Modelli di interazione

Individuabili anche in macchine monoprocessore basate sul modello di von Neumann, ovvero macchine SISD.

- A memoria comune (ambiente globale)

Memoria comune per tutti i processi. Usato per macchine monoprocessore multitasking. Complesso da implementare in sistemi distribuiti. Rischio problemi quando 2 processi richiedono accesso a stessa risorsa condivisa.

- A scambio di messaggi

Usata in sistemi distribuiti (+ macchine ma in luoghi diversi) e macchine comunicano con messaggi. Utilizzate le funzioni Send(dest, mess) e Recieve(source, mess). Mess è composto da: ***Header****(dest, source, type, priority, length…)* e **Messaggio** (string). Queste possono essere funzioni **bloccanti (sincrone)** o **non** **bloccanti (asincrone)**.

|  |  |  |
| --- | --- | --- |
| Funzione | **Bloccante/Sincrona** | **Non bloccante/Asincrona** |
| **Send** | Quando processo che la esegue si blocca finché non confermato che mess è ricevuto. | Dopo mess, prosegue esecuzione. |
| **Recieve** | Quando processo si ferma in attesa di un messaggio dal mittente. | Non attende mess: 1) mess inviato prima di Recieve() (riceve se fa Recieve), 2) mess inviato dopo Recieve() (perdita mess) |

**Rendez-vous** --> solo per comunicazione **sincrona/bloccante**

Incontro tra 2 o + processi in un punto specifico del codice nel quale questi si scambiano messaggi tramite un canale.

- **Stretto**: Si limita alla trasmissione del messaggio da mittente a destinatario.

- **Esteso**: Una volta inviato il messaggio, il mittente aspetta una risposta (*ack*) da parte del destinatario.

**Comunicazione**

- **Asimmetrica**: **Mittente** **nomina** **esplicitamente** **destinatario**, ma destinatario **non** nomina esplicitamente mittente.

- **Simmetrica**: Entrambi si nominano esplicitamente.

**Indirizzamento**

- **diretto:**

P0 (in pc 0) fa la Send direttamente a P1 (pc 1) che fa la Recieve e riceve il mess direttamente.

- **indiretto:**

P0 fa la Send in una **coda**, dove possono venir salvati multipli mess. P1 con Recieve prende messaggi 1 per volta.

Gestori (o allocatori)

Segmenti di codice che gestiscono tutte le richieste fatte da diversi processi che richiedono la risorsa a cui è assegnato; quindi questo assegna o nega la risorsa in base a stato e se sta già venendo usata.

Compiti del gestore:

- Tenere aggiornato lo stato di allocazione della risorsa,

- Fornire a processi che richiedono risorsa i meccanismi per accedervi,

- Implementare la strategia di allocazione della risorsa definendo a quale processo e per quanto tempo.

Risorse

Qualunque oggetto, fisico o logico, di cui un processo necessita per portare a termine il suo compito.

|  |  |  |
| --- | --- | --- |
| Risorse | **Dedicate** (visibili sempre da 1 processo) | **Condivise** (visibili sempre da + processi) |
| **Statiche** | **Private** | **Comuni** |
| **Dinamiche** | **Comuni** | **Comuni** |

**Statiche**: gestore = programmatore, perché definite prima che programma starta.

**Dedicate**: gestore = SO, perché utilizzo esclusivo di 1 solo processo e programmatore non serve.

**Condivise**: gestore = ?, programmatore definisce solo regole di visibilità (pubbliche o private) e modalità di sync.

Anche un’**istanza di struttura dati allocata in memoria comune** è una **risorsa**.

Dedicate --> sincronizzazione non necessaria.

Condivise --> sincronizzazione necessaria, assicurarsi che accessi eseguiti in maniera **non divisibile** (mutex).

Competizione

Interazione tra processi PREVEDIBILE e INDESIDERATA.

- risorse **condivise statiche**: competizione al momento di accesso a risorsa

- risorse **dedicate dinamiche**: competizione al momento della richiesta di utilizzo

Soluzione: sincronizzazione **indiretta** o **implicita**.

Cooperazione

Interazione tra processi PREVEDIBILE e DESIDERATA (insita in logica del programma).

- risorse **condivise statiche**: logica di **produttore e consumatore** (produttore scrive, consumatore legge da risorsa)

- risorse **dedicate dinamiche**: logica di **lettore e scrittore**, (cooperazione c’è quando gestore **sa** che processi cooperano, quindi non cancella risultato prodotto in risorsa da P1 cosicché P2 possa usarlo.

Soluzione: sincronizzazione **diretta** o **esplicita**.

Interferenza

Interazione tra processi IMPREVEDIBILE e INDESIDERATA. Causata da:

- competizione tra + processi che utilizzano risorse condivise senza opportune autorizzazioni,

- errata soluzione per problema di competizione e cooperazione.

Errori in programmi concorrenti

Oltre a correttezza **logica** dei programmi, verificare anche quella **temporale**, infatti se in programmi sequenziali i risultati variavano solo in base a un cambiamento di input, ora con programmazione concorrente, se non uso tecniche di **sincronizzazione** adatte, risultati di un’elaborazione rischiano di variare anche solo in **base al tempo** (tipo quale processo viene eseguito prima). Un obiettivo di questa programmazione è **sfruttare al max la CPU**, quindi non posso inserire **ritardi** **in** **esecuzione** (cmq non garantita la correttezza dell’esecuzione) ma è una tecnica inaccettabile.

Vincoli

- Determinismo: i risultati devono essere uguali per ogni esecuzione,

- Timing costraints: i risultati vanno prodotti entro certi limiti di tempo (***deadlines***).

Caratteristiche

1) Irriproducibili, ovvero possono verificarsi con alcune sequenze e non con altre;

2) Indeterminati, ovvero esito ed effetti dipendono dalla sequenza;

3) Latenti, ovvero possono presentarsi soltanto con sequenze rare;

4) Difficili da verificare testare, poiché tecniche di verifica e testing si basano sulla riproducibilità del comportamento.

Overlapping

Quando più processi vengono eseguiti **contemporaneamente** e la loro esecuzione si **sovrappone** **nel** **tempo** (ciò grazie a CPU multiple di **sistemi** **multiprocessore**). Questo è detto **parallelismo fisico o reale**.

Interleaving

Quando (in macchine **monoprocessore**), dato che non è possibile il parallelismo fisico, i processi vengono eseguiti **alternati nel tempo** ma con velocità tali da dare l’impressione di avere multiprocessore. Ciò si presenta come un’alternanza casuale nell’esecuzione di processi in base alle politiche di **scheduling**. Questo è parallelismo **virtuale**.

Interleaving e overlapping possono essere **combinati** in presenza di + macchine multitasking.

Condizioni di Bernstein

1) rango(Ia) ∩ rango(Ib) = ∅

2) rango(Ia) ∩ dominio(Ib) = ∅

3) dominio(Ia) ∩ rango(Ib) = ∅

Dominio

È 1 o + aree di memoria di un’istruzione o procedura (variabili il cui valore è usato per operazioni e altro…)

Rango

È il contenuto 1 o + aree di memoria di un’istruzione o procedura (variabili il cui contenuto è modificato/sovrascritto)

Sezione critica

Sequenza di istruzioni tramite la quale un processo accede e modifica un insieme di variabili condivise.

Mutua esclusione

Modalità di accesso a risorse condivise per cui, ad ogni istante, al max 1 processo per volta può accedervi.

Secondo la sua regola, una risorsa o è libera o è assegnata a 1 solo processo.

Regole

- Nessuna coppia di processi può trovarsi **simultaneamente** nella sezione critica;

- L'accesso alla regione critica **non** deve essere **regolato** da alcuna **assunzione** **temporale** o dal **n°** di **CPU**;

- Nessun processo in esecuzione al di fuori della regione critica può **bloccare** un processo che prova a entrarvi;

- Nessun processo deve attendere **indefinitamente** per poter accedere alla regione critica.

*Busy* *waiting*

È una soluzione adottabile al fine di prevenire l’accesso a sezioni critiche (risorse già assegnate), che consiste nel far aspettare i processi che provano ad accedervi finché la risorsa non viene rilasciata.

*Deadlock* (o blocco multiplo/stallo)

Quando 2 o + processi rimangono in attesa di eventi che non accadranno mai a causa di condizioni cicliche nel possesso e richiesta di risorse.

(Quando un processo sta usando una risorsa e ci mette troppo tempo perché è molto lento, quindi quello che aspetta va in *deadlock*, ovvero attende lunghi periodi finché il processo che sta usando al risorsa non la rilascia).

*Starvation* (o blocco individuale)

Un processo “muore di fame”, ovvero, rimane in attesa di un evento che non accadrà mai, quindi non può completare la sua esecuzione, e dato che non può terminare, consuma risorse.

(Processi con priorità + alta continuano ad occupare una risorsa lasciando quello con – priorità in attesa infinita).

*Safety*

Condizione che deve sempre verificarsi x buona esecuzione di un processo o che non deve mai verificarsi per la sicurezza del sistema operativo.

Questa proprietà è garantita dai meccanismi di **sincronizzazione** e sancisce che i processi non devono **mai** **interferire** nell’accesso a risorse condivise in modo che le risorse si trovino sempre in stato **consistente**.

(Un programma rispetta la condizione di *safety* quando non permette a dei processi di accedere a una risorsa in uso)

*Liveness*

Sancisce la necessità che i processi vengano **portati a termine** (prima o poi) al fine di evitare il *deadlock*.

*Fairness* (o equità)

S.O. deve **sempre** mandare in esecuzione **ogni processo** (facendoli evolvere tutti + o – per lo stesso tempo) soddisfacendo (prima o poi) tutte le richieste di esecuzione così da **non causare** **starvation**.

Comunicazione tra processi

Comunicazione con struct Buffer condiviso: P0 scrive dati e P1 legge dati. Questi 2 non possono usare buffer contemporaneamente, quindi si usa un bool **StatoBuffer**. True: P0 sta scrivendo | False: risorsa libera.

**Problema**: **busy**-**waiting**, se P0 scrive, P1 consuma memoria continuando a testare il while.

Semafori

Sono variabili contenenti: un **contatore** int e una **coda** **associata**. Tramite essi i processi comunicano scambiandosi **segnali**. Solo 2 funzioni: **Wait**() --> riceve i segnali di un semaforo. **Signal**() --> invia segnale e contatore lo conta. Quando un semaforo non riceve segnali (perché non arrivano) il processo va in **wait** e spostato in coda associata.

Con contatore a 1 --> Mutua esclusione tra processi. Con contatore a 0 --> sincronizzazione tra processi.

Tecnica produttore-consumatore

Tecnica usata quando 1 o + processi (produttori) producono dati e li inseriscono in una coda/buffer condivisi, mentre altri processi (consumatori) accedono alla coda/buffer per prelevare i dati prodotti dai produttori.

Sincronizzazione: mutua esclusione --> solo 1 processo (non importa se produttore o consumatore) può leggere o scrivere per volta nella coda/buffer.

Tecnica lettore-scrittore

Tecnica usata quando si ha una risorsa condivisa (tipo database) che può essere letta da + processi (lettori) ma può essere scritta solo da 1 processo per volta (scrittore).

Sincronizzazione: mutex? --> lettori possono leggere come e quando vogliono, ma quando uno scrittore deve scrivere, solo lui ha accesso alla risorsa finché non finisce.

Istruzioni (primitiva) atomiche

Sono delle istruzioni che non possono essere interrotte durante la loro esecuzione, ed essendo **non** **interrompibili** vanno programmate come **sezioni** **critiche**. (Tipo Wait e Signal dei semafori)

Algoritmi e procedure

|  |  |  |
| --- | --- | --- |
| P0  {  Conto Dati0;  do  {  EstraiDatiCliente();  ScriviDatiBuffer();  } while (Clienti.Count > 0);  } | P1  {  Conto Dati1;  do  {  LeggiDatiBuffer();  StampaDatiCliente();  } while (Clienti.Count > 0);  } | **Problema**: Se 1 dei 2 processi cerca di usare il Buffer quando è in uso, si rischiano interferenze e risultati sbagliati.  **Soluzione**: variabile booleana che indica stato del Buffer. |
| P0  {  do  {  ProduceDatiClienti();  while (StatoBuffer == true);  ScriviDatiBuffer();  StatoBuffer = true;  } while (Clienti.Count > 0);  } | P1  {  do  {  while (StatoBuffer == false);  LeggiDatiBuffer();  StatoBuffer = false;  StampaDatiCliente();  } while (Clienti.Count > 0);  } | **Problema**: busy-waiting  Se P0 scrive, P1 aspetta indefinitamente continuando a sprecare risorse testando la condizione in while.  **Soluzione**: semafori. |
| void Signal(Semaforo s)  {  Processo p;  s.Count++;  if (s.Count <= 0)  {  p = s.Coda.Estrai();  p.Resume();  }  } | void Wait(Semaforo s)  {  s.Count--;  if (s.Count < 0)  {  s.Coda.Add();  p.Suspend();  }  } | Class Semaforo  {  public int Count;  public Queue Coda;  }  If (Count <= 0) --> |Count| = n proc in queue  Else (Count > 0) --> n signal sent non ricevuti |

|  |  |  |
| --- | --- | --- |
| P0  {  do  {  Wait(s);  // Sezione Critica;  Signal(s);  } while (true);  } | P1  {  do  {  Wait(s);  // Sezione Critica;  Signal(s);  } while (true);  } | Esempio semafori mutua esclusione  void main()  {  Semaforo s ( Count = 1 );  } |
| P0  {  do  {  // AzioniPrePartenza();  Wait(s);  // AzioniDopoPartenza();  } while (true);  } | P1  {  do  {  // RilevaEvento();  Signal(s);  // AltreOperazioni();  } while (true);  } | Esempio semafori sincronizzazione  void main()  {  Semaforo s ( Count = 0 );  } |
| void Produttore()  {  string Dato;  do  { Produce ( Dato );  Wait ( Prelevato );  Buffer = Dato;  Signal ( Depositato );  } while (Dati.Count > 0);  } | void Consumatore()  {  string Dato;  do  { Wait ( Depositato );  Dato = Buffer;  Signal ( Prelevato );  Consuma ( Dato );  } while (Dati.Count > 0);  } | Esempio produttore-consumatore  void main()  {  Semaforo Prelevato ( Count = 1 );  Semaforo Depositato ( Count = 0 );  string Buffer;  } |
| void Scrittore()  {  do  {  Wait (Scrivi);  ScriviDati();  Signal (Scrivi);  } while (true);  } | void Lettore()  {  do {  Wait (Mutex);  Lettori++;  if (Lettori == 1)  Wait(Scrivi); Signal(Mutex);  LeggiDati();  Wait ( Mutex );  Lettori--;  if (Lettori == 0)  Signal(Scrivi); Signal(Mutex);  } while (true);  } | Esempio lettore-scrittore  void main()  {  int Lettori = 0;  Semaforo Scrivi ( Count = 1 );  Semaforo Mutex ( Count = 1 );  } |
| void Produttore()  {  string Dato;  do  {  Produce( Dato );  Send ( Consumatore, Dato );  } while (Dati.Count > 0);  } | void Consumatore()  {  string Dato;  do  {  Recieve ( Produttore, Dato );  Consuma(Dato);  } while (Dati.Count > 0);  } | Esempio sync rendez-vous  void main()  {  Semaforo Prelevato ( Count = 1 );  Semaforo Depositato ( Count = 0 );  string Buffer;  } |
| void P0  {  string Msg;  while(true)  {  …  Recieve ( Casella, Msg );  Sezione Critica;  Send ( Casella, Msg );  …  }  } | void P1  {  string Msg;  while(true)  {  …  Recieve ( Casella, Msg );  Sezione Critica;  Send ( Casella, Msg );  …  }  } | Esempio mutex scambio di messaggi  void main()  {  Mailbox Casella;  Send ( Casella, “ “ );  } |