Teoria Współbieżności

Laboratorium 5 - Problem czytelników-pisarzy oraz blokowanie drobnoziarniste.

Magdalena Pastuła

Data laboratorium: 3.11.2020

1. Zadanie do wykonania.

Na laboratorium należało zaimplementować problem czytelników i pisarzy, a następnie przetestować pod względem wydajności dla różnej liczby wątków. Dodatkowo należało zaimplementować problem blokowania drobnoziarnistego i również przetestować pod względem wydajności.

2. Koncepcja rozwiązania.

2.1 Problem czytelników i pisarzy.

Problem ten rozwiązano w dwóch wariantach: pierwszym była sytuacja, w której czytelnicy mają pierwszeństwo do zasobu i w związku z tym może dojść do zagłodzenia pisarzy. Drugi wariant operiał się na sytuacji odwrotnej: to pisarze mieli pierwszeństwo w dostępie do zasobów i w związku z tym mogło dojść do zagłodzenia czytelników.
W obydwóch przypadkach wykorzystano dwa semafory oraz zmienną warunkową. Semafory blokują dostęp do zasobu, jeden jest dla czytelników, drugi dla pisarzy. Pisarz, gdy chce uzyskać dostęp do zasobu, pobiera swój semafor, a po wykorzystaniu oddaje. Natomiast czytelnik ma zachowuje się już odrobinę inaczej. Najpierw pobiera dostęp do zasobu poprzez semafor czytelników, a następnie zajmuje semafor pisarzy, aby żaden pisarz nie mógł pisać w momencie czytania. Po zajęciu semafora dla pisarzy, czytelnik zwalnia semafor dla czytelników, aby jednocześnie kilku czytelników mogło mieć dostęp do zasobu. Po użyciu dostępu do zasobu czytelnik zajmuje semafor czytelników, oddaje semafor pisarzy i oddaje semafor czytelników. Natomiast wykorzystanie zmiennej warunkowej jest inne w tych dwóch implementacjach. W przypadku, gdy to czytelnicy mają pierwszeństwo w dostępie do pliku, zmienna warunkowa informuje pisarzy, że mogą pisać. W przypadku, gdy to pisarze mają pierwszeństwo, zmienna ta informuje czytelników, że aktualnie nie ma żadnych pisarzy i można czytać.

2.2 Problem blokowania drobnoziarnistego.

W celu zaprezentowania tego problemu utworzona dwie klasy, z których jedna repezentuja listę z blokowaniem drobnoziarnistym, a druga z blokowaniem całej listy przy jakimkolwiek dostępie. Dodatkowo, utworzono klasy na wątki obsługujące te dwie listy.

3. Implementacja i wyniki.

Wszystkie pliki źródłowe do tego laboratorium znajdują się w pliku .zip.

3.1 Problem czytelników i pisarzy.

Poniżej znajduje się implementacja klasy Reader oraz Writer dla przypadku pierwszeństwa czytelników.

```
public class Reader extends Thread {
    private Random gen = new Random();
    private Semaphore _writers_sem;
    private Semaphore _readers_sem;
    private SynchData _synchData;

    public Reader(Semaphore readers_sem, Semaphore writers_sem, SynchData
synchData) {
        _writers_sem = writers_sem;
        _readers_sem = readers_sem;
        _synchData = synchData;
    }

    public void run() {
        try {
            sleep(gen.nextInt(10));
        } catch (InterruptedException e) {;}
        while (!_readers_sem.tryAcquire()) {}
        _synchData.waiting_readers_no++;
        _synchData.reading_no++;
        if (_synchData.reading_no == 1)
        {
            while (!_writers_sem.tryAcquire()) {}

            _synchData.canWrite = false;
        }
        _synchData.waiting_readers_no--;
        _readers_sem.release();

        try {
            int time_to_sleep = 10;
            sleep(time_to_sleep);
        } catch (Exception e) {}

        while (!_readers_sem.tryAcquire()) {}
        _synchData.reading_no--;
        if (_synchData.reading_no == 0) {
            _writers_sem.release();
            if (_synchData.waiting_readers_no == 0) {
                _synchData.lock.lock();
                _synchData.canWrite = true;
                _synchData.write_cond.signal();
                _synchData.lock.unlock();
            }
        }
        _readers_sem.release();
    }
}
```

```
public class Writer extends Thread {
    private Random gen = new Random();
    private Semaphore _sem;
    private SynchData _synchData;

    public Writer(Semaphore sem, SynchData synchData) {
        _sem = sem;
        _synchData = synchData;
    }

    public void run() {
        _synchData.lock.lock();
        try {
            while (!_synchData.canWrite)
            {
                _synchData.write_cond.await();
            }
            while (!_sem.tryAcquire()) {}
            int time_to_sleep = 10;
            sleep(time_to_sleep);
            if (_synchData.waiting_readers_no != 0)
            {
                _synchData.canWrite = false;
            } else {
                _synchData.write_cond.signal();
            }
        } catch (Exception e) {}
        _sem.release();
        _synchData.lock.unlock();
    }
}
```

Natomiast poniższe fragmenty kodu to te same klasy dla przypadku pierwszeństwa pisarzy.

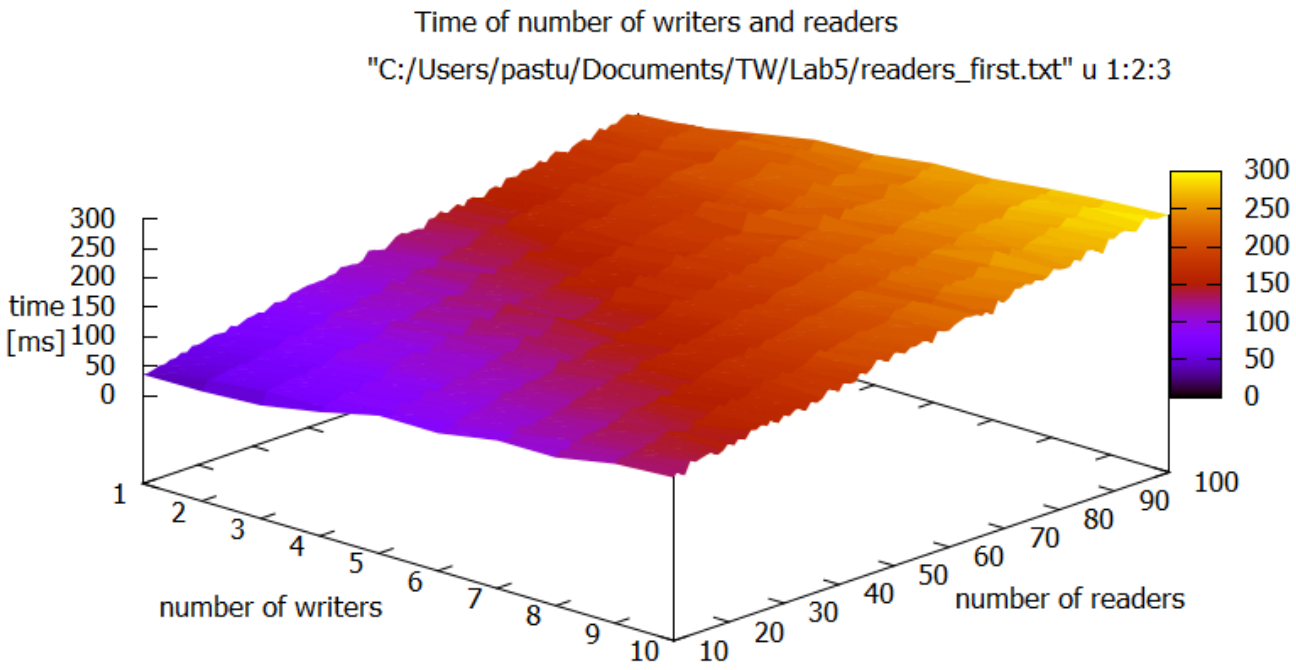
```
public class Reader extends Thread {
    private Random gen = new Random();
    private Semaphore _writers_sem;
    private Semaphore _readers_sem;
    private SynchData _synchData;

    public Reader(Semaphore readers_sem, Semaphore writers_sem, SynchData
synchData) {
        _writers_sem = writers_sem;
        _readers_sem = readers_sem;
        _synchData = synchData;
    }

    public void run() {
        try {
            sleep(gen.nextInt(25));
        } catch (InterruptedException e) {;}
```

```
        _synchData.lock.lock();
        try {
            while (!_synchData.canRead)
            {
                _synchData.read_cond.await();
            }
        } catch (Exception e) {}
        _synchData.lock.unlock();

        while (!_readers_sem.tryAcquire()) {}
        _synchData.reading_no++;
        if (_synchData.reading_no == 1)
        {
            while (!_writers_sem.tryAcquire()) {}
        }
        _readers_sem.release();

        try {
            int time_to_sleep = 10;
            sleep(time_to_sleep);
        } catch (Exception e) {}

        while (!_readers_sem.tryAcquire()) {}
        _synchData.reading_no--;
        if (_synchData.reading_no == 0) {
            _writers_sem.release();
        }
        if (_synchData.waiting_writers_no != 0)
        {
            _synchData.canRead = false;
        } else {
            _synchData.lock.lock();
            _synchData.read_cond.signal();
            _synchData.lock.unlock();
        }
        _readers_sem.release();
    }
}
```

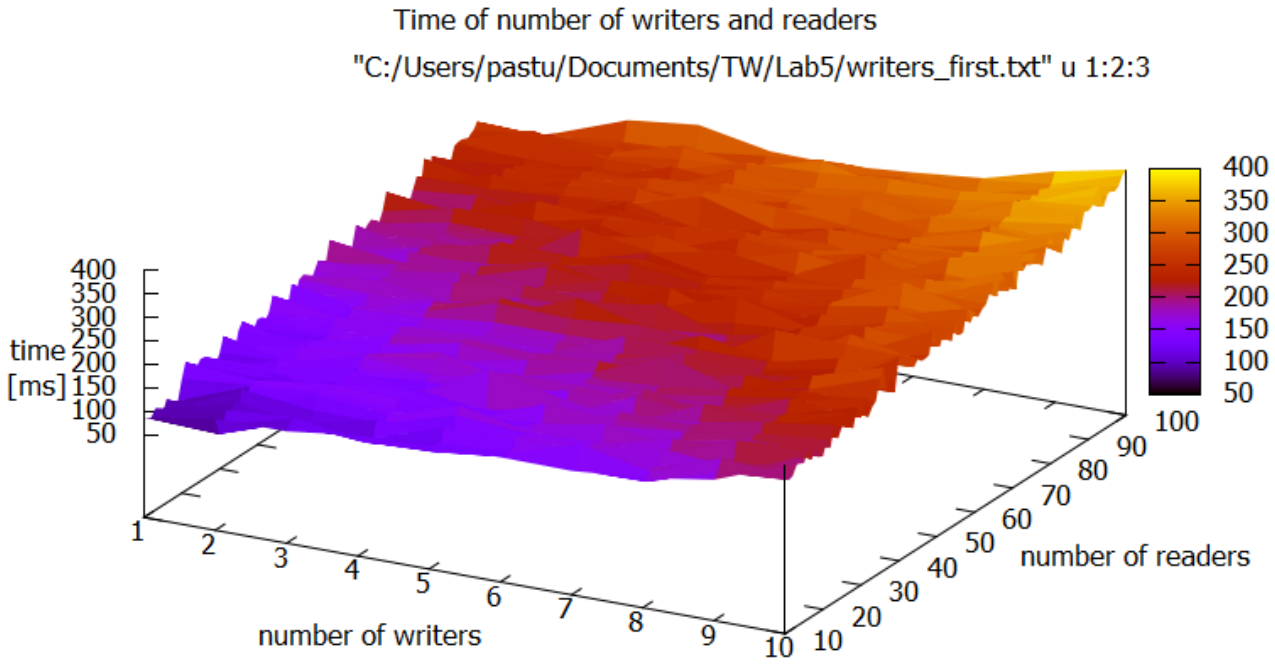
```
public class Writer extends Thread {
    private Random gen = new Random();
    private Semaphore _sem;
    private SynchData _synchData;

    public Writer(Semaphore sem, SynchData synchData) {
        _sem = sem;
        _synchData = synchData;
    }

    public void run() {
```

```
            try {
                sleep(gen.nextInt(10));
            } catch (InterruptedException e) {;}

            try {
                _synchData.lock.lock();
                _synchData.waiting_writers_no++;
                _synchData.lock.unlock();
                while (!_sem.tryAcquire()) {}
                _synchData.lock.lock();
                _synchData.canRead = false;
                _synchData.lock.unlock();
                _synchData.waiting_writers_no--;
                int time_to_sleep = 10;
                sleep(time_to_sleep);

            } catch (Exception e) {}

            if (_synchData.waiting_writers_no == 0) {
                _synchData.lock.lock();
                _synchData.canRead = true;
                _synchData.read_cond.signal();
                _synchData.lock.unlock();
            }

            _sem.release();
        }
    }
```

Dla przypadku, w którym pierwszeństwo mają czytelnicy, otrzymano następujący wykres czasu działania programu od liczby czytelników i pisarzy.



Natomiast poniżej znajduje się ten sam wykres, ale dla przypadku z pierwszeństwem pisarzy.



W obydwóch tych przypadkach czas wykonania programu rośnie liniowo od liczby czytelników i pisarzy. Jedynie dla przypadku, gdzie pisarze mają pierwszeństwo, można zauważyć małe górki dla 3-4 pisarzy.

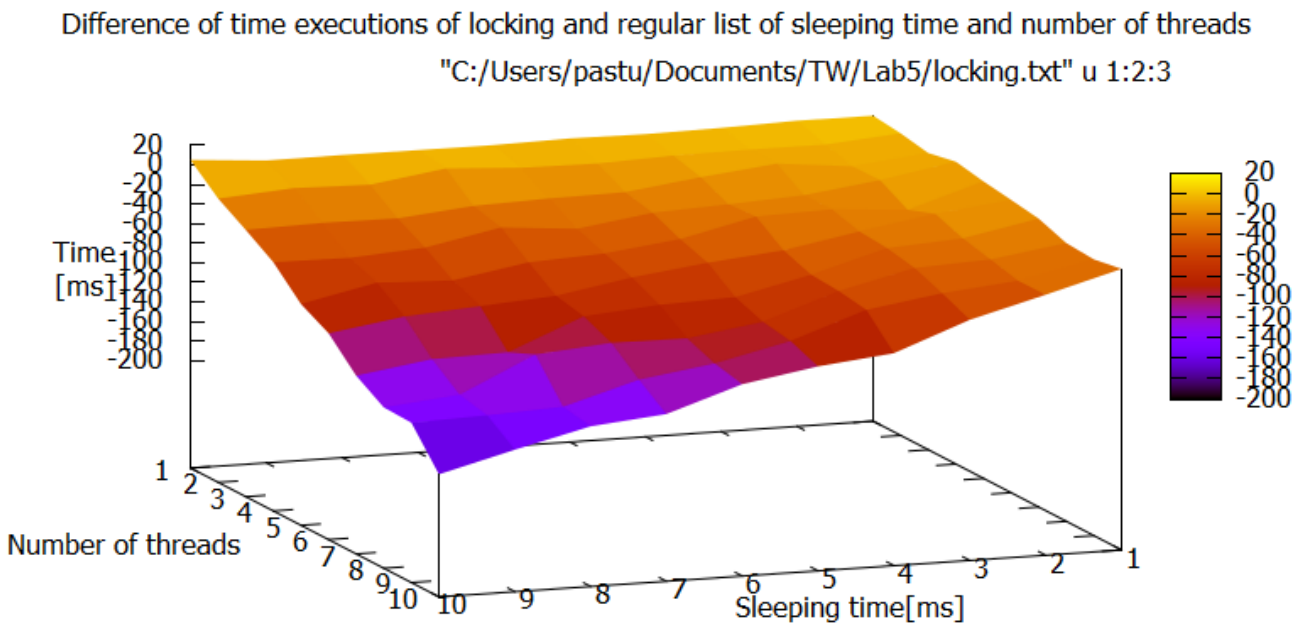
3.2 Problem blokowania drobnoziarnistego.

Implementacja listy z blokowaniem drobnoziarnistym:

```
public class LockingList {
    private ListElem head = null;
    private int sleep_time;

    public LockingList(int sleeping_time) {
        sleep_time = sleeping_time;
    }

    boolean contains(Object o)
    {
        ListElem indx = head;
        if (indx != null)
        {
            indx._lock.lock();
        }
        while (indx != null)
        {
            if (indx._object == o)
            {
                try {
                    Thread.sleep(sleep_time);
                } catch (Exception e) {}
                indx._lock.unlock();
                return true;
            }
            if (indx._nextElem != null)
            {
                indx._nextElem._lock.lock();
            }
            indx._lock.unlock();
            indx = indx._nextElem;
        }
        return false;
    }

    boolean remove(Object o)
    {
        if (head == null) {
            return false;
        }
        head._lock.lock();
        if (head._object == o)
        {
            try {
                Thread.sleep(sleep_time);
            } catch (Exception e) {}
            ListElem prev_head = head;
            head = head._nextElem;
            prev_head._lock.unlock();
            return true;
        }
        ListElem indx = head;
```

```
        while (indx._nextElem != null)
        {
            indx._nextElem._lock.lock();
            if (indx._nextElem._object == o)
            {
                try {
                    Thread.sleep(sleep_time);
                } catch (Exception e) {}
                indx._nextElem = indx._nextElem._nextElem;
                indx._lock.unlock();
                indx._nextElem._lock.unlock();
                return true;
            }
            indx._lock.unlock();
            indx = indx._nextElem;
        }
        return false;
    }

    boolean add(Object o)
    {
        if (head == null)
        {
            try {
                Thread.sleep(sleep_time);
            } catch (Exception e) {}
            head = new ListElem(o, null);
            return true;
        }

        ListElem indx = head;
        indx._lock.lock();
        while (indx._nextElem != null)
        {
            indx._nextElem._lock.lock();
            indx._lock.unlock();
            indx = indx._nextElem;
        }

        try {
            Thread.sleep(sleep_time);
        } catch (Exception e) {}
        indx._nextElem = new ListElem(o, null);
        indx._lock.unlock();

        return true;
    }
}
```

Poniżej znajduje się wykres przedstawiający różnicę czasów wykonania programu z listą z blokowaniem drobnoziarnistym a z blokowaniem całej listy w zależności od liczby wątków oraz czasu wykonania operacji.



W każdym przypadku liczba wątków listy z blokowaniem drobnoziarnistym i listy z pełnym blokowaniem była taka sama. Testy wykonano tylko dla operacji dodawania do listy.

Dla jednego wątku i czasu operacji jednostkowej równej około 1 ms różnica wyniosła 2,2 ms, co jest bardzo małą różnicą na korzyść listy z pełnym blokowaniem.

4. Wnioski z ćwiczenia.

Problem czytelników i pisarzy.

Przyrost czasu wykonania programu jest liniowy względem liczby pisarzy oraz czytelników. Nie wygląda na to, by były od tej zależności jakieś większe odchylenia. Jedynie w przypadku, gdy pierwszeństwo mają pisarze, można zauważyć większe odchylenia, niż w drugim przypadku.

Blokowanie drobnoziarniste.

Jak można zauważyć w przypadku bardzo małej liczby wątków i krótkiego czasu dostępu do zasobu blokowanie całej listy jest niewiele szybsze niż blokowanie element po elemencie. Natomiast w pozostałych przypadkach to lista z blokowaniem drobnoziarnistym okazuje się być szybsza, im więcej wątków i większy czas dostępu, tym większa jest ta różnica. W przypadku wykonywania pozostałych operacji (wyszukiwanie i usuwanie) trend pozostanie ten sam, jak w przypadku dodawania do listy, jednakże nachylenie tej płaszczyzny może być mniejsze.

Bibliografia.

1. Dokumentacja interfejsu Condition
2. Dokumentacja klasy ConditionObject implementującej interfejs Condition
3. Dokumentacja interfejsu Lock