# Problema dei lettori-scrittori

## Problema e possibili soluzioni

Francesco Lucia

Università degli studi della Basilicata

A.A. 2021/2022

# Sommario

# Il Problema

Il problema dei lettori-scrittori è un classico problema di concorrenza e sincronizzazione che analizza uno scenario che si verifica in molti programmi applicativi.
Nel problema si ha un dato condiviso (che può essere un file, una variabile, un database, un'area di memoria) e una serie di processi che vogliono accedervi in modalità di scrittura o lettura. I processi che chiameremo "lettori" intendono leggere il contenuto condiviso, i processi "scrittori" intendono modificarlo.
La differenza con un problema di sincronizzazione classico è nel fatto che è possibile differenziare le strategie da adottare in base al tipo di processo che accede alla risorsa ottimizzando la soluzione.

# Analisi del problema

Infatti se i processi lettori leggessero contemporaneamente ad un processo scrittore, potrebbero leggere solo una parte del contenuto o non riuscire ad accedervi affatto. Se due scrittori scrivessero contemporaneamente potrebbero sovrascrivere uno il lavoro dell'altro o generare dati corrotti.
Mentre se due lettori dovessero leggere contemporaneamente non si verificherebbe alcun problema.
Analizzando il problema si possono quindi riassumere una serie di considerazioni:

- Più lettori possono leggere contemporaneamente i dati.
- Se uno scrittore sta scrivendo, nessun processo può leggere.
- Solo uno scrittore per volta può scrivere.

La soluzione si serve di strutture per regolare l'accesso alla risorsa imponendo che siano rispettate le 3 considerazioni appena fatte.

# Creazione di lettori e scrittori

Nell'implementare la soluzione sono stati creati 12 threads lettori e 7 threads scrittori con l'utilizzo della libreria pthreads. Questa permette di avviare un thread con la sintassi

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

Che prende come parametri il riferimento al descrittore del thread (che contiene il thread_id), un puntatore alla struttura (attr) con le caratteristiche del thread, la funzione eseguita dal thread e il puntatore ai parametri da passare alla funzione. In questo caso per la creazione degli scrittori è sufficiente l'istruzione

```
pthread_create(&scrittori[i], NULL, scrittore, (void*)i);
```

Dove l'array scrittori è stato creato precedentemente con l'istruzione

```
pthread_t scrittori[NUMERO_SCRITTORI];
```

# Soluzione con semafori e lock mutex

Per semplicità nella soluzione proposta la memoria condivisa è rappresentata da una semplice variabile con visibilità globale, inizialmente con valore 1.
I lettori non fanno altro che stampare il contenuto della variabile sullo schermo, gli scrittori invece modificano la variabile moltiplicandone il contenuto per un numero casuale tra 1 e 10, con l'istruzione

```
variabile_condivisa *= rand()%10+1;
```

A causa della semplicità delle operazioni di lettura e scrittura (trattandosi di una sola variabile), le operazioni vengono effettuate istantaneamente dai threads alla loro creazione. Per simulare una casualità la prima istruzione di un thread è una sleep con un parametro generato casualmente tra 0 e 3 secondi dalla funzione *float randomFloat(float min, float max)*

# Soluzione con semafori e lock mutex

La soluzione privilegia i lettori. La variabile *num_lettori* conta i lettori attualmente attivi. La sincronizzazione è gestita attraverso l'utilizzo in un lock mutex (*mutex*) per la sezione critica relativa all'aggiornamento della variabile contatore dei lettori, e un semaforo (*semaforo_scrittura*) per regolare l'autorizzazione a scrivere per gli scrittori.
Il problema della soluzione è la possibilità di starvation per gli scrittori, se infatti ci sono uno o più lettori attivi e continuano ad arrivarne altri, il programma darà precedenza a questi autorizzando lo scrittore a scrivere solo nel momento in cui tutti i lettori avranno terminato.
Per ovviare al problema si potrebbero implementare dei sistemi che mettano un limite al tempo che uno scrittore può attendere, andando a sospendere le operazioni di lettura per consentire di smaltire la coda degli scrittori.

# Soluzione con semafori e lock mutex

Lock Mutex
Variabile booleana, un processo acquisisce il lock e il successivo che tenta di acquisirlo rimane in attesa finché il primo non lo rilascia.

```
acquire() {
while (!available) ;
available = false;
}
release() {
available = true;
}
```

Semaforo
Variabile intera o booleana, un processo acquisisce il semaforo e i successivi in attesa vengono inseriti in una coda dei processi

```
wait(semaphore *S) {
S->value--;
if (S->value < 0) {
aggiungi il processo a S->list;
sleep();
}
}
signal(semaphore *S) {
S->value++;
if (S->value <= 0) {
togli un processo P da S->list;
wakeup(P);
}
}
```

# Soluzione con semafori e lock mutex

Il lock mutex utilizza il tipo *pthread_mutex* di pthread. Un thread può acquisire il lock su un lock mutex prima di entrare in una sezione critica e successivamente rilasciarlo con le procedure pthread_mutex_lock(&mutex) e pthread_mutex_unlock(&mutex).
I semafori sono implementati nella libreria *semaphore.h*, un semaforo di tipo *sem_t* può essere inizializzato con la procedura *sem_init(&semaforo,0, 1)*, dove 0 indica che il semaforo è condiviso solo nei threads del processo che l'ha generato, il terzo parametro (1) è il valore iniziale del semaforo.
Le istruzioni per acquisire e rilasciare il semaforo sono rispettivamente *sem_wait(&semaforo)* e *sem_post(&semaforo)*

# Il codice

```
pthread_t lettori[NUMERO_LETTORI], scrittori[NUMERO_SCRITTORI];
// inizializza il semaforo e il mutex
pthread_mutex_init(&mutex, NULL);
sem_init(&semaforo_scrittura, 0, 1);

// Creazione Threads
for (unsigned long i = 0; i < NUMERO_SCRITTORI; i++) {
    pthread_create(&scrittori[i], NULL, scrittore, (void*)i);
}

for (unsigned long i = 0; i < NUMERO_LETTORI; i++) {
    pthread_create(&lettori[i], NULL, lettore, (void*)i);
}

// Join sui threads in esecuzione
for (unsigned long j = 0; j < NUMERO_SCRITTORI; j++) {
    pthread_join(scrittori[j], NULL);
}

for (unsigned long h = 0; h < NUMERO_LETTORI; h++) {
    pthread_join(lettori[h], NULL);
}

// Libera le risorse
sem_destroy(&semaforo_scrittura);
pthread_mutex_destroy(&mutex);
return 0;
```

Metodo main
Dopo l'inizializzazione del semaforo e del mutex, due cicli for creano i threads per gli scrittori e i lettori, poi con *pthread_join* attenda che questi terminino. Successivamente è importante deallocare le risorse (terzultima e penultima riga), dimenticarsene potrebbe causare perdita di memoria in alcune situazioni.

# Il codice

```c
void *scrittore(void* id_thread) {
    sleep(randomFloat(0, 3)); // sleep per ritardare l'esecuzione a scopo di test
    unsigned long tid = (unsigned long) id_thread;
    sem_wait(&semaforo_scrittura); // attende la possibilità di scrivere
    variabile_condivisa *= rand()%10+1;
    printf("Scrittore %lu ha moltiplicato la variabile per un numero casuale\n", tid);
    sem_post(&semaforo_scrittura); // rilascia il semaforo

}
```

Lo scrittore chiama una *sem_wait* sul semaforo, una volta acquisito modifica la variabile e stampa la scritta di logging, poi chiama una *sem_post* per liberare il semaforo e permette ad altri scrittori/lettori di agire.

# Il codice

```c
void *lettore(void* id_thread) {
    sleep(randomFloat(0, 3)); // sleep per ritardare l'esecuzione a scopo di test
    unsigned long tid = (unsigned long)id_thread;
    pthread_mutex_lock(&mutex); // entra nella sezione critica
    numero_lettori++;
    if (numero_lettori == 1) {
        // se si tratta del primo lettore attende che la scrittura termini
        sem_wait(&semaforo_scrittura);
    }
    pthread_mutex_unlock(&mutex);

    printf("Lettore %lu - valore letto → %d\n", tid, variabile_condivisa);

    pthread_mutex_lock(&mutex);
    numero_lettori--;
    if (numero_lettori == 0) {
        // se si tratta dell'ultimo lettore rilascia il semaforo per la scrittura
        sem_post(&semaforo_scrittura);
    }
    pthread_mutex_unlock(&mutex);
}
```

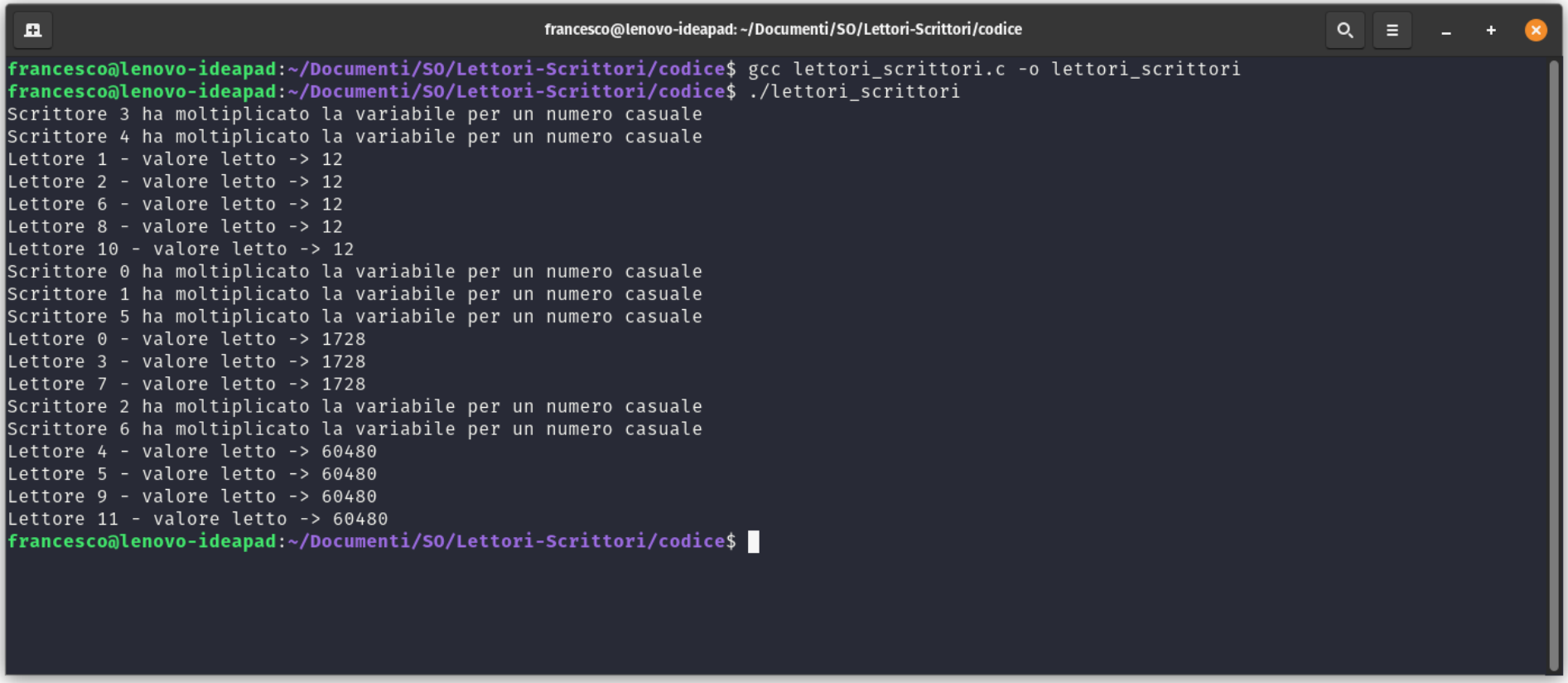
Il lettore acquisisce il lock sul mutex, incrementa la variabile che conta il numero di lettori attualmente attivi, se è il primo lettore (if) acquisisce il semaforo_scrittura (assicurandosi così che non ci siano processi che stanno scrivendo), rilascia poi il lock sul mutex in modo da permettere ad altri lettori di leggere contemporaneamente e legge il valore.

# Il codice

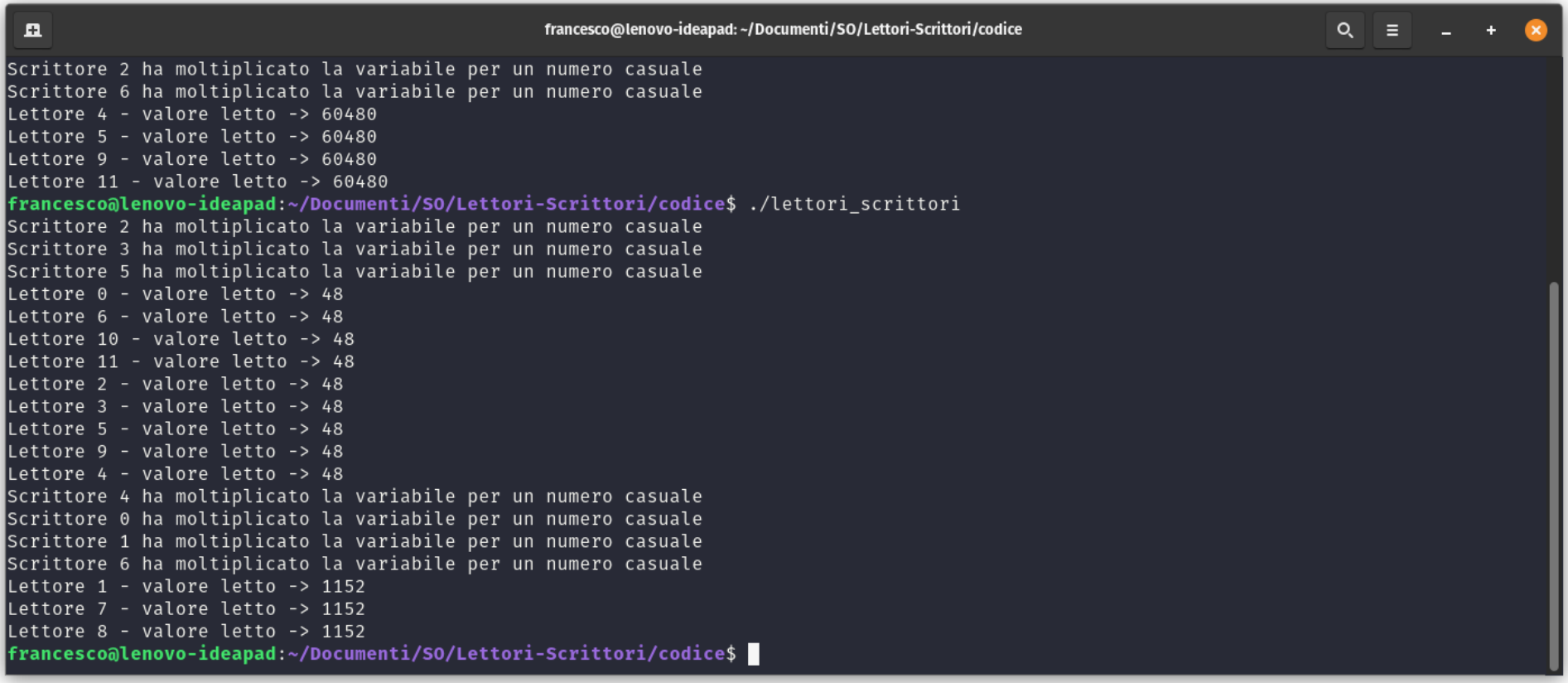
```
void *lettore(void* id_thread) {
    sleep(randomFloat(0, 3)); // sleep per ritardare l'esecuzione a scopo di test
    unsigned long tid = (unsigned long)id_thread;
    pthread_mutex_lock(&mutex); // entra nella sezione critica
    numero_lettori++;
    if (numero_lettori == 1) {
        // se si tratta del primo lettore attende che la scrittura termini
        sem_wait(&semaforo_scrittura);
    }
    pthread_mutex_unlock(&mutex);

    printf("Lettore %lu - valore letto → %d\n", tid, variabile_condivisa);

    pthread_mutex_lock(&mutex);
    numero_lettori--;
    if (numero_lettori == 0) {
        // se si tratta dell'ultimo lettore rilascia il semaforo per la scrittura
        sem_post(&semaforo_scrittura);
    }
    pthread_mutex_unlock(&mutex);
}
```

Dopo la lettura riacquisisce il lock sul mutex per decrementare la variabile numero_lettori, se non ci sono più processi che stanno leggendo rilascia il semaforo_scrittura per permettere ad uno scrittore di iniziare. Successivamente rilascia il lock sul mutex

# Prima esecuzione



# Seconda esecuzione



# Monitor

L'uso dei semafori in alcune situazioni è macchinoso e necessita di uno sforzo particolare da parte dello sviluppatore. Un errore logico può causare bug che si manifestano solo in condizioni particolari e pertanto difficili da individuare.
Un monitor è un tipo di dato che incapsula la logica di gestione della sincronizzazione, esponendo delle funzioni utilizzabili dagli sviluppatori nei programmi che slegano la singola operazione dalla logica di sincronizzazione.
Il secondo esempio di implementazione è una soluzione in linguaggio Java che utilizza una classe Monitor per mostrare come questa semplifichi poi la gestione dei lettori e scrittori (e di conseguenza il codice).

# Soluzione con Monitor

La classe *Memoria* non fa altro che mantenere lo stato della memoria condivisa tra i threads, espone i metodi scrivi e leggi.
La classe *LettoriScrittori* contiene il main del programma, si occupa di creare gli oggetti monitor e memoria e di avviare i threads (la struttura del programma è simile a quella vista per la prima soluzione con le dovute differenze sintattiche e nelle librerie utilizzate per l'avvio dei threads).
Le classi *Lettore* e *Scrittore* estendono la classe astratta *Thread* (metodo standard per la creazione di thread in java), nel metodo *run()* svolgono le operazioni di lettura/scrittura, con la semplice accortezza di chiamare (prima e dopo l'accesso alla risorsa) i rispettivi metodi della classe monitor *iniziaLettura()*/*terminaLettura()* e *iniziaScrittura()*/*terminaScrittura()*.

# Classe Lettore

```java
public class Lettore extends Thread {

    private Monitor monitor;
    private Memoria memoria;

    public Lettore(Monitor monitor, Memoria memoria) {

        this.memoria = memoria;
        this.monitor = monitor;
    }

    @Override
    public void run() {
        try {
            Thread.sleep((long) (Math.random()*3001)); // attesa casuale per i test
            monitor.iniziaLettura();
            System.out.println("Lettore " + getName() + " - valore → " + memoria.leggi());
            monitor.terminaLettura();
        } catch(InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

# Classe Scrittore

```java
public class Scrittore extends Thread{
    private Monitor monitor;
    private Memoria memoria;

    public Scrittore(Monitor monitor, Memoria memoria) {

        this.monitor = monitor;
        this.memoria = memoria;
    }

    @Override
    public void run() {
        try {
            Thread.sleep((long) (Math.random()*3001)); // attesa casuale per i test
            monitor.iniziaScrittura();
            System.out.println("Scrittore " + getName() + " moltiplica per 2 ");
            memoria.scrivi(2);
            monitor.terminaScrittura();

        } catch(InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

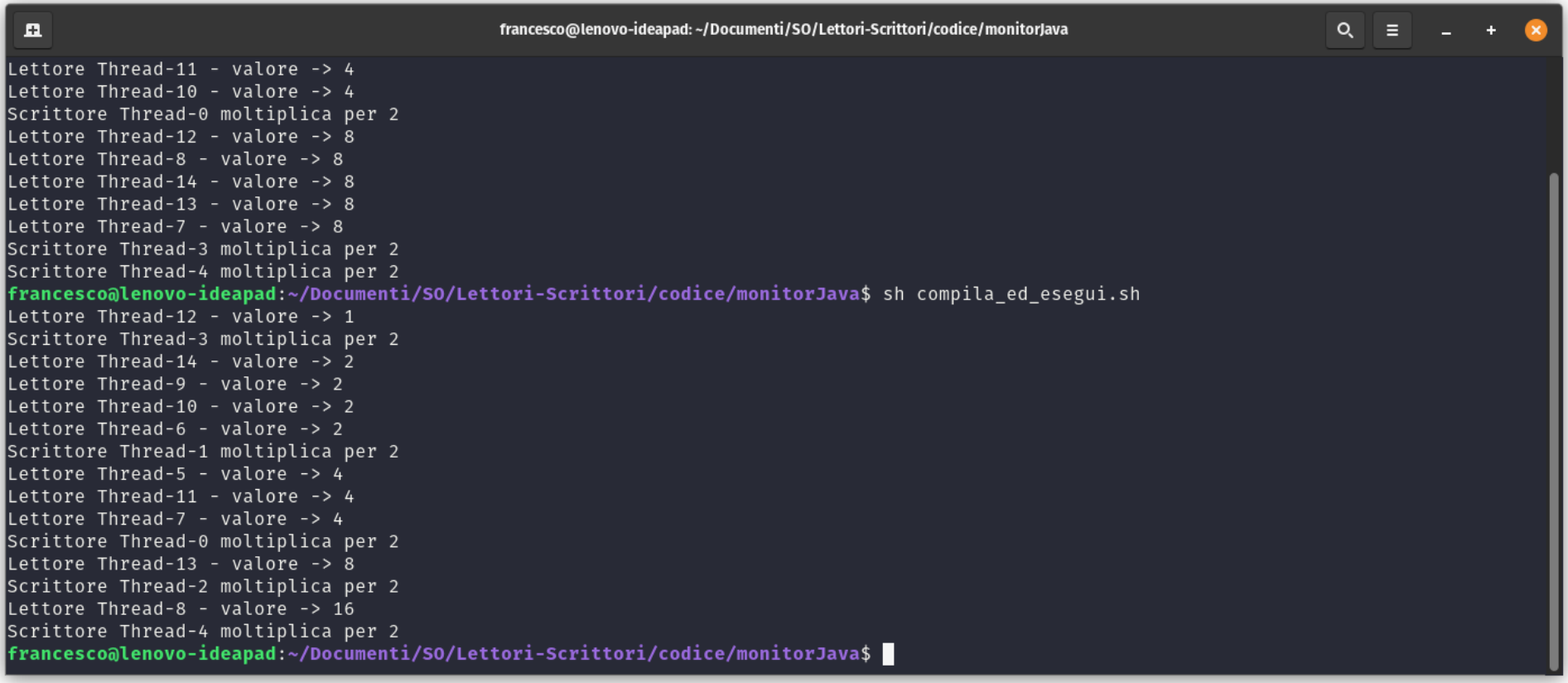
# Sincronizzazione in Java

Il locking è gestibile in Java con i lock rientranti (*ReentrantLock*) che svolge nel Monitor funzioni analoghe al lock mutex nell'implementazione in C. I metodi per acquisire e rilasciare il lock sono rispettivamente *lock()* e *unlock()*.
Le variabili condizionali sono associate a un lock, il metodo *newCondition()* chiamato sul riferimento ad un oggetto *ReentrantLock* crea un nuovo oggetto di tipo *Condition*, che espone i metodi *wait()* e *signal()*.
All'interno della classe Monitor troviamo diversi contatori che mantengono informazioni lo stato degli accessi alle risorse: *numeroLettori*, *numeroScrittori*, *lettoriInAttesa*, *scrittoriInAttesa*.
La classe Monitor utilizza un *ReentrantLock* per gestire le sezioni critiche per l'aggiornamento dei contatori e due variabili condition *condScrittura* e *condLettura*.

# La classe Monitor

# Prima esecuzione

# Seconda esecuzione



Link al repository con i sorgenti delle soluzioni viste e delle slides.
https://github.com/FrancescoLucia/Lettori-Scrittori.git