

Sistemi Operativi 1

AA 2018/2019

Sincronizzazione tra processi

Sommario

- Introduzione
- Problema della sezione critica
 - Consistenza dei dati
- Soluzioni basate su attesa attiva (busy waiting)
 - Metodi software
 - Metodi hardware
- Semafori
- Primitive ad alto livello
 - Monitor
 - Classi sincronizzate



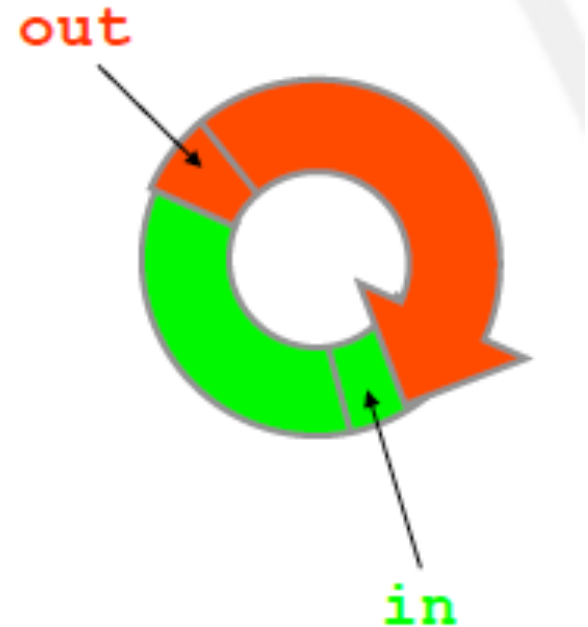
Sincronizzazione tra processi

- Modello astratto (produttore-consumatore)
 - Produttore: produce un messaggio
 - Consumatore: consuma un messaggio
 - Esecuzione concorrente
 - Produttore aggiunge al buffer
 - Consumatore toglie dal buffer
 - Vincoli (buffer limitato)
 - Non posso aggiungere in buffer pieni
 - Non posso consumare da buffer vuoti



Buffer P/C: modello SW

- Buffer circolare di N posizioni
 - in = prima posizione libera
 - out = prima posizione occupata
- Buffer vuoto
 - in = out
- Buffer pieno
 - $out = (in + 1) \% n$
- Per semplicità
 - Usiamo una variabile *counter* per indicare il numero di elementi nel buffer
 - counter = 0 buffer vuoto
 - counter = N buffer pieno



Buffer P/C: modello SW

Produttore

```
void deposit (item p)
{
    while (counter == N)
        no_op;
    buffer[in] = p;
    in = (in+1) % N;
    counter++;
}
```

Consumatore

```
item remove ()
{
    while (counter == 0)
        no_op;
    next = buffer[out];
    out = (out+1) % N;
    counter--;
    return next;
}
```



Buffer P/C: problema

Come sono implementate `counter++` e `counter--`

- In Assembly sono separate in più istruzioni

`counter++`

```
reg1 = counter;  
reg1 = reg1 + 1;  
counter = reg1;
```

`counter--`

```
reg2 = counter;  
reg2 = reg2 - 1;  
counter = reg2;
```

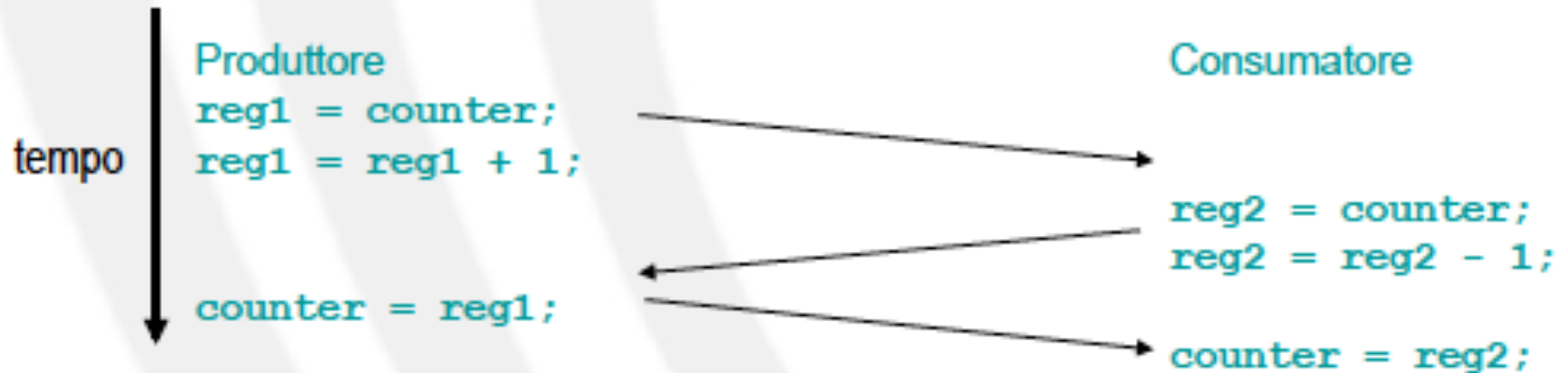
In che ordine vengono eseguite le istruzioni
Assembly?

- Sequenzialmente, ma non è noto l'ordine di interleaving



Buffer P/C: problema

- Se l'esecuzione è alternata nel seguente modo:



- Inconsistenza!
- Es.: supponiamo *counter* = 5
 - P produce un item → *counter*++ (idealmente *counter* diventa 6)
 - C consuma un item → *counter*-- (idealmente *counter* ritorna 5)
 - Quanto vale *counter*? 4 invece di 5!

Buffer P/C: problema

- Qual è il problema?
 - P e C possono modificare *counter* contemporaneamente
- E' importante proteggere l'accesso alla **sezione critica**



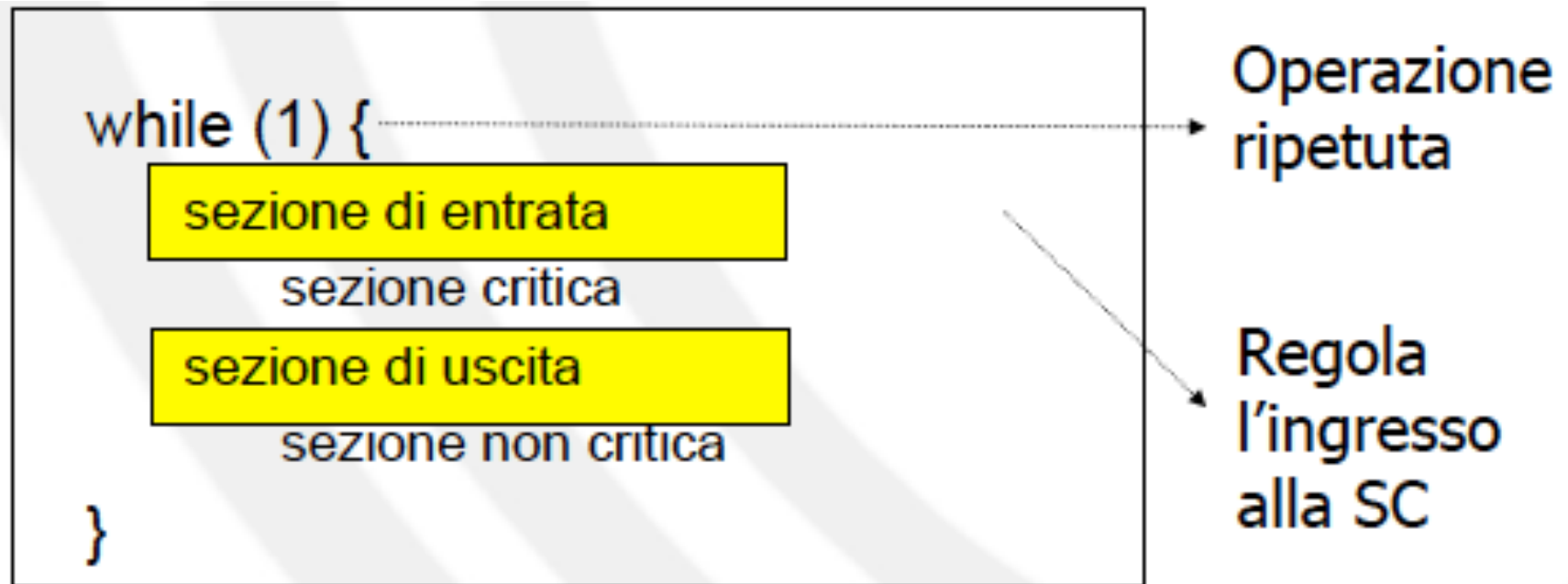
Sezione Critica (SC)

- Porzione di codice in cui si accede ad una risorsa condivisa (es.: modifica variabile)
- Soluzione al problema deve rispettare 3 criteri:
 - **Mutua esclusione**
 - Un processo alla volta può accedere alla sezione critica
 - **Progresso (progress)**
 - Solo i processi che stanno per entrare nella sezione critica possono decidere chi entra
 - La decisione non può essere rimandata all'infinito
 - **Attesa limitata (bounded waiting)**
 - Deve esistere un massimo numero di volte per cui un processo può aspettare (di seguito)



Sezione critica

- Struttura di un generico processo che accede ad una risorsa condivisa



Sezione critica: soluzioni

- Assunzione:
 - Sincronizzazione in ambiente globale
 - Condivisione di celle di memoria (variabili “condivise”)
- Soluzioni software:
 - Aggiunta di codice alle applicazioni
 - Nessun supporto hardware o del S.O.
- Soluzioni “Hardware”:
 - Aggiunta di codice alle applicazioni
 - Necessario supporto hardware



SOLUZIONI SOFTWARE

Algoritmo 1

2 soli processi
($i=0,1$); $j = 1-i$

```
PROCESS i
int turn; /* se  $turn = i$  allora entra il processo  $i$  */
while (1) {
    while (turn != i); /* sezione di entrata */
    sezione critica
    turn = j; /* sezione di uscita */
    sezione non critica
}
```

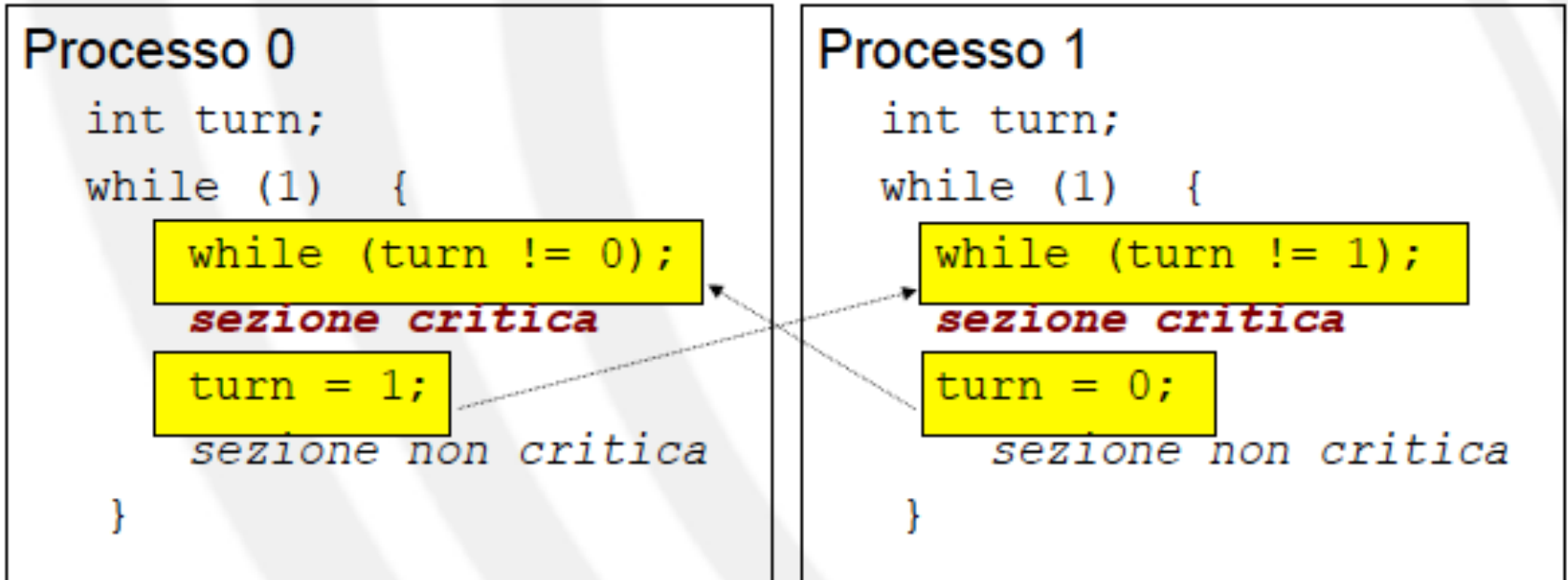


Algoritmo 1: problema

- Richiede stretta alternanza tra i processi
 - Se i oppure j non sono interessati ad entrare in SC, anche l'altro processo non può più entrare in SC
- Non rispetta il criterio del progresso
 - Non c'è nessuna nozione di “stato”



Algoritmo 1: problema



- Se P_0 cede il turno a P_1 e non ha più necessità di entrare nella sezione critica anche P_1 non può più entrare nella sezione critica!

Algoritmo 2

2 soli processi
(i=0,1); j = 1-i

```
PROCESS i
boolean flag[2]; /* inizializzato a FALSE */
while (1) {
    flag[i]=true; /* vuole entrare in SC */
    while (flag[j]==true); /* sezione di entrata */
    sezione critica
    flag[i]=false; /* sezione di uscita */
    sezione non critica
}
```



Algoritmo 2: problema

- Risolve problema dell'algoritmo 1 ma...
- l'esecuzione in sequenza dell'istruzione `flag[]=true` da parte dei due processi porta a deadlock



Algoritmo 2

- Sequenza di operazioni critica
 - t_0 : P0 esegue `flag[0]=TRUE`
 - t_1 : P1 esegue `flag[1]=TRUE`
 - t_2 : P0 esegue `while(flag[1]==TRUE);`
 - t_3 : P1 esegue `while(flag[0]==TRUE);`
 - t_4 : DEADLOCK!
- P0 e P1 bloccati sulla condizione del while



Algoritmo 2 (variante)

2 soli processi
($i=0,1$); $j = 1-i$

- Cosa succede se invertiamo le istruzioni della sezione di entrata?

```
PROCESS i
boolean flag[2]; /* inizializzato a FALSE */
while (1) {
```

```
    while (flag[j]==true); /* sezione di entrata */
    flag[i]=true; /* vuole entrare in SC */
```

sezione critica

```
    flag[i]=false; /* sezione di uscita */
```

sezione non critica

```
}
```

Algoritmo 2 (variante)

- Invertendo le istruzioni della sezione di entrata violiamo la mutua esclusione
 - Entrambi i progetti possono trovarsi in SC se eseguono in sequenza il *while* prima di impostare la *flag* a true



Algoritmo 3

2 soli processi

```
PROCESS i
int turn;                                /* di chi è il turno? */
boolean flag[2];                         /* iniz. a FALSE */
while (1) {
    flag[i] = TRUE;                      /* voglio entrare */
    turn = j;                            /* tocca a te, se vuoi */
    while (flag[j] == TRUE && turn == j);
    sezione critica
    flag[i] = FALSE;
    sezione non critica
}
```

Algoritmo 3

- E' la soluzione corretta
 - Entra il primo processo che esegue $\text{turn} = j$ (oppure $\text{turn} = i$)
- Come si dimostra?



Algoritmo 3 (dimostrazione)

- Mutua esclusione
 - P_i entra nella SC sse $flag[j]=false$ o $turn=i$
 - Se P_i e P_j sono entrambi in SC allora $flag[i]=flag[j]=true$
 - Ma P_i e P_j non possono aver superato entrambi il *while*, perché *turn* vale *i* oppure *j*
 - Quindi solo uno dei due P è entrato



Algoritmo 3 (dimostrazione)

- Progresso e attesa limitata
 - Se P_j non è pronto per entrare nella SC allora $flag[j]=false$ e P_i può entrare
 - Se P_j ha impostato $flag[j]=true$ e si trova nel *while* allora $turn=i$ oppure $turn=j$
 - Se $turn=i$ P_i entra nella SC
 - Se $turn=j$ P_j entra nella SC
 - In ogni caso quando P_j esce dalla SC imposta $flag[j]=false$ e quindi P_i può entrare nella SC
 - Quindi P_i entra nella SC al massimo dopo un'entrata di P_j



Algoritmo del fornaio

- Risolve il problema con N processi
- Idea:
 - Ogni processo sceglie un numero ($\text{choosing}[i]=1$)
 - Il numero più basso verrà servito per primo
 - Per situazioni di numero identico (può accadere), si usa un confronto a due livelli (*numero, i*)
- Algoritmo corretto
 - Soddisfa le tre proprietà (esercizio)



Algoritmo del fornaio versione 0.1

N processi

PROCESS *i*

Iniz. a 0 `int number[N];` */* ultimo numero scelto */*

Prendo un
numero

`while(1) {`

`number[i] = Max(number[0], ..., number[N-1]) + 1;`

`for(j = 0; j < N; j++){`

`while(number[j] != 0 && number[j] < number[i]);`
`}`

*j è in CS e
ha numero
inferiore*

sezione critica

`number[i] = 0;`

sezione non critica

`}`



Algoritmo del fornaio versione 0.2

N processi

```
PROCESS i
Iniz. a false   boolean choosing[N];   /* Pi sceglie un numero */
Iniz. a 0       int number[N];         /* ultimo numero scelto */

while(1) {
    Prendo un numero → choosing[i] = TRUE;
                      number[i] = Max(number[0], ..., number[N-1]) + 1;
                      choosing[i] = FALSE;
                      for(j = 0; j < N; j++) {
j sta scegliendo → while(choosing[j] == TRUE);
j è in CS e ha numero inferiore → while(number[j] != 0 && number[j] < number[i]);
                                }
                                sezione critica
                                number[i] = 0;
                                sezione non critica
                      }
}
```

Algoritmo del fornaio versione 1.0

PROCESS i

boolean choosing[N];

/* Pi sceglie un numero */

int number[N];

/* ultimo numero scelto */

while(1){

choosing[i] = TRUE;

number[i] = Max(number[0],...,number[N-1])+1;

choosing[i] = FALSE;

for(j = 0; j < N; j++){

while(choosing[j] == TRUE);

while(number[j] != 0 &&

(number[j] < number[i] || (number[j] == number[i]) && (j < i)));

}

sezione critica

number[i] = 0;

sezione non critica

}



SOLUZIONI HARDWARE

Soluzioni hardware

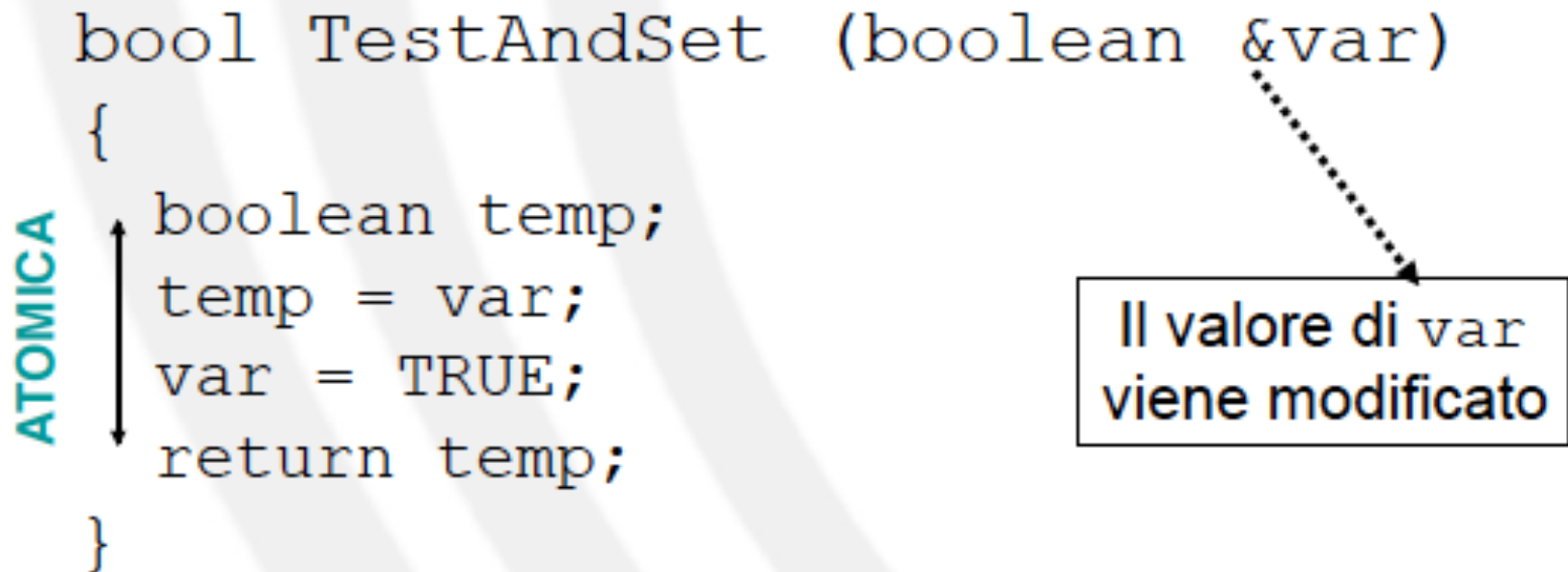
- Un modo “hardware” per risolvere il problema SC è quello di **disabilitare gli interrupt** mentre una variabile condivisa viene modificata
- Problema
 - Se il test per l’accesso è “lungo”, gli interrupt devono essere disabilitati per troppo tempo
- Alternativa
 - L’operazione per l’accesso alla risorsa deve occupare un unico ciclo di istruzione (non interrompibile!)
 - Soluzioni:
 - Test-and-set
 - Swap

**Istruzioni
“atomiche”**

Test and Set

```
bool TestAndSet (boolean &var)
{
    boolean temp;
    temp = var;
    var = TRUE;
    return temp;
}
```

ATOMICA



Il valore di `var`
viene modificato

- Valore di ritorno: vecchio valore di `var`
- Assegna `TRUE` a `var`

Test and Set - Utilizzo

```
boolean lock;    /* globale iniz. FALSE */
```

```
while (1) {  
    while (TestAndSet(lock)) ;
```

sezione critica

```
    lock = FALSE;
```

sezione non critica

```
}
```

Passa solo il primo
processo che arriva
trova *lock = FALSE*

Swap

ATOMICA

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

- Concetto: scambio il valore di a e b



Swap - Utilizzo

```
boolean lock;          /* globale, inizializzata a FALSE */

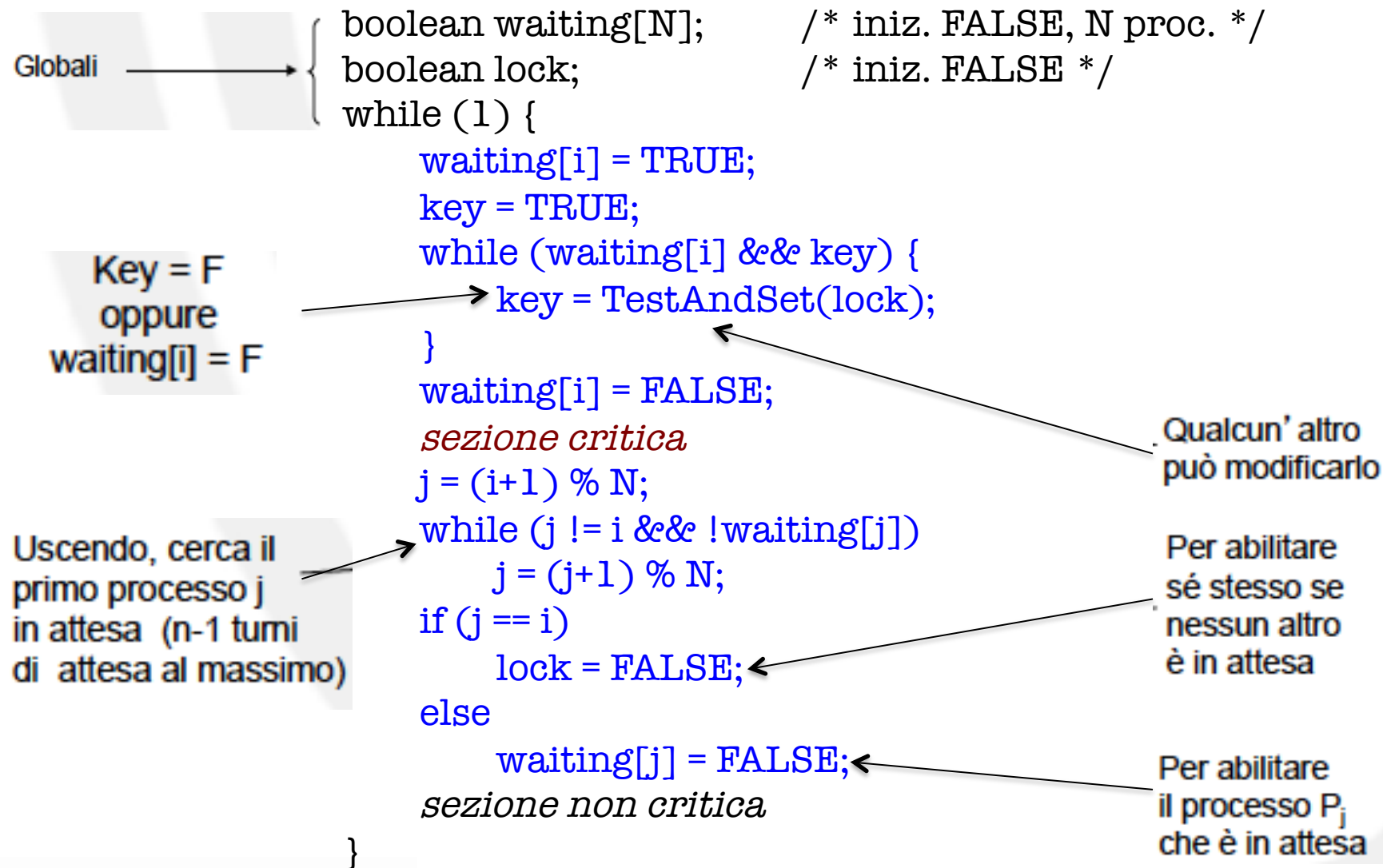
while (1) {
    dummy = TRUE;      /* locale al processo */
    do
        Swap(dummy, lock);
    while (dummy == TRUE);
    sezione critica
    lock = FALSE;
    sezione non critica
}
```

Quando dummy=false P_i accede alla SC.
Gli altri processi continuano a scambiare *true* con *true* e non accedono a SC finché P_i non pone lock=false

NB: TestAndSet e Swap non rispettano attesa limitata

- Manca l'equivalente della variabile turn
- Necessarie variabili aggiuntive

Test and Set con attesa limitata



Soluzioni hardware

- Vantaggi
 - Scalabili
 - indipendenti dal numero di processi coinvolti
 - Estensione a N sezioni critiche immediato
- Svantaggi
 - Maggiore complessità per il programmatore rispetto alle soluzioni SW
 - Es.: come impongo l'attesa limitata alle soluzioni precedenti?
 - Serve busy waiting → spreco CPU

SEMAFORI

Semafori

- Problemi soluzioni precedenti
 - Non banali da aggiungere a programmi
 - Basate su busy waiting (attesa attiva)
- Alternativa: semafori
 - Soluzione generica che funziona sempre



Semafori

- E' una variabile intera S a cui si accede attraverso due primitive (atomiche)
 - Signal: $V(s)$ (dall'olandese Verhogen = incrementare)
 - incrementa il valore di S di 1
 - Wait: $P(s)$ (dall'olandese Proberen = testare)
 - tenta di decrementare il valore di S di 1
 - se il valore di S è = 0
 - Non si può decrementare
 - Necessario attendere
- Semafori binari ($S = 0$ oppure 1)
- Semafori generici ($S =$ valori interi ≥ 0)



Semafori binari

- Implementazione “concettuale” (\neq da reale)

P(s):

```
while (s == FALSE); // attesa  
s = FALSE;
```

V(s):

```
s = TRUE;
```

- N.B.: I semafori binari hanno lo stesso potere espressivo di quelli a valori interi



Semafori a valori interi

- Implementazione “concettuale” (\neq da reale)

P(s):

```
while (s == 0); // attesa  
s--;
```

V(s):

```
s ++;
```

- Problema: devo garantire l'atomicità!
- Implementazione?



Semafori binari - implementazione

- Con busy waiting

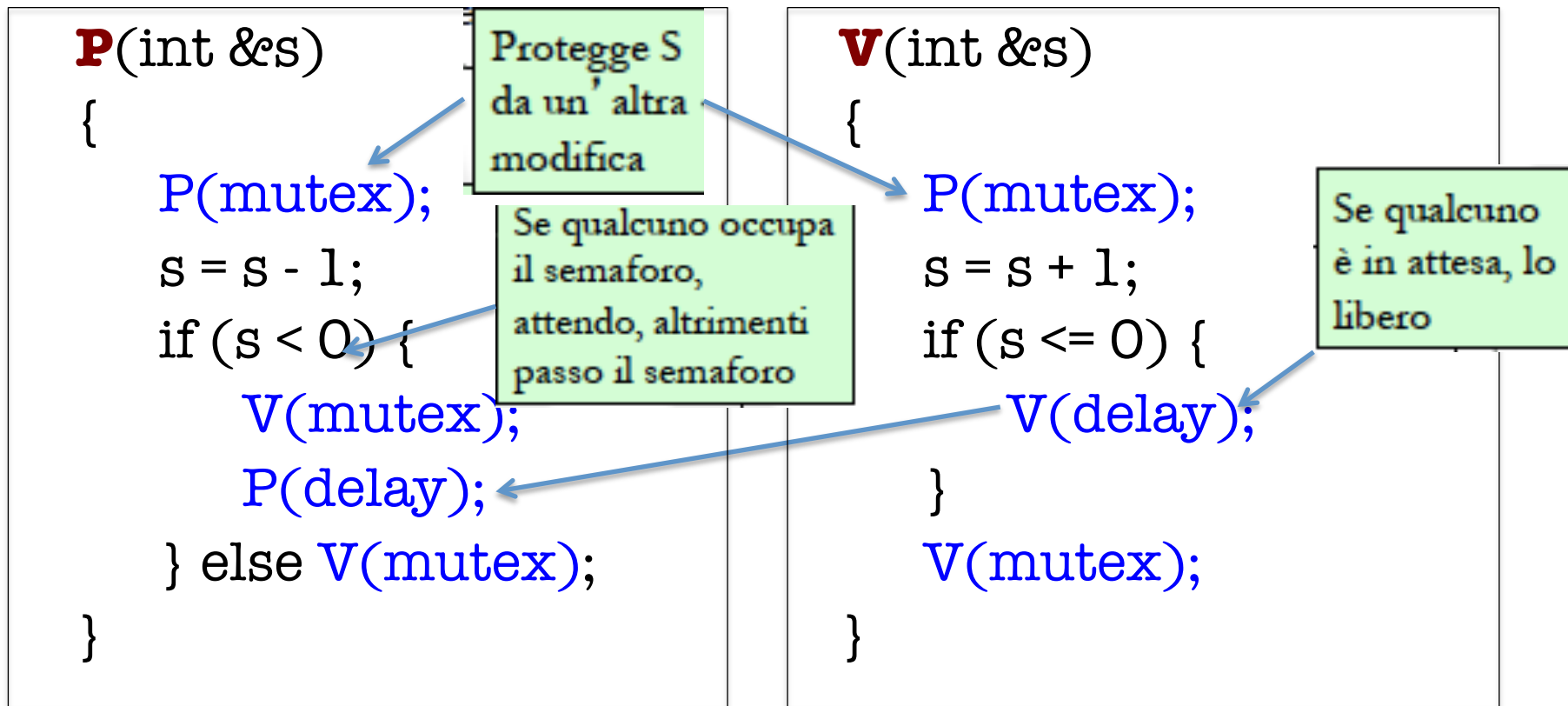
```
/* s inizializzato a TRUE */  
P(bool &s)  
{  
    key = FALSE;  
    do {  
        Swap(s, key);  
    } while (key == FALSE);  
}
```

```
V(bool &s)  
{  
    s = TRUE;  
}
```

Semafori interi - implementazione

- Con busy-waiting

```
bool mutex; /* Sem. binario iniz. TRUE */  
bool delay; /* Sem. binario iniz. FALSE */
```

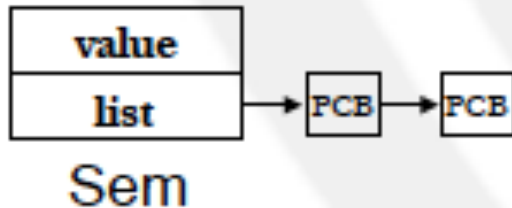


Semafori interi - implementazione

- senza busy-waiting

bool mutex /* sem bin. inizializzato a true */

```
typedef struct {  
    int value;  
    PCB *List;  
} Sem;
```



```
P (Sem &s){  
    P(mutex);  
    s.value = s.value - 1;  
    if (s.value < 0) {  
        V(mutex);  
        append (process i, s.List);  
        sleep();  
    } else  
        V(mutex);  
}
```

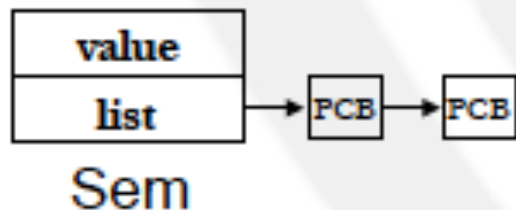
Mette il processo nello stato waiting

Semafori interi - implementazione

- senza busy-waiting

bool mutex /* sem bin. inizializzato a true */

```
typedef struct {  
    int value;  
    PCB *List;  
} Sem;
```



```
V(Sem &s){  
    P(mutex);  
    s.value = s.value + 1;  
    if (s.value <= 0) {  
        V(mutex);  
        PCB *p = remove(s.List);  
        wakeup(p);  
    } else  
        V(mutex);  
}
```

Mette il processo nello stato ready. In che ordine?

Semafori senza busy-waiting

- Busy waiting eliminato dalla entry section
 - Entry section può essere lunga → risparmio
- Rimane nella P e nella V del mutex
 - Modifica del mutex è veloce → poco spreco
- Alternativa:
 - Disabilitare interrupt durante **P** e **V**
 - Istruzioni di processi diversi non possono essere eseguite in modo alternato

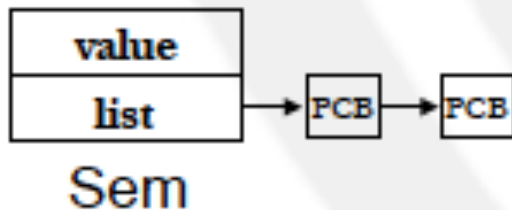


Semafori interi - implementazione

- senza busy-waiting

bool mutex /* sem bin. inizializzato a true */

```
typedef struct {  
    int value;  
    PCB *List;  
} Sem;
```



```
P (Sem &s){  
    P(mutex);  
    s.value = s.value - 1;  
    if (s.value < 0) {  
        V(mutex);  
        append (process i, s.List);  
        sleep();  
    } else  
        V(mutex);  
}
```

Senza busy waiting?

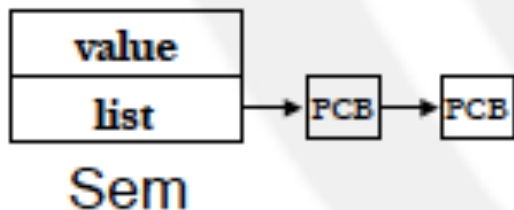
Mette il processo nello stato waiting

Semafori interi - implementazione

- senza busy-waiting

bool mutex /* sem bin. inizializzato a true */

```
typedef struct {  
    int value;  
    PCB *List;  
} Sem;
```



```
V(Sem &s){  
    P(mutex);  
    s.value = s.value + 1;  
    if (s.value <= 0) {  
        V(mutex);  
        PCB *p = remove(s.List);  
        wakeup(p);  
    } else  
        V(mutex);  
}
```

Senza busy waiting?

Mette il processo nello stato ready. In che ordine?

Semafori - implementazione

- Implementazione reale \neq concettuale
 - Il valore di s può diventare < 0 per semafori interi!
 - Conta quanti processi sono in attesa
- La lista dei PCB può essere FIFO (*strong semaphore*)
 - Garantisce attesa limitata



Semafori - applicazioni

- Usi principali
 - Semaforo binario con valore iniziale = 1 (mutex)
 - Utilizzo: Protezione di sezione critica per n processi
 - Semaforo (binario) con valore iniziale = 0
 - Utilizzo: sincronizzazione (del tipo attesa di evento) tra processi

Semafori e sezione critica

- Mutex = semaforo binario di mutua esclusione
- N processi condividono la variabile S

```
/* valore iniziale di s = 1 (mutex) */  
while (1) {  
    P(s);  
    sezione critica  
    V(s);  
    sezione non critica  
}
```



Semafori per attesa evento

- Sincronizzazione generica
 - Processi P1 e P2 devono sincronizzarsi rispetto all'esecuzione di due operazioni A e B
 - P2 può eseguire B soltanto dopo che P1 ha eseguito A

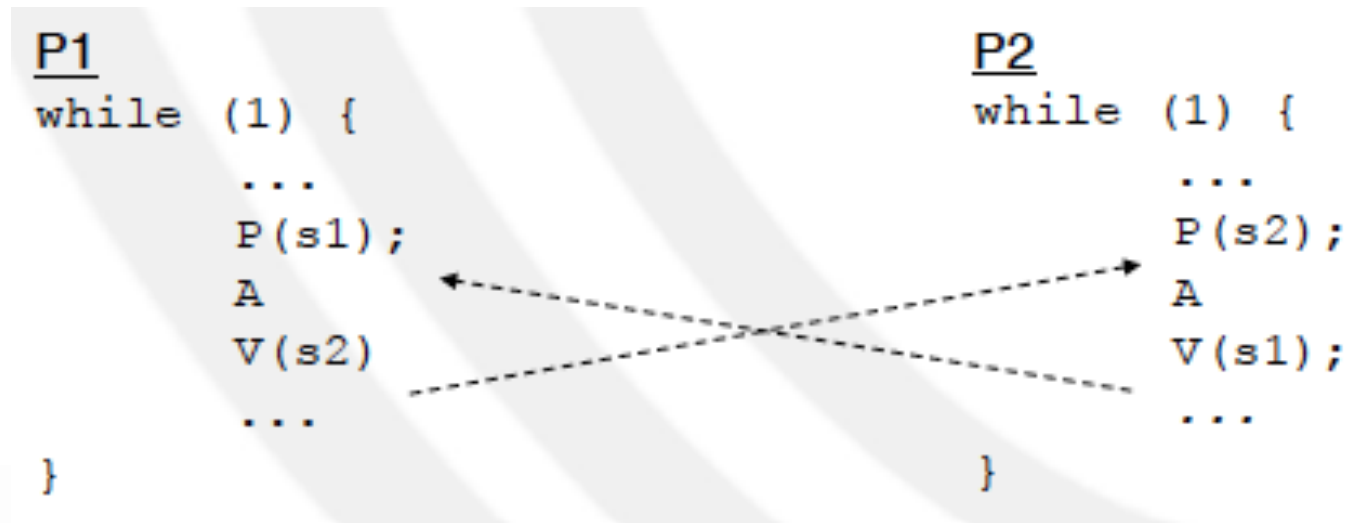
Soluzione: uso di un semaforo binario s inizializzato a 0



Semafori per attesa evento

- Sincronizzazione generica
 - Processi P1 e P2 devono sincronizzarsi rispetto all'esecuzione di un'operazione A
 - Utilizzo di A: P1 → P2 → P1 → P2 → ...

Soluzione: Uso di due semafori binari: S1 inizializzato a 1 e s2 inizializzato a 0



Nascosta

Semafori - limitazioni

- Deadlock (blocco critico)
 - Processo bloccato in attesa di un evento che solo lui può generare



- Starvation
 - Attesa indefinita all'interno di semaforo

Semafori – problemi classici

- Problema del produttore – consumatore
- Problema dei dining philosophers
- Problema dello sleepy barber

Produttore - Consumatore

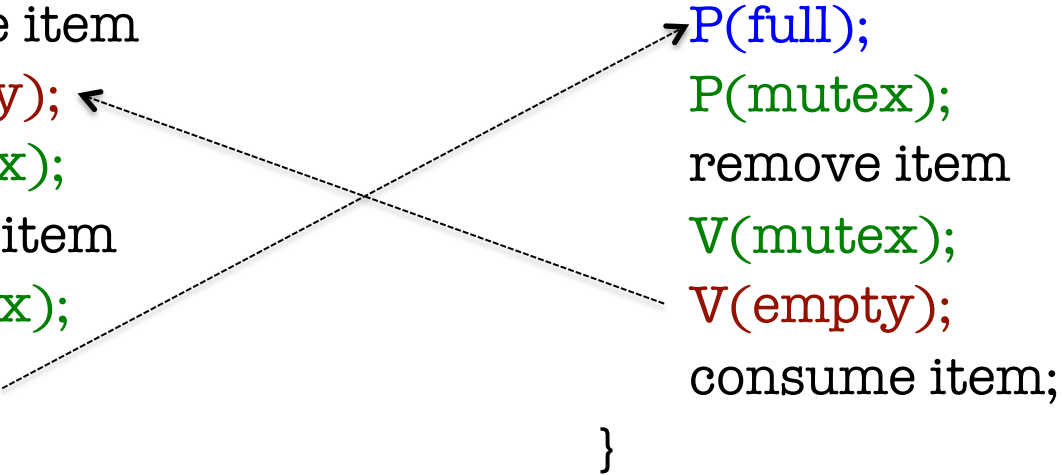
- 3 semafori:
 - mutex, binario inizializzato a TRUE (mutua esclusione per buffer)
 - empty, intero inizializzato a N (blocca P se buffer è pieno)
 - full, intero inizializzato a 0 (blocca C se buffer è vuoto)

PRODUCER

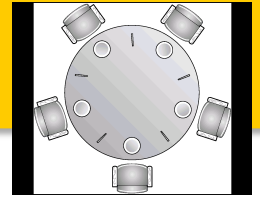
```
while (1) {  
    produce item  
    P(empty);  
    P(mutex);  
    deposit item  
    V(mutex);  
    V(full);  
}
```

CONSUMER

```
while (1) {  
    P(full);  
    P(mutex);  
    remove item  
    V(mutex);  
    V(empty);  
    consume item;  
}
```



Dining philosophers



- N filosofi passano la vita mangiando e pensando
- 1 tavola con N bacchette e una ciotola di riso
- Se un filosofo pensa non interagisce con gli altri
- Se un filosofo ha fame prende 2 bacchette e inizia a mangiare
 - Il filosofo può prendere solo le bacchette che sono alla sua destra e alla sua sinistra
 - Il filosofo può prendere una bacchetta alla volta
 - Se non ci sono 2 bacchette libere il filosofo non può mangiare
- Quando un filosofo termina di mangiare rilascia le bacchette

Dining philosophers – soluzione 1

- Dati condivisi
 - semafori $s[N]$ inizializzati a 1
 - $P(s[j])$ = cerco di prendere la bacchetta j
 - $V(s[j])$ = rilascio la bacchetta j
- Soluzione incompleta
 - Possibile deadlock se tutti i filosofi tentano di prendere la bacchetta alla loro destra (sinistra) contemporaneamente
- Filosofo i (soluzione intuitiva)

```
do {  
    P(s[i])  
    P(s[(i+1) % N])  
    ...  
    // mangia  
    ...  
    V(s[i]);  
    V(s[(i+1) % N]);  
    ...  
    // pensa  
    ...  
} while (1);
```

Problemi con soluzione 1

- Deadlock: ogni filosofo prende la bacchetta di destra, tutti ad aspettare che si liberi quella di sinistra, che nessuno può rilasciare
- Possibili soluzioni (tutte parziali)
 - Permettere solo a 4 filosofi di mangiare contemporaneamente
 - Soluzione asimmetrica:
 - I filosofi in posizioni pari prendono la bacchetta sinistra seguita dalla destra
 - I filosofi in posizioni dispari fanno viceversa
 - Si passano un token
 - Permettere ai filosofi di prendere la forchetta solo se entrambi disponibili



Dining philosophers

- Soluzione corretta
 - Ogni filosofo può essere in tre stati
 - Pensante (THINKING)
 - Affamato (HUNGRY)
 - Mangiante (EATING)

Variabili condivise

- semaphore mutex = 1;
- semaphore f[N]=0;
- int stato[N] = THINKING;

```
Void Philosopher (int i)
{
    while (1) {
        Think();
        Take_fork(i);
        Eat();
        Drop_fork(i);
    }
}
```

```
Void test (int i)
{
    if (stato[i] == HUNGRY &&
        stato[i-1] != EATING &&
        stato[i+1] != EATING)
    {
        stato[i] = EATING;
        V(f[i]);
    }
}
```

```
Void Drop_fork (int i)
{
    P(mutex);
    stato[i] = THINKING;
    test((i-1)%N);
    test((i+1)%N);
    V(mutex);
};
```

I vicini
possono
mangiare

```
Void Take_fork (int i)
{
    P(mutex);
    stato[i] = HUNGRY;
    test(i);
    V(mutex);
    P(f[i]);
}
```



Sleepy barber

- Un negozio ha una sala d'attesa con N sedie, ed una stanza con la sedia del barbiere
- In assenza di clienti, il barbiere si addormenta
- Quando entra un cliente
 - Se le sedie sono occupate, il cliente se ne va
 - Se il barbiere e' occupato il cliente si siede
 - Se il barbiere e' addormentato, il cliente lo sveglia



Sleepy barber

```
Sem intero customers = 0;    // sveglia il barbiere
BinSem barbers = 0;    // stato del barbiere
BinSem mutex = 1;    // protegge la sezione critica
int waiting = 0;    // conta i clienti in attesa
```

BARBER

```
while (1) {
    P(customer);
    P(mutex);
    waiting--;
    V(barbers);
    V(mutex);
    cut hair;
}
```

COSTUMER

```
P(mutex);
if ( waiting < N) {
    waiting++;
    V(customers);    //sveglia!!
    V(mutex);
    P(barbers);    //pronto x taglio
    get haircut;
} else {
    V(mutex);    //non c'e' posto
}
```



Semafori – limitazioni

- L'utilizzo dei semafori presenta alcune difficoltà
 - Difficoltà nella scrittura dei programmi
 - Scarsa “visibilità” della correttezza delle soluzioni
- In alternativa, vengono utilizzati specifici costrutti forniti da linguaggi di programmazione ad alto livello
 - Monitor (Hoare, 1974)
 - Classi synchronized di Java
 - CCR (Conditional Critical Region)
 - ...



Monitor

- Costrutti per la condivisione sicura ed efficiente di dati tra processi
- Simile al concetto di classe

```
monitor xyz{  
    // dichiarazione di variabili (stato del monitor)  
    entry P1 (...) {  
        ...  
    }  
    ....  
    entry Pn (...) {  
        ...  
    }  
    {  
        // codice di inizializzazione  
    }  
}
```

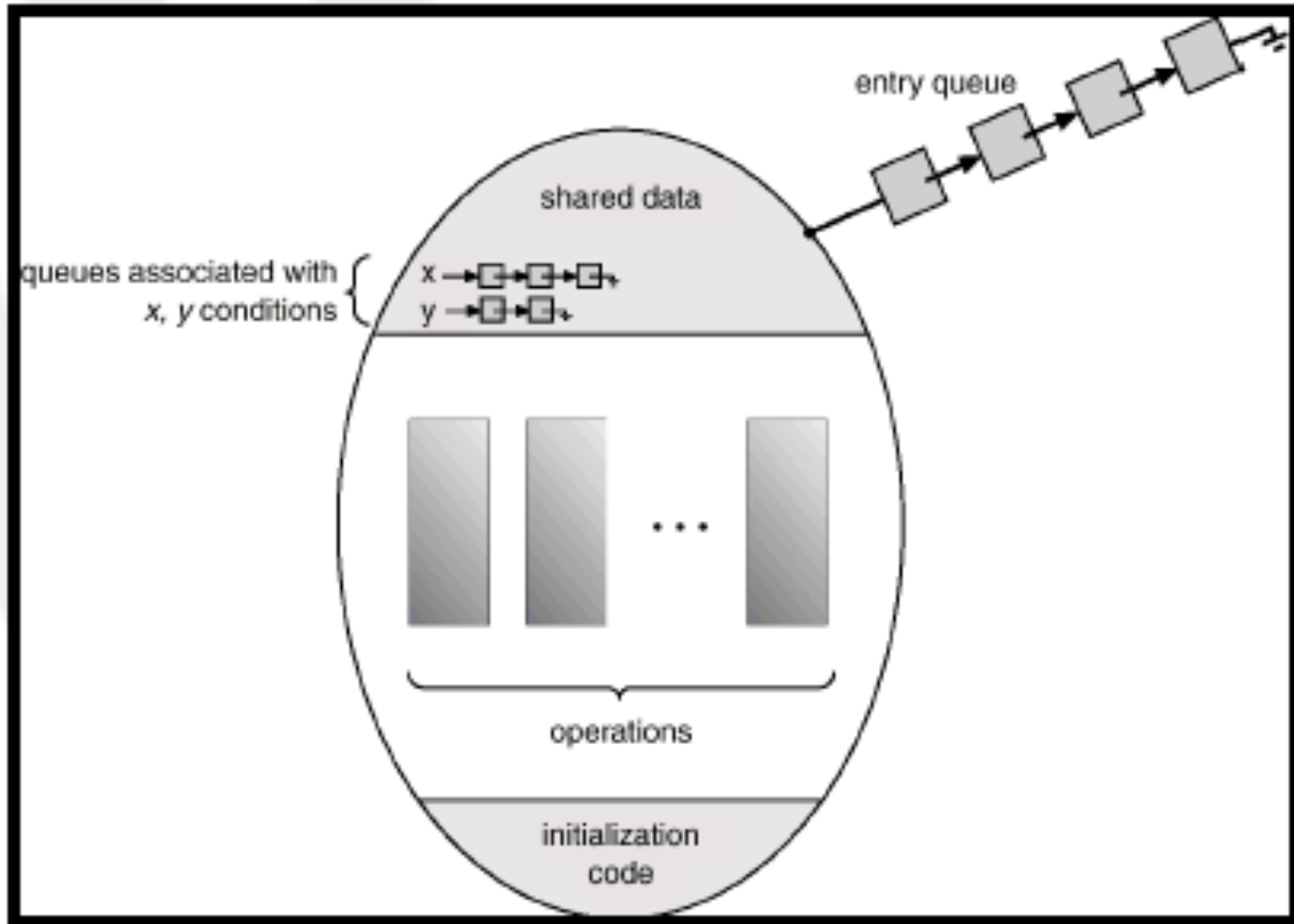


Monitor

- Le variabili del monitor sono visibili solo all'interno del monitor stesso
- Procedure del monitor accedono solo alle variabili definite nel monitor
- Un solo processo alla volta attivo in un monitor
 - Il programmatore non deve codificare esplicitamente la mutua esclusione



Monitor



Monitor

- Per permettere ad un processo di attendere all'interno del monitor, necessari opportuni tipi di sincronizzazione
- Variabili *condition*
 - Dichiarate all'interno del monitor
 - Es: *condition* x, y;
 - Accessibili solo tramite due primitive (tipo semafori)
 - *wait()* [tipo P()]
 - *signal()* [tipo V()]
 - Il processo che invoca *x.wait()* è bloccato fino all'invocazione della corrispondente *x.signal()* da parte di un altro



Monitor

- Comportamento della wait
 - Blocca sempre il processo che la chiama
 - Decidere con altro codice se chiamarla o no (i.e., *if..then..else*)

Monitor

- Comportamento della signal
 - Sveglia esattamente un processo
 - Se più processi in attesa, lo scheduler decide quale processo può entrare
 - Se nessun processo in attesa, nessun effetto
- Comportamento successivamente ad una signal
 - Diverse scelte possibili
 - Processo che invoca *signal* si blocca e l'esecuzione passa all'eventuale processo sbloccato
 - Processo che ha invocato *signal* esce dal monitor (*signal* deve essere ultima istruzione di una procedura)



Monitor – Buffer P/C

```
Producer()
{
    while (TRUE){
        make_item();           // crea nuovo item
        ProducerConsumer.enter(); // chiamata alla funzione enter
    }
}

Consumer()
{
    while (TRUE){
        ProducerConsumer.remove(); // chiamata alla funzione remove
        consume_item();           // consuma item
    }
}
```



Monitor – Buffer P/C

```
monitor ProducerConsumer {  
    condition full, empty;  
    int count;  
  
    entry enter(){  
        if (count == N)  
            full.wait();           //se buffer è pieno, blocca  
  
        put_item();                // mette item nel buffer  
        count = count + 1;         // incrementa count  
  
        if (count == 1)            ← Solo 1 signal  
            empty.signal();        verrà intercettata  
        // se il buffer era vuoto, sveglia il  
        // consumatore  
    }  
}
```

Solo 1 signal
verrà intercettata



Monitor – Buffer P/C

```
entry remove(){
    if (count == 0)
        empty.wait();           // se buffer è vuoto, blocca

    remove_item();              // rimuove item dal buffer
    count = count - 1;          // decrementa count

    if (count == N-1)
        full.signal();          // se il buffer era pieno, sveglia il produttore
    }

    count = 0;                  // inizializzazione di count
end monitor;
}
```

Solo 1 *signal* verrà intercettata



Monitor - esempio

```
monitor BinSem
{
    boolean busy; /* iniz. FALSE */
    condition idle;

    entry void P()
    {
        if (busy) idle.wait();
        busy = TRUE;
    }
    entry void V()
    {
        busy = FALSE;
        idle.signal ();
    }
    busy = FALSE; /* inizializzazione */
}
```



Monitor – limitazioni

- Programmazione con meno errori rispetto ai semafori, ma...
 - Pochi linguaggi forniscono monitor
 - Richiedono presenza memoria condivisa

Sincronizzazione in Java

- Sezione critica
 - keyword `synchronized`
- Metodi `synchronized`
 - Metodo che può essere eseguito da una sola thread alla volta
 - Realizzati mantenendo un singolo lock (detto monitor) per oggetto

Sincronizzazione in Java

- Metodi synchronized static
 - Un lock per classe
- Blocchi synchronized
 - Possibile mettere lock su un qualsiasi oggetto per definire una sezione critica
- Sincronizzazioni aggiuntive
 - wait(), notify(), notifyAll()
 - Ereditati da tutti gli oggetti



Sincronizzazione in Java – Buffer P/C

```
public class BoundedBuffer {  
    Object [] buffer;  
    int nextin;  
    int nextout;  
    int size;  
    int count;  
}  
// costruttore  
public BoundedBuffer (int n){  
    size = n;  
    buffer = new Object[size];  
    nextin = 0;  
    nextout = 0;  
    count = 0;  
}
```

```
public synchronized deposit(Object x) {  
    while (count == size) wait();  
    buffer[nextin] = x;  
    nextin = (nextin+1) mod N;  
    count = count + 1;  
    notifyAll();  
}
```

```
public synchronized Object remove(){  
    Object x;  
    while (count == 0) wait();  
    x = buffer[nextout];  
    nextout = (nextout+1) mod N;  
    count = count - 1;  
    notifyAll();  
    return x;  
}
```



Conclusioni

- Problema della sezione critica come astrazione della concorrenza tra processi
 - Soluzioni con diversi compromessi complessità/difficoltà di utilizzo
- Problemi da gestire
 - Gestione del blocco critico di un insieme di processi (deadlock)
 - Dipendente dalla sequenza temporale degli accessi

