4-Semafori

Problema del `BoundedBuffer`

```
public class BoundedBuffer {
    private static final int BUFFER_SIZE = 2;
    private Semaphore mutex;
    private Semaphore empty;
    private Semaphore full;
    private int in, out;
    private Object[] buffer;

    public BoundedBuffer() {
        // count = 0;
        in = 0;
        out = 0;
        buffer = new Object[BUFFER_SIZE];
        mutex = new Semaphore(1);  // semaforo di mutual exclusion
        empty = new Semaphore(BUFFER_SIZE);  // empty = numero celle del
                                             // buffer quando inizializzato
        full = new Semaphore(0);  // viene inizializzato vuoto
    }

    public void enter(Object item) {...}

    public Object remove() {...}
}
```

Quando un semaforo tra `empty` e `full` aumenta l'altro diminuisce, tengono traccia se un processo può apparire sul buffer. Se `empty==BUFFER_SIZE` il consumatore non deve fare richieste, se `full==BUFFER_SIZE` il produttore non deve inviare processi.
L'utilizzo del semaforo `mutex` garantisce che le operazioni di inserimento e rimozione nel buffer siano eseguite in modo sincronizzato e sicuro, evitando così situazioni di race condition o corruzione dei dati dovute a accessi concorrenti non sincronizzati al buffer condiviso.

```
public void enter(Object item) {
        empty.P(); // decremento semaforo slot vuoti
        mutex.P();
        buffer[in] = item;
        // context switch tra queste due operzioni causa problemi
        in = (in + 1) % BUFFER_SIZE;
        mutex.V();
        full.V();  // incremento semaforo slot pieni
    }
```

```
public Object remove() {
        full.P();   // decremento slot pieni, il consumatore
                    // si blocca se ci sono solo celle vuote
        mutex.P(); // non bisogna invertire l'ordine di queste due .P()
        Object item = buffer[out];
        // context switch tra queste due operazioni farebbe prelevare due
        // consumatori dalla stessa cella
        out = (out + 1) % BUFFER_SIZE;
        mutex.V();
        empty.V();  // decremento slot vuoti
        return item;
    }
```

Problema Reader-Writer

Bisogna implementare una struttura che impedisca a un processo reader e writer di essere eseguiti in parallelo su uno stesso database, consentendo però a più processi di lettura di essere eseguiti in parallelo e impedisce a più writer di accedervi contemporaneamente. Il database è una risorsa condivisa complessa, deve esserci mutual exclusion tra lettori e scrittori.

```
public class Database {
    private int readerCount; // number of active readers
    Semaphore mutex; // controls access to readerCount
    Semaphore db;
    public Database() {
        readerCount = 0;
        mutex = new Semaphore(1);
        db = new Semaphore(1);
    }

    public int startRead() {...}

    public int endRead() {...}

    public void startWrite() {
        db.P();  // cerca di acquisire il semaforo, se ha successo nessun
                 // altro potrà accedere al database
    }

    public void endWrite() {
        db.V();
    }
}
```

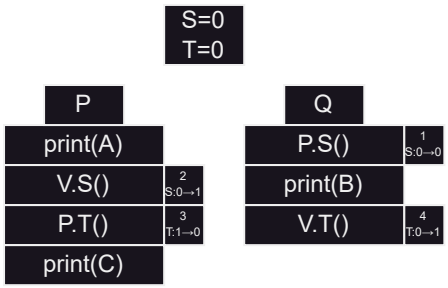
Soluzione con rischio di starvation molto basso

```
public int startRead() {
        mutex.P();  // garantisce che questa funzione venga eseguita senza context switch
        /*
```

```
        Garantisce che non ci siano context switch
        Il primo lettore acquisice mutex, a procede,
        i seguenti rimangono bloccati qua
        */
        ++readerCount;
        if (readerCount == 1)
            db.P(); // i lettori si impossessano del semaforo,
                    // gli scrittori rimangono in attesa
                    // questa operazione viene saltata se c'è già un reader
        mutex.V();
        return readerCount;
    }
```

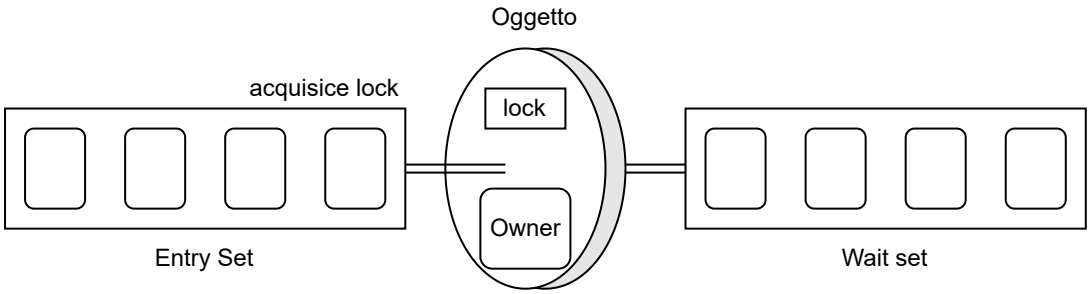
```
    public int endRead() {
        mutex.P(); // il primo lettore che arriva acquisisce il mutex, garantendo
                   // che le istruzioni vengano eseguite senza interruzione
        --readerCount;
        if (readerCount == 0) // se non ci sono lettori il lock viene rilasciato
                              // e gli scrittori possono accedere al database.
            db.V();
        mutex.V();  // fine CS
        return readerCount;
    }
```

Con i semafori si costruiscono strutture più complesse e astratte. Possono essere utilizzati anche per controllare la velocità di esecuzione di due processi paralleli (non deterministici).



Sincronizzazione Java

In java ogni oggetto ha un lock associato, e per eseguire i metodi segnati come `synchronized` bisogna avere quel lock. Se il thread che chiama quel metodo non possiede il lock viene posto sul wait set del lock di quel oggetto. Il lock viene rilasciato quando il thread esce dal metodo sincronizzato (o per terminazione o per interrupt).



Entry set: processi pronti ad essere eseguiti (nello stato ready, non running)
Wait set: thread che avevano acquisito il lock ma durante l'esecuzione sono stati messi in wait.

```
public synchronized void enter(Object item) {
    while (count == BUFFER_SIZE)  // se il buffer è pieno ferma esecuzione e fai
                                  // eseguire un altro thread
            Thread.yield(); // bisogna uscire dal while per lasciare al consumatore
                            // accesso al lock.
    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
}

public synchronized Object remove() {
    Object item;
    while (count == 0)
        Thread.yield();
    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    return item;
}
```

Avendo dichiarato enter e remove come `synchronized` solo un thread lavorerà sul buffer in un dato momento.
Quando un thread invoca il metodo `wait()`, rilascia il lock dell'oggetto, lo stato del thread viene messo a "blocked" e viene messo sul wait set. A differenza dello `yield()` libera il lock.
Quando un thread invoca il metodo `notify()`, viene scelto un thread casuale dal wait set e viene posto sul entry set e poi messo nello stato "runnable". Il metodo `notifyAll()` rimuove tutti i thread dal wait set e li pone sull'entry set. In casi di incertezza garantisce di non incorrere in deadlock.

Operazione	Buffer	Lock	Entry Set	Wait Set
1. Produttore esegue enter	full	P	∅	∅
2. Produttore esegue wait	full	libero	∅	P
3. Consumatore inizia remove	full	C	∅	P
4. Consumatore libera una posizione nel buffer	full	C	∅	P
5. Consumatore esegue notify	full	C	P	∅
6. Consumatore invoca il metodo remove	full	libero	P	∅
7. Produttore riprende l'esecuzione	full	P	∅	∅
8. Produttore inserisce item nel buffer	full	P	∅	∅
9. Produttore termina enter	full	libero	∅	∅

Soluzione problemi di reading e writing con costrutti java

Un database può essere in stato "free", "ready", o "writing".

```
public class Database {
    private int readerCount;
    private boolean dbReading;
```

```
    private boolean dbWriting;

    public Database() {
        readerCount = 0;
        dbReading = false;
        dbWriting = false;
    }

    public synchronized int startRead() {  // il lettore vuole accedere al database
        while (dbWriting == true) {
            try {
                wait(); // autosospende sul waitset
            } catch (InterruptedException e) {
            }
            ++readerCount;
            if (readerCount == 1) // se c'è un lettore si sta accedendo al database
                dbReading = true;
            return readerCount;
        }
    }

    public synchronized int endRead() {
        --readerCount;
        if (readerCount == 0)  // se non ci sono lettori notifica tutti i processi in
                               // waiting perché il database è libero
            db.notifyAll();  // basterbbe anche una notify semplice
        return readerCount;
    }

    public synchronized void startWrite() {
        while (dbReading == true || dbWriting == true)
            try {
                wait();  // perché è occupato da un lettore o scrittore
            } catch (InterruptedException e) {
            }
        dbWriting = true;
    }

    public synchronized void endWrite() {
        dbWriting = false;
        notifyAll(); // libera tutti i thread bloccati perché si vuole permettere
                     // a tutti i lettori sul wait set di accedere al database.
                     // Potrebbero esserci anche degli scrittori svegliati nel wait set
    }
}
```

Sincronizzazione di blocchi

Al posto di dichiarare interi metodi come synchronized, si possono dichiarare blocchi.

```
Object mutexLock = new Object();
...
```

```
public void someMethod() {
    // non-critical section
    synchronized(mutexLock) { // lock da acquisire per eseguire questo blocco
    // critical section
    }
// non-critical section
}
```