1. Assumiamo un array di 6 interi A inizializzato con [0,0,0,0,0,0] condiviso da thread che appartengono a tre tipi:
- thread di tipo 1: ciclicamente generano un numero random k ed effettuano l'operazione A[0] = A[0] + k; A[1] = A[1] + k; A[2] = A[2] + k, che deve essere indivisibile sul dato A[0,2]
- thread di tipo 2: ciclicamente generano un numero random k ed effettuano l'operazione A[3] = A[3] + k; A[4] = A[4] + k; A[5] = A[5] + k, che deve essere indivisibile sul dato A[3,5]
- thread di tipo 3: ciclicamente stampano il valore A[0]+A[1]+A[2]+A[3]+A[4]+A[5].
Usando i semafori con la semantica tradizionale, scrivere il codice dei 3 tipi di thread, rispettando il seguente vincolo: un thread può essere in waiting su un semaforo solo se ciò è necessario per garantire le indivisibilità delle operazioni dei thread di tipo 1 e 2.

2. Assumiamo un array di int A inizializzato con [10,20].
Un thread esegue l'operazione A[1]=A[2]+5.
L'altro thread A[2]=A[1]+55.
Si argomenti in modo formale se possono verificarsi race condition su A.

3. Si spieghi il ruolo e il funzionamento della IPT.

Esercizio 1.

Variabili:

wrk1: numero thread di tipo 1 che stanno lavorando. Valore iniziale 0. Valori possibili: 0,1.
wrk2: numero thread di tipo 2 che stanno lavorando. Valore iniziale 0. Valori possibili: 0,1.
wrk3: numero thread di tipo 3 che stanno lavorando. Valore iniziale 0. Valori possibili: 0,1,2,3,……

tw1: numero thread di tipo 1 in waiting. Valore iniziale 0.
tw2: numero thread di tipo 2 in waiting. Valore iniziale 0.
tw3: numero thread di tipo 3 in waiting. Valore iniziale 0.

Semafori:

mutex. Valore iniziale 1. Serve per garantire accesso alle variabili condivise di cui sopra in sezioni critiche.

s1: Valore iniziale 0. Serve per mettere in waiting i thread di tipo 1.
s2: Valore iniziale 0. Serve per mettere in waiting i thread di tipo 2.
s3: Valore iniziale 0. Serve per mettere in waiting i thread di tipo 3.

Thread tipo 1:

```
while(true){
 // some work having nothing to do with our array
wait(mutex);
if(wrk1>0 ll wrk3>0){
   tw1++; signal(mutex); wait(s1);}
else{
   wrk1++; signal(mutex);
}

k= …..
A[0]=A[0]+k; A[1]=A[1]+k; A[2]=A[2]+k;

wait(mutex);
 wrk1- - ;
 if(tw1>0){tw1 - - ; wrk1++; signal(s1);}
 else{
      while(tw3>0 & wrk2==0){tw3- - ; wrk3++; signal(s3);}
 }
signa(mutex);
}
// some work having nothing to do with our array
}
```

Thread tipo 2: analogo a thread di tipo 1.

Thread tipo 3:

```
while(true){
 // some work having nothing to do with our array
wait(mutex);
if(wrk1>0 ll wrk2>0){
   tw3++; signal(mutex); wait(s3);}
else{
   wrk3++; signal(mutex);
}

print(A[0]+…+A[6]);

wait(mutex);
 wrk3- - ;
 if(wrk3==0 & tw1>0){tw1 - - ; wrk1++; signal(s1);}
 if(wrk3==0 & tw2>0){tw1 - - ; wrk2++; signal(s2);}
signa(mutex);
}
```

```
// some work having nothing to do with our array
}
```

Funzione che formalizza l'operazione del primo thread (cioè A[1] = A[2] + 5):
f1([m,n]) = [n+5,n].
Nota: in particolare, vale che
f1([10,20]) = [20+5,20] = [25,20] e
f1([10,65]) = [65+5,65] = [70,65].

Funzione che formalizza l'operazione del secondo thread (cioè A[2] = A[1] + 55):
f2([m,n]) = [m,m+55].
Nota: in particolare, vale che
f2([10,20]) = [10,10+55] = [10,65] e
f2([25,20]) = [25,25+55] = [25,80].

Valori ammissibili su A, cioè valore che otterremmo in caso di esecuzione sequenziale delle due operazioni:
f2(f1([x,y])) = f2([y+5,y]) = [y+5,y+5+55] = [y+5,y+60]
e
f1(f2([x,y])) = f1([x,x+55]) = [x+55+5,x+55] = [x+60,x+55].

Nel nostro esempio:
f2(f1([10,20])) = [20+5,20+60] = [25,80]
f2(f1([10,20])) = [10+60,10+55] = [70,65]

Possiamo avere r.c. perchè l'array può assumere, per esempio, il valore [25,65] che è diverso dai due ammissibili.