

Comunicazioni fra processi/thread

- Processi/thread eseguiti concorrentemente hanno bisogno di interagire per comunicare e sincronizzarsi :
 - scambiare dati
 - utilizzare correttamente strutture dati condivise
 - eseguire azioni nella sequenza corretta

1

PROCESSI CONCORRENTI E COOPERANTI

Modelli di interazione fra processi

- Modello a memoria condivisa: i processi (o thread) condividono alcune variabili in memoria. Se un processo modifica una di queste variabili, tutti gli altri processi vedranno il nuovo valore.
- Modello a scambio di messaggi: i processi non condividono aree di memoria, ma possono inviarsi messaggi utilizzando istruzioni *send* e *receive*.

2

Sezioni/Regioni Critiche

Regione Critica : porzione di un processo che accede a strutture dati condivise

- punti potenziali di interferenza

Obiettivo : fare in modo che le regioni critiche di due processi non vengano mai eseguite contemporaneamente (mutua esclusione)

↳ Risultati dipendono da scheduling

3

Problemi di sincronizzazione fra thread dovuti alla condivisione di memoria

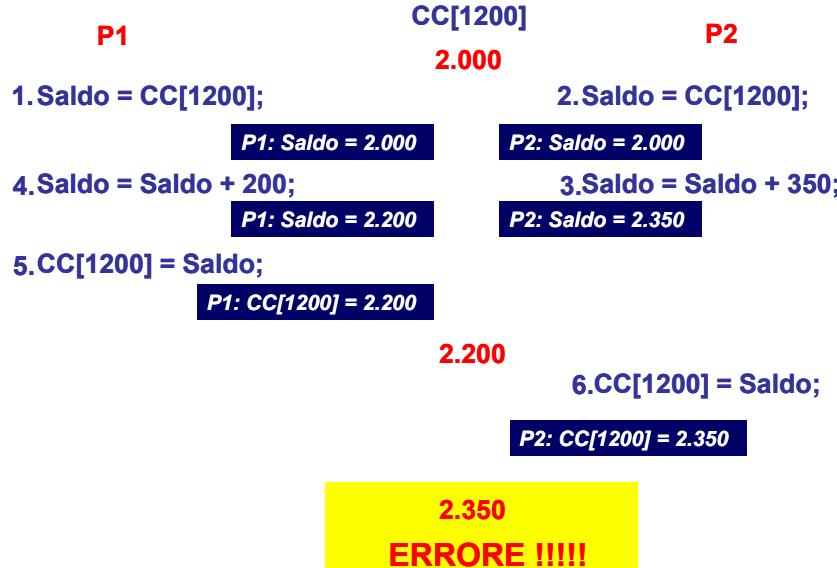
```
VersaSulConto(int numconto,int versamento)
{ Saldo = CC[numconto];
  Saldo = Saldo + versamento;
  CC[numconto] = Saldo; }
```

Supponiamo che due thread eseguano contemporaneamente la procedura VersaSulConto.

I thread condividono il vettore CC[] che contiene il saldo di tutti i conti correnti. La variabile Saldo è locale alla procedura, quindi diversa nei due thread (ognuno ha il suo stack dove si trovano le variabili locali). Invece CC[] è condiviso.

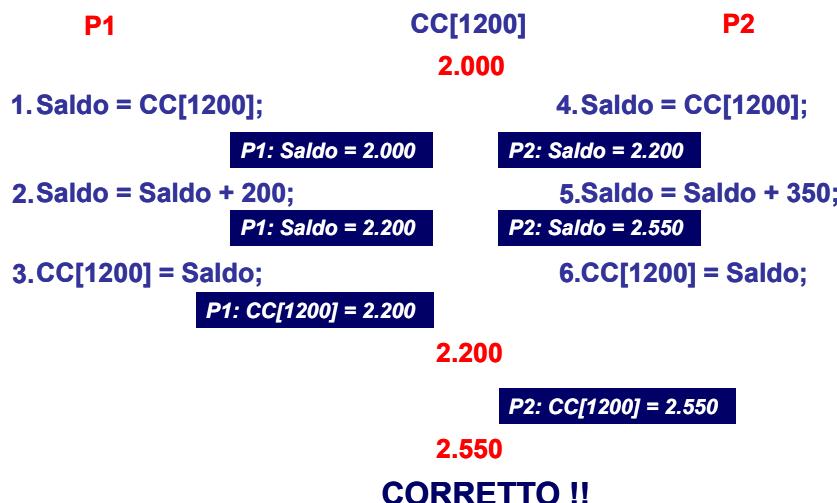
4

Problemi di sincronizzazione fra thread dovuti alla condivisione di memoria



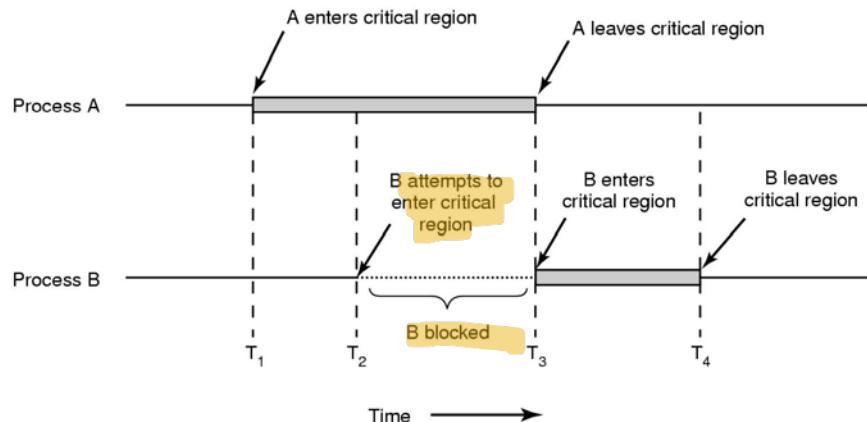
5

Problemi di sincronizzazione fra thread dovuti alla condivisione di memoria



6

Sezioni/Regioni Critiche



Mutua esclusione con sezioni critiche

7

Mutua esclusione di sezioni critiche

Requisiti per una buona soluzione

Una soluzione soddisfacente al problema della mutua esclusione di sezioni critiche, deve rispettare i seguenti quattro requisiti:

1. Non devono mai esserci due (o più) processi simultaneamente in sezione critica
2. Non possiamo fare alcuna assunzione sulla velocità relativa di esecuzione dei processi (soluzione indipendente da velocità e scelte di sched.)
3. Non vogliamo che un processo che non sia in sezione critica possa tenerne un altro bloccato fuori dalla propria sezione critica (un thread non deve attendere per operazioni fatte da altri thread non in rayore critico)
4. Non deve mai accadere che un processo debba attendere all'infinito di entrare nella propria sezione critica

Assumiamo che tutte le sezioni critiche abbiano un tempo di esecuzione finito (non ammettiamo che qualche sezione critica contenga un loop infinito).

8

Soluzione dei problemi di mutua esclusione in sezione critica

↑ Tipicamente non ci sono system call in sezione critica. Thread dovrebbe interruppi non consentendo a schedulatore di togliere la CPU.

DISABILITAZIONE INTERRUPT

PROBLEMI DI QUESTA SOLUZIONE:

- 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)
- la soluzione funziona correttamente solo se il sistema ha un'unica CPU: nei sistemi multiprocessore non funziona perché la disabilitazione degli interrupt è locale a ciascuna CPU, e i processi possono eseguire davvero in parallelo (anzichè in pseudo-parallelismo).

9

Soluzione Mutua Esclusione con attesa attiva (*busy waiting*)

- Soluzioni software
 - alternanza stretta
 - soluzione di Peterson
- Soluzioni hardware-software
 - l'istruzione TSL

10

Mutua Esclusione con attesa attiva

Alternanza stretta

Soluzione software implementa la mutua esclusione

```
while (TRUE) {
    while (turn != 0) /* loop */;
    critical_region();
    turn = 1;
    noncritical_region();
}
```

(a)

Processo 0

```
while (TRUE) {
    while (turn != 1) /* loop */;
    critical_region();
    turn = 0;
    noncritical_region();
}
```

(b)

Processo 1

Una soluzione non soddisfacente per il problema della ME

- Non soddisfa il requisito 3. Il processo più lento può rallentare notevolmente l'altro.

11

Mutua Esclusione con attesa attiva Soluzione di Peterson (semplificata)

```
#define FALSE 0
#define TRUE 1
#define N 2

int turn; /* number of processes */
int interested[N]; /* whose turn is it? */
/* all values initially 0 (FALSE) */

void enter_region(int process); /* process is 0 or 1 */
{
    int other; /* number of the other process */

    other = 1 - process; /* the opposite of process */
    interested[process] = TRUE; /* show that you are interested */
    turn = process; /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */;
}

void leave_region(int process); /* process: who is leaving */
{
    interested[process] = FALSE; /* indicate departure from critical region */
}
```

Pi: while(Fine lavoro)
{sez. non. critica;
enter_region(i);
sez. critica
leave_region(i); }
i ∈ {0,1}

avviso che entra e esce dalla regione

12

Se eseguo in momenti diversi enter_region, sono anch'essi interessati. Chi entra per primo? L'ultimo che scrive turn si ferma e aspetta. turn=0, 0 si ferma e l'altro passa.

Allora 1 è entrato nella regione e sta da solo, gli altri aspettano. Per uscire sette interessed[1] a falso, rendendo falsa seconda condizione da AND e farà entrare 0.
Se 1 vuole restare ancora non può perché interessed[0]=TRUE.

Mutua Esclusione con attesa attiva Uso di una variabile di lock

Inizializzazione:

lock = 0; /* lock è una variabile condivisa dai processi
vale 0 se nessuno sta eseguendo all'interno
della sezione critica, 1 altrimenti */

Prima di entrare in sezione critica:

while (lock == 1) /* attendi */;
lock = 1;

Mandare il problema dell'attesa
attiva.

In uscita dalla sezione critica:

lock = 0;

ATTENZIONE: Cosa accade se due processi valutano la condizione
(lock==1), entrambi la trovano falsa e passano entrambi ad eseguire
l'assegnazione lock = 1 ?!?!?

13

assembler: loop: MOVE LOCK, Dx
MOVE #1, LOCK
CMP Dx, #1
BEQ LOOP

intervento del thread → METTO IN DO LOCK CHE È 0.
INTERRUZIONE DI THREAD.
SECONDO THREAD FA LO STESSO E TROVA
LOCK=0.

Mutua Esclusione con attesa attiva L'istruzione Test and Set Lock (TSL)

Supponiamo che nel linguaggio macchina esista la seguente
istruzione: **TSL registro, indirizzo**
che svolge la seguente funzione **IN MODO ATOMICO**
(l'esecuzione non può essere intercalata con altre operazioni in
memoria ne' in caso di pseudo-parallelismo, ne' in caso di
parallelismo vero nei sistemi multiprocessore)

enter_region:

TSL REGISTER,LOCK
CMP REGISTER,#0
JNE enter_region

| copy lock to register and set lock to 1
| was lock zero?
| if it was non zero, lock was set, so loop

Busy Waiting

leave_region:

MOVE LOCK,#0
RET | return to caller

| store a 0 in lock

14

TST Registro Processore, Lock (zona di memoria).

↓ Legge variabile e setta 1 da Lock ma non molla ancora.

Soluzioni senza attesa attiva

Le primitive *Sleep* e *Wakeup* (1)

- Idea di base : un processo viene bloccato finché non è in grado di entrare nella sezione critica (in modo da non sprecare cicli di CPU)
 - Due primitive realizzate come system call
 - sleep()* :: blocca il processo che la invoca *Thread si mette in stato waiting*
 - wakeup(P)* :: sveglia il processo P
 - ↓ System call per svegliare
 - La attiva il processo che era.
- System Call che risveglia il processo che ha fatto sleep

15

Soluzione errata al produttore-consamatore con sleep e wakeup

```
void producer(void)
{
    int item;
    while (TRUE){
        item = produce_item();
        if (count==N) sleep();
        insert_item(item);
        count++;
        if (count==1) wakeup(consumer);
    }
}

void consumer(void)
{
    int item;
    while (TRUE){
        if (count==0) sleep();
        item = remove_item();
        count--;
        if (count==N-1) wakeup(producer);
        process_item(item);
    }
}
```

Supponiamo di essere nella seguente situazione: il consumatore ha trovato *count==0* e sta per eseguire la *sleep()*, ma appena prima di eseguirla è stato interrotto (finito il suo timeslice) ed è stato schedulato il produttore il quale ha inserito un item, ha incrementato *count*, e poiché *count==1* ha eseguito *wakeup(consumer)*. Quest'ultima tuttavia non ha effetto dato che il consumatore non è bloccato (non ha ancora eseguito la *sleep()*). A questo punto il consumatore esegue la *sleep()*, e va a dormire ... per sempre...

Il problema è accedere alla variabile *count* condivisa.

16

Ovvio che sleep e wakeup devono essere usati in modo corretto e abusivo: quando la lampadina è fermata non può essere accesa

Meccanismi di sincronizzazione

I semafori

Il semaforo è un tipo di dato con associate le operazioni (atomiche):

- `init(int val_ini);`
- `up();`
- `down();`

Sia `s` un semaforo: esso ha un valore `s.val >= 0` che al momento della creazione deve essere impostato tramite la procedura `init`. Una volta inizializzato, `s.val` si può modificare solo attraverso le procedure `up` e `down` (**ATOMICHE!**) che si comportano nel modo seguente:

```
s.down() : if (s.val == 0)
            appendi(s.queue, processo);
            sleep(); // attendiamo che il valore diventi positivo
            else s.val--;
            // decrementiamo s.val

s.up() : if (s.queue non vuota)
            wakeup(estrai(s.queue));
            else
                s.val++;
```

PSEUDO CODICE

17

`s.val` è la lampadina di colore `lock` 0 oppure 1. Basso accenduto = spagnolo.

Prima di entrare ← `s.down` guarda stato del semaforo. Vede se rosso o verde. Se valore del semaforo è 0 chiama `sleep`.
In reg. entrata 0=rosso e non fermo. Se è verde lo mette a rosso.

Prima di uscire ← `s.up`: guarda la coda e lo sveglia. Lampadina rimane a rosso, affannosi molto a verde.

Mutua Esclusione con semafori

Per garantire l'esecuzione in mutua esclusione di sezioni critiche si può utilizzare un semaforo `s`, condiviso da tutti i processi che contengono sezioni critiche relative a determinate strutture dati (per esempio al vettore `CC[]` dell'esempio `VersaSulConto`):

```
semaphore s;
s.init(1);
...
s.down();
Sezione Critica
s.up();
```

Il semaforo è verde per l'ingresso in sezione critica quando vale 1, mentre è rosso quando vale 0. Un processo che trova il semaforo verde lo imposta a rosso ed entra (down). Il semaforo torna verde quando il processo esce dalla sezione critica (up).

18

Problemi di mutua esclusione nella procedura VersaSulConto: uso dei semafori

P1	sem_cc	CC[1200]	P2
	1	2.000	
1. sem_cc.down();	0		7. sem_cc.down();
2. Saldo = CC[1200];			8. Saldo = CC[1200];
			P2: Saldo = 2.200
4. Saldo = Saldo + 200;			9. Saldo = Saldo + 350;
			P2: Saldo = 2.550
5. CC[1200] = Saldo;			10. CC[1200] = Saldo;
			P1: CC[1200] = 2.200
6. sem_cc.up();	0	2.200	11. sem_cc.up();
			P2: CC[1200] = 2.550
	1	2.550	
<p style="text-align: center;"><i>da slega manuale il valore del semaforo a 0.</i></p>			

19

Meccanismi di sincronizzazione I semafori

Le procedure up e down devono essere implementate come azioni atomiche.

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 (senza interruzioni) si potrebbe usare la tecnica della disabilitazione degli interrupt (in questo caso si potrebbe fare dato che verrebbe usata esclusivamente dal S.O.) oppure, se il sistema di calcolo è multiprocessore, si puo' usare la soluzione al problema della mutua esclusione basato sulla istruzione TSL. Anche se questa soluzione richiede l'uso dell'attesa attiva, in questo caso puo' essere accettabile poiche' la sezione critica è molto breve (si tratta del codice della up o della down).

Al termine della sezione critica si chiude la sezione critica.

20

Ottimamente facciamo synchronization su questi metodi brevi, che è accettabile.

implementano meccanismo di sincronizzazione

Produttore/Consumatore con semafori (Produttore)

Down: se valore di var.
è > 0 decrement. Altrimenti
Sleep.

Se n'ha 2+ semafori a
2, ma va bene da 1.
DOWN e UP possono entrare
fino a max di thread.

Thread entra e cerca un
posto. Quando completo esco.
e faccio entrare il successivo.

```
#define N 100
typedef int semaphore;
semaphore mutex = 1; semaforo binario
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}
```

/* number of slots in the buffer */
/* semaphores are a special kind of int */
/* controls access to critical region */
/* counts empty buffer slots */
/* counts full buffer slots */

21

Problema: produttore riempie la coda. Deve evitare che produttori in coda prima non entrano nella
regione critica. Voglio che produttori con coda prima si fermino prima di entrare.

Produttore/Consumatore con semafori (Consumatore)

```
void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

*Esempio: per i produttori questi posti liberi
sono disponibili. Full dice ai consumatori
quanti oggetti sono disponibili nella coda.*

/* infinite loop */
/* decrement full count */
/* enter critical region */
/* take item from buffer */
/* leave critical region */
/* increment count of empty slots */
/* do something with the item */

22

Deadlock: per qualche motivo, i thread sincronizzati male, allora finisce l'uno con l'altro e il programma va in stallo



Possibile situazione di deadlock

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}
```

Semplice scambio ordine provoca lo stallo

23

Cosa succede se scambio semafori? Thread produce ha mutex verde. Poi fa down su empty e ha then rosso e si ferma. Up(&empty) può svegliare quel thread. Potrebbe svegliarsi su empty può essere svegliato solo da consumer, ma dove trovare mutex a verde, e non lo ha.

Monitor (1)

- Oggetti (Strutture dati + procedure per accedervi)
- Mutua esclusione nell'esecuzione delle procedure
- Variabili di Condizione + `wait()` e `signal()`
- `wait(X)`
 - sospende sempre il processo che la invoca in attesa di una `signal(X)`
- `signal(X)`
 - sveglia uno dei processi in coda su X
 - se nessun processo è in attesa va persa (i signal non si accumulano)
 - deve essere eseguita solo come ultima istruzione prima di uscire dal monitor (il processo svegliato deve avere l'uso esclusivo del monitor)

24

Ho code che voglio rendere accessibile ad thread per esempio, con metodi per inserire e rimuovere oggetti nella col. Monitor: oltre sono struttura della condizion e metodi di accesso ma tutti i metodi della classe sono eseguiti in mutua esclusione. All'interno del monitor ci deve essere al massimo 1 thread. In più, dentro la classe posso inserire variabili di condizione: struttura dati su cui i thread possono fare wait, cioè bloccarsi su una code associata a questa variabile. I thread si bloccano su queste code facendo wait. Wait è simile a quello dei semafori: rilascio lock su monitor e aspetto. Op. di wait e signal sono protetti da

condizioni logiche, perab' variabile da condizionare. I thread possono fare op. di signal, che sveglia un processo che fa tutto vivere.

31/03/2017

ULTIMA REGOLA: signal alla fine, SEMPRE

Vantaggio: si risolve in un solo oggetto tutti i problemi di sincronizzazione nell'oggetto di partenza.

Monitor (2)

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count - 1;
        if count = N - 1 then signal(full)
    end;
    count := 0;
end monitor;
```

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end;
end;
```

- Schema di soluzione Produttore/Consumatore

- ad ogni istante solo una procedura del monitor è in esecuzione
- il buffer ha N posizioni

25

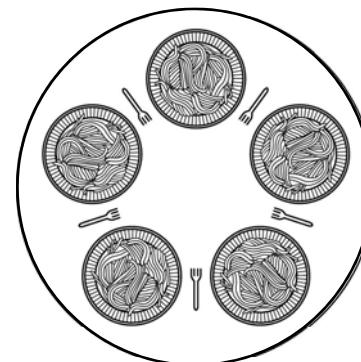
empty: mi dice se la struttura è vuota e mi fanno in questi casi

full: mi dice qui se ho capo pieno per inserire.

Ricorda: accesso a metodi è mutua esclusione. Se entro in insert sto da solo.

I filosofi a cena (1)

- I filosofi mangiano e pensano
- Per mangiare servono due forchette
- Ogni filosofo prende una forchetta per volta
- Come si può prevenire il deadlock



26

I filosofi a cena (2)

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                       /* i: philosopher number, from 0 to 4 */

{
    while (TRUE) {
        think();                                /* philosopher is thinking */
        take_fork(i);                           /* take left fork */
        take_fork((i+1) % N);                  /* take right fork; % is modulo operator */
        eat();                                   /* yum-yum, spaghetti */
        put_fork(i);                            /* put left fork back on the table */
        put_fork((i+1) % N);                  /* put right fork back on the table */
    }
}
```

Una falsa soluzione al problema dei filosofi

Dadlock! Immaginiamo tutti prendono la forchetta sinistra.

27

I filosofi a cena (3)

```
#define N 5                                     /* number of philosophers */
#define LEFT (i+N-1)%N                         /* number of i's left neighbor */
#define RIGHT (i+1)%N                          /* number of i's right neighbor */
#define THINKING 0                             /* philosopher is thinking */
#define HUNGRY 1                               /* philosopher is trying to get forks */
#define EATING 2                               /* philosopher is eating */
typedef int semaphore;                        /* semaphores are a special kind of int */
int state[N];                                /* array to keep track of everyone's state */
semaphore mutex = 1;                          /* mutual exclusion for critical regions */
semaphore s[N];                             /* one semaphore per philosopher */

void philosopher(int i)                      /* i: philosopher number, from 0 to N-1 */

{
    while (TRUE) {
        think();                                /* repeat forever */
        take_forks(i);                         /* philosopher is thinking */
        eat();                                   /* acquire two forks or block */
        put_forks(i);                           /* yum-yum, spaghetti */
        /* put both forks back on table */
    }
}
```

Tutti a
rossi davanti

Una soluzione corretta al problema (parte 1)

28

Le prendo tutte e due oppure mi blocco

La vera sincronizzazione è un semaforo mutex e un semaforo

* Semafori privati, che sostanzialmente sono usati dai

filosofi per sapere se ci sono le condizioni per mangiare.

I filosofi a cena (4)

```

void take_forks(int i)
{
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}

void put_forks(i)
{
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}

void test(i)
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}

```

perché state[i] è vero risulta di una esclusione

/* i: philosopher number, from 0 to N-1 */
 /* enter critical region */
 /* record fact that philosopher i is hungry */
 /* try to acquire 2 forks */
 /* exit critical region */
 /* block if forks were not acquired */
 /* i: philosopher number, from 0 to N-1 */
 /* enter critical region */
 /* philosopher has finished eating */
 /* see if left neighbor can now eat */
 /* see if right neighbor can now eat */
 /* exit critical region */
 /* i: philosopher number, from 0 to N-1 */

Una soluzione corretta al problema (parte 2)

29