6° lezione S.O.

Due Date	@March 25, 2025
Multi-select	Comunicazioni tra Processi
Status	Done

Comunicazione tra processi

Modello a Pipe

Si hanno due o più processi

Quando si usa la shell è prassi abbastanza comune utilizzare una sintassi di questo tipo

```
cd pippo | ls pappo | g++ main.cpp
```

Due o più comandi possono essere eseguiti in parallelo in pipeline. Quando si scrive un comando del genere, la shell crea altrettanti processi in modo indipendente, ma questi processi verranno messi in comunicazione sfruttando i canali di I/O che ciascuno di essi possiede.

Si ha anche un coordinamento: in una situazione come quella descritta sopra, ad esempio, il comando `ls pippo` può essere eseguito solo dopo che `cd pippo` è stato completato.

Si ha anche qui il concetto di pausa che può essere attiva o passiva:

- **Attiva:** un approccio prono alla speco.
- **Passiva:** indica una sorta di pausa ottimale che non va ad utilizzare cicli di CPU per usarla

Come viene risolto il problema del coordinamento?

Il problema del coordinamento viene affrontato grazie alla natura bloccante delle operazioni di I/O. Ad esempio, supponiamo di voler acquisire un valore da tastiera con lo `scanf()`: il processo rimarrà interrotto fino a quando l'operazione non sarà completata e verrà inserito il carattere `\n` (delimitatore di informazioni).

Se si hanno due processi P_1 e P_2, quando P_1 produce un output e P_2 sta eseguendo un'altra operazione, P_1 deve bloccarsi in attesa che P_2 si liberi. Questo tipo di sincronizzazione evita la perdita di dati e garantisce che il flusso di comunicazione tra i processi avvenga in modo ordinato ed efficiente.P_1

Dietro i canali di I/O vi sono dei buffer, il cui scopo è evitare i vincoli e i tempi morti imposti dal requisito di perfetta sincronia. L'idea è che il buffer, con una dimensione prestabilita, funzioni da cuscinetto per accodare i dati inseriti in questo canale, mentre P_2 li estrae dallo stesso.

Ciò non significa che il problema del blocco venga eliminato del tutto, ma offre un margine maggiore per evitare che i requisiti di sincronia interrompano entrambi i processi. Ad esempio, se P_2 è molto lento nell'elaborare ciò che ha prodotto P_1 il buffer permette a P_1 di continuare a produrre dati fino a riempirlo, evitando un blocco immediato.

Tuttavia, se il buffer si satura e P_2 non ha ancora consumato i dati, allora P_1 sarà costretto a bloccarsi fino a quando il buffer non avrà nuovamente spazio disponibile. Questo meccanismo permette una comunicazione più efficiente tra i processi, riducendo i tempi di inattività e migliorando le prestazioni complessive del sistema.

Comunicazione tra processi

Supponiamo di aver due processi:

- un processo padre
- un processo figlio

Questi processi sono indipendenti tra loro.

Per permettere la comunicazione tra di essi, è necessario creare un canale di comunicazione o uno strumento utile per gestire il loro isolamento. Possiamo immaginare ogni processo con il proprio spazio di memoria separato.

L'idea è quella di creare una **zona franca**, ovvero un **segmento di memoria condiviso**, che può essere mappato nello spazio di indirizzamento dei processi.

Questo segmento funge da **finestra di comunicazione**, permettendo ai processi di leggere e scrivere dati, proprio come una variabile condivisa.

Nel segmento di memoria condiviso è possibile creare una **struttura dati** (come una coda o uno stack) per facilitare lo scambio di informazioni tra i processi. In questo modo, un processo può scrivere dei dati che l'altro processo può leggere e utilizzare, abilitando una comunicazione bidirezionale.

Oltre alla creazione del segmento di comunicazione, è simile a quello in cui un processo contiene più thread. In quel caso, la variabile x condivisa può essere collocata ovunque all'interno dello spazio di memoria del processo per permettere la comunicazione tra i vari thread.

Un aspetto cruciale è affrontare il **secondo problema**: cosa succede se più entità accedono contemporaneamente alla stessa struttura dati? Questo è lo stesso problema che si presenta nei thread e nella gestione della "finestra" di comunicazione tra processi. Senza un meccanismo di sincronizzazione, si potrebbero verificare condizioni di gara (*race conditions*), inconsistenze nei dati e comportamenti imprevedibili.

Un'osservazione interessante è che la **struttura utilizzata come area di scambio dati può essere un file**. In questo caso, i processi comunicano leggendo e scrivendo su un file condiviso, sfruttando il file system come mezzo di interazione. Questo approccio introduce nuove sfide, come la gestione della concorrenza e il controllo degli accessi, ma può essere utile quando i processi non condividono direttamente la memoria.

Si implementeranno vari **meccanismi di sincronizzazione**, e l'uso di file permette di avere strumenti che gestiscono in modo naturale queste situazioni. Questi strumenti consentono di **mettere in attesa** (*addormentare*) i thread o i processi che stanno aspettando dati dai loro pari.

Un aspetto fondamentale è garantire che chi deve sbloccare la condizione di attesa lo faccia in modo efficiente, evitando sprechi di risorse e ottimizzando i **cicli della CPU**. La sincronizzazione deve essere gestita con attenzione per prevenire **stalli (deadlock)** e garantire che i processi possano procedere senza inutili attese.(?)

Come detto sopra, nel caso delle **pipe**, la sincronizzazione avviene automaticamente grazie alla natura bloccante delle operazioni di lettura e scrittura.

Concorrenza tra Strutture Dati Condivise

Esempio

Supponiamo di avere due processi che condividono un **segmento di memoria** contenente diverse strutture dati. Tra queste, vi è una variabile intera condivisa c, che rappresenta il saldo di un conto corrente.

I due processi devono eseguire più versamenti incrementando il valore di c con l'operazione:

$$c = c + 1$$

Tuttavia, questa operazione non è atomica. Infatti, il valore attuale di c viene probabilmente:

1. **Caricato su un registro** (ad esempio, R_0 per il processo P_1).
2. **Incrementato** di 1 nel registro.
3. **Salvato nuovamente nella variabile c**.

Se nel frattempo il processo P_2 esegue la stessa operazione utilizzando un altro registro (ad esempio, R_1), si può verificare una **condizione di gara (race condition)**. Ciò accade perché entrambi i processi leggono lo stesso valore di c prima di aggiornarlo, causando possibili **inconsistenze nei dati**.

Lo stesso problema si verifica in un **sistema multicore**, dove due o più core eseguono codice concorrente che accede alla stessa locazione di memoria. In questo caso, il sistema dispone di un **bus dedicato** attraverso il quale vengono incanalate le richieste di **fetch** (lettura) e **store** (scrittura). Questo non è un caso fortuito, ma una situazione che accade **sempre** in sistemi multicore.

Il problema si verifica anche all'interno del **kernel**, poiché possono crearsi **flussi concorrenti** di esecuzione che operano su **strutture dati condivise**. In un sistema operativo, i **thread** o i **processi** che eseguono codice simultaneamente possono accedere a risorse comuni, come segmenti di memoria condivisi ecc senza un'adeguata sincronizzazione. Questo si verifica, ad esempio, nel caso degli **interrupt**, poiché si ha una **coda dei processi pronti condivisa** tra il kernel e la gestione dell'interrupt. Se si sta scrivendo la stessa locazione di meoria si ha un problema

Ogni core può cercare di accedere simultaneamente alla stessa locazione di memoria, e senza una gestione adeguata della sincronizzazione, si potrebbero verificare **condizioni di gara** (*race conditions*) che portano a dati incoerenti. In tali sistemi, i meccanismi di **coerenza della cache** e la **sincronizzazione tra i**

core sono fondamentali per evitare conflitti nell'accesso alla memoria condivisa.

Questo problema dimostra la necessità di utilizzare **meccanismi di sincronizzazione** per garantire un accesso controllato alle variabili condivise, evitando perdite o sovrascritture di aggiornamenti.

Idea di Soluzione

Si deve garantire **mutua esclusione** nell'esecuzione delle **sezioni critiche** durante l'accesso ai dati condivisi.

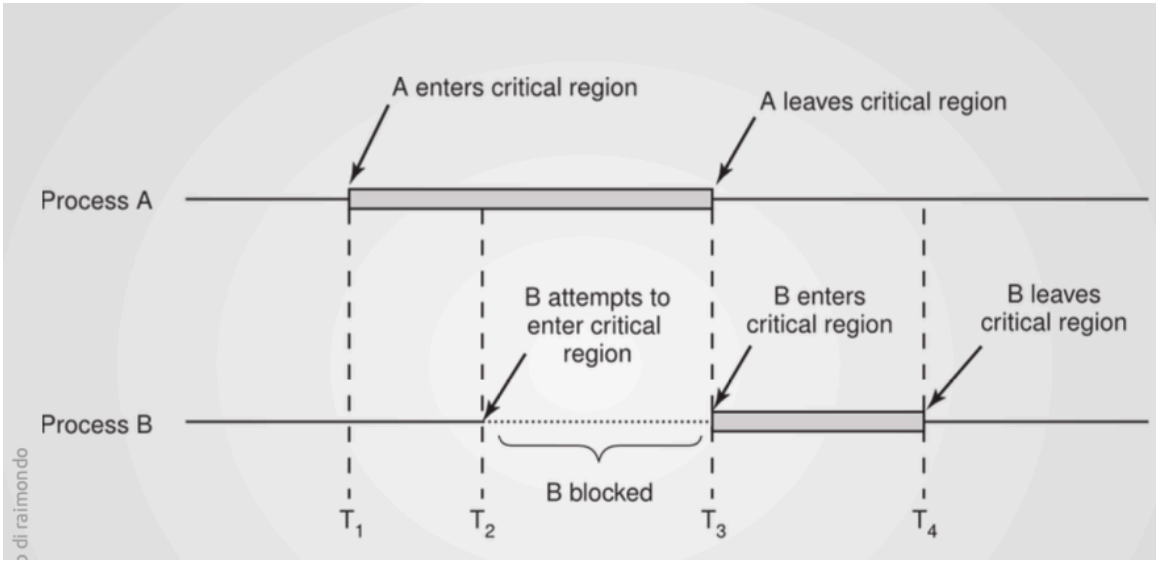
Ad esempio, se il processo P_1 sta lavorando sulla variabile x il processo P_2 dovrà essere

bloccato dall'eseguire la stessa azione su x finché P_1 non ha completato la sua operazione.

Che cos'è una sezione critica?

È un modo formalizzare ed isolare i codici sensibili a questo tipo di approccio. Supponiamo di avere due processi P_1 e P_2.Se entrambi devono accedere alla stessa risorsa condivisa, osserviamo il codice e si **isola** la parte in cui si verifica l'accesso alla risorsa condivisa, questa è definita "zona critica". Solo un processo alla volta dovrebbe essere in grado di accedere alla zona critica, per evitare condizioni di gara e garantire la coerenza dei dati.

Come si implementa la zona Critica?



Nell'immagine di cui sopra, è rappresentata una **timeline** in cui le **fasi evidenziate** indicano i momenti in cui i rispettivi processi entrano nelle **sezioni critiche**.

- **Al tempo** T_1, il processo A entra nella **sezione critica**. Poiché in questo momento non ci sono altri processi in esecuzione nella stessa sezione, non si verifica alcun blocco. Tuttavia, lo strumento di **coordinamento** viene comunque attivato per garantire che, nel caso in cui un altro processo tentasse di accedere alla sezione critica, venga **bloccato opportunamente**.
- **Al tempo** T_2, il processo B tenta di entrare nella **sezione critica**, ma scopre che A è ancora all'interno. A questo punto, il processo B viene **bloccato** in attesa che A termini la sua esecuzione. Questo blocco è gestito tramite un **evento software**.
- **Al tempo** T_3, il processo A esce dalla **zona critica**, permettendo così a B di accedere alla sezione critica e riprendere l'esecuzione. In questo momento, nessun altro processo è in attesa di entrare nella sezione critica.

Le **zone critiche** dovrebbero essere distinte in base alle **strutture dati** coinvolte, ed è il **programmatore** a decidere come gestirle.

Si possono individuare diversi tipi di **sezioni critiche**:

- Alcune sezioni monitorano un singolo **processo**.
- Altre sezioni monitorano **più processi**, e vengono chiamate **ibride**.

Quali sono le proprita dediderabili che impletano una sezione critica?

1. **mutua esclusione nell'accesso alle rispettive sezioni critiche;**
2. **nessuna assunzione sulla velocità di esecuzione o sul numero di CPU;**
3. **nessun processo fuori dalla propria sezione critica può bloccare un altro processo;**
4. **nessun processo dovrebbe restare all'infinito in attesa di entrare nella propria sezione critica.**

Sezioni Critiche: Tipologie

- **Enter Region**: fase di **entrata** nella sezione critica; se necessario, diventa una chiamata esplicita per garantire l'accesso controllato.
- **Leave Region**: fase di **uscita** dalla sezione critica, che permette ad altri processi di accedere alla risorsa condivisa.

Tipologie Di Sezioni Critiche: Come implementarle?

Nell'esempio del **prelievo in banca**, il meccanismo di **prelazione** si attiva a metà della sezione critica, influenzando l'accesso concorrente alle risorse condivise.

Se si **inibisse la prelazione** all'interno delle sezioni critiche, si potrebbe raggiungere lo scopo, ma solo **in modo limitato nel caso in cui si abbia una singola CPU**.

Su tutte le **architetture** che implementano il **modello TRAP**, esiste una funzione nel set di **istruzioni** della **CPU** che si occupa di inibire il **meccanismo di interrupt**.

Questa operazione può essere eseguita **solo in modalità kernel**, poiché potrebbe causare gravi problemi al sistema. Infatti, **inibire gli interrupt per un tempo prolungato** non è una pratica consigliata, in quanto interrompe diversi meccanismi essenziali, come la gestione dell'**I/O**, la **scheduling dei processi**, e altre operazioni critiche del sistema operativo.

Per questo motivo, **non può essere affidata direttamente al programmatore**, poiché potrebbe **dimenticare di riattivare gli interrupt**, causando un blocco del sistema o comportamenti imprevedibili.

Se si hanno due core, **CPU_1** e **CPU_2**, e su questi stanno girando i processi **P_1** e **P_2**, che operano su una variabile condivisa **c**, **disabilitare gli interrupt non risolve il problema**.

Infatti, il processo concorrente potrebbe essere in esecuzione in **reale parallelismo** sull'altro core, continuando a modificare la variabile **c** indipendentemente dalla disabilitazione degli interrupt su una sola CPU.

Un'altra soluzione potrebbe essere **l'inserimento di una variabile di lock**. Se questa assume il valore **0**, significa che **nessun processo sta lavorando sulla variabile condivisa c**. Se invece la variabile di lock assume il valore **1**, indica che **un processo sta attualmente operando su c**, impedendo così ad altri processi di accedervi contemporaneamente.

Se immaginiamo che due processi **eseguano quasi contemporaneamente** la funzione `enter region`, si potrebbe verificare una situazione in cui **P_1** vede che il lock è impostato a **0** e sta per cambiarlo a **1**.

Nel frattempo, però, arriva **P_2**, che verifica anch'esso che il lock sia **0** e lo imposta a **1**. **Entrambi i processi accedono alla sezione critica contemporaneamente**, violando il meccanismo di mutua esclusione e causando **condizioni di gara**(*race conditions*), con il rischio di **dati corrotti** o **comportamenti imprevedibili**.

Soluzione: Alternanza Stretta

```
int N = 2;
int turn;

void enter_region(int process) {
    while (turn != process) {
        // Attesa attiva
    }
}

void leave_region(int process) {
    turn = 1 - process;
}
```

Questa soluzione è valida **solo per due processi**. Le due procedure prendono come **parametro in ingresso** un **identificativo del processo**, che può essere **0 o 1**. Questo valore indica **quale processo vuole eseguire una delle due procedure**, ovvero accedere o lasciare la sezione critica.

Nella funzione `enter_region()`, se **le condizioni non sono favorevoli** per entrare nella **sezione critica**, il processo **si blocca in attesa**, restando in un ciclo di **attesa attiva** (*busy waiting*).

La variabile `turn`, come suggerisce il nome, **indica a quale processo spetta il turno di accesso** alla sezione critica. Questo meccanismo garantisce **un'alternanza stretta** tra i due processi,P_0 e P_1, evitando che entrambi possano accedere contemporaneamente alla risorsa condivisa.

La **condizione di blocco** si verifica quando la variabile `turn` è **diversa** dall'identificativo del processo che sta tentando di

eseguire `enter_region()`. In questo caso, il processo rimane in attesa attiva (**busy waiting**) fino a quando `turn` non assume il valore corretto, consentendogli di entrare nella sezione critica.

- La variabile `turn` stabilisce **di chi è il turno** per accedere alla sezione critica.
- Se `turn` coincide con l'identificativo del processo (`process`), allora il processo può accedere.
- Se `turn` è diverso, il processo **si blocca** in un ciclo di attesa attiva.
- Quando il processo **esce dalla sezione critica**, cambia `turn`, concedendo il permesso all'altro processo.

Questa soluzione è generalizzabile al caso N.

La problematica principale di questa soluzione risiede nel fatto che impone un'alternanza rigida dei turni, il che porta a una violazione della **terza condizione** precedentemente discussa, ovvero la **condizione di progresso**.

A causa di questa rigidità, si potrebbe verificare la seguente situazione:

- Supponiamo di avere due processi P_0 e P_1
- Inizialmente, P_0 entra nella sezione critica ed esce, impostando `turn = 1`, cedendo così il controllo a P_1.
- Tuttavia, P_1 non ha la necessità di accedere alla sezione critica e continua la sua esecuzione al di fuori di essa.
- Successivamente, P_0 ha nuovamente bisogno di accedere alla sezione critica, ma **non può entrarci** perché il valore di `turn` è ancora impostato su `1`, in attesa che P_1 acceda alla sezione critica.
- Dato che P_1 non ha alcuna intenzione di entrare nella sezione critica, P_0 **rimane bloccato inutilmente**, anche se la sezione critica è libera.

2° Soluzione: Soluzione di Peterson

È una soluzione software. Prevede alcune variabili condivise, in particolare:

- Due strutture dati condivise (**istred**) che hanno tante posizioni quanti sono i processi.

- Una variabile condivisa per dare la soluzione al ricevente, destinata a segnalare all’altro la volontà di entrare.

```
#include <stdio.h>
#include <stdbool.h>
#include <pthread.h>

#define N 2  // Numero di processi

int turn;
bool interested[N] = {false, false};  // Stato di interesse per i due processi

void enter_region(int process) {
    int other = 1 - process;  // Identifica l'altro processo
    interested[process] = true;  // Segnala l'intenzione di entrare
    turn = process;  // Assegna il turno a se stesso
    while (interested[other] && turn == process);  // Attendi se l'altro processo v
}

void leave_region(int process) {
    interested[process] = false;  // Segnala che ha lasciato la sezione critica
}

void* process_function(void* arg) {
    int process = *(int*)arg;

    for (int i = 0; i < 5; i++) {  // Esegui più iterazioni
        enter_region(process);

        // Sezione critica
        printf("Processo %d è nella sezione critica\n", process);

        leave_region(process);
    }
    return NULL;
}

int main() {
```

```
    pthread_t p1, p2;
    int p0 = 0, p1_idx = 1;

    pthread_create(&p1, NULL, process_function, &p0);
    pthread_create(&p2, NULL, process_function, &p1_idx);

    pthread_join(p1, NULL);
    pthread_join(p2, NULL);

    return 0;
}
```

Se due processi scrivono sulla stessa variabile senza sincronizzazione, può verificarsi una **race condition**, cioè il risultato dipende dall'ordine in cui le operazioni vengono eseguite.

Nel caso specifico del codice con l'algoritmo di Dekker, il meccanismo di mutua esclusione evita che entrambi i processi entrino contemporaneamente nella sezione critica. La variabile `turn` è usata per decidere quale processo può procedere quando entrambi sono interessati a entrare.

Se entrambi i processi impostano `turn` quasi contemporaneamente, l'ultimo valore scritto sarà quello effettivo. Tuttavia, la condizione `while (interested[other] && turn == process);` assicura che solo uno dei due possa effettivamente entrare nella sezione critica alla volta.

L'idea è lasciare che entri per primo chi è riuscito a impostare la variabile `turn`, mentre l'altro rimane in attesa finché il primo non esce dalla sezione critica e imposta `interested[process] = false`.

Questa soluzione puo essere genereallto a N processi prendendo i processi afue a due e vedendo chi vince