UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Dipartimento di Ingegneria “Enzo Ferrari”

Corso di Laurea Magistrale in Ingegneria Informatica

RISOLUZIONE DI ESERCIZI DI PROGRAMMAZIONE CONCORRENTE IN LINGUAGGIO C

Tutor:

Chiar.ma Prof.ssa Letizia Leonardi

Tesina di:

Labieni Tommaso

Anno Accademico 2021-2022

INDICE

INTRODUZIONE

Nella seguente tesina è presente la risoluzione di quattro esercizi presi dal libro: "*Little Book Of Semaphores (da pag. 127 a pag. 250)"* inerenti alla programmazione concorrente.

Gli esercizi proposti sono i seguenti:

1. Il problema del "Senate Bus";
2. Il problema del "Multi-Car Roller Coaster";
3. Il problema del "Search-Inser-Delete";
4. Il problema del barbiere di Hilzer.

Di cui i problemi 2 e 4 sono delle *estensioni* di certi problemi noti in letteratura come i *classici problemi della programmazione concorrente*.

Nel libro da cui sono stati recuperati i vari esercizi sono presenti delle soluzioni ai vari problemi; tuttavia, per alcuni di essi, sono stati proposti dei cambiamenti e in certi casi anche delle migliorie rispetto le soluzioni proposte.

In questa tesina i problemi riportati sono ordinati per difficoltà crescente in cui, per ogni singolo esercizio, viene riportato:

- La descrizione del problema;
- Gli approcci adottati per risolvere i vincoli di concorrenza;
- Le parti di codice rilevanti:
- Commenti sull'esecuzione del codice.

ESERCIZIO 1 – IL PROBLEMA DEL "SENATE BUS"

1.1 Descrizione del problema

Ci sono due tipologie di thread:

- Thread rider (o passeggero) che sale sul bus;
- Thread bus che trasporta passeggeri.

I rider arrivano alla fermata dell'autobus in attesa che arrivi. Quando l'autobus giunge alla fermata, **tutti i rider in attesa** invocano la funzione *boardBus*, tuttavia, chiunque arrivi alla fermata **mentre l'autobus è gia arrivato**, dovrà aspettare la corsa successiva. La capacità del bus è di 50 rider; se ce ne dovessero essere più di 50 in attesa, alcuni di essi dovranno aspettare il prossimo bus.

Quanto tutti i rider in attesa sono saliti sul bus, quest'ultimo invocherà la funzione *depart*. Se l'autobus dovesse arrivare quando non c'è alcun rider in attesa, questo partirà immediatamente.

La risoluzione del seguente problema implica la considerazione di diversi vincoli:

1. Occorre **regolare** il numero di rider che possono essere in attesa; infatti, se ci dovessero essere N persone in attesa (con N > 50), occorre **bloccare** dalla 51-esima alla N-esima persona fino a che non si liberano i posti in attesa;
2. Occorre **bloccare** i rider che arrivano alla fermata PRIMA dell'arrivo del bus (e che dunque rimangono in attesa di salire);
3. Occorre **bloccare** i rider che arrivano alla fermata DOPO l'arrivo del bus (ossia vengono bloccati i rider *"in ritardo"*);
4. Occorre **bloccare** il bus affinché questo **non parta** fino a che l'ultimo passeggero non sia salito.

1.2 Approccio per risolvere i vincoli di concorrenza

Per risolvere i vincoli elencati nel paragrafo 1.1 è stato utilizzato il costrutto MONITOR. Le variabili condition utilizzate sono illustrate di seguito:

- C_BUS_CAPACITY: Su questa condition variable andranno a bloccarsi i rider che **superano** la capacità consentita. Per verificare la condizione è stata utilizzata una variabile INTERA

(num_rider_bus_stop) che conta il numero di passeggeri giunti alla fermata del bus e dunque, nel caso in cui il valore di questa variabile NON SIA MINORE della capacità del bus, si causerà la sospensione del thread → Risolve il problema imposto dal vincolo 1. Le parti rilevanti dell'utilizzo di questa condition variable sono riportate nel paragrafo 1.3;

- C_BUS: Su questa condition variable andranno a bloccarsi i rider che arrivano PRIMA del bus, ossia i passeggeri che attendono di salire sul bus. Per verificare la condizione è stata utilizzata una variabile BOOLEANA (bus_pronto) che vale *true* se l'autobus è arrivato *false* altrimenti; fino a che il valore di questa variabile NON SIA TRUE (e dunque fino a che il bus non è arrivato), si causerà la sospensione del thread → Risolve il problema imposto dal vincolo 2. Le parti rilevanti dell'utilizzo di questa condition variable sono riportate nel paragrafo 1.3;
- C_LATE_RIDER: Su questa condition variable andranno a bloccarsi i thread dei rider che arrivano DOPO il bus. Per verificare la condizione è stata utilizzata la stessa variabile booleana precedentemente descritta (bus_pronto), tuttavia qui la condizione da verificare è DUALE in quanto, non appena un rider arriva alla fermata, se la variabile bus_pronto vale *true* significa che l'autobus è arrivato e che dunque sta facendo salire i passeggeri; ciò causerà la sospensione del thread → Risolve il problema imposto dal vincolo 3. Le parti rilevanti dell'utilizzo di questa condition variable sono riportate nel paragrafo 1.3;
- C_LAST_RIDER_ON: Su questa condition variable andrà a bloccarsi il thread del bus in attesa che salga l'ultimo passeggero. Per verificare la condizione è stata utilizzata una variabile BOOLEANA (ultimo_rider_salito) che vale *true* se, appunto, l'ultimo rider è salito sul bus, *false* altrimenti. Fino a che questa variabile varrà *false* il rimarrà sospeso → Risolve il problema imposto dal vincolo 4. Le parti rilevanti dell'utilizzo di questa condition variable sono riportate nel paragrafo 1.3;

1.3 Parti rilevanti del codice

Le parti più significative del codice del passeggero sono riportate di seguito:

```
pthread_mutex_lock(&mutex);
while(num_rider_bus_stop >= C) {  ← Fino a che il numero dei rider alla fermata del bus è maggiore della capacità C, il passeggero rimane in attesa
        pthread_cond_wait(&C_BUS_CAPACITY, &mutex);
}
num_rider_bus_stop++;
while(bus_pronto) {
        pthread_cond_wait(&C_LATE_RIDER, &mutex);
}
```

```
num_rider_attesa++;

while (!bus_pronto) {
   /* Il rider aspetta l'arrivo del bus */
   pthread_cond_wait(&C_BUS, &mutex);
}
/* Bus arrivato --> il rider invoca boardBus */
boardBus(*ptr);
```

Da questo codice si evince che:

1. Fino a che il numero dei passeggeri alla fermata del bus (numero memorizzato nella variabile *num_rider_bus*_stop) NON è minore della capacità *C* → Il passeggero rimane in attesa sulla condition variable C_BUS_CAPACITY;
2. Una volta giunti alla fermata (e dunque superato il controllo di cui al punto precedente), oltre ad incrementare il valore della variabile globale *num_rider_bus*_stop si verifica se l'autobus è arrivato (denotato dal fatto che la variabile *bus_pronto* valga *true*) → in tal caso il passeggero attende la corsa successiva (ossia rimane in attesa sulla condition variable C_LATE_RIDER);
3. Una volta superati i precedenti controlli, si incrementa il contatore dei rider in attesa del bus (numero memorizzato nella variabile globale *num_rider_attesa*) e il passeggero si mette in attesa del bus → Rimane in attesa sulla condition variable C_BUS.

Alla fine, il rider, come da specifica, invoca la funzione *boardBus* le cui parti più rilevanti sono riportate di seguito:

```
num_rider_attesa--;
num_rider_bus_stop--;
if (num_rider_attesa == 0) {
      ultimo_rider_salito = true;
      pthread_mutex_unlock(&mutex);
      pthread_cond_signal(&C_LAST_RIDER_ON);
      return;
}
pthread_mutex_unlock(&mutex); ←Questo è il mutex bloccato nella funzione routine del rider
```

Funzione nella quale si decrementa il numero dei rider in attesa e il numero dei rider alla fermata del bus. Quando *num_rider_attesa* vale 0, significa che l'ultimo rider è salito sul bus e che dunque si può risvegliare il thread del bus aggiornando la variabile booleana *ultimo_rider_salito* descritta nel paragrafo 1.2 ed effettuando la signal sulla variabile condition C_LAST_RIDER_ON. Si riportano ora le parti principali del thread del bus (il quale opera all'infinito):

```
while(1) {
```

```
        pthread_mutex_lock(&mutex);
        bus_pronto = true;
        if (num_rider_attesa > 0) {
              pthread_cond_broadcast(&C_BUS);
              while(!ultimo_rider_salito) {
                    pthread_cond_wait(&C_LAST_RIDER_ON, &mutex);
              }
        } else {
              ➔ L'autobus fa una corsa senza passeggeri ←
        }
        pthread_mutex_unlock(&mutex);
        depart(*ptr);
}
```

Non appena l'autobus arriva, imposta la variabile *bus_pronto* precedentemente descritta a *true*, in seguito controlla se ci sono dei rider in attesa di partire. Qui gli scenari possibili sono due:

1. Nel caso in cui non ci dovessero essere rider in attesa (denotato dal fatto che la variabile *num_rider_attesa* sia uguale 0), come da specifica, l'autobus partirà **immediatamente;**

2. Nel caso in cui ci dovesse essere **almeno** un rider in attesa, il bus risveglia tutti i rider in attesa di partire (chiamando la funzione *pthread_cond_broadcast* sulla condition variable C_BUS) per poi aspettare la salita dell'ultimo passeggero sulla variabile condition C_LAST_RIDER_ON. Dopo ciò, come da specifica, il bus invoca la funzione *depart* le cui parti rilevanti sono mostrate di seguito:

```
bus_pronto = false;
ultimo_rider_salito = false;
pthread_cond_broadcast(&C_LATE_RIDER);
pthread_cond_broadcast(&C_BUS_CAPACITY);
pthread_mutex_unlock(&mutex);
```

Funzione in cui si resettano le variabili *bus_pronto* e *ultimo_rider_salito* a *false* e si risvegliano **sia tutti** i thread dei passeggeri in ritardo **sia tutti** i thread dei passeggeri bloccati per superamento della capacità *C*. Qui sorge una domanda non scontata: perché è possibile risvegliarli tutti senza effettuare alcun controllo? I motivi sono due:

1. **Per quanto riguarda i thread dei passeggeri in ritardo:** Sicuramente il numero **massimo** di questi thread è *C* in quanto può capitare che l'autobus arrivi prima di tutti i seguenti thread (e che dunque parta senza passeggeri) e nel mentre si accodano esattamente *C* passeggeri; la garanzia che non si accodino più di *C* passeggeri è dovuta dal fatto che prima di mettersi in attesa del bus, si controlla il numero dei rider alla fermata attraverso la variabile globale *num_rider_bus_stop*.

2. **Per quanto riguarda i thread dei passeggeri bloccati per superamento della capacità:** Una volta risvegliati tutti questi thread, prima di potersi mettere in coda, il rider controlla che il numero di passeggeri alla fermata del bus (ossia che il valore della variabile *num_rider_bus_stop*) sia **maggiore** della capacità *C* del bus stesso (dal momento che il controllo è stato inserito in un loop *while*); in tal caso il thread si blocca nuovamente e dunque non si supera la capacità C → il programma rimane valido.

Infine, è stata effettuata una modifica al main durante la creazione del thread del bus con il fine di avere un comportamento "aleatorio" del programma. Il codice di creazione dei thread bus è riportato di seguito:

```
srand(time(NULL));      ← inizializzazione per generare il numero pseudo-casuale
bus_i = rand() % NUM_THREADS;     ← Creazione dell'indice del thread bus
for (i=0; i < NUM_THREADS; i++)
{
      taskids[i] = i;
      if (i != bus_i) {
            …      ← creazione del thread rider
      } else {          ← Qui viene generato il thread del bus
            if (pthread_create(&thread[i], NULL, eseguiBus, (void *)
      (&taskids[i])) != 0)
            {
                  exit(8);
            }
      }
}
```

Si nota che la creazione del thread del bus è pseudo-casuale; in questo modo si avrà (potenzialmente) un comportamento diverso per ogni esecuzione come mostrato nel paragrafo 1.4.

1.4 Commenti su un'esecuzione del programma

In questo capitolo si riporta una rilevante invocazione dell'esercizio appena descritto. L'invocazione è la seguente:

```
$ ./senate_bus 51
```

E il risultato è il seguente: (← si noti che sono stati rimossi i messaggi di creazione dei thread di tipo rider per evitare di occupare un numero eccessivo di pagine)

```
Il thread RIDER di indice 0 con identificatore 140516129376000 e' in attesa del bus.
Il thread RIDER di indice 1 con identificatore 140516120983296 e' in attesa del bus.
Il thread RIDER di indice 2 con identificatore 140516112590592 e' in attesa del bus.
Il thread RIDER di indice 4 con identificatore 140516029101824 e' in attesa del bus.
Il thread RIDER di indice 5 con identificatore 140516020709120 e' in attesa del bus.
```

Il thread RIDER di indice 6 con identificatore 140516012316416 e' in attesa del bus.
Il thread RIDER di indice 7 con identificatore 140516003923712 e' in attesa del bus.
Il thread RIDER di indice 9 con identificatore 140515987138304 e' in attesa del bus.
Il thread RIDER di indice 10 con identificatore 140515978745600 e' in attesa del bus.
Il thread RIDER di indice 11 con identificatore 140515961587456 e' in attesa del bus.
Il thread RIDER di indice 3 con identificatore 140515969980160 e' in attesa del bus.
Il thread RIDER di indice 14 con identificatore 140515936409344 e' in attesa del bus.
Il thread RIDER di indice 12 con identificatore 140515953194752 e' in attesa del bus.
Il thread RIDER di indice 13 con identificatore 140515944802048 e' in attesa del bus.
Il thread RIDER di indice 8 con identificatore 140515995531008 e' in attesa del bus.
Il thread RIDER di indice 15 con identificatore 140515928016640 e' in attesa del bus.
Il thread RIDER di indice 16 con identificatore 140515919623936 e' in attesa del bus.
Il thread RIDER di indice 18 con identificatore 140514821142272 e' in attesa del bus.
Il thread RIDER di indice 19 con identificatore 140514812749568 e' in attesa del bus.
Il thread RIDER di indice 17 con identificatore 140515911231232 e' in attesa del bus.
Il thread RIDER di indice 20 con identificatore 140514804356864 e' in attesa del bus.
Il thread RIDER di indice 23 con identificatore 140514779178752 e' in attesa del bus.
Il thread RIDER di indice 22 con identificatore 140514787571456 e' in attesa del bus.
Il thread RIDER di indice 24 con identificatore 140514770786048 e' in attesa del bus.
Il thread RIDER di indice 26 con identificatore 140514410096384 e' in attesa del bus.
Il thread RIDER di indice 21 con identificatore 140514795964160 e' in attesa del bus.
Il thread RIDER di indice 25 con identificatore 140514418489088 e' in attesa del bus.
Il thread RIDER di indice 27 con identificatore 140514401703680 e' in attesa del bus.
Il thread RIDER di indice 28 con identificatore 140514393310976 e' in attesa del bus.
Il thread RIDER di indice 32 con identificatore 140514284271360 e' in attesa del bus.
Il thread RIDER di indice 33 con identificatore 140514275878656 e' in attesa del bus.
Il thread RIDER di indice 31 con identificatore 140514368132864 e' in attesa del bus.
Il thread RIDER di indice 30 con identificatore 140514376525568 e' in attesa del bus.
Il thread RIDER di indice 29 con identificatore 140514384918272 e' in attesa del bus.
Il thread RIDER di indice 35 con identificatore 140514259093248 e' in attesa del bus.
Il thread RIDER di indice 36 con identificatore 140514250700544 e' in attesa del bus.
Il thread RIDER di indice 34 con identificatore 140514267485952 e' in attesa del bus.
Il thread RIDER di indice 37 con identificatore 140514242307840 e' in attesa del bus.
Il thread RIDER di indice 38 con identificatore 140514233915136 e' in attesa del bus.
Il thread RIDER di indice 39 con identificatore 140513210529536 e' in attesa del bus.
Il thread RIDER di indice 40 con identificatore 140513202136832 e' in attesa del bus.
Il thread RIDER di indice 41 con identificatore 140513193744128 e' in attesa del bus.
Il thread RIDER di indice 42 con identificatore 140513185351424 e' in attesa del bus.
Il thread RIDER di indice 43 con identificatore 140513176958720 e' in attesa del bus.
Il thread RIDER di indice 44 con identificatore 140513168566016 e' in attesa del bus.

Si mettono in attesa 48 persone su 50

Il thread RIDER di indice 45 con identificatore 140513160173312 e' in attesa del bus.

Il thread RIDER di indice 46 con identificatore 140512673658624 e' in attesa del bus. ——

Il thread RIDER di indice 47 con identificatore 140512665265920 e' in attesa del bus. ← **SALGONO TUTTI I RIDER DALL'INDICE 0 ALL'INDICE 47 INCLUSO!**

SONO IL MAIN e ho creato il Pthread 48-esimo con id=140512656873216 (bus)

Sto per creare il thread 49-esimo (rider)

Thread BUS di indice 48 partito: Ho come identificatore 140512656873216

L'autobus e' arrivato. Chiunque arrivi da ora in poi dovra' aspettare la corsa successiva. ← **AUTOBUS ARRIVATO !!!**

SONO IL MAIN e ho creato il Pthread 49-esimo con id=140512648480512 (rider)

Sto per creare il thread 50-esimo (rider)

Thread RIDER di indice 49 partito: Ho come identificatore 140512648480512

SONO IL MAIN e ho creato il Pthread 50-esimo con id=140512640087808 (rider)

Thread RIDER di indice 50 partito: Ho come identificatore 140512640087808

Si bloccano i rider di indice 49 e 50 perché arrivati in ritardo!!

Il thread RIDER di indice 49 con identificatore 140512648480512 e' bloccato a causa dell'autobus arrivato .

Il thread RIDER di indice 12 con identificatore 140515953194752 e' salito sul bus.

Il thread RIDER di indice 3 con identificatore 140515969980160 e' salito sul bus.

Il thread RIDER di indice 16 con identificatore 140515919623936 e' salito sul bus.

Il thread RIDER di indice 8 con identificatore 140515995531008 e' salito sul bus.

Il thread RIDER di indice 50 con identificatore 140512640087808 e' bloccato a causa dell'autobus arrivato .

Il thread RIDER di indice 15 con identificatore 140515928016640 e' salito sul bus.

Il thread RIDER di indice 18 con identificatore 140514821142272 e' salito sul bus.

→ **SI SALTANO TUTTE LE STAMPE DEI THREAD RIDER CHE SALGONO SUL BUS** ←

Il thread BUS di indice 48 con identificatore 140512656873216 e' partito.

Il numero dei rider saliti e': 48

Il thread RIDER di indice 50 con identificatore 140512640087808 e' in attesa del bus.

Il thread RIDER di indice 49 con identificatore 140512648480512 e' in attesa del bus.

I rider precedentemente bloccati salgono nella corsa successiva

L'autobus e' arrivato. Chiunque arrivi da ora in poi dovra' aspettare la corsa successiva.

Il thread RIDER di indice 49 con identificatore 140512648480512 e' salito sul bus.

Il thread RIDER di indice 50 con identificatore 140512640087808 e' salito sul bus.

Il thread BUS di indice 48 con identificatore 140512656873216 e' partito.

Il numero dei rider saliti e': 2

L'autobus e' arrivato. Chiunque arrivi da ora in poi dovra' aspettare la corsa successiva. ← **non ci sono più passeggeri; fino a che i rider non terminano l'esecuzione il bus continuerà a fare dei viaggi senza passeggeri**

L'autobus fa una corsa senza passeggeri

ESERCIZIO 2 – IL PROBLEMA DEL "MULTI-CAR ROLLER COASTER"

2.1 Descrizione del problema

Questo problema è un'estensione del "*Roller Coaster Problem"* il quale è formulato come segue:

Ci sono due tipologie di thread:

- Thread di tipo passeggero che sale a bordo dell'auto;
- Thread di tipo auto che gira attorno ad un tracciato.

Di cui *n* thread di tipo passeggero ed un solo thread di tipo auto.

I passeggeri **ripetitivamente** aspettano di fare un giro sull'auto la quale può contenere *C* passeggeri (con $C < n$). L'auto può iniziare il giro **soltanto se piena**. Seguono ulteriori dettagli:

- I passeggeri invocano le funzioni *board* e *unboard* rispettivamente per salire e scendere dall'auto;
- L'auto invoca le funzioni *load*, *run*, e *unload* rispettivamente per consentire ai passeggeri di salire, iniziare il giro e consentire ai passeggeri di scendere;
- I passeggeri **non possono** invocare la funzione *board* (ossia salire nell'auto) **fino a che** l'auto non abbia invocato la funzione *load;*
- I passeggeri **non possono** invocare la funzione *unboard* (ossia scendere dall'auto) **fino a che** l'auto non abbia invocato la funzione *unload;*
- L'auto non può partire **fino a che** non sono saliti **esattamente** *C* passeggeri al suo interno (ossia fino a che i posti al suo interno non sono esauriti).

Come citato sopra, il problema risolto in questa tesina, prevede un'estensione dell'esercizio appena descritto aggiungendo le seguenti funzionalità:

- È consentito ad un numero *m* di auto (con $m >= 1$) di girare nel tracciato **contemporaneamente**;

- Soltanto una macchina alla volta può caricare persone al suo interno (ossia soltanto una macchina alla volta può invocare la funzione *load*; l'auto successiva potrà invocare tale funzione **solamente** dopo che l'auto corrente abbia caricato tutti i passeggeri);
- Le auto **non** possono sorpassarsi l'un l'altra, e dunque, devono invocare la funzione *unload*, per consentire la discesa dei passeggeri, **nello stesso ordine** in cui hanno caricato quest'ultimi (ossia nello stesso ordine in cui le auto hanno invocato la funzione *load*);
- **Tutti** i thread di tipo passeggero di un "*carload"* (ossia tutti i thread dentro la stessa identica auto) devono scendere (ossia invocare la funzione *unboard*) **prima** di uno dei qualsiasi altri thread (chiaramente di tipo passeggero) di un "*carload"* successivo.

La risoluzione del seguente problema implica la considerazione di diversi vincoli:

1. Occorre **regolare** l'ordine in cui i thread di tipo auto possono far salire i passeggeri (ossia l'ordine in cui questi potranno invocare il metodo *load*) consentendolo **ad una macchina alla volta**;
2. Occorre **regolare** l'ordine in cui i thread di tipo auto possono far scendere i passeggeri (ossia l'ordine in cui questi potranno invocare il metodo *unload*) in base all'ordine in cui è stata effettuata la salita dei passeggeri;
3. Occorre **bloccare** i thread di tipo passeggero che provano a invocare la funzione *board* **prima** che il thread di tipo auto abbia invocato la funzione *load*;
4. Occorre **bloccare** i thread di tipo passeggero che provano a invocare la funzione *unboard* **prima** che il thread di tipo auto abbia invocato la funzione *unload*;
5. Occorre **bloccare** i thread di tipo auto che tentano di far partire il giro nel circuito **fino a che** non sono saliti tutti i *C* passeggeri al suo interno;
6. Occorre che **tutti** i thread di tipo passeggero **di un certo** *carload* (termine specificato nel paragrafo 2.1) scendano dall'auto (ossia invochino il metodo *unboard*) **PRIMA** che un qualsiasi thread di un *carload* successivo possa invocare il metodo *unboard*.

2.2 Approccio per risolvere i vincoli di concorrenza

Per risolvere i vincoli elencati nel paragrafo 2.1 è stato utilizzato il meccanismo dei SEMAFORI. I semafori utilizzati sono illustrati di seguito:

- *S_LOAD_QUEUE: Questo array dinamico di *m* semafori viene utilizzato per **regolare** l'ordine in cui i thread di tipo auto possono far salire i passeggeri. Dovendo consentire **ad**

una sola auto per volta la salita dei passeggeri, il seguente array ha il primo semaforo (di indice 0) inizializzato 1 e i restanti *m – 1* semafori PRIVATI (dunque inizializzati a 0). Il modo in cui sono stati gestiti gli indici è riportato nel paragrafo 2.3. → Questo array di semafori risolve il problema imposto dal vincolo 1;

- *S_UNLOAD_QUEUE: Questo array dinamico di *m* semafori viene utilizzato per **regolare** l'ordine in cui i thread di tipo auto possono far scendere i passeggeri. Dovendo regolare la "*discesa*" dei passeggeri **nello stesso** ordine in cui è stata effettuata la "*salita*", il seguente array ha il primo semaforo (di indice 0) inizializzato 1 e tutti gli *m – 1* semafori PRIVATI (dunque inizializzati a 0) esattamente come per l'array dinamico di semafori S_LOAD_QUEUE. Il modo in cui sono stati gestiti gli indici è analogo a quello dell'array S_LOAD_QUEUE il quale è riportato nel paragrafo 2.3. Questo array di semafori risolve il problema imposto dal vincolo 2;
- S_CAR_LOAD: Questo semaforo **privato** viene utilizzato per consentire ai passeggeri di SALIRE sull'auto (varrà 0 fino a che un thread auto non invochi la funzione *load*); viene dunque utilizzato per risolvere il problema imposto dal vincolo 3. Le parti rilevanti dell'utilizzo di questo semaforo sono riportate nel paragrafo 2.3;
- S_CAR_UNLOAD: Questo semaforo **privato** viene utilizzato per consentire ai passeggeri di SCENDERE dall'auto (varrà 0 fino a che un thread auto non invochi la funzione *unload*); viene dunque utilizzato per risolvere il problema imposto dal vincolo 4. Il modo in cui è stato utilizzato questo semaforo è analogo a quello del semaforo S_CAR_LOAD il quale è riportato nel paragrafo 2.3;
- S_PASSEGGERI_SALITI: Questo semaforo **privato** viene utilizzato per verificare se un'auto contiene il numero esatto di passeggeri per partire; viene dunque utilizzato per risolvere il problema imposto dal vincolo 5. Le parti rilevanti dell'utilizzo di questo semaforo sono riportate nel paragrafo 2.3;
- S_PASSEGGERI_SCESI: Questo semaforo **privato** viene utilizzato per verificare se un'auto ha fatto scendere tutti i passeggeri al suo interno (e dunque tutti i passeggeri di un *carload*). Solamente dopo questo evento sarà possibile far scendere i passeggeri del *carload* successivo; viene dunque utilizzato per risolvere il problema imposto dal vincolo 6. Il suo utilizzo è riportato nel paragrafo 2.3.

2.3 Parti rilevanti del codice

Gli array dinamici di semafori S_LOAD_QUEUE e S_UNLOAD_QUEUE sono trattati come *array circolari* per consentire alla/e auto di invocare le funzioni *load* e *unload* in ordine di indice crescente da 0 a *m – 1* per poi ricominciare da capo (esattamente come negli *array circolari*). Come specificato nel paragrafo 2.2, il primo semaforo dell'array di semafori S_LOAD_QUEUE (e, analogamente, di S_UNLOAD_QUEUE) ha valore iniziale uguale ad 1, mentre i restanti semafori (se presenti[1]) hanno come valore iniziale 0. Così facendo, avendo *m* thread di tipo auto, l'ordine in cui questi si risveglieranno è il seguente:

0 → 1 → 2 → … → *m – 1* → 0 → …

Un thread di tipo auto, prima di eseguire la funzione *load*, si mette in attesa sul semaforo S_LOAD_QUEUE di indice UGUALE all'indice del thread medesimo come mostrato di seguito:

```
sem_wait(&S_LOAD_QUEUE[*ptr])  ← *ptr denota l'indice del thread auto
```

Per consentire **ad una macchina alla volta** la salita dei passeggeri, la signal al **prossimo** thread di tipo auto avverrà SOLTANTO dopo che **tutti i passeggeri siano saliti** e ciò avviene all'interno della funzione *load* mostrata di seguito:

```
void load(int indice) {← il parametro indice ha lo stesso valore di *ptr
        sem_post(&S_CAR_LOAD);
        sem_wait(&S_PASSEGGERI_SALITI);
        num_passeggeri_saliti = 0;
        sem_post(&S_LOAD_QUEUE[next_indice]);
}
```

Si nota che non appena il thread auto invoca la funzione *load*, effettua una *signal* sul semaforo S_CAR_LOAD (il cui utilizzo è stato specificato nel paragrafo 2.2) e si mette subito in attesa sul semaforo S_PASSEGGERI_SALITI sul quale verrà risvegliato dall'ultimo passeggero salito sull'auto; si nota, tuttavia, un problema: se ci fossero più di due passeggeri in attesa di salire sull'auto (e dunque in attesa sul semaforo S_CAR_LOAD), **se ne sveglierebbe soltanto uno** causando, dunque, deadlock. Come risolvere questo problema? Nella soluzione riportata sul PDF, si effettuava una signal particolare sul semaforo S_CAR_LOAD la quale andava ad impostare il valore di tale

[1] Si è detto "*se presenti*" in quanto nell'array di semafori S_LOAD_QUEUE (ma anche nell'array S_UNLOAD_QUEUE) vi sono tanti semafori quante macchine e dunque, se ci fosse soltanto una macchina, non ci sarebbe alcun semaforo inizializzato a 0.

semaforo esattamente a *C*; tutto questo, però, non è possibile in linguaggio C in quanto, la signal (ossia la funzione *sem_post*), può soltanto **incrementare** di 1 il valore di un semaforo. La soluzione a tale problema è riportata nella funzione *board* del passeggero riportata di seguito:

```
void board(int indice) {
      num_passeggeri_saliti++;
      if (num_passeggeri_saliti == C) { ← SE VERA, TUTTI I PASSEGERI SONO SALITI
            /* Si risveglia il thread dell'auto */
            pthread_mutex_unlock(&mutex);
            sem_post(&S_PASSEGGERI_SALITI);
            return;
      } else {
            /* Altrimenti si risveglia il prossimo passeggero */
            pthread_mutex_unlock(&mutex);
            sem_post(&S_CAR_LOAD);
      }
}
```

Si nota, che ogni volta che un passeggero sale nell'auto, viene incrementata una variabile (globale) *num_passeggeri_saliti* la quale permette di verificare se tutti i *C* passeggeri sono saliti all'interno dell'auto; se così non fosse, si risveglia il prossimo passeggero (evitando, dunque, il deadlock prima menzionato) altrimenti tutti i passeggeri sono saliti e dunque si risveglia il thread dell'auto in attesa sul semaforo S_PASSEGGERI_SALITI. Una volta saliti tutti i passeggeri, il thread dell'auto, risvegliato resetta la variabile globale *num_passeggeri_saliti* precedentemente illustrata per il prossimo *carload.* La prossima operazione da svolgere è quella di consentire al prossimo thread auto di invocare il metodo *load*; per fare ciò occorre recuperare l'indice del prossimo thread auto il cui calcolo è il seguente:

```
next_indice = (indice+1) % m
```

in questo modo *next_indice* potrà assumere i valori 0, 1, 2, …, *m – 1*, 0, 1, … esattamente come specificato all'inizio del seguente paragrafo.

Un dettaglio importante è che i passeggeri devono continuamente salire e scendere dall'auto (o dalle auto) perché altrimenti si rischierebbe di creare un deadlock per almeno un thread di tipo auto in quanto la capacità *C* rischierebbe di non essere colmata se un passeggero, dopo aver girato nel tracciato a bordo di un'auto, se ne andasse. Per interrompere dunque l'esecuzione del programma sarà necessario "*inviare*" il segnale SIGINT al processo utilizzando la coppia di comandi *ctrl+C.*

Per ultimo, si vuol far notare che questo programma ha un'invocazione differente dagli altri esercizi in quanto la corretta invocazione prevede la specifica dei seguenti parametri:

- Numero *n* di passeggeri;

- Capacità *C* della/e auto (assunta **uguale** per tutte le auto nel caso ce ne sia più di una);
- Numero *m* di auto.

Le quali devono rispettare le seguenti proprietà:

1. **Tutti** i parametri elencati devono essere numeri strettamente positivi (> 0);
2. Dal momento che il problema non esteso (ossia il *Roller Coaster Problem*) richiedeva che il numero dei passeggeri *n* sia strettamente maggiore della capacità *C*, vista la presenza di un certo numero *m* di auto (potenzialmente con *m > 1*), è richiesto che *n* sia **strettamente maggiore del numero dei posti TOTALI disponibili** (ossia *C*m*) al fine di garantire un'eterogeneità tra questi due problemi.

2.4 Commenti su diverse esecuzioni del programma

In questo capitolo si riportano due invocazioni rilevanti dell'esercizio appena descritto. La prima invocazione è la seguente:

```
$ ./multi-car-RollerCoaster 4 2 2
Errore: La variabile n non e' tale che n > C*m, infatti n=4 e C*m=4!
```

Questo è un esempio di invocazione errata. Il messaggio di errore è riportato sotto.

La seconda invocazione è la seguente:

$ ./multi-car-RollerCoaster 7 2 3

Questa invocazione andrà a generare: **7 passeggeri** e **3 auto con 2 posti ciascuna!**

E il risultato è il seguente: (← si noti che sono stati rimossi i messaggi di creazione dei thread di tipo rider per evitare di occupare un numero eccessivo di pagine)

```
Thread 0 PASSEGGERO partito: Ho come identificatore 140194616854272
Thread 1 PASSEGGERO partito: Ho come identificatore 140194608461568
Thread 3 PASSEGGERO partito: Ho come identificatore 140194591676160
Thread 4 PASSEGGERO partito: Ho come identificatore 140194376316672
Thread 6 PASSEGGERO partito: Ho come identificatore 140194359531264
Thread 7 AUTO partito: Ho come identificatore 140194351138560
Thread AUTO di indice 7 con identificatore 140194351138560 inizia a far salire i passeggeri
Il Thread PASSEGGERO di indice 0 con identificatore 140194616854272 e' salito nell'auto di indice 7.
Il Thread PASSEGGERO di indice 1 con identificatore 140194608461568 e' salito nell'auto di indice 7.
Thread AUTO di indice 7 con identificatore 140194351138560 ha caricato tutti i 2 passeggeri
Thread AUTO di indice 7 con identificatore 140194351138560 sta girando nel tracciato
Thread 8 AUTO partito: Ho come identificatore 140194342745856
```

Thread 9 AUTO partito: Ho come identificatore 140194334353152

Thread AUTO di indice 8 con identificatore 140194342745856 inizia a far salire i passeggeri

Il Thread PASSEGGERO di indice 2 con identificatore 140194600068864 e' salito nell'auto di indice 8.

Il Thread PASSEGGERO di indice 3 con identificatore 140194591676160 e' salito nell'auto di indice 8.

Thread AUTO di indice 8 con identificatore 140194342745856 ha caricato tutti i 2 passeggeri

Thread AUTO di indice 8 con identificatore 140194342745856 sta girando nel tracciato

Thread AUTO di indice 9 con identificatore 140194334353152 inizia a far salire i passeggeri← **L'AUTO DI INDICE 9 NON RIESCE A CARICARE PASSEGGERI FINO A CHE L'AUTO 8 NON ABBIA FINITO DI FARLO!!!**

Il Thread PASSEGGERO di indice 4 con identificatore 140194376316672 e' salito nell'auto di indice 9.

Il Thread PASSEGGERO di indice 5 con identificatore 140194367923968 e' salito nell'auto di indice 9.

Thread AUTO di indice 9 con identificatore 140194334353152 ha caricato tutti i 2 passeggeri

Thread AUTO di indice 9 con identificatore 140194334353152 sta girando nel tracciato

Thread AUTO di indice 7 con identificatore 140194351138560 inizia a far scendere i passeggeri

Il Thread PASSEGGERO di indice 0 con identificatore 140194616854272 e' sceso dall'auto di indice 7.

Il Thread PASSEGGERO di indice 1 con identificatore 140194608461568 e' sceso dall'auto di indice 7.

Thread AUTO di indice 7 con identificatore 140194351138560 ha fatto scendere tutti i 2 passeggeri

Thread AUTO di indice 7 con identificatore 140194351138560 inizia a far salire i passeggeri ← Si ricomincia da capo

Il Thread PASSEGGERO di indice 6 con identificatore 140194359531264 e' salito nell'auto di indice 7.

Thread AUTO di indice 8 con identificatore 140194342745856 inizia a far scendere i passeggeri

Il Thread PASSEGGERO di indice 2 con identificatore 140194600068864 e' sceso dall'auto di indice 8.

Il Thread PASSEGGERO di indice 3 con identificatore 140194591676160 e' sceso dall'auto di indice 8.

Thread AUTO di indice 8 con identificatore 140194342745856 ha fatto scendere tutti i 2 passeggeri

Thread AUTO di indice 9 con identificatore 140194334353152 inizia a far scendere i passeggeri — Si rispetta l'ordine delle persone salite

Il Thread PASSEGGERO di indice 4 con identificatore 140194376316672 e' sceso dall'auto di indice 9.

Il Thread PASSEGGERO di indice 5 con identificatore 140194367923968 e' sceso dall'auto di indice 9.

Thread AUTO di indice 9 con identificatore 140194334353152 ha fatto scendere tutti i 2 passeggeri

Il Thread PASSEGGERO di indice 3 con identificatore 140194591676160 e' salito nell'auto di indice 7.

Thread AUTO di indice 7 con identificatore 140194351138560 ha caricato tutti i 2 passeggeri

Thread AUTO di indice 7 con identificatore 140194351138560 sta girando nel tracciato

^C ← COME DESCRITTO NEL PARAGRAFO 2.3, PER INTERROMPERE IL PROGRAMMA OCCORRE DIGITARE CTRL+C

ESERCIZIO 3 – IL PROBLEMA DEL "SEARCH-INSERT-DELETE"

3.1 Descrizione del problema

Ci sono tre tipologie di thread operanti su una *Singly-Linked List* (assunta essere una lista di numeri interi):

- Thread di tipo **searcher** che esamina la lista (legge il suo contenuto);
- Thread di tipo **inserter** che inserisce un elemento in coda alla lista;
- Thread di tipo **deleter** che elimina un elemento dalla lista (in una qualsiasi posizione).

Tuttavia, occorre rispettare i seguenti vincoli di concorrenza:

1. Un qualsiasi numero di thread di tipo **searcher** può operare **contemporaneamente**;
2. Soltanto un thread di tipo **inserter** per volta può modificare la lista (ossia aggiungere un elemento in coda) → gli **inserter** dovranno essere esclusi mutualmente;
3. Un **inserter** può operare **contemporaneamente** con un qualsiasi numero di **searcher**;
4. Soltanto un thread di tipo **deleter** per volta può modificare la lista (ossia eliminare un elemento) → i **deleter** dovranno essere esclusi mutualmente;
5. Un **deleter** NON può operare **contemporaneamente** con nessun altro tipo di thread, dunque, quando opera un **deleter**, esso sarà l'unico thread in esecuzione.

Il testo del problema sarebbe finito qui tuttavia, la soluzione proposta nel libro da cui sono stati recuperati gli esercizi, causa *starvation* per i thread di tipo deleter; non a caso, vedendo le specifiche, questi sono i thread più vincolati ad operare in quanto **non solo si devono escludere mutualmente tra loro** (come nel caso dei thread di tipo inserter) ma fino a che un thread di altro tipo (dunque searcher e/o inserter) opera, questi **non possono andare in esecuzione**. La risoluzione a questo problema è spiegata nel paragrafo 3.2.

3.2 Approccio per risolvere i vincoli di concorrenza

I vincoli relativi a questo problema sono stati specificati nel paragrafo 3.1. Per risolverli è stato utilizzato il meccanismo dei SEMAFORI. I semafori utilizzati sono illustrati di seguito:

- S_SEARCHER: Per regolare l'esecuzione dei thread di tipo searcher;

- S_INSERTER: Per regolare l'esecuzione dei thread di tipo inserter;
- S_DELETER: Per regolare l'esecuzione dei thread di tipo deleter.

Tutti questi sono semafori **privati** e per rispettare le varie specifiche sono state introdotte anche alcune variabili globali:

- l: (Lista) La *Singly-Linked List* condivisa tra i thread;
- num_elementi: (int) Serve a tenere traccia del numero di elementi presenti nella lista;
- searcher_attivi: (int) Serve per contare il numero di searcher attivi;
- searcher_bloccati: (int) Serve per contare il numero di searcher bloccati;
- inserter_attivo: (bool) Serve per verificare se vi è un inserter attivo;
- inserter_bloccati: (int) Serve per contare il numero di inserter bloccati;
- deleter_attivo: (bool) Serve per verificare se vi è un deleter attivo;
- deleter_bloccati: (int) Serve per contare il numero di deleter bloccati;
- deleter_trigger: (bool) Serve per evitare la starvation dei deleter; questa variabile sarà usata come condizione per evitare l'esecuzione dei thread di tipo searcher. Nel paragrafo 3.3 vi è riportato il suo utilizzo.

La definizione del tipo *Lista* è la seguente:

```
typedef struct nodo {
      int element;
      struct nodo* next;
} Nodo;
typedef Nodo* Lista;
```

Ossia una Lista non è altro che un puntatore di Nodi in cui ogni nodo contiene:

- Un *elemento* (assunto intero);
- Il puntatore al prossimo nodo della lista.

I motivi per cui sono state utilizzate le variabili precedentemente elencate sono molteplici:

1) *searcher_attivi* consente di tenere traccia del numero di searcher in esecuzione e dunque, per il vincolo 5, fino a che questa variabile assume un valore **maggiore di 1** l'esecuzione dei deleter è interrotta;

2) *inserter_attivo* consente di verificare se un inserter è in esecuzione e dunque, in tal caso, rispettivamente per i vincoli 2 e 5, l'esecuzione di un altro inserter è interrotta come quella di un deleter;

3) *deleter_attivo* consente di verificare se un inserter è in esecuzione e dunque, in tal caso, per il vincolo 5, l'esecuzione di un **qualsiasi altro thread** (sia esso searcher, inserter, deleter) è interrotta;

4) Le variabili *searcher_bloccati*, *inserter_bloccati* e *deleter_bloccati* consentono di effettuare delle signal **mirate** a quei thread in attesa di eseguire.

3.3 Parti rilevanti del codice

Questo programma richiede l'interazione con delle *Singly-Linked List*; per rendere il codice più leggibile sono stati realizzati due file:

1) Un file header con la definizione del tipo lista e di tutte le funzioni necessarie per "*lavorare*" con una lista;

2) Un file contenente l'implementazione di tali funzioni.

Le funzioni appena menzionate sono tre e permettono di:

1) Analizzare la lista → *esamina_lista*;

2) Aggiungere un elemento in coda alla lista → *aggiungi_elemento*;

3) Eliminare un elemento in una certa posizione della lista → *elimina_elemento*.

Si riporta il codice delle funzioni appena menzionate. La funzione *esamina_lista* è la seguente:

```
void esamina_lista(Lista L, int id) {
      while(L != NULL) {
            printf("Thread Searcher con indice %d ha letto l'elemento %d\n",
      id, L->element); ← id è l'indice del thread searcher!
            L = L->next;
      }
}
```

Funzione in cui si scorre tutta la lista *L* e, per ogni elemento presente al suo interno, se ne stampa il valore e l'indice del thread searcher per un output più rilevante.

Le parti più rilevanti della funzione *aggiungi_elemento* sono sotto riportate:

```
int aggiungi_elemento(Lista* L, int el, int id) {
      Lista tmp = *L;   /* Variabile per scorrere la lista */
```

```
	Nodo* nd = (Nodo*) malloc(sizeof(Nodo));
	nd->next = NULL;
	nd->element = el;
	if (nd == NULL) {
		perror(error);
		return -1;
	}
	if (tmp == NULL) {
		*L = nd;
	} else {
		while(tmp->next != NULL) {
			tmp = tmp->next;
		}
		tmp->next = nd;
	}
	return 0;
}
```

Come già descritto, questa funzione aggiunge in coda alla lista un nuovo nodo (il cui valore è memorizzato nel parametro *el*). Come prima cosa occorre allocare memoria per il nuovo nodo da aggiungere in coda alla lista; se si dovessero generare degli errori in questa fase, la funzione ritorna il valore -1 altrimenti:

- **Se la lista è vuota**: Il nodo appena creato corrisponde alla nuova lista;
- **Se la lista NON è vuota**: Si scorrono tutti i nodi della lista fino a che la variabile *next* di un nodo (che si ricorda essere **il puntatore al prossimo nodo della lista**) non è *NULL* (ossia fino a che non si è giunti all'ultimo nodo **effettivo** della lista). Successivamente si assegna il nodo precedentemente creato come *next* dell'attuale ultimo nodo **effettivo** della lista. In questo modo il nodo creato diventerà l'ultimo nodo **effettivo** della lista nonché la **coda** della lista stessa.

Le parti più rilevanti della funzione *elimina_elemento* sono sotto riportate:

```
void elimina_elemento(Lista* L, int pos, int id) {
	Lista prev = NULL;
	Lista temp = *L;
	int i = 0;
	if (temp == NULL)
		return;
	if (pos == 0) {
		*L = temp->next;
		free(temp);
		return;
	}
	for (int i = 0; temp != NULL && i < (pos - 1); i++)
		temp = temp->next;
	Lista next = temp->next->next;
	free(temp->next);
	temp->next = next;
```

```
}
```

Come precedentemente descritto, in questa funzione si elimina un elemento in certa posizione della lista (in cui la posizione dell'elemento da eliminare è memorizzata nel parametro *pos*). La posizione di un certo elemento nella lista è semanticamente uguale agli indici di un array e dunque il primo elemento è in posizione 0, il secondo è in posizione 1 e così via.

In questa funzione, come prima cosa, si controlla se la lista è vuota → In tal caso si termina l'esecuzione. In caso contrario:

- **Se la posizione dell'elemento da eliminare vale 0**: Si libera la memoria allocata per la testa della lista, si pone come nuova testa il prossimo elemento (se presente) e si termina l'esecuzione;
- **Se la posizione dell'elemento da eliminare NON vale 0**: Si scorre la lista fino all'elemento precedente a quello da eliminare[2], si assegna come *next* di questo elemento il *next* dell'elemento da eliminare, si libera la memoria dell'elemento da eliminare e si termina l'esecuzione.

Si descrive ora il codice per la risoluzione del problema. Nella programmazione concorrente, ogni thread opera secondo il seguente schema di esecuzione:

- Prologo;
- Sezione Critica;
- Epilogo.

In questo programma sono state rese esplicite queste fasi creando le funzioni di prologo ed epilogo per ogni tipologia di thread. Si elencano, dunque, le parti di codice più rilevanti (delle fasi sopra elencate) per ogni tipologia di thread a partire dai thread di tipo searcher:

```
/* Prologo Searcher */
pthread_mutex_lock(&mutex);
if (!deleter_attivo && !deleter_trigger) {
	searcher_attivi++;
	sem_post(&S_SEARCHER);
} else {
	searcher_bloccati++;
}
pthread_mutex_lock(&mutex);
```

[2] perché scorrendo fino all'elemento da eliminare si perderebbero le informazioni degli elementi successivi a quello da eliminare

```
sem_wait(&S_SEARCHER);
```

Nel prologo del searcher, come prima cosa, si controlla se può eseguire. L'unica condizione per non eseguire è se un thread deleter è in esecuzione (come da specifica) oppure se la variabile deleter_trigger (utilizzata per evitare starvation dei deleter) è attiva. Se almeno una delle seguenti condizioni è vera il thread incrementerà il numero di searcher bloccati e si mette in attesa sul semaforo S_SEARCHER; altrimenti si incrementa il numero di searcher attivi e il thread prosegue con l'esecuzione. La sezione critica del searcher è molto semplice e prevede semplicemente l'esecuzione della funzione *esamina_lista*.

L'epilogo di questa tipologia di thread è riportato di seguito:

```
/* Epilogo Searcher */
pthread_mutex_lock(&mutex);
searcher_attivi--;
if (searcher_attivi == 0) {
      if (deleter_bloccati > 0) {
            if (!inserter_attivo) {
                  deleter_trigger = false;
                  deleter_bloccati--;
                  deleter_attivo = true;
                  sem_post(&S_DELETER);
            } else {
                  deleter_trigger = true;
            }
      }
} else if (deleter_bloccati > 0) {
      deleter_trigger = true;
}
pthread_mutex_unlock(&mutex);
```

In cui, come prima cosa, si decrementa il numero di searcher attivi e successivamente si controlla se questo è l'ultimo searcher ad eseguire; qui vi possono essere due possibilità:

1) Nel caso in cui il searcher sia l'ultimo rimasto: si controlla se vi sono dei deleter bloccati, in tal caso, se non ci dovesse essere un insert in esecuzione, ne viene risvegliato uno, altrimenti si abilita il trigger per evitare starvation dei deleter! Si noti che nel risvegliare un deleter, il trigger appena menzionato viene disabilitato (posto a *false*) per evitare di dare troppa priorità ai deleter.

2) Nel caso in cui il searcher NON sia l'ultimo rimasto: Se ci dovessero essere dei deleter bloccati si abilita il trigger precedentemente descritto per evitare che ulteriori searcher vadano in esecuzione (e dunque per evitare starvation dei deleter).

Si riporta ora il prologo degli inserter:

```
/* Prologo Inserter */
pthread_mutex_lock(&mutex);
if (!(deleter_attivo) && !(inserter_attivo)) {
	inserter_attivo = true;
	sem_post(&S_INSERTER);
} else {
	inserter_bloccati++;
}
pthread_mutex_unlock(&mutex);
sem_wait(&S_INSERTER);
```

Questo codice è semanticamente molto simile al prologo del searcher, l'unica differenza è la condizione di esecuzione del thread; infatti, i thread di tipo inserter non potranno eseguire nel caso in cui ci sia **o** un deleter in esecuzione **o** un altro inserter in esecuzione; si noti che qui non si considera la variabile per evitare starvation dei thread deleter (deleter_trigger) in quanto se si considerasse, si provocherebbe una starvation degli inserter. Gli inserter hanno una sezione critica particolare e dunque risulta utile commentarla:

```
prologo_inserter(*ptr);
r = rand() % 1000;
ret = aggiungi_elemento(&l, r, *ptr);
epilogo_inserter(*ptr, ret);
```

Si nota che dopo il prologo, l'inserter genera un numero pseudo-casuale (assunto che vada da 0 a 999) per poi inserirlo nella lista. Come specificato all'inizio del seguente paragrafo, la funzione *aggiungi_elemento* ritorna un valore uguale a -1 nel caso in cui si generassero degli errori in fase di creazione dell'elemento da aggiungere; questo valore di ritorno è memorizzato nella variabile locale *ret*. L'ultima operazione del thread inserter è l'epilogo le cui parti rilevanti sono riportate di seguito:

```
/* Epilogo Inserter */
pthread_mutex_lock(&mutex);
if (ret != -1)
	num_elementi++;
inserter_attivo = false;
if (deleter_bloccati != 0) {
	if (searcher_attivi > 0)
		deleter_trigger = true;
	else {
		deleter_trigger = false;
		deleter_bloccati--;
		deleter_attivo = true;
		sem_post(&S_DELETER);
	}
} else if (inserter_bloccati > 0) {
	inserter_bloccati--;
	inserter_attivo = true;
	sem_post(&S_INSERTER);
}
pthread_mutex_unlock(&mutex);
```

Come prima cosa si controlla se l'aggiunta dell'elemento è andata a buon fine (ossia se *ret* assume il valore diverso da -1) in tal caso si incrementa il numero di elementi presenti nella lista. Successivamente, si pone il flag *inserter_attivo* a *false* per segnalare che, al momento, non vi è alcun thread inserter in esecuzione. Ora si presentano due possibilità:

1. **Se vi sono thread deleter bloccati**: Occorre controllare se, attualmente, vi è almeno un thread searcher in esecuzione; in tal caso **non è possibile risvegliare il thread deleter** perché si romperebbe il vincolo 5 descritto nel paragrafo 3.1 e dunque, in tal caso, si imposta a *true* la variabile *deleter_trigger* in modo tale da **sospendere** l'esecuzione di altri thread searcher (evitando starvation dei deleter); se invece non vi fosse alcun thread searcher in esecuzione si aggiornano tutte le variabili necessarie e si risveglia un deleter.
2. **Se non vi sono thread deleter bloccati**: Si controlla se vi è almeno un inserter bloccato e, in tal caso, se ne risveglia uno aggiornando tutte le variabili necessarie.

Ora resta soltanto da descrivere il thread di tipo **deleter** il cui prologo è riportato di seguito.

```
/* Prologo Deleter */
pthread_mutex_lock(&mutex);
if ((searcher_attivi == 0) && !(inserter_attivo) && !(deleter_attivo)) {
        deleter_attivo = true;
        deleter_trigger = false;
        sem_post(&S_DELETER);
} else {
        deleter_bloccati++;
}
pthread_mutex_unlock(&mutex);
sem_wait(&S_DELETER);
```

Questo codice è semanticamente molto simile al prologo del searcher e dell'inserter, l'unica differenza da questi è la condizione di esecuzione del thread; infatti, i thread di tipo deleter non potranno eseguire nel caso in cui ci sia **o** almeno un searcher in esecuzione **o** un inserter in esecuzione **o** un altro deleter in esecuzione (ossia se c'è almeno un thread in esecuzione). Un'altra considerazione da far notare è che, quando un deleter va in esecuzione, si resetta la variabile *deleter_trigger* (il cui utilizzo è stato descritto già diverse volte) per evitare di dare troppa priorità ai thread deleter.

Anche questa tipologia di thread, come per gli inserter, presenta una sezione critica particolare ed è riportata in seguito:

```
prologo_deleter(*ptr);
bool elemento_eliminato = false;
pthread_mutex_lock(&mutex);
if (num_elementi > 0) {
```

```
        r = rand() % num_elementi;
        elimina_elemento(&l, r, *ptr);
        elemento_eliminato = true;
}
epilogo_deleter(*ptr, elemento_eliminato);
```

Dal momento che il generico thread deleter ha il compito di eliminare un elemento dalla lista, occorre estrarre l'indice dell'elemento da eliminare che, per semplicità, è stato scelto in maniera pseudo-casuale; tuttavia, c'è una particolarità da sottolineare; l'istruzione per estrarre l'indice dell'elemento da eliminare è la seguente:

```
r = rand() % num_elementi;
```

e dunque *r* assumerà un valore che varia da 0 a *num_elementi – 1* (si ricorda che la variabile globale *num_elementi* memorizza il numero di elementi presenti nella lista) se però *num_elementi* valesse 0 (ossia se la lista fosse **vuota**) si genererebbe un *floating point exception* e dunque, per risolvere questo problema, prima di eseguire la funzione *elimina_elemento* si controlla che la lista **non sia vuota**. La variabile locale booleana *elemento_eliminato* consente di verificare se c'è bisogno di aggiornare il numero di elementi presenti nella lista (ossia se bisogna decrementare di uno questo valore) in tal caso, come mostrato in seguito, l'aggiornamento della variabile *num_elementi* avverrà nel prologo del deleter. Un'ultima particolarità rispetto alle altre tipologie di thread è che il deleter blocca il mutex **prima** di eseguire il proprio prologo, questo perché si accede alla variabile globale *num_elementi*.

L'ultima funzione da descrivere è l'epilogo del deleter il quale è riportato di seguito:

```
/* Epilogo Deleter */
→ Nota: Il mutex è stato preso nella sezione critica del thread deleter
if (elemento_eliminato)
        num_elementi--;
deleter_attivo = false;

if (deleter_bloccati > 0) {
        deleter_bloccati--;
        deleter_trigger = false;
        deleter_attivo = true;
        sem_post(&S_DELETER);
} else if ((searcher_bloccati > 0) || (inserter_bloccati > 0)) {
        while (searcher_bloccati > 0) {
                /* Sveglio TUTTI i searcher bloccati (se presenti) */
                searcher_bloccati--;
                searcher_attivi++;
                sem_post(&S_SEARCHER);
        }
        if (inserter_bloccati > 0) {
                inserter_bloccati--;
                inserter_attivo = true;
                sem_post(&S_INSERTER);
        }
```

```
}
pthread_mutex_unlock(&mutex);
```

Funzione nella quale come prima cosa viene controllato se un elemento è stato eliminato e dunque se occorre decrementare il numero di elementi nella lista (come descritto poco fa quando si parlava della sezione critica) in seguito, dopo aver resettato la variabile globale booleana che indica se vi è un deleter in esecuzione (ossia *deleter_attivo*), vi si possono presentare due scenari:

1) **Se vi è un deleter bloccato**: Se ne risveglia uno (andando ad aggiornare tutte le variabili necessarie);
2) **Se non vi è alcun deleter bloccato**: Si controlla se vi è un searcher **o** un inserter bloccato; in tal caso si svolgono le seguenti operazioni:
    - Si risvegliano **tutti** i thread searcher (se presenti);
    - Si risveglia **un solo** thread inserter (se presente).

L'ultima (seppur minimale) modifica effettuata è nella generazione dei thread la quale è riportata qui sotto

```
r = rand() % 3;
if(r == 0) {
      ➔ Qui si genera il thread searcher ←
}else if (r == 1) {
      ➔ Qui si genera il thread inserter ←
} else
      ➔ Qui si genera il thread deleter ←
```

Si nota che prima di generare un thread si estrae un numero pseudo-casuale da 0 a 2 in cui:

- Se vale 0 si genera un thread di tipo searcher;
- Se vale 1 si genera un thread di tipo inserter;
- Se vale 2 si genera un thread di tipo deleter.

In modo tale da avere un comportamento aleatorio del programma

3.4 Commenti su un'esecuzione del programma

In questo capitolo si riporta una rilevante invocazione dell'esercizio appena descritto. L'invocazione è la seguente:

```
$ ./search_insert_delete 10
```

E il risultato è il seguente: (← si noti che sono stati rimossi i messaggi di creazione dei thread di tipo rider per evitare di occupare un numero eccessivo di pagine)

Thread Searcher di indice 0 partito: Ho come identificatore 139673168058112

Thread Searcher di indice 4 partito: Ho come identificatore 139673134487296

Thread Inserter di indice 2 partito: Ho come identificatore 139673151272704

Thread Inserter di indice 6 partito: Ho come identificatore 139673117701888

Thread Inserter di indice 1 partito: Ho come identificatore 139673159665408

Thread Inserter di indice 8 partito: Ho come identificatore 139673100916480

Il thread Searcher di indice 0 con identificatore 139673168058112 e' nella sua sezione critica

Il thread Searcher di indice 0 con identificatore 139673168058112 terminato. 0 rimanenti

Il primo thread che parte è un searcher ma la lista è ancora vuota → non si visualizza nulla.

Il thread Inserter di indice 2 con identificatore 139673151272704 e' nella sua sezione critica

Thread Inserter con indice 2 ha aggiunto l'elemento 652

Il thread Inserter di indice 6 con identificatore 139673117701888 e' bloccato a causa di inserter attivo

Il thread Inserter di indice 1 con identificatore 139673159665408 e' bloccato a causa di inserter attivo

Il thread Inserter di indice 8 con identificatore 139673100916480 e' bloccato a causa di inserter attivo

Provano ad eseguire tre thread inserter (di indice 6, 1, 8) contemportaneamente ma il thread (sempre inserter) di indice 2 non ha ancora terminato!!!!

Il thread Searcher di indice 4 con identificatore 139673134487296 e' nella sua sezione critica

Thread Searcher con indice 4 ha letto l'elemento 652

Il thread Searcher di indice 4 con identificatore 139673134487296 terminato. 0 rimanenti

Nel mentre esegue il searcher di indice 4 e riesce, fortunatamente, a leggere il valore inserito dal thread inserter di indice 2

Il thread Inserter di indice 2 con identificatore 139673151272704 terminato.

Thread Inserter con indice 6 ha aggiunto l'elemento 596

Il thread Inserter di indice 6 con identificatore 139673117701888 terminato.

Thread Inserter con indice 1 ha aggiunto l'elemento 265

Il thread Inserter di indice 1 con identificatore 139673159665408 terminato.

Thread Inserter con indice 8 ha aggiunto l'elemento 962

Il thread Inserter di indice 8 con identificatore 139673100916480 terminato.

I tre thread inserter (di indice 6,1,8) riescono ad eseguire dopo che il thread inserter di indice 2 ha terminato l'esecuzione.

Thread Searcher di indice 3 partito: Ho come identificatore 139673142880000

Il thread Searcher di indice 3 con identificatore 139673142880000 e' nella sua sezione critica

Thread Searcher con indice 3 ha letto l'elemento 652

Thread Inserter di indice 5 partito: Ho come identificatore 139673126094592

Il thread Inserter di indice 5 con identificatore 139673126094592 e' nella sua sezione critica

Thread Searcher con indice 3 ha letto l'elemento 596

Thread Searcher con indice 3 ha letto l'elemento 265

In questo punto riescono ad eseguire contemporaneamente sia il thread searcher di indice 3 sia il thread inserter di indice 5. (Il searcher riesce anche a leggere il valore 161 inserito dal thread inserter) che, come richiesto, è stato inserito in coda.

Thread Searcher con indice 3 ha letto l'elemento 962

Thread Searcher con indice 3 ha letto l'elemento 161

Thread Searcher di indice 9 partito: Ho come identificatore 139673092523776

Thread Deleter di indice 7 partito: Ho come identificatore 139673109309184

Il thread Deleter di indice 7 con identificatore 139673109309184 e' bloccato a causa di inserter attivo

Thread Inserter con indice 5 ha aggiunto l'elemento 161

Il thread Inserter di indice 5 con identificatore 139673126094592 terminato.

Il thread Searcher di indice 3 con identificatore 139673142880000 terminato. 0 rimanenti

Il thread Searcher di indice 9 con identificatore 139673092523776 e' bloccato a causa di deleter trigger

Thread Deleter con indice 7 ha rimosso l'elemento 265

Il thread Deleter di indice 7 con identificatore 139673109309184 terminato.

Il thread Searcher di indice 9 con identificatore 139673092523776 e' nella sua sezione critica

Thread Searcher con indice 9 ha letto l'elemento 652

Thread Searcher con indice 9 ha letto l'elemento 596

Thread Searcher con indice 9 ha letto l'elemento 962

Thread Searcher con indice 9 ha letto l'elemento 161

Il thread Searcher di indice 9 con identificatore 139673092523776 terminato. 0 rimanenti

Questa parte è molto significativa; il deleter di indice 7 si blocca perché ancora, a questo punto, l'inserter di indice 5 non ha terminato. Quando quest'ultimo termina vede che ancora il searcher di indice 3 non ha terminato l'esecuzione → si abilita il trigger. Subito dopo prova a partire il thread searcher di indice 9 che però non partirà in quanto il trigger è abilitato! Questo esempio mostra molto bene come si è evitata la starvation dei deleter.

Succesivamente esegue il deleter che rimuove l'elemento 265

E infine il searcher precedentemente bloccato legge gli elementi rimanenti.

ESERCIZIO 4 – IL PROBLEMA DEL BARBIERE DI HILZER

4.1 Descrizione del problema

Questo problema si può vedere come una versione molto più complessa del problema del *Barbiere Addormentato*.

Ci sono due tipologie di thread:

- Thread di tipo barbiere che taglia i capelli ai clienti;
- Thread di tipo cliente.

Di cui tre thread barbieri ed un numero *n* (con $n > 0$) di thread clienti. Il problema è il seguente:

In una barbieria (che per semplicità verrà denotata anche come "negozio") ci sono **tre** sedie, **tre** barbieri e **una** sala d'attesa nella quale vi è:

- Un divano che può accomodare **quattro** clienti;
- Un'ulteriore sala in cui i clienti **aspettano in piedi**.

Per motivi di sicurezza, il numero **massimo** di clienti all'interno del negozio è limitato a 20.

Un cliente non entra nel negozio se la capacità di 20 persone è stata colmata. Una volta all'interno, il cliente si siede nel divano se ci dovesse essere un posto libero nel divano altrimenti sta in piedi. Quando un barbiere è libero, il cliente che è seduto da più tempo nel divano viene servito e, se ci dovesse essere almeno un cliente in piedi, colui che è in piedi da più tempo occupa il posto del cliente che ha appena liberato il divano. Quando il taglio di capelli di un cliente è terminato, un qualsiasi barbiere accetta il pagamento, tuttavia, dal momento che vi è un solo registro di cassa, il pagamento è accettato **per un cliente alla volta**. I barbieri dividono dunque il loro tra le seguenti attività:

- Tagliare i capelli;
- Accettare i pagamenti;
- Dormire in attesa di un cliente.

Le funzioni da realizzare sono le seguenti:

- I clienti invocano **in ordine** le seguenti funzioni: *enterShop, sitOnSofa, getHairCut, pay*;

- I barbieri invocano **in ordine** le seguenti funzioni: *cutHair* e *acceptPayment;*

Mentre vincoli da soddisfare sono i seguenti:

1. I clienti non possono invocare la funzione *enterShop* se il numero dei clienti al suo interno supera la capacità di 20 clienti;
2. Se non vi è alcun posto libero nel divano il cliente NON può invocare la funzione *sitOnSofa*;
3. Quando si è liberato un posto nel divano si siede la persona in piedi da più tempo (se presente);
4. Quando un barbiere è libero viene servito il cliente seduto nel divano da più tempo (se presente);
5. Quando un cliente invoca *getHairCut* ci dovrebbe essere un barbiere che invoca *cutHair* concorrentemente e viceversa;
6. Dovrebbe essere possibile eseguire fino a tre clienti di eseguire la funzione *getHairCut* "allo stesso tempo", e fino a tre barbieri di eseguire *cutHair* "allo stesso tempo";
7. Il cliente deve invocare la funzione *pay* prima che il barbiere possa invocare la funzione *acceptPayment*;
8. Il barbiere deve invocare la funzione *acceptPayment* prima che il cliente possa uscire dal negozio.

4.2 Approccio per risolvere i vincoli di concorrenza

I vincoli relativi a questo problema sono stati specificati nel paragrafo 4.1. Per risolverli è stato utilizzato il meccanismo dei SEMAFORI. I semafori utilizzati sono illustrati di seguito:

- S_CLIENTI_ENTRANTI: Dal momento che il testo cita: "*Un cliente non entra nel negozio se la capacità di 20 persone è stata colmata*" ma non dice che quello specifico cliente **esce dal negozio**, è stato creato questo semaforo per *regolare* il numero di clienti nel negozio e dunque, nel caso in cui il numero di clienti totali sia maggiore di 20, i clienti in eccesso rimarranno in attesa su questo semaforo (il suo valore di inizializzazione è dunque uguale alla capacità consentita ossia 20). Si noti che questo semaforo non è presente nella soluzione originaria del problema;
- S_CLIENTE_TO_BARBIERE_ENTRATO: Semaforo privato utilizzato per segnalare al/ai barbiere/i che un cliente è entrato nel negozio (sveglia dunque un barbiere);

- S_DIVANO: Semaforo utilizzato per far sì che il numero massimo di clienti nel divano sia quattro (e dunque, il suo valore di inizializzazione è quattro);
- *S_DIVANO_OCCUPATO: Array di semafori privati utilizzati per gestire l'ordine dei clienti in piedi che attendono di sedersi sul divano. Il modo per gestire l'ordine è specificato dopo il seguente elenco;
- *S_CLIENTE_TO_BARBIERE_SERVITO: Questo è un array di semafori privati il cui utilizzo è particolare e verrà spiegato nel paragrafo 4.3;
- *S_CLIENTI_SERVITI: Questo è un array di semafori privati per gestire l'ordine dei clienti nel divano che attendono di essere serviti. Il modo per gestire l'ordine è specificato dopo il seguente elenco;
- S_BARBIERE_TO_CLIENTE: Semaforo privato utilizzato per segnalare al cliente che il taglio di capelli è terminato;
- S_ACCEDI_CASSA: Semaforo per regolare l'accesso dei barbieri al registratore di cassa (dunque, essendoci un solo registratore, questo semaforo è inizializzato a uno);
- S_CLIENTE_PAGA: Semaforo privato utilizzato per segnalare al barbiere che il cliente è pronto a pagare;
- S_CONTO_SALDATO: Semaforo privato utilizzato per segnalare al cliente che il pagamento è stato accettato e che dunque può uscire dal negozio.

Una delle parti più complesse del programma è sicuramente la gestione degli array di semafori. Si nota infatti che per i vincoli 3 e 4 occorre, rispettivamente, fare in modo di:

- Far sedere nel divano la persona in piedi da più tempo.
- Servire il cliente seduto nel divano da più tempo;

Per soddisfare questi vincoli è infatti necessario effettuare del *wait* e delle *signal* **mirate** in modo tale da bloccare e risvegliare i clienti in maniera **selettiva**. Gli array di semafori S_DIVANO_OCCUPATO ed S_CLIENTI_SERVITI sono stati creati proprio per questo motivo; il numero di semafori per ciascuna variabile è uguale al numero di thread di tipo cliente, e dunque ognuno di questi andrà ad effettuare una *wait* sul semaforo corrispondente **al proprio indice** e, una volta che è giunto il suo turno (dunque **o** di sedersi nel divano **o** di essere servito) verrà fatta una

signal utilizzando **l'indice del cliente**. Tutto questo è necessario dal momento che la funzione *sem_post* su un semaforo non preserva l'ordine in cui sono state effettuale le *wait* e dunque, al fine di rispettare le specifiche richieste dal problema, non è possibile realizzare una soluzione **corretta** senza l'utilizzo di questi array di semafori.

Non è difficile notare che rimane ancora una problematica: Come capire chi è il cliente in attesa da più tempo? Per far sì di memorizzare i clienti in piedi e seduti nel divano da più tempo, sono stati utilizzati due array dinamici, la cui definizione è riportata di seguito:

- int *clienti_in_piedi: utilizzato per memorizzare gli indici dei clienti attualmente in piedi;
- int *clienti_divano: utilizzato per memorizzare gli indici dei clienti attualmente seduti nel divano.

Gestendo questi array con una politica FIFO si riesce a garantire che il cliente in attesa da **più tempo** (first-in) è anche il cliente che sarà servito **per primo** (first-out). Questi array hanno anche una dimensione specifica:

- L'array *clienti_in_piedi* ha dimensione pari al numero **massimo** di clienti che possono essere *contemporaneamente* nel negozio, ossia 20;
- L'array *clienti_divano* ha dimensione pari al numero **massimo** di clienti che possono essere seduti *contemporaneamente* nel divano, ossia 4.

Infine, sono state specificate delle costanti per rendere il codice più leggibile e versatile:

- *NUM_CLIENTI_MAX:* che assume il valore massimo di clienti ammessi nel negozio → 20;
- *NUM_POSTI_DIVANO*: che assume il valore di posti nel divano → 4;
- *NUM_BARBIERI:* che assume come valore il numero di barbieri nonché il numero massimo di clienti che possono essere serviti concorrentemente → 3;
- *NUM_REGISTRI_CASSA:* che assume il valore del numero dei registri di cassa → 1.

4.3 Parti rilevanti del codice

Una volta introdotte le variabili utilizzate per la risoluzione di questo esercizio, non resta che descriverne le (tante) parti rilevanti.

Nel paragrafo 4.2 sono stati definiti gli array per memorizzare gli indici dei clienti *in piedi* e *seduti*; se ne riporta il codice di inizializzazione:

```
for (i=0;i < NUM_CLIENTI_MAX; i++) {
      clienti_in_piedi[i] = -1;
}
for (i=0;i < NUM_POSTI_DIVANO; i++) {
      clienti_divano[i] = -1;
}
```

Dunque, i valori che possono memorizzare questi array sono:

- **O** l'indice di un thread di tipo cliente;
- **O** -1 nel caso in cui non ci siano clienti a sufficienza a colmare l'array per intero.

Dal momento che nel seguente programma si effettuano tante operazioni su questi array, è stato deciso di realizzare due file:

1) Un file header con la definizione di tutte le funzioni utili per l'uso di questi array;
2) Un file contenente l'implementazione di tali funzioni.

Con il fine di rendere il codice più leggibile.

Le funzioni appena menzionate sono tre e permettono di:

1. Stampare l'array (utile esclusivamente a scopo di debug) → *printArray*;
2. Recuperare l'indice della prima posizione libera in un array→ *getFirstFreePosition*;
3. Recuperare l'indice della prima posizione NON libera in un array → *getFirstNonFreePosition*;

Si riporta il codice delle ultime due funzioni tra quelle sopra menzionate (in quanto *printArray* non risulta significativa per il funzionamento del programma). La funzione *getFirstFreePosition* è la seguente:

```
int getFirstFreePosition(int* v, int n) {
      for (int i=0;i < n; i++) {
            if (v[i] == -1) {
                  return i;
            }
      }
      return -1;
}
```

Dal momento che una posizione libera, come anche specificato all'inizio del seguente paragrafo, è definita da un **-1**, quando scorrendo un certo array (sia esso *clienti_in_piedi* o *clienti_divano*) si trova questa occorrenza, viene ritornato l'indice del primo elemento contenente questo valore. Nel caso in cui non ci fosse nemmeno una posizione libera viene ritornato il valore -1.

La funzione *getFirstNonFreePosition* svolge il compito duale della funzione appena descritta:

```
int getFirstNonFreePosition(int* v, int n) {
	for (int i=0;i < n; i++) {
		if (v[i] != -1) {
			return i;
		}
	}
	return -1;
}
```

Ora che sono state descritte queste funzioni "*di supporto*" è possibile descrivere il codice della soluzione del seguente esercizio.

Si riportano le parti di codice rilevanti del thread di tipo cliente:

```
sem_wait(&S_CLIENTI_ENTRANTI);
pthread_mutex_lock(&mutex);
enterShop(*ptr);
```

Prima di entrare nel negozio, il cliente si mette in attesa sul semaforo S_CLIENTI_ENTRANTI (che si ricorda essere inizializzato a 20); se la capacità del negozio non è stata colmata il cliente avrà modo di eseguire. Superato questo "controllo" il cliente invoca la funzione *enterShop* (→ vincolo 1 soddisfatto) la quale svolge le seguenti istruzioni:

```
int i = getFirstFreePosition(clienti_in_piedi, NUM_CLIENTI_MAX);
clienti_in_piedi[i] = indice;
```

In cui si recupera l'indice della prima posizione libera dell'array dei clienti in piedi per poi **accodare** il cliente in tale array nella posizione restituita dalla funzione *getFirstFreePosition*. Si noti, che non c'è bisogno di controllare se l'array appena menzionato sia pieno in quanto prima di eseguire la funzione appena descritta, il cliente ha effettuato una *wait* sul semaforo S_CLIENTI_ENTRANTI il quale, non a caso, è inizializzato a 20 che è anche la dimensione dell'array *clienti_in_piedi*.

Successivamente, il cliente svolge le seguenti istruzioni:

```
pthread_mutex_unlock(&mutex);
sem_post(&S_CLIENTE_TO_BARBIERE_ENTRATO);
sem_wait(&S_DIVANO_OCCUPATO[(*ptr)]);
sem_wait(&S_DIVANO);
```

in cui il cliente segnala al barbiere che è arrivato attraverso il semaforo S_CLIENTE_TO_BARBIERE_ENTRATO per poi attendere sul primo dei due array di semafori privati, ossia S_DIVANO_OCCUPATO. Il cliente si mette in attesa sul semaforo nella posizione associata al proprio indice (ad es. se il cliente ha indice 0 esso si metterà in attesa su S_DIVANO_OCCUPATO[0] e così via). Successivamente, aspetta che si liberi un posto nel divano

ossia si mette in attesa sul semaforo S_DIVANO che si ricorda essere inizializzato a 4 ossia il numero **massimo** di clienti seduti nel divano contemporaneamente. Superato quest'altro "controllo" sarà giunto il turno del cliente di sedersi nel divano e dunque potrà eseguire la funzione *sitOnSofa* (→ vincolo 2 soddisfatto)[3] il cui funzionamento è molto simile a quello della funzione *enterShop* con l'unica differenza che il cliente si **accoda** nell'array *clienti_divano* come mostrato di seguito:

```
i = getFirstFreePosition(clienti_divano, NUM_POSTI_DIVANO);
clienti_divano[i] = indice;
```

A questo punto vi è una situazione molto particolare; la prossima istruzione eseguita dal cliente è la seguente:

```
sem_post(&S_CLIENTE_TO_BARBIERE_SERVITO[(*ptr)]);
```

Il semaforo S_CLIENTE_TO_BARBIERE_SERVITO, il cui utilizzo non è stato descritto nel paragrafo 4.2, viene utilizzato come strumento di sincronizzazione tra cliente e barbiere, infatti, senza questo semaforo, c'è il rischio che il barbiere vada a servire un cliente che non si è ancora seduto nel divano e che dunque **non è pronto ad essere servito** infrangendo il vincolo 6.

```
sem_wait(&S_CLIENTI_SERVITI[(*ptr)]);
```

In seguito, il cliente si mette in attesa nell'array di semafori S_CLIENTI_SERVITI in attesa, appunto, di essere servito. Una volta giunto il turno del cliente, esso invocherà la funzione *getHairCut* nella quale si svolge una semplice *wait*:

```
sem_wait(&S_BARBIERE_TO_CLIENTE);
```

con la quale il cliente aspetta che il taglio di capelli sia terminato. Prima di uscire il cliente deve pagare il conto, e proprio per questo invoca l'ultima funzione richiesta, ossia *pay* le cui parti rilevanti sono riportate di seguito:

```
sem_post(&S_CLIENTE_PAGA);
sem_wait(&S_CONTO_SALDATO);
```

Si nota che il cliente, come prima, cosa segnala al barbiere che è pronto a pagare (effettuando una signal sul semaforo S_CLIENTE_PAGA), successivamente si mette in attesa sul semaforo S_CONTO_SALDATO aspettando che il barbiere accetti il pagamento → vincolo 7 soddisfatto).

[3] Un difetto di questa soluzione è che la "*risorsa*" divano è sottoutilizzata dal momento che i clienti si possono sedere nel divano soltanto quando risvegliati da un barbiere e, dal momento che i barbieri sono tre e nel divano ci sono 4 posti, questi non verrebbero sfruttati totalmente. Come mostrato nella soluzione nel PDF ci sarebbe bisogno di un thread "*usher*" che assegna i posti nel divano ai clienti

Dopo che il pagamento è stato accettato, il cliente incrementa il valore del semaforo S_CLIENTI_ENTRANTI (come mostrato di seguito):

```
sem_post(&S_CLIENTI_ENTRANTI);
```

in modo tale da consentire ad un eventuale cliente in attesa di entrare nel negozio (e che dunque era bloccato a causa della capacità colmata); e in seguito, esce dal negozio (→ vincolo 8 soddisfatto).

Si riportano le parti di codice rilevanti del thread di tipo barbiere (che opera in un ciclo infinito):

```
sem_wait(&S_CLIENTE_TO_BARBIERE_ENTRATO);
```

Il Barbierie, come prima cosa, si addormenta in attesa che un cliente lo svegli.

```
pthread_mutex_lock(&mutex);
i = getFirstNonFreePosition(clienti_in_piedi, NUM_CLIENTI_MAX);
```

Una volta sveglio, il barbiere recupera l'indice della persona in piedi da più tempo; si noti che non c'è bisogno di controllare l'indice ritornato dalla funzione *getFirstNonFreePosition* in quanto, a questo punto del programma, il cliente ha invocato **sicuramente** la funzione *enterShop* e dunque si sarà già accodato nell'array *clienti_in_piedi (*e quindi vi è sicuramente una posizione NON libera nell'array appena menzionato). Successivamente, occorre liberare il posto occupato dal cliente nel seguente array per garantire il corretto funzionamento del programma. Per liberare il posto si effettua uno shift all'indietro dell'array *clienti_in_piedi* e il codice per effettuare questa operazione è il seguente:

```
next_cliente_divano = clienti_in_piedi[i]; ← si memorizza l'indice del cliente
for (j=i; j < (NUM_CLIENTI_MAX-1); j++) {
      clienti_in_piedi[j] = clienti_in_piedi[j+1];
}
clienti_in_piedi[j] = -1;
pthread_mutex_unlock(&mutex);
```

A questo punto si segnala il cliente in attesa sul semaforo S_DIVANO_OCCUPATO nella posizione associata al proprio indice: (si noti che, come specificato nel codice, l'indice del cliente è stato memorizzato nella variabile locale *next_cliente_divano*).

```
sem_post(&S_DIVANO_OCCUPATO[next_cliente_divano]);
```

A questo punto, prima di servire il cliente, bisogna aspettare che quest'ultimo si sieda nel divano e dunque il barbiere si mette in attesa sul semaforo S_CLIENTE_TO_BARBIERE_SERVITO (il cui utilizzo è stato descritto nel seguente paragrafo nella parte di codice del cliente).

```
sem_wait(&S_CLIENTE_TO_BARBIERE_SERVITO[next_cliente_divano]);
```

Successivamente, il barbiere segnala al cliente che il taglio di capelli può iniziare; come prima cosa si recupera l'indice del cliente seduto nel divano da più tempo:

```
pthread_mutex_lock(&mutex);
i = getFirstNonFreePosition(clienti_divano, NUM_POSTI_DIVANO);
next_cliente_da_servire = clienti_divano[i];
```

Una volta recuperato l'indice di tale cliente, e memorizzato nella variabile locale *next_cliente_da_servire*, si shifta all'indietro l'array dei clienti seduti nel divano (analogamente a quanto fatto sull'array delle persone in piedi):

```
for (j=i; j < (NUM_POSTI_DIVANO-1); j++) {
      clienti_divano[j] = clienti_divano[j+1];
}
clienti_divano[j] = -1;
pthread_mutex_unlock(&mutex);
```

A questo punto, dal momento che si è liberato un posto nel divano, si incrementa il valore del semaforo S_DIVANO (il cui utilizzo è stato descritto in questo paragrafo nel codice del cliente):

```
sem_post(&S_DIVANO);
```

Ora è possibile servire il cliente → il barbiere invoca la funzione *cutHair* il cui codice rilevante è il seguente:

```
void cutHair(int indice, int indice_cliente) {
      sleep(2);  ← Sleep usata per rallentare il flusso di esecuzione
      sem_post(&S_BARBIERE_TO_CLIENTE);
}
```

In cui viene fatta una signal sul semaforo S_BARBIERE_TO_CLIENTE per segnalare al cliente che il taglio di capelli è terminato.

L'ultima azione che deve svolgere il barbiere è accettare il pagamento. Come prima cosa si aspetta che il cliente abbia invocato la funzione *pay* (aspettando la signal del cliente sul semaforo S_CLIENTE_PAGA):

```
sem_wait(&S_CLIENTE_PAGA);
```

A questo punto il barbiere aspetta di poter accedere al registratore di cassa

```
sem_wait(&S_ACCEDI_CASSA);
```

Una volta che il barbiere "si trova" al registratore di cassa, può invocare la funzione *acceptPayment* la quale è riportata qui sotto:

```
void acceptPayment(int indice) {
      sem_post(&S_CONTO_SALDATO);
}
```

In questa funzione semplicemente si segnala al cliente che il pagamento è stato accettato e che dunque può uscire dal negozio. Dopo aver accettato il pagamento, il barbiere libera il registratore di cassa per il prossimo (eventuale) pagamento.

```
sem_post(&S_ACCEDI_CASSA);
```

4.4 Commenti su un'esecuzione del programma

In questo capitolo si riporta una rilevante invocazione dell'esercizio appena descritto. L'invocazione è la seguente:

```
$ ./hilzer_barbershop 10
```

E il risultato è il seguente: (← si noti che sono stati rimossi i messaggi di creazione dei thread di tipo rider per evitare di occupare un numero eccessivo di pagine. Inoltre sono state rimosse tutte (eccetto la prima) le stampe delle posizioni vuote degli array)

Il Thread CLIENTE di indice 0 con identificatore 140704462198528 e' entrato nel negozio.
Il Thread CLIENTE di indice 1 con identificatore 140704453805824 e' entrato nel negozio.
Il Thread CLIENTE di indice 2 con identificatore 140704311195392 e' entrato nel negozio.
Il Thread CLIENTE di indice 3 con identificatore 140704445413120 e' entrato nel negozio.
Il Thread CLIENTE di indice 4 con identificatore 140704437020416 e' entrato nel negozio.
Il Thread CLIENTE di indice 5 con identificatore 140704428627712 e' entrato nel negozio.
Il Thread CLIENTE di indice 7 con identificatore 140704336574208 e' entrato nel negozio.
Il Thread CLIENTE di indice 6 con identificatore 140704420235008 e' entrato nel negozio.
Il Thread CLIENTE di indice 8 con identificatore 140704328181504 e' entrato nel negozio.
Il Thread CLIENTE di indice 9 con identificatore 140704319788800 e' entrato nel negozio.

Tutti i clienti entrano nel negozio

Thread BARBIERE di indice 10 partito: Ho come identificatore 140704302802688
Stampo l'array: IN PIEDI
0: 0
1: 1
2: 2
3: 3
4: 4
5: 5
6: 7
7: 6

Appena parte il barbiere vengono stampati gli indici dei clienti in piedi in ordine di arrivo (viene dunque stampato l'array *clienti_in_piedi*). Questa sarà l'unica stampa completa, dalla prossima stampa verranno riportate soltanto le posizioni non libere.

Thread BARBIERE di indice 11 partito: Ho come identificatore 140704294409984
8: 8
9: 9
10: -1
11: -1

12: -1

13: -1

14: -1

15: -1

16: -1

17: -1

18: -1

19: -1

Stampo l'array: IN PIEDI

Thread BARBIERE di indice 12 partito: Ho come identificatore 140704286017280

Thread BARBIERE di indice 10 con identificatore 140704302802688 ha risvegliato il thread cliente di indice 0 CHE ERA IN PIEDI.

Il barbiere di indice 10 sveglia il cliente in piedi da più tempo ossia il cliente di indice 0

0: 1

1: 2

2: 3

3: 4

4: 5

5: 7

6: 6

7: 8

8: 9

9: -1

Il Thread CLIENTE di indice 0 con identificatore 140704462198528 si e' seduto nel divano. Il numero di clienti seduti e': 1

Successivamente il cliente di indice 0 si siede nel divano e attende di essere servito

Stampo l'array: DIVANO

0: 0

1: -1

2: -1

3: -1

Thread BARBIERE di indice 11 con identificatore 140704294409984 ha risvegliato il thread cliente di indice 1 CHE ERA IN PIEDI.

Il barbiere di indice 11 sveglia il prossimo cliente in piedi da più tempo ossia il cliente di indice 1

Stampo l'array: IN PIEDI

0: 2

1: 3

2: 4

3: 5

4: 7

5: 6

6: 8

7: 9

Qui si può notare che i clienti di indice 0 e 1 sono stato rimossi dall'array *clienti_in_piedi*

Thread BARBIERE di indice 10 partito con identificatore 140704302802688 ha risvegliato il thread di indice 0 CHE ERA NEL DIVANO.

Il Thread BARBIERE di indice 10 con identificatore 140704302802688 INIZIA il taglio di capelli per il cliente di indice: 0.

8: -1 ← Qui le stampe sono sovrapposte perché le stampe sopra riportate non sono eseguite in una sezione critica

Thread BARBIERE di indice 12 con identificatore 140704286017280 ha risvegliato il thread cliente di indice 2 CHE ERA IN PIEDI.

Il barbiere di indice 12 sveglia il prossimo thread cliente, ossia quello di indice 2 il quale, in seguito, si siede sul divano

Il Thread CLIENTE di indice 1 con identificatore 140704453805824 si e' seduto nel divano. Il numero di clienti seduti e': 1

Il Thread CLIENTE di indice 2 con identificatore 140704311195392 si e' seduto nel divano. Il numero di clienti seduti e': 2

Stampo l'array: DIVANO

0: 1

1: 2

2: -1

3: -1

Thread BARBIERE di indice 11 partito con identificatore 140704294409984 ha risvegliato il thread di indice 1 CHE ERA NEL DIVANO.

Il Thread BARBIERE di indice 11 con identificatore 140704294409984 INIZIA il taglio di capelli per il cliente di indice: 1.

Stampo l'array: DIVANO

0: 2

1: -1

2: -1

3: -1

Thread BARBIERE di indice 12 partito con identificatore 140704286017280 ha risvegliato il thread di indice 2 CHE ERA NEL DIVANO.

Il Thread BARBIERE di indice 12 con identificatore 140704286017280 INIZIA il taglio di capelli per il cliente di indice: 2.

Il Thread BARBIERE di indice 11 con identificatore 140704294409984 HA TERMINATO il taglio di capelli per il cliente di indice: 1.

Il Thread BARBIERE di indice 10 con identificatore 140704302802688 HA TERMINATO il taglio di capelli per il cliente di indice: 0.

Il Thread BARBIERE di indice 12 con identificatore 140704286017280 HA TERMINATO il taglio di capelli per il cliente di indice: 2.

Vengono serviti i clienti precedentemente elencati

Taglio dei capelli per il Thread CLIENTE di indice 0 con identificatore 140704462198528 terminato.

Taglio dei capelli per il Thread CLIENTE di indice 2 con identificatore 140704311195392 terminato.

Il Thread CLIENTE di indice 0 con identificatore 140704462198528 ha pagato il conto.

Thread CLIENTE di indice 0 con identificatore 140704462198528 esce dal negozio.

Il cliente di indice 0 paga ed esce dal negozio

Stampo l'array: IN PIEDI

0: 3

1: 4

2: 5

3: 7

4: 6

5: 8

6: 9

7: -1

Il Thread CLIENTE di indice 2 con identificatore 140704311195392 ha pagato il conto.

Thread CLIENTE di indice 2 con identificatore 140704311195392 esce dal negozio.

Il cliente di indice 2 paga ed esce dal negozio

Taglio dei capelli per il Thread CLIENTE di indice 1 con identificatore 140704453805824 terminato.

Thread BARBIERE di indice 11 con identificatore 140704294409984 ha risvegliato il thread cliente di indice 3 CHE ERA IN PIEDI.

Il Thread CLIENTE di indice 1 con identificatore 140704453805824 ha pagato il conto.

Thread CLIENTE di indice 1 con identificatore 140704453805824 esce dal negozio.

Il cliente di indice 1 paga ed esce dal negozio

Stampo l'array: IN PIEDI

Dopo aver servito i clienti, si continua in questo modo per tutti i restanti 7 clienti. Dalle stampe si può notare come l'ordine di arrivo e l'ordine con cui i clienti vengono serviti permane!

0: 4

1: 5

2: 7

3: 6

4: 8

5: 9

6: -1

Stampo l'array: IN PIEDI

0: 5

1: 7

2: 6

3: 8

4: 9

5: -1

Thread BARBIERE di indice 10 con identificatore 140704302802688 ha risvegliato il thread cliente di indice 4 CHE ERA IN PIEDI.

Il Thread CLIENTE di indice 3 con identificatore 140704445413120 si e' seduto nel divano. Il numero di clienti seduti e': 1

Thread BARBIERE di indice 12 con identificatore 140704286017280 ha risvegliato il thread cliente di indice 5 CHE ERA IN PIEDI.

Stampo l'array: DIVANO

0: 3

1: -1

2: -1

3: -1

Il Thread CLIENTE di indice 4 con identificatore 140704437020416 si e' seduto nel divano. Il numero di clienti seduti e': 1

Thread BARBIERE di indice 11 partito con identificatore 140704294409984 ha risvegliato il thread di indice 3 CHE ERA NEL DIVANO.

Il Thread BARBIERE di indice 11 con identificatore 140704294409984 INIZIA il taglio di capelli per il cliente di indice: 3.

Il Thread CLIENTE di indice 5 con identificatore 140704428627712 si e' seduto nel divano. Il numero di clienti seduti e': 2

Stampo l'array: DIVANO

0: 4

1: 5

2: -1

3: -1

Stampo l'array: DIVANO

Thread BARBIERE di indice 10 partito con identificatore 140704302802688 ha risvegliato il thread di indice 4 CHE ERA NEL DIVANO.

Il Thread BARBIERE di indice 10 con identificatore 140704302802688 INIZIA il taglio di capelli per il cliente di indice: 4.

0: 5

1: -1

2: -1

3: -1

Thread BARBIERE di indice 12 partito con identificatore 140704286017280 ha risvegliato il thread di indice 5 CHE ERA NEL DIVANO.

Il Thread BARBIERE di indice 12 con identificatore 140704286017280 INIZIA il taglio di capelli per il cliente di indice: 5.

Il Thread BARBIERE di indice 11 con identificatore 140704294409984 HA TERMINATO il taglio di capelli per il cliente di indice: 3.

Il Thread BARBIERE di indice 10 con identificatore 140704302802688 HA TERMINATO il taglio di capelli per il cliente di indice: 4.

Il Thread BARBIERE di indice 12 con identificatore 140704286017280 HA TERMINATO il taglio di capelli per il cliente di indice: 5.

Taglio dei capelli per il Thread CLIENTE di indice 5 con identificatore 140704428627712 terminato.

Taglio dei capelli per il Thread CLIENTE di indice 4 con identificatore 140704437020416 terminato.

Taglio dei capelli per il Thread CLIENTE di indice 3 con identificatore 140704445413120 terminato.

Il Thread CLIENTE di indice 4 con identificatore 140704437020416 ha pagato il conto.

Thread CLIENTE di indice 4 con identificatore 140704437020416 esce dal negozio.

Stampo l'array: IN PIEDI

0: 7

1: 6

2: 8

3: 9

4: -1

5: -1

Il Thread CLIENTE di indice 3 con identificatore 140704445413120 ha pagato il conto.

Thread CLIENTE di indice 3 con identificatore 140704445413120 esce dal negozio.

Il Thread CLIENTE di indice 5 con identificatore 140704428627712 ha pagato il conto.

Thread CLIENTE di indice 5 con identificatore 140704428627712 esce dal negozio.

6: -1

Stampo l'array: IN PIEDI

0: 6

1: 8

Thread BARBIERE di indice 11 con identificatore 140704294409984 ha risvegliato il thread cliente di indice 7 CHE ERA IN PIEDI.

2: 9

3: -1

Stampo l'array: IN PIEDI

0: 8

1: 9

2: -1

Thread BARBIERE di indice 10 con identificatore 140704302802688 ha risvegliato il thread cliente di indice 6 CHE ERA IN PIEDI.

Il Thread CLIENTE di indice 7 con identificatore 140704336574208 si e' seduto nel divano. Il numero di clienti seduti e': 1

Thread BARBIERE di indice 12 con identificatore 140704286017280 ha risvegliato il thread cliente di indice 8 CHE ERA IN PIEDI.

Il Thread CLIENTE di indice 6 con identificatore 140704420235008 si e' seduto nel divano. Il numero di clienti seduti e': 2

Stampo l'array: DIVANO

0: 7

1: 6

2: -1

3: -1

Il Thread CLIENTE di indice 8 con identificatore 140704328181504 si e' seduto nel divano. Il numero di clienti seduti e': 2

Stampo l'array: DIVANO

0: 6

1: 8

2: -1

3: -1

Thread BARBIERE di indice 11 partito con identificatore 140704294409984 ha risvegliato il thread di indice 7 CHE ERA NEL DIVANO.

Il Thread BARBIERE di indice 11 con identificatore 140704294409984 INIZIA il taglio di capelli per il cliente di indice: 7.

Stampo l'array: DIVANO

0: 8

1: -1

2: -1

3: -1

Thread BARBIERE di indice 12 partito con identificatore 140704286017280 ha risvegliato il thread di indice 8 CHE ERA NEL DIVANO.

Il Thread BARBIERE di indice 12 con identificatore 140704286017280 INIZIA il taglio di capelli per il cliente di indice: 8.

Thread BARBIERE di indice 10 partito con identificatore 140704302802688 ha risvegliato il thread di indice 6 CHE ERA NEL DIVANO.

Il Thread BARBIERE di indice 10 con identificatore 140704302802688 INIZIA il taglio di capelli per il cliente di indice: 6.

Il Thread BARBIERE di indice 12 con identificatore 140704286017280 HA TERMINATO il taglio di capelli per il cliente di indice: 8.

Il Thread BARBIERE di indice 11 con identificatore 140704294409984 HA TERMINATO il taglio di capelli per il cliente di indice: 7.

Il Thread BARBIERE di indice 10 con identificatore 140704302802688 HA TERMINATO il taglio di capelli per il cliente di indice: 6.

Taglio dei capelli per il Thread CLIENTE di indice 7 con identificatore 140704336574208 terminato.

Taglio dei capelli per il Thread CLIENTE di indice 8 con identificatore 140704328181504 terminato.

Stampo l'array: IN PIEDI

0: 9

1: -1

Il Thread CLIENTE di indice 8 con identificatore 140704328181504 ha pagato il conto.

Il Thread CLIENTE di indice 9 con identificatore 140704319788800 si e' seduto nel divano. Il numero di clienti seduti e': 1

Il Thread CLIENTE di indice 7 con identificatore 140704336574208 ha pagato il conto.

Taglio dei capelli per il Thread CLIENTE di indice 6 con identificatore 140704420235008 terminato.

Thread CLIENTE di indice 8 con identificatore 140704328181504 esce dal negozio.

Il Thread CLIENTE di indice 6 con identificatore 140704420235008 ha pagato il conto.

Thread CLIENTE di indice 6 con identificatore 140704420235008 esce dal negozio.

Thread BARBIERE di indice 12 con identificatore 140704286017280 ha risvegliato il thread cliente di indice 9 CHE ERA IN PIEDI.

Stampo l'array: DIVANO

0: 9

1: -1

2: -1

3: -1

Thread CLIENTE di indice 7 con identificatore 140704336574208 esce dal negozio.

Thread BARBIERE di indice 12 partito con identificatore 140704286017280 ha risvegliato il thread di indice 9 CHE ERA NEL DIVANO.

Il Thread BARBIERE di indice 12 con identificatore 140704286017280 INIZIA il taglio di capelli per il cliente di indice: 9.

Il Thread BARBIERE di indice 12 con identificatore 140704286017280 HA TERMINATO il taglio di capelli per il cliente di indice: 9.

Taglio dei capelli per il Thread CLIENTE di indice 9 con identificatore 140704319788800 terminato.

Il Thread CLIENTE di indice 9 con identificatore 140704319788800 ha pagato il conto.

Thread CLIENTE di indice 9 con identificatore 140704319788800 esce dal negozio.

CONCLUSIONI

Nella seguente tesina sono stati proposti quattro esercizi inerenti alla programmazione concorrente. La risoluzione di ogni singolo esercizio ha portato all'utilizzo di diversi strumenti di sincronizzazione, a partire dalle condition variables dell'esercizio 1, per poi passare all'utilizzo semafori per gli esercizi 2, 3 e 4 in cui negli esercizi 2 e 4 vengono utilizzate delle code semaforiche.

Il codice di tutti i seguenti esercizi è stato mantenuto su un repository GitHub per una migliore organizzazione e gestione delle diverse parti del codice. Nell'appendice vi è il procedimento per scaricare il codice sorgente dei vari esercizi.

APPENDICE

Come specificato nel capitolo conclusivo, il codice sorgente descritto in questa tesina è disponibile su GitHub al link:

https://github.com/TommasoLabieni/PSO_Excercises

Per scaricarlo su una macchina Linux occorre utilizzare il comando *git*. Se questo comando non dovesse essere installato occorre aprire un terminale e digitare il seguente comando:

sudo apt-get install git

Verificare la corretta installazione digitando:

git --version

Dovrebbe essere stampata a video la versione di git; un esempio è il seguente:

```
$ git --version
git version 2.17.1
```

Da ora è possibile scaricare il repository contenente il codice sorgente degli esercizi descritti. A tal fine è sufficiente digitare:

git clone https://github.com/TommasoLabieni/PSO_Excercises