**THREADS**

I thread sono processi leggeri (o anche sotto processi perché i processi possono essere formati da più threads) che sono stati introdotti per poter collaborare tra di loro. Tra di loro ci sono delle caratteristiche condivise che sono il codice, le variabili locali e i file aperti altre che sono specifiche di un thread che sono lo stack e i valori dei registri della CPU.

**Perché sono stati introdotti?**

Vengono introdotti i threads perché si volevano realizzare dei processi cooperanti tra di loro quindi questi processi dovevano condividere file, codice e dati globali ma si è notato che queste operazioni sono molto costose in termini di **cambio di contesto** perché file, dati globali e codice non sono condivisi nei processi e durante i cambi di contesto per dei processi cooperanti non è necessario cambiare sempre file, codice e dati globali e quindi il contesto non dovrebbe cambiare molto. Per questo motivo vengono introdotti i threads che sono dei sotto-processi il cui codice, variabili locali e file sono comuni tra di loro e quindi il contesto durante il context switch non cambierà di molto.

**Perché sono utili?**

1)Aumentano il livello di parallelismo all’interno di una singola applicazione.

2)Tra i vari threads c’è condivisione di risorse fra attività cooperanti in un’applicazione.

3)Creazione di un threads e molto meno costosa della creazione di un processo.

**Come sono implementati:**

1)**A livello utente:** in questo caso il sistema operativo non è a conoscenza dei threads e vengono gestiti completamente dalla modalità utente quindi per creare, terminare o sospendere l’esecuzione di un threads non si richiede l’intervento del sistema operativo. Il sistema di gestione dei threads (esiste in pratica una tabella ciascun threads) è costituito da una libreria di funzioni e deve tenere traccia dello stato e di quanto tempo sono all’interno della CPU i threads proprio come faceva la PCB per i processi.

2)**A livello kernel**

**Vantaggi e svantaggi dei thread**

**Livello utente🡪** in questo caso i threads sono invisibili per il sistema operativo poiché li vede come un unico processo. Esiste una tabella dei thread separata per ogni processo e questo consente una maggiore flessibilità perché ogni processo può stabilire una propria politica di scheduling per i propri threads. Lo svantaggio è che se si blocca un threads o il mancato rilascio di quest’ultimo da parte della CPU porta al blocco dell’intero processo.

**Livello kernel🡪** a livello kernel esiste una tabella a livello di sistema per tutti i thread di tutti i processi. Quindi la politica di scheduling viene decisa a livello di sistema ed il blocco di un thread non è fatale per gli altri threads che vengono visti come entità autonome dal sistema operativo.

**ESEMPIO DI MULTITHREAD:** è un word processor di tipo ‘’quello che vedi è quello che è” realizzato mediante 3 threads parzialmente indipendenti in cui il 2) e il 3) possono girare liberamente:

1)accetta il testo inserito da tastiera e modifica la riga corrente

2)riformata l’intero documento quando la modifica della riga corrente ha effetto su tutto il documento

3)salva periodicamente il documento su disco

**SINCRONIZZAZIONE E COMUNICAZIONE TRA PROCESSI/THREADS**

Per poter realizzare delle applicazioni tra processi o threads per farli collaborare occorrono dei meccanismi per:

1)**Attivazione e terminazione** di processi e threads

2)**Sincronizzazione** cioè dobbiamo attendere che il processo P1 sia arrivato ad un punto per poter attivare il processo P2.

3)**Comunicazione** cioè il processo P2 passa i dati al processo P1

I meccanismi per far rispettare queste caratteristiche si possono trovare nei linguaggi concorrenti tipo JAVA, nelle system calls e nelle librerie di funzioni.

**Due modelli di interazione**

**-Modello a memoria condivisa:** i processi (o thread) condividono almeno alcune variabili in memoria. Se un processo/thread modifica una di queste variabili, tutti gli altri processi/thread vedranno il nuovo valore. La condivisione è naturale per threads, per i processi è stata introdotta con meccanismi appositi.

**-Modello a scambio di messaggi:** i processi non condividono aree di memoria, ma possono inviarsi messaggi utilizzando istruzioni sende receive. Lo scambio di informazioni è sempre esplicito.

I meccanismi di interazione devono permettere

•lo scambio di informazioni

•il corretto ordinamento di azioni compiute da diverse entità

•l‘accesso controllato a risorse condivise

Si possono dunque avere interazioni di tipo *cooperativo* o *competitivo* che utilizzano le stesse operazioni messe a disposizione

Affinché l’interazione avvenga in modo corretto i processi devono rispettare alcune regole (per esempio fare richiesta esplicita di una risorsa condivisa ed eventualmente attendere il proprio turno per l’uso): se le regole non vengono rispettate si possono verificare interferenze(malfunzionamenti)

**CORSA CRITICA**

La corsa critica avviene quando ci sono delle variabili condivise e avviene a causa dell’intromissione del S.O. che stabilisce quale sia la sequenza delle istruzioni da eseguire. Supponiamo che ci siano più processi/threads che eseguono la procedura VersaSulConto.

**VersaSulConto(int numconto,int versamento)**

**{ Saldo = CC[numconto];**

**Saldo = Saldo + versamento;**

**CC[numconto] = Saldo; }**

Esempio1

Ipotizziamo che due processi lavorino sul contocorrente 1200🡪 CC[1200] e supponiamo che ci sia al suo interno una cifra pari a 2000.

Il progetto P1 versa una cifra 200 e il progetto P2 versa una cifra di 350. Alla fine del versamento noi ci aspettiamo che il nostro saldo sia di 2550 ma c’è un problema però perché questi processi sono eseguiti in modo pseudoconcorrente.

P1 P2

1.saldo=CC[1200](P1 saldo 2000) **context switch🡪** 2.saldo=CC[1200](P2 saldo 2000)🡪proseguo P2

4.saldo=saldo+200(P1 saldo 2200)  **<--context switch** 3.saldo=saldo+350(P2 saldo 2350)

5.CC[1200]=saldo (P1 saldo 2200) **context switch🡪** 6.CC[1200]=saldo (P2 saldo 2350)

Il CC[1200] alla fine dei processi avrà saldo 2350 invece che 2550 e quindi per colpa dell’intromissione del sistema operativo mi perdo un aggiornamento. La sequenza corretta sarebbe che un processo faccia tutte le sue operazioni e poi un altro processo faccia le sue operazioni.

Esempio2

Si riempie un array ‘a’ condiviso usando una variabile ‘i’ condivisa come indice:

(i==5)

a[i]=10; a[i]=20;

i++; i++;

Vorremmo presumibilmente ottenere: i==7 e a[5]==10, a[6]==20 oppure viceversa. Ma se anche le istruzioni C fossero «atomiche» (indivisibili), le operazioni potrebbero avvenire ad esempio nell’ordine:

a[i]=10; 🡪a[i]=20; 🡪 i++; 🡪 i++; ma in questo caso andrei a scrivere in a[5] 10

poi sovrascriverei 20 in a[5].

**Come risolvere queste situazioni:** per risolvere queste situazioni dobbiamo garantire l’atomicità di queste istruzioni perché anche istruzioni come a++ può essere tradotta in più istruzioni a livello ISA ed è proprio in quel momento che avviene l’intervento del sistema operativo. Dobbiamo fare in modo che il sistema operativo non intervenga. Una soluzione possibile è fare in modo che queste istruzioni vengano fatte in modo atomico.

La procedura VersaSulConto è una sezione critica perché c’è una variabile condivisa e ci sono due processi che la vogliono aggiornare. Noi dobbiamo evitare che se c’è un processo in esecuzione in quella sezione critica di far entrare dentro quella sezione un altro processo. Quindi per realizzare ciò dobbiamo bloccare P2 finché P1 non ha finito di eseguire tutta la sezione critica. Questo problema si chiama **mutua esclusione.** Quindi per controllare l’accesso alle sezioni critiche nel codice dei processi devono essere introdotte istruzioni di sincronizzazione all’inizio ed alla fine di ogni sezione critica.

**Mutua esclusione🡪** se c’è un processo nella sezione critica non ne può entrare un altro e quindi mi assicuro che non avvengano delle race condition.

**Requisiti per una buona soluzione al problema delle corse critiche**

1)**[Mutua esclusione]** se il processo P è in esecuzione nella propria sezione critica, nessun altro processo può essere in esecuzione nella propria sezione critica (sugli stessi dati).

2)**[Progresso]** se nessun processo è in esecuzione nella propria sezione critica, e vi sono dei processi che intendono entrare nelle rispettive sezioni critiche, la scelta su chi può procedere dipende solo da quali sono questi ultimi processi, e questa scelta non può essere rimandata indefinitamente.

3)**[Attesa limitata]** quando un processo P ha richiesto di entrare in sezione critica, esiste un limite massimo al numero di volte per cui viene consentito ad altri processi di entrare nelle rispettive sezioni critiche prima che si accordi l’accesso a P.

**Questi 3 sono requisiti ideali e non è detto che tutte le soluzioni che illustriamo li soddisfino. Qui sotto illustro delle buone alternative:**

1)deve essere garantita la mutua esclusione nell’esecuzione delle sezioni critiche da parte di tutti i processi coinvolti;

2)la soluzione non può basarsi su ipotesi riguardo alle velocità relative di esecuzione dei processi coinvolti e al numero di CPU;

3)un processo che sta eseguendo sezioni non critiche non deve impedire ad altri processi di accedere alla propria sezione critica;

4)non può accadere che un processo debba attendere indefinitamente il proprio turno di entrare in sezione critica.

**Una possibile soluzione (un po’ drastica) per risolvere il problema delle corse critiche è la disabilitazione degli interrupt.**

**DISABILITAZIONE INTERRUPT**

Un processo prima di entrare in sezione critica disabilità l’interrupt e quindi il sistema operativo non sente i segnali di interrupt e il processo può andare avanti fino alla fine.

**Problemi di questa soluzione**

1)Per motivi di protezione non si vuole permettere ad un processo che gira in modalità utente di disabilitare gli interrupt (potrebbe approfittarne per monopolizzare la CPU.

2)La soluzione funziona correttamente solo se il sistema ha un’unica CPU: nei sistemi multiprocessore la disabilitazione degli interrupt è locale a ciascuna CPU, e i processi possono eseguire davvero in parallelo.

**Altra soluzione: questa soluzione prevede di usare una variabile condivisa da tutti i processi**

lock è una variabile condivisa che vale 0 se nessuno sta eseguendo nella zona critica e vale 1 se c’è un processo in zona critica

| lock=0; 🡪**inizializzazione**

| while(lock==1); 🡪**per entrare in sezione critica**

| lock=1;

**| corpo d’istruzioni della zona critica**

| lock=0;🡪**uscita dalla sezione critica**

Dato che inizialmente lock=0 supererà il ciclo while che funge da controllo per vedere se ci sono processi che stanno girando in zona critica e successivamente dato che poi ci sarebbe un processo verrà impostato a 1 la variabile lock. Se inizialmente lock fosse uguale a 1 starebbe all’interno del ciclo while perché vuol dire che c’è un processo all’interno della sezione critica.

Questa soluzione però **non è corretta** poiché la procedura per entrare in sezione critica è anche essa una sezione critica e in questo caso **non garantisce la mutua esclusione.**

1)2 processi valutano in pseudo-parallelo (lock==1).

2)entrambi trovano falsa la condizione del while ed eseguono quindi lock=1 ed entrambi entrano in sezione critica.

**ISTRUZIONE TEST AND SET LOCK(TSL)**

Un’altra strada è sfruttare l’aiuto dell’hardware, torniamo all’idea di utilizzare una variabile “lock” condivisa. Per avere come ATOMICA (indivisibile o elementare) la sequenza di azioni che effettua il test sul valore della variabile condivisa e poi la imposta a 1, ci si fa fornire dall’hardware una istruzione DI LINGUAGGIO MACCHINA non interrompibile.

**TSL *registro*, *variabile***

che ATOMICAMENTE opera sul *registro* e sulla *variabile* in memoria:

1)copia *variabile* in *registro* (che può poi essere *test*ato)

2)imposta *variabile* a 1 (*set*)

Nei sistemi multiprocessore, la TSL deve riservareil bus e mantenerlo finché non sono concluse sia la lettura che la scrittura della variabile

**Come può essere usata la TSL?**

**Ingresso in sezione critica**

**TSL register, lock** // copia lock che è la variabile condivisa in register e setta lock a 1

**CMP register, #0** // qua chiediamo se registro==0 e se è uguale a 0 vado avanti se è diverso da zero rivado a Ingresso sezione critica con l’istruzione successiva perché vuol dire che c’è un processo in sezione critica.

**JNE ingresso sezione critica**

**RET return to caller;** ingresso sezione critica

**Uscita dalla regione critica**

**MOVE lock, #0** // inserisce 0 nella variabile lock

**RET** // ritorna da dove è stato chiamato

**NOTA BENE🡪** anche se per poche istruzioni **ho un busy waiting da TSL fino a JNE.**

L’istruzione TSL può essere fornita come istruzione di libreria ad alto livello, quindi per un linguaggio ad alto livello.

**Inizializzazione 🡪** lock=0;

**entrata in sezione critica 🡪** while(TestAndSet(&lock)); // **busy waiting**

**corpo di istruzioni della sezione critica …**

**uscita della sezione critica🡪**  lock=0;

Questa condizione non soddisfa il requisito di attesa limitata ma si usa per sezioni di breve durata.

Però l’esecuzione in mutua esclusione di sezioni critiche non è l’unico tipo di problema di sincronizzazione tra processi.

1)Gestire la condivisione di un pool di N risorse fra 2 o più processi/threads ogni processo/thread deve chiedere di acquisire una risorsa prima di utilizzarla, rimanere in attesa se non è disponibile, procedere ad usarla se l’acquisizione della risorsa è andata a buon fine, rilasciarla nel momento in cui l’utilizzo è concluso.

2)Garantire che il codice A del Processo1 sia eseguita prima della porzione di codice B del Processo2.

3)Garantire che N thread completino tutti una prima fase di esecuzione prima di poter passare tutti ad una seconda fase.

**Problema del produttore consumatore**

Il problema “classico” del Produttore e Consumatore incorpora diversi problemi di sincronizzazione: la necessità di ordinare correttamente le attività dei due processi e di far attendere un processo quando non sono disponibili risorse (per esempio questo problema si presenta quando due processi.

**Produttore🡪** il produttore può inserire dati nel buffer solo se quest’ultimo è vuoto. Possiamo considera le posizioni vuote del buffer come risorse.

**Consumatore🡪** il consumatore può prelevare dati dal buffer solo se il buffer contiene dati che non sono stati prelevati. Possiamo considerare risorse le posizioni piene del buffer.

Soluzioni basate su variabili di lock e istruzioni come test-and-set-lock **sono adeguate solo per alcuni problemi di sincronizzazione semplici**, ma **sono eccessivamente di “basso livello” per problemi più complessi**. Per sviluppare applicazioni concorrenti (con più thread o più processi che cooperano o che devono coordinarsi nell’uso di risorse condivise) sono utili meccanismi più di alto livello. È inoltre opportuno cercare di eliminare (o almeno ridurre al minimo) **l’attesa attiva (busy waiting)** introducendo operazioni sospensive.

**Che cos’è il busy waiting?**

Il busy waiting è una tecnica di sincronizzazione per cui un processo o un thread che deve attendere il verificarsi di una certa condizione lo faccia verificando ripetutamente tale condizione.

Il ***busy waiting*** deve essere evitato nei limiti del possibile (o almeno utilizzato solo quando la probabilità di dover attendere è bassa e comunque la condizione che causa l’attesa dura poco) perchè:

•Spreca tempo di CPU (il processo/thread che attende è ready/running e se la CPU è una sola la condizione non può cambiare fino a che non viene data ad un altro processo/thread).

•Nel caso di processore singolo può verificarsi il problema dell’**inversione di priorità**:

Consideriamo due processi P1 e P2, dove P1 ha priorità su P2 (tra P1 e P2, lo scheduler sceglie sempre P1; NB lo scheduler sceglie tra i processi pronti, non si occupa di sincronizzazione e non va certo a vedere il codice per accorgersi che stanno facendo attesa attiva). Se P1 cicla, ad es., while (lock) e P2 è il processo il cui codice dovrebbe fare lock=0, non esiste via d’uscita dato che lo scheduler selezionerà sempre P1.

Un’idea per evitare il busy waiting è quella dello **sleep** e **wake up** che sono 2 funzioni realizzate come delle system call. La **wake up** non ha effetto se eseguita su un processo non bloccato. Quando si fa la **wake up** non va subito in esecuzione il processo ma va prima nello stato va ready.

**Esempio1:** In questa prima versione sleep() non ha parametri e wake up ha come parametro il processo che deve svegliare.

P1: … P2: …

Sleep() WakeUp(P1)

… …

**Esempio2:** in questa seconda versione sleep deve specificare l’id di una condizione d’attesa ‘’cond’’, alla quale farà riferimento anche la wakeup; se più processi sono in attesa su ‘’cond’’ occorre dire se la wakeup deve svegliarne 1 o svegliarli tutti.

P1: … P2: …

Sleep(cond) WakeUp(cond)

... …

Con le primitive sleep e wakeup si può risolvere in parte il problema del produttore e del consumatore.

**utilizzo🡪** la utilizzo nel momento in cui non ci sono posizioni vuote e quindi il produttore attende

**Sleep** **🡪** la utilizzo nel momento in cui non ci sono posizioni piene e quindi il consumatore attende

**Utilizzo 🡪** il produttore sveglia il consumatore bloccato in attesa di un dato

**Wakeup🡪** il consumatore sveglia il produttore bloccato in attesa di uno spazio libero

Void producer(void){ Void consumer(void){

Int item ; int item;

While(True){ While(True){

Item=produce\_item(); if(Count==0)

If(Count ==N) sleep();

Sleep(); item=remve\_item();

Insert\_item(item);//inserisco oggetto // Count--;

Count++; //nel buffer// if(count==N-1)

If(Count==1) wakeup(producer);

Wakeup(consumer); process\_item(item);

} }

} }

Ricorda che la wakeup deve essere usata per processi che si trovano nello stato di waiting.

Se il consumatore è nell’if (count==0)e sta per eseguire la sleep() però, qui non è in mutua esclusione e ci potrebbe essere lo scheduler che interviene e appena prima di fare la sleep() si interrompe e passa al producer(). Il producer inserisce un item incrementa count ed essendo count ==1 si esegue una wakeup(consumer) ma il consumer non era in sleeping.

Quindi l’ultima wakeup non ha effetto dato che il consumer non era bloccato, solo dopo si ritorna ad eseguire la sleep() del consumer e in questo caso il consumer si addormenta per sempre lasciando all’interno del buffer un item. Siamo quindi in una situazione di stallo.