Esercizio 1

Si consideri il monitor seguente che sincronizza le stampe di due processi concorrenti Hip e Urrà.

Siano **nH** e **nU** le variabili che contano rispettivamente il numero di sequenze "hip" e "urrà" stampate in certo stato. Si dimostri che il monitor preserva il seguente invariante:

$$0 \leq nH - 2*nU \leq 2$$

```
monitor Hip_Urra{
int count ← 0;
condition URR;
condition HIP;

 public void StampaHip {
  if (count = 2) wait(HIP);
  <stampa"hip">;
  count++;
  if (count =2) signal(URR);
 }

 public void StampaUrrà {
  if (count < 2) wait(URR);
  <stampa"urrà">;
  count= 0;
  signal(Hip);
 }
}
```

```
process Hip {
  loop forever
    Hip_urra.StampaHip
}


process Urrà {
   loop forever
     Hip_Urra.StampaUrrà
}
```

Soluzione esercizio 1

Per provare che $0 \leq nH - 2*nU \leq 2$
è un invariante di monitor proviamo:

a) $0 \leq count \leq 2$
b) $count = nH - 2*nU$

```
public void StampaHip {
 if (count = 2) wait(HIP);
 <stampa"hip">;
 count++;
 if (count =2) signal(URR);
}

public void StampaUrrà {
 if (count < 2) wait(URR);
 <stampa"urrà">;
 count= 0;
 signal(Hip);
}
```

Soluzione esercizio 1

a) $0 \le$ **count** ≤ 2

La prova è per induzione sul numero di chiamate delle procedure di monitor

All' inizio vero perché `count=0`.

Supponiamo la formula vera prima di una chiamata di una procedura di monitor e dimostriamo che è vera anche dopo l' esecuzione della stessa.

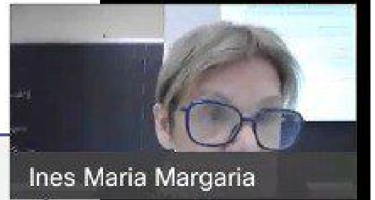
La procedura `StampaHip` potrebbe falsificare la formula perché l' incremento di `count` potrebbe portare il valore della variabile a 3, ma lo statement **`if`** garantisce che l' incremento non viene eseguito se il valore di `count` vale 2.

Se poi la `StampaHip` termina con la **`signal,`** viene svegliato il processo Urrà che porta count a 0 e quindi la formula è vera.

La procedura `StampaUrrà` non può falsificare la formula perché se termina senza svegliare il processo Hip, pone il valore di `count` a 0, se lo sveglia invece il valore di `count` viene portato a 1.

```
public void StampaHip {
 if (count = 2) wait(HIP);
 <stampa"hip">;
 count++;
 if (count =2) signal(URR);
}

public void StampaUrrà {
 if (count < 2) wait(URR);
 <stampa"urrà">;
 count= 0;
 signal(Hip);
}
```



Soluzione esercizio 1

b) **count = nH – 2*nU**

La prova è per induzione sulle chiamate delle procedure di monitor

All' inizio banalmente vero.

Supponiamo la formula vera prima di una chiamata di una procedura di monitor e dimostriamo che è vera anche dopo l' esecuzione della stessa.

La procedura `StampaHip` provoca un incremento di `count` ma anche un incremento di **nH**, quindi se la procedura termina senza svegliare il processo Urrà l' uguaglianza rimane vera.

In caso contrario,il risveglio di Urrà avviene solo in uno stato **s** per cui $nH^s = 2nU^s+2$; la ripresa dell' esecuzione del processo Urrà non provoca cambiamenti su **nH** ma provoca un incremento di 1 su **nU**, quindi nello stato successivo **s'** avremo che $nH^{s'} = nH^s = 2nU^s+2$ e $nU^{s'} = 1+nU^s$. Quindi si avrà che $nH^{s'} = 2(nU^s+1) = 2nU^{s'}$ dunque il valore `count`**=0** rende vera la formula.

La procedura `StampaUrrà` viene eseguita solo quando **count=2** e ci si riporta al caso precedente, se termina con la **signal**, sveglia Hip che porta `count` a 1 ma anche nH viene incrementato di 1.

```
public void StampaHip {
 if (count = 2) wait(HIP);
 <stampa"hip">;
 count++;
 if (count =2) signal(URR);
}

public void StampaUrrà {
 if (count < 2) wait(URR);
 <stampa"urrà">;
 count= 0;
 signal(Hip);
}
```

1:18:15 / 1:27:21

Esercizio 2

Si consideri il codice del pasticcere pigro modellato con i semafori. Si riscriva questo sistema, con le ***stesse caratteristiche*** di sincronizzazione, usando il costrutto monitor
Si provi poi la validità del seguente invariante di monitor:

$$0 \le \mathbf{porzioni} \le \mathbf{m}$$

Algoritmo del pasticcere	
semaphore mutex ← 1, **semaphore** pieno ← 0 **semaphore** vuoto ← 0, **int** porzioni ← 0, **const** m ≥ 1	
pasticcere	**cliente$_i$**
loop forever P1 **P**(vuoto); P2 <prepara m biscotti>; P3 porzioni ← m; P4 **V**(pieno);	<altro> q1 **P**(mutex) q2 **if** (porzioni =0) q3 **V**(vuoto); q4 **P**(pieno); q5 porzioni **--;** q6 **V**(mutex); q7 <mangia biscotto>;

1:18:15 / 1:27:21

Soluzione Esercizio 2 (una possibile)

Premi Esc per uscire dalla modalità a schermo intero


```
process pasticcere {
 loop forever
   bakery.aspetta_cliente
  <prepara i biscotti>
   bakery.porta_biscotti
}


process cliente {
 :
 bakery.entra_negozio;
 bakery.prendi_biscotto;
 bakery.esci_negozio;
 <mangia biscotto>

  :
}
```

1:18:15 / 1:27:21

Soluzione Esercizio 2 (una possibile)

```
monitor bakery {
const       m ≥ 1;
int         porzioni ←  0;
boolean     occupato ← false;
condition coda;
condition cliente;
condition pronto;

public void entra_negozio {
if occupato  wait(coda);
occupato ← true;
}

public void prendi_biscotto() {
if (porzioni = 0)
          signal(cliente)
          wait(pronto);
porzioni--;
}
```

```
public void esci_negozio {
occupato ← false,
signal (coda);
}

public void aspetta_cliente {
if empty(pronto) wait(cliente);
}

public void porta_biscotti() {
porzioni ← m;
signal(pronto);
 }
}
```

Ines Maria Margaria

1:18:16 / 1:27:21

1

Soluzione Esercizio 2 (una possibile)

```
public void prendi_biscotto() {
if (porzioni = 0)
          signal(cliente)
          wait(pronto);
porzioni--;
}


public void porta_biscotti() {
porzioni ← m;
signal(pronto);
 }
```

Proviamo che

0 ≤ porzioni ≤ m

è un invariante di monitor.

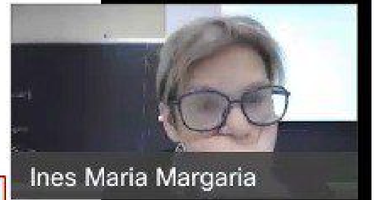
La formula è banalmente vera all'inizio.

Le uniche procedure di monitor che possono modificare la variabile porzioni sono la procedura `prendi_biscotto` e la procedura `porta_biscotti`.

La procedura `prendi_biscotto` potrebbe rendere falsa la formula se il decremento portasse ad un valore negativo, ma questo non può accadere per via dello statement **if**.

L'eventuale risveglio del pasticcere non provoca modifiche sulla variabile `porzioni`.

La procedura `porta_biscotti` forza la variabile al valore di `m`, se la **`signal`** successiva provoca il risveglio del cliente, `porzioni` viene decrementata di 1 e il suo valore risulta `m-1` ma la formula rimane valida.



1:18:16 / 1:27:21

Premi Esc per uscire dalla modalità a schermo intero

Esercizio 3

Si consideri il monitor ForkMonitor per il **problema della cena dei filosofi**.

```
monitor ForkMonitor {
 int array[0..4] fork //initially [2, . . . , 2]//
 condition array[0..4] OKtoEat
 public void takeForks(integer i) {
   if (fork[i] < 2) wait(OKtoEat[i]);
   fork[(i+1)%5] --;
   fork[(i-1)%5] --;
  }
 public void releaseForks(integer i) {
  fork[i+1] ++;
  fork[i-1] ++;
  if (fork[(i+1)%5] = = 2) signal(OKtoEat[(i+1)%5]);
  if (fork[(i-1)%5] = = 2) signal(OKtoEat[(i-1)%5]);
 }
}
```

Esercizio 3 (cont)

Sia *E* la variabile di stato che conta quanti filosofi stanno mangiando, cioè hanno eseguito la procedura takeForks e non hanno ancora invocato la procedura releaseForks.

Si dimostri che il monitor preserva il seguente invariante:

$$\Sigma^{4}_{i=0} \text{fork}[i] = 10 - 2*E$$

Soluzione esercizio 3

```
monitor ForkMonitor {
 int array[0..4] fork //initially
                              [2,.,2]//
 condition array[0..4] OKtoEat
 public void takeForks(integer i) {
   if (fork[i] < 2)
          wait(OKtoEat[i]);
   fork[(i+1)%5] --;
   fork[(i-1)%5] --;
  }
 public void releaseForks(integer i) {
  fork[i+1] ++;
  fork[i-1] ++;
  if (fork[(i+1)%5] = = 2)
          signal(OKtoEat[(i+1)%5]);
  if (fork[(i-1)%5] = = 2)
          signal(OKtoEat[(i-1)%5]);
 }
}
```

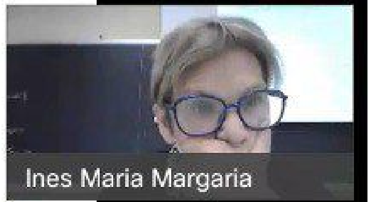
$$\boxed{\Sigma^{4}_{i=0}\ fork[i] = 10 - 2*E}$$

Dimostrazione per induzione sul numero di esecuzioni delle procedure di monitor
All' inizializzazione del monitor l' invariante è vero (E=0).

L'esecuzione di una procedura **`takeForks`** provoca l' incremento del valore di E di un' unità, ma anche il decremento di 2 unità sulla somma delle componenti dell' array **`fork`**, quindi l' invariante resta vero.

Analogamente l' esecuzione della procedura **`releaseForks`** che termina senza risvegli di filosofi in attesa, decrementa di 1 il valore di E ma aumenta di 2 unità la somma delle componenti dell' array **`fork`**, quindi l' invariante continua ad essere vero.

1:18:17 / 1:27:21



Esercizio 4

Lo schema di codice sotto riportato presenta una soluzione tramite monitor di una variante del problema dei cinque filosofi a cena. In questa versione la condizione di accesso alla cena (possesso di entrambe le forchette) è sostituito dalla condizione che due filosofi contigui non possano mangiare contemporaneamente (distanziamento sociale!).

```
monitor  filosofi {
bool eating[0. .4] = [F F F F F]
condition attesa[0. .4]
integer right(i) ← (i+1)%5
integer left(i) ← (i-1)%5

public void iniziocena (i) {
if eating[right(i)].OR. eating[left(i)]
                      wait(attesa[i]);
eating[i] = T
}
```

```
public void finecena(i) {
eating[i] = F
if !empty(attesa[right(i)].AND.
!eating[right(right(i))]
           signal (attesa[right(i)]);
if !empty(attesa[left(i)] .AND.
!eating[left(left(i))]
           signal (attesa[left(i)]);
 }
}
```

Sia $\mathbf{E}_i$ una variabile intera che assume valore 1 se **eating[i]=T** e valore 0 se **eating[i]=F**.

Si provi che valgono i seguenti invarianti:

a) **eating[i]** → **!eating[right(i)]**

b) $0 \le \sum_{i=0}^{4} \mathbf{E}_i \le 2$

1:18:17 / 1:27:21

Esercizio 4 (cont)


```
monitor  filosofi {
bool eating[0. .4] = [F F F F F]
condition attesa[0. .4]
integer right(i) ← (i+1)%5
integer left(i) ← (i-1)%5

public void iniziocena (i) {
if eating[right(i)].OR. eating[left(i)]
                    wait(attesa[i]);
eating[i] = T
}

public void finecena(i) {
eating[i] = F
if !empty(attesa[right(i)].AND.
!eating[right(right(i))]
          signal (attesa[right(i)]);
if !empty(attesa[left(i)] .AND.
!eating[left(left(i))]
         signal (attesa[left(i)]);
 }
}
```

```
Filosofo_i
  loop forever
<pensa>
filosofi.iniziocena(i)
<mangia>
filosofi.finecena(i)
```

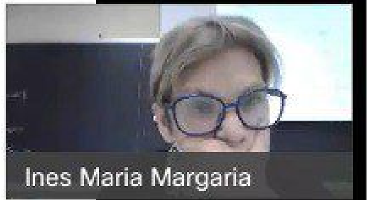
Sia E_i una variabile intera che assume valore 1 se **eating[i]=T** e valore 0 se **eating[i]=F**.

Si provi che valgono i seguenti invarianti:

a) eating[i] →!eating[right(i)]
b) $0 \le \sum_{i=0}^{4} E_i \le 2$

1:18:18 / 1:27:21

Esercizio 4 (cont)




```
monitor  filosofi {
bool eating[0. .4] = [F F F F F]
condition attesa[0. .4]
integer right(i) ← (i+1)%5
integer left(i) ← (i-1)%5

public void iniziocena (i) {
if eating[right(i)].OR. eating[left(i)]
                    wait(attesa[i]);
eating[i] = T
}

public void finecena(i) {
eating[i] = F
if !empty(attesa[right(i)].AND.
!eating[right(right(i))]
          signal (attesa[right(i)]);
if !empty(attesa[left(i)] .AND.
!eating[left(left(i))]
          signal (attesa[left(i)]);
 }
}
```

```
Filosofo_i
  loop forever
<pensa>
filosofi.iniziocena(i)
<mangia>
filosofi.finecena(i)
```

Sia E_i una variabile intera che assume valore 1 se **eating[i]=T** e valore 0 se **eating[i]=F**.

Si provi che valgono i seguenti invarianti:

a) eating[i] →!eating[right(i)]
b) $0 \leq \sum_{i=0}^{4} E_i \leq 2$

1:18:18 / 1:27:21

Soluzione esercizio 4

```
monitor  filosofi {
bool eating[0. .4] = [F F F F F]
condition attesa[0. .4]
integer right(i) ← (i+1)%5
integer left(i) ← (i-1)%5

public void iniziocena (i) {
if eating[right(i)].OR. eating[left(i)]
                 wait(attesa[i]);
eating[i] = T
}

public void finecena(i) {
eating[i] = F
if !empty(attesa[right(i)].AND.
!eating[right(right(i))]
         signal (attesa[right(i)]);
if !empty(attesa[left(i)] .AND.
!eating[left(left(i))]
          signal (attesa[left(i)]);
 }
}
```

a) **eating[i]** $\rightarrow$ **!eating[right(i)]**

b) $0 \leq \Sigma^{4}_{i=0} E_i \leq 2$

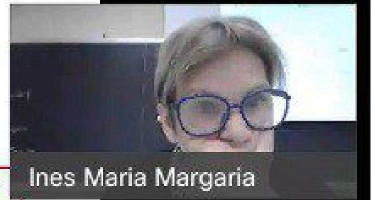
La prova è per induzione sul numero di chiamate delle procedure di monitor

a) **eating[i]** $\rightarrow$ **!eating[right(i)]**

Inizialmente la formula è vera perché l'antecedente è falso per ogni filosofo i.
Supposto vero prima della chiamata di `iniziocena(i)` proviamo che è vero anche alla fine. La procedura pone **eating[i]** a true ma questo avviene solo se **eating[right(i)]** è false quindi l'invariante si mantiene vero.
La procedura `finecena(i)` pone **eating[i]** a false e quindi l'invariante è vero per il filosofo i.
La procedura può risvegliare li processo right[i], ma questo accade solo se **eating[right(right(i)]** è false.
La procedura può anche risvegliare il processo **left[i]**, ma in questo caso il vicino destro è il processo i per cui **eating[i] =F**.

1:18:19 / 1:27:21

Soluzione esercizio 4




```
monitor  filosofi {
bool eating[0. .4] = [F F F F F]
condition attesa[0. .4]
integer right(i) ← (i+1)%5
integer left(i) ← (i-1)%5

public void iniziocena (i) {
if eating[right(i)].OR. eating[left(i)]
                 wait(attesa[i]);
eating[i] = T
}

public void finecena(i) {
eating[i] = F
if !empty(attesa[right(i)].AND.
!eating[right(right(i))]
         signal (attesa[right(i)]);
if !empty(attesa[left(i)] .AND.
!eating[left(left(i))]
         signal (attesa[left(i)]);
 }
}
```

a) eating[i] → !eating[right(i)]
b) $0 \leq \Sigma^4_{i=0}\, E_i \leq 2$

La prova è per induzione sul numero di chiamate delle procedure di monitor

b) $0 \leq \Sigma^4_{i=0}\, E_i \leq 2$

Inizialmente la formula è vera perché $\Sigma^4_{i=0}\, E_i = 0$.

Supposto vero prima della chiamata di `iniziocena(i)` proviamo che è vero anche alla fine. La procedura termina ponendo **eating[i]** a true quindi comporta un incremento di $\Sigma^4_{i=0}\, E_i$ che potrebbe portare il suo valore a 3 se al momento della chiamata il suo valore fosse uguale a 2.

Dal codice sappiamo però che l'incremento avviene solo se i vicini di destra e di sinistra del filosofo i non stanno mangiando, quindi a mangiare dovrebbero essere gli altri due filosofi, che però sono contigui quindi uno dei due avrebbe a destra un vicino che sta mangiando, che è assurdo per il punto a).

1:18:19 / 1:27:21

Soluzione esercizio 4

```
monitor  filosofi {
bool eating[0. .4] = [F F F F F]
condition attesa[0. .4]
integer right(i) ← (i+1)%5
integer left(i) ← (i-1)%5

public void iniziocena (i) {
if eating[right(i)].OR. eating[left(i)]
                 wait(attesa[i]);
eating[i] = T
}

public void finecena(i) {
eating[i] = F
if !empty(attesa[right(i)].AND.
!eating[right(right(i))]
         signal (attesa[right(i)]);
if !empty(attesa[left(i)] .AND.
!eating[left(left(i))]
         signal (attesa[left(i)]);
 }
}
```

a) **eating[i] → !eating[right(i)]**

b) $0 \leq \Sigma^4_{i=0}\ \mathbf{E_i} \leq 2$

b) $0 \leq \Sigma^4_{i=0}\ \mathbf{E_i} \leq 2$

Supponiamo l' invariante vero prima della chiamata di `finecena(i)` proviamo che è vero anche alla fine. La procedura termina ponendo **eating[i]** a false, quindi comporta un decremento di $\Sigma^4_{i=0}\ \mathbf{E_i}$ che non può portare il suo valore a −1 perché al momento della chiamata eating[i]= true e dunque $\Sigma^4_{i=0}\ \mathbf{E_i} >0$. La procedura potrebbe svegliare entrambi i vicini del filosofo i e questo comporterebbe un incremento di 2 del valore di $\Sigma^4_{i=0}\ \mathbf{E_i}$ a fronte di un decremento di 1, per cui $\Sigma^4_{i=0}\ \mathbf{E_i}$ potrebbe assumere il valore 3, se al momento della chiamata avesse avuto valore uguale a 2. Ma anche in questo caso si può provare che se al momento della chiamata un altro filosofo oltre al filosofo i sta mangiando, non è possibile risvegliare entrambi i vicini perché uno di questi avrebbe un vicino di destra che sta mangiando.