POLITECNICO DI MILANO

Esercizi di mutua esclusione e sincronizzazione

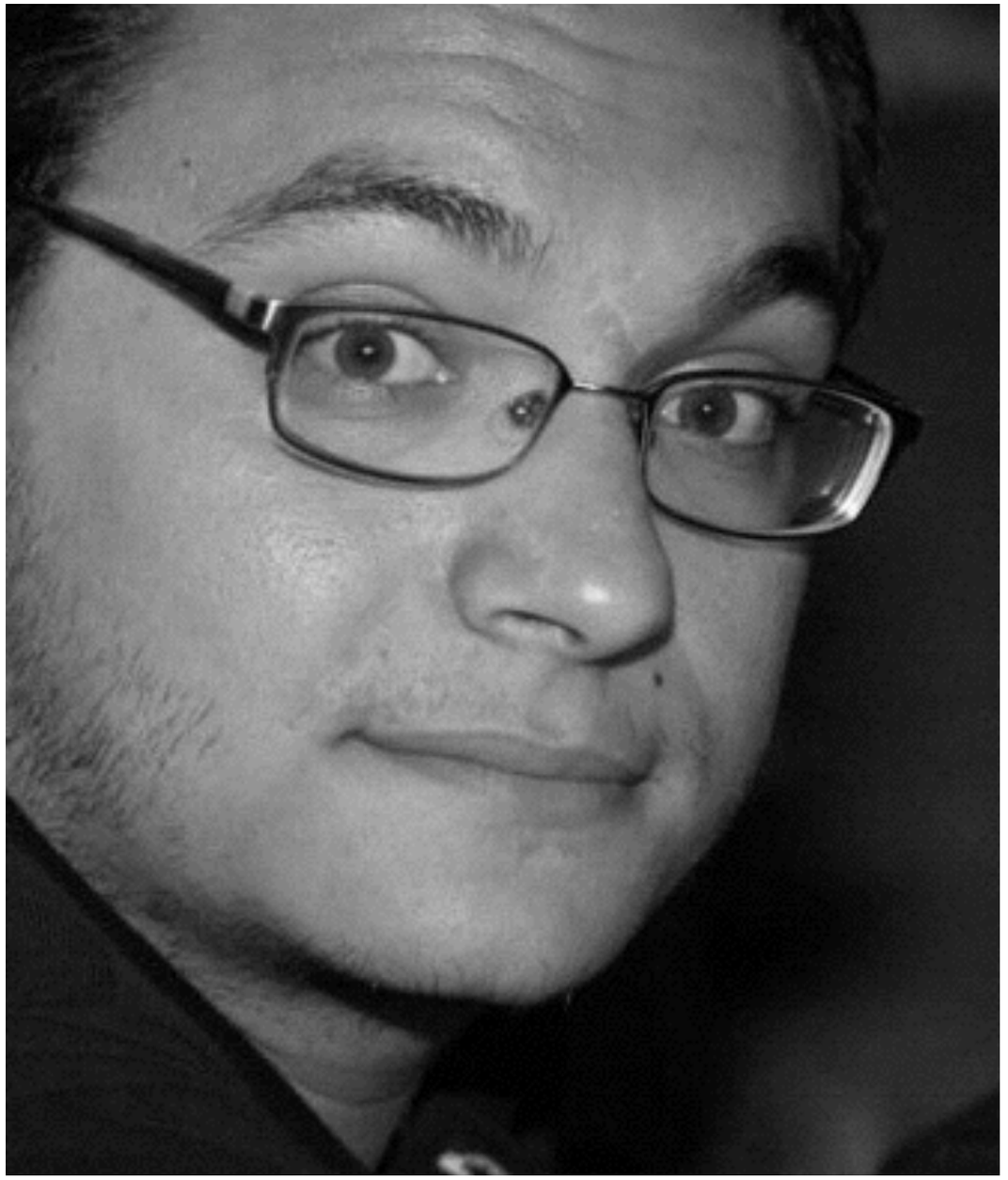
Alessandro A. Nacci
alessandro.nacci@polimi.it

ACSO

2014/2014

Chi sono

- Dottorando in Ingegneria Informatica (3° anno)
- Il vostro esercitatore di ACSO per quest'anno
- Lavoro al NECSTLab & JOL
- Contattatemi per email
  - alessandro.nacci@polimi.it
- Il materiale del corso sarà disponibile su
  - www.alessandronacci.it

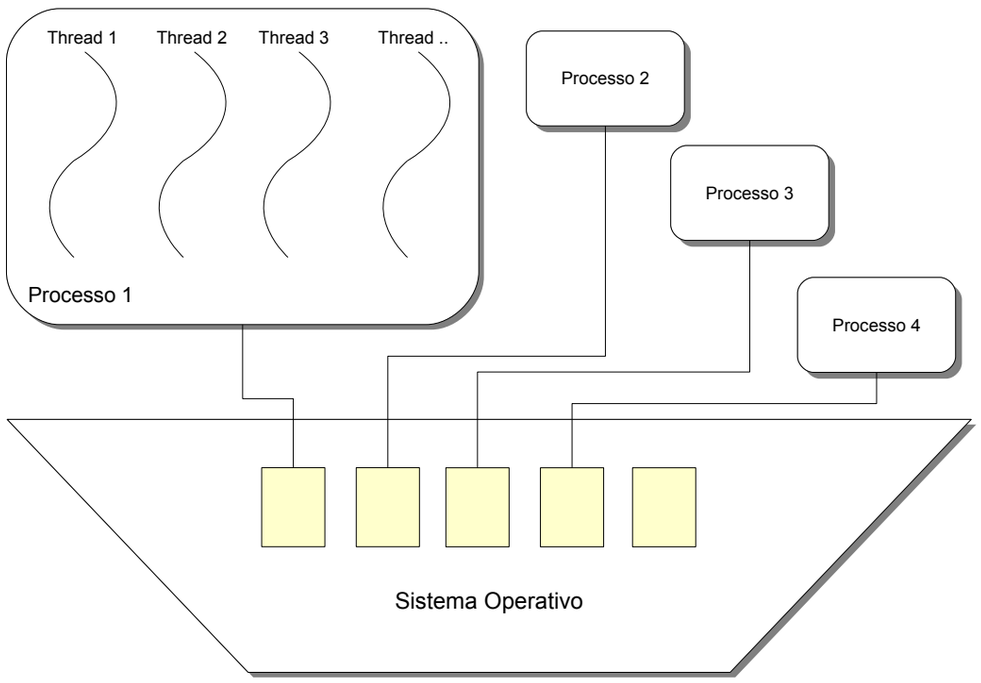



START

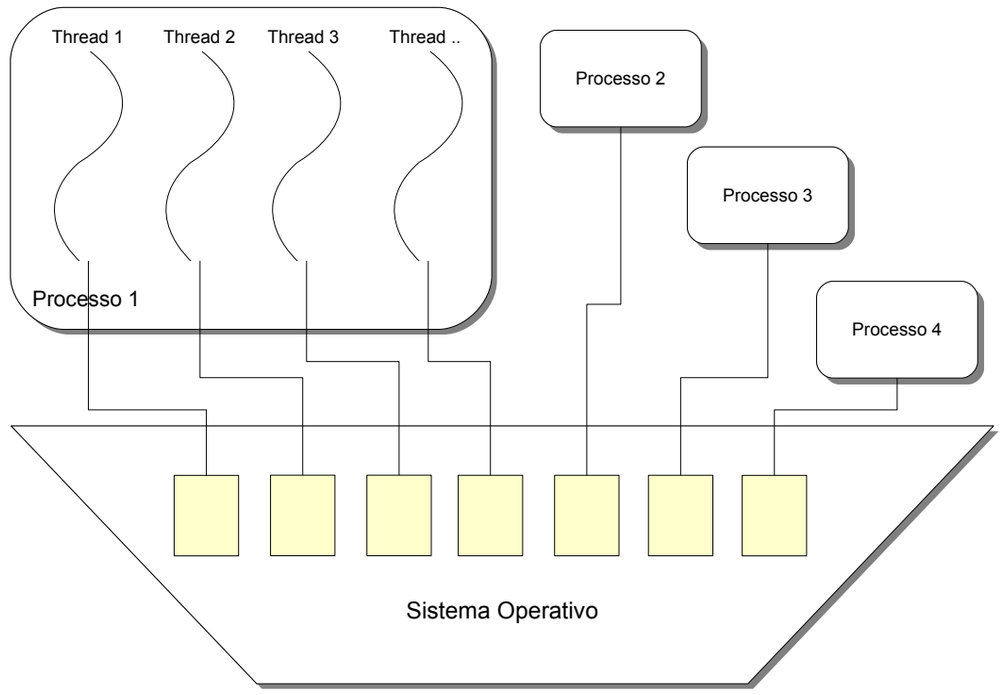
Thread: cosa sono

- Nell'ambito di un processo è possibile attivare diversi flussi di controllo, detti thread, che procedono in parallelo tra loro
- I thread di uno stesso processo condividono lo spazio di indirizzamento e altre risorse.
- Normalmente la creazione e gestione di un thread da parte del Sistema Operativo è meno costosa, in termini di risorse quali il tempo macchina, della creazione e gestione di un processo.

Thread e SO



(a) User-Level



(b) Kernel-Level

User Level: il sistema operativo ha conoscenza solamente dell'esistenza dei processi; i thread esistono all'interno del processo e sono ad esso strettamente collegati: se uno di questi thread si blocca (per esempio in attesa di una operazione di I/O) tutto il processo viene bloccato, inclusi gli altri thread che lo compongono. Siccome il processo viene visto dal sistema operativo come una unità atomica e indivisibile, non è possibile, nel caso di sistemi multi-processore, eseguire in contemporanea su processori diversi thread appartenenti allo stesso processo. La gestione di tali thread ha un costo inferiore rispetto ai thread kernel-level in quanto non passa attraverso il sistema operativo.

Kernel Level: il sistema operativo ha diretta conoscenza dell'esistenza dei thread, che possono essere, di conseguenza, gestiti separatamente dallo stesso: (a) un thread si può porre in attesa di un evento (e.g. I/O) e gli altri thread del processo continuano la loro esecuzione, (b) in un sistema multi-processore (come quasi tutti i normali PC al giorno d'oggi) thread di uno stesso processo possono essere mandati in esecuzione in contemporanea, (c) etc. La gestione di questi thread richiede il passaggio attraverso il sistema operativo quindi, come vedrete in seguito nel corso, ha associato un costo non indifferente in termini di prestazioni.

Thread: usiamoli

PThread: Principali Funzioni

- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);`
- `int pthread_join(pthread_t thread, void **retval);`

Esercizio 1

Scrivere un programma che faccia partire tre thread: il secondo e il terzo possono partire solamente dopo la terminazione del primo. Tutti e tre i thread stampano a schermo il loro identificatore usando la funzione `pthread_t pthread_self(void)`.

Esercizio 1

Scrivere un programma che faccia partire tre thread: il secondo e il terzo possono partire solamente dopo la terminazione del primo. Tutti e tre i thread stampano a schermo il loro identificatore usando la funzione `pthread_t pthread_self(void)`.

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

void * th_fun(void *arg){
  printf("%d\n", pthread_self());
}
int main(int argc, char * argv[]){
    pthread_t th1, th2, th3;

    pthread_create(&th1, NULL, th_fun, NULL);
    pthread_join(th1, NULL);
    pthread_create(&th2, NULL, th_fun, NULL);
    pthread_create(&th3, NULL, th_fun, NULL);

    return 0;
}
```

Esercizio 1

Scrivere un programma che faccia partire tre thread: il secondo e il terzo possono partire solamente dopo la terminazione del primo. Tutti e tre i thread stampano a schermo il loro identificatore usando la funzione `pthread_t pthread_self(void)`.

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

void * th_fun(void *arg){
   printf("%d\n", pthread_self());
}
int main(int argc, char * argv[]){
     pthread_t th1, th2, th3;

     pthread_create(&th1, NULL, th_fun, NULL);
     pthread_join(th1, NULL);
     pthread_create(&th2, NULL, th_fun, NULL);
     pthread_create(&th3, NULL, th_fun, NULL);

     return 0;
}
```

Soluzione errata: il flusso di esecuzione principale (main) potrebbe terminare prima che siano terminati i thread 2 e 3, impedendo quindi una loro corretta esecuzione.

Esercizio 1
Soluzione

Scrivere un programma che faccia partire tre thread: il secondo e il terzo possono partire solamente dopo la terminazione del primo. Tutti e tre i thread stampano a schermo il loro identificatore usando la funzione `pthread_t pthread_self(void)`.

```
void * th_fun(void *arg){
  printf("%d\n", pthread_self());
}
int main(int argc, char * argv[]){
    pthread_t th1, th2, th3;

    pthread_create(&th1, NULL, th_fun, NULL);
    pthread_join(th1, NULL);
    pthread_create(&th2, NULL, th_fun, NULL);
    pthread_create(&th3, NULL, th_fun, NULL);
    pthread_join(th2, NULL);
    pthread_join(th3, NULL);

    return 0;
}
```

Soluzione corretta:è necessario usare la funzione join anche per i thread 2 e 3, in modo tale da assicurazione che il il flusso di esecuzione principale (main) non termini prima dei thread da lui creati.

Esercizio 2

Scrivere un programma che crei due thread per accellerare il calcolo del fattoriale del numero specificato alla riga di comando del programma stesso.

Come possiamo accelerare il calcolo del fattoriale?

Esercizio 2

Scrivere un programma che crei due thread per accellerare il calcolo del fattoriale del numero specificato alla riga di comando del programma stesso.

Come possiamo accelerare il calcolo del fattoriale?

$$n! = n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot \ldots \cdot 1$$

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \quad \ldots \quad \cdot (\frac{n}{2}) \cdot (\frac{n}{2}-1) \cdot (\frac{n}{2}-2) \cdot \quad \ldots \quad \cdot 1$$

Fino a n/2 — **Da n/2 a 1**

Possono andare in parallelo

Esercizio 2
Fattoriale parziale

- Vogliamo un funzione che
  - dato in ingresso un numero di partenza n_start
  - dato in ingresso un numero di arrivo n_end
  - restituisca il parziale del fattoriale tra n_start ed n_end

```
int calcolaPazialeFattoriale(int start, int end)
{









}
```

Esercizio 2
Fattoriale parziale

- Vogliamo un funzione che
  - dato in ingresso un numero di partenza n_start
  - dato in ingresso un numero di arrivo n_end
  - restituisca il parziale del fattoriale tra n_start ed n_end

```
int calcolaPazialeFattoriale(int start, int end)
{
     int fattoriale = 1;
     int i;

     for (i = start; i <= end; i++)
     {
          fattoriale += i;
     }

     return fattoriale;
}
```

Esercizio 2

Ma con i thread…

- La funzione implementata non può essere associata al alcun thread poiché non ne rispetta l'interfaccia

```
void * fact(void *arg){




}
```

Esercizio 2

Ma con i thread…

- La funzione implementata non può essere associata al alcun thread poiché non ne rispetta l'interfaccia

```
typedef struct num_{
    int start, end;
} num;


void * fact(void *arg){




}
```

Esercizio 2

Ma con i thread…

- La funzione implementata non può essere associata al alcun thread poiché non ne rispetta l'interfaccia

```
typedef struct num_{
    int start, end;
} num;


void * fact(void *arg){
    int fattoriale = 1;
    int i = 0;
    for(i = ((num *)arg)->start; i <= ((num *)arg)->end; i++){
        fattoriale *= i;
    }
    return (void *)fattoriale;
}
```

Esercizio 2

Pensiamo al main()

```
int main(int argc, char * argv[]){

}
```

Esercizio 2

Pensiamo al main()

```
int main(int argc, char * argv[]){
    pthread_t th1, th2;
    long fattoriale1 = 0, fattoriale2 = 0;
    num th_arg1, th_arg2;
    th_arg1.start = 1;
    th_arg1.end = atoi(argv[1])/2;
    th_arg2.start = atoi(argv[1])/2 + 1;
    th_arg2.end = atoi(argv[1]);

}
```

argv[1] è il primo valore passato come parametro all'esecuzione del programma

./mioProgramma 4
argv[1] = 4

Esercizio 2

Pensiamo al main()

```
int main(int argc, char * argv[]){
    pthread_t th1, th2;
    long fattoriale1 = 0, fattoriale2 = 0;
    num th_arg1, th_arg2;
    th_arg1.start = 1;
    th_arg1.end = atoi(argv[1])/2;
    th_arg2.start = atoi(argv[1])/2 + 1;
    th_arg2.end = atoi(argv[1]);

    pthread_create(&th1, NULL, fact, (void *)&th_arg1);
    pthread_create(&th2, NULL, fact, (void *)&th_arg2);



}
```

argv[1] è il primo valore passato come parametro all'esecuzione del programma

./mioProgramma 4
argv[1] = 4

Esercizio 2

Pensiamo al main()

```
int main(int argc, char * argv[]){
    pthread_t th1, th2;
    long fattoriale1 = 0, fattoriale2 = 0;
    num th_arg1, th_arg2;
    th_arg1.start = 1;
    th_arg1.end = atoi(argv[1])/2;
    th_arg2.start = atoi(argv[1])/2 + 1;
    th_arg2.end = atoi(argv[1]);

    pthread_create(&th1, NULL, fact, (void *)&th_arg1);
    pthread_create(&th2, NULL, fact, (void *)&th_arg2);

    pthread_join(th1, (void **)&fattoriale1);
    pthread_join(th2, (void **)&fattoriale2);



}
```

argv[1] è il primo valore passato come parametro all'esecuzione del programma

./mioProgramma 4
argv[1] = 4

Esercizio 2

Pensiamo al main()

```
int main(int argc, char * argv[]){
    pthread_t th1, th2;
    long fattoriale1 = 0, fattoriale2 = 0;
    num th_arg1, th_arg2;
    th_arg1.start = 1;
    th_arg1.end = atoi(argv[1])/2;
    th_arg2.start = atoi(argv[1])/2 + 1;
    th_arg2.end = atoi(argv[1]);

    pthread_create(&th1, NULL, fact, (void *)&th_arg1);
    pthread_create(&th2, NULL, fact, (void *)&th_arg2);

    pthread_join(th1, (void **)&fattoriale1);
    pthread_join(th2, (void **)&fattoriale2);

    printf("Il fattoriale del numero %d e' %ld\n", atoi(argv[1])
        , fattoriale1*fattoriale2);

    return 0;
}
```

argv[1] è il primo valore passato come parametro all'esecuzione del programma

./mioProgramma 4
argv[1] = 4

Esercizio 2
Considerazioni

- Notate come sia fondamentale dichiarare fattoriale1 e fattoriale2 come *long* e non come *int*

- Questo garantisce correttezza di funzionamento anche su sistemi a 64bit, dove i puntatori puntano a celle di memoria grandi 64bit

Thread e *scope* delle variabili

Lo scoping delle variabili all'interno dei thread funziona in modo analogo a quanto succede per le normali funzioni, con l'unica differenza che esistono più pile, una per ogni thread; per quanto riguarda lo heap e le variabili globali, invece è presenta una sola area che li contiene, condivisa tra tutti i thread del processo.

REGOLE IMPORTANTI

Stesso heap tra thread differenti

Stesse variabili globali tra thread differenti

Stack differenti tra thread differenti

Esercizio 3

Descrizione output programma

```
int valore = 0;

void * somma(void * arg){
    int i = 0;
    for(i = 0; i < (int)arg; i++){
        valore++;
    }
    printf("thread: valore = %d\n", valore);
    return NULL;
}

int main(int argc, char * argv[]){
    pthread_t th1, th2, th3;

    pthread_create(&th1, NULL, somma, (void *)3);
    pthread_join(th1, NULL);

    pthread_create(&th2, NULL, somma, (void *)4);
    pthread_join(th2, NULL);

    pthread_create(&th3, NULL, somma, (void *)5);
    pthread_join(th3, NULL);

    return 0;
}
```

Esercizio 3

Descrizione output programma

```c
int valore = 0;

void * somma(void * arg){
    int i = 0;
    for(i = 0; i < (int)arg; i++){
        valore++;
    }
    printf("thread: valore = %d\n", valore);
    return NULL;
}

int main(int argc, char * argv[]){
    pthread_t th1, th2, th3;

    pthread_create(&th1, NULL, somma, (void *)3);
    pthread_join(th1, NULL);

    pthread_create(&th2, NULL, somma, (void *)4);
    pthread_join(th2, NULL);

    pthread_create(&th3, NULL, somma, (void *)5);
    pthread_join(th3, NULL);

    return 0;
}
```

```
> thread: valore = 3
```

Esercizio 3

Descrizione output programma

```c
int valore = 0;

void * somma(void * arg){
    int i = 0;
    for(i = 0; i < (int)arg; i++){
        valore++;
    }
    printf("thread: valore = %d\n", valore);
    return NULL;
}

int main(int argc, char * argv[]){
    pthread_t th1, th2, th3;

    pthread_create(&th1, NULL, somma, (void *)3);
    pthread_join(th1, NULL);

    pthread_create(&th2, NULL, somma, (void *)4);
    pthread_join(th2, NULL);

    pthread_create(&th3, NULL, somma, (void *)5);
    pthread_join(th3, NULL);

    return 0;
}
```

```
> thread: valore = 3
```

```
> thread: valore = 7
```

Esercizio 3

Descrizione output programma

```c
int valore = 0;

void * somma(void * arg){
    int i = 0;
    for(i = 0; i < (int)arg; i++){
        valore++;
    }
    printf("thread: valore = %d\n", valore);
    return NULL;
}

int main(int argc, char * argv[]){
    pthread_t th1, th2, th3;

    pthread_create(&th1, NULL, somma, (void *)3);
    pthread_join(th1, NULL);

    pthread_create(&th2, NULL, somma, (void *)4);
    pthread_join(th2, NULL);

    pthread_create(&th3, NULL, somma, (void *)5);
    pthread_join(th3, NULL);

    return 0;
}
```

```
> thread: valore = 3
```

```
> thread: valore = 7
```

```
> thread: valore = 12
```

Esercizio 3

Descrizione output programma

```c
int valore = 0;

void * somma(void * arg){
    int i = 0;
    for(i = 0; i < (int)arg; i++){
        valore++;
    }
    printf("thread: valore = %d\n", valore);
    return NULL;
}

int main(int argc, char * argv[]){
    pthread_t th1, th2, th3;

    pthread_create(&th1, NULL, somma, (void *)3);
    pthread_join(th1, NULL);

    pthread_create(&th2, NULL, somma, (void *)4);
    pthread_join(th2, NULL);

    pthread_create(&th3, NULL, somma, (void *)5);
    pthread_join(th3, NULL);

    return 0;
}
```

```
> thread: valore = 3
```

```
> thread: valore = 7
```

```
> thread: valore = 12
```

Esercizio 4

Descrizione output programma

```
void * somma(void * arg){
    int valore = 0;
    int i = 0;
    for(i = 0; i < (int)arg; i++){
        valore++;
    }
    printf("thread: valore = %d\n", valore);
    return NULL;
}

int main(int argc, char * argv[]){
    pthread_t th1, th2, th3;

    pthread_create(&th1, NULL, somma, (void *)3);
    pthread_join(th1, NULL);

    pthread_create(&th2, NULL, somma, (void *)4);
    pthread_join(th2, NULL);

    pthread_create(&th3, NULL, somma, (void *)5);
    pthread_join(th3, NULL);

    return 0;
}
```

Esercizio 4

Descrizione output programma

```c
void * somma(void * arg){
    int valore = 0;
    int i = 0;
    for(i = 0; i < (int)arg; i++){
        valore++;
    }
    printf("thread: valore = %d\n", valore);
    return NULL;
}

int main(int argc, char * argv[]){
    pthread_t th1, th2, th3;

    pthread_create(&th1, NULL, somma, (void *)3);
    pthread_join(th1, NULL);

    pthread_create(&th2, NULL, somma, (void *)4);
    pthread_join(th2, NULL);

    pthread_create(&th3, NULL, somma, (void *)5);
    pthread_join(th3, NULL);

    return 0;
}
```

```
> thread: valore = 3
```

Esercizio 4

Descrizione output programma

```c
void * somma(void * arg){
    int valore = 0;
    int i = 0;
    for(i = 0; i < (int)arg; i++){
        valore++;
    }
    printf("thread: valore = %d\n", valore);
    return NULL;
}

int main(int argc, char * argv[]){
    pthread_t th1, th2, th3;

    pthread_create(&th1, NULL, somma, (void *)3);
    pthread_join(th1, NULL);

    pthread_create(&th2, NULL, somma, (void *)4);
    pthread_join(th2, NULL);

    pthread_create(&th3, NULL, somma, (void *)5);
    pthread_join(th3, NULL);

    return 0;
}
```

```
> thread: valore = 3
```

```
> thread: valore = 4
```

Esercizio 4

Descrizione output programma

```c
void * somma(void * arg){
    int valore = 0;
    int i = 0;
    for(i = 0; i < (int)arg; i++){
        valore++;
    }
    printf("thread: valore = %d\n", valore);
    return NULL;
}

int main(int argc, char * argv[]){
    pthread_t th1, th2, th3;

    pthread_create(&th1, NULL, somma, (void *)3);
    pthread_join(th1, NULL);

    pthread_create(&th2, NULL, somma, (void *)4);
    pthread_join(th2, NULL);

    pthread_create(&th3, NULL, somma, (void *)5);
    pthread_join(th3, NULL);

    return 0;
}
```

```
> thread: valore = 3
```

```
> thread: valore = 4
```

```
> thread: valore = 5
```

Esercizio 4

Descrizione output programma

```c
void * somma(void * arg){
    int valore = 0;
    int i = 0;
    for(i = 0; i < (int)arg; i++){
        valore++;
    }
    printf("thread: valore = %d\n", valore);
    return NULL;
}

int main(int argc, char * argv[]){
    pthread_t th1, th2, th3;

    pthread_create(&th1, NULL, somma, (void *)3);
    pthread_join(th1, NULL);

    pthread_create(&th2, NULL, somma, (void *)4);
    pthread_join(th2, NULL);

    pthread_create(&th3, NULL, somma, (void *)5);
    pthread_join(th3, NULL);

    return 0;
}
```

```
> thread: valore = 3
```

```
> thread: valore = 4
```

```
> thread: valore = 5
```

Esercizio 4

Descrizione output programma

```c
void * somma(void * arg){
    int valore = 0;
    int i = 0;
    for(i = 0; i < (int)arg; i++){
        valore++;
    }
    printf("thread: valore = %d\n", valore);
    return NULL;
}

int main(int argc, char * argv[]){
    pthread_t th1, th2, th3;

    pthread_create(&th1, NULL,
    pthread_join(th1, NULL);

    pthread_create(&th2, NULL,
    pthread_join(th2, NULL);

    pthread_create(&th3, NULL,
    pthread_join(th3, NULL);

    return 0;
}
```

SERVONO LE JOIN IN QUESTO CASO?

Esercizio 4

Descrizione output programma

```
int main(int argc, char * argv[]){
    pthread_t th1, th2, th3;

    pthread_create(&th1, NULL, somma, (void *)3);
    pthread_create(&th2, NULL, somma, (void *)4);
    pthread_create(&th3, NULL, somma, (void *)5);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    pthread_join(th3, NULL);

    return 0;
}
```

Esercizio 5

Valore delle variabili


```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <pthread.h>
4
5  int c = 0;
6
7  void * th_fun(void *arg){
8      int a = 0, b = (int)arg;
9      a = b + c;
10     c = a*a;
11     return (void *)a;
12 }
13 int main(int argc, char * argv[]){
14     pthread_t th1, th2, th3;
15     long d;
16
17     pthread_create(&th1, NULL, th_fun, (void *)3);
18     pthread_join(th1, (void **)&d);
19     pthread_create(&th2, NULL, th_fun, (void *)d);
20     pthread_join(th2, (void **)&d);
21     pthread_create(&th3, NULL, th_fun, (void *)d);
22     pthread_join(th3, (void **)&d);
23
24     return 0;
25 }
```

Dopo Linea	Valore delle Variabili				
	a	b	c	d	arg
15					
10 (dopo instr. 17)					
18					
10 (dopo instr. 19)					
20					
10 (dopo instr. 21)					
22					

Esercizio 5

Valore delle variabili


```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <pthread.h>
4
5  int c = 0;
6
7  void * th_fun(void *arg){
8      int a = 0, b = (int)arg;
9      a = b + c;
10     c = a*a;
11     return (void *)a;
12 }
13 int main(int argc, char * argv[]){
14     pthread_t th1, th2, th3;
15     long d;
16
17     pthread_create(&th1, NULL, th_fun, (void *)3);
18     pthread_join(th1, (void **)&d);
19     pthread_create(&th2, NULL, th_fun, (void *)d);
20     pthread_join(th2, (void **)&d);
21     pthread_create(&th3, NULL, th_fun, (void *)d);
22     pthread_join(th3, (void **)&d);
23
24     return 0;
25 }
```

Dopo Linea	Valore delle Variabili				
	a	**b**	**c**	**d**	**arg**
15	unk	unk	0	unk	unk
10 (dopo instr. 17)					
18					
10 (dopo instr. 19)					
20					
10 (dopo instr. 21)					
22					

Esercizio 5

Valore delle variabili


```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <pthread.h>
4
5  int c = 0;
6
7  void * th_fun(void *arg){
8      int a = 0, b = (int)arg;
9      a = b + c;
10     c = a*a;
11     return (void *)a;
12 }
13 int main(int argc, char * argv[]){
14     pthread_t th1, th2, th3;
15     long d;
16
17     pthread_create(&th1, NULL, th_fun, (void *)3);
18     pthread_join(th1, (void **)&d);
19     pthread_create(&th2, NULL, th_fun, (void *)d);
20     pthread_join(th2, (void **)&d);
21     pthread_create(&th3, NULL, th_fun, (void *)d);
22     pthread_join(th3, (void **)&d);
23
24     return 0;
25 }
```

Dopo Linea	Valore delle Variabili				
	a	**b**	**c**	**d**	**arg**
15	unk	unk	0	unk	unk
10 (dopo instr. 17)	3	3	9	unk	3
18					
10 (dopo instr. 19)					
20					
10 (dopo instr. 21)					
22					

Esercizio 5

Valore delle variabili


```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <pthread.h>
4
5  int c = 0;
6
7  void * th_fun(void *arg){
8      int a = 0, b = (int)arg;
9      a = b + c;
10     c = a*a;
11     return (void *)a;
12 }
13 int main(int argc, char * argv[]){
14     pthread_t th1, th2, th3;
15     long d;
16
17     pthread_create(&th1, NULL, th_fun, (void *)3);
18     pthread_join(th1, (void **)&d);
19     pthread_create(&th2, NULL, th_fun, (void *)d);
20     pthread_join(th2, (void **)&d);
21     pthread_create(&th3, NULL, th_fun, (void *)d);
22     pthread_join(th3, (void **)&d);
23
24     return 0;
25 }
```

Dopo Linea	Valore delle Variabili				
	a	**b**	**c**	**d**	**arg**
15	unk	unk	0	unk	unk
10 (dopo instr. 17)	3	3	9	unk	3
18	unk	unk	9	3	unk
10 (dopo instr. 19)					
20					
10 (dopo instr. 21)					
22					

Esercizio 5

Valore delle variabili


```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <pthread.h>
4
5  int c = 0;
6
7  void * th_fun(void *arg){
8      int a = 0, b = (int)arg;
9      a = b + c;
10     c = a*a;
11     return (void *)a;
12 }
13 int main(int argc, char * argv[]){
14     pthread_t th1, th2, th3;
15     long d;
16
17     pthread_create(&th1, NULL, th_fun, (void *)3);
18     pthread_join(th1, (void **)&d);
19     pthread_create(&th2, NULL, th_fun, (void *)d);
20     pthread_join(th2, (void **)&d);
21     pthread_create(&th3, NULL, th_fun, (void *)d);
22     pthread_join(th3, (void **)&d);
23
24     return 0;
25 }
```

Dopo Linea	Valore delle Variabili				
	a	**b**	**c**	**d**	**arg**
15	unk	unk	0	unk	unk
10 (dopo instr. 17)	3	3	9	unk	3
18	unk	unk	9	3	unk
10 (dopo instr. 19)	12	3	144	3	3
20					
10 (dopo instr. 21)					
22					

Esercizio 5

Valore delle variabili


```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <pthread.h>
4
5  int c = 0;
6
7  void * th_fun(void *arg){
8      int a = 0, b = (int)arg;
9      a = b + c;
10     c = a*a;
11     return (void *)a;
12 }
13 int main(int argc, char * argv[]){
14     pthread_t th1, th2, th3;
15     long d;
16
17     pthread_create(&th1, NULL, th_fun, (void *)3);
18     pthread_join(th1, (void **)&d);
19     pthread_create(&th2, NULL, th_fun, (void *)d);
20     pthread_join(th2, (void **)&d);
21     pthread_create(&th3, NULL, th_fun, (void *)d);
22     pthread_join(th3, (void **)&d);
23
24     return 0;
25 }
```

Dopo Linea	Valore delle Variabili				
	a	**b**	**c**	**d**	**arg**
15	unk	unk	0	unk	unk
10 (dopo instr. 17)	3	3	9	unk	3
18	unk	unk	9	3	unk
10 (dopo instr. 19)	12	3	144	3	3
20	unk	unk	144	12	unk
10 (dopo instr. 21)					
22					

Esercizio 5

Valore delle variabili


```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <pthread.h>
4
5  int c = 0;
6
7  void * th_fun(void *arg){
8      int a = 0, b = (int)arg;
9      a = b + c;
10     c = a*a;
11     return (void *)a;
12 }
13 int main(int argc, char * argv[]){
14     pthread_t th1, th2, th3;
15     long d;
16
17     pthread_create(&th1, NULL, th_fun, (void *)3);
18     pthread_join(th1, (void **)&d);
19     pthread_create(&th2, NULL, th_fun, (void *)d);
20     pthread_join(th2, (void **)&d);
21     pthread_create(&th3, NULL, th_fun, (void *)d);
22     pthread_join(th3, (void **)&d);
23
24     return 0;
25 }
```

Dopo Linea	Valore delle Variabili				
	a	**b**	**c**	**d**	**arg**
15	unk	unk	0	unk	unk
10 (dopo instr. 17)	3	3	9	unk	3
18	unk	unk	9	3	unk
10 (dopo instr. 19)	12	3	144	3	3
20	unk	unk	144	12	unk
10 (dopo instr. 21)	156	12	156*156	12	12
22					

Esercizio 5

Valore delle variabili


```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <pthread.h>
4
5  int c = 0;
6
7  void * th_fun(void *arg){
8      int a = 0, b = (int)arg;
9      a = b + c;
10     c = a*a;
11     return (void *)a;
12 }
13 int main(int argc, char * argv[]){
14     pthread_t th1, th2, th3;
15     long d;
16
17     pthread_create(&th1, NULL, th_fun, (void *)3);
18     pthread_join(th1, (void **)&d);
19     pthread_create(&th2, NULL, th_fun, (void *)d);
20     pthread_join(th2, (void **)&d);
21     pthread_create(&th3, NULL, th_fun, (void *)d);
22     pthread_join(th3, (void **)&d);
23
24     return 0;
25 }
```

Dopo Linea	Valore delle Variabili				
	a	**b**	**c**	**d**	**arg**
15	unk	unk	0	unk	unk
10 (dopo instr. 17)	3	3	9	unk	3
18	unk	unk	9	3	unk
10 (dopo instr. 19)	12	3	144	3	3
20	unk	unk	144	12	unk
10 (dopo instr. 21)	156	12	156*156	12	12
22	unk	unk	156*156	156	unk

Esercizio 6

Valore delle variabili

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <pthread.h>
4
5  int c = 0;
6
7  void * th_fun(void *arg){
8     int a = 0, b = (int)arg;
9     a = b + c;
10    return (void *)a;
11 }
12 int main(int argc, char * argv[]){
13       pthread_t th1, th2, th3;
14       long d;
15
16       pthread_create(&th1, NULL, th_fun, (void *)3);
17       pthread_create(&th2, NULL, th_fun, (void *)4);
18       pthread_create(&th3, NULL, th_fun, (void *)5);
19
20       pthread_join(th1, (void **)&d);
21       pthread_join(th2, (void **)&d);
22       pthread_join(th3, (void **)&d);
23
24       return 0;
25 }
```

Dopo Linea	Valore delle Variabili				
	a	b	c	d	arg
14					
9 (in thread 1)					
9 (in thread 2)					
9 (in thread 3)					
20					
21					
22					

Esercizio 6

Valore delle variabili

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <pthread.h>
4
5  int c = 0;
6
7  void * th_fun(void *arg){
8     int a = 0, b = (int)arg;
9     a = b + c;
10    return (void *)a;
11 }
12 int main(int argc, char * argv[]){
13       pthread_t th1, th2, th3;
14       long d;
15
16       pthread_create(&th1, NULL, th_fun, (void *)3);
17       pthread_create(&th2, NULL, th_fun, (void *)4);
18       pthread_create(&th3, NULL, th_fun, (void *)5);
19
20       pthread_join(th1, (void **)&d);
21       pthread_join(th2, (void **)&d);
22       pthread_join(th3, (void **)&d);
23
24       return 0;
25 }
```

Dopo Linea	Valore delle Variabili				
	a	b	c	d	arg
14	unk	unk	0	unk	unk
9 (in thread 1)					
9 (in thread 2)					
9 (in thread 3)					
20					
21					
22					

Esercizio 6

Valore delle variabili

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <pthread.h>
4
5  int c = 0;
6
7  void * th_fun(void *arg){
8     int a = 0, b = (int)arg;
9     a = b + c;
10    return (void *)a;
11 }
12 int main(int argc, char * argv[]){
13       pthread_t th1, th2, th3;
14       long d;
15
16       pthread_create(&th1, NULL, th_fun, (void *)3);
17       pthread_create(&th2, NULL, th_fun, (void *)4);
18       pthread_create(&th3, NULL, th_fun, (void *)5);
19
20       pthread_join(th1, (void **)&d);
21       pthread_join(th2, (void **)&d);
22       pthread_join(th3, (void **)&d);
23
24       return 0;
25 }
```

Dopo Linea	Valore delle Variabili				
	a	**b**	**c**	**d**	**arg**
14	unk	unk	0	unk	unk
9 (in thread 1)	3	3	0	unk	3
9 (in thread 2)					
9 (in thread 3)					
20					
21					
22					

Esercizio 6

Valore delle variabili

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <pthread.h>
4
5  int c = 0;
6
7  void * th_fun(void *arg){
8     int a = 0, b = (int)arg;
9     a = b + c;
10    return (void *)a;
11 }
12 int main(int argc, char * argv[]){
13       pthread_t th1, th2, th3;
14       long d;
15
16       pthread_create(&th1, NULL, th_fun, (void *)3);
17       pthread_create(&th2, NULL, th_fun, (void *)4);
18       pthread_create(&th3, NULL, th_fun, (void *)5);
19
20       pthread_join(th1, (void **)&d);
21       pthread_join(th2, (void **)&d);
22       pthread_join(th3, (void **)&d);
23
24       return 0;
25 }
```

Dopo Linea	Valore delle Variabili				
	a	**b**	**c**	**d**	**arg**
14	unk	unk	0	unk	unk
9 (in thread 1)	3	3	0	unk	3
9 (in thread 2)	4	4	0	unk	4
9 (in thread 3)	5	5	0	unk	5
20	5	5	0	5	5
21	5	5	0	5	5
22	5	5	0	5	5

Esercizio 6

Valore delle variabili

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <pthread.h>
4
5  int c = 0;
6
7  void * th_fun(void *arg){
8     int a = 0, b = (int)arg;
9     a = b + c;
10    return (void *)a;
11 }
12 int main(int argc, char * argv[]){
13       pthread_t th1, th2, th3;
14       long d;
15
16       pthread_create(&th1, NULL, th_fun, (void *)3);
17       pthread_create(&th2, NULL, th_fun, (void *)4);
18       pthread_create(&th3, NULL, th_fun, (void *)5);
19
20       pthread_join(th1, (void **)&d);
21       pthread_join(th2, (void **)&d);
22       pthread_join(th3, (void **)&d);
23
24       return 0;
25 }
```

Dopo Linea	Valore delle Variabili				
	a	**b**	**c**	**d**	**arg**
14	unk	unk	0	unk	unk
9 (in thread 1)	3	3	0	unk	3
9 (in thread 2)	4	4	0	unk	4
9 (in thread 3)	5	5	0	unk	5
20					
21					
22					

Esercizio 6

Valore delle variabili

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <pthread.h>
4
5  int c = 0;
6
7  void * th_fun(void *arg){
8     int a = 0, b = (int)arg;
9     a = b + c;
10    return (void *)a;
11 }
12 int main(int argc, char * argv[]){
13       pthread_t th1, th2, th3;
14       long d;
15
16       pthread_create(&th1, NULL, th_fun, (void *)3);
17       pthread_create(&th2, NULL, th_fun, (void *)4);
18       pthread_create(&th3, NULL, th_fun, (void *)5);
19
20       pthread_join(th1, (void **)&d);
21       pthread_join(th2, (void **)&d);
22       pthread_join(th3, (void **)&d);
23
24       return 0;
25 }
```

Dopo Linea	Valore delle Variabili				
	a	**b**	**c**	**d**	**arg**
14	unk	unk	0	unk	unk
9 (in thread 1)	3	3	0	unk	3
9 (in thread 2)	4	4	0	unk	4
9 (in thread 3)	5	5	0	unk	5
20	unk	unk	0	3	unk
21					
22					

Esercizio 6

Valore delle variabili

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <pthread.h>
4
5  int c = 0;
6
7  void * th_fun(void *arg){
8     int a = 0, b = (int)arg;
9     a = b + c;
10    return (void *)a;
11 }
12 int main(int argc, char * argv[]){
13       pthread_t th1, th2, th3;
14       long d;
15
16       pthread_create(&th1, NULL, th_fun, (void *)3);
17       pthread_create(&th2, NULL, th_fun, (void *)4);
18       pthread_create(&th3, NULL, th_fun, (void *)5);
19
20       pthread_join(th1, (void **)&d);
21       pthread_join(th2, (void **)&d);
22       pthread_join(th3, (void **)&d);
23
24       return 0;
25 }
```

Dopo Linea	Valore delle Variabili				
	a	**b**	**c**	**d**	**arg**
14	unk	unk	0	unk	unk
9 (in thread 1)	3	3	0	unk	3
9 (in thread 2)	4	4	0	unk	4
9 (in thread 3)	5	5	0	unk	5
20	unk	unk	0	3	unk
21	unk	unk	0	4	unk
22					

Esercizio 6

Valore delle variabili

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <pthread.h>
4
5  int c = 0;
6
7  void * th_fun(void *arg){
8     int a = 0, b = (int)arg;
9     a = b + c;
10    return (void *)a;
11 }
12 int main(int argc, char * argv[]){
13       pthread_t th1, th2, th3;
14       long d;
15
16       pthread_create(&th1, NULL, th_fun, (void *)3);
17       pthread_create(&th2, NULL, th_fun, (void *)4);
18       pthread_create(&th3, NULL, th_fun, (void *)5);
19
20       pthread_join(th1, (void **)&d);
21       pthread_join(th2, (void **)&d);
22       pthread_join(th3, (void **)&d);
23
24       return 0;
25 }
```

Dopo Linea	Valore delle Variabili				
	a	**b**	**c**	**d**	**arg**
14	unk	unk	0	unk	unk
9 (in thread 1)	3	3	0	unk	3
9 (in thread 2)	4	4	0	unk	4
9 (in thread 3)	5	5	0	unk	5
20	unk	unk	0	3	unk
21	unk	unk	0	4	unk
22	unk	unk	0	5	unk

Mutex & Deadlock

*I **mutex** (mutual exclusion) sono un procedimento di **sincronizzazione** fra processi o thread concorrenti, con il quale si impedisce che più task paralleli accedano contemporaneamente ai dati in memoria o ad altre risorse soggette a race condition (o corsa critica).*

*Usando i mutex, sia ha **possibilità di deadlock** nel caso due o più thread diversi acquisiscano due o più mutex in ordine diverso, con tali sequenze di acquisizione innestate una dentro l'altra; questa è la situazione più comune, ma non l'unica, attraverso la quale si può avere una situazione di deadlock.*

Esercizio 7

Trova il Deadlock


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

pthread_mutex_t mutexA = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutexB = PTHREAD_MUTEX_INITIALIZER;

void * thread1(void *arg){
    pthread_mutex_lock(&mutexA);
    pthread_mutex_lock(&mutexB);
    printf("Sono il thread_1\n");
    pthread_mutex_unlock(&mutexB);
    pthread_mutex_unlock(&mutexA);
    return NULL;
}
void * thread2(void *arg){
    pthread_mutex_lock(&mutexB);
    pthread_mutex_lock(&mutexA);
    printf("Sono il thread_2\n");
    pthread_mutex_unlock(&mutexA);
    pthread_mutex_unlock(&mutexB);
    return NULL;
}

int main(int argc, char * argv[]){
    pthread_t th1, th2;

    pthread_create(&th1, NULL, thread1, NULL);
    pthread_create(&th2, NULL, thread2, NULL);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);

    return 0;
}
```

Traccia dell'esercizio: Indicare se nel seguente programma sono presenti deadlock e, in tal caso, identificare la sequenza di istruzioni che porta alla situazione di deadlock.

Esercizio 7

Trova il Deadlock

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

pthread_mutex_t mutexA = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutexB = PTHREAD_MUTEX_INITIALIZER;

void * thread1(void *arg){
    pthread_mutex_lock(&mutexA);
    pthread_mutex_lock(&mutexB);
    printf("Sono il thread_1\n");
    pthread_mutex_unlock(&mutexB);
    pthread_mutex_unlock(&mutexA);
    return NULL;
}
void * thread2(void *arg){
    pthread_mutex_lock(&mutexB);
    pthread_mutex_lock(&mutexA);
    printf("Sono il thread_2\n");
    pthread_mutex_unlock(&mutexA);
    pthread_mutex_unlock(&mutexB);
    return NULL;
}

int main(int argc, char * argv[]){
    pthread_t th1, th2;

    pthread_create(&th1, NULL, thread1, NULL);
    pthread_create(&th2, NULL, thread2, NULL);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);

    return 0;
}
```

Traccia dell'esercizio: Indicare se nel seguente programma sono presenti deadlock e, in tal caso, identificare la sequenza di istruzioni che porta alla situazione di deadlock.

Soluzione: thread1 blocca mutexA e contemporaneamente thread2 blocca mutexB; nessuno dei due thread potrà più procedere, in quanto thread1 aspetta che thread2 sblocchi mutexB e thread2 aspetta che thread1 sblocchi mutexA.

Esercizio 7

Grafo di attesa

- I rettangoli sono i threads
- I cerchi sono le risorse
- Una freccia da un thread ad una risorsa indica che il thread è in attesa della risorsa (Fig. 1)
- Una freccia da una risorsa ad un thread indica che la risorsa è bloccata da un thread (Fig. 2)

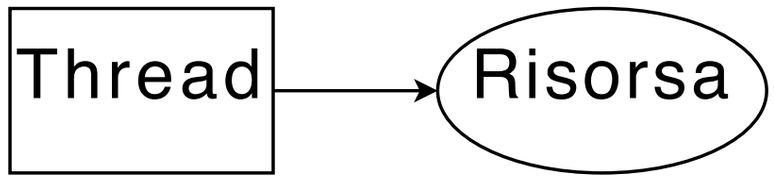


Figure 1: Thread in attesa di una risorsa.

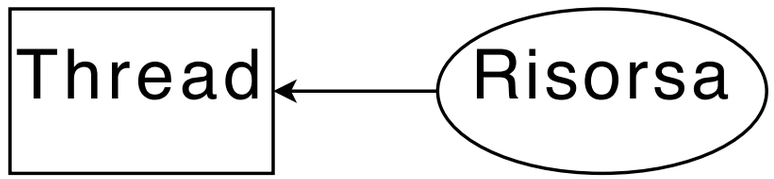


Figure 2: Thread che possiede una risorsa.

NEL NOSTRO ESEMPIO

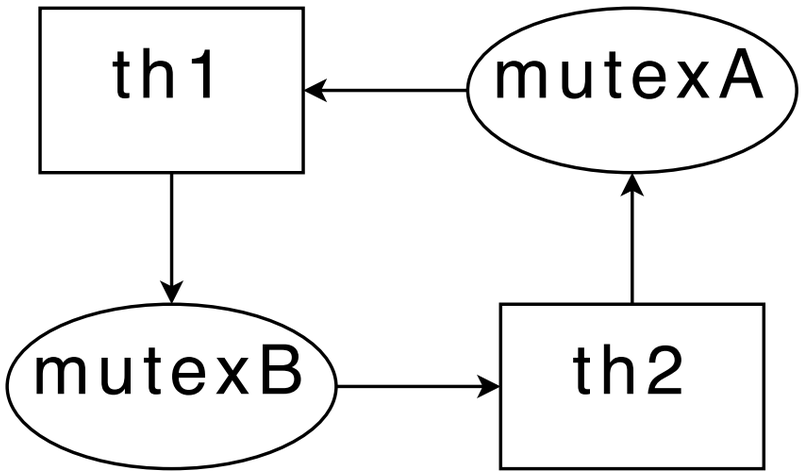


Figure 3: Deadlock.

Esercizio 8

Trova il Deadlock (2)

```
pthread_mutex_t mutexA = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutexB = PTHREAD_MUTEX_INITIALIZER;

void foo(int val){
    pthread_mutex_lock(&mutexB);
    val *= 2;
    if(val > 4)
        pthread_mutex_unlock(&mutexB);
}

void * thread1(void *arg){
    pthread_mutex_lock(&mutexA);
    foo((int)arg);
    pthread_mutex_unlock(&mutexA);
    if((int)arg < 2)
        pthread_mutex_unlock(&mutexB);
    pthread_mutex_lock(&mutexA);
    printf("Sono alla fine del thread1\n");
    pthread_mutex_unlock(&mutexA);
    return NULL;
}
void * thread2(void *arg){
    pthread_mutex_lock(&mutexA);
    pthread_mutex_lock(&mutexB);
    printf("Sono alla fine del thread2\n");
    pthread_mutex_unlock(&mutexB);
    pthread_mutex_unlock(&mutexA);
    return NULL;
}

int main(int argc, char * argv[]){
    pthread_t th1, th2;

    pthread_create(&th1, NULL, thread1, (void *)2);
    pthread_create(&th2, NULL, thread2, NULL);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);

    return 0;
}
```

Esercizio 8

Trova il Deadlock (2)

```
pthread_mutex_t mutexA = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutexB = PTHREAD_MUTEX_INITIALIZER;

void foo(int val){
    pthread_mutex_lock(&mutexB);
    val *= 2;
    if(val > 4)
        pthread_mutex_unlock(&mutexB);
}

void * thread1(void *arg){
    pthread_mutex_lock(&mutexA);
    foo((int)arg);
    pthread_mutex_unlock(&mutexA);
    if((int)arg < 2)
        pthread_mutex_unlock(&mutexB);
    pthread_mutex_lock(&mutexA);
    printf("Sono alla fine del thread1\n");
    pthread_mutex_unlock(&mutexA);
    return NULL;
}
void * thread2(void *arg){
    pthread_mutex_lock(&mutexA);
    pthread_mutex_lock(&mutexB);
    printf("Sono alla fine del thread2\n");
    pthread_mutex_unlock(&mutexB);
    pthread_mutex_unlock(&mutexA);
    return NULL;
}

int main(int argc, char * argv[]){
    pthread_t th1, th2;

    pthread_create(&th1, NULL, thread1, (void *)2);
    pthread_create(&th2, NULL, thread2, NULL);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);

    return 0;
}
```

Questo programma è stato chiaramente strutturato male, in quanto la funzione thread1 non libererà mai la risorsa mutexB nel caso l'argomento val della funzione foo sia uguale a 2. In questo caso, se la funzione thread1 blocca mutexB prima che lo faccia la funzione thread2, quest'ultima resterà bloccata in attesa che mutexB venga liberato, cosa che non accadrà mai. Il grafo di attesa è rappresentato in Figura 4.

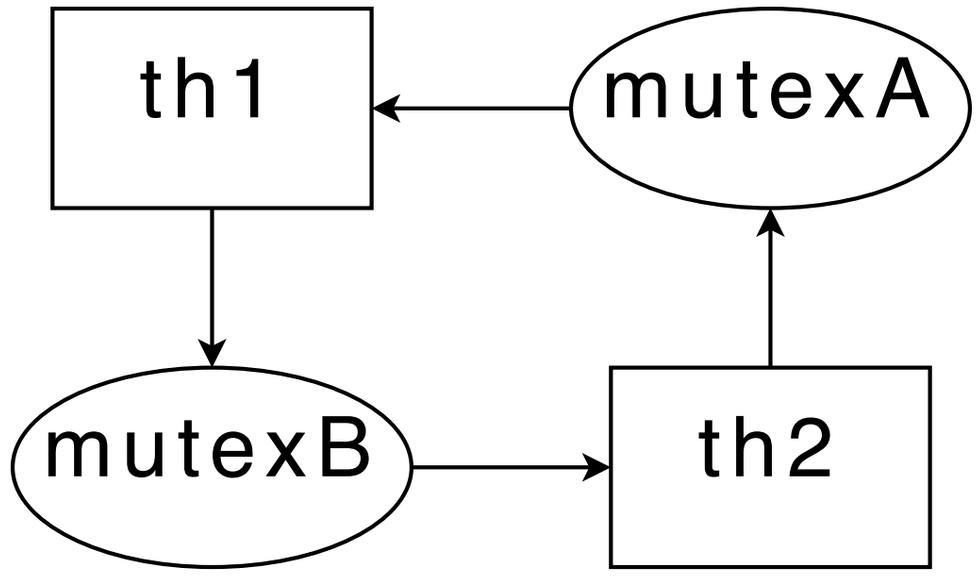


Figure 4: Deadlock.

Semafori

- Un semaforo è una variabile intera
    1. `sem_init(…)` inizializza il semaforo ad un valore
    2. `sem_wait(…)` decrementa il valore del semaforo di 1
        1. se il valore già 0 il thread viene bloccato
        2. il valore del semaforo può essere negativo (*quanti sono in attesa*)
    3. `sem_post(…)` incrementa il valore del semaforo di 1
        1. se ci sono thread in attesa sul semaforo, uno di questi viene sbloccato

- Il valore di un semaforo può essere interpretato nel modo seguente:
    1. se il valore è positivo, rappresenta il numero di thread che lo possono decrementare senza andare in attesa
    2. se è negativo, rappresenta il numero di thread che sono bloccati in attesa
    3. se è 0 indica che non ci sono thread in attesa, ma il prossimo thread che eseguirà una wait andrà in attesa.

Esercizio 9

Quattro amici fanno una scommessa, il premio per il vincitore è di poter fare bere uno dei rimanenti amici. Chi vince la scommessa sceglie casualmente a chi tocca bere il cocktail tutto d'un fiato mentre gli altri stanno a guardare. Alla fine, viene proposta un'ulteriore scommessa e il gioco va avanti all'infinito. Rappresentare il problema utilizzando i semafori.

Esercizio 9

Quattro amici fanno una scommessa, il premio per il vincitore è di poter fare bere uno dei rimanenti amici. Chi vince la scommessa sceglie casualmente a chi tocca bere il cocktail tutto d'un fiato mentre gli altri stanno a guardare. Alla fine, viene proposta un'ulteriore scommessa e il gioco va avanti all'infinito. Rappresentare il problema utilizzando i semafori.

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

sem_t sem_turno[3];
sem_t sem_bevuto;

void * amico(void *arg){
    while(1){
        sem_wait(&sem_turno[(int)arg]);
        // E' STATO SCELTO PER BERE
        sem_post(&sem_bevuto);
        // HA FINITO DI BERE
    }
    return NULL;
}
void * vincitore(void *arg){
    int turno = 0;
    while(1){
        sem_wait(&sem_bevuto);
        turno = rand()%3;
        sem_post(&sem_turno[turno]);
    }
    return NULL;
}
```

```c
int main(int argc, char * argv[]){
    pthread_t th_amici[3];
    pthread_t th_vincitore;
    int i = 0;

    for(i = 0; i < 3; i++){
        sem_init(&sem_turno[i], 0, 0);
    }
    sem_init(&sem_bevuto, 0, 1);

    for(i = 0; i < 3; i++){
        pthread_create(&(th_amici[i]), NULL, amico, (void *)i);
    }
    pthread_create(&th_vincitore, NULL, vincitore, NULL);

    for(i = 0; i < 3; i++){
        pthread_join(th_amici[i], NULL);
    }
    pthread_join(th_vincitore, NULL);

    return 0;
}
```

Esercizio 10

Trova il Deadlock (3)

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

sem_t sem_vuoto1;
sem_t sem_pieno1;

sem_t sem_vuoto2;
sem_t sem_pieno2;

int coda1;
int coda2;

void * th_fun1(void *arg){
    sem_wait(&sem_vuoto1);
    sem_wait(&sem_pieno2);
    coda1 = rand();
    printf("Il valore della coda2 e' %d\n", coda2);
    sem_post(&sem_pieno1);
    sem_post(&sem_vuoto2);
    return NULL;
}
void * th_fun2(void *arg){
    sem_wait(&sem_vuoto2);
    sem_wait(&sem_pieno1);
    coda2 = rand();
    printf("Il valore della coda1 e' %d\n", coda1);
    sem_post(&sem_pieno2);
    sem_post(&sem_vuoto1);
    return NULL;
}
int main(int argc, char * argv[]){
    pthread_t th1, th2;

    sem_init(&sem_vuoto1, 0, 0);
    sem_init(&sem_vuoto2, 0, 0);
    sem_init(&sem_pieno1, 0, 1);
    sem_init(&sem_pieno2, 0, 1);
    pthread_create(&th1, NULL, th_fun1, (void *)3);
    pthread_create(&th2, NULL, th_fun2, (void *)4);

    pthread_join(th1, NULL);
    pthread_join(th2, NULL);

    return 0;
}
```

Esercizio 10

Trova il Deadlock (3)

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

sem_t sem_vuoto1;
sem_t sem_pieno1;

sem_t sem_vuoto2;
sem_t sem_pieno2;

int coda1;
int coda2;

void * th_fun1(void *arg){
    sem_wait(&sem_vuoto1);
    sem_wait(&sem_pieno2);
    coda1 = rand();
    printf("Il valore della coda2 e' %d\n", coda2);
    sem_post(&sem_pieno1);
    sem_post(&sem_vuoto2);
    return NULL;
}
void * th_fun2(void *arg){
    sem_wait(&sem_vuoto2);
    sem_wait(&sem_pieno1);
    coda2 = rand();
    printf("Il valore della coda1 e' %d\n", coda1);
    sem_post(&sem_pieno2);
    sem_post(&sem_vuoto1);
    return NULL;
}
int main(int argc, char * argv[]){
    pthread_t th1, th2;

    sem_init(&sem_vuoto1, 0, 0);
    sem_init(&sem_vuoto2, 0, 0);
    sem_init(&sem_pieno1, 0, 1);
    sem_init(&sem_pieno2, 0, 1);
    pthread_create(&th1, NULL, th_fun1, (void *)3);
    pthread_create(&th2, NULL, th_fun2, (void *)4);

    pthread_join(th1, NULL);
    pthread_join(th2, NULL);

    return 0;
}
```

Il thread *th1*, corrispondente alla funzione *th_fun1*, non riesce a procedere perchè è in attesa (sulla chiamata *sem_wait{&sem_vuoto1}*) che il thread *th2*, corrispondente alla funzione *th_fun2*, chiami la funzione *sem_post* sul semaforo *sem_vuoto1*, mentre il thread *th2* è in attesa (sulla chiamata *sem_wait{&sem_vuoto2}*) che il thread corrispondente alla funzione *th_fun1* chiami la funzione *sem_post* sul semaforo *sem_vuoto2*.

Nessuno dei due thread riescirà quindi a procedere in quanto aspettano entrambi che sia l'altro thread ad incrementare per primo il valore del semaforo sul quale ciascuno è in attesa.

Esercizio 11

Valore delle variabili

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <pthread.h>
4
5  pthread_mutex_t mutexA = PTHREAD_MUTEX_INITIALIZER;
6
7  int c = 0, e = 0;
8
9  void * th_fun1(void *arg){
10     int a = 0, b = (int)arg;
11     pthread_mutex_lock(&mutexA);
12     e = 1;
13     a = b + c;
14     pthread_mutex_unlock(&mutexA);
15     return (void *)a;
16 }
17 void * th_fun2(void *arg){
18     while(e == 0)
19         ;
20     pthread_mutex_lock(&mutexA);
21     c = ((int)arg)*2;
22     pthread_mutex_unlock(&mutexA);
23     return (void *)c;
24 }
25 int main(int argc, char * argv[]){
26     pthread_t th1, th2;
27     int d;
28
29     pthread_create(&th1, NULL, th_fun1, (void *)3);
30     pthread_create(&th2, NULL, th_fun2, (void *)4);
31
32     pthread_join(th1, (void *)&d);
33     pthread_join(th2, (void *)&d);
34
35     return 0;
36 }
```

Il valore di una variabile puo' essere indicato come:

- Intero, carattere, stringa, se univocamente individuabile.
- `NE`: se la variabile **non esiste**.
- `UND`: se la variabile esiste ma non e' stata ancora inizializzata.
- `V1`/`V2`: nel caso in cui una variabile puo' avere al massimo i due valori `V1` e `V2`.
- `UNK`: per i casi in cui la variabile puo' avere piu' di due valori.

		Valore delle Variabili				
Thread	**Dopo Linea**	**a**	**b**	**c**	**d**	**e**
th1	12					
th1	13					
th1	14					
th2	22					
main	32					
main	33					

Esercizio 11

Valore delle variabili

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <pthread.h>
4
5  pthread_mutex_t mutexA = PTHREAD_MUTEX_INITIALIZER;
6
7  int c = 0, e = 0;
8
9  void * th_fun1(void *arg){
10     int a = 0, b = (int)arg;
11     pthread_mutex_lock(&mutexA);
12     e = 1;
13     a = b + c;
14     pthread_mutex_unlock(&mutexA);
15     return (void *)a;
16 }
17 void * th_fun2(void *arg){
18     while(e == 0)
19         ;
20     pthread_mutex_lock(&mutexA);
21     c = ((int)arg)*2;
22     pthread_mutex_unlock(&mutexA);
23     return (void *)c;
24 }
25 int main(int argc, char * argv[]){
26     pthread_t th1, th2;
27     int d;
28
29     pthread_create(&th1, NULL, th_fun1, (void *)3);
30     pthread_create(&th2, NULL, th_fun2, (void *)4);
31
32     pthread_join(th1, (void *)&d);
33     pthread_join(th2, (void *)&d);
34
35     return 0;
36 }
```

Il valore di una variabile puo' essere indicato come:

- Intero, carattere, stringa, se univocamente individuabile.
- `NE`: se la variabile **non esiste**.
- `UND`: se la variabile esiste ma non e' stata ancora inizializzata.
- `V1`/`V2`: nel caso in cui una variabile puo' avere al massimo i due valori `V1` e `V2`.
- `UNK`: per i casi in cui la variabile puo' avere piu' di due valori.

		Valore delle Variabili				
Thread	**Dopo Linea**	**a**	**b**	**c**	**d**	**e**
th1	12	0	3	0	UND	1
th1	13					
th1	14					
th2	22					
main	32					
main	33					

Esercizio 11

Valore delle variabili

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <pthread.h>
4
5  pthread_mutex_t mutexA = PTHREAD_MUTEX_INITIALIZER;
6
7  int c = 0, e = 0;
8
9  void * th_fun1(void *arg){
10     int a = 0, b = (int)arg;
11     pthread_mutex_lock(&mutexA);
12     e = 1;
13     a = b + c;
14     pthread_mutex_unlock(&mutexA);
15     return (void *)a;
16 }
17 void * th_fun2(void *arg){
18     while(e == 0)
19         ;
20     pthread_mutex_lock(&mutexA);
21     c = ((int)arg)*2;
22     pthread_mutex_unlock(&mutexA);
23     return (void *)c;
24 }
25 int main(int argc, char * argv[]){
26     pthread_t th1, th2;
27     int d;
28
29     pthread_create(&th1, NULL, th_fun1, (void *)3);
30     pthread_create(&th2, NULL, th_fun2, (void *)4);
31
32     pthread_join(th1, (void *)&d);
33     pthread_join(th2, (void *)&d);
34
35     return 0;
36 }
```

Il valore di una variabile puo' essere indicato come:

- Intero, carattere, stringa, se univocamente individuabile.
- `NE`: se la variabile **non esiste**.
- `UND`: se la variabile esiste ma non e' stata ancora inizializzata.
- `V1`/`V2`: nel caso in cui una variabile puo' avere al massimo i due valori `V1` e `V2`.
- `UNK`: per i casi in cui la variabile puo' avere piu' di due valori.

		Valore delle Variabili				
Thread	**Dopo Linea**	**a**	**b**	**c**	**d**	**e**
th1	12	0	3	0	UND	1
th1	13	3	3	0	UND	1
th1	14					
th2	22					
main	32					
main	33					

Esercizio 11

Valore delle variabili

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <pthread.h>
4
5  pthread_mutex_t mutexA = PTHREAD_MUTEX_INITIALIZER;
6
7  int c = 0, e = 0;
8
9  void * th_fun1(void *arg){
10     int a = 0, b = (int)arg;
11     pthread_mutex_lock(&mutexA);
12     e = 1;
13     a = b + c;
14     pthread_mutex_unlock(&mutexA);
15     return (void *)a;
16 }
17 void * th_fun2(void *arg){
18     while(e == 0)
19         ;
20     pthread_mutex_lock(&mutexA);
21     c = ((int)arg)*2;
22     pthread_mutex_unlock(&mutexA);
23     return (void *)c;
24 }
25 int main(int argc, char * argv[]){
26     pthread_t th1, th2;
27     int d;
28
29     pthread_create(&th1, NULL, th_fun1, (void *)3);
30     pthread_create(&th2, NULL, th_fun2, (void *)4);
31
32     pthread_join(th1, (void *)&d);
33     pthread_join(th2, (void *)&d);
34
35     return 0;
36 }
```

Il valore di una variabile puo' essere indicato come:

- Intero, carattere, stringa, se univocamente individuabile.
- `NE`: se la variabile **non esiste**.
- `UND`: se la variabile esiste ma non e' stata ancora inizializzata.
- `V1`/`V2`: nel caso in cui una variabile puo' avere al massimo i due valori `V1` e `V2`.
- `UNK`: per i casi in cui la variabile puo' avere piu' di due valori.

		Valore delle Variabili				
Thread	**Dopo Linea**	**a**	**b**	**c**	**d**	**e**
th1	12	0	3	0	UND	1
th1	13	3	3	0	UND	1
th1	14	3	3	0 (prima di riga 21)/8 (dopo)	UND	1
th2	22					
main	32					
main	33					

Esercizio 11

Valore delle variabili

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <pthread.h>
4
5  pthread_mutex_t mutexA = PTHREAD_MUTEX_INITIALIZER;
6
7  int c = 0, e = 0;
8
9  void * th_fun1(void *arg){
10     int a = 0, b = (int)arg;
11     pthread_mutex_lock(&mutexA);
12     e = 1;
13     a = b + c;
14     pthread_mutex_unlock(&mutexA);
15     return (void *)a;
16 }
17 void * th_fun2(void *arg){
18     while(e == 0)
19         ;
20     pthread_mutex_lock(&mutexA);
21     c = ((int)arg)*2;
22     pthread_mutex_unlock(&mutexA);
23     return (void *)c;
24 }
25 int main(int argc, char * argv[]){
26     pthread_t th1, th2;
27     int d;
28
29     pthread_create(&th1, NULL, th_fun1, (void *)3);
30     pthread_create(&th2, NULL, th_fun2, (void *)4);
31
32     pthread_join(th1, (void *)&d);
33     pthread_join(th2, (void *)&d);
34
35     return 0;
36 }
```

Il valore di una variabile puo' essere indicato come:

- Intero, carattere, stringa, se univocamente individuabile.
- `NE`: se la variabile **non esiste**.
- `UND`: se la variabile esiste ma non e' stata ancora inizializzata.
- `V1`/`V2`: nel caso in cui una variabile puo' avere al massimo i due valori `V1` e `V2`.
- `UNK`: per i casi in cui la variabile puo' avere piu' di due valori.

		Valore delle Variabili				
Thread	**Dopo Linea**	**a**	**b**	**c**	**d**	**e**
th1	12	0	3	0	UND	1
th1	13	3	3	0	UND	1
th1	14	3	3	0 (prima di riga 21)/8 (dopo)	UND	1
th2	22	3/NE	3 /NE	8	UND/3	1
main	32					
main	33					

Esercizio 11

Valore delle variabili

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <pthread.h>
4
5  pthread_mutex_t mutexA = PTHREAD_MUTEX_INITIALIZER;
6
7  int c = 0, e = 0;
8
9  void * th_fun1(void *arg){
10     int a = 0, b = (int)arg;
11     pthread_mutex_lock(&mutexA);
12     e = 1;
13     a = b + c;
14     pthread_mutex_unlock(&mutexA);
15     return (void *)a;
16 }
17 void * th_fun2(void *arg){
18     while(e == 0)
19         ;
20     pthread_mutex_lock(&mutexA);
21     c = ((int)arg)*2;
22     pthread_mutex_unlock(&mutexA);
23     return (void *)c;
24 }
25 int main(int argc, char * argv[]){
26     pthread_t th1, th2;
27     int d;
28
29     pthread_create(&th1, NULL, th_fun1, (void *)3);
30     pthread_create(&th2, NULL, th_fun2, (void *)4);
31
32     pthread_join(th1, (void *)&d);
33     pthread_join(th2, (void *)&d);
34
35     return 0;
36 }
```

Il valore di una variabile puo' essere indicato come:

- Intero, carattere, stringa, se univocamente individuabile.
- `NE`: se la variabile **non esiste**.
- `UND`: se la variabile esiste ma non e' stata ancora inizializzata.
- `V1`/`V2`: nel caso in cui una variabile puo' avere al massimo i due valori `V1` e `V2`.
- `UNK`: per i casi in cui la variabile puo' avere piu' di due valori.

		Valore delle Variabili				
Thread	**Dopo Linea**	**a**	**b**	**c**	**d**	**e**
th1	12	0	3	0	UND	1
th1	13	3	3	0	UND	1
th1	14	3	3	0 (prima di riga 21)/8 (dopo)	UND	1
th2	22	3/NE	3 /NE	8	UND/3	1
main	32	NE	NE	0 (prima di riga 21)/8 (dopo)	3	1
main	33					

Esercizio 11

Valore delle variabili

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <pthread.h>
4
5  pthread_mutex_t mutexA = PTHREAD_MUTEX_INITIALIZER;
6
7  int c = 0, e = 0;
8
9  void * th_fun1(void *arg){
10     int a = 0, b = (int)arg;
11     pthread_mutex_lock(&mutexA);
12     e = 1;
13     a = b + c;
14     pthread_mutex_unlock(&mutexA);
15     return (void *)a;
16 }
17 void * th_fun2(void *arg){
18     while(e == 0)
19         ;
20     pthread_mutex_lock(&mutexA);
21     c = ((int)arg)*2;
22     pthread_mutex_unlock(&mutexA);
23     return (void *)c;
24 }
25 int main(int argc, char * argv[]){
26     pthread_t th1, th2;
27     int d;
28
29     pthread_create(&th1, NULL, th_fun1, (void *)3);
30     pthread_create(&th2, NULL, th_fun2, (void *)4);
31
32     pthread_join(th1, (void *)&d);
33     pthread_join(th2, (void *)&d);
34
35     return 0;
36 }
```

Il valore di una variabile puo' essere indicato come:

- Intero, carattere, stringa, se univocamente individuabile.
- `NE`: se la variabile **non esiste**.
- `UND`: se la variabile esiste ma non e' stata ancora inizializzata.
- `V1`/`V2`: nel caso in cui una variabile puo' avere al massimo i due valori `V1` e `V2`.
- `UNK`: per i casi in cui la variabile puo' avere piu' di due valori.

		Valore delle Variabili				
Thread	**Dopo Linea**	**a**	**b**	**c**	**d**	**e**
th1	12	0	3	0	UND	1
th1	13	3	3	0	UND	1
th1	14	3	3	0 (prima di riga 21)/8 (dopo)	UND	1
th2	22	3/NE	3 /NE	8	UND/3	1
main	32	NE	NE	0 (prima di riga 21)/8 (dopo)	3	1
main	33	NE	NE	8	8	1

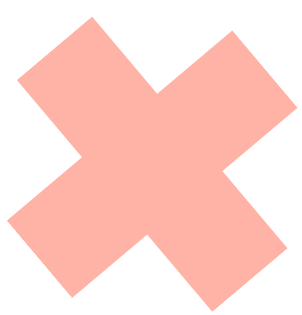
Esercizio 12

Produttore/Consumatore

```c
char buffer;
int fine = 0;

void * scrivi(void *arg){
    int i;
    for (i=0; i < 5; i++){
        buffer = 'a' + i;
        printf( "scrivi: ho scritto '%c'\n", buffer );
    }
    fine = 1;
    return NULL;
}
void * leggi(void *arg){
    char miobuffer = '\x0';
    while (fine == 0){
        miobuffer = buffer;
        printf( "leggi: ho letto '%c'\n", miobuffer );
    }
    return NULL;
}
int main(void){
    pthread_t th1, th2;

    pthread_create(&th1, NULL, &scrivi, NULL);
    pthread_create(&th2, NULL, &leggi, NULL);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    return 0;
}
```



VERSIONE ERRATA

```c
char buffer;
int fine = 0;
sem_t pieno, vuoto;

void * scrivi(void *arg){
    int i;
    for (i=0; i < 5; i++){
        sem_wait(&vuoto);
        buffer = 'a' + i;
        sem_post(&pieno);
    }
    sem_wait(&vuoto);
    fine = 1;
    sem_post(&pieno);
    return NULL;
}

void * leggi(void *arg){
    char miobuffer = '\x0';
    while ( 1 ){
        sem_wait(&pieno);
        if( fine != 0 )
            return NULL;
        miobuffer = buffer;
        sem_post(&vuoto);
    }
}

int main(void){
    pthread_t th1, th2;

    sem_init(&pieno, 0, 0);
    sem_init(&vuoto, 0, 1);
    pthread_create(&th1, NULL, &scrivi, NULL);
    pthread_create(&th2, NULL, &leggi, NULL);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    return 0;
}
```

Aspetta che sia vuoto

Notifica che ora è pieno

Aspetta che sia pieno

Notifiche che ora è vuoto

Esercizio 12
Barriera

Si consideri il seguente problema (problema della barriera): un numero T di thread deve sincronizzarsi in un punto, cioè ogni thread deve arrivare a un punto (punto di sincronizzazione) dove deve attendere che tutti gli altri thread siano arrivati al loro punto di sincronizzazione.

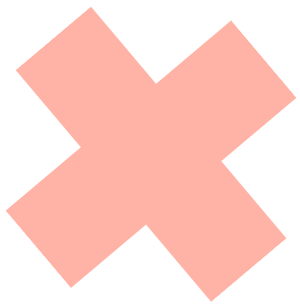
Esercizio 12
Barriera

Si consideri il seguente problema (problema della barriera): un numero T di thread deve sincronizzarsi in un punto, cioè ogni thread deve arrivare a un punto (punto di sincronizzazione) dove deve attendere che tutti gli altri thread siano arrivati al loro punto di sincronizzazione.

Una prima soluzione (sbagliata) consiste nell'uso di un semaforo A e un contatore, applicata da tutti i thread nel modo seguente:

programma 1 (eseguito da ogni thread)

```
1       conta++;
2       if (conta==T) post A;
3       wait A;
```



VERSIONE ERRATA

Questo programma contiene 2 errori gravi: individuarli e correggerli■

Esercizio 12
Barriera

Il primo errore consiste nella mancanza di protezione del contatore durante l'aggiornamento. Questo errore può essere corretto introducendo un mutex M.

Il secondo errore è il deadlock. Supponiamo T = 10. Dopo l'arrivo di 9 thread il valore di conta sarà 9 e quello del semaforo A sarà -9; a questo punto arriva l'ultimo thread, conta diventa 10, viene eseguito un post, un thread in attesa viene sbloccato e il semaforo viene incrementato a -8. Tutti gl ialtri thread restano bloccati.

Per correggere questi errori il programma deve diventare

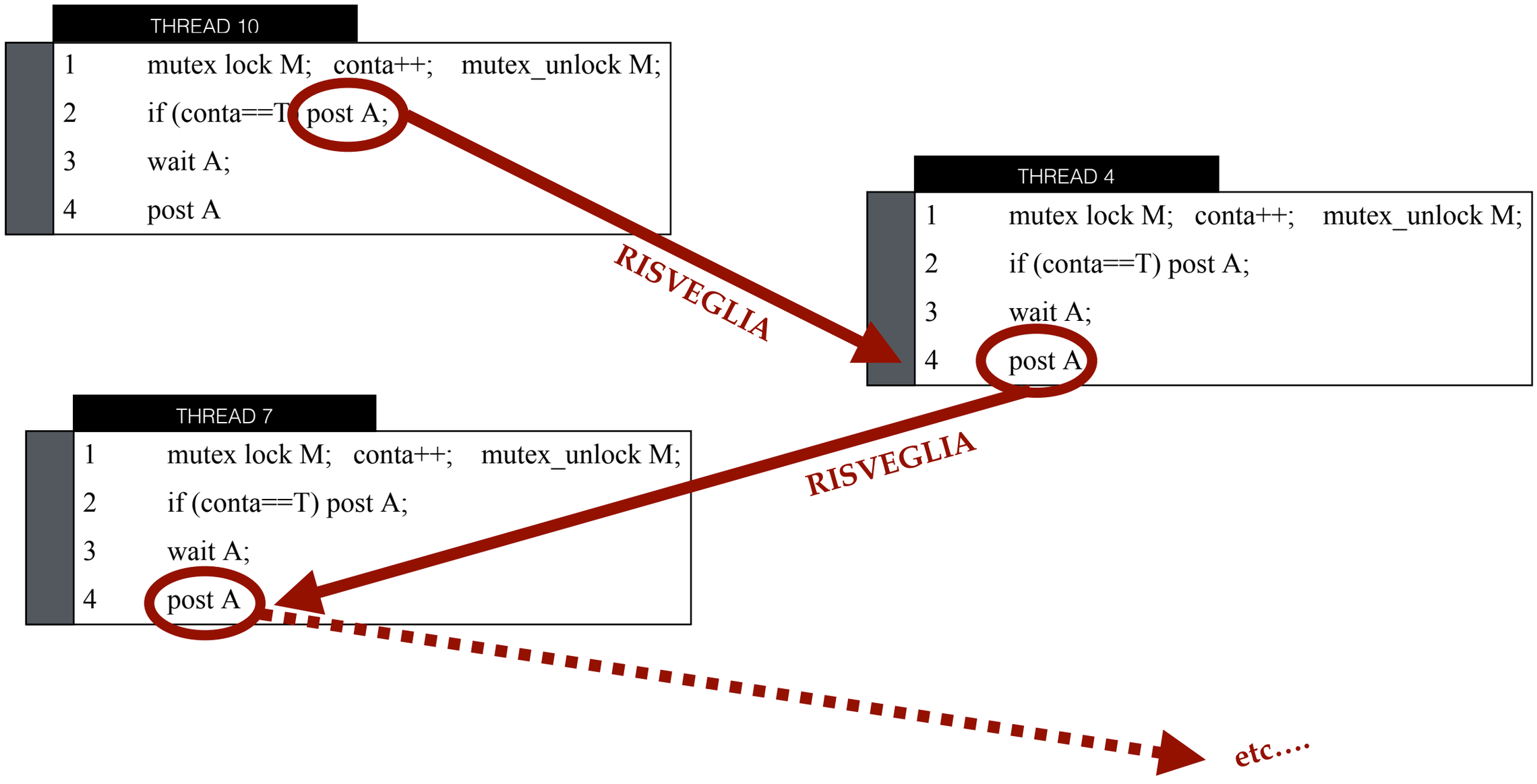
```
1       mutex lock M;   conta++;    mutex_unlock M;
2       if (conta==T) post A;
3       wait A;
4       post A
```

In questo modo i primi 9 thread vanno in attesa, l'ultimo sblocca un thread, che parte ed esegue il post (riga 4), sbloccandone un altro, che esegue il post, e così via fino a sbloccare tutti i thread.

Esercizio 12

Barriera

Tutti e T=10 i thread sono partiti. Quindi abbiamo, dopo linea 2, *conta=10* e A = -9



Esercizio 13

Il problema dei cinque filosofi

Ci sono cinque filosofi seduti stabilmente ad un tavolo rotondo e ciascuno ha davanti un piatto di riso (che supponiamo inesauribile), oppure esiste un unico grande piatto al centro della tavola da cui ogni filosofo mangia (questo non cambia il problema). Tra un piatto ed un altro (o tra un filosofo ed un altro) però non vi sono due bastoncini come sarebbe opportuno, ma solo uno, per un totale di sole 5 bacchette, contro le 10 che sarebbero richieste per permettere a tutti di mangiare contemporaneamente. Ciascun filosofo per tutta la sua vita non fa altro che tre cose: pensare quando e' sazio, mangiare e mettersi in attesa di mangiare quando ha fame. Un filosofo decide di mettersi a pensare a qualche problema filosofico per un certo intervallo di tempo (che possiamo ritenere casuale e finito), poi si siede a tavola (supponiamo ciascuno abbia assegnato un posto fisso, ma questo non è necessario) e decide di mangiare il riso. Impiega un certo tempo a mangiare (anche questo lo faremo variare casualmente, ma sarà sempre finito), poi decide di mettersi di nuovo a pensare, poi mangia di nuovo, ecc. Quando decide di mangiare, prima di poterlo fare, ha bisogno di prendere in mano due bacchette. Un filosofo può prendere solo le due bacchette che stanno alla sua destra e alla sua sinistra, una per volta, e solo se sono libere, ovvero non può sottrarre la risorsa bacchetta ad un altro filosofo che l'ha già acquisita, cioè che sta mangiando (no preemption). Finché non riesce a prendere le bacchette, il filosofo deve aspettare affamato. Quando invece, appena dopo aver mangiato, è sazio, posa le bacchette al loro posto e si mette a pensare per un certo tempo.

La situazione è tale che due filosofi vicini non possono mai mangiare contemporaneamente. Si chiede di trovare una soluzione, ossia una strategia tramite la quale i filosofi sono costretti ad acquisire le risorse (le bacchette) che soddisfi i seguenti due obiettivi:

1. non può accadere che tutti i filosofi rimangono indefinitamente bloccati in attesa reciproca che l'altro rilasci l'ulteriore bacchetta di cui hanno bisogno per mangiare (no deadlock)
2. non accade mai che uno o più filosofi non riescano mai a mangiare e quindi muoiano di fame perché gli altri sono più svelti di loro a prendere le bacchette (no starvation)

Esercizio 13

Il problema dei cinque filosofi

Alla prossima lezione

alessandro.nacci@polimi.it