Assignment 3: Concurrency

Benjamin Hugo (HGXBEN001)

September 15, 2011

Contents

Abstract

We have been asked to simulate a paint ball practice session concurrently. Issues regarding implementation, synchronization and thread-liveness will be discussed in this report.

1 Problem

In this simulation there are four bouncer threads monitoring the number of volunteers in the simulation and letting in new volunteers if there are not more than a maximum number of volunteers.
Lets define this bounding condition as v_{max} and the number of volunteers in the room as v_{num}).
Once $v_{num} = v_{max}$ the bouncers should stop letting in more volunteers until $v_{num} \leq \frac{v_{max}}{2}$.
Sam can be hit a maximum of 500 times and she checks her own health status every $2ms$. She has the opportunity of eliminating a volunteer every $10ms$, but every volunteer has the chance to hit her on a $10ms$ time step.

2 Alterations and additional features added

The original specification of the problem did not take the effects of distance and number of volunteers into account when specifying the probability that a shot will hit its target. In my simulation, however, I've taken these two factors into account.
I assume that all volunteers have the ability to shoot Sam and that the probability of a hit (P_{hit}) obeys $P_{hit} \propto \frac{1}{dist}$, where *dist* is defined as the distance between Sam and the volunteer. In order to implement this successfully, I've made the volunteers threads themselves.
Another issue arises when the number of volunteers is checked. The instructions told us to let Sam perform this check every 2 seconds. However, the threshold, v_{max}, may be exceeded due to the fact that we're possibly adding a volunteer every $10ms$ and only checking the bound every $2s$. Due to this issue I have changed it to enable the bouncers to keep track of the number of volunteers they let into the room (they will check before letting in any new volunteers).
Furthermore I've added a GUI to visualize the simulation.

3 Implementation

3.1 Java project structure

The following classes have been used in the *Java* implementation of this simulation:

1. *Driver.java.* The driver class of the system. Tasked with instantiating and starting a Sam thread and 4 other bouncer threads. It also creates the visual display.
2. *WndRepresentation.java.* This is the Swing-based window that visualizes the simulation and displays statistics.
3. *Bouncer.java.* The bouncer class is a thread. It is tasked with letting volunteers into the simulation.
4. *Sam.java.* This class represents Sam. The class is a thread and is tasked with monitoring Sam's health and eliminating volunteers.
5. *Volunteer.java.* The volunteer class is also a thread. It moves the representation of a volunteer around the simulated arena and is tasked with shooting Sam.

4 Problems regarding the concurrency of the system

4.1 Mutual exclusion and dealing with synchronization (thread safety)

The following coupling between the classes exists, which needed synchronization:

- The bouncer class adds volunteers to the system if $v_{num} < v_{max}$. Since there are multiple bouncers and the collection of volunteers is shared, there is a chance that interleaving can cause a race condition. Therefore this non-atomic process requires synchronization.

- A check then act mechanism is used in the Swing window's paint routine to draw a visual representation of every volunteer. This creates a race-condition in the event that one (or more) of the volunteers is eliminated by Sam while they are been drawn onto the screen. The draw process therefore need to be atomic and a lock is placed on the entire Volunteer class.
- Each volunteer in the collection of volunteers have simultaneous access to Sam's health. Since the decrement operation is not atomic by nature (due to reassignment) a race condition will arise when it is accessed by different volunteers at the same time. Acquiring a lock on the Sam object for the duration of this decrement is thus necessary. It also prevents the occurence of a race condition when Sam checks her health during a decrement.
- The removal of a volunteer from the collection can create a race condition if the collection's cardinality is obtained at the same time as the removal. Therefore a lock on the collection is acquired during a call to the remove operation.
- The Java Swing system uses an asynchronous draw mechanism. This can cause a race condition if not implemented correctly (a new frame is drawn before the other has completed) and causes flickering (which a double buffering mechanism cannot eliminate by itself). Serializing the paint method is thus critical.

Some of the couplings do not require synchronization and has been ignored in order to increase parallelization:

- Since there will never be more than one Sam object in the system we can safely assume that the non-atomic check-then-act situation we have when Sam removes a volunteer (if $v_{num} > 0$) will not cause a race condition (since only Sam can decrease the number of volunteers in the system).
- We are told that Sam checks her own health every $2ms$ and calls an end to the session if she sustained more than 500 hits, we don't have to take the case where a volunteer decreases Sam's health below 0 into account (in other words Sam is shot more than 500 times before she checks her health).
- A volunteer is added to the collection of volunteers only after it's properties have been set. A race condition involving the add operation and accessors for the positions can thus not occur and there is no reason to synchronize this operation.
- In the event that the bouncers are waiting for the population to drop below $\frac{v_{max}}{2}$ it has been assumed that the bouncers don't have to react immediately, since they only check v_{num} every $10ms$. We thus don't have to synchronize Sam's elimination of volunteers and this check.

I do, however, require the driver to execute its logic in serial order (Sam needs to be eliminated before the final training statistics are displayed). Therefore I wait for the threads to join up, before I display the statistics.

4.2 Liveness of the system

I've already mentioned a few areas where I re-factored the simulation to improve the responsiveness of the system.
Take for example the case where the threshold v_{max} was checked every $2s$. If the bouncers had to wait for Sam to perform this check (synchronization), they wouldn't have been able to let in volunteers every $10ms$.
The only area where a liveness issue may arise, is where the volunteers are drawn onto the screen (capped at 30 frames per second). Since it is quite likely for a race condition to occur at this point (as I've mentioned), it was necessary to use synchronization on this part of the code. Because of this serialization, all the processes using the volunteer collection will have to wait for the draw operation to complete before continuing. However, since the draw operation is not very computationally expensive, it finishes quickly and the interruption is not visible (there are no recognizable jolts in the simulation).
Although there is nothing that can be done to prevent statistical anomalies, I will discuss the situations where the probabilities we use in the simulation can by cause a liveness problem to occur:

- Sam has a 40% chance of eliminating a volunteer every $10ms$. Although highly unlikely, the possibility of Sam missing volunteers for an extended period of time does exist. From a statistical point of view this case can be described by $P(t) = (1 - \frac{40}{100})^t$ where $t \in \mathbb{N}$ is the number of trials where Sam missed. In this event the number of volunteers may increase till it reaches v_{max}, at which point the simulation will appear to be frozen.
- A similar situation can arise when no new volunteers are added to the simulation. The probability of this can be described by $P(t) = (1 - \frac{10}{100})^t$ where $t \in \mathbb{N}$.

5 Output of the program

5.1 Raw data

The following data has been gathered from experiments, where the threshold, v_{max}, has been set to 5, 10 and 100 respectively. Note that $v_{eliminated}$ is the number of volunteers Sam has eliminated during the trial and that throughput is calculated as $\frac{v_{eliminated}}{\Delta t}$.

Trial	$v_{max} = 5$			$v_{max} = 10$			$v_{max} = 100$		
	$v_{eliminated}$	Δt	Throughput	$v_{eliminated}$	Δt	Throughput	$v_{eliminated}$	Δt	Throughput
1	493	$15.7s$	31.21	484	$14.1s$	34.25	469	$13.1s$	35.64
2	471	$14.8s$	31.69	508	$16.1s$	31.54	477	$13.1s$	36.33
3	485	$15.0s$	30.48	520	$15.3s$	33.87	495	$13.5s$	36.55
4	500	$16.2s$	30.77	511	$15.7s$	32.48	507	$13.6s$	37.17
5	499	$16.8s$	29.70	516	$15.3s$	33.54	521	$14.4s$	36.14
Σ	2448	$78.5s$	153.85	2539	$76.5s$	165.68	2469	$67.7s$	181.83
$\overline{x}$	489.6	$15.1s$	30.7	507.8	$15.3s$	33.1	493.8	$13.5s$	36.3

5.2 Interpretation

First of all I would like to note that the data above does not contain any outliers, which may influence the averages that were calculated.
Adding more volunteers to the room did have a negative impact on the time Sam was able to survive. This was anticipated, because Sam wouldn't have been able to eliminate all the volunteers closest to her (these volunteers were more likely to hit Sam) and would be struggling to keep the number of volunteers on the court to a small group.
If the changes weren't taken into account, Sam's survival time would have stayed approximately the same during all the trials (it would not have depended on v_{num} and would be far longer).
The fluxuations in throughput (there is a tendency of the throughput increasing with a higher v_{max}) can be explained by taking into account the fact that a smaller group of volunteers will be completely eliminated by Sam ($v_{num} = 0$), more often than the case with a larger group. In other words, if there are no volunteers in the room more often, then Sam's throughput will decrease while she is waiting for volunteers to be let in.