Concurrent Objects

Report IV
Concurrent Programming ID1217

Emil Ståhl
Student ID: 970410

Concurrent Programming ID1217
Report IV

Emil Ståhl

February 24, 2020

1 Introduction

This report covers the implementation of a number of programs written in Java that utilizes parallel execution with multiple threads. Synchronization is done solely with monitors. In concurrent programming, a monitor is a thread-safe class, object, or module that wraps around a mutex in order to safely allow access to a method or variable by more than one thread. The defining characteristic of a monitor is that its methods are executed with mutual exclusion: At each point in time, at most one thread may be executing any of its methods. By using one or more condition variables it can also provide the ability for threads to wait on a certain condition by utilizing the `wait()` and `notify()` constructs.[1] The main topic covered in this work is to get a deeper understanding of the problems that a multithreaded program results in and how they can be solved with monitors.

2 The programs and its purposes

The work consists of three different multithreaded programs written in Java. Below is a description of each program and its purposes.

2.1 The Unisex Bathroom Problem

This program simulates a unisex bathroom that is used by an arbitrary number of men and women. The men and women are represented as threads that work (sleeps) and uses the bathroom for a random amount of time. Synchronization is done solely with monitors. The persons will sleep for a random amount of time between visits to the bathroom as well as sleep for a smaller random amount of time to simulate the time it takes to be in the bathroom. The program prints a trace of interesting simulation events and then terminates when every person has used the bathroom for a specified number of times.

[1] Thread Synchronization by Bill Venners

2.2 The Bear and Honeybees Problem

This program simulates the concept of multiple producers and a single consumer. Given are n honeybees and a hungry bear. They share a pot of honey. The pot is initially empty; its capacity is H portions of honey. The bear sleeps until the pot is full, then eats all the honey and goes back to sleep. Each bee repeatedly gathers one portion of honey and puts it in the pot; the bee who fills the pot awakens the bear. The bees are represented as concurrent threads (i.e. array of "bees" threads and a "bear" thread), and the pot as a critical shared resource that can be accessed by at most one thread at a time. Monitors are used for synchronization.

2.3 The Hungry Birds Problem

This program simulates the concept of one producer and multiple consumers. Given are n baby birds and one parent bird. The baby birds eat out of a common dish that initially contains W worms. Each baby bird repeatedly takes a worm, eats it, sleeps for a while, takes another worm, and so on. If the dish is empty, the baby bird who discovers the empty dish chirps real loud to awaken the parent bird. The parent bird flies off and gathers W more worms, puts them in the dish, and then waits for the dish to be empty again. This pattern repeats forever. The birds are represented as concurrent threads (i.e. array of "babyBird" threads and a "parentBird" thread), and the dish as a critical shared resource that can be accessed by at most one bird at a time. Monitors are used for synchronization.

3 Main problems and solutions

Below is a description of the different problems that each implementation resulted in and how they were solved.

3.1 The Unisex Bathroom Problem

The monitor (bathroomMonitor.java) has four public procedures: manEnter, manExit, womanEnter, and womanExit. A man thread calls manEnter to get permission to use the bathroom and calls manExit when finished. A woman thread calls womanEnter and womanExit. Each procedure is equipped with a synchronized keyword. All synchronized blocks synchronized on the same object can only have one thread executing inside them at a time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

However, once a thread has entered the synchronized block it starts with incrementing a counter of how many persons of the corresponding gender are waiting in line to use the bathroom. If there are people inside the bathroom of the opposite gender the thread are set to `wait()` until being notified. When the thread gets notified it starts by decrementing the counter of the queue, sets

the state to "gender entering" and lastly increments the counter of how many persons are in the bathroom. Before exiting the monitor the thread is set to "use the bathroom" for up to three seconds. Upon calling the Exit() procedure the thread decrements the counter of how many persons are in the bathroom and when the bathroom is empty the last thread to exit calls `notifyAll()` to let the opposite gender enter.

When the bathroom is empty and there are persons of the opposite gender waiting they are prioritized in order to ensure fairness and avoid starvation. Since the waiting queue is FIFO that will ensure fairness. In addition to this, since the program won't terminate until every person has used the bathroom an equal number of times this will work as a proof for fairness if the program terminates.

3.2 The Bear and Honeybees Problem

The program consists of one consumer thread and multiple producer threads that all manipulates a shared global variable `honeyPot`. The producing threads will manipulate the shared variable thru the monitor procedure `create()` that is equipped with a synchronized keyword. Once inside the thread will be set to `wait()` if the honeypot is full. If the honeypot is not full the thread will increment the shared variable honeypot. Before leaving, the thread checks if the honeypot is full and if so it calls `notifyAll()` which will awaken the consuming thread. The consuming thread will manipulate the shared variable honeypot thru a procedure `eat()`. While the honeypot is not full the thread will be set to wait until notified. Once notified it sets the honeypot to a value of zero in order to simulate consuming. Lastly it calls `notifyAll()` to notify the producer threads to start producing again.

3.3 The Hungry Birds Problem

The program consists of one producer thread and multiple consumer threads that all manipulates a shared global variable `worms`. The consuming threads will manipulate the shared variable thru the monitor procedure `eat()` that is equipped with a synchronized keyword. Once inside the thread will be set to wait if the variable is zero. If the variable `worms` is not zero the thread will decrement the shared variable. Before leaving, the thread checks if the variable is zero and if so it calls notifyAll() which will awaken the producing thread. The producing thread will manipulate the shared variable `worms` thru a procedure `refill()`. While the variable `worms` is not empty the thread will be set to wait until notified. Once notified it sets the variable to a predetermined value in order to simulate producing. Lastly it calls `notifyAll()` to notify the consumer threads to start consuming again.

4 Evaluation

A test was performed for each program:

4.1 The Unisex Bathroom Problem

```
emilstahl$ java Main 2 2 1
numWomen = 2
numMen = 2
numVisits = 1

State:[    Empty     ] Bathroom: W:[  ]M:[  ] Queues: W:[]M:[] Women 1 wants to enter
State:[Women Entering] Bathroom: W:[ ]M:[  ] Queues: W:[]M:[] Women 1 enters: Visit: 1
State:[Women Entering] Bathroom: W:[ ]M:[  ] Queues: W:[]M:[] Men 1 wants to enter
State:[Women Entering] Bathroom: W:[ ]M:[  ] Queues: W:[]M:[] Men 0 wants to enter
State:[Women Entering] Bathroom: W:[ ]M:[  ] Queues: W:[]M:[] Women 0 wants to enter
State:[Women Entering] Bathroom: W:[]M:[  ] Queues: W:[]M:[] Women 0 enters: Visit: 1
State:[Women Leaving ] Bathroom: W:[ ]M:[  ] Queues: W:[]M:[] Women 1 leaves
State:[Women Leaving ] Bathroom: W:[  ]M:[  ] Queues: W:[]M:[] Women 0 leaves
State:[ Men Entering ] Bathroom: W:[  ]M:[ ] Queues: W:[]M:[] Men 1 enters: Visit: 1
State:[ Men Entering ] Bathroom: W:[  ]M:[] Queues: W:[]M:[] Men 0 enters: Visit: 1
State:[ Men Leaving  ] Bathroom: W:[  ]M:[ ] Queues: W:[]M:[] Men 1 leaves
State:[ Men Leaving  ] Bathroom: W:[  ]M:[  ] Queues: W:[]M:[] Men 0 leaves
emilstahl$
```

4.2 The Bear and Honeybees Problem

```
Emils-MBP-15:2. The Bear and Honeybees Problem emilstahl$ ./make.sh
Running HoneyBees

Number of bees = 10
Max honey = 10

Bee nr 1 created honey Quantity = 1
Bee nr 0 created honey Quantity = 2
Bee nr 9 created honey Quantity = 3
Bee nr 8 created honey Quantity = 4
Bee nr 2 created honey Quantity = 5
Bee nr 5 created honey Quantity = 6
Bee nr 6 created honey Quantity = 7
Bee nr 4 created honey Quantity = 8
Bee nr 7 created honey Quantity = 9
Bee nr 3 created honey Quantity = 10
Bear ate all the honey
Bee nr 1 created honey Quantity = 1
Bee nr 3 created honey Quantity = 2
Bee nr 7 created honey Quantity = 3
Bee nr 4 created honey Quantity = 4
Bee nr 6 created honey Quantity = 5
Bee nr 5 created honey Quantity = 6
Bee nr 9 created honey Quantity = 7
Bee nr 2 created honey Quantity = 8
Bee nr 0 created honey Quantity = 9
Bee nr 8 created honey Quantity = 10
Bear ate all the honey
```

4.3 The Hungry Birds Problem

```
Emils-MBP-15:3. The Hungry Birds Problem emilstahl$ ./make.sh
Running Hungrybirds

Number of birds = 10
Number of worms = 10

BabyBird nr 3 ate a worm Quantity = 9
BabyBird nr 8 ate a worm Quantity = 8
BabyBird nr 2 ate a worm Quantity = 7
BabyBird nr 9 ate a worm Quantity = 6
BabyBird nr 6 ate a worm Quantity = 5
BabyBird nr 7 ate a worm Quantity = 4
BabyBird nr 5 ate a worm Quantity = 3
BabyBird nr 0 ate a worm Quantity = 2
BabyBird nr 1 ate a worm Quantity = 1
BabyBird nr 4 ate a worm Quantity = 0
BabyBird nr 4 SQUEEEELS!!!!
ParentBird added 10 worms to the dish
BabyBird nr 2 ate a worm Quantity = 9
BabyBird nr 4 ate a worm Quantity = 8
BabyBird nr 7 ate a worm Quantity = 7
BabyBird nr 6 ate a worm Quantity = 6
BabyBird nr 5 ate a worm Quantity = 5
BabyBird nr 3 ate a worm Quantity = 4
BabyBird nr 1 ate a worm Quantity = 3
BabyBird nr 9 ate a worm Quantity = 2
BabyBird nr 0 ate a worm Quantity = 1
BabyBird nr 8 ate a worm Quantity = 0
BabyBird nr 8 SQUEEEELS!!!!
ParentBird added 10 worms to the dish
```

5 Conclusions

This work has focused on using monitors for synchronization in a number of multithreaded programs. The main topics covered was to better understand the advantages and problems that a multithreaded system results in.