Operating Systems Coursework 1

By Gines Moratalla

November - 2023

Keywords: Sensor (SS), Analysis (AA), Actuator (CC), Probability of generating k tasks (k_prob), Robot Controller Problem (T1), Multiple Sensors Problem (T2), Poisson Distribution (PD).

1 Main Task (Robot controller)

The goal of this report is to describe the approach taken to solve the robot controller problem with the use of concurrent programming in Java.

1.1 Problem analysis

I first assumed the whole process of moving the robot as a single thread that would go "jumping" through the different components of the controller (*SS, AA, and CC*). I then realized that this would mean only one single "process" would be running at a time, therefore, it would need to fully end to act on more tasks, so it would not be making use of concurrency correctly.
This explains my second approach, where I thought of each component as an individual thread (Java class implementing runnable object), each of which would perform its respective operations concurrently. Along with this, I should handle the multithreading via a "workflow" class that instantiates the threads and runs as the main robot controller simulator.

1.2 Queuing tasks

Following the structure of the Producer-Consumer problem from the Operating Systems lecture (LZSCC211, 4.2, Stovold), to not lose generated tasks, I thought of two blocking queues where one queue would link the *SS* with the *AA*, and the other, *AA* with *CC*. The enqueueing and dequeueing are found in the run() methods for the different threads, using synchronized statements, so this way each thread "blocks" the access to it, thus, avoiding threading problems such as deadlock or mutual exclusion.
I decided not to use any already existing Java library that implemented a blocking queue, as I assumed this would be solving part of the problem. Instead, I limit a normal queue's buffer (In this case to 6) and wait (with proper error handling via stdout in case of not being able to generate, analyse or process) until they have space available.

1.3 Sensor

The *SS* class has a unique **id** for every task generated, complexity **c**, (with random normal distribution), where $0.1 \leq c \leq 0.5$, and two elements passed from the workflow; these are **lambda** (given as user input), and the first blocking queue that will add the generated tasks. The run method for this thread would first generate k tasks (*PD*), and then it would add these to the queue one by one. In case of a full queue, the synchronized statement would notify when it has available space + print saying it has too many tasks to analyse. I then put the thread to sleep for 1 second, to simulate one task batch generated per second.

1.4 Analyser

For the *AA* class, I included the same blocking queue (same reference) as the *SS* class, as well as a reference to the second queue (*AA - CC*). The way I built it is so that tasks from the first queue are dequeued, then the thread sleeps for the set amount in milliseconds w.r.t. task complexity ($c \cdot 1000$ ms) simulating the analysis, to later be enqueued to the second queue right after sleeping.
This thread would access both queues again on their synchronized statements, with a similar error handling (stdout print) avoiding any potential deadlock/mutual exclusion.

1.5 Actuator

The *CC*, as followed above, will dequeue any task that has already been analysed by the *AA*, and will perform the necessary operations, this is, calculating the distance to move given by $\Upsilon = \sqrt{\frac{1}{c}}$, in the **converResult()** method. Later, to tackle the "rebounding" problem, I built a recursive function called **calculateResult()** that keeps subtracting/adding the distance helped by a boolean variable that gives robot facing direction (changes when exceeded either 1 or 0). Recursion helps when there is remaining space to move the robot, calling itself again with the remaining distance. The previous position of the robot is something that will reference only one thread of type *CC*, therefore, it will be updated correctly without the need for getters or setters to save the last position.

1.6 Worfklow

Workflow.java (changed bellow, **2.1**) class will act as the robot controller, instantiating, starting, and interrupting the different threads. The class first declares all the variables/structures and then asks the user (via some user input methods inside of it) for input on lambda and the initial position of the robot. The direction in which the robot starts facing is assumed to always be right, as nothing about it was mentioned.

1.7 Other relevant information

User input

User input, instead of being asked in the bash files, is asked within the program itself (Rig file to choose between *T1/T2* and Workflow classes for variables). I did so because I thought it was easier to modify and I could run the process several times with different values without the need to reset (also applies to *T2*).

Poisson distribution

PD, given by $P(k) = \frac{e^{-\lambda}\lambda^k}{k!}$, was achieved with the function **PoissonDistribution()** in the *SS* class. My approach was to create an array that would contain the different *k_prob*, where k is the array index. The solution I implemented to get an index of an array based on the elements in the array as probabilities was, setting a limit to 20 tasks (array size) to avoid crazy amounts being generated. Then, to choose k [0, 19), I used a random number that will try to match the k_prob with another variable that will count the accumulated sum of *k_probs* iterating through the array and choosing the k closest to it. The accumulated probability will ignore any random number bigger than it, so this would mean that the bigger the accumulated probability is, the bigger the chance for k_prob to be within its range. This would mean that at some point (end of the array), in case no k has been chosen, the accumulated probability will eventually be bigger than the random number (with the respective lambda and array size limitations).

Task tracking

In order to keep track of the task id for the last task analysed/processed in case an error needs to be outputted, I built some setter and getter methods in *AA* and *CC*, and whenever a thread object interacted with a non-empty/non-full queue, it updates the last task analysed/processed respectively.

Rig and Problem

In order to implement a menu interface to be used by the workflow as an initializer method for the whole process, I used (and edited respectively) the **Rig.java** and **Problem.java** provided in the Operating System Labs (Stovold, 2023). It now lets me choose between *T1* or *T2* as well as providing the input variables later. (Also applies for *T2*). As for the Problem interface, I use it to call *T1* and *T2* from the Rig file both as problems for the exercise, implementing the interface methods.

2 Second Task (Multiple sensors)

2.1 Problem analysis

For *T2*, I thought of using the same class for the multiple *SS* problem, but, since I had to work with a shared resource from the same type of thread (that being the first queue and task id), I decided to change the approach,

making a separate class for the new workflow **Worflow_Multiple.java** (That explains the change of name of the first task, now **Workflow_Simple.java**) that will be called as a second option from the Rig, this will instantiate the threads a bit differently from *T1*.

2.2 Worflow_Multiple

Workflow class includes another private class inside, **private class MultipleSensor**, basically identical to the *SS* used in *T1*, but it now includes an integer for the sensor id, that I will use as extra information for the printing statement when the robot is moved (This id will also be pulled from the dequeued task in the *CC*, just as task id and complexity are).

2.3 MultipleSensor

As justified above, to use the task id as a shared resource, I decided to implement the new *SS* class as private within the Workflow_Multiple file. This allowed me to increase the task id collectively rather than individually, with a separate counter per *SS*. The way this was done, in order to avoid multiple *SS* incrementing the task id at the same time, was surrounding the task id incrementation with a lock (synchronized keyword) so only one of the *SS* threads would increment the current task at a time. The rest of the structure is the same as *SS* in *T1*.

Order of queued tasks

When running *T2*, I observed that the order of the tasks printed when moving the robot was not always incrementing[1]. At first, I thought this was a problem, but later realized it was a result of the FCFS structure of three sensors enqueuing elements at the same time; only "locking" the id incrementation, so not every batch was following a processed order. This does not affect the total number of tasks generated and there is no mention of following an incrementing order on processed tasks by id, so I it was not changed.

3 Testing/Analysis

Pre-analysis testing was done to solve problems that were affecting the process, and the following documented changes were made:

3.1 Limit to lambda

A very high/low input in lambda (e.g. 60) was causing some runtime errors due to the PoissonDistribution() return statement, which is why I limited its range on **acceptLambda()**.

3.2 Tasks being lost (safe execution)

As the process was terminated with thread.interrupt(), the generation of tasks was stopped suddenly, so tasks were lost. To solve this, I made a safer execution for *T2* in which there is a boolean (condition for the *SS* while statement in the run() method) **system_exit** that is set to true after some seconds, which will cut the run method in the *SS* threads. Then all threads will be interrupted after an extra 6 seconds of analysing/processing the remaining queued tasks (It first joins *SS* threads and then interrupts, as an extra layer to make sure they finish). For *T1*, after some seconds, *SS* is interrupted and the other threads have an extra 6 seconds to finish, without any boolean condition, which wasn't perceived as necessary in *T1*, only having 1 *SS*.

3.3 Unsolved Task Tracking issue

For *T2*, I observed a case during the testing where the task id for certain tasks was repeated, causing 2/3 tasks to have the same id every so often. With some debugging (adding an extra task counter in the *CC*) I was able to confirm that it does not affect the total task count and it is not referring to the same task (different positions and complexities), however, I have not been able to find the source of the problem.

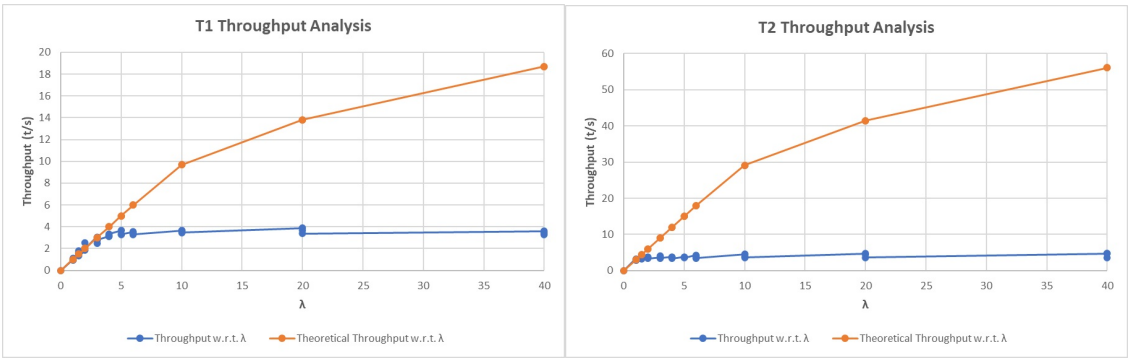
[1] ***Incrementing*** *refers to no backjumping (e.g., 1, 3, 0...).*

3.4 Throughput Analysis

Analysis was made by running the process several times with different inputs/values to measure the throughput. No impact on the throughout from the robot's initial position was noticed, so it was removed from the analysis. The chosen values then were the following:

- **Lambda**: different values in ascending order.
- **Seconds**: seconds the process was running.
- **Throughput** = (tasks generated / seconds running). *tasks per second (t/s).*
- **Theoretical Throughput**: Throughput without analysis sleep constraints. This parameter was created to simulate an ideal scenario, where the average k for each λ was calculated separately (with PD) as $\bar{k}_\lambda$, making it: **Theoretical Throughput** = $\bar{k}_\lambda \times$ Seconds Running $\times$ Number of Sensors

The following line charts (*Figure 1.a.* and *Figure 1.b.*) will show the results of the system throughput study with the parameters defined above.



(a) Figure 1.a (b) Figure 1.b

*Note: Turn to **Appendix No.1** in case of wanting to consult the table with the real data that has been used for these analysis charts.*

The charts helped me notice that there are a couple of elements causing the generated tasks to be limited by a bottleneck. The smaller (practically unremarkable) issue is the limited buffer in the queues, which makes the sensor(s) wait to generate more tasks until the analyser makes space.
This is followed directly by the big issue, which is the time slept simulating the **task analysis** (*given c, each task takes around 0.3 seconds on average to analyse, meaning that every* $\approx$ *3 tasks analyzed with a full queue, would lose 1 task generation*). If we were to remove this, our task production would look more or less like the Theoretical Throughput flow seen in *Figure 1.a* and *Figure 1.b*, since tasks would be analysed almost immediately, without time to fill up the queue.
To better understand the situation, we can see asymptotes in both charts *Figure 1.a* and *Figure 1.b*, which represent that, with the specific data used in this study, the real throughput does not exceed $4t/s$ in *T1* and $5t/s$ in *T2*, so adding more sensors isn't of much help.

In case of changing the data for the throughput study, if the this same structure is maintained, a similar conclusion would be reached, and, having stayed within the limitations set, and developed the most "ambiguous" parts of the circuit as I interpreted them, this is my personal understanding of what is going on with the robot controller and what are its limitations.

References

- **Stovold, J.** (2023). Lecture 4-2: Semaphores. In LZSCC.211: Operating Systems [PDF Document]. Retrieved from Moodle, Lancaster University Leipzig.
- **Stovold, J.** (2023). Operating Systems - LZSCC211, GitHub. Available at: `https://github.lancs.ac.uk/stovold/SCC211_AY23` (Accessed: 14 November 2023).

Appendix No.1 - Throughput Analysis Table (Excel)

For clarification, T2 was tested with 3 sensors here, so the result for that scenario is the following:

λ	Seconds	Tasks Generated	Throughput	Theoretical Throughput	Problem
0	20	0	0	0	T1
1	20	19	0,95	1	T1
1	30	33	1,1	1	T1
1	100	93	0,93	1	T1
1,5	20	33	1,65	1,5	T1
1,5	30	54	1,8	1,5	T1
1,5	100	134	1,34	1,5	T1
2	20	37	1,85	2	T1
2	30	76	2,5333333	2	T1
2	100	199	1,99	2	T1
3	20	61	3,05	3	T1
3	30	74	2,4666667	3	T1
3	100	277	2,77	3	T1
4	20	63	3,15	4	T1
4	30	93	3,1	4	T1
4	100	334	3,34	4	T1
5	20	73	3,65	5	T1
5	30	98	3,2666667	5	T1
5	100	333	3,33	5	T1
6	20	71	3,55	6	T1
6	30	103	3,4333333	6	T1
6	100	329	3,29	6	T1
10	20	73	3,65	9,7	T1
10	30	108	3,6	9,7	T1
10	100	345	3,45	9,7	T1
20	20	78	3,9	13,8	T1
20	30	110	3,6666667	13,8	T1
20	100	337	3,37	13,8	T1
40	20	72	3,6	18,7	T1
40	30	104	3,4666667	18,7	T1
40	100	330	3,3	18,7	T1
0	20	0	0	0	T2
1	20	65	3,25	3	T2
1	30	87	2,9	3	T2
1	100	281	2,81	3	T2
1,5	20	67	3,35	4,5	T2
1,5	30	110	3,6666667	4,5	T2
1,5	100	326	3,26	4,5	T2
2	20	76	3,8	6	T2
2	30	110	3,6666667	6	T2
2	100	331	3,31	6	T2
3	20	72	3,6	9	T2
3	30	120	4	9	T2
3	100	343	3,43	9	T2
4	20	75	3,75	12	T2
4	30	112	3,7333333	12	T2
4	100	336	3,36	12	T2
5	20	76	3,8	15	T2
5	30	114	3,8	15	T2
5	100	356	3,56	15	T2
6	20	85	4,25	18	T2
6	30	119	3,9666667	18	T2
6	100	343	3,43	18	T2
10	20	91	4,55	29,1	T2
10	30	119	3,9666667	29,1	T2
10	100	358	3,58	29,1	T2
20	20	95	4,75	41,4	T2
20	30	135	4,5	41,4	T2
20	100	359	3,59	41,4	T2
40	20	95	4,75	56,1	T2
40	30	142	4,7333333	56,1	T2
40	100	363	3,63	56,1	T2

Figure 2: Analysis table, made in Excel.