3 - Comunicazione e sincronizzazione

Sincronizzazione e Comunicazione tra Processi

I processi necessitano di metodi per comunicare, condividere dati e sincronizzarsi durante l'esecuzione. I problemi principali sono:

1. Come un processo può passare informazioni a un altro.
2. Assicurarsi che due o più processi o thread non si intralcino a vicenda.
3. Garantire la corretta sequenzialità quando vi sono dipendenze tra processi o thread.

Race Conditions

I processi possono condividere memorizzazione comune, che può essere nella memoria principale o in un file condiviso. Situazioni in cui due o più processi leggono o scrivono gli stessi dati e il risultato dipende dai tempi di esecuzione sono chiamate race conditions.

Regione Critica

Per evitare le race conditions è necessaria la mutua esclusione. La parte di programma che accede alla memoria condivisa è chiamata regione critica o sezione critica. La soluzione alle race conditions consiste nell'impedire a due processi/thread di accedere contemporaneamente alla regione critica. Quattro condizioni devono essere rispettate:

1. Due processi non possono trovarsi contemporaneamente nelle rispettive regioni critiche.
2. Non si possono fare ipotesi sulla velocità o sul numero di CPU.
3. Nessun processo 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 riabilitarli quando ne esce. Questa tecnica funziona solo per sistemi a CPU singola.

(b) Bloccare le Variabili

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

(c) Alternanza Stretta

La variabile `turn` tiene traccia dei turni dei processi. Un processo entra nella regione critica se il turno è il suo, altrimenti attende in un ciclo continuo (busy waiting) che andrebbe evitato perché consuma CPU. Questo metodo ha limiti significativi, come non permettere a un processo di entrare nella regione critica due volte di seguito.

```
// Processo A
while (TRUE) {
  while (turn != 0);
  critical_region();
  turn = 1;
  non_critical_region();
}

// Processo B
while (TRUE) {
  while (turn != 1);
  critical_region();
  turn = 0;
  non_critical_region();
}
```

(d) Peterson’s Algorithm

Un algoritmo che permette a due processi di condividere una risorsa senza interferenze. I processi segnalano il loro interesse a usare la risorsa e si alternano nell'accesso, garantendo mutua esclusione.

```
#define N 2
int turn;
int interested[N];

void enter_region(int process) {
  int 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)

Utilizzata in computer con più processori per garantire accesso atomico alla memoria condivisa. Se `lock` è 0, un processo può impostarla a 1 e accedere alla memoria condivisa. Al termine, resetta `lock` a 0.

```
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 in modo atomico. Utilizzata in CPU x86 Intel per la sincronizzazione di basso livello.

Sleep e Wakeup

Nonostante l'algoritmo di Peterson funzioni, il problema dello spinlock causato dal busy waiting rimane. 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

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.

Semafori

Un semaforo è una variabile che può essere 0 (nessun wakeup) o un valore positivo (wakeup in attesa). Le operazioni principali sui semafori sono:

- **down**: Se il valore del semaforo è maggiore di zero, viene decrementato e il processo continua l'esecuzione. Se è zero, il processo viene bloccato e messo in una coda di attesa.
- **up**: Se il valore è zero, i processi nella coda di attesa vengono svegliati. In ogni caso, il valore viene incrementato e il processo continua l'esecuzione.

Le operazioni sui semafori sono atomiche, evitando conflitti.

Problema Produttore-Consumatore con Semafori

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

- **mutex**: Garantisce l'esclusione mutua.
- **full**: Indica se tutti i posti sono occupati.
- **empty**: Indica se tutti i posti sono liberi.

Problema Scrittori e Lettori

Regola: Ci sono molti lettori e un solo scrittore. I lettori possono accedere contemporaneamente, ma solo un singolo scrittore può accedere alla risorsa.

Mutex

Un mutex è una versione semplificata dei semafori, usata per gestire la mutua esclusione di risorse o codice condiviso. Può essere in due stati: locked (bloccato) o unlocked (sbloccato). Le operazioni principali sono:

- **mutex lock**: Un thread chiama mutex lock per accedere alla regione critica. Se il mutex è unlocked, il thread entra; se è locked, il thread attende.
- **mutex unlock**: Al termine dell'accesso, il thread chiama mutex unlock per liberare la risorsa.

Mutexes in Pthread

La libreria Posix Pthread fornisce funzioni per la sincronizzazione tra thread:

- **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?

- **Mutex**:
  - Utilizzato principalmente per garantire l'esclusione mutua.
  - Semantica di proprietà, solo chi ha il mutex può rilasciarlo
  - Più semplice e comportamento più prevedibile
- **Semaforo**:
  - Utilizzato per la sincronizzazione tra thread e il controllo delle risorse.
  - Non ha semantica di proprietà, chiunque può aumentare o diminuire il conteggio del semaforo

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. Per gestire le attese, i monitor utilizzano variabili condizionali e operazioni come wait e signal.

Scambio di Messaggi

Metodo di comunicazione tra processi usando primitive `send(dest, &msg)` e `receive(src, &msg)`. Può essere utilizzato in vari scenari, compresi sistemi distribuiti. Problemi includono la perdita di messaggi, la necessità di feedback, gestione dei messaggi duplicati e autenticazione dei processi.

Problema Produttore-Consumatore con Messaggi

Utilizza una soluzione senza memoria condivisa, basata solo su messaggi. Il consumatore invia al produttore N messaggi vuoti, il produttore riempie un messaggio vuoto e lo invia al consumatore.

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. Utili nei calcoli paralleli su matrici.

Inversione delle Priorità

L'inversione delle priorità si verifica quando un thread con priorità bassa detiene una risorsa che un thread con priorità alta deve utilizzare. Soluzioni includono disattivare gli interrupt, Priority Ceiling, Priority Inheritance e Random Boosting.

Read-Copy-Update

L'obiettivo è accedere in modo concorrente senza lock, evitando l'inconsistenza dei dati. I lettori vedono o la versione vecchia o quella nuova dei dati, mai un mix delle due. È diffuso nel kernel dei sistemi operativi.