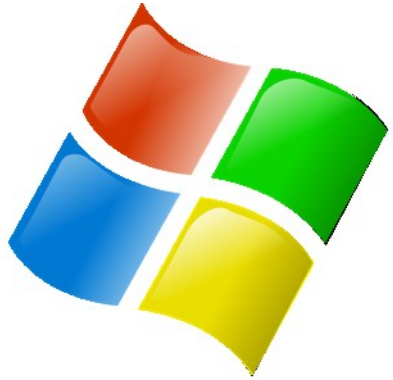
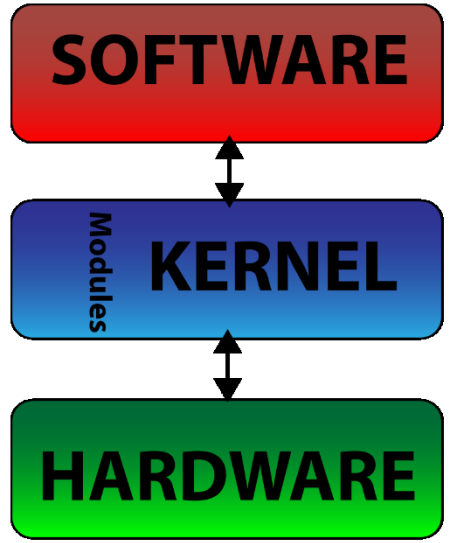
UNIVERSITÀ DEGLI STUDI DELLA BASILICATA



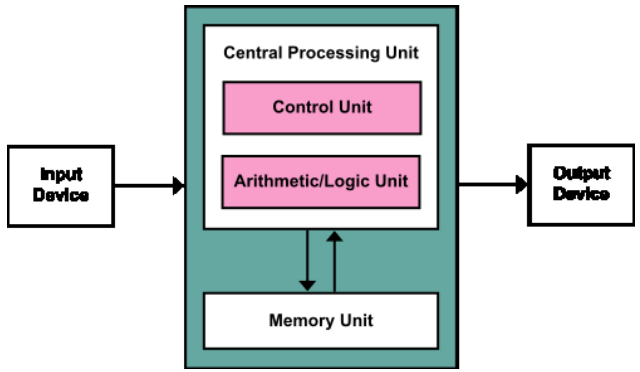
Corso di Sistemi Operativi
A.A. 2019/20

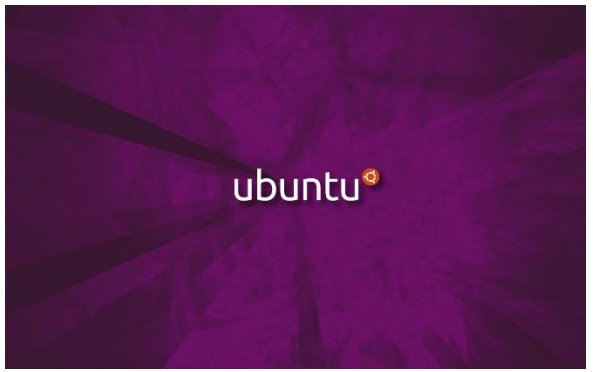
Esercitazione
Sincronizzazione



Docente:
Domenico Daniele Bloisi

Novembre 2019

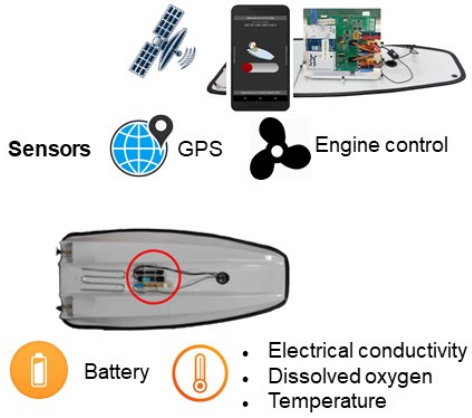


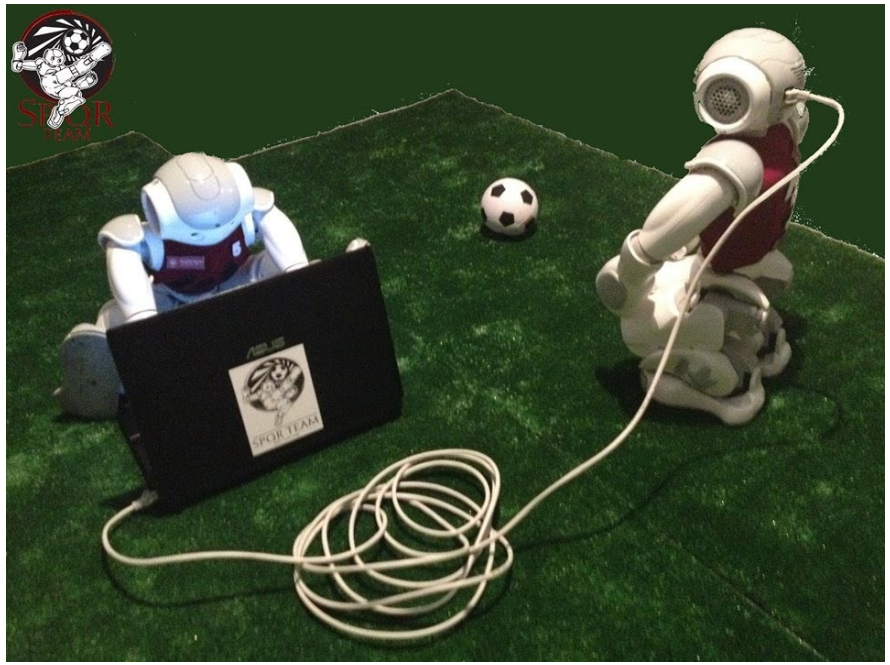


Domenico Daniele Bloisi

- Ricercatore RTD B
  Dipartimento di Matematica, Informatica ed Economia
  Università degli studi della Basilicata
  http://web.unibas.it/bloisi

- SPQR Robot Soccer Team
  Dipartimento di Informatica, Automatica e Gestionale Università degli studi di Roma “La Sapienza”
  http://spqr.diag.uniroma1.it





Ricevimento

- In aula, subito dopo le lezioni
- Martedì dalle 11:00 alle 13:00 presso:
  Campus di Macchia Romana
  Edificio 3D (Dipartimento di Matematica, Informatica ed Economia)
  II piano, stanza 15

Email: domenico.bloisi@unibas.it

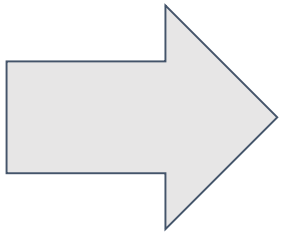
Domanda 1

Che cosa si intende con "sezione critica"?

Risposta alla Domanda 1

Una sezione critica (CS) è una porzione di codice in cui i dati condivisi da processi cooperanti possono essere manipolati

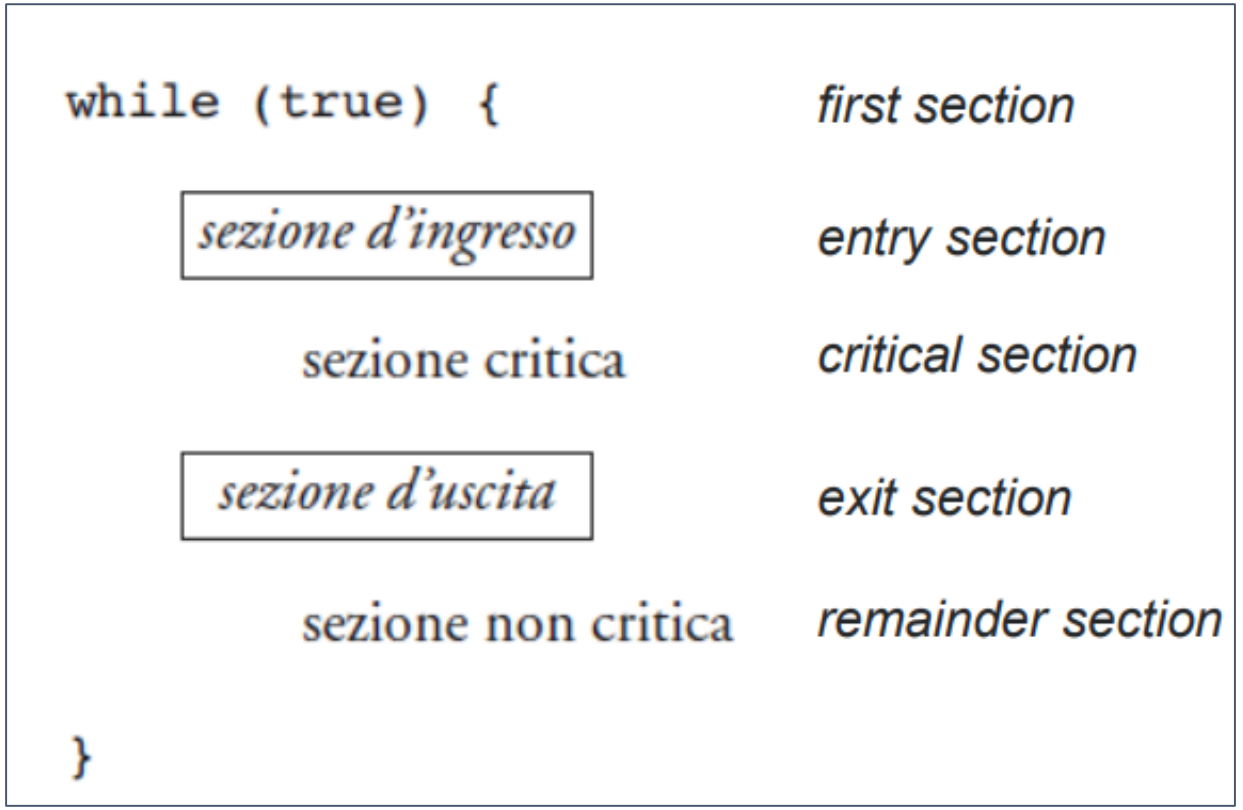
Quando un processo è in esecuzione nella sua sezione critica, nessun altro processo può essere in esecuzione nella propria sezione critica



Risposta alla Domanda 1

La sezione critica di un processo è sempre preceduta da una sezione di ingresso (entry section) e una sezione di uscita (exit section). Si veda lo schema a lato

Schema di codice per un processo contenente una sezione critica



Domanda 2

Perché è importante mantenere la sezione critica il più piccola possibile?

Risposta alla Domanda 2

Una sezione critica può essere eseguita da un unico processo/thread alla volta.

Questo significa che altri processi/thread saranno in attesa fino a che il thread nella sezione critica non avrà terminato la propria esecuzione.

Lunghe sezioni critiche portano ad un calo del throughput del sistema.

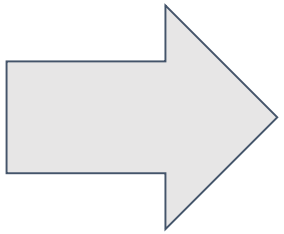
Domanda 3

Quali sono le differenze tra lock mutex e semafori?

Risposta alla Domanda 3

La principale differenza tra semafori e mutex è la seguente:

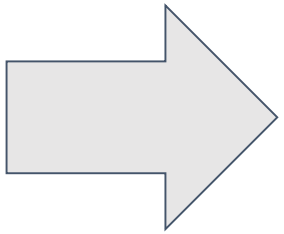
- un semaforo è un meccanismo di segnalazione (tramite le istruzioni wait() e signal() i processi possono indicare se stanno acquisendo o rilasciando la risorsa)
- un mutex è un meccanismo di blocco (un processo deve detenere il mutex prima di poter acquisire la risorsa).



Risposta alla Domanda 3

Quindi:

- un semaforo è un meccanismo di segnalazione tra processi e serve a condividere una risorsa che può essere usata contemporaneamente da un numero limitato di processi
- I mutex hanno lo scopo di proteggere una risorsa condivisa, in modo che più processi non possano accedervi contemporaneamente.



Risposta alla Domanda 3

Inoltre:

- Il valore di un semaforo può essere modificato da qualunque processo che acquisisca o rilasci la risorsa.
- Un mutex può rilasciato solo dal processo che aveva acquisto in precedenza il lock sulla risorsa.

Esercizio 1

- Sia S un semaforo inizializzato a 2
- Si consideri un programma avente la seguente sequenza di istruzioni

```
wait(S)
wait(S)
signal(S)
wait(S)
```

Il programma andrà in blocco?

Semafori

Un **semaforo S** è una variabile intera cui si può accedere, escludendo l'inizializzazione, solo tramite due operazioni atomiche predefinite: wait() e signal().

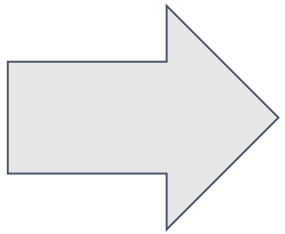
```
wait(S) {
    while(S <= 0)
        ; /* busy wait */
    S--;
}
```

```
signal(S) {
    S++;
}
```

Soluzione Esercizio 1

Analizziamo come la sequenza di istruzioni vada a modificare il valore di S

Inizialmente S vale 2
La prima `wait(S)` decrementa S da 2 a 1
La seconda `wait(S)` decrementa S da 1 a 0
La `signal(S)` incrementa S da 0 a 1
La terza `wait(S)` decrementa S da 1 a 0



Soluzione Esercizio 1

Possiamo concludere sostenendo che il programma non andrà in blocco

Esercizio 2

- Sia S un semaforo inizializzato a 2
- Si consideri un programma avente la seguente sequenza di istruzioni

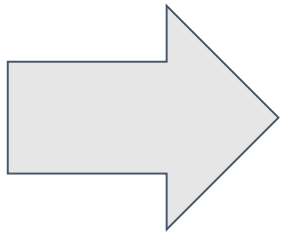
```
wait(S)
wait(S)
signal(S)
wait(S)
wait(S)
```

Il programma andrà in blocco?

Soluzione Esercizio 2

Analizziamo come la sequenza di istruzioni vada a modificare il valore di S

Inizialmente S vale 2
La prima `wait(S)` decrementa S da 2 a 1
La seconda `wait(S)` decrementa S da 1 a 0
La `signal(S)` incrementa S da 0 a 1
La terza `wait(S)` decrementa S da 1 a 0
La quarta `wait(S)` blocca il processo

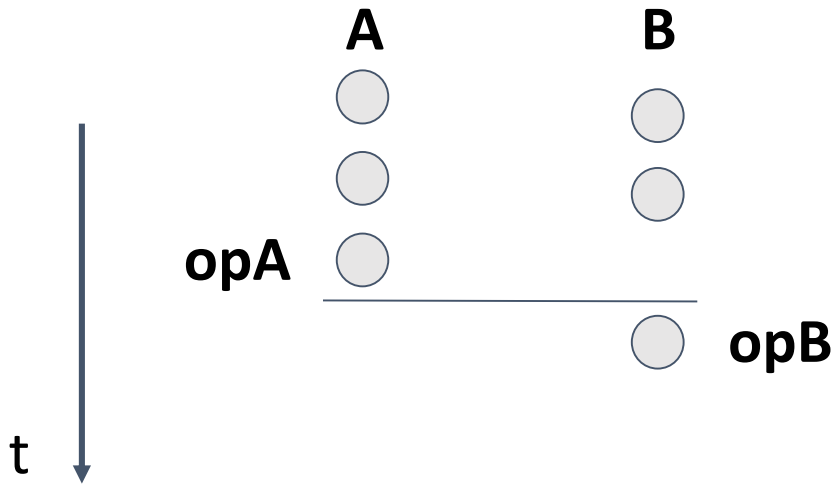


Soluzione Esercizio 2

Possiamo concludere sostenendo che il programma andrà in blocco

Esercizio 3

- Si supponga di avere due thread **A** e **B** in esecuzione concorrente
- Si vuole ottenere che **B** svolga l'operazione **opB** dopo che **A** abbia svolto **opA** (si veda la figura)



Fornire una soluzione al problema usando i semafori

Uso dei semafori

Si considerino i processi ***P1*** e ***P2*** che richiedano di eseguire l'istruzione I_1 prima dell'istruzione I_2

Per assicurare la corretta esecuzione dei due processi in modo concorrente creiamo un semaforo **S** inizializzato a 0 e usiamo signal e wait nel modo seguente

P1:

```
I1;
signal(S);
```

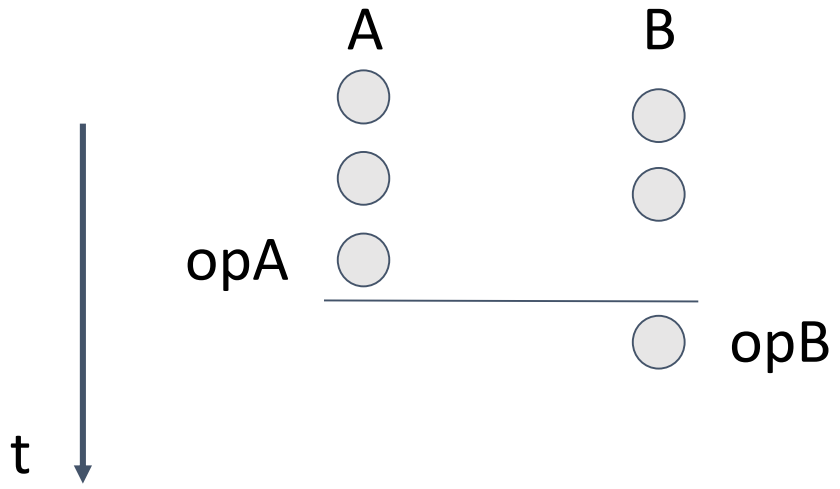
P2:

```
wait(S);
I2;
```

Soluzione Esercizio 3

Inizialmente S vale 0

```
A:                B:
  opA;              wait(S);
  signal(S);        opB;
```



Soluzione Esercizio 3

Verifica della soluzione proposta tramite l'analisi delle possibili sequenze di esecuzione

Inizialmente S vale 0

A:

```
opA;
signal(S);
```

B:

```
wait(S);
opB;
```

A (opA viene eseguita)
B (va in busy waiting perchè S <= 0)
A (esegue signal e S passa da 0 a 1)
B (esce dal ciclo while perchè S > 0, S passa da 1 a 0)
B (esegue opB)

OK

B (va in busy waiting perchè S <= 0)
A (opA viene eseguita)
A (esegue signal e S passa da 0 a 1)
B (esce dal ciclo while perchè S > 0, S passa da 1 a 0)
B (esegue opB)

OK

Abbiamo verificato che la soluzione proposta permette di ottenere sempre prima l'esecuzione di opA e poi l'esecuzione di opB

Esercizio 4

Si verifichi che la soluzione seguente non risolve correttamente il problema dell'Esercizio 3

Inizialmente S vale 1

```
A:                  B:
  wait(S);            wait(S);
  opA;                opB;
  signal(S);          signal(S);
```

Soluzione Esercizio 4

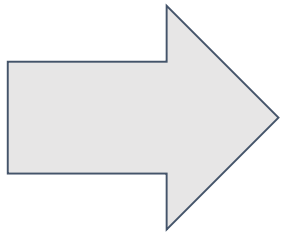
Verifica della soluzione proposta tramite l'analisi delle possibili sequenze di esecuzione

A (esegue wait e S passa da 1 a 0)
B (va in busy waiting perchè S <= 0)
A (esegue opA)
A (esegue signal e S passa da 0 a 1)
B (esce dal ciclo while perchè S > 0, S passa da 1 a 0)
B (esegue opB)
B (esegue signal e S passa da 0 a 1)

Inizialmente S vale 1

```
A:                B:
 wait(S);          wait(S);
 opA;              opB;
 signal(S);        signal(S);
```

OK



Soluzione Esercizio 4

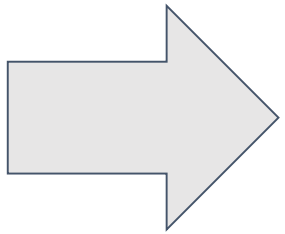
Altra possibile esecuzione

B (esegue wait e S passa da 1 a 0)
A (va in busy waiting perchè S <= 0)
B (esegue opB)
B (esegue signal e S passa da 0 a 1)
A (esce dal ciclo while perchè S > 0, S passa da 1 a 0)
A (esegue opA)
A (esegue signal e S passa da 0 a 1)

Inizialmente S vale 1

```
A:                  B:
 wait(S);            wait(S);
 opA;                opB;
 signal(S);          signal(S);
```

NO

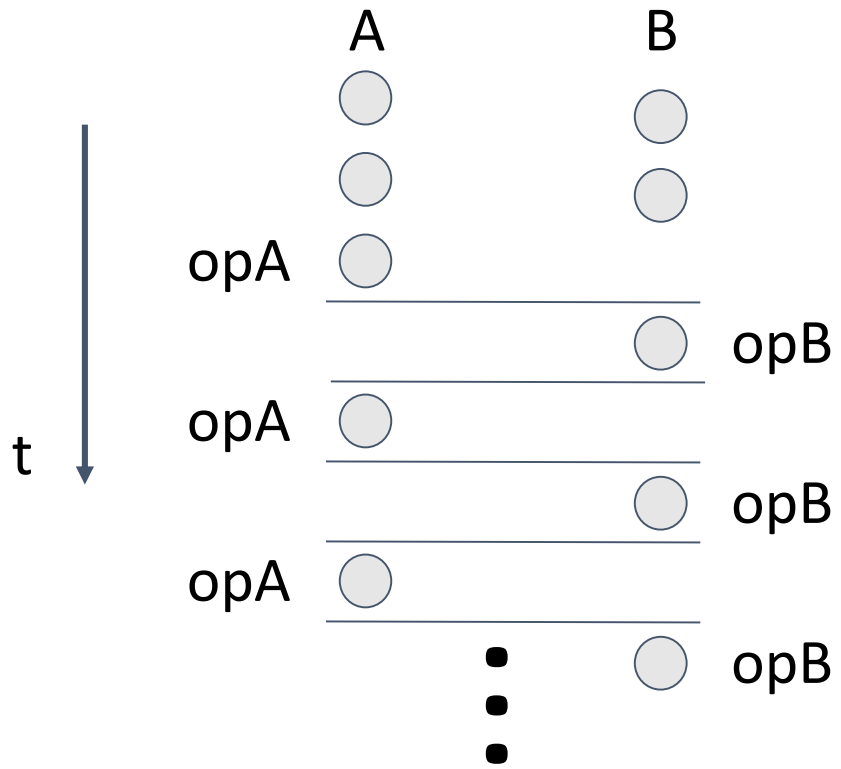


Soluzione Esercizio 4

Possiamo concludere sostenendo che la soluzione proposta non risolve correttamente il problema dell’Esercizio 3

Esercizio 5

- Si supponga di avere due thread A e B in esecuzione concorrente
- Si supponga inoltre che A e B debbano svolgere le operazioni opA e opB a turno, con A ad iniziare prima (si veda la figura)



Esercizio 5

Dimostrare che la soluzione seguente è corretta

Inizialmente S_1 vale 0 e S_2 vale 0

A:

```
while(TRUE){
   opA;
   signal(S1);
   wait(S2);
}
```

B:

```
while(TRUE){
   wait(S1);
   opB;
   signal(S2);
}
```

Soluzione Esercizio 5

Inizialmente S_1 vale 0 e S_2 vale 0

```
A:                      B:
while(TRUE){            while(TRUE){
 opA;                    wait(S1);
 signal(S1);             opB;
 wait(S2);               signal(S2);
}                       }
```

Verifica della soluzione proposta tramite l'analisi delle possibili sequenze di esecuzione

A (esegue opA)
B (va in busy waiting perchè S_1 <= 0)

B (va in busy waiting perchè S_1 <= 0)
A (esegue opA)

A (esegue signal e S_1 passa da 0 a 1)

A (va in busy waiting perchè S_2 <= 0)
B (esce dal ciclo while perchè S_1 > 0, S_1 passa da 1 a 0)
B (esegue opB)
B (esegue signal e S_2 passa da 0 a 1)
A (esce dal ciclo while perchè S_2 > 0, S_2 passa da 1 a 0)

B (esce dal ciclo while perchè S_1 > 0, S_1 passa da 1 a 0)

OK

Soluzione Esercizio 5

Inizialmente S_1 vale 0 e S_2 vale 0

```
A:                      B:
while(TRUE){            while(TRUE){
 opA;                    wait(S1);
 signal(S1);             opB;
 wait(S2);               signal(S2);
}                       }
```

B (esce dal ciclo while perchè $S_1 > 0$, S_1 passa da 1 a 0)

A (va in busy waiting perchè $S_2 <= 0$)
B (esegue opB)
B (esegue signal e S_2 passa da 0 a 1)
A (esce dal ciclo while perchè $S_2 > 0$,
S_2 passa da 1 a 0)

OK

B (esegue opB)

A (va in busy waiting perchè $S_2 <= 0$)
B (esegue signal e S_2 passa da 0 a 1)
A (esce dal ciclo while perchè $S_2 > 0$,
S_2 passa da 1 a 0)

OK

B (esegue signal e S_2 passa da 0 a 1)
A (esegue wait e S_2 passa da 1 a 0)

OK

Esercizio 6

Date le stesse assunzioni dell'esercizio 5, la soluzione seguente è corretta?

Inizialmente S_1 vale 0 e S_2 vale 1

A:

```
while(TRUE){
   wait(S2);
   opA;
   signal(S1);
}
```

B:

```
while(TRUE){
   wait(S1);
   opB;
   signal(S2);
}
```

Esercizio 7

Cosa stampa il seguente programma?

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

pthread_mutex_t mutex;

int SHARED_DATA = 0;

void *f1(void *arg)
{
  pthread_mutex_lock(&mutex);
  SHARED_DATA = SHARED_DATA + 2;
  SHARED_DATA = SHARED_DATA * 2;
  pthread_mutex_unlock(&mutex);
  pthread_exit(0);
}
```

```
void *f2(void *arg)
{
  pthread_mutex_lock(&mutex);
  SHARED_DATA = SHARED_DATA + 3;
  SHARED_DATA = SHARED_DATA * 3;
  pthread_mutex_unlock(&mutex);
  pthread_exit(0);
}
```

```
int main()
{
  pthread_t thread1, thread2;

  pthread_mutex_init(&mutex, NULL);

  if(pthread_create(&thread1, NULL, f1, NULL) < 0)
  {
     printf("errore creazione thread 1\n");
     exit(1);
  }

  if(pthread_create(&thread2, NULL, f2, NULL) < 0)
  {
     printf("errore creazione thread 2\n");
     exit(1);
  }

  pthread_join (thread1, NULL);
  pthread_join (thread2, NULL);
  printf("SHARED_DATA: %d\n", SHARED_DATA);
}
```

Esercizio 7

```
bloisi@bloisi-U36SG: ~/workspace/mutex
bloisi@bloisi-U36SG:~/workspace/mutex$ ./mutex-pthreads
SHARED_DATA: 21
bloisi@bloisi-U36SG:~/workspace/mutex$ ./mutex-pthreads
SHARED_DATA: 21
bloisi@bloisi-U36SG:~/workspace/mutex$ ./mutex-pthreads
SHARED_DATA: 21
bloisi@bloisi-U36SG:~/workspace/mutex$ ./mutex-pthreads
SHARED_DATA: 21
bloisi@bloisi-U36SG:~/workspace/mutex$ ./mutex-pthreads
SHARED_DATA: 21
bloisi@bloisi-U36SG:~/workspace/mutex$ ./mutex-pthreads
SHARED_DATA: 21
bloisi@bloisi-U36SG:~/workspace/mutex$ ./mutex-pthreads
SHARED_DATA: 21
bloisi@bloisi-U36SG:~/workspace/mutex$ ./mutex-pthreads
SHARED_DATA: 22
bloisi@bloisi-U36SG:~/workspace/mutex$ ./mutex-pthreads
SHARED_DATA: 21
bloisi@bloisi-U36SG:~/workspace/mutex$ ./mutex-pthreads
SHARED_DATA: 21
bloisi@bloisi-U36SG:~/workspace/mutex$ ./mutex-pthreads
SHARED_DATA: 21
bloisi@bloisi-U36SG:~/workspace/mutex$ ./mutex-pthreads
SHARED_DATA: 21
bloisi@bloisi-U36SG:~/workspace/mutex$ ./mutex-pthreads
SHARED_DATA: 22
bloisi@bloisi-U36SG:~/workspace/mutex$ ./mutex-pthreads
SHARED_DATA: 22
bloisi@bloisi-U36SG:~/workspace/mutex$ ./mutex-pthreads
SHARED_DATA: 21
bloisi@bloisi-U36SG:~/workspace/mutex$
```

Esercizio 8

Cosa stampa il seguente programma?

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

pthread_mutex_t mutex;

int SHARED_DATA = 0;

void *f1(void *arg)
{
 sleep(5);
 pthread_mutex_lock(&mutex);
 SHARED_DATA = SHARED_DATA + 2;
 SHARED_DATA = SHARED_DATA * 2;
 pthread_mutex_unlock(&mutex);
 pthread_exit(0);
}
```

```
void *f2(void *arg)
{
 pthread_mutex_lock(&mutex);
 SHARED_DATA = SHARED_DATA + 3;
 SHARED_DATA = SHARED_DATA * 3;
 pthread_mutex_unlock(&mutex);
 pthread_exit(0);
}
```

```
int main()
{
 pthread_t thread1, thread2;

 pthread_mutex_init(&mutex, NULL);

 if(pthread_create(&thread1, NULL, f1, NULL) < 0)
 {
    printf("errore creazione thread 1\n");
    exit(1);
 }

 if(pthread_create(&thread2, NULL, f2, NULL) < 0)
 {
    printf("errore creazione thread 2\n");
    exit(1);
 }

 pthread_join (thread1, NULL);
 pthread_join (thread2, NULL);
 printf("SHARED_DATA: %d\n", SHARED_DATA);
}
```

Esercizio 8

```
bloisi@bloisi-U36SG: ~/workspace/mutex
bloisi@bloisi-U36SG:~/workspace/mutex$ ./mutex-pthreads-sleep
SHARED_DATA: 22
bloisi@bloisi-U36SG:~/workspace/mutex$ ./mutex-pthreads-sleep
SHARED_DATA: 22
bloisi@bloisi-U36SG:~/workspace/mutex$ ./mutex-pthreads-sleep
SHARED_DATA: 22
bloisi@bloisi-U36SG:~/workspace/mutex$ ./mutex-pthreads-sleep
SHARED_DATA: 22
bloisi@bloisi-U36SG:~/workspace/mutex$ ./mutex-pthreads-sleep
SHARED_DATA: 22
bloisi@bloisi-U36SG:~/workspace/mutex$ ./mutex-pthreads-sleep
SHARED_DATA: 22
bloisi@bloisi-U36SG:~/workspace/mutex$ ./mutex-pthreads-sleep
SHARED_DATA: 22
bloisi@bloisi-U36SG:~/workspace/mutex$ ./mutex-pthreads-sleep
SHARED_DATA: 22
bloisi@bloisi-U36SG:~/workspace/mutex$ ./mutex-pthreads-sleep
SHARED_DATA: 22
bloisi@bloisi-U36SG:~/workspace/mutex$ ./mutex-pthreads-sleep
SHARED_DATA: 22
bloisi@bloisi-U36SG:~/workspace/mutex$ ./mutex-pthreads-sleep
SHARED_DATA: 22
bloisi@bloisi-U36SG:~/workspace/mutex$ ./mutex-pthreads-sleep
SHARED_DATA: 22
bloisi@bloisi-U36SG:~/workspace/mutex$ ./mutex-pthreads-sleep
SHARED_DATA: 22
bloisi@bloisi-U36SG:~/workspace/mutex$ ./mutex-pthreads-sleep
SHARED_DATA: 22
bloisi@bloisi-U36SG:~/workspace/mutex$ ./mutex-pthreads-sleep
SHARED_DATA: 22
bloisi@bloisi-U36SG:~/workspace/mutex$
```

Esercizio 9

Cosa stampa il seguente programma?

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int SHARED_DATA = 0;

void *f1(void *arg)
{
  SHARED_DATA = SHARED_DATA + 2;
  SHARED_DATA = SHARED_DATA * 2;
  pthread_exit(0);
}
```

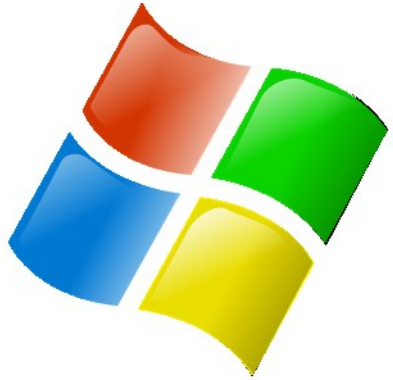
```
void *f2(void *arg)
{
  SHARED_DATA = SHARED_DATA + 3;
  SHARED_DATA = SHARED_DATA * 3;
  pthread_exit(0);
}
```

```
int main()
{
  pthread_t thread1, thread2;

  if(pthread_create(&thread1, NULL, f1, NULL) < 0)
  {
     printf("errore creazione thread 1\n");
     exit(1);
  }

  if(pthread_create(&thread2, NULL, f2, NULL) < 0)
  {
     printf("errore creazione thread 2\n");
     exit(1);
  }

  pthread_join (thread1, NULL);
  pthread_join (thread2, NULL);
  printf("SHARED_DATA: %d\n", SHARED_DATA);
}
```

Esercizio 9

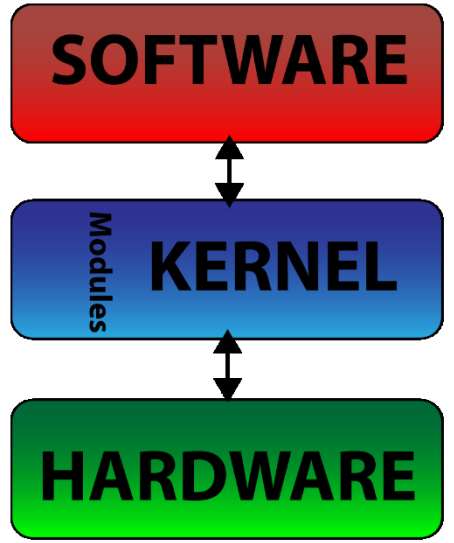
```
bloisi@bloisi-U36SG: ~/workspace/mutex
bloisi@bloisi-U36SG:~/workspace/mutex$ ./pthreads
SHARED_DATA: 21
bloisi@bloisi-U36SG:~/workspace/mutex$ ./pthreads
SHARED_DATA: 21
bloisi@bloisi-U36SG:~/workspace/mutex$ ./pthreads
SHARED_DATA: 4
bloisi@bloisi-U36SG:~/workspace/mutex$ ./pthreads
SHARED_DATA: 21
bloisi@bloisi-U36SG:~/workspace/mutex$ ./pthreads
SHARED_DATA: 21
bloisi@bloisi-U36SG:~/workspace/mutex$ ./pthreads
SHARED_DATA: 21
bloisi@bloisi-U36SG:~/workspace/mutex$ ./pthreads
SHARED_DATA: 21
bloisi@bloisi-U36SG:~/workspace/mutex$ ./pthreads
SHARED_DATA: 21
bloisi@bloisi-U36SG:~/workspace/mutex$ ./pthreads
SHARED_DATA: 21
bloisi@bloisi-U36SG:~/workspace/mutex$ ./pthreads
SHARED_DATA: 21
bloisi@bloisi-U36SG:~/workspace/mutex$ ./pthreads
SHARED_DATA: 21
bloisi@bloisi-U36SG:~/workspace/mutex$ ./pthreads
SHARED_DATA: 21
bloisi@bloisi-U36SG:~/workspace/mutex$ ./pthreads
SHARED_DATA: 21
bloisi@bloisi-U36SG:~/workspace/mutex$ ./pthreads
SHARED_DATA: 21
bloisi@bloisi-U36SG:~/workspace/mutex$ ./pthreads
SHARED_DATA: 9
bloisi@bloisi-U36SG:~/workspace/mutex$ ./pthreads
SHARED_DATA: 21
bloisi@bloisi-U36SG:~/workspace/mutex$
```

UNIVERSITÀ DEGLI STUDI DELLA BASILICATA



Corso di Sistemi Operativi
A.A. 2019/20

Esercitazione
Sincronizzazione



Docente:
Domenico Daniele Bloisi

Novembre 2019