UNIVERSITÀ DEGLI STUDI DI NAPOLI
PARTHENOPE

Sistemi Operativi

Sincronizzazione dei Processi

LEZIONE 7

prof. Antonino Staiano

Corso di Laurea in Informatica – Università di Napoli Parthenope

antonino.staiano@uniparthenope.it

Agenda

- Cosa è la sincronizzazione dei processi?
- Race condition
- Sezioni critiche
- Sincronizzazione di controllo e operazioni indivisibili
- Approcci alla sincronizzazione
- Problemi di sincronizzazione di processi classici
- Approccio algoritmico per implementare le sezioni critiche
- Semafori
- Monitor

Introduzione: processi concorrenti

- Consideriamo la sequenza di codice che un compilatore genera per l'aggiornamento di un contatore
- Supponiamo di voler incrementare di 1 il contatore (**counter**)
- La sequenza di codice da eseguire potrebbe essere (in assembly x86):

```
100 mov    0x8049a1c, %eax
105 add    $0x1, %eax
108 mov    %eax, 0x8049a1c
```

OS	Thread 1	Thread 2	PC (after instruction)	eax	counter
	before critical section		100	0	50
	`mov 8049a1c,%eax`		105	**50**	50
	`add $0x1,%eax`		108	**51**	50
interrupt					
save T1					
restore T2			100	0	50
		`mov 8049a1c,%eax`	105	**50**	50
		`add $0x1,%eax`	108	**51**	50
		`mov %eax,8049a1c`	113	51	**51**
interrupt					
save T2					
restore T1			108	51	51
	`mov %eax,8049a1c`		113	51	**51**

Introduzione: una visione di insieme

- Race condition o data race
  - Il risultato dipende dall'istante dall'stante di esecuzione del codice
    - Il risultato può essere quello sbagliato
- La computazione non è deterministica abbiamo cioè un esito indeterminato
  - non sappiamo dire quale sarà l'output, che può essere diverso in diverse esecuzioni dello stesso codice
- Il pezzo di codice che i thread concorrenti accedono causando una race condition è chiamato sezione critica
  - Codice che accede ad una variabile condivisa o più in generale una risorsa condivisa che non deve essere eseguito concorrentemente da più di un thread
    - mutua esclusione
      - Questa proprietà garantisce che mentre un thread è in esecuzione all'interno della sezione critica ai restanti thread concorrenti ciò non è permesso

Cosa è la Sincronizzazione dei Processi?

- Useremo il termine processo per indicare sia processi che thread
- Notazione
  - *read_set*$_i$ -> insieme di dati letti dal processo P_i e messaggi interprocesso o segnali ricevuti da P_i
  - *write_set*$_i$ -> insieme di dati modificati dal processo P_i e messaggi interprocesso o segnali inviati da P_i
- *Processi interagenti*: i processi P_i e P_j sono processi interagenti se la *write_set* di uno dei processi si sovrappone con *la write_set* o *read_set* dell'altro
- I processi che non interagiscono sono *processi indipendenti*
- La *sincronizzazione dei processi* indica le tecniche usate per ritardare e ripristinare i processi per implementare le interazione tra i processi

Convenzioni in Pseudocodice per i Programmi Concorrenti

- The control structure **Parbegin** <*list of statements*> **Parend** encloses code that is to be executed in parallel. (Parbegin stands for parallel-begin, and Parend for parallel-end.) If <*list of statements*> contains n statements, execution of the **Parbegin–Parend** control structure spawns n processes, each process consisting of the execution of one statement in <*list of statements*>. For example, **Parbegin** S_1, S_2, S_3, S_4 **Parend** initiates four processes that execute S_1, S_2, S_3 and S_4, respectively.

  The statement grouping facilities of a language such as **begin–end**, can be used if a process is to consist of a block of code instead of a single statement. For visual convenience, we depict concurrent processes created in a **Parbegin–Parend** control structure as follows:

Parbegin

$$\begin{array}{cccc} S_{11} & S_{21} & \cdots & S_{n1} \\ \vdots & \vdots & & \vdots \\ S_{1m} & S_{2m} & \cdots & S_{nm} \end{array}$$

Parend

Process P_1 | *Process* P_2 | ... | *Process* P_n

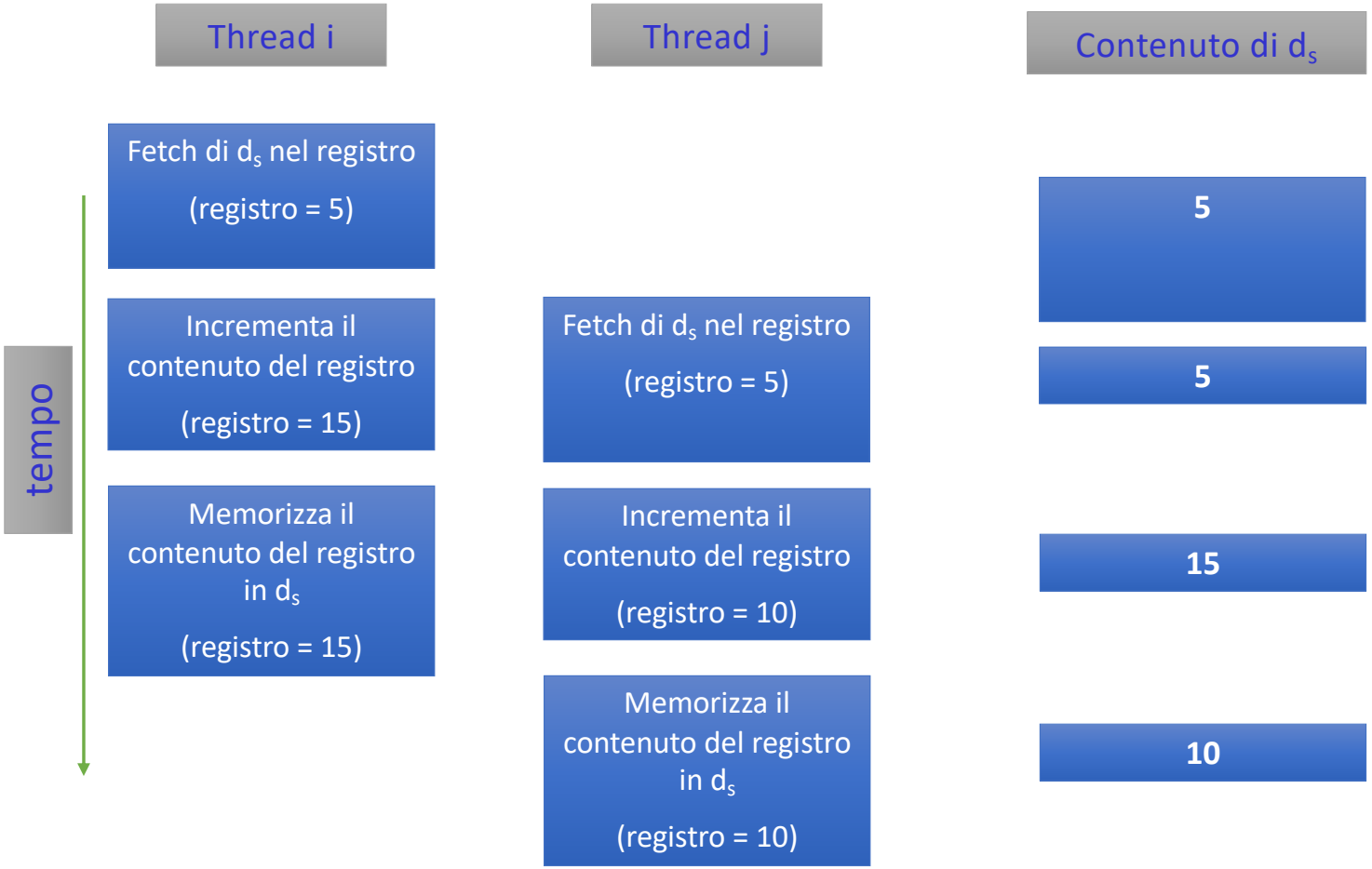
  where statements $S_{11} \cdots S_{1m}$ form the code of process P_1, etc.
- Declarations of shared variables are placed before a **Parbegin**.
- Declarations of local variables are placed at the start of a process.
- Comments are enclosed within braces "{ }".
- Indentation is used to show nesting of control structures.

Pseudocode conventions for concurrent programs.

Race Condition

- Gli accessi non coordinati ai dati condivisi possono compromettere la consistenza dei dati
- Consideriamo i processi P_i e P_j che aggiornano il valore di d_s con le operazioni a_i e a_j, rispettivamente:
  - Operazione a_i: $d_s := d_s + 10$; Sia $f_i(d_s)$ il risultato
  - Operazione a_j: $d_s := d_s + 5$; Sia $f_j(d_s)$ il risultato
  - Cosa può succedere se tali operazioni sono eseguite concorrentemente?
- *Race condition:* Una condizione in cui il valore di un oggetto condiviso d_s, risultante dall'esecuzione delle operazioni a_i e a_j su d_s nei processi interagenti, può essere diversa da ambo $f_i(f_j(d_s))$ e $f_j(f_i(d_s))$

Due thread (no sincro) che incrementano la stessa variabile



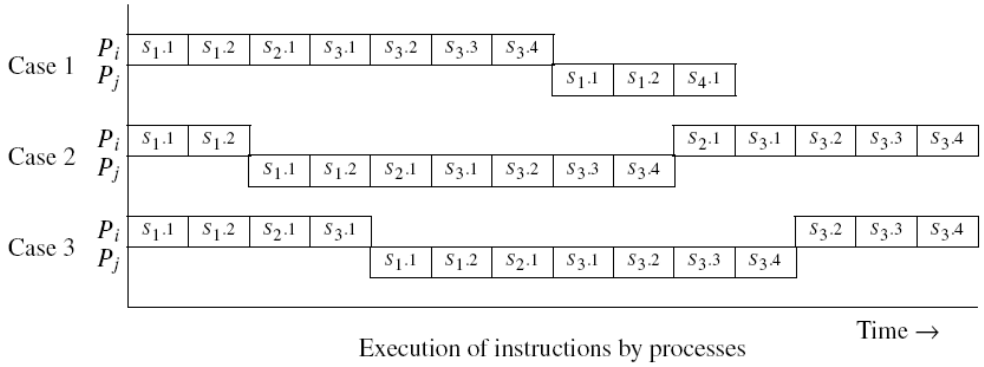
Esempio di *Race Condition*

nextseatno = 200
Capacity = 200

P_i e P_j eseguono lo stesso codice

	Code of processes		Corresponding machine instructions
S_1	**if** *nextseatno* ≤ *capacity*	$S_1.1$	Load *nextseatno* in reg_k
		$S_1.2$	If reg_k > *capacity* goto $S_4.1$
	then		
S_2	*allotedno:=nextseatno;*	$S_2.1$	Move *nextseatno* to *allotedno*
S_3	*nextseatno:=nextseatno+1;*	$S_3.1$	Load *nextseatno* in reg_j
		$S_3.2$	Add 1 to reg_j
		$S_3.3$	Store reg_j in *nextseatno*
		$S_3.4$	Go to $S_5.1$
	else		
S_4	*display "sorry, no seats available"*	$S_4.1$	Display "*sorry, . . .*"
S_5	. . .	$S_5.1$	. . .

Some execution cases



Execution of instructions by processes

I casi 2 e 3 portano a risultati errati

Condivisione dei dati di processi di un'applicazione di prenotazione

Race Condition (cont.)

- Le race condition sono prevenute garantendo che le operazioni a_i e a_j non siano eseguite concorrentemente
  - *mutua esclusione*
    - Solo un'operazione accede ai dati in ogni istante

- La sincronizzazione per l'accesso ai dati è il coordinamento dei processi per realizzare la mutua esclusione su dati condivisi

- Con la mutua esclusione è assicurato che il risultato delle operazioni a_i e a_j sarà $f_i(f_j(d_s))$ oppure $f_j(f_i(d_s))$

Race Condition (cont.)

- Definiamo
  - $update\_set_i$ -> insieme di dati aggiornati dal processo Pi (letti, modificati e scritti)
- Per prevenire una race condition:
  - Si controlla che la logica dei processi causa una race
    - $update\_set_i \cap update\_set_j \neq \emptyset$
      - Se qualche variabile è modificata sia da Pi che Pj
  - Es.: nel sistema di prenotazione aereo
    - $update\_set_i = update\_set_j = \{\text{nextseatno}\}$
  - Si individua il dato su cui c'è una race
    - Si usano tecniche di sincronizzazione che implementano la mutua esclusione per l'accesso ai dati

Sezioni Critiche

- Sezione critica (SC):
  - Un oggetto o una porzione di codice su cui c'è una race condition se acceduto concorrentemente da più processi/thread
- una sezione critica per un oggetto d_s deve essere protetta in modo che possa essere eseguita concorrentemente con se stessa o con altre sezioni critiche per d_s
  - La mutua esclusione è il modo con cui proteggere una sezione critica di codice
- Nei codici di esempio, una SC è indicata con un rettangolo grigio
- Se un processo P_i sta eseguendo una SC per d_s, un altro processo che intendesse eseguire una SC per d_s dovrebbe attendere la fine dell'esecuzione della SC di P_i
  - Una SC per un dato d_s, è una regione di mutua esclusione rispetto agli accessi a d_s

Sezioni Critiche (cont.)

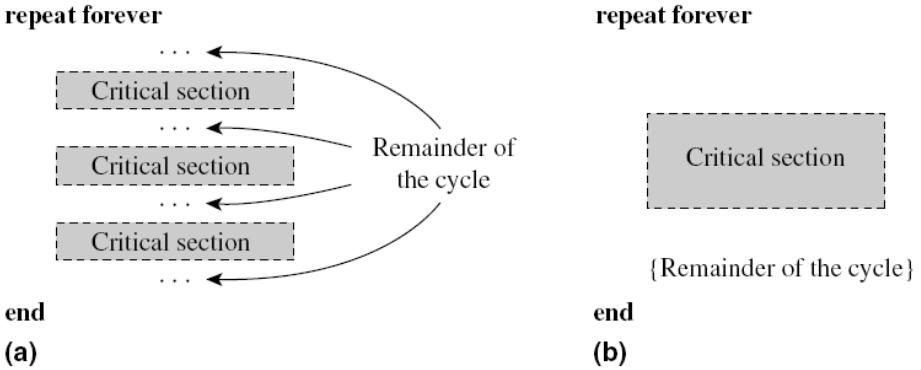


Figure 6.3 (a) A process with many critical sections; (b) a simpler way of depicting this process.

```
if nextseatno ≤ capacity
then
    allotedno:=nextseatno;
    nextseatno:=nextseatno+1;
else
    display "sorry, no seats
             available";
```

Process P_i

```
if nextseatno ≤ capacity
then
    allotedno:=nextseatno;
    nextseatno:=nextseatno+1;
else
    display "sorry, no seats
             available";
```

Process P_j

Uso di sezioni critiche in un sistema di prenotazione aereo

Sezioni Critiche (cont.)

- Usare SC causa ritardi nelle operazioni dei processi
  - Un processo non deve essere eseguito a lungo in una SC
  - Il processo non deve invocare chiamate di sistema che possono portarlo in uno stato bloccato, all'interno di una SC
  - Il kernel non dovrebbe prelazionare un processo che è alle prese con l'esecuzione di una SC
    - Il kernel dovrebbe essere sempre informato se un processo è in una SC
    - Non è possibile implementarlo nel caso un processo implementi una SC autonomamente (senza informare il kernel)
- Assumeremo che un processo trascorra poco tempo in una SC

Proprietà Implementazione di una Sezione Critica

- Proprietà essenziali di un'implementazione di una SC

Property	Description
Mutual exclusion	At any moment, at most one process may execute a CS for a data item d_s.
Progress	When no process is executing a CS for a data item d_s, one of the processes wishing to enter a CS for d_s will be granted entry.
Bounded wait	After a process P_i has indicated its desire to enter a CS for d_s, the number of times other processes can gain entry to a CS for d_s ahead of P_i is bounded by a finite integer.

- Il *progresso* e l'*attesa limitata* insieme prevengono la ***starvation***

Sincronizzazione di Controllo

- I processi interagenti devono coordinare la loro esecuzione uno rispetto all'altro, per eseguire le rispettive azioni nell'ordine desiderato
  - Requisito soddisfatto mediante la sincronizzazione di controllo

{Perform operation a_i only after P_j performs operation a_j}	Perform operation a_j
	. . .
Process P_i	*Process* P_j

Processi che richiedono la sincronizzazione di controllo

  - La segnalazione è una tecnica generale di sincronizzazione di controllo

Sincronizzazione di Controllo e Operazioni Indivisibili

```
var
        operation_aj_performed : boolean;
        pi_blocked : boolean;
begin
        operation_aj_performed := false;
        pi_blocked := false;

Parbegin
        . . .
        if operation_aj_performed = false
        then
            pi_blocked := true;
            block (Pi);
        {perform operation ai}
        . . .
        . . .
        . . .
Parend;
end.
```

Process P_i

```
. . .
{perform operation aj}
if pi_blocked = true
then
    pi_blocked := false;
    activate (Pi);
else
    operation_aj_performed := true
. . .
```

Process P_j

Un tentativo di segnalazione naïve mediante variabili booleane

Sincronizzazione di Controllo e Operazioni Indivisibili (cont.)

- Una segnalazione naïve non funziona
  - P_i potrebbe bloccarsi indefinitamente in alcune situazioni

Race condition nella sincronizzazione di Processi

Time	Actions of process P_i	Actions of process P_j
t_1	**if** *action_aj_performed = false*	
t_2		*{perform action a_j}*
t_3		**if** *pi_blocked = true*
t_4		*action_aj_performed :=true*
⋮		
t_{20}	*pi_blocked :=true;*	
t_{21}	*block (P_i);*	

- Usare invece operazioni atomiche o indivisibili

Sincronizzazione di Controllo e Operazioni Indivisibili (cont.)

Operazione indivisibile: un'operazione su un insieme di oggetti che non può essere interrotta durante la sua esecuzione su un oggetto incluso nell'insieme

```
procedure check_aj
begin
    if operation_aj_performed=false
     then
        pi_blocked:=true;
        block (Pi)
end;

procedure post_aj
begin
    if pi_blocked=true
     then
        pi_blocked:=false;
        activate(Pj)
    else
         operation_aj_performed:=true;
end;
```

Operazioni indivisibili *check_a$_j$* e *post_a$_j$* per la segnalazione

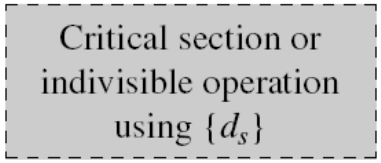
Approcci alla Sincronizzazione

- Ciclare vs bloccare
- Supporto HW per la sincronizzazione dei processi
- Approcci algoritmici, primitive di sincronizzazione e costrutti di programmazione concorrente

Ciclare vs Bloccare

- Busy wait (attesa attiva)

while (some process is in a critical section on $\{d_s\}$ or
is executing an indivisible operation using $\{d_s\}$)
{ do nothing }

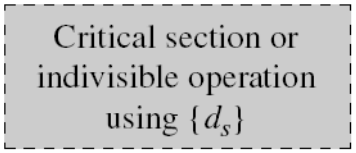


- Un'attesa attiva ha molte conseguenze negative
  - Non può fornire la proprietà di attesa limitata
  - Degrado delle prestazioni a causa del ciclare
  - Deadlock
  - Inversione di priorità
    - Tipicamente affrontato mediante il protocollo di ereditarietà della priorità

Ciclare vs Bloccare (cont.)

- Per evitare le attese attive, un processo in attesa di entrare in una SC è posto in uno stato *bloccato*
  - Cambiato in *pronto* solo quando può entrare nella SC

if (some process is in a critical section on $\{d_s\}$ or
is executing an indivisible operation using $\{d_s\}$)
then *make a system call to block itself;*



- Il processo decide se ciclare o bloccarsi
  - La decisione è soggetta a race condition
    - Evitata attraverso
      - Un approccio algoritmico
      - Uso di caratteristiche HW del computer

Supporto HW per la Sincronizzazione dei Processi

- Istruzioni indivisibili
  - Evitano race condition sulle locazioni di memoria
- Usate con una variabile di lock per implementare la SC e le operazioni indivisibili

```
entry_test:   if lock = closed                    Performed by
                  then goto entry_test;           an indivisible
              lock := closed;                     instruction

              {Critical section or
              indivisible operation}

              lock := open;
```

- *entry_test* eseguita con un'istruzione indivisibile
  - istruzione Test-and-set (TS)
  - Istruzione swap

Istruzione Test-and-Set

```
LOCK          DC    X'00'           Lock is initialized to open
ENTRY_TEST    TS    LOCK            Test-and-set lock
              BC    7, ENTRY_TEST   Loop if lock was closed

              ...                   { Critical section or
                                      indivisible operation }

              MVI   LOCK, X'00'     Open the lock(by moving 0s)
```

Figure 6.9 Implementing a critical section or indivisible operation by using test-and-set.

Implementazione di SC o operazione indivisibile con Test-and-Set

Esempio di Uso di TestaAndSet

```
bool TestAndSet (bool &target){
   bool retValue = target;
   target = True;
   return retValue;
}
```

critical section with test and set

Shared state:

```
lock = False
```

Thread one code:

```
1: while test_and_set(lock):
2:    do nothing
3: # critical section
4: lock = False
```

Thread two code: (same)

```
5: while test_and_set(lock):
6:    do nothing
7: # critical section
8: lock = False
```

Supporto HW per la Sincronizzazione: Swap

```
TEMP        DS    1              Reserve one byte for TEMP
LOCK        DC    X'00'          Lock is initialized to open
            MVI   TEMP, X'FF'    X'FF' is used to close the lock
ENTRY_TEST  SWAP  LOCK, TEMP
            COMP  TEMP, X'00'    Test old value of lock
            BC    7, ENTRY_TEST  Loop if lock was closed

            ...                  { Critical section or
                                   indivisible operation }

            MVI   LOCK, X'00'    Open the lock
```

Implementazione di SC o operazione indivisibile con Swap

Esempio d'uso di Swap

```
bool lock;
bool key;
void swap(bool &a, bool &b){
    boolean temp = a;
    a=b;
    b=temp;
}
```

```
while(1){
     key = true;
while(key)
     swap(lock,key);
/* SC */
lock = false;
}
```

Approcci alla Sincronizzazione

- Approcci algoritmici
  - Per implementare la mutua esclusione
  - Indipendente dalla piattaforma HW o SW
    - Attesa attiva per la sincronizzazione
- Primitive di sincronizzazione
  - Implementate usando istruzioni indivisibili e supporto del kernel
  - Ad esempio, *wait* e *signal* dei semafori
    - Problema: possono essere usate alla rinfusa
- Costrutti di programmazione concorrente
  - Monitor