SISTEMI OPERATIVI

Che cos'è un sistema operativo?

Il sistema operativo è il software più importante che possiamo trovare nel calcolatore la cui funzione principale e gestire le risorse della CPU, memoria principale e delle periferiche e inoltre ha il compito di controllore e di non permettere ai programmi in esecuzione di accedere a delle informazioni di altri programmi in esecuzione e ai dati del sistema operativo. Questo software gira in modalità Kernel.

Le modalità Kernel e Utente→ 2 modalità di esecuzione della CPU.

Kernel→ In modalità kernel gira il sistema operativo ed è una modalità in cui può eseguire qualsiasi istruzione si possa eseguire sul calcolatore. Inoltre **comprende la gestione dei processi, la gestione della memoria virtuale e del file system, i driver delle periferiche**. Queste sono le astrazioni introdotte dal sistema operativo.

Utente→ In modalità Utente girano tutto il resto dei software e in questa modalità non si possono eseguire tutte le istruzioni come invece si poteva fare in modalità kernel per esempio la gestione della memoria o la gestione dei dispositivi di I/O non si può fare.

La gestione delle risorse di un sistema operativo include **il multiplexing**. Il multiplexing permette l'esecuzione protetta e simultanea di più programmi.

1)**Multiplexing space→** in questo caso la memoria principale viene suddivisa in più programmi in esecuzione, in modo che ognuno presente in memoria allo stesso tempo sia in attesa di usare al proprio turno la CPU.

2)**Time multiplexing→** quando una risorsa è condivisa temporalmente i programmi fanno a turno per usarla. Il compito del sistema operativo è vedere per quanto tempo usa la risorsa e chi è il prossimo ad usare la risorsa.

MULTIPROGRAMMAZIONE

L'obiettivo della multiprogrammazione è usare al massimo la CPU evitando tempi morti dovuti alla lentezza dell'I/O rispetto alla CPU. La soluzione pensata fu quella di partizionare la RAM in tanti pezzi assegnando un diverso JOB a ogni partizione. Mentre un JOB rimane fermo in attesa del completamento dell'I/O un altro usa la CPU. Potrebbe succedere che un JOB richieda alla sistema operativo di usare un dispositivo di I/O in questo caso allora la CPU viene assegnata ad un nuovo JOB. Questo è il vantaggio dei sistemi multi-programmati.

CONTEXT SWITCH→ Il trasferimento della CPU da un programma all'altro viene chiamato cambio di contesto. Viene eseguito dal sistema operativo che deve salvare lo stato dei registri del programma che esce dalla CPU e caricare all'interno di quest'ultima il nuovo programma.

SYSTEM CALL

La system call è una chiamata di sistema e serve per ottenere servizi dal sistema operativo. Un programma utente fa una system call e quest'ultima entra nel kernel e fa una richiesta al sistema operativo

SISTEMI INTERATTIVI (TIME SHARING)

L'obiettivo di questi sistemi è di garantire tempi di reazione rapidi ai programmi interattivi, evitando che uno monopolizzi la CPU. La CPU viene assegnata a turno a vari programmi in esecuzione e questo turno ha una durata di un time-slice. Se ci sono più programmi che devono usare la CPU un dispositivo hardware chiamato timer si occupa di dire alla CPU attraverso un interrupt che un altro programma ha bisogno della CPU.

TIME SLICE→ Il time slice è la durata di tempo di occupazione da parte di un programma della CPU è viene impostato un valore molto piccolo. Però time slice troppo brevi causano un overhead cioè un costo di gestione molto grande rispetto all'esecuzione del programma. Questo accade perché con time slice troppo brevi vengono effettuati molti context switch e quindi si passa una frazione di tempo troppo grossa a fare i cambi di contesto.

GESTIONE MEMORIA

La memoria ha al suo interno più processi/programmi ma come possono convivere tutti insieme? Dobbiamo fare attenzione a dove andiamo ad allocare i processi/programmi e vedere se in quella zona di memoria che abbiamo deciso di usare per allocare il programma non è già occupata**. L'idea è utilizzare gli indirizzi relativi (o virtuali) e poi fare traduzione in indirizzi fisici quando si sa dove è allocato il programma**. La traduzione viene effettua a tempo di esecuzione poiché si ha una massima flessibilità. La traduzione da indirizzo logico a indirizzo fisico viene controllata dalla **MMU**.

PROTEZIONE MEMORIA→ Devo inoltre assicurarmi che il processo A e il processo B non vadano a scriversi nelle loro relative zone di memoria e per assicurarci che ciò non accada usiamo due registri che ci fornisce la nostra architettura.

BASE→ indica dove si trova la prima istruzione del processo.

LIMIT→ indica dove si trova la fine della zona di memoria del processo.

Questi due registri possono essere modificati solamente in modalità kernel. Il controllo del non superamento del registro limit viene effettuato dalla **MMU**.

Interrupt→ Un interrupt è un segnale da parte di una periferica che indicano alla CPU il verificarsi di eventi esterni. In questo modo il processore deve salvare lo stato (context switch) dell'esecuzione del programma fino all'arrivo dell'interrupt e poi inizierà l'operazione di I/O. Al contrario di quanto succede con la richiesta di un operazione di I/O da parte di un processo se l'operazione di I/O è causato da un interrupt la CPU è libera da quel processo è viene avvertita che I/O è terminato tramite un interrupt. **Le operazioni per la comunicazione con i controller dei dispositivi di I/O sono privilegiate. Quindi totalmente invisibili al programma.** In questo modo solo il sistema operativo può comunicare con i dispositivi di I/O garantendogli protezione ai rispetto ai programmi utente.

Cosa succede alla CPU?

1) Appena arriva l'interrupt la CPU completa l'ultima istruzione che era in corso.

2)Come accade nel context switch salva lo stato d'esecuzione del processo interrotto.

3)Si salta nella zona di memoria dove è contenuto il gestore della periferica.

4)Dopo aver gestito l'interrupt viene ripristinato lo stato del programma interroto.

Interrupt annidati→ Può capitare che subito dopo la richiesta di un primo interrupt ce ne subito dopo un'altra, come possiamo gestirli? La soluzione è impostare un registro chiamato **PSW** che indica che ci stiamo occupando già di un interrupt.

SET ISTRUZIONI ISA PRIVILEGIATE

Queste istruzioni sono eseguibili solo in modalità kernel.

RETINT→ Riabilita gli interrupt e ripristina il programma interrotto.

DISINT→ Disattiva gli interrupt.

ENABINT→ Abilita gli interrupt.

Se queste istruzioni non sono eseguite in mode kernel si incorre in una **TRAP.**

TRAP

La Trap è un'interruzione provocata dall'esecuzione di una determinata istruzione di un programma in esecuzione. Il motivo dell'interruzione è causato per esempio da un'istruzione non eseguibile in mode utente.

DIFFERENZA TRA INTERRUPT E TRAP

La differenza tra Interrupt e Trap è che il primo è sincrono quindi è causato dal programma in esecuzione il secondo è asincrono rispetto al programma.

PROCESSO

Un processo è una attività di elaborazione guidata da un programma la cui velocità di esecuzione dipende da quanti processi condividono la CPU e la memoria.

VELOCITA' DI ESECUZIONE DEI PROCESSI

In un sistema multi-programmato non si possono dare garanzie sulla velocità dei processi poiché non si può prevedere quali parti del codice verranno eseguite e non possiamo prevedere quando la CPU verrà interrotta da una richiesta di I/O.

SISTEMI REAL-TIME→ Solo in questi sistemi sono garantite delle informazioni sulla velocità di esecuzione di un processo.

CREAZIONE PROCESSI

1)Inizializzazione del sistema

2)Esecuzione di una system call di creazione di un processo

3)Richiesta dell'utente di creare un processo

4)Inizio di un JOB in modalità BATCH.

All'avvio di un sistema operativo vengono creati parecchi processi. Alcuni sono processi attivi ossia interagiscono con gli utenti ed eseguono lavoro per loro. Altri processi lavorano in background e sono chiamati demoni.

TERMINAZIONE PROCESSO

1)Quando un programma ha finito correttamente la sua esecuzione.

2)Quando si verifica una condizione di errore rilevata dal programma ed esso decide di terminare.

3)Quando si verifica una condizione di errore che causa una TRAP.

4)Quando un processo A ne richiama un altro e quindi il processo A deve terminare.

STATI DI UN PROCESSO

Il sistema operativo deve tenere traccia dello stato di tutti i processi.

I processi possono essere in due stati: **ATTIVO** e **WAITING.**

Nello stato attivo il processo è in un punto del programma in cui non sta aspettando l'esecuzione di I/O al contrario nello stato di waiting il processo sta aspettando l'esecuzione di un I/O.

Nello stato attivo il processo si divide in due ulteriori fasi in quella di **RUNNING** e **READY.**

1)**Running→** Il processo in questo caso viene eseguito dalla CPU.

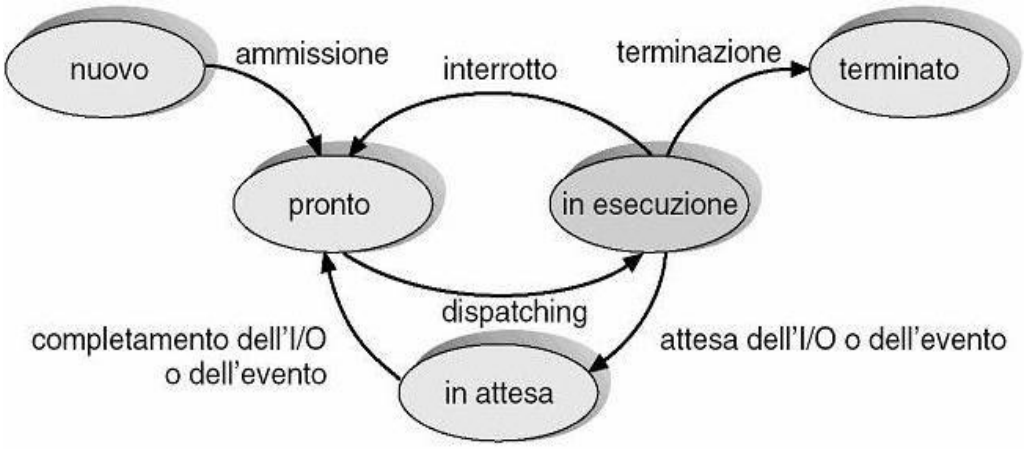
2)**Ready→** Il processo in questo caso è pronto ma non è eseguito dalla CPU perché quest'ultima è occupata.

Il passaggio dallo stato di ready allo stato di running avviene dopo un certo quanto di tempo.

Chi sceglie il processo che deve passare dalla fase di ready alla fase running? Lo SCHEDULER.

Chi attiva il context switch? Il DISPATCHER.

Entrambi sono componenti del sistema operativo



La terminazione può avvenire da un qualunque stato però solo dallo stato running ci sarà una terminazione volontaria.

NUOVO→ è uno stato molto transitorio. Il processo sta qui il tempo che serve per allocare le risorse.

TERMINATO→in questo stato il processo è terminato ma il sistema operativo ne tiene ancora traccia perché possono servire informazioni sulla terminazione.

IN CORSO→ il processo può essere eseguito, in esecuzione o bloccato in attesa di un evento.

STATI DEL PROCESSO P DOPO L'EVENTO

1)Il processo viene creato. (dallo stato new in cui sta per poco tempo va allo stato ready).

2)Il processo viene scelto dallo scheduler. (da stato ready a stato waiting).

3)Il processo chiama una read().un'operazione di I/O. (da stato ready a stato waiting).

4)Il processo passa nello stato di ready una volta che riceve l'interrupt.

5)Il processo riviene scelto dallo scheduler.

6)Il processo termina.

PCB

Il sistema operativo mantiene una scrittura chiamata PCB (Process control block). Essa è una tabella dei processi che contiene delle caratteristiche dello stato dei processi.

I contenuti tipici sono:

1)Registri CPU (per esempio PC, PSW, SP).

2)Dati dello scheduler (processo che ha la priorità, parametri dello scheduler).

3)Dati per accounting (tempo uso della CPU, tempo da quando il processo è iniziato).

4)Informazioni utili per protezione (ID utente, ID group).

Lo stato dei registri viene salvato nel PCB (Process control block) e ripristinato in occasione del context switch. In breve si può vedere come una tabella ma i PCB in diversi sistemi operativi sono delle strutture dati infatti possono essere nodi di liste che costituiscono la coda dei processi pronti.

MODELLO UTILIZZO CPU→ $CPU=1-p^n$ Ogni processo ha una certa attività continua di CPU (CPU burst) oppure un'attività continua di I/O (I/O burst).

p= proporzione di tempo di tutti i processi quando sono in I/O.

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 send e 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 riservare il bus e mantenerlo finché non sono concluse sia la lettura che la scrittura della variabile

Come può essere usata la TSL?

<u>Ingresso in sezione critica</u>

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){
Int item ;
While(True){
        Item=produce_item();
        If(Count ==N)
        Sleep();
        Insert_item(item);//inserisco  oggetto //
        Count++;            //nel buffer//
        If(Count==1)
          Wakeup(consumer);
        }
}
```

```
Void consumer(void){
 int item;
 While(True){
          if(Count==0)
            sleep();
          item=remve_item();
          Count--;
          if(count==N-1)
            wakeup(producer);
          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.

MECCANISMI DI SINCRONIZZAZIONE: I SEMAFORI

I semafori sono una primitiva evoluzione della Sleep() e della Wakeup(). Il semaforo è un tipo di dato astratto con associate le operazioni atomiche. Il semaforo ha tre operazioni possibili:

1)**INIT→** da usare solo una volta quando si crea il semaforo.

2)**DOWN→** la down controlla se il valore del semaforo è maggiore di 0 e se è così il valore del semaforo viene decrementato di 1 se invece è uguale a 0 si sospende il processo che ha fatto la down. Quindi si passa dallo stato di running allo stato di waiting e quindi lo scheduler non può selezionarlo.

3)**UP→** se ci sono processi in attesa per effetto di una down uno di questi termina l'attesa e conclude l'esecuzione di down. Se non c'erano processi in attesa cioè nessun processo sospeso sul quel semaforo allora il valore del semaforo viene incrementato.

UP e **DOWN** sono operazioni atomiche quindi i processi che le seguono non devono interrompersi.

COME SI FA A GARANTIRE LA MUTUA ESCLUSIONE CON I SEMAFORI?

Per garantire l'esecuzione in mutua esclusione di sezioni critiche si può utilizzare un semaforo *s*, inizializzato a 1 e condiviso da tutti i processi/thread che contengono sezioni critiche relative a determinate strutture dati.

verde per l'ingresso in sezione critica: vale 1

rosso: vale 0.

Un processo che vuole entrare in sezione critica (down(&s)):

•se trova il semaforo verde, lo imposta a rosso ed entra;

•se lo trova rosso viene sospeso

Quando il processo esce dalla sezione critica (up) il semaforo torna verde se non ci sono processi in attesa; o se ve ne sono, uno può entrare.

1.Il semaforo da verde (=1) diventa rosso (=0) grazie alla down().

2.Esegue il processo P2.

3.Poi il sistema operativo fa un context switch e passa a P2 ma dato che c'è il semaforo rosso P2 non si può proseguire e quindi la down() di P2 va allo stato di waiting.

4.= 5. Si effettua il processo P1.

6.Si effettua una UP e se ci sono processi in attesa vengono passati dallo stato di waiting a quello di ready.

7.Riviene eseguita una nuova down che rimanda da 1 a 0 il semaforo.

8. = 9. = 10. Si fa il processo P2

11. Si esegue la up per il processo P2. **(GUARDA SLIDE 4 BLOCCO SINCRONIZZAZIONE B)**

ALTRI TIPI DI SINCRONIZZAZIONE

Si vuole eseguire B in P_k solo dopo che A è stato eseguito in P_i.

P_I P_K

… …

A B

Usiamo un flag=0 così mi assicuro che P_K non possa andare avanti.

P_I P_K

… …

A B

UP(&flag) **DOWN(&flag)** (quindi con up e down si possono risolvere i problemi di mutua esclusione e di sincronizzazione).

IMPLEMENTAZIONE DEI SEMAFORI

Per una possibile realizzazione come chiamate di sistema, indichiamo con s.queue la lista di processi/thread associata a s.

down(&s) : if (s.val ==0)

{ inserisci il processo corrente p in s.queue;

cambia stato di p a *bloccato;*

scheduler(); dispatcher();

}

else s.val—

1)C'è una richiesta del sistema operativo e inserisco in coda il processo/thread in quel determinato semaforo

2)Cambia lo stato di questo processo P in bloccato(waiting)

3)Poi il sistema operativo deve schedulare un nuovo processo tra quelli disponibili e fare il dispatcher.

Else→ se il valore non è 0 decremento il semaforo.

up(&s) : if(s.queue non vuota)

{estrai un processo p da s.queue;

cambia stato di p a *pronto;*

}

else s.val++ ;

Controllo che questa coda non sia vuota e se non è vuota estraggo un processo da questa lista e lo porto dallo stato di waiting allo stato ready. Altrimenti se la coda è vuota incrementa il semaforo.

Non è detto che s.queue sia gestita con la **politica First-In-First-Out** ma è il modo più semplice per garantire attesa limitata.

politica First-In-First-Out→ il primo a sospendersi in coda è il primo ad essersi risvegliato.

Per una possibile realizzazione come chiamate di sistema, indichiamo con s.queue la lista di processi/thread associata a s

down(&s) : if (s.val ==0)

{inserisci il processo corrente p in s.queue;

cambia stato di p a *bloccato;*

scheduler(); dispatcher();

} else s.val—

down indivisibile : evita ad es. che con s.val==1 due processi «vedano» s.val>0 e procedano (violando mutua esclusione).

up(&s) : if (s.queue non vuota)

{estrai un processo p da s.queue;

cambia stato di p a *pronto;*

}

else

s.val++;

se non indivisibili, cosa potrebbe accadere con due up?Ad es., estraggono e mettono pronto lo stesso processo, un altro rimane inutilmente sospeso (viola progresso).

SEMAFORI IMPLEMENTATI COME SYSTEM CALL

La up() e la down() potrebbero essere implementate all'interno del sistema operativo, come system call. Per garantire che esse vengano eseguite in modo atomico, su un sistema uniprocessore si può usare la tecnica della disabilitazione degli interrupt (che in questo caso verrebbe usata esclusivamente dal S.O., non data in mano al programmatore).

Su un sistema multiprocessore, si può usare la soluzione al problema della mutua esclusione basato sulla istruzione TSL.

L'attesa attiva in questo caso può essere accettabile poiché la sezione critica è molto breve: se i processi/thread girano su processori diversi, uno fa attesa attiva al massimo per il tempo necessario agli altri per eseguire il codice della up o della down, che consiste in poche istruzioni, a differenza di quello di una sezione critica che inserisce chi scrive i programmi che usano up/down, la cui durata non è limitata.

NOTA SUI SEMAFORI E PROGRAMMAZIONE

Negli esempi tratti dal testo i semafori sono dichiarati così:

typedef int semaphore;

semaphore mutex = 1;

ma la notazione è usata solo per analogia con il C:

è vero che come indica un commento "i semafori sono un tipo speciale di interi", ma il loro essere speciali consiste nell'essere strutture che devono essere messe a disposizione dal sistema operativo, insieme con le operazioni predefinite (down e up) realizzate attraverso chiamate di sistema. Sono stati realizzati (ma poco usati) dei linguaggi per la programmazione concorrente in cui esiste veramente un tipo "semaphore" e il compilatore interagisce con il sistema operativo

SEMAFORI BINARI E SEMAFORI CONTATORE

Un semaforo binario può assumere solo i valori 0 o 1 (o si tratta di un semaforo generale, usato solo in modo che assuma tali valori). Ha le operazioni di Up() e Down(). Ma nel caso si tratti di un vero e proprio semaforo binario (non un semaforo generale usato come binario) e si esegue una up() quando s.val è già 1, questa non ha alcun effetto (oppure, come è comodo dire negli standard: l'effetto è indefinito, cioè: meglio scrivere i programmi in modo che non succeda, perché le conseguenze sono a scelta dell'implementazione). Tipicamente sono usati per risolvere il problema della mutua esclusione inizializzandoli a 1 e se sono usati in tale modo sono chiamati **mutex.**

Ma è un semaforo binario anche quello usato per "*B* in *P*k solo dopo *A* in *Pi"* e in questo caso il semaforo deve essere inizializzato a 0. Quindi per risolvere questo problema di precedenza deve essere inizializzato a 1.

Un semaforo contatore può assumere qualsiasi valore ? 0. Può essere usato ad esempio per il problema di sincronizzazione con un numero N di risorse da assegnare:

si inizializza a N, numero di "risorse" (in senso lato) disponibili

preleva = down(&s)

rilascia = up(&s)

N processi/threads possono superare down senza nessuna up, l'N+1 esimo viene sospeso

In generale, in ogni momento s.val è >= 0, e vale:

s.val = N – numero down completate + numero up completate cioè, intendendolo come numero di "risorse" disponibili: risorse totali – risorse prelevate + risorse rilasciate.

Utilizzando i semafori contatori possiamo risolvere il problema di produttore e consumatore. Come farlo?

1)Definiamo il numero totale di slot disponibili nel buffer.

2)Dichiaro un tipo semaforo di tipo intero per semplificazione è di tipo intero.

3)Dopodichè dichiaro 3 semafori uno di tipo binario(quindi un mutex) e mi serve per accedere al buffer in mutua esclusione e gli altri due sono due semafori che mi contano le risorse disponibili. Uno mi conta le risorse disponibili per il produttore cioè mi conta gli spazi vuoti nel buffer e all'inizio sarà uguale a N perché gli spazi vuoti saranno tutti mentre l'altro semaforo mi conta gli spazi pieni per il consumatore e all'inizio è uguale a 0 poiché il buffer è vuoto.

(GUARDA SLIDE 12 BLOCCO DI SLIDE SINCRONIZZAZIONE B)

Il produttore se il buffer è pieno vuol dire che in down(empty) in cui empty è zero farà in modo di sospendere il produttore.

Il consumatore se il buffer è vuoto vuol dire che in down(full) in cui full è zero farà in modo di sospendere il consumatore.

Problemi di sincronizzazione:il deadlock (stallo)

Deadlock (stallo) – si verifica quando due o più processi attendono indefinitamente il verificarsi di un evento che può essere causato solo da uno dei processi stessi

Es.: siano *S* e *Q* due semafori inizializzati a 1

P0 **P1**

1. down(&S) **2. down(&Q)**

4. down(&Q) **3. down(&S)**

.. **..**

up(&S) **up(&Q)**

up(&Q) **up(&S)**

In questo caso P1 è bloccato su S in attesa di P0 ;P0 è bloccato su Q in attesa di P1.

Problemi di sincronizzazione:la starvation

Starvation (morte di fame) → si verifica quando un processo P rimane per sempre in una coda d'attesa (per es. di un semaforo) perché altri processi vengono ripetutamente risvegliati prima di **P. Esempio**→ 3 (o più) processi usano un semaforo per la mutua esclusione. Se la coda del semaforo viene **gestita con politica FIFO (First In First Out)**, l'attesa di un processo in coda al semaforo è certamente limitata. Se la coda venisse gestita con politica LIFO (Last In First Out), è possibile che un processo rimanga in coda all'infinito (es.: P1 è in coda, P3 è in s.c., P2 arriva in coda, quando P3 lascia la s.c., P2 passa avanti a P1; se poi P3 arriva in coda prima che P2 lasci la s.c., P3 passa avanti a P1, ecc… all'infinito)

Problemi di sincronizzazione:filosofi a cena

I filosofi pensano, ma ogni tanto mangiano e per mangiare necessitano di 2 forchette, quella alla propria sinistra e quella alla propria destra. Rappresenta il caso di vari processi che per una parte del loro codice (pensare) non devono sincronizzarsi; per un'altra (mangiare) devono utilizzare più risorse in mutua esclusione.

(STAMPA SLIDE 16 BLOCCO SLIDE SINCRONIZZAZIONE B)

Può portare ad una situazione di deadlock: se tutti i filosofi prelevano la forchetta di sinistra e poi si sospendono attendendo di acquisire la forchetta di destra, rimarranno bloccati in questo stato indefinitamente

Riprendiamo la definizione di deadlock: un insieme di processi in attesa di un evento che può essere provocato solo da un processo nell'insieme stesso, in questo caso:

Il filosofo 0 supera la down(&fork[0]) ma rimane sospeso sulla down(&fork[1]);

il filosofo 1 supera la down(&fork[1]) ma rimane sospeso sulla down(&fork[2]),

il filosofo N supera la down(&fork[N]) ma rimane sospeso sulla down(&fork[0])

(STAMPA SLIDE 18 BLOCCO SLIDE SINCRONIZZAZIONE B)

Una possibile soluzione consiste nel cambiare l'ordine di acquisizione delle forchette ad uno dei filosofi: per esempio il filosofo 4 potrebbe acquisire prima la forchetta di destra (0) e poi quella di sinistra (4). Un'altra possibile soluzione consiste nell'ammettere al più 4 filosofi nella fase di acquisizione forchette, utilizzando un semaforo contatore inizializzato a 4, e aggiungendo down(&count) prima dell'acquisizione delle forchette ed una up(&count) subito dopo l'acquisizione di entrambe le forchette. Una terza soluzione possibile consiste nell'acquisire le risorse (forchette) contemporaneamente, anziché una alla volta. Usiamo un array di N semafori s[N]. Quando il processo i non può acquisire le forchette (perché non sono entrambe disponibili) si sospende sul semaforo s[i]. Quando un semaforo viene usato in questo modo, si chiama semaforo privato.

SEMAFORI PRIVATI

I semafori privati sono detti tali per il modo in cui sono usati infatti il meccanismo messo a disposizione dalle funzioni è lo stesso, senza nessun controllo su quale processo/threads usa i semafori e come.

QUALI SONO LE PROPRIETA' AFFINCHE' UN SEMAFORO SIA PRIVATO?

Un semaforo è privato quando è assicurato a uno specifico processo oppure ad una classe di processi. Questo vuol dire che nel problema del produttore-consumatore potrebbe avere più produttori e più consumatori e ci potrebbe essere un semaforo che si occupa dei consumatori e un altro semaforo ai produttori. Questi semafori sono inizializzati a 0.

Solo il processo P (o un processo della classe associata al semaforo) esegue down (&s priv_P); la esegue quando deve attendere che diventi vera una condizione (booleana) di sincronizzazione.

Qualsiasi processo (P incluso) può eseguire up (&s_priv_P) se serve svegliare P, o serve non farlo sospendere se fa down, perché è vera la condizione di sincronizzazione.

(STAMPA SLIDE 21-22 BLOCCO SLIDE SINCRONIZZAZIONE B)

La soluzione può essere elaborata per garantire l'assenza di starvation, per esempio non permettendo ad un filosofo di mangiare più di k volte di fila se un suo vicino è affamato:

• ogni volta che il filosofo i preleva la forchetta che serve anche a un vicino affamato si incrementa un contatore

• quando il contatore arriva a k il filosofo si sospende

IL PROBLEMA DEI LETTORI E SCRITTORI

Supponiamo di avere una struttura dati che viene utilizzata da molti utenti contemporaneamente (come un database, anche se qui non parliamo di DBMS in cui ovviamente questi problemi esistono). Le operazioni che gli utenti possono richiedere sono di due tipi: consultazione (lettura) o modifica (scrittura). Per assicurare un utilizzo efficiente del sistema vogliamo che le consultazioni(lettura del database) possano procedere in parallelo.

Tuttavia per assicurare la consistenza della struttura dati e delle informazioni ottenute durante la consultazione, occorre garantire che ogni modifica avvenga in mutua esclusione con qualsiasi altra operazione (lettura o scrittura).

L'idea è che i lettori li faccio proseguire contemporaneamente, gli scrittori uno alla volta mentre un lettore e uno scrittore deve prima proseguire lo scrittore e poi il lettore così legge l'informazione aggiornata. Quindi n lettori possono proseguire contemporaneamente mentre gli scrittori uno alla volta.

Come la utilizziamo?

• una variabile condivisa **rc** (inizialmente =0) che indica quanti lettori stanno usando il database

• due semafori:

• **db** inizializzato a 1 per accedere al database in «mutua» esclusione (in questo caso: 1 scrittore oppure N lettori). Questo semaforo serve quindi per proteggere il database.

• **mutex** inizializzato a 1 associato alla variabile rc.

```
void writer(void)
{
        While(TRUE)
        {
                think_up_data();
inzio scrittura down(&db);
                write_data_base();
fine scrittura   up(&db);
        }
}
```

(STAMPA SLIDE 7 BLOCCO SLIDE SINCRONIZZAZIONE C)

Questa soluzione non garantisce l'assenza di starvation, es. per uno scrittore in attesa se c'è un flusso continuo di lettori.

Anche in questo caso si può realizzare una politica più sofisticata utilizzando semafori privati diversi per i lettori e per gli scrittori.Per evitare la starvation:

• se un lettore arriva quando ci sono degli scrittori in attesa, si blocca (per non superarli)

• quando un lettore termina di usare il DB e non vi sono altri lettori che operano sul DB, sveglia uno scrittore in attesa (se c'è)

• quando uno scrittore termina di usare il DB, se vi sono lettori in attesa li sveglia tutti, se no sveglia uno scrittore

Usiamo due semafori privati **sem_priv_lettori e sem_priv_scrittori**, entrambi inizializzati a 0, che servono ai lettori o agli scrittori per sospendersi quando non possono accedere al DB.

(STAMPA SLIDE 9 BLOCCO SLIDE SINCRONIZZAZIONE C)