POLITECNICO DI MILANO





Sincronizzazione

Laboratorio Software 2008-2009
C. Brandolese

Problemi della multiprogrammazione

Più processi in esecuzione in un sistema

- Interferenza tra i processi
- Errori, diminuzione delle prestazioni, deadlock...

Alcuni processi cooperano per una funzione comune

- Una tale separazione può portare un beneficio in termini di effcienza
- Le varie attività devono essere coordinate

L'accesso alle risorse condivise deve essere gestito

- Più processi possono utilizzare risosrse comuni simultaneamente
- Il sistema operativo deve allocare e gestire tali risorse

Gli errori di programmazione sono più difficili da individuare

Problemi di concorrenza a livello di SO

Tenere traccia dei processi attivi

Allocare e deallocare le risorse

- Processore
- Memoria
- File
- Dispositivi di I/O

Proteggere dati e risorse

Garantire che il risultato di un processo sia indipendente dalla velocità di esecuzione

- Altri processi possono condividere il processore e quindi influire sul tempo di esecuzione

Interazione tra processi

Tre possibili situazioni

Processi indipendenti

- Non possono influenzarsi a vicenda
- Possibile competizione per le risorse

Processi indirettamente dipendenti

- Condividono l'accesso a qualche tipo di oggetto
- Possibile cooperazione tra i processi

Processi dipendenti

- I processi cooperano mediante comunicazione
- L'interazione tra processi ha natura

Competizione

Competizione tra processi per le risorse

Se due processi vogliono accedere alla stessa risorsa

- Uno acquisisce la risorsa
- L'altro deve aspettare

Problemi

- Mutua esclusione
  - Sezioni critiche
- Deadock
- Starvation

Cooperazione indiretta

Cooperazione tra processi mediante condivisione

I processi usano e modificano dati comuni

- Variabili condivise
- File
- Database

La scrittura deve essere mutuamente esclusiva

- Mediante il concetto di sezione critica si garantisce l'integrità dei dati

Problemi

- Mutua esclusione
- Deadlock
- Starvation
- Consistenza dei dati

Cooperazione diretta

Cooperazione tra processi mediante comunicazione

La comunicazione fornisce un modo per

- Sincronizzare le attività
- Comunicare dati

Problemi

- Possibile deadlock
  - Ogni processo in attesa di un messaggio dall'altro
- Possibile starvation
  - Due processi mandano un messagio l'uno all'altro mentre un terzo processo è in attesa di un messaggio

Mutua eclusione: requisiti

Un solo processo alla volta

- Può essere nella sezione critica relativa ad un data risorsa

Un processo che termina nella sua sezione critica

- Non deve interferire con gli atri

Un processo non deve subire ritardi all'ingresso della sua sezione critica

- Quando nessun altro processo la sta utilizzando

Non si fanno assunzioni su

- Velocità dei processi
- Numero dei processi

Un processo rimane nella sua sezione critica

- Solo per un termpo finito

Sezione critica

Più processi

- Cooperano ad un compito comune
- Condividendo alcune risorse

Una sezione critica è una porzione di codice in cui

- Un processo utilizza qualche risorsa condivisa
- Nessun altro processo può interferire

L'esecuzione delle sezioni critiche deve essere mutuamente esclusiva nel tempo

- È richiesto un protocollo che garantisca che un solo processo alla volta si trovi nella sezione critica

Diverse sezioni critiche per diversi insiemi di risorse

Sezione critica

Ogni soluzione al problema deve soddisfare i seguenti requisiti

- Mutua esclusione
  - Se un processo è in esecuzione nella sua sezione critica, allora nessun altro processo può eseguire il codice della sua sezione critica
- Progresso
  - Se nessun processo sta eseguendo il codice della sua sezione critica ed esiste qualche processo che è in attesa di entrare nella sua sezione critica, la scelta del prossimo processo da eseguire non può essere ritardata indefinitamente
- Attesa limitata
  - Dopo che un processo ha effettuato la richiesta di entrare nella sua sezione critica, solo un numero finito di processi può essere eseguito prima del processo in esame

Assunzioni

- Tutti i processi sono in esecuzione ad una velocità non nulla
- Nessuna restrizione sulle velocità relative

Busy waiting

Algoritmo 1

- I Processi P0 e P1 condividono una variabile comune turn
- La variabile turn è inizializzata a 0 oppure 1
- Se `turn = 0` il processo P0 entra nella sua critical section
- Se `turn = 1` il processo P1 entra nella sua critical section

```
/* Code for PROCESS Pi */

while( 1 )
{
    while( turn != i )
      ; /* Do nothing */
    /* Critical section */
    turn = j;
    /* Reminder section */
}
```

Busy waiting

Valutazione dell'Algoritmo 1

- Mutua esclusione
  - Si, un solo processo alla volta può eseguire la propria sezione critica
- Progresso
  - No, è richiesta un'alternanza rigorosa
- Attesa limitata
  - Si, al più un periodo di attesa è richiesto

Busy waiting

Algoritmo 2

- Si usa un array **`flag[]`** di due elementi
- Se **`flag[i]=true`** allora il processo Pi è pronto ad entrare nella sua sezione critica

```
/* Code for PROCESS Pi */

while( 1 )
{
     while( flag[j] == true )
       ; /* Do nothing */
     flag[i] = true;
     /* Critical section */
     flag[i] = false;
     /* Reminder section */
}
```

Busy waiting

Valutazione dell'Algoritmo 2

- Mutua esclusione
  - No, ogni processo può verificare I flag ed entrare simulataneamente nella propria sezione critica
- Progresso
  - Si, se nessun altro processo è nella seizone critica un processo modifica il flag in modo da indicare che è pronto per entrare nella procpria critical section
- Attesa limitata
  - Si, ogni processo assegna il flag all'uscita dalla sezioen critica in modo che l'altro possa continuare

Busy waiting

Algoritmo 3

- Si usa un array **`flag[]`**
- Se **`flag[i]=true`** allora il processo Pi è pronto ad entrare nella sua sezione critica
- Gli elemti di **`flag[]`** sono inzializzati a **`false`**

```
/* Code for PROCESS Pi */

while( 1 )
{
     flag[i] = true;
     while( flag[j] == true )
       ; /* Do nothing */
     /* Critical section */
     flag[i] = false;
     /* Reminder section */
}
```

Busy waiting

Problema

- Cosa accade se P0 mette `flag[0]` a `true` e P1 mette `flag[1]` a `true` prima che entrambi entrino nel loro ciclo while?
- Si entra in due loop infiniti
  - L'algoritmo dipende troppo dall'esatta temporizzazione dei processi
  - Molto raro, ma può accadere

Valutazione dell'Algoritmo 3

- Mutua esclusione
  - Si, un solo processo alla volta può eseguire la propria sezione critica
- Progresso
  - No, si possono avere loop infiniti
- Attesa limitata
  - Si, ogni processo assegna il flag all'uscita dalla sezioen critica in modo che l'altro possa continuare

Busy waiting

Algoritmo di Peterson

- Si usano un array `flag[]` ed una variabile `turn`

Soluzione corretta al problema della mutua esclusione

```
/* Code for PROCESS Pi */

while( 1 )
{
     flag[i] = true;
     turn = j;
     while( flag[j] == true & turn == j )
       ; /* Do nothing */
     /* Critical section */
     flag[i] = false;
     /* Reminder section */
}
```

Supporto hardware

Disabilitazione degli interrupt

- Un processo continua la propria esecuzione finché
  - Invoca una chiamata di sistema operativo
  - Riceve un interrupt
- Gli interrupt sono disabilitati durante l'esecuzione della critical section
- Il processore (sistema operativo) ha una libertà limitata nel gestire l'alternanza dei processi
  - Senza interrupt non si può avere l'alternanza dei processi
- L'efficienza può essere notevolmente degradata
- Potenzialmente rischioso
  - Un processo può bloccare l'intero sistema in casi quali
    - Loop infinito
    - Attesa per risorse

Supporto hardware

Istruzioni speciali

- Eseguono due operazioni in modo indivisibile o atomico
- Eseguite in un solo ciclo di istruzione
- Non soggette ad interferenza da parte di altre istruzioni

Test-and-set

- Esegue il test su un valore
  - Se è false viene posto a true
  - Se è true viene posto a false

Exchange

- Scambia il valore di due variabili

Busy waiting

Svantaggi

- Il busy-waiting consuma tempo di processore

Si può avere starvation

- Quando un processo esce dalla propria critical section e più processi sono in attesa, chi viene eseguito come procesos seguente?

Si può avere deadlock

- Per via di un fenomeno noto come priority inversion
  - Si hanno due processi H (High priority) ed L (Low priority)
  - Scheduling: H viene eseguito non appaena diviene ready
  - Se L è nella regione critica e H diviene ready, H inizia il bust waiting, ma dato che L non può essere eseguito mentre H è running, L non uscirà mai dalla regione critica e H entra in un ciclo infinito

Semafori

Utilizzati nel caso generale di sincronizzazione tra n processi

- Funzione fondamentale offerta da molti sistemi operativi

Un semaforo

- È una variabile intera
  - Accessibile solo mediante operazioni atomiche
- Operazioni
  - Wait
  - Signal

Funzionamento

- Se un processo è in attesa di un segnale, è sospeso in attesa che il segnale venga inviato
- Quando un processo si sospende su un semaforo, viene aggiunto alla coda dei processi in attesa

Semaforo intero – Implementazione

Un semaforo intero è un record

```
type semaphore is record
    counter:  integer;
    queue:    list of process;
end;
```

La funzione wait(s)

- Aggiunge il processo alla lista dei processi sospesi sul semaforo s
- Sospende il processo

```
wait(S)
begin
    S.counter := S.counter – 1;
    if S.counter < 0 then
    begin
        add this process to S.queue;
        suspend( this process );
    end;
end;
```

Semaforo intero – Implementazione

La funzione signal(s)

- Incrementa il contatore di un semaforo
- Rimuove un processo dalla lista dei processi sospesi sul semaforo s
- Risveglia il processo

```
signal(S)
begin
    S.counter := S.counter – 1;
    if S.counter <= 0 then
    begin
        remove a process P from S.queue;
        wakeup( P );
    end;
end;
```

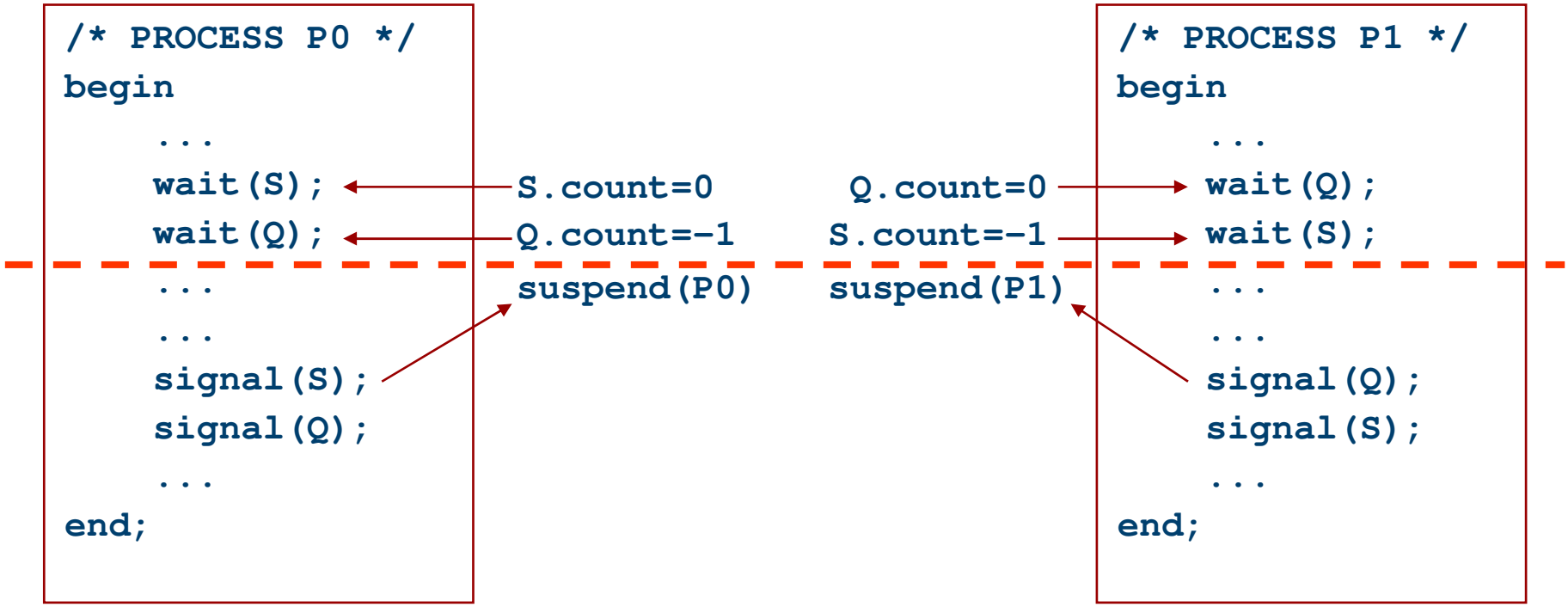
Le operazioni wait(s) e signal(s) devono essere atomiche!

Deadlock

Si verifica quando due o più processi sono in atesa di un evento che solo uno di essi può generare

Esempio:

- Due processi P0 e P1
- Due semafori S e Q inizializzati a 1



Semaforo binario – Implementazione

Un semaforo binario è un record

```
type semaphore is record
    value:    {0,1};
    queue:    list of process;
end;
```

La funzione wait(s)

- Aggiunge il processo alla lista dei processi sospesi sul semaforo s
- Sospende il processo

```
wait(S)
begin
    if S.value == 1 then
        S.value := 0;
    else begin
        add this process to S.queue;
        suspend( this process );
    end;
end;
```

Semaforo intero – Implementazione

La funzione signal(s)

- Rimuove un processo dalla lista dei processi sospesi sul semaforo s
- Risveglia il processo

```
signal(S)
begin
    if S.queue is empty then
        S.value := 1;
    else begin
        remove a process P from S.queue;
        wakeup( P );
    end;
end;
```

Le operazioni wait(s) e signal(s) devono essere atomiche!

Esempio: Produttore/Consumatore

Due processi

- Utilizzano un buffer condiviso
- Un solo processo alla volta può accedere al buffer

Produttore

- Genera dati e li scrive nel buffer, uno alla volta

Consumatore

- Legge i dati dal buffer, uno alla volta

Utilizziamo tre semafori

- Due rappresentano la quantità di dati nel buffer
  - empty, full
- Uno indica che il processo può accede al buffer
  - mutex

Esempio: Produttore/Consumatore

Dati condivisi e inizializzazione

- Tipo dato e buffer di quel tipo
- Semafori
- Variabili temporanee per contenere i dati

```
type item: ...

buff: buffer of item

full:  semaphore := 0;
empty: semaphore := n;
mutex: semaphore := 1;

pitem: item;
citem: item;
```

Esempio: Produttore/Consumatore

Due processi

```
/* PRODUCER        */

while( true )
begin
     ...
     produce item pitem;
     ...
     wait(empty);
     wait(mutex);
     ...
     add pitem to buff;
     ...
     signal(mutex);
     signal(full);
end;
```

```
/* CONSUMER        */

while( true )
begin
     wait(full);
     wait(mutex);
     ...
     read citem from buff;
     ...
     signal(mutex);
     signal(empty);
     ...
     consume item citem
     ...
end;
```

Monitor

Un monitor è simile ad un ADT e raccoglie

- Procedure
- Strutture dati
- Variabili

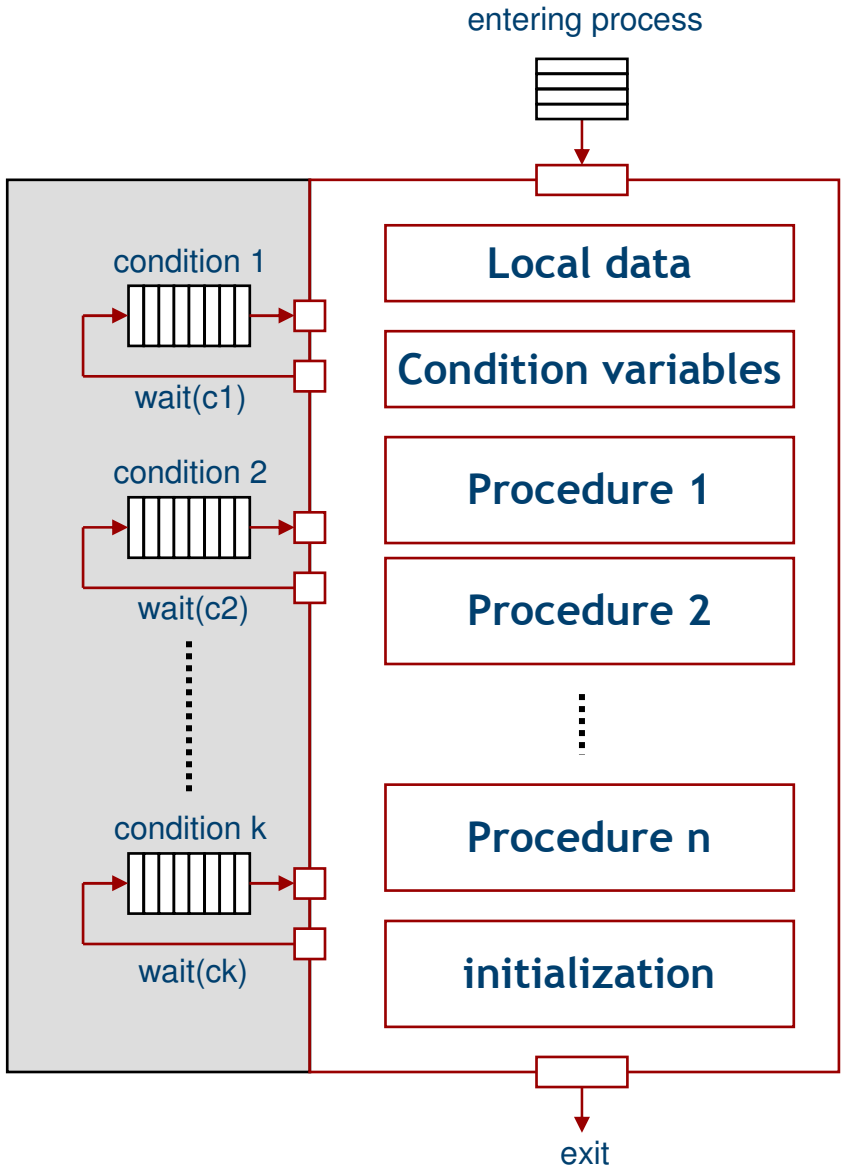
I processi

- Possono invocare le funzioni nel monitor
- Non possono accedere direttamente ai dati
- Vengono accodati mediante wait() e signal()

Mutua esclusione

- Un solo processo alla volta può essere attivo nel monitor
- Ogni procedura è una sezione critica
- Semplificazione della programmazione

Monitor




```
type newmonitor = monitor
begin
    variable declaration;

    procedure entry P1(...)
    begin ... end;

    procedure entry P2(...)
    begin ... end;
    ...
    procedure entry Pn(...)
    begin ... end;

    begin
        initialization code;
    end;
end;
```