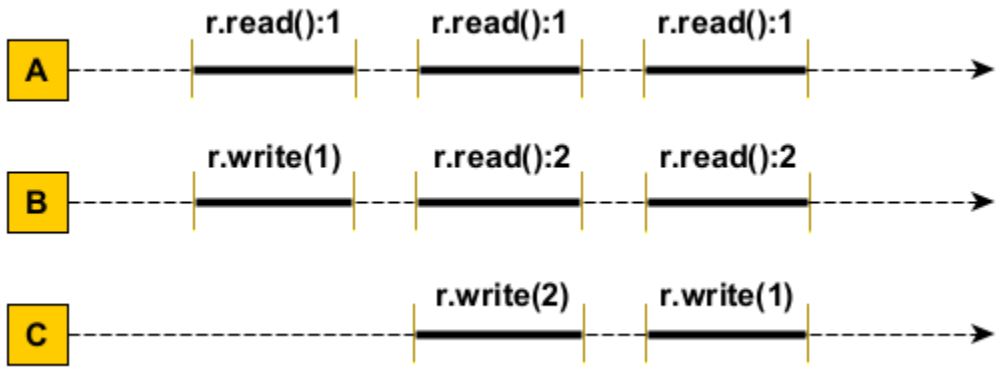
Tema 1

Tema se realizeaza in echipa de maxim 3 studenti. Tema se va trimite prin e-mail catre responsabilul de laborator fie ca arhiva zip cu fisiere incluzand raspunsurile si sursele implementarilor, fie ca link catre un repository, avand ca subiect **TEMA 1 TPM**. In e-mailul trimis se va preciza componenta completa a echipei. Termenul strict de trimitere a temei: **9 noiembrie** (inclusiv).
Finalizarea evaluarii temei se va face intr-o sesiune online in saptamana a opta ce va fi programata in perioada urmatoare, la care vor participa toti membrii echipei.

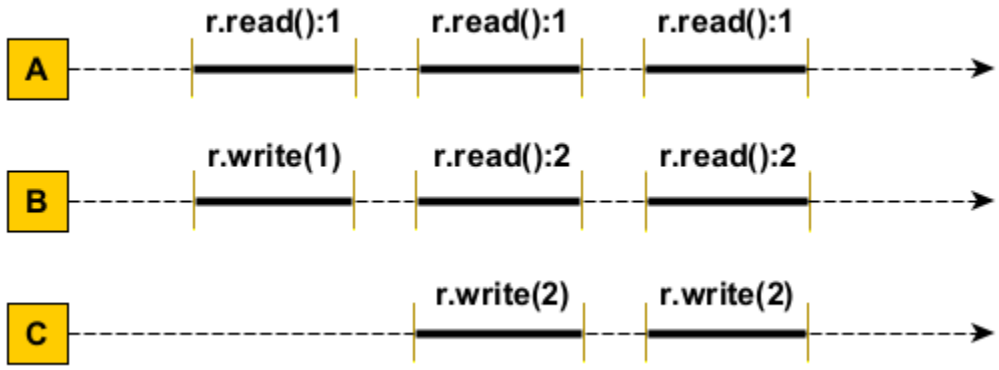
Orice incercare de frauda in rezolvarea temei va fi penalizata cu depunctarea totala a tuturor membrilor echipei.

1. (5 puncte) Se dau urmatoarele secvente (istorii) de executie de mai jos. Sunt acestea linearizabile? Dar consistent secventiale? Se considera valoarea initiala r = 0 (a se raspunde doar pentru exemplul indicat pentru grupa din care faceti parte).
Argumentati raspunsul oferind explicatiile (eventual secventa istoriei de executie) si/sau o diagrama cu punctele de linearizare dupa caz.

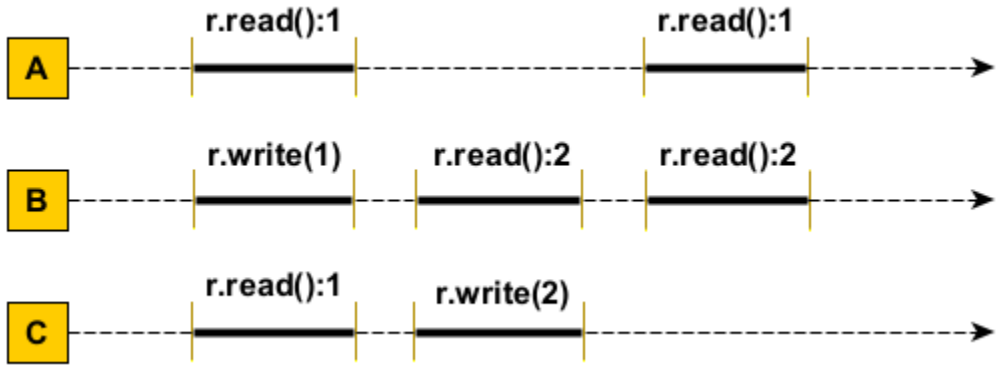
Grupa TPM1:



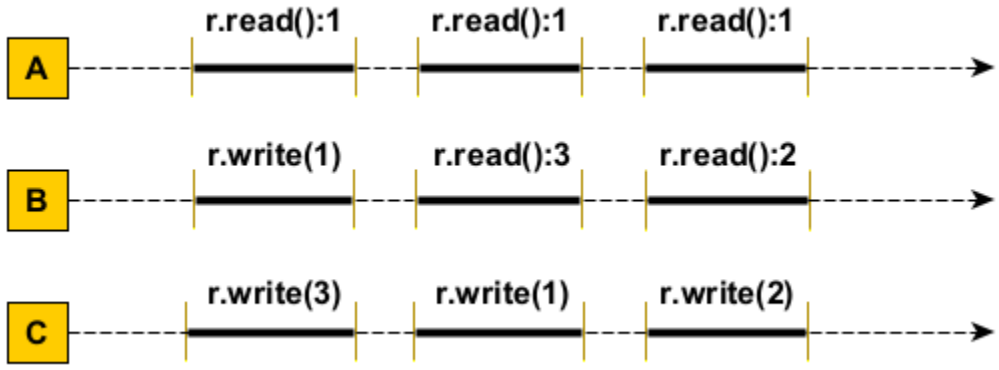
Grupa TPM2:



Grupa TPM3:



Grupa TPM4:



2. a) (7 puncte) Se considera mai jos exemplul lock-free pentru coada FIFO, cu dimensiune limitata, enuntata in finalul cursului 3, ce poate fi folosita de doua thread-uri, unul producator (apeleaza doar enq) si unul consumator (apeleaza doar deq). Acesta este urmat de exemple de pseudocod ce

incearca generalizarea cozii pentru n thread-uri - mai multi producatori si mai multi consumatori - folosind lock-uri.

Functioneaza corect generalizarea propusa? (a se raspunde doar pentru exemplul indicat pentru grupa din care faceti parte). Argumentati raspunsul. Se pot include si eventuale trace-uri demonstrative pentru executia unor thread-uri sau optional implementari ale exemplelor daca este cazul.

Algoritmul de coada limitata lock-free pentru 2 threaduri (un producator si un consumator):

```
public class LockFreeQueue {
        int head = 0, tail = 0;
        int items [] = new int[ QSIZE ];

        public void enq(int x) {
                while ( tail - head == QSIZE ) {};
                items [ tail % QSIZE ] = x;
                tail ++;
        }

        public int deq () {
                while ( tail == head ) {};
                int item = items [ head % QSIZE ];
                head ++;
                return item;
        }
}
```

Algoritmul candidat pentru coada limitata folosind lock pentru n threaduri producator si consumator - grupele TPM1 si TPM2:

```
public class LockBasedQueue {
        int head = 0, tail = 0;
        int items [] = new int[ QSIZE ];
        ReentrantLock lock = new ReentrantLock();

        public void enq(int x) {
                while ( tail - head == QSIZE ) {};
                lock.lock();
                try {
                        items [ tail % QSIZE ] = x;
                        tail ++;
                } finally {
                        lock.unlock();
                }
        }

        public int deq () {
                while ( tail == head ) {};
                lock.lock();
                try {
                        int item = items [ head % QSIZE ];
                        head ++;
```

```
                        return item;
                } finally {
                        lock.unlock();
                }
        }
}
```

Algoritmul candidat pentru coada limitata folosind lock pentru n threaduri producator si consumator - grupele TPM3 si TPM4:

```
public class DoubleLockBasedQueue {
        int head = 0, tail = 0;
        int items [] = new int[ QSIZE ];
        ReentrantLock enqlock = new ReentrantLock();
        ReentrantLock deqlock = new ReentrantLock();

        public void enq(int x) {
                while ( tail - head == QSIZE ) {};
                enqlock.lock();
                try {
                        items [ tail % QSIZE ] = x;
                        tail ++;
                } finally {
                        enqlock.unlock();
                }
        }

        public int deq () {
                while ( tail == head ) {};
                deqlock.lock();
                try {
                        int item = items [ head % QSIZE ];
                        head ++;
                        return item;
                } finally {
                        deqlock.unlock();
                }
        }
}
```

2. b) (5 puncte) De ce in algoritmul Bakery prezentat in cursul 2, al carui pseudocod este reamintit mai jos, in comparatia tuplelor din metoda lock (label[i],i) > (label[k],k) nu este suficienta doar comparatia etichetei (label)? Argumentati raspunsul descriind o situatie concreta pentru doua thread-uri care ar folosi doar etichetele in comparatia respectiva.

```
class Bakery implements Lock {
   boolean[] flag;
   Label[] label;

   public Bakery (int n) {
        flag  = new boolean[n];
        label = new Label[n];
        for (int i = 0; i < n; i++) {
                flag[i] = false; label[i] = 0;
```

```
        }
    }

    public void lock() {
        flag[i]  = true;
        label[i] = max(label[0], ... ,label[n-1])+1;
        while (exists k!=i with flag[k]==true && (label[i],i) > (label[k],k))
{};
    }

    public void unlock() {
        flag[i] = false;
    }
}
```

2. c) (3 puncte) De ce in mod obisnuit in utilizarea unui lock se prefera ca apelul lock() sa fie executat inainte de blocul try, si nu in cadrul acestuia (prima varianta de mai jos si nu a doua)? Argumentati.

lock inainte de try:

```
someLock.lock();
try {
   .....
}
finally {
   someLock.unlock();
}
```

lock in cadrul try:

```
try {
   someLock.lock();
   .....
}
finally {
   someLock.unlock();
}
```

3. Consideram urmatoarea problema: Un trib de salbatici mananca dintr-o singura oala mare ce are o capacitate de N portii. Cand un membru al tribului mananca, va lua o portie din oala daca oala are cel putin o portie disponibila. Daca oala este goala, membrul de trib va ordona bucatarului sa reumple oala si va astepta pana ce aceasta este din nou complet plina. Bucatarul face exclusiv reumpleri complete (N portii). Pe scurt: membrii tribului nu pot lua o portie din oala daca aceasta este goala si bucatarul nu poate reumple oala decat daca este goala.

a) (5 puncte) Scrieti un program care sa simuleze comportamentul membrilor de trib si al bucatarului, unde fiecare dintre acestia este reprezentat de un thread, iar oala este o resursa partajata, respectand constrangerile enuntate mai sus. Considerati ca fiecare dintre membrii de trib doreste sa manance

doar o singura masa, dar numarul lor total este mai mare decat capacitatea oalei, deci aceasta va necesita reumpleri.

b) (5 puncte) Considerati situatia in care membrii de trib sunt permanent flamanzi (thread-urile executa o bucla continua incercand sa ia o noua portie din oala dupa ce mananca o data). Modificati programul intr-un mod in care se asigura garantat ca fiecare dintre membrii de trib va manca la un moment dat (hint: ganditi-va la o modalitate de a face executia fair, astfel incat un membru sa nu manance mai des decat altul). Numarul de membri de trib este fix, si fiecare dintre acestia il cunoaste. Masurati timpul de executie si comparati-l cu cel de la punctul a). Raportati rezultatele obtinute intr-un fisier.

Restrictii:

- Nu este permisa utilizarea in implementarea solutiei a tipurilor atomice din Java.
- Nu este permisa utilizarea in implementarea solutiei pentru punctul b) a mecanismelor din Java care ofera implicit garantii de fairness (ex., utilizarea unei instante Semaphore cu suport de fairness ce se poate initializa prin constructor).