int process)    /* process: chi esce */
{
  interested[process] = FALSE;  /* indica l'uscita dalla regione critica */
}
```

Funzionamento:

1. Ogni processo che vuole entrare imposta nella regione critica imposta il propirio flag a **TRUE**.
2. Il processo assegnaa turn il valore dell'altro processo.
3. Dopo avere impostato la flag il processo entra in attesa in cui verifica le condizioni:

- Se l'altro processo ha il flag impostato su TRUE;
- Se è il turno dell'altro processo, quindi si deve aspettare;

4. Quando le condizioni sono soddisfate si può entrare nella regione critica.
5. Quando si finisce si imposta il flag su FALSE, indicando che ha finito.

Problema del Produttore e Consumatore

Il problema delle soluzioni che abbiamo trovato è che richiedono il busy waiting.

> **Busy waiting**
>
> Il busy waiting (attesa attiva) è una tecnica in cui un processo rimane in un ciclo di attesa, continuamente controllando una condizione, senza fare nulla di utile nel frattempo occupando la CPU inutilmente. (Spin Lock).

La soluzione al problema è la seguente: Quando un processo è in attesa di entrare nella regione critica questo restituisca volontariamente la CPU allo scheduler.
Questa idea si può implementare tramite due funzioni primitive:

```
void sleep(){
        //set own state to BLOCKES
        //five CPU to scheduler
```

```
}

void wakeup(process){
        //set state of process to READY
        //gve CPU to scheduler
}
```

Problema del produttore-consumatore

Classico problema sulla sincronizzazione tra processi. Abbiamo due processi, uno *produttore* ed uno *consumatore* che condividono un buffer in comune. Compito del produttore è generare dati e depositarli nel buffer in continuo. Contemporaneamente, il consumatore utilizzerà i dati prodotti, rimuovendoli di volta in volta dal buffer. Il problema è assicurare che il produttore non elabori nuovi dati se il buffer è pieno, e che il consumatore non cerchi dati se il buffer è vuoto.

Ricapitolando:

- Il produttore inserisce dati nel buffer.
- Il consumatore prende i dati dal buffer.

Sono 3 i problemi che dobbiamo risolvere:

1. Gestione condivisa della memoria, infatti più processi potrebbero voler entrare nel buffer.
2. Controllare se il buffer si riempe, non vogliamo che i produttori inseriscano dati sul buffer pieno.
3. Controllare se il buffer è vuoto, non vogliamo che i consumatori prendano dati da un buffer vuoto.

```
#define N 100 /* numero di posti nel buffer */
int count = 0; /* numero di elementi nel buffer */

void producer(void) {
    int item;
    while (TRUE) { /* ripeti sempre */
        item = produce_item(); /* genera l'elemento successivo */

        if (count == N) {
            sleep(); /* se il buffer è pieno, vai a dormire */
        }

        insert_item(item); /* metti l'elemento nel buffer */
```

```
        count = count + 1; /* incrementa il numero di elementi nel buffer */
        if (count == 1) {
            wakeup(consumer); /* il buffer era vuoto? */
        }
    }
}

void consumer(void) {
    int item;
    while (TRUE) { /* ripeti sempre */
        if (count == 0) {
            sleep(); /* se il buffer è vuoto, vai a dormire */
        }

        item = remove_item(); /* togli un elemento dal buffer */
        count = count - 1; /* decrementa il numero di elementi nel buffer */

        if (count == N - 1) {
            wakeup(producer); /* il buffer era pieno? */
        }

        consume_item(item); /* stampa l'elemento */
    }
}
```

Adesso **questa soluzione produce una race condition fatale**.

Infatti se il buffer è vuoto e il consumatore ha verificato se count è a 0, lo scheduler decide di fermare il consumatore e di far partire il produttore.
Il produttore inserisce qualcosa nel buffer e, rilevando che count è aumentato, suppone che il consumatore sia in sleep, quindi il produttore chiama wakeup.
Il consumatore però non è ancora in sleep, quindi si perde il segnale di wakeup.
Quando poi il consumatore sarà eseguito leggerà il count è troverà 0 è andrà in sleep.
Prima o poi il produttore riempirà il buffer è andrà a dormire, lasciandoci in una situazione in cui entrambi **i processi stanno dormendo (deadlock).**

La causa del deadlock è che, in alcuni casi, il **segnale di risveglio** (`wakeup`) del consumatore viene perso, causando la situazione in cui entrambi i processi si trovano in uno stato di attesa reciproca:

- Il **produttore** si addormenta quando il buffer è pieno, senza poter svegliare il consumatore.

- Il **consumatore** si addormenta aspettando che ci siano elementi nel buffer, ma non è stato correttamente svegliato quando il produttore ha messo il primo elemento nel buffer.

In questo modo, i due processi finiscono in un **deadlock**, poiché nessuno dei due può procedere.

Una soluzione sarebbe quella di usare un **bit di attesa del wakeup**, cioè un bit che viene impostato quando viene inviato un *wakeup* ad un processo non dormiente.
Quando il processo vuole entrare in sleep se il bit di wakeup è acceso il processo lo spegne e rimane sveglio.

> **Il consumatore ripulisce il bit di attesa wakeup a ogni iterazione del ciclo**

Semafori

Un semaforo è un semplice valore intero non negativo e condiviso tra thread che ci permette di andare a gestire la sincronizzazione dei processi tramite le chiamate di wakeup.

> **Operazioni atomiche di un semaforo**
>
> - wait ( ) o down ( ) --> P (Dalla parola olandese proberen, significa testare).
> - signal ( ) o Up ( )--> V (Dalla parola olandese verhogen, significa incrementare).

I valori che un semaforo può assumere sono:

- 0 nelle situazioni in cui non ci sta nessun wakeup da gestire.
- 1 nelle situazioni in cui ci sta un wakeup in attesa.

> **down ( )**
>
> Se il valore del semaforo è maggiore di zero, questo valore viene decrementato, e il processo continua la sua esecuzione.
> Se il valore sel semaforo è 0, il processo che ha invocato down viene bloccato e messo in coda di attesa associata al semaforo.

> **Up ( )**
>
> Se il valore è 0, ci sono processi nella coda di attesa, vengono "svegliati", in ogni caso il valore viene incrementato e il processo continua la sua esecuzione.

Esistono diversi tipi di semafori:

- **mutex**
- **full**
- **empty**

Il primo usato per prevenire accessi simulateni, gli altri servono a coordinare le attività.

Lettori e scrittori

> ⓘ **Problema dei lettori-scrittori**
>
> Problema simile a quello precedente dove abbiamo tanti processi che devono accedere ad un database, ovviamente più processi possono entrare contemporaneamente, ma nel momenti in cui un processo scrive qualcosa gli altri devono fermarsi.
> L'obiettivo è garantire che:
>
> 1. **Più lettori** possano leggere simultaneamente, ma **solo un scrittore** possa scrivere alla volta.
> 2. **L'accesso esclusivo** per la scrittura viene concesso solo a uno scrittore alla volta, mentre i lettori non possono accedere al database quando un scrittore sta scrivendo.
> 3. **Mutua esclusione** è garantita in vari punti per evitare che lettori e scrittori accedano simultaneamente al database in modo incoerente.

```
typedef int semaphore; /* usate l'immaginazione */

semaphore mutex = 1; /* controlla l'accesso a rc */
semaphore db = 1;    /* controlla l'accesso al database */
int rc = 0;          /* # di processi che leggono o vogliono leggere */

void reader(void) {
    while (TRUE) { /* ripeti per sempre */
        down(&mutex);     /* ottieni accesso esclusivo a rc */
        rc = rc + 1;      /* un lettore in più */
        if (rc == 1)
            down(&db);    /* se questo è il primo lettore ... */

        up(&mutex);       /* rilascia accesso esclusivo a rc */
        read_data_base(); /* accedi ai dati */
        down(&mutex);     /* ottieni accesso esclusivo a rc */
        rc = rc - 1;      /* un lettore in meno */
        if (rc == 0)
```

```
            up(&db);      /* se questo è l'ultimo lettore ... */

        up(&mutex);       /* rilascia accesso esclusivo a rc */

        use_data_read();  /* regione non critica */
    }
}

void writer(void) {
    while (TRUE) { /* ripeti per sempre */
        think_up_data();    /* regione non critica */
        down(&db);          /* ottieni accesso esclusivo */
        write_data_base();  /* aggiorna i dati */
        up(&db);            /* rilascia accesso esclusivo */
    }
}
```