# Sistemi Operativi

Ionut Zbirciog

19 October 2023

## Sincronizzazione e Comunicazione tra Processi

I processi hanno bisogno di un modo per comunicare in modo da condividere i dati e sincronizzarsi con altri processi durante l'esecuzione. Molto brevemente, i problemi sono 3. Il primo è relativo al modo in cui un processo può passare informazioni ad un altro. Il secondo è accettarsi che due o più processi o thread non si intralcino a vicenda. Il terzo riguarda la corretta sequenzialità quando vi sono delle dipendenze: se il thread A produce dei dati e il thread B li stampa, B deve attendere che A abbia prodotto qualche dato prima di iniziare a stampare.

### Race Conditions

I processi possono lavorare insieme condividendo una parte di memorizzazione comune che ciascuno può leggere e scrivere. Questa memorizzazione condivisa può trovarsi nella memoria principale o può essere un file condiviso. Situazioni in cui due o più processi leggono o scrivono i medesimi dati condivisi e il risultato finale dipende dai tempi precisi in cui vengono eseguiti, sono chiamate *race conditions*. Ovvero, i processi "corrono" insieme per ottenere l'accesso ad una risorsa.

### Regione Critica

Per evitare le race conditions serve una mutua esclusione, ossia un qualche sistema per essere certi che, se un processo sta usando un variabile o un file condiviso, agli altri processi venga impedito di fare la stessa cosa. La parte di programma in cui accede alla memoria condivisa è chiamata regione critica o sezione critica, quindi la soluzione alle race conditions è di non fare accedere due processi/thread contemporaneamente nella regione critica. Per ottenere una buona soluzione servono 4 condizioni da rispettare:

1. Due processi non possono trovarsi contemporaneamente all'interno delle rispettive regioni critiche
2. Non si possono fare ipotesi sulla velocità o sul numero di 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

## Mutua Esclusione con Busy Waiting

### (a) Disabilitare gli Interrupt

Disabilitare gli interrupt di un processo appena entrato nella sua regione critica e li riabiliti quando ne esce. In questo modo non c'è la possibilità che un altro processo entri nella regione critica, poiché la riallocazione della CPU avviene ad ogni interrupt di clock e di altri interrupt. È una tecnica che può essere usata solo dal kernel per aggiornare alcuni dati critici, ma non è opportuno dare la possibilità ai processi utente di disabilitare gli interrupt. Funziona solo per sistemi a CPU singola (single core), poiché nei sistemi multicore, se gli interrupt di un core vengono disabilitati, gli altri core possono comunque interferire.

### (b) Bloccare le Variabili

Proteggere le regioni critiche con variabili 0/1. Le 'corse' si verificano ora sulle variabili di blocco.

### (c) Alternanza Stretta

La variabile *turn*, inizialmente a 0, tiene traccia dei turni dei processi che vogliono entrare nella regione critica. Il processo A vede che *turn* è 0, quindi entra nella regione critica impostando *turn* a 1. Nel mentre il processo B vede che *turn* è 1, quindi si mette in attesa in un rapido ciclo. L'azione di testare continuamente una variabile finché non è valorizzata si chiama *busy waiting*. Andrebbe generalmente evitato, dato che consuma tempo alla CPU. Di solito si usa quando l'attesa è relativamente breve.

- Processo A:

```
while(TRUE){
    while(turn != 0);
    critcal_region();
    turn = 1;
    noncritical_region();
}
```

- Processo B:

```
while(TRUE){
    while(turn != 1);
    critcal_region();
    turn = 0;
    noncritical_region();
}
```

Purtroppo, questa è un'altra non soluzione perché: - Non permette ai processi di entrare nelle loro regioni critiche per due volte di seguito. - Un processo fuori dalla regione critica può effettivamente bloccarne un altro violando la 3 condizione.

### (d) Peterson's Algorithm

Alice e Bob vogliono usare un'unica postazione computer in un ufficio. Ma ci sono delle regole:

1. Solo una persona può usare il computer alla volta.
2. Se entrambi vogliono usarlo contemporaneamente, devono decidere chi va per primo.

**Idea:** Alice e Bob devono segnalare il loro interesse a usare il computer. Se l'altro non è interessato, la persona interessata può usarlo subito. Se entrambi mostrano interesse, registrano il loro nome su un foglio. Ma se scrivono quasi allo stesso tempo, l'ultimo nome sul foglio ha precedenza. Perché: garantisce che sempre uno avrà scritto e l'ultimo sarà quello che si prende la risorsa. La persona che non ha la precedenza aspetta finché l'altra ha finito. Una volta finito, la persona che ha usato il computer segnala che ha finito, e l'altra può iniziare.

```
#define N 2

int turn;
int interested[N];

void enter_region(int process){
    int other;
    other = 1 - process;
    interested[process] = TRUE;
    turn = process;
    while(turn == process && interested[other] == TRUE);
}

void leave_region(int process){
    interested[process] = FALSE;
}
```

### (e) TSL e XCHG

**Istruzione TSL (Test and Set Lock)**

È presente in computer con più processori. Legge il contenuto della memoria "lock", salva un valore non zero, e blocca altre CPU dall'accesso alla memoria. Purtroppo, anche disabilitando gli interrupt su un processore, non c'è garanzia che un processo "non faccia danni" da un'altra CPU e quindi, si bloccano tutti fino al termine dell'esecuzione di TSL. Viene fatto solo per operazioni atomiche, come per assegnare un valore.

**Funzionamento:** Quando *lock* è 0, un processo può impostare *lock* a 1 con TSL e accedere alla memoria condivisa. Al termine, il processo resetta *lock* a 0. Metodo per gestire Regioni Critiche:

- Processi chiamano *enter_region* prima di entrare nella regione critica e *leave_region* dopo.
- Se chiamati correttamente, garantisce la mutua esclusione.
- Se usati in modo errato, la mutua esclusione fallisce.

```
enter_region:
    TSL REGISTER, LOCK
    CMP REGISTER, #0
    JNE enter_region
    RET

leave_region:
    MOVE LOCK, #0
    RET
```

**Istruzione XCHG:** Scambia i contenuti di due posizioni atomicamente. Usata in tutte le CPU x86 Intel per sincronizzazione di basso livello.

## Sleep e Wakeup

Nonostante l'algoritmo di Peterson funzioni, rimane il problema dello spinlock causato dal busy waiting, dove il processo tiene occupata la CPU in attesa di poter entrare nella sua regione critica.

La soluzione consiste nel permettere al processo in attesa di entrare nella sua regione critica di restituire volontariamente la CPU allo scheduler.

```
void sleep(){
    set own state to BLOCKED;
    give CPU to scheduler;
}

void wakeup(process){
    set state of process to READY;
    give CPU to scheduler;
}
```

### Problema Produttore-Consumatore

Nel problema del produttore-consumatore, due processi condividono un buffer di dimensioni fisse. Il produttore inserisce informazioni nel buffer, mentre il consumatore le preleva.

```
#define N 100
int count = 0;

void producer(void){
    int item;
    while(TRUE){
        item = produce_item();
        if(count == N) sleep();
        insert_item(item);
        count++;
        if(count == 1) wakeup(cons);
    }
}
```

```
void consumer(void){
    int item;
    while(TRUE){
        if(count == 0) sleep();
        item = remove_item();
        count--;
        if(count == N - 1) wakeup(prod);
        consume_item(item);
    }
}
```

Problema: il produttore potrebbe svegliare il consumatore un attimo prima di andare in sleep, e nessuno lo risveglierebbe più perché non potendo più consumare, il produttore non sa se deve produrre o no.

## Semafori

Fondamentalmente, il semaforo è una variabile che può essere 0 (nessun wakeup) o un valore positivo (wakeup in attesa). Su questa variabile si possono eseguire le seguenti operazioni:

- `down`: Se il valore del semaforo è maggiore di zero, questo valore viene decrementato, e il processo continua la sua esecuzione. Se il valore del semaforo è 0, il processo che ha invocato `down` viene bloccato e messo in una coda di attesa associata al semaforo (va a dormire, `sleep()`).
- `up`: Se il valore è 0, ci sono processi nella coda di attesa che vengono "svegliati" (eventualmente per entrare in competizione ed eseguire di nuovo `down`). In ogni caso, il valore viene incrementato e il processo continua la sua esecuzione.

**Atomicità**: Le operazioni sui semafori sono "indivisibili", evitando conflitti.

### Problema Produttore-Consumatore

Utilizzo dei semafori per gestire l'accesso e la capacità di un buffer.

- `mutex` (mutual exclusion, accesso esclusivo).
- `full` (tutti i posti occupati).
- `empty` (tutti i posti liberi).

`mutex` previene eccessi simultanei, `full` e `empty` coordinano attività.

`{6.2_produce_consumer_semaphore.c}`

### Problema Scrittori e Lettori

Regola: Ci sono tanti lettori (consumer) e un solo scrittore (producer). Ad esempio, si possono avere molteplici letture su un database, ma solo un singolo scrittore.

**Funzionamento Sintetico**:

- Il primo lettore blocca l'accesso al database.
- Lettori successivi incrementano un contatore.
- L'ultimo lettore libera l'accesso al database così lo scrittore può svolgere il suo lavoro.

`{6.3_reader_writer_semaphore.c}`

I semafori sno usati per sincronizzare più processi tra loro.

## Mutex

Un *mutex* è una versione esplicita e semplificata dei semafori, usata per gestire la mutua esclusione di risorse o codice condiviso, quando non è necessario contare gli accessi e altri fenomeni. Può essere in due stati:

- **locked** (bloccato)
- **unlocked** (sbloccato)

Un solo bit può rappresentarlo, ma spesso viene utilizzato un intero (0 - unlocked, altri - locked). Due procedure principali sono `mutex_lock` e `mutex_unlock`:

- Quando un thread vuole accedere a una regione critica, chiama `mutex_lock`.
- Se il *mutex* è *unlocked*, il thread può entrare; se è *locked*, il thread attende.
- Al termine dell'accesso, il thread chiama `mutex_unlock` per liberare la risorsa.
- **IMPORTANTE**: Non si utilizza il *busy waiting*. Se un thread non può acquisire un lock, chiama `thread_yield` per cedere la CPU ad un altro thread.
- Alcuni pacchetti di thread offrono `mutex_trylock` che tenta di acquisire il lock o restituisce un errore senza bloccare. Questa opzione offre la possibilità di provare a prendere il lock, e se non è già occupato, continua ad eseguire altre operazioni senza sprecare tempo.

### Mutexes in pthread

La libreria Posix Pthread fornisce funzioni per la sincronizzazione tra thread. Il *mutex* è una variabile che può essere *locked* o *unlocked* ed è utilizzato per proteggere le regioni critiche. Il thread tenta di bloccare (*lock*) un mutex per accedere alla regione critica. Se il mutex è *unlocked*, l'accesso è immediato e atomico. Se è *locked*, il thread attende.

- `pthread_mutex_init`: Crea il mutex.
- `pthread_mutex_destroy`: Distrugge il mutex.

- `pthread_mutex_lock`: Acquisisce il mutex o si blocca.
- `pthread_mutex_trylock`: Acquisisce il mutex o fallisce.
- `pthread_mutex_unlock`: Rilascia il mutex.
- `pthread_cond_init`: Crea una variabile condizionale.
- `pthread_cond_destroy`: Distrugge una variabile condizionale.
- `pthread_cond_wait`: Si blocca in attesa di un segnale.
- `pthread_cond_signal`: Segnala un altro thread e lo sveglia.
- `pthread_cond_broadcast`: Segnala multipli thread e li sveglia.

**Semafori o Mutex?**

**Finalità**

- **Mutex**: È utilizzato principalmente per garantire l'esclusione mutua. È destinato a proteggere l'accesso a una risorsa condivisa, garantendo che solo un thread possa accedervi alla volta.
- **Semaforo**: Può essere usato per controllare l'accesso a una risorsa condivisa, ma è anche spesso usato per la sincronizzazione tra thread.

**Semantica**

- **Mutex**: Di solito ha una semantica di "proprietà", il che significa che solo il thread che ha acquisito il mutex può rilasciarlo.
- **Semaforo**: Non ha una semantica di "proprietà". Qualsiasi thread può aumentare o diminuire il conteggio del semaforo, indipendentemente da chi lo ha modificato l'ultima volta.

**Casistica**

- **Per l'esclusione mutua**: Un mutex è generalmente preferibile. È più semplice (di solito ha operazioni di lock e unlock) e spesso offre una semantica più rigorosa e un comportamento più prevedibile.
- **Per la sincronizzazione tra thread**: Un semaforo può essere più adatto, specialmente quando si tratta di coordinare tra diversi thread o di gestire risorse con un numero limitato di istanze disponibili.

`{6.4_produce_consumer_mutex.c}`

**NOTA**

- **Protezione della risorsa condivisa**: `pthread_mutex_lock` assicura che solo un thread alla volta possa accedere e modificare la risorsa condivisa.
- **Attesa condizionale**: Quando un thread chiama `pthread_cond_wait`, due operazioni avvengono atomicamente. Il thread rilascia il mutex e mette il thread in uno stato di attesa sulla variabile condizionale. Anche se il produttore ha acquisito il mutex, non lo detiene mentre è in attesa sulla variabile condizionale. Questo permette al consumatore (o a un altro thread) di acquisire il mutex, fare le sue operazioni, e poi mandare un segnale alla variabile condizionale usando `pthread_cond_signal`.

# Monitor

Un *monitor* raggruppa procedure, variabili e strutture dati. I processi possono chiamare le procedure di un monitor, ma non possono accedere direttamente alle sue strutture dati interne. Solo un processo può essere attivo in un monitor in un dato momento, garantendo la mutua esclusione. Il compilatore gestisce la mutua esclusione dei monitor, riducendo la probabilità di errori da parte del programmatore. Per gestire situazioni in cui i processi devono attendere, i monitor utilizzano variabili condizionali e due operazioni su di esse: `wait` e `signal`. A differenza dei semafori, le variabili condizionali non accumulano segnali; se un segnale viene inviato e non c'è un processo in attesa, il segnale viene perso. Linguaggi come Java supportano i monitor, permettendo una sincronizzazione e mutua esclusione più sicura e semplice in contesti multithreading. I metodi sono dichiarati `synchronized` in modo che solo un thread possa accedervi.

# Scambio di Messaggi

Metodo di comunicazione tra processi usando due primitive: `send(dest, &msg)` e `receive(src, &msg)`. Può essere utilizzato in diversi scenari, compresi sistemi distribuiti. Problemi includono messaggi persi in rete, necessità di feedback per confermare la ricezione, gestione dei messaggi duplicati usando numeri sequenziali, autenticazione e denominazione dei processi. Malgrado l'inaffidabilità, lo scambio di messaggi è cruciale nello studio delle reti. Nel problema del produttore-consumatore, si può utilizzare una soluzione senza memoria condivisa, basata solo su messaggi. Questa soluzione impiega un totale di N messaggi, simili ai N posti del buffer nella memoria condivisa. Il consumatore invia al produttore N messaggi vuoti, e il produttore prende un messaggio vuoto, lo riempie e lo invia.

## Barriere

Le barriere sono utilizzate per sincronizzare processi in fasi diverse. Quando un processo raggiunge una barriera, attende fino a quando tutti gli altri processi la raggiungono. Ad esempio, in calcoli paralleli su matrici, i processi non possono avanzare a un'iterazione successiva finché tutti non hanno terminato l'iterazione attuale. Sono utilizzate per sincronizzare processi in fasi diverse.

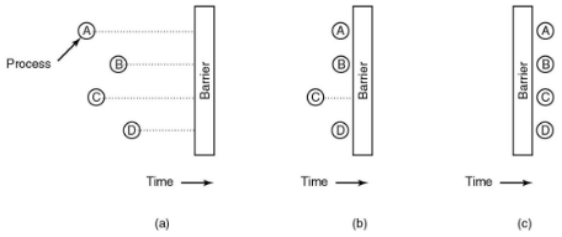


Figure 1: Barriere

## Inversione delle Priorità

Dati tre thread T1, T2, T3 con rispettive priorità p1 > p2 > p3, l'inversione delle priorità può verificarsi quando un thread con priorità più bassa detiene una risorsa che un thread con priorità più alta deve utilizzare. Soluzioni includono disattivare gli interrupt, *Priority Ceiling*, *Priority Inheritance*, e *Random Boosting*.

## Read-Copy-Update

L'obiettivo del *Read-Copy-Update* è di accedere in modo concorrente senza lock, cercando di evitare l'incosistenza dei dati. L'idea è di aggiornare strutture dati consentendo letture simulate senza incappare in versioni inconsistenti dei dati. I lettori vedono o la versione vecchia o quella nuova, mai un mix delle due. È diffuso nel kernel dei sistemi operativi.

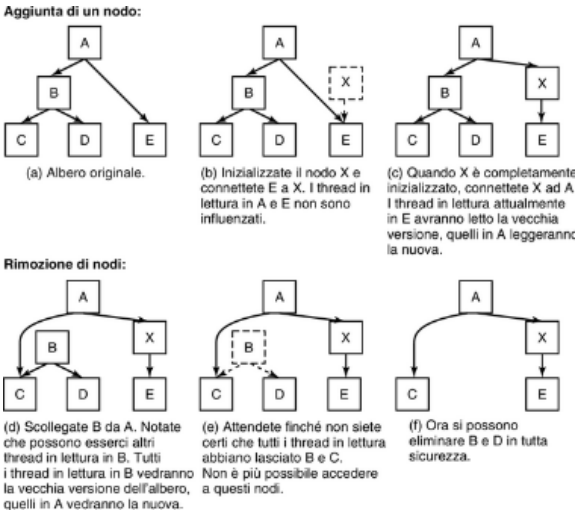


Figure 2: Inserimento e Cancellazione di un nodo