Corso di

Sistemi per l'Elaborazione delle Informazioni

Sincronizzazione dei processi

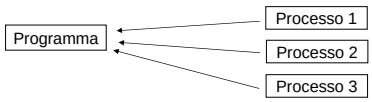
prof. Bruno Carpentieri

Di Giorgio Domenico	0521000063
Malinconico Cris	0521000118

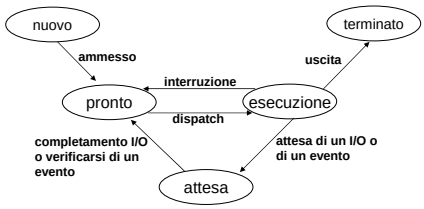
Sincronizzazione dei processi

Processo

- Informalmente un processo è un programma in esecuzione e non il suo codice (sezione di testo).
- Errore se si associa un programma (entità passiva) ad un processo (entità attiva).
- Esso rappresenta qualcosa di più del codice di un programma, infatti comprende:
  1. Il valore del PC (Program Counter)
  2. Il contenuto dei registri della CPU
  3. Il contenuto dello Stack con relativa sezione dati



Stati di un processo



nuovo: il processo viene creato

esecuzione: un'unità di elaborazione esegue le istruzioni del relativo programma

attesa: il processo attende che si verifichi qualche evento (ricezione di un segnale o completamento di un'op. di I/O)

pronto: il processo attende di essere assegnato ad un'unità di elaborazione

terminato: il processo termina l'esecuzione

Tipi di Processi

Processi indipendenti: ogni processo non può influire su altri processi nel sistema o subirne l'influsso, chiaramente, essi non condividono dati.

Processi cooperanti: ogni processo può influenzare o essere influenzato da altri processi in esecuzione nel sistema, chiaramente, essi possono o condividere direttamente uno spazio logico di indirizzi (codice e dati) oppure condividere dati mediante file.

PROBLEMA: l'esecuzione concorrente di processi cooperanti può condurre a situazioni d'inconsistenza dei dati.

Concorrenza

- **Multiprogramazione: processi multipli in un sistema monoprocessore,**
- **Multiprocessing: processi multipli in un sistema multiprocessore**
- **Processi Distribuiti: gestione di processi multipli eseguiti su sistemi distribuiti**

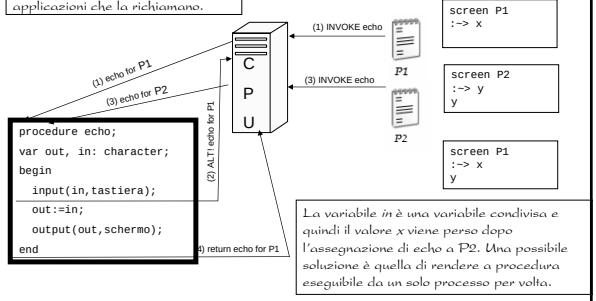
Temi

Aspetti

- **Applicazioni Multiple: divisine dinamica del tempo nella multiprogrammazione**
- **Applicazioni Strutturate:applicazioni programmate efficacemente come insiemi di processi concorrenti**
- **Struttura del S.O.: implementazione dei S.O. come insiemi di processi**

Un semplice esempio

Un'unica copia su un sistema multiprogrammato con più applicazioni che la richiamano.



La variabile *in* è una variabile condivisa e quindi il valore *x* viene perso dopo l'assegnazione di echo a P2. Una possibile soluzione è quella di rendere a procedura eseguibile da un solo processo per volta.

Interazione fra processi

Grado di conoscenza	*Relazione*	*Influenza che un processo ha sugli altri*	*Possibili problemi di controllo*
Processi che non si vedono tra di loro	Competizione	•I risultati di un processo non dipendono dalle informazioni ottenute dagli altri •Il tempo di esecuzione di processi può cambiare	•Mutua esclusione •Stallo (risorse riutilizzabili) •Starvation
Processi che vedono altri processi indirettamente(es. oggetti condivisi)	Cooperazione tramite condivisione	•I risultati di un processo possono dipendere dalle informazioni ottenute dagli altri •Il tempo di esecuzione dei processi può cambiare	•Mutua esclusione •Stallo (risorse riutilizzabili) •Starvation •Coerenza dei Dati
Processi che vedono altri processi direttamente(usano primitive di comunicazione)	Cooperazione tramite comunicazione	•I risultati di un processo possono dipendere dalle informazioni ottenute dagli altri •Il tempo di esecuzione dei processi può cambiare	•Stallo (risorse riutilizzabili) •Starvation

Produttore/Consumatore

Problema

- Un processo "produttore" produce informazioni che sono consumate da un processo "consumatore".
- Per permettere un'esecuzione concorrente dei processi, occorre disporre di un vettore di elementi che possano essere inseriti dal produttore e prelevati dal consumatore.
- Un produttore può produrre un elemento mentre il consumatore ne sta consumando un altro.
- Essi devono essere sincronizzati in modo che il consumatore non tenti di consumare un elemento che non è stato ancora prodotto.

Problema produttore-consumatore II

Osservazioni

- Se la memoria è illimitata non abbiamo limiti alla dimensione del vettore, il consumatore può trovarsi ad attendere nuovi processi, ma il produttore può sempre produrne.
- Se la memoria è limitata, la dimensione del vettore deve essere fissata. In questo caso il consumatore deve attendere se il vettore è vuoto e il produttore se è pieno.
- Il vettore può essere fornito dal sistema operativo attraverso l'uso di una funzione di comunicazione tra processi (Interprocess Comunication IPC), oppure codificato esplicitamente, facendo uso di memoria condivisa, dal programma dell'applicazione.

Memoria limitata

```
#define BUFFER_SIZE 10
typedef struct {
. . .
} item;
item buffer[BUFFER_SIZE];
int inserisci = 0;
int preleva = 0;
```

indica la successiva posizione libera nel vettore

indica la prima posizione piena nel vettore

Produttore/Consumatore

Produttore

```
item nextProduced;
while (1) {
  while (((inserisci +1) % BUFFER_SIZE)==preleva);
    /* il vettore è pieno */
  buffer[inserisci] = nextProduced;
  inserisci = (inserisci + 1) % BUFFER_SIZE;
}
```

Consumatore

```
item nextConsumed;
while (1) {
  while (inserisci==preleva);
    /* il vettore è vuoto*/
  nextConsumed = buffer[preleva];
  preleva = (preleva + 1) % BUFFER_SIZE;
}
```

Produttore/Consumatore

Osservazioni

- La soluzione del problema dei produttori e dei consumatori con memoria limitata, facendo uso di memoria condivisa,consente la presenza contemporanea nel vettore di non più di *n–1* elementi.
- Una soluzione per utilizzare tutti gli *n* elementi del vettore non è semplice:
  - *Potremmo aggiungere una variabile intera, counter, inizializzata a 0, che si incrementa ogni volta che si inserisce un nuovo elemento nel vettore (e si decrementa ogni volta che si preleva un elemento dal vettore).*

Produttore/Consumatore II

Produttore

```
item nextProduced;
while (1) {
  while (counter == BUFFER_SIZE);
    /* il vettore è pieno */
  buffer[in] = nextProduced;
  in = (in + 1) % BUFFER_SIZE;
  counter++;
}
```

Consumatore

```
item nextConsumed;
while (1) {
  while (counter == 0);
    /*il vettore è vuoto */
  nextConsumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  counter--;
}
```

Transazioni atomiche

- Gli statement ***counter++*** e ***counter--*** devono essere eseguiti atomicamente.
- Una operazione è atomica se non può essere interrotta fino al suo completamento.

counter++

```
register1 = counter
register1 = register1 + 1
counter = register1
```

counter--

```
register2 = counter
register2 = register2 – 1
counter = register2
```

Interleaving

- *Se il produttore ed il consumatore tentano di accedere concorrentemente al buffer, le esecuzioni in linguaggio macchina dell'incremento e del decremento del contatore potrebbero interfogliarsi (**interleaving**).*
- *Il risultato dell'interfogliamento dipende da come produttore e consumatore sono schedulati.*

Esempio

```
produttore:  register1 = counter       (register1 = 5)
produttore:  register1 = register1 + 1 (register1 = 6)
consumatore: register2 = counter       (register2 = 5)
consumatore: register2 = register2 – 1 (register2 = 4)
produttore:  counter   = register1     ( counter = 6 )
consumatore: counter   = register2     ( counter = 4 )
```

Il valore di **counter** è quindi 4, ma il risultato corretto è 5.

Race Condition

Race condition: Situazione in cui più processi accedono e modificano gli stessi dati concorrentemente con risultati dipendenti dall'ordine degli accessi.

ESEMPI

Nei S.O. molto spesso capitano situazioni di race condition dovute a componenti che operano su dati condivisi.

STRATEGIA

Garantire l'esecuzione di un solo processo alla volta operante su dati condivisi mediante tecniche di ***SINCRONIZZAZIONE*** e ***COORDINAZIONE*** dei processi stessi

Problema della sezione critica

Situazioni di race condition conducono al concetto di ***sezione critica***

Definizione

Sezione critica

segmento di codice, nel quale il processo può modificare variabili comuni come aggiornare una tabella o scrivere in un file condiviso .

Problema generico

1. N processi P_0 P_1… P_{n-1} che utilizzano dati condivisi
2. Ogni Processo P_i ha una propria sezione critica

SOLUZIONE

Realizzare un protocollo, che consenta ai processi di poter cooperare, in cui sono verificate 3 proprietà.

Sezione critica

1. ***Mutua esclusione.*** *Se il processo P_i è nella sua sezione critica, allora nessun altro processo può essere nella propria* ***sezione critica****.*
2. ***Progresso.*** *Se nessun processo è in esecuzione nella sua sezione critica ed esiste qualche processo che desidera entrarvi, solo i processi che sono fuori dalle rispettive* ***sezioni non critiche*** *possono partecipare alla decisione di stabilire quale processo può entrare per primo nella propria sezione critica; tale scelta non può essere rimandata indefinitamente.*
3. ***Attesa Limitata.*** *Se un processo ha già richiesto l'ingresso nella sua* ***sezione critica****, esiste un limite al numero di volte che si consente ad altri processi di entrare nelle rispettive sezioni critiche prima che si accordi la richiesta del primo processo.*

Struttura di un processo

Permesso di entrare nella sezione critica

Solo 2 processi, P_0 e P_1

Struttura generale di un tipico processo P_i

```
do {
    sezione di ingresso
    sezione critica
    sezione di uscita
    sezione non critica
} while (1);
```

Istruzioni successive alla sezione critica

Osservazione: I processi possono utilizzare delle variabili condivise per sincronizzarsi.

Algoritmo di Dekker (1° tentativo)

- ✓ **Variabile condivisa:** int turno
- ✓ **Osservazione:** inizialmente turno=0;

Processo P_0

```
do{
 while(turno!=0)do{nothing}
     <sezione critica>
      turno:=1;
     <sezione non critica>
 }while(1);
```

Processo P_1

```
do{
 while(turno!=0)do{nothing}
     <sezione critica>
      turno:=0;
     <sezione non critica>
 }while(1);
```

Vantaggi

1. Rispetta la mutua esclusione.

Svantaggi

1. Stretta alternanza dei processi (progresso non soddisfatto).
2. Se un processo fallisce l'altro rimane bloccato.

Algoritmo di Dekker (2° tentativo)

- ✓ **Osservazione:** *per un processo è utile avere info. sullo stato dell'altro processo.*
- ✓ **Variabile condivisa:** *boolean flag[2]={false,false}*

Processo P_0

```
do{
while(flag[1])do{nothing}
     flag[0]:=true;
     <sezione critica>
     flag[0]:=false;
     <sezione non critica>
 }while(1);
```

Processo P_1

```
do{
while(flag[0])do{nothing}
     flag[1]:=true;
     <sezione critica>
     flag[1]:=false;
     <sezione non critica>
 }while(1);
```

Vantaggi

1. Rispetta la mutua esclusione anche se non sempre. (Nello stesso istante entrambi i processi eseguono il ciclo while).

Svantaggi

1. Il progresso non è soddisfatto
2. Se un processo fallisce mentre pone a falso il proprio flag l'altro rimane bloccato per sempre.

Algoritmo di Dekker (3° tentativo)

- ✓ **Osservazione:** *poniamo al di fuori del while l'assegnamento di verità.*
- ✓ **Variabile condivisa:** *boolean flag[2]={false,false}*

Processo P_0

```
do{
flag[0]:=true;
while(flag[1])do{nothing}
     <sezione critica>
     flag[0]:=false;
     <sezione non critica>
 }while(1);
```

Processo P_1

```
do{
flag[1]:=true;
while(flag[0])do{nothing}
     <sezione critica>
     flag[1]:=false;
     <sezione non critica>
 }while(1);
```

Vantaggi

1. Rispetta in ogni caso la mutua esclusione.

Svantaggi

1. Se nello stesso istante entrambi i processi eseguono flag[i]=true si bloccano entrambi (stallo).

Algoritmo di Dekker (4° tentativo)

- ✓ **Osservazione:** *l'assegnamento di verità indica solo il desiderio di entrare nella sezione critica.*
- ✓ **Variabile condivisa:** *boolean flag[2]={false,false}*

Processo P_0

```
do{
flag[0]:=true;
while(flag[1])do{
     flag[0]:=false;
     <breve pausa>
     flag[0]:=true;
}
<sezione critica>
flag[0]:=false;
<sezione non critica>
  }while(1);
```

Processo P_1

```
do{
flag[1]:=true;
while(flag[0])do{
     flag[1]:=false;
     <breve pausa>
     flag[1]:=true;
}
<sezione critica>
flag[1]:=false;
<sezione non critica>
  }while(1);
```

Vantaggi

1. Garantita la mutua esclusione

Svantaggi

1. Lo stallo può verificarsi anche se in situazioni piuttosto improbabili (<breve pausa> identica).

Algoritmo di Dekker (corretto)

- ✓ **Osservazione:** *la possibilità di esprimere il desiderio di entrare è determinata da una variabile turno.*
- ✓ **Variabile condivisa:** *boolean flag[2]=(false,false), int turno=1*

Processo P0

```
do{
  flag[0]=true;
  while(flag[1])do{
    if turno==1{
      flag[0]:=false;
      while(turno==1)do;
      flag[0]:=true;
  }
  <sezione critica>
  turno=1;
  flag[0]:=false;
  <sezione non critica>
  }while(1);
```

Vantaggi

1. Si risolvono i vari problemi

Processo P1

```
do{
  flag[1]=true;
  while(flag[0])do{
    if turno==0{
      flag[1]:=false;
      while(turno==0)do;
      flag[1]:=true;
  }
  <sezione critica>
  turno=0;
  flag[1]:=false;
  <sezione non critica>
  }while(1);
```

Svantaggi

1. L'algoritmo si presenta un po' complicato.

Algoritmo di Peterson

- ✓ **Osservazione:** *Dekker presenta un'algoritmo piuttosto complesso Peterson fornisce una soluzione semplice ed elegante.*
- ✓ **Variabile condivisa:** *boolean flag[2]=(false,false), int turno=1*

Processo P0

```
do{
  flag[0]=true;
  turno=1;
  while(flag[1]and turno==1)do;
  <sezione critica>
  flag[0]:=false;
  <sezione non critica>
}while(1);
```

Processo P1

```
do{
  flag[1]=true;
  turno=0;
  while(flag[0]and turno==0)do;
  <sezione critica>
  flag[1]:=false;
  <sezione non critica>
}while(1);
```

Vantaggi

1. Concorrenza tra i processi

Svantaggi

Correttezza dell'algoritmo di Peterson

Per dimostrare la correttezza dell'algoritmo di Peterson è necessario verificare le seguenti proprietà:

1. Mutua esclusione
2. Progresso
3. Attesa limitata

Peterson:Mutua esclusione

- **Mutua Esclusione**: Pi entra nella propria sezione critica solo se pronto[j]==false o turno==i. Se entrambi i processi fossero contemporaneamente in esecuzione nelle rispettive sezioni critiche, si avrebbe pronto[i]==pronto[j]==true.
- L'istruzione while non può essere eseguita da Pi e Pj contemporaneamente perché turno può assumere valore i o j, ma non entrambi. Supponendo che Pj ha eseguito con successo l'istruzione while avremo che pronto[j]==true e turno=j, condizione che persiste fino a che Pj si trova nella propria sezione critica.

Peterson: Attesa limitata e Progresso

- **Attesa Limitata e Progresso:** Si può impedire a un processo Pi di entrare nella propria sezione critica solo se questo è bloccato nel ciclo while dalla condizione pronto[j]==true e turno==j.
- Se Pj non è pronto per entrare, allora pronto[j]==false, e Pi può entrare nella propria sezione critica.
- Se Pj è pronto per entrare, allora pronto[j]==true, se turno==i entra Pi, mentre se turno==j, entra Pj.
- Se entra Pj, all'uscita imposta pronto[j]==false e Pi può entrare.
- Pi entra nella sezione critica (progresso) al massimo dopo un ingresso da parte di Pj (attesa limitata).

Soluzione per più processi: L'algoritmo del fornaio

Si vogliono sincronizzare n processi avente ognuno una sezione critica

- ✓ Questo algoritmo si basa su uno schema di servizio usato "nelle panetterie".
- ✓ Al suo ingresso nel negozio,ogni cliente riceve un numero.
- ✓ Viene servito di volta in volta il cliente con il numero piu' basso.
- ✓ L' algoritmo non assicura che due clienti (processi) non ricevano lo stesso numero.
- ✓ Nel caso in cui Pi e Pj hanno lo stesso numero e i<j viene servito prima Pi.
- ✓ Lo schema di numerazione genera sequenze crescenti di numeri: ad es., 1,2,3,3,3,3,4,5…
- ✓ Notazione <= ordine lessicografico (numero, process id)
  - ✓ $(a,b) < (c,d)$ if $a < c$ or if $a = c$ and $b < d$
  - ✓ $\max(a_0, \ldots, a_{n-1})$ è un numero, k, tale che $k \geq a_i$ for $i = 0, \ldots, n-1$

Algoritmo del fornaio (II)

Variabili condivise

```
boolean scelta[n]={false}
int numero[n]={0}
```

Processo Pi

```
do {
    scelta [i] = true;
    numero [i] = max(numero[0], numero[1], …,numero [n – 1])+1;
    scelta [i] = false;
    for (j = 0; j < n; j++) {
      while (scelta[j]) ;
      while ((numero[j] != 0) &&(numero[j,j] < numero[i,i]));
    }
    <sezione critica>
    numero[i] = 0;
    <sezione non critica>
} while (1);
```

Algoritmo del fornaio: correttezza

- **Mutua esclusione**: Se Pi si trova nella propria sezione critica e Pk(k!=i) ha già scelto il proprio numero[k]!=0, allora abbiamo che (numero[i],i) < (numero[k],k).

  Se Pi è nella propria sezione critica e Pk tenta di entrare nella propria, il processo Pk esegue la seconda istruzione while per j==i, trova che:
  - numero[i]!=0
  - (numero[i],i)<(numero[k],k) (per come viene assegnato in numero)
  - Quindi continua il ciclo nell'istruzione while fino a che Pi lascia la propria sezione critica

- **Progresso e Attesa Limitata:** Questi requisiti sono garantiti poiché i processi entrano nelle rispettive sezioni critiche secondo il criterio FCFS; ossia entra nella sezione critica chi arriva prima.

Architetture di sincronizzazione:soluzioni hardware al problema della sezione critica

- ✓ Certe caratteristiche dell'architettura di macchina possono rendere più semplice la programmazione e migliorare l'efficienza del sistema.
- ✓ E'possibile utilizzare delle semplici istruzioni messe a disposizione delle unità di elaborazione per risolvere in modo efficace il problema della sezione critica..
- ✓ Due fra le istruzioni più importanti consentono di controllare il contenuto di una parola di memoria e di scambiare il contenuto di due parole in modo ***atomico***,ossia come un'unità non interrompibile.

L'istruzione TestAndSet

- Modifica il contenuto di una parola di memoria non soggetta ad interruzioni.
- Se si eseguono contemporaneamente due istruzioni TestAndSet, ciascuna in una unità di elaborazione diversa, queste vengono eseguite in maniera sequenziale in un ordine arbitrario.

Definizione

```
boolean TestAndSet(boolean &obiettivo){
   boolean valore=obiettivo;
   obiettivo=true;
   return valore;
}
```

Mutua esclusione con Test-and-Set

Avendo a disposizione l'istruzione TestAndSet si può realizzare la mutua esclusione utilizzando una variabile globale ***blocco*** inizializzata a false.

Processo Pi

```
do {
    while (TestAndSet(blocco));
       <sezione critica>
       blocco = false;
       <sezione non critica>
} while (1);
```

Osservazione: non soddisfa l'attesa limitata!

L'istruzione Swap

Scambia il contenuto di due parole di memoria in modo inscindibile.

Definizione

```
void Swap(boolean &a, boolean &b) {
  boolean temp = a;
  a = b;
  b = temp;
}
```

Mutua esclusione con Swap

Avendo a disposizione l'istruzione Swap, la mutua esclusione può essere realizzata dichiarando e inizializzando a false una variabile booleana globale ***blocco*** e facendo uso una variabile booleana locale ***chiave*** per ogni processo.

Processo Pi

```
do {
   chiave = true;
   while (chiave == true)
      Swap(blocco, chiave);
   <sezione critica>
   blocco = false;
   <sezione non critica>
}while(1);
```

Osservazione: non soddisfa l'attesa limitata!

Mutua esclusione con attesa limitata con Test-and-Set

✓ Variabili condivise:

```
boolean blocco=false;
boolean attesa[n]={false};
```

```
do{
   attesa[i] = true;
   chiave = true;
   while(attesa[i] && chiave)
     chiave = TestAndSet(blocco);
   attesa[i] = false;
<sezione critica>
j = (i + 1) % n;
while ((j!=i)&&!attesa[j])
  j = (j + 1) % n;
if (j == i) blocco=false;
else attesa[ j ] = false;
<sezione non critica>
}while(1);
```

Correttezza dell'algoritmo

- **Mutua Esclusione:** Pi può entrare nella propria sezione critica solo se ***attesa[i]***==false oppure chiave==false. ***Chiave*** è impostata a false solo se si esegue TestAndSet e quindi ***blocco***=false. Il primo processo che esegue TestAndSet trova ***chiave***==false perchè ***blocco*** è false e tutti gli altri devono attendere. La variabile ***attesa[i]*** può diventare false solo se un altro processo esce dalla propria sezione critica (solo una variabile ***attesa[i]*** può valere false).
- **Progresso**: Un processo che esce dalla sezione critica o imposta ***blocco*** al valore false oppure ***attesa[j]*** al valore false, consentendo a un processo che attende di entrare.
- **Attesa Limitata:** Un processo, quando lascia la propria sezione critica, scandisce il vettore ***attesa*** nell'ordinamento ciclico (i+1,i+2,…..n-1,0, ….,i-1) e designa il primo processo in questo ordinamento presente nella sezione d'ingresso (***attesa[j]***==true) come il primo processo che deve entrare nella propria sezione critica. Ogni processo che cerca di entrare può farlo entro n-1 turni.

Vantaggi vs Svantaggi

VANTAGGI

- Si possono applicare ad un numero qualunque di processi.
- E'un meccanismo semplice e facile da verificare.
- Possono essere utilizzate per fornire più di una sezione critica in cui ciascuna avrà una propria variabile.

SVANTAGGI

- Si utilizza la tecnica dell'attesa attiva e quindi vengono sottratti cicli di CPU.
- E' possibile il verificarsi di uno stallo; se consideriamo il caso di singolo processore in cui un processo P1 esegue un'istruzione speciale ed accede alla sezione critica dopo di che viene interrotto per concedere il processore al processo P2 che ha priorità più alta. Se P2 tenta di accedere alla stessa risorsa di P1 riceverà un rifiuto a causa della mutua esclusione entrando in attesa attiva mentre P1 non verrà mai riattivato perché possiede una priorità più bassa.

Semafori

- Sincronizzazione mediante segnali scambiati tra i processi (Dijkstra).
- Semaforo S – variabile intera
- Ad un semaforo si può accedere solo tramite due operazioni che bisogna eseguire in modo atomiche:

Funzione wait

```
wait (S):{
  while (S <= 0) ; // non-op;
  S--;
}
```

Funzione signal

```
signal (S):{
  S++;
}
```

Mutua esclusione con semafori ad attesa attiva (Spinlock)

Variabili condivise:

```
semaforo mutex; // inizialmente mutex = 1
```

Process Pi:

```
do {
    wait(mutex);
        <sezione critica>
           signal(mutex);
        <sezione non critica>
} while (1);
```

Un processo attende fino a quando il semaforo non diventa positivo. Logicamente i semafori ad attesa attiva sono utili nei sistemi multiprocessore in cui non si necessita di un cambio di contesto.

Sincronizzazione

Eseguire B in P_j solo dopo l'esecuzione di A in P_i

- Useremo il semaforo **pronto** initializzato a 0

Evoluzione concorrente

Pi	Pj
⋮	⋮
A	wait(**pronto**)
signal(**pronto**)	B

Poiché pronto è inizializzato a 0, P_j esegue B solo dopo che P_i haeseguito signal(pronto) che si trova dopo A

Realizzazione di un semaforo con coda d'attesa

Si può definire un semaforo come una struttura:

```
typedef struct {
    int valore;
    struct processo *L;
} semaforo;
```

Assumiamo l'esistenza di due funzioni al fine di evitare l'attesa attiva:

- **block** sospende il processo che la invoca
- **wakeup(P)** fa riprendere l'esecuzione di un processo **P** precedentemente bloccato

Realizzazione di un semaforo (II)

- Le operazioni sui semafori sono ora definite da:

```
void wait (semaforo S) {
       S.valore --;
       if (S.value < 0) {
         aggiungi questo processo a S.L;
         block;
       }
}
```

Definizione wait

Definizione signal

```
void signal (semaforo S) {
       S.valore++;
       if (S.valore <= 0) {
         togli un processo P da S.L;
         wakeup(P);
       }
}
```

Nota: nei semafori ad attesa attiva i semafori non potevano assumere valori negativi mentre in questo caso se s.valore è negativo la dimensione del semaforo è data dal numero dei processi che attendono a quel semaforo

Stallo e attesa indefinita (deadlock and starvation)

- **Stallo (deadlock)** – un insieme di processi attendono indefinitivamente un evento (ad es. un *signal*) che può essere causato solo da uno dei processi dello stesso insieme.

Evoluzione con S e Q: due semafori inizializzati a 1

P0	P1
wait(S);	wait(Q);
wait(Q);	wait(S);
⋮	⋮
signal(S);	signal(Q);
signal(Q)	signal(S);

- **Attesa indefinita (starvation)** –attesa indefinita nella coda di un semaforo, ad esempio si può presentare se la coda del semaforo è gestita tramite criterio LIFO.

Due tipi di semafori

Semaforo *Contatore* – a valore intero, che può variare in un dominio non limitato.

Semaforo *Binario* – a valore intero, che può variare solo tra 0 e 1. Più semplice da implementare.

È possibile implementare un semaforo contatore S tramite semafori binari. Sono necessari a tal fine le seguenti strutture dati:

```
semaforo-binario S1, S2;
int C:
```

Inizializzazione:

S1 = 1
S2 = 0
C = valore iniziale del semaforo contatore **S**

Implementare un semaforo contatore S tramite semafori binari

wait

```
wait(S1);
C--;
  if (C < 0) {
    signal(S1);
    wait(S2);
  }
else
  signal(S1);
```

signal

```
wait(S1);
C++;
if (C > 0)
  signal(S2);
else
  signal(S1);
```

1. ***S1 garantisce l'accesso esclusivo alla variabile C***
2. ***S2 l'accesso mutuamente esclusivo alla sezione critica da parte dei processi.***

Problemi tipici di sincronizzazione

- Problema dei produttori e consumatori con memoria limitata
- Problema del barbiere
- Problema dei lettori e degli scrittori
- Problema dei cinque filosofi

Produttori e consumatori con memoria limitata

✓ Processo Produttore e Processo Consumatore

✓ Il processo consumatore necessita di informazioni prodotte dal processo produttore e condividono un buffer di elementi

Regole

✓ Il produttore non può produrre se il vettore è pieno

✓ Il consumatore non può consumare se il vettore è vuoto

Variabili condivise:

```
semaforo piene, vuote,
mutex;
  vettore di n elementi
```

vuote e piene conteggiano rispettivamente il numero di posizioni vuote e piene nel vettore

Inizializzazione:

```
piene = 0, vuote = n, mutex = 1
```

Produttori e consumatori con memoria limitata: processo produttore

Produttore

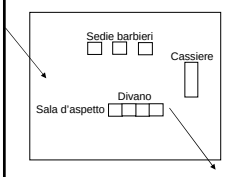
```
do{
   …
   produce un elemento in appena_prodotto
   …
   wait(vuote);
   wait(mutex);
   …
   inserisci in vettore l'elemento in appena_prodotto
   …
   signal(mutex);
   signal(piene);
}while (1);
```

Produttori e consumatori con memoria limitata: processo consumatore

Consumatore

```
do{
   wait(piene);
   wait(mutex);
   …
   rimuovi un elemento da vettore e mettilo in
   da_consumare
   …
   signal(mutex);
   signal(vuote);
   …
   consuma l'elemento contenuto in da_consumare
   …
}while (1);
```

Il barbiere



Problema

✓ Nel negozio abbiamo tre barbieri e tre sedie

✓ Una sala d'aspetto con un divano per quattro persone e spazio per altre persone in piedi

✓ Le norme antincendio limitano il massimo numero di persone presenti nel negozio pari a 20

✓ Supponiamo che il negozio debba soddisfare 50 clienti nell'arco di una giornata

Regole

1. ***Quando il negozio è pieno nessun nuovo cliente può entrare***
2. ***Una volta entrato, il cliente o si siede sul divano o rimane in piedi***
3. ***Quando si libera un barbiere il cliente servito è quello seduto da più tempo sul divano***
4. ***Quando un posto sul divano si libera si siede il cliente che è rimasto più a lungo in piedi.***
5. ***Quando un barbiere ha finito il cliente deve pagare ma esiste solamente un registratore di cassa e quindi un solo cliente alla volta può pagare.***
6. ***I barbieri passano il tempo tagliando i capelli, alla cassa o dormendo sulla sedia aspettando che arrivi qualche cliente***

Problema del barbiere (II)

- ***Capacita del negozio e del divano:*** sono governate da i semafori ***capacità_max*** e ***divano;*** ogni volta che un cliente tenta di entrare nel negozio il primo viene decrementato di un unità, inversamente se un cliente esce. Se il negozio è pieno il processo cliente è sospeso su ***capacità_max*** dalla funzione *wait*. Stesso ragionamento per il semaforo divano.
- ***Capacita della sedia:*** le tre sedie devono essere usate correttamente. Il semaforo ***sedia_barbiere*** fa si che non più di tre clienti alla volta chiedano di essere serviti.
- ***Assicurarsi che ci siano clienti sulle sedie:*** il semaforo ***cliente_pronto*** serve per svegliare un barbiere che dorme; senza tale semaforo un barbiere non dormirebbe mai (taglierebbe l'aria senza clienti)!
- ***Tenere clienti sulle sedie:*** una volta seduto il cliente rimane sulla sedia finchè il barbiere non segnala che il taglio è finito mediante il semaforo ***finito.***
- ***Assicurarsi che ci sia un solo cliente sulla sedia:*** il semaforo ***lasci_sedia_b*** serve ad impedire che il barbiere inviti un nuovo cliente prima che il precedente si sia alzato.

Problema del barbiere (III)

- ***Pagare ed incassare:*** il cliente deve pagare prima di lasciare il negozio con relativa ricevuta. Ciò è realizzato mediante un pagamento faccia a faccia; ossia ogni cliente dopo essersi alzato dalla sedia paga ed avverte il cassiere attendendo la ricevuta mediante i due semafori ***pagamento*** e ***ricevuta.***

- ***Coordinare le funzioni di barbiere e cassiere:*** per risparmiare il negozio non ha assunto un cassiere ma affida tale compito ai barbieri quando non sono impegnati in un taglio. Il semaforo ***coord*** fa si che un barbiere esegui esattamente un solo compito alla volta.

Processo cliente

```
wait(capacità_max);
<entra nel negozio>
wait(divano);
<Siedi sul divano>
wait(sedia_barbiere);
<Alzati dal divano>
signal(divano);
<Siedi sulla sedia del barbiere>
signal(cliente_pronto);
wait(finito);
<Lascia la sedia del barbiere>
signal(lascia_sedia_b);
<Paga>
signal(pagamento);
wait(ricevuta);
<Esci dal negozio>
signal(capcità_max);
```

Problema del barbiere (IV)

Processo barbiere

```
wait(cliente_pronto);
wait(coord);
<taglia i capelli>
signal(coord);
signal(finito);
wait(lasia_sedia_b);
signal(sedia_barbiere);
```

Processo cassiere

```
wait(pagamento);
wait(coord);
<prendi il denaro>
signal(coord);
signal(ricevuta);
```

Problemi

- ✓ Esiste un problema di temporizzazione che conduce ad un trattamento ingiusto dei clienti
- ✓ Se un barbiere è molto veloce o un cliente ha pochi capelli la sedia viene lasciata dal primo cliente che si è seduto che può non aver terminato il taglio!
- ✓ Il cliente che invece ha terminato è costretto ad attendere che un altro segnali di aver lasciato la sedia!

Lettori e scrittori

Problema

C'è un'area di dati condivisi fra vari processi, ad esempio un file,un blocco di memoria o un banco di registri del processore. Ci sono dei processi che possono solo leggere (lettori) e processi che possono sia scrivere che leggere (scrittori).

Regole

- ✓Più lettori possono leggere il file contemporaneamente
- ✓Solo uno scrittore alla volta può scrivere nel file
- ✓Se uno scrittore sta scrivendo in un file, nessun lettore può leggerlo.

```
Variabili condivise:
   semaforo mutex, scrittura;
   int numlettori
Inizialmente
   mutex = 1, scrittura = 1,
   numlettori = 0
```

Problema dei lettori e degli scrittori: processo scrittore

Processo scrittore

```
wait(scrittura);
   …
esegui l'operazione di scrittura
   …
signal(scrittura);
```

Problema dei lettori e degli scrittori: processo lettore

Processo lettore

```
wait(mutex);
numlettori++;
if (numlettori == 1)
  wait(scrittura);
signal(mutex);
   …
esegui l'operazione di lettura
   …
wait(mutex);
numlettori--;
if (numlettori == 0)
  signal(scrittura);
signal(mutex):
```

I cinque filosofi

Problema

- ✓ Ci sono 5 filosofi intorno ad un tavolo rotondo con 5 sedie, una per ciascun filosofo.
- ✓ Al centro del tavolo c'è una zuppiera colma di riso.
- ✓ La tavola è apparecchiata con 5 bacchette

Regole

1. *I filosofi possono pensare o mangiare.*
2. *Quando un filosofo pensa non interagisce con i colleghi*
3. *Quando ha fame tenta di prendere le bacchette più vicine (a destra e a sinistra)*
4. *Un filosofo può prendere una bacchetta alla volta e non può prendere una bacchetta che è già nelle mani di un suo vicino.*
5. *Un filosofo mangia quando ha 2 bacchette e le rilascia nel momento in cui ha terminato.*

Struttura del filosofo *i*

Variabile condivise: *le 5 bacchette;*
Rappresentazione: *semaforo bacchetta[5]={1};*

Processo Filosofo i

```
do {
    wait(bacchetta[i]);
    wait(bacchetta[(i+1)%5]);
    …
    mangia
    …
    signal(bacchetta[i]);
    signal(bacchetta[(i+1)%5]);
    …
    pensa
    …
} while(1);
```

✓ Non può accadere che due vicini mangino contemporaneamente.

✓ E' possibile lo stallo nel momento in cui i 5 filosofi prendano contemporaneamente la bacchetta a sinistra.

Regioni critiche (condizionali)

- Costrutto di sincronizzazione ad alto livello.
- Una variabile condivisa **v** di tipo **T**, viene dichiarata come:

  v: shared T;
- Alla variabile **v** si può accedere solo dall'interno di un'istruzione **region** della forma:

  region v when (B) do S

  dove **B** è un'espressione booleana.
- Mentre lo statement **S** è in esecuzione, nessun altro processo può accedere alla variabile **v**.
- Quando un processo vuole accedere alla variabile condivisa nella regione critica, l'espressione booleana *B* viene valutata. Se *B* è vera, lo statement *S* viene eseguito. Se è falsa, il processo viene sospeso finché *B* diventa vero e nessun altro processo si trova nella regione associata a *v*.

Esempio – Produttore/Consumatore

```
struct vettore {
        int gruppo[n];
 int contatore,inserisci,preleva;
}
```

Processo produttore

```
region vettore when(contatore<n){
 gruppo[inserisci]=appena_prodotto;
 inserisci:=(inserisci+1)%n;
 contatore++;
}
```

Processo consumatore

```
region vettore when (contatore>0){
da_consumare = gruppo[preleva];
             preleva = (preleva+1)%n;
             contatore--;
}
```

Monitor

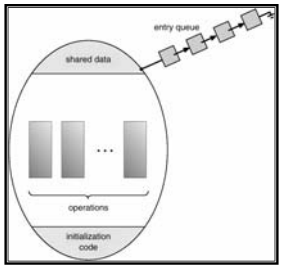
- Un altra primitiva di sincronizzazione ad alto livello è il monitor.
- Più facile da controllare rispetto ai semafori.

È definito dal programmatore tramite un insieme di operatori. La rappresentazione di un monitor non può essere usata direttamente dai diversi processi e quindi sono necessarie delle procedure.

Sintassi di un monitor

```
monitor nome-monitor{
  dichiarazione di varabili condivise
  procedure body P1 (…) {
        . . .
  }
  procedure body P2 (…) {
        . . .
  }
  procedure body Pn (…) {
        . . .
  }
  {
    codice di inizializzazione
  }
}
```

Schema di un monitor



All'interno di un monitor è attivo solo un processo per volta e quindi è garantita la mutua esclusione.

Questo costrutto non è abbastanza potente per realizzare la sincronizzazione e per questo ad esso vengono aggiunte le variabili condition

Monitor (II)

- I monitor forniscono una maniera semplice per ottenere la mutua esclusione.
- Abbiamo anche bisogno di un modo di bloccare i processi quando non possono proseguire (ad. es. produttore-consumatore).
- Un programmatore che deve scrivere un proprio schema di sincronizzazione può definire una o più variabili condizionali tramite il costrutto **condition**:

condition x, y;

Le uniche operazioni eseguibili su una variabile condition sono **wait** and **signal**.

L'operazione **x.wait()** implica che il processo che la invoca rimanga sospeso fino a che un altro processo non invoca l'operazione **x.signal()**

L'operazione **x.signal** risveglia esattamente un processo sospeso.

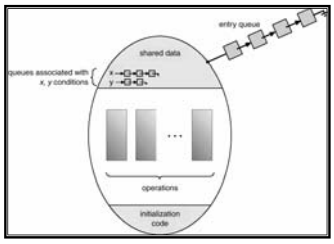
Se non ci sono processi sospesi **signal** non ha alcun effetto.

La signal viene utilizzata anche per avvisare che una variabile condition sta cambiando, l'avviso è rivolto a tutti i processi legati ad essa.

Schema di un monitor con variabili condition

Supponiamo esista un processo P che invoca l'operazione x.signal e che esista un altro processo Q in attesa associato alla variabile condition x. Chiaramente se a Q viene permessa la ripresa P dovrà attendere e allora si presentano due alternative per permettere l'esecuzione di entrambi.

1. P attende che Q lasci il monitor o attenda ad un'altra variabile cndition
2. Q attende che P lasci il monitor o attenda ad un'altra variabile condition

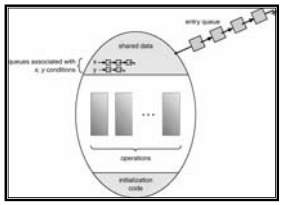


Definizione di monitor di Hoare

La definizione di monitor di Hoare prevede che, se c'è almeno un processo in coda rispetto ad una variabile condition, un processo che era in coda venga attivato immediatamente, quando un altro processo effettua la signal per quella condition. Pertanto il processo che ha effettuato la signal deve abbandonare immediatamente il monitor o restare sospeso.

Svantaggi di Hoare:

- Se il processo effettua la signal prima di dover terminare le sue operazioni allora sanno necessari due cambi di contesto:sospensione e ripresa
- Quando viene effettuata la signal deve entrare nel monitor solo il processo associato a quella variabile condition e ci deve entrare immediatamente altrimenti potrebbe non valere più la condizione. Lo scheduler deve essere completamente affidabile.



Una soluzione con monitor al problema dei cinque filosofi

I 5 filosofi

```
monitor fc {
enum {pensa, affamato, mangia} stato [5];
condition auto [5];
void prende(int i)
void posa(int i)
void verifica(int i)
void inizializzazione(){
  for (int i = 0; i < 5; i++)
    stato[i] = pensa;
  }
}
```

- Stati del filosofo
- Variabili condition
- Operatori definiti dal programmatore
- Codice d'inizializzazione

Ricordiamo che ciascun filosofo può mangiare solo se sono disponibili entrambi le bacchette

Una soluzione con monitor al problema dei cinque filosofi (II)

Funzione

```
void prende(int i){
  stato[i] = affamato;
  verifica(i);
  if (stato [i] != mangia)
    auto[i].wait();
}
```

Ciascun filosofo prima di cominciare a mangiare deve invocare l'operazione prende e ciò può determinare la sospensione del processo filosofo.

Una soluzione con monitor al problema dei cinque filosofi (IV)

Funzione

```
void verifica(int i) {
  if((stato[(i+4)%5] != mangia)&&
     (stato[i] == affamato)&&
     (stato[(i + 1) % 5] != mangia)){
    stato[i] = mangia;
    auto[i].signal();
  }
}
```

Funzione che controlla i vicini a destra e a sinistra se essi non stanno mangiando permette al filosofo che l' ha invocata di mangiare ed esegue una signal anche se il filosofo non era in wait.

Una soluzione con monitor al problema dei cinque filosofi (III)

Funzione

```
void posa(int i){
  stato[i] = pensa;
  // verifica left and right neighbors
  verifica((i+4) % 5);
  verifica((i+1) % 5);
}
```

Completata con successo l'operazione "prende" il filosofo può mangiare e alla fine invocherà l'operazione "posa" e incomincia a pensare..

Una soluzione con monitor al problema dei cinque filosofi (IV)

Sequenza filosofo i

```
Fc.prende(i);
…<mangia>
Fc.posa(i);
```

Cosa succede se tale sequenza non viene rispettata dal programmatore ?

Scambio di messaggi

I processi che interagiscono tra loro necessitano di

- **Sincronizzarsi per garantire la mutua esclusione**
- **Scambiarsi informazione per poter cooperare**

→ ***SCAMBIO DI MESSAGGI***

Primitive

```
send(destinazione,messaggio);
receive(sorgente,messaggio);
```

Scambio di messaggi: sincronizzazione

- ***Send e receive bloccante:*** sia il mittente che il ricevente sono bloccati fino al completamento dell'operazione (rendez-vous). Consente una stretta sincronizzazione di processi.
- ***Send non bloccante e receive bloccante:*** anche se il mittente può continuare il ricevente deve attendere l'arrivo del messaggio (un processo può mandare vari messaggi a più destinatari molto velocemente).
- ***Send non bloccante e receive non bloccante:*** nessuno dei due deve attendere.

Scambio di messaggi: indirizzamento

- ***Indirizzamento diretto:*** la primitiva send possiede un identificatore specifico del processo destinatario.
- ***Indirizzamento indiretto:*** i messaggi non viaggiano direttamente dal mittente al destinatario ma sono mandati ad un dispositivo condiviso avente code che contengono temporaneamente i messaggi (caselle di posta o porta).
  1. ***Associazione statica:*** *porta creata ed assegnata permanentemente al processo (uno a uno).*
  2. ***Associazione dinamica:*** *utilizzo di primitive* `connect` *e* `disconnect` *nel caso di molti mittenti.*

Scambio di messaggi: i messaggi

1. **Messaggi di lunghezza fissa:** *si minimizza lo spazio ed il sovraccarico totale.*
2. **Messaggi di lunghezza variabile:** *gestione flessibile ma più onerosa per il sistema operativo.*

Scambio di messaggi: mutua esclusione

Casella di posta condivisa: mutex = {messaggio vuoto}

Mutua esclusione

```
do{
   receive(mutex,msg);
   <seziona critica>
   send(mutex,msg);
   <resto del programma>
}while(1;)
```

Osservazioni

1. Nel caso in cui più processi eseguono la receive e c'è un messaggio esso è consegnato solo ad uno dei processi in attesa.
2. Se la coda è vuota, tutti i processi sono bloccati, e quando arriva un messaggio solo un processo lo riceve mentre gli altri rimangono bloccati.

Produttori/Consumatori con scambio di messaggi

Processo produttore

```
do{
   receive(cas_produci,pmsg);
   pmsg=produci;
   send(cas_consuma,pmsg);
   <resto del programma>
}while(1);
```

La casella cas_produci contiene un numero di messaggi vuoti pari alla capacità totale del buffer

La casella cas_consuma rappresenta il buffer vero e proprio organizzato come una coda di messaggi

Processo consumatore

```
do{
   receive(cas_consuma,cmsg);
   consuma(csmg);
   send(cas_produci,null);
   <resto del programma>
}while(1);
```