Synchronization: CalTrainII Automation

MCO2 Final Technical Report - INTR-OS
August 6, 2018

Ato, Paolo Miguel
Student
2401 Taft Ave, Malate,
Manila, 1004 Metro Manila
+639954887524
paolo_ato@dlsu.edu.ph

Manzano, Joshua
Student
2401 Taft Ave, Malate,
Manila, 1004 Metro Manila
+639178949025
josh_manzano@dlsu.edu.p
h

Tinsay, Shenn
Margareth
Student
2401 Taft Ave, Malate,
Manila, 1004 Metro Manila
+639175025538
shenn_tinsay@dlsu.edu.ph

ABSTRACT
The purpose of the research is to analyze which process synchronization technique is the most efficient. The two techniques used are through Monitors and Semaphores. The test results will then be analyzed in order to form a conclusion and find a solution to the problem.

KEYWORDS
Synchronization, Locks, Monitors, Semaphores

1 INTRODUCTION

There are different modes of transportation present in the world today. An individual can ride a bus, a car, or even a bike towards their desired destination from where they currently are. One of the most evident modes of transportation would be a train. According to the Oxford Dictionary, a train is "a series of connected railway carriages or wagons moved by a locomotive or by integral motors. [1]" It runs on a designated track that could lead to anywhere. When there are trains, there will always be train stations along its track where passengers can go on and off the vehicle. With the information stated, an example of an implementation of the train system would be CalTrainII.

CalTrainII is an automation consisting of 8 train stations and it dispatches at most 16 trains. The number of trains being dispatched is depended on passenger demand. The details given in this project is that the passengers of the trains are robots and each train and passenger is controlled by a thread.

This technical report aims to differentiate the performance of process synchronization methods. To acquire the data, the researchers are tasked to think of solutions to synchronize the arrival and departure of each train in the station as well as the waiting, boarding, and departure of each passenger on and off the train through semaphores, locks, and monitors (and condition variables). Two separate programs were made to simulate the train automation system and java was the language that was used. Through using the stated process, the researchers would be able to compare and contrast to see the results that could lead to the conclusion on which process would yield the most efficient solution to the problem presented by CalTrainII.

2 PROCESS SYNCHRONIZATION

To solve the problem, there were two process synchronization techniques implemented: one using Monitors and the other using Semaphores.

The following classes were created for the simulation:

- *CalTrain.java* is responsible for generating trains and passengers.
- *Condition.java* represents the condition variable.
- *ControlBar.java* is the GUI representation of the controls.
- *Interface.java* is the main component of the GUI.
- *Lock.java* represents the lock.
- *Main.java* contains main class for execution.
- *Monitor.java* represents the monitor.
- *Passenger.java* is the runnable passenger model.
- *Station.java* is responsible for synchronizing trains and passengers.
- *Train.java* is the runnable train model.
- *TrainStations.java* is the GUI component.

2.1 Functions and Locks for Monitors (Condition Variables). To explain monitors, according to the article about Multithreading, monitors are a synchronization construct that allows threads to have mutual exclusion with the use of locks [4]. Monitors also help all the threads. Locks on the other hand help the threads so that they work independently on the data without interfering with each other [4].

2.1.1 Classes. The following classes are made to implement our solution:

- *Monitor.java* - contains various integer variables, a lock to ensure mutual exclusion, and a condition variable where passengers will execute wait.
- *Condition.java* - contains two integer variables to represent wait and signal.
- *Lock.java* - only contains one integer, which makes it act like a binary semaphore.

The following classes are used to achieve synchronization:

- *Station.java* - contains a lock for entering trains, a monitor for passenger waiting, and a condition variable for unloading trains.
- *Train.java* - contains a lock and eight condition variables for the purpose of unloading passengers.

2.1.2 Implementation. For the implementation of locks and condition variables, the *synchronized* block statement was used for certain methods and blocks of code to achieve mutual exclusion. The keyword *synchronized* only allows a certain method or block of code to be executed by one thread only at any given time. This prevents data inaccuracies and race conditions.

For the implementation of the monitors, both locks and condition variables were used to ensure mutual exclusion and proper waiting on condition.

Once the required classes and variables are initialized, the simulation can start. Each arriving passenger will go to their station and subsequently enter the station's monitor and wait at its condition variable if there are no trains available. A passenger will only board a train if the condition variable is signaled.

Once a train arrives at a station, it first attempts to acquire the lock of the station then signals one of the condition variables it contains (the eight condition variables) to unboard any passengers who should be unboarding at that station. After all passengers have been unboarded and the train still has remaining seats, it then enters the station's monitor and signals its condition variable to wake up any waiting passengers who will be boarding the train. It leaves the station and releases the lock when all the passengers have been boarded or if it runs out of remaining seats.

After the passengers wake up and board the train, they will then wait on one of the train's condition variables depending on their destination. Once they are at the destination, they will be signaled by the train to unboard.

With this solution, we ensure that mutual exclusion is always present whether it be the trains or the passengers. The use of condition variables ensure that the code will not result into busy-waiting.

2.2 Semaphores. To further elaborate on Semaphores, based on an article on Operating Systems found in Geeks for Geeks, they are simply considered as variables [2]. To achieve process synchronization in a multi-threaded environment these are used in solving critical section problems [2]. There are different types of semaphores and the two most common kinds are counting semaphores and binary semaphores. The difference between the two would be counting semaphores can take integers that are not negative while binary semaphores can only get the values of 0 and 1 [2].

2.2.1 Classes. The Semaphore class from java.util.concurrent.Semaphore was used to implement our solution. It has the methods acquire() and release(), making it suitable for achieving mutual exclusion and data accuracy.

2.2.2 Implementation. Similar to our implementation using monitors, we use Semaphores to ensure mutual exclusion and waiting. The difference is instead of using *synchronized* blocks, we made use of Semaphore constructs that have the methods acquire() and release(), which can ensure that only one thread utilizes a certain part of code at any given time.

Our implementation with semaphores is fundamentally the same to our previous implementation. The biggest difference is only the constructs used and the number of lines of code utilized.

In terms of constructs and lines of code used, using semaphores significantly made the implementation simpler and easier to understand. Having said that, monitors and condition variables still provided more flexibility and modularity than semaphores.

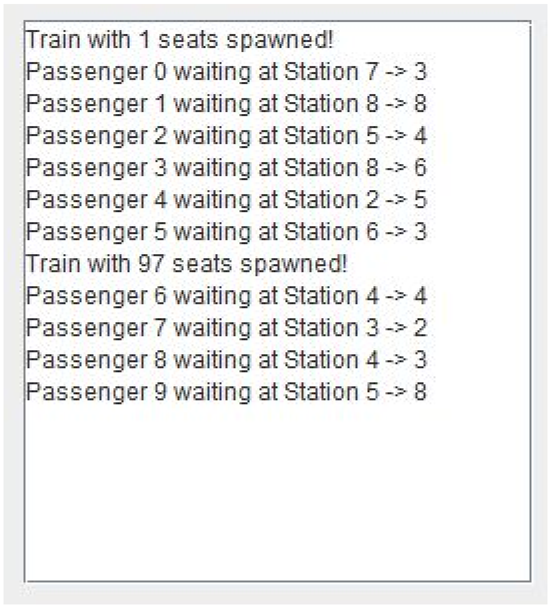
3 RESULTS AND ANALYSIS

Through testing on multiple cases and research [3], the team has realized that there is indeed a difference between monitors and semaphores and these are the findings:

MONITORS	SEMAPHORES
Utilizes the Java method *Synchronized*	Utilizes the Java object *Semaphore*
Has condition variables	Does not have condition variables
Abstract data type	Integer variable S

To further elaborate on their difference, the following scenarios were considered in order to acquire information that may lead to conclusions in comparing and contrasting the performance of the two processes that were used in synchronizing the processes in the implementation of the CalTrainII Automation.

3.1 When a passenger arrives at the train station while the train is waiting for passengers.

Sample Input


```
Train with 1 seats spawned!
Passenger 0 waiting at Station 7 -> 3
Passenger 1 waiting at Station 8 -> 8
Passenger 2 waiting at Station 5 -> 4
Passenger 3 waiting at Station 8 -> 6
Passenger 4 waiting at Station 2 -> 5
Passenger 5 waiting at Station 6 -> 3
Train with 97 seats spawned!
Passenger 6 waiting at Station 4 -> 4
Passenger 7 waiting at Station 3 -> 2
Passenger 8 waiting at Station 4 -> 3
Passenger 9 waiting at Station 5 -> 8
```

MONITORS	SEMAPHORES
Trial 1: A passenger arrived at 7690 ms while a train was waiting.	Trial 1: A passenger arrived at 6600 ms while a train was waiting.
Trial 2: A passenger arrived at 6870 ms while a train was waiting.	Trial 2: A passenger arrived at 6790 ms while a train was waiting.
Trial 3: A passenger arrived at 5840 ms while a train was waiting.	Trial 3: A passenger arrived at 5360 ms while a train was waiting.

Figure 1

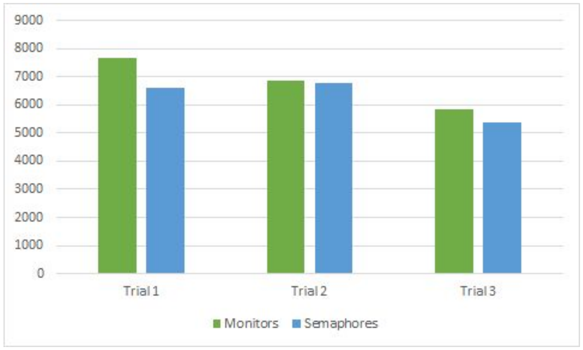
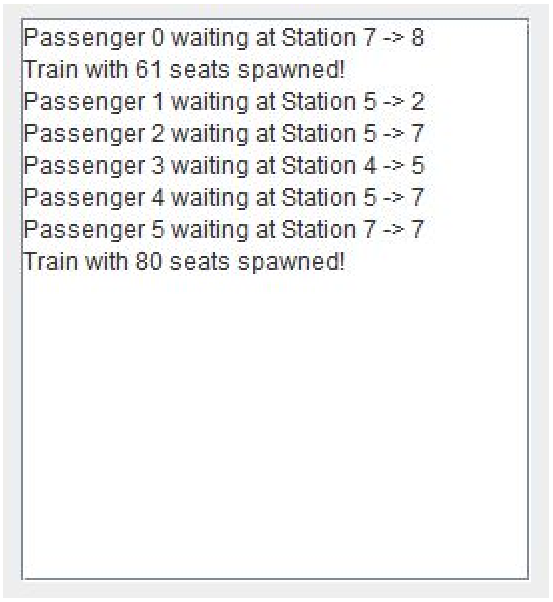


Figure 2

Figure 1 shows a sample input of when a train was waiting at a station which was letting passengers on board it also shows the values of the three trials done in this test case. In this case, the time was considered when a passenger arrived yet the train has already left the station. In relation to Figure 1, Figure 2 is a graphical representation of the values in the previous figure. When looking at it, it is clearly shown that Monitors took a longer period of time than semaphores in this test case. The average completion time of the Monitors are 6800 ms while the one for Semaphores was 6250.

3.2 When a train arrives at the train station and there are no passengers waiting.

Sample Input



```
Passenger 0 waiting at Station 7 -> 8
Train with 61 seats spawned!
Passenger 1 waiting at Station 5 -> 2
Passenger 2 waiting at Station 5 -> 7
Passenger 3 waiting at Station 4 -> 5
Passenger 4 waiting at Station 5 -> 7
Passenger 5 waiting at Station 7 -> 7
Train with 80 seats spawned!
```

MONITORS	SEMAPHORES
Trial 1: The train arrived at a station with no passengers waiting at 4980 ms.	Trial 1: The train arrived at a station with no passengers waiting at 4460 ms.
Trial 2: The train arrived at a station with no passengers waiting at 11630 ms.	Trial 2: The train arrived at a station with no passengers waiting at 4440 ms.
Trial 3: The train arrived at a station with no passengers waiting at 6700 ms.	Trial 3: The train arrived at a station with no passengers waiting at 3950 ms.

Figure 3

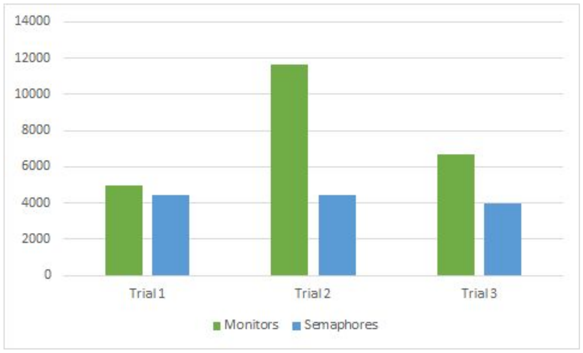
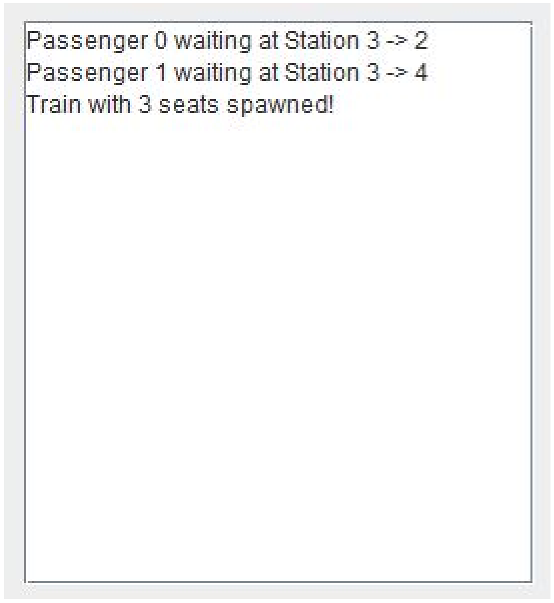


Figure 4

For this test case, we observed the fastest time as to when the train reaches a station with no waiting passengers (which can be seen in Figure 3). The average time the trials with the version that used Monitors was 7770 ms while the average time for the version that used Semaphores was 4283.333 ms. Through analyzing Figure 4, it can be seen that at the second trial, the one with Monitors took a longer period of time than the one with Semaphores before it reached a station without any passengers waiting.

3.3 When passengers board the train at the same station at the same time.

Sample Input



```
Passenger 0 waiting at Station 3 -> 2
Passenger 1 waiting at Station 3 -> 4
Train with 3 seats spawned!
```

MONITORS	SEMAPHORES
Trial 1: There were waiting passengers who boarded the train at the same time at 23510 ms.	Trial 1: There were waiting passengers who boarded the train at the same time at 11400 ms.
Trial 2: There were waiting passengers who boarded the train at the same time at 6690 ms.	Trial 2: There were waiting passengers who boarded the train at the same time at 5330 ms.
Trial 3: There were waiting passengers who boarded the train at the same time at 10420 ms.	Trial 3: There were waiting passengers who boarded the train at the same time at 9450 ms.

Figure 5

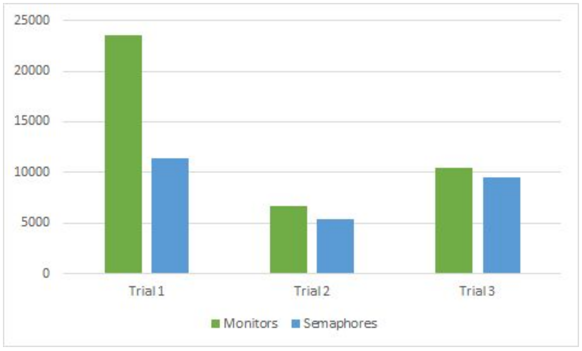
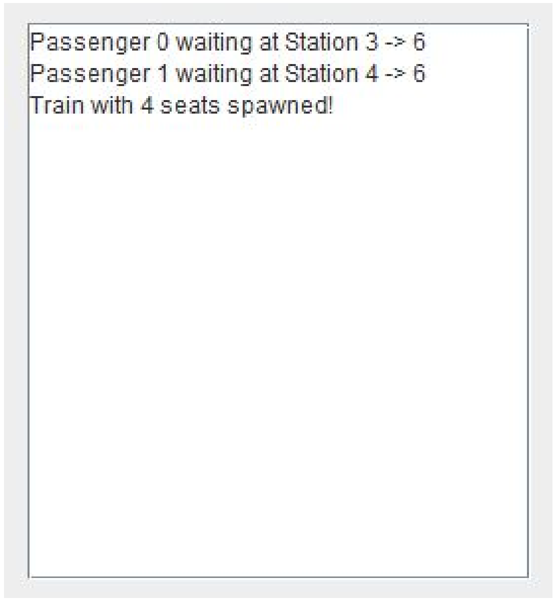


Figure 6

Similar to the previous test cases, Figure 5 contains the sample log of one of the trials tested together with the time measured. This test case focuses on waiting passengers on a certain station who board the train at the same time. Figure 6 shows a graphical representation of the data found in Figure 5. It can also be analyzed that Monitors took a longer period of time than Semaphores in synchronizing processes in this test case. The average score of the trials with the Monitor was 13540 ms while the Semaphore version yielded an average of 8726.667 ms which further explains that Monitors took longer than Semaphores.

3.4 When passengers leave the train at the same station at the same time.

Sample Input



```
Passenger 0 waiting at Station 3 -> 6
Passenger 1 waiting at Station 4 -> 6
Train with 4 seats spawned!
```

MONITORS	SEMAPHORES
Trial 1: At 21990 ms, there are passengers that left the same train at the same time.	Trial 1: At 16520 ms, there are passengers that left the same train at the same time.
Trial 2: At 34100 ms, there are passengers that left the same train at the same time.	Trial 2: At 22330 ms, there are passengers that left the same train at the same time.
Trial 3: At 30740 ms, there are passengers that left the same train at the same time.	Trial 3: At 24010 ms, there are passengers that left the same train at the same time.

Figure 7

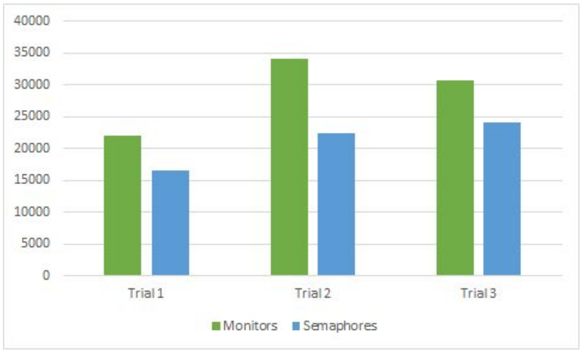
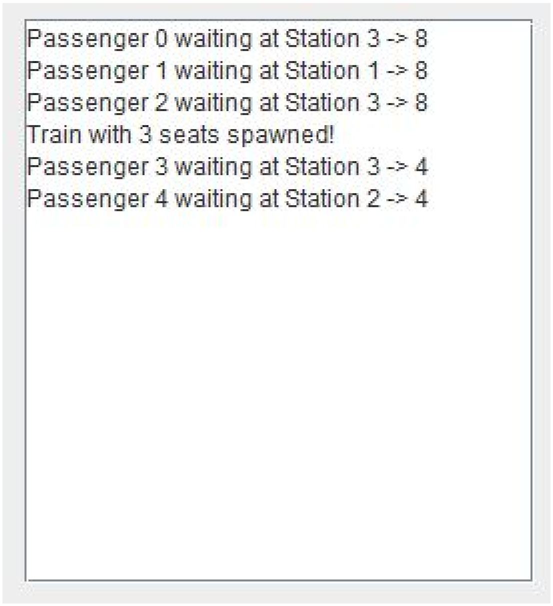


Figure 8

This test case depicts the scenario where passengers leave the train at the same station at the same time. Seen in Figure 7 is the tabular form of the data collected through timing each trial. Figure 8 depicts its graphical representation similar to what was done in the previous test cases. It can be seen as well that Monitors took longer than Semaphores. The average time Monitors took in executing this scenario was 28943.33 ms while the other technique implemented, using Semaphores, had an average time of 20953.33 ms.

3.5 When passengers board and leave the train at the same station at the same time.

Sample Input


```
Passenger 0 waiting at Station 3 -> 8
Passenger 1 waiting at Station 1 -> 8
Passenger 2 waiting at Station 3 -> 8
Train with 3 seats spawned!
Passenger 3 waiting at Station 3 -> 4
Passenger 4 waiting at Station 2 -> 4
```

MONITORS	SEMAPHORES
Trial 1: It took at least 36740 ms for passengers to board and leave the same train station at the same time.	Trial 1: It took at least 33450 ms for passengers to board and leave the same train station at the same time.
Trial 2: It took at least 35780 ms for passengers to board and leave the same train station at the same time.	Trial 2: It took at least 34890 ms for passengers to board and leave the same train station at the same time.
Trial 3: It took at least 29860 ms for passengers to board and leave the same train station at the same time.	Trial 3: It took at least 28320 ms for passengers to board and leave the same train station at the same time.

Figure 9

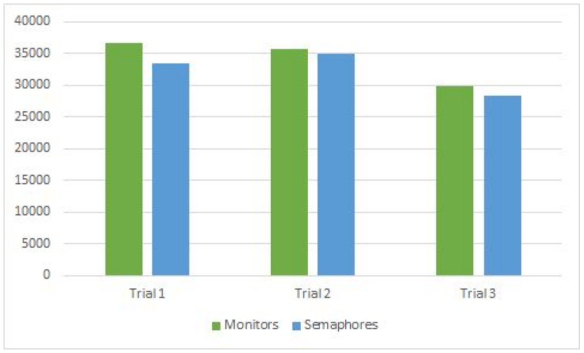


Figure 10

The results presented in Figure 9 for this test case was significantly longer than the previous ones. The scenario presented in this test case was when passengers board and leave the same train station at the same time. Following the logic of the program, this will really yield longer times since it needs to go through all the train station again before it reaches itself. In this case, the Monitor still took the longer time in executing this scenario having an average of 34126.67 ms while the version that used Semaphores had an average time of 32220 ms.

4 CONCLUSION

Synchronization is significantly important in everything that we do. In a multi-threaded environment, it is essential that synchronization is practiced, otherwise the threads will output significantly wrong data and disrupt the flow of the program. In the two implementations analyzed, semaphore was marginally faster than monitors, due to their simplicity. In addition, the amount of instructions used to write the semaphore implementation was significantly lower than the monitor implementation. Therefore, it can be said that semaphores are faster and simpler, while monitors are slower and more modular and complex.

5 REFERENCES

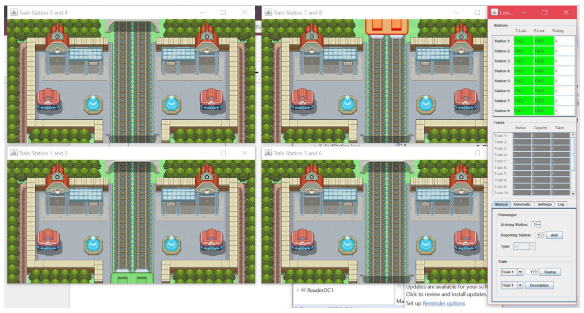
[1] Oxford Dictionary. (N/A). *Train*. Retrieved on August 5, 2018 from https://en.oxforddictionaries.com/definition/train.

[2] GeeksforGeeks. (2018). *Operating Syste, | Semaphores in Operating System*. Retrieved on August 6, 2018 from https://www.geeksforgeeks.org/semaphores-operating-system/.

[3] TechDifferences. (2017). *Difference Between Semaphore and Monitor in OS*. Retrieved on August 6, 2018 from https://techdifferences.com/difference-between-semaphore-and-monitor-in-os.html.

[4] Gupta, L.(2016, February 09). Multithreading - Difference between lock and monitor in java. Retrieved on August 6, 2018 from https://howtodoinjava.com/core-java/multi-threading/multithreading-difference-between-lock-and-monitor/.

6 APPENDIX



Stations

	T-Lock	P-Lock	Waiting
Station 1:	FREE	FREE	0
Station 2:	FREE	FREE	0
Station 3:	FREE	FREE	0
Station 4:	FREE	FREE	0
Station 5:	FREE	FREE	0
Station 6:	FREE	FREE	0
Station 7:	FREE	FREE	0
Station 8:	FREE	FREE	0

Trains

	Station	Capacity	Taken
Train 1:	0	0	0
Train 2:	0	0	0
Train 3:	0	0	0
Train 4:	0	0	0
Train 5:	0	0	0
Train 6:	0	0	0
Train 7:	0	0	0
Train 8:	0	0	0
Train 9:	0	0	0
Train 10:	0	0	0

Manual | Automatic | Settings | Log

Passenger
Arriving Station: 1
Departing Station: 1 Add
Type: 2B

Train
Train 1 1 Deploy
Train 1 Demobilize

Figure 11

Figure 11 shows what the user sees when the program is ran.

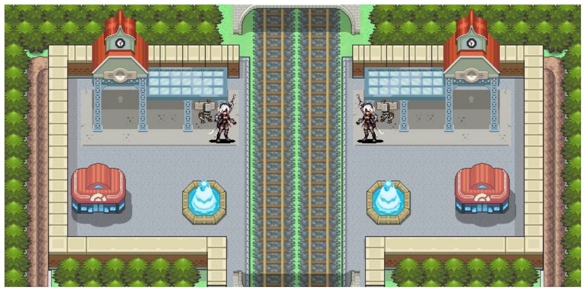


Figure 12

Figure 12 shows passengers waiting at designated train stations.

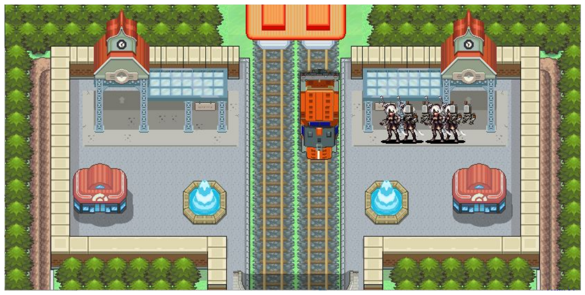


Figure 13

Figure 13 shows a train going to a station.

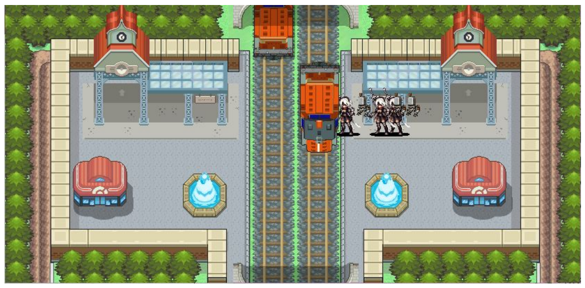


Figure 14

Figure 14 depicts the scenario that passengers are boarding the train.

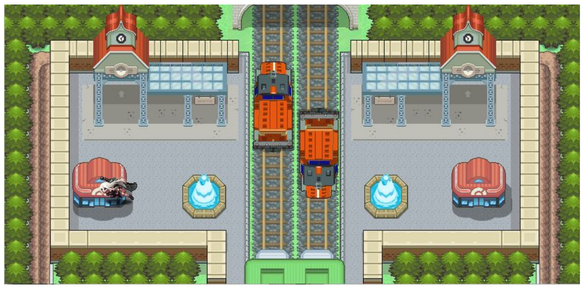
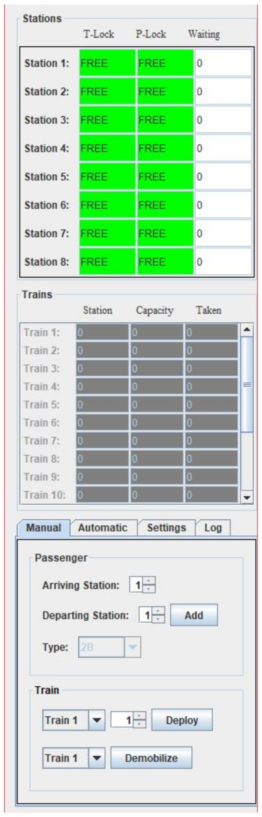


Figure 15

Figure 15 represents the scenario where a passenger just got off the train.



Stations

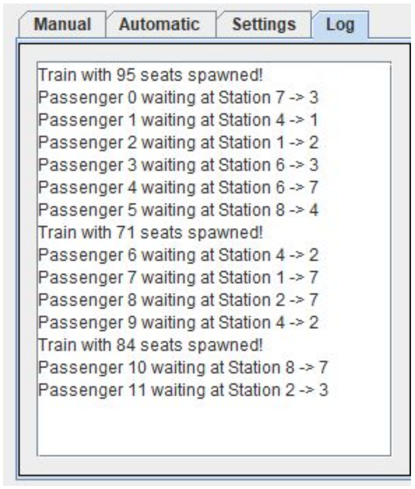
	T-Lock	P-Lock	Waiting
Station 1:	FREE	FREE	0
Station 2:	FREE	FREE	0
Station 3:	FREE	FREE	0
Station 4:	FREE	FREE	0
Station 5:	FREE	FREE	0
Station 6:	FREE	FREE	0
Station 7:	FREE	FREE	0
Station 8:	FREE	FREE	0

Trains

	Station	Capacity	Taken
Train 1:	0	0	0
Train 2:	0	0	0
Train 3:	0	0	0
Train 4:	0	0	0
Train 5:	0	0	0
Train 6:	0	0	0
Train 7:	0	0	0
Train 8:	0	0	0
Train 9:	0	0	0
Train 10:	0	0	0

Figure 16

Figure 16 shows the control bar of the simulation of the CalTrainII Automation.



```
Train with 95 seats spawned!
Passenger 0 waiting at Station 7 -> 3
Passenger 1 waiting at Station 4 -> 1
Passenger 2 waiting at Station 1 -> 2
Passenger 3 waiting at Station 6 -> 3
Passenger 4 waiting at Station 6 -> 7
Passenger 5 waiting at Station 8 -> 4
Train with 71 seats spawned!
Passenger 6 waiting at Station 4 -> 2
Passenger 7 waiting at Station 1 -> 7
Passenger 8 waiting at Station 2 -> 7
Passenger 9 waiting at Station 4 -> 2
Train with 84 seats spawned!
Passenger 10 waiting at Station 8 -> 7
Passenger 11 waiting at Station 2 -> 3
```

Figure 17

Figure 17 shows the log of when passengers are waiting at train stations and when trains start to roam around the tracks.