Detecting Race Conditions and Deadlocks in the Train Game for Concurrent Programming Education

s1270112 Shota Kamei, Supervisor: Prof. Maxim Mozgovoy

Abstract

Parallel programming is difficult to master. That's because it requires understanding many abstract concepts such as threads, processes, race conditions, and deadlocks. Also, in asynchronous programming, the order of process execution changes with each execution, making debugging very difficult. To address these issues, we are developing a 2D game using a train metaphor that allows users to learn parallel programming in a fun and intuitive way. Each element of the game is carefully designed to map to a specific concept in parallel programming, making it easy to transfer knowledge from the game to the real world. However, our game lacked a strict validation mechanism. In this work, our goal is to correctly detect crashes and deadlocks. By implementing the correct detection mechanism, we believe that our game will be able to provide correct feedback to students, which will lead to improved learning efficiency.

1. Introduction

1.1. Purpose and goals

When learning something, it is very important to have a fun learning environment. We are developing a 2D game that uses a railroad metaphor to help users learn the basics of parallel programming in a fun way. [1] Each element of the game corresponds to a specific concept of parallel programming, allowing users to learn in a natural way. [2] For example, a train corresponds to the instruction pointer, and placing multiple trains can represent multiple threads. Switches and crossings correspond to semaphores, [3] and can represent resource occupation. Basic stages exist by default, allowing users to quickly gain knowledge. More advanced users can also edit the stages themselves to deepen their understanding. The game is primarily targeted at students taking parallel programming courses.

Accurate validation is crucial to provide students with the right feedback. The most frequent need is to detect race conditions, which in our game are mainly crashes and deadlocks. A race condition is a situation in which multiple threads access a shared resource, and the order in which they access it has a significant effect on the execution result. Since the execution result depends on the order in which each thread executes, debugging is very difficult. However, since the conditions for a race condition to occur depend on the initial positions and speeds of the trains, it is not easy to detect it in advance. A crash is simply a collision between trains, and it has significant meaning in the game. A deadlock is a situation in which two or more threads occupy a shared resource, and each thread continues to request the other's locked resource. When a deadlock occurs, the system comes to a complete halt. In this work, we discuss crashes and specific methods for detecting deadlocks.

1.2. Game Description

Before we get into the main topic, I would like to explain the basic elements of our game.

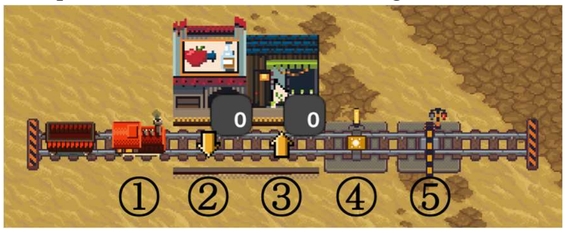

Fig. 1. Numbering in-game elements.

- Train (1) It consists of a locomotive and cars. Items can be loaded onto the cars, and when they pass in front of the shop (2) or the warehouse (3), they will receive or hand over items.

- Shop (2) Shops supply items to trains. The type and number of items the shop can provide to your train is displayed in the bottom right corner of the shop. A train can receive items from the shop as many times as there are cars in the train.
- Warehouse (3) The warehouse receives items from the train. The type and number of items that can be received from the train are displayed in the bottom right of the warehouse, and the game is cleared when all warehouses are filled with items.
- Button (4) Available in red, blue and yellow. When a train passes over a button it will open the crossing of the same color as the button.
- Crossing (5) Available in red, blue and yellow. At the start of the game the crossings are closed and trains cannot pass through. When the corresponding colored button is stepped on, the crossing opens, allowing trains to pass through.

2. Method

2.1. Crash Definition

A crash is a train collision, and in the game, it is the main cause of defeat. A crash is one of the results of a race condition, and whether a crash occurs depends on the initial position and speed of each train.

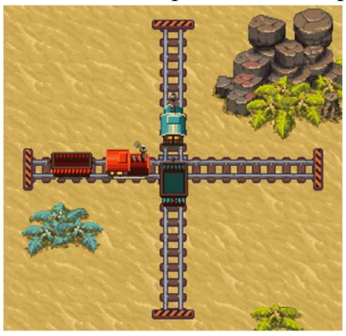
Fig. 2. Shortly afterwards, a train crash occurs.

For example, in a particular stage, a crash occurs if the blue train is slightly slower than the red train, as shown in Figure 3.

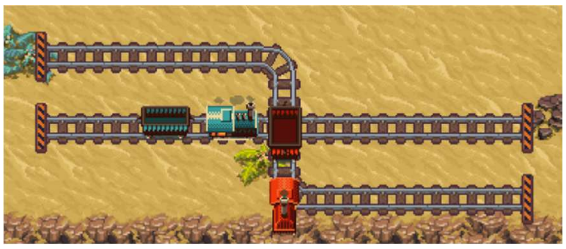
Fig. 3. For example, if the blue train is slow, a crash will occur.

However, a crash does not occur when the blue train is significantly faster than the red train, as in Figure 4.

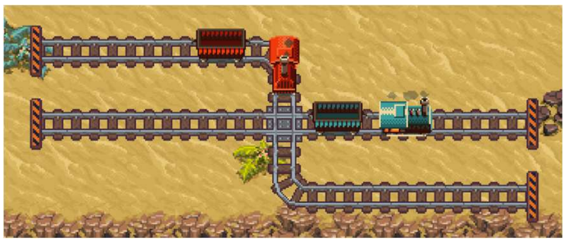
Fig. 4. For example, if the blue train is enough fast, no crash will occur.

In the context of the metaphor, the speed of the train corresponds to the speed at which a program runs in the real world. In real life, you can't be sure that a program will finish executing at exactly a specific time. If you run the same program multiple times, the time it takes to complete will likely vary slightly. It might be affected by other programs running at the same time, or by hardware resource allocation. To reflect these factors, in the version of the game I use, the train's speed is initialized randomly.

2.2. Crash Detection Implementation

There are a huge number of combinations of train speeds, and we can also increase the number of trains. Therefore, it is very difficult to check all the patterns. I decided to use a stochastic method to detect crashes. I run a sufficient number of simulations with trains initialized to random speeds and observe whether train collisions occur. If a collision is observed even once, a crash is detected. With this method, even if no collisions occur during multiple simulations, we cannot completely rule out the possibility of a crash, because we have not examined all possible patterns. However, if we calculate a confidence interval and find that the probability is very low, we can determine whether a crash exists or not at a level that is practically useful.

2.3. Deadlock Definition

A deadlock refers to a state in which multiple threads hold shared resources and each other requests the locked resources. For example, consider a situation in which thread A holds resource A and requests resource B. However, thread B holds resource B and is waiting for resource A to be released. In this case, each thread will be waiting for the locked resource, and threads A and B will be unable to proceed. In our game, we can recreate deadlocks using switches and crossings. All switches are at the end of the crossing, and neither train can open the crossing.

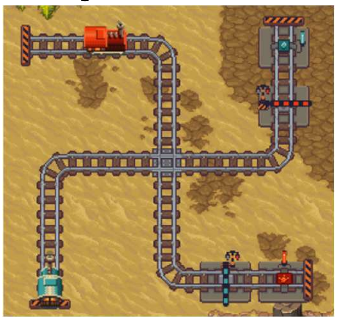

Fig. 5. The simplest case involving a deadlock.

2.4. Deadlock Detection Implementation

Unlike crashes, deadlock detection employs a mathematical model. [4] Nodes can store information about all trains and all crossings, and can record snapshots of the game board. The initial state is stored in the initial node, and the state transitions to a new node every time the train moves one tile. A depth-first search is used to explore all situations, allowing rigorous deadlock verification. If the search in one direction is unsuccessful, it is possible to roll back to the parent node and try another direction.

3. Result and Discussion

3.1. Crash

We tested in Stage A and Stage B, which have the potential for collisions.

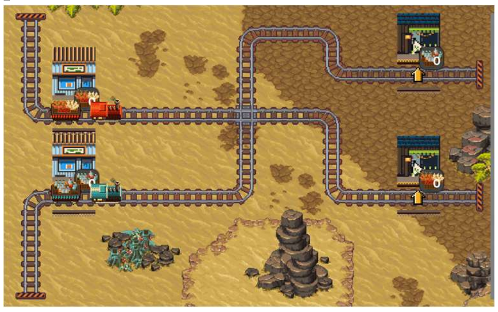

Fig. 6. Stage A: Crashes are relatively easy to detect.

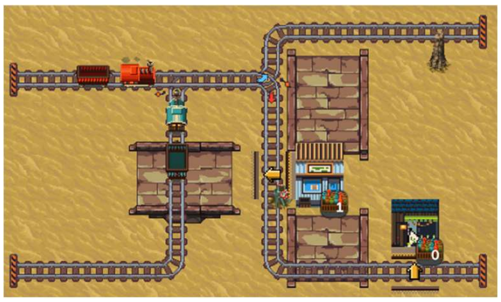

Fig. 7. Stage B: Validation is relatively difficult.

We simulated each 600 times, and the number of times and probability that a crash did not occur are shown in Table 1.

	Number of times crashes do not occur (times)	Probability of crashes not occurring (%)
Stage A	369	61.5
Stage B	465	77.5

Table. 1. Number of times and probability that crash did not occur after 600 trials.

In other words, there is a fairly high probability that a crash will not be detected in a single simulation. Therefore, as mentioned above, we decide to reduce the probability of a false negative by running n trials. I want to introduce the method I used to calculate the confidence interval. “p” is the probability of a false occurring in one trial. “n” is the total number of trials. “x” is the number of false negatives occurring in n trials. In that case, p is calculated using x and n as follows.

$$p = \frac{x}{n}$$

Based on the properties of the binomial distribution, the standard error "SE" can be calculated using p and n as follows.

$$SE = \sqrt{\frac{p(1-p)}{n}}$$

The 95 percent confidence interval "CI" for the occurrence of a false negative is calculated as follows.

$$CI = [p - z \times SE, p + z \times SE]$$

"z" is a value in the standard normal distribution, and for the 95% confidence interval, it is z = 1.96. In this case, it means that the probability of a false negative occurring falls within this interval with a 95% probability. Furthermore, this confidence interval is used to calculate the range of probability when multiple trials are performed. This can be calculated by exponentiating the lower and upper limits of the confidence interval. If we consider t trials,

$$[(p - z \times SE)^t, (p + z \times SE)^t]$$

Using this formula, we actually calculated 95 percent confidence intervals for the probability of false negatives occurring after n trials for stage A and stage B, and summarized the results in Table 1.

	n = 5 (%)	n = 10 (%)	n = 15 (%)	n = 20 (%)
Stage A	[6.3 , 11.0]	[0.4, 1.4]	[0.025, 0.170]	[0.001, 0.020]
Stage B	[22, 34]	[5, 11]	[1.1, 4.1]	[0.25, 1.40]

Table. 2. Probability of false negatives after 5, 10, 15, and 20 trials.

The train speeds are initialized randomly, it is highly likely that a crash will not be detected in a single attempt. However, if we run a sufficient number of attempts for many combinations of train speeds, the chances of detecting a crash increase.

3.2. Deadlock

Let's actually look at deadlock detection step by step. In a deadlock detection, the victory condition is met when all trains reach the end of the track.

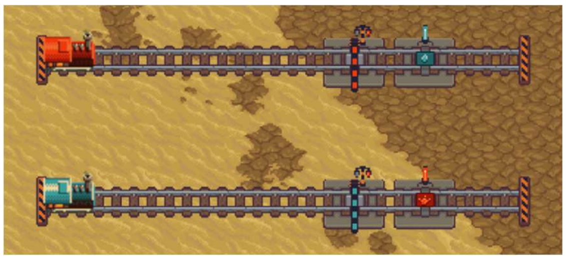

Fig. 8. Initial state.

As you can see, there is a deadlock in the stage of figure 8. Both trains are blocked by the railroad crossing and cannot meet the victory condition. The deadlock detection logic is as follows. First, the red train moves three tiles and stops at the red railroad crossing. This is due to the nature of depth-first search, which prioritizes moving the red train as much as possible. In my implementation, the board is saved every time the train moves one tile, so it can be rolled back at any time.

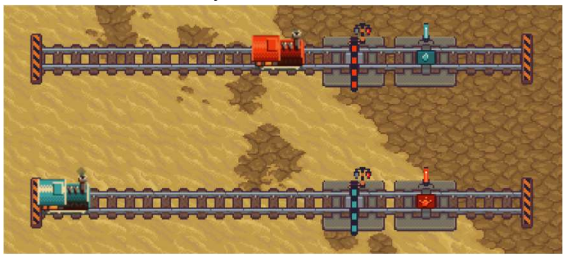

Fig. 9. The red train moved 3 tiles.

Currently, the red train is stopped as shown in figure 9, and the depth-first search algorithm next selects the blue train. Similarly, the blue train also moves three tiles.

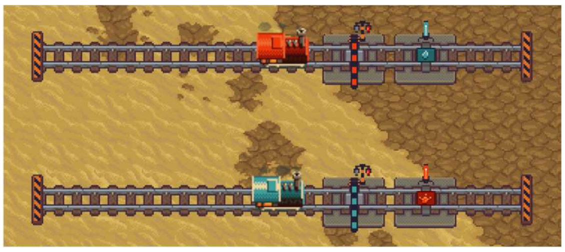
Fig. 10. The blue train moved 3 tiles.

Now, all trains are stopped at the railroad crossing, but they have not yet reached the goal at the end of the track. Since none of the trains can be moved, the algorithm decides to roll back one step. Since the blue train was the most recently moved, we rewind the blue train one tile.

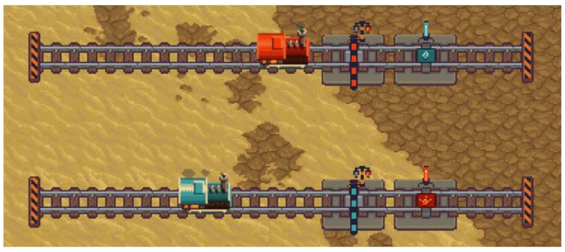
Fig. 11. A rollback occurred and the blue train reeled back.

The goal condition is checked when a rollback occurs. A rollback is required if all trains have stopped. If the goal condition is not met in this situation, a deadlock should be detected. The goal condition is for all trains to reach the end of the track, which is not achieved this time. Therefore, the deadlock is correctly detected. When a deadlock is detected, the user is notified and the path in which it occurred is reproduced graphically. The user can confirm that a deadlock will occur without any effort. For example, this example reproduces six steps: red, red, red, blue, blue, blue. If a deadlock is found on a path, it can be determined that the stage contains a deadlock, and there is no need to search for additional paths. Therefore, the simulation ends immediately after the rollback in this stage.

Let's verify this result with a theoretical graph. At the stage of Figure 8, there are red trains and blue trains, which can be thought of as representing different processes. Here, we'll represent the process held by the red train as process R, and the process held by the blue train as process B. We can get some information from the initial state, Figure 8. First, all the crossings are locked to begin with. Also, there is a blue switch on the red train's tracks that opens the blue crossings, so process R has locked the blue crossing resource and expects to release it at some point. Similarly, process B has locked the red crossings. Let's now break down and organize the tasks performed by Process R and Process B. The tasks of Process R are executed in the following order.

1. Request to pass through the red crossing, then lock it.
2. Step on the blue switch to open the blue crossing.

The tasks of Process B are similarly as follows:

1. Request to pass through the blue crossing, then lock it.
2. Step on the red switch to open the red crossing.

However, the red and blue crossings are already locked from the start, and neither process R nor process B can execute their first tasks. This is a deadlock, and can be represented, for example, by the following graph. The round nodes represent processes, and the square nodes represent resources. The arrows leaving the round nodes represent resource requests, and the arrows entering the round nodes represent resource possession.

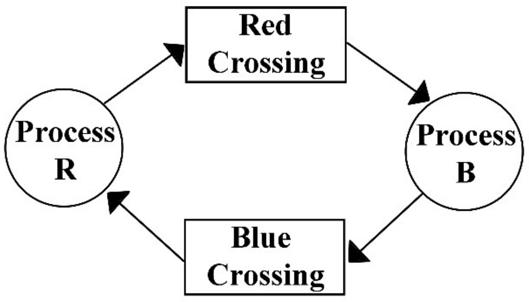

Fig. 12. Graph showing the relationship between processes and resources and the deadlock state.

Process R and process B are both waiting to request a locked resource. The graph is circular, which means that deadlocks can occur.

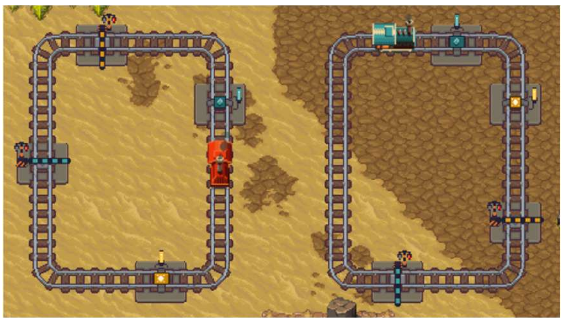
Fig. 13. The stage with loop structures.

There is also the stage with even more complex loop structures in figure 13.
In this stage, a deadlock will occur if the situation in figure 14 occurs, for example.

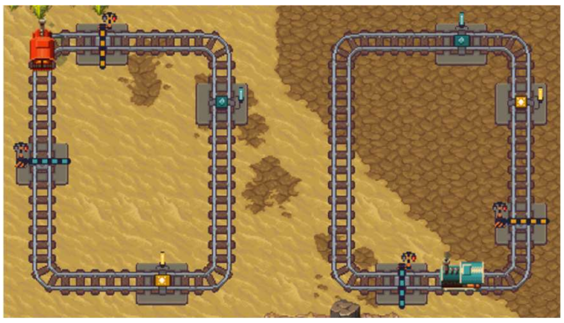
Fig. 14. The two trains are blocked at the crossings and deadlocked.

I think Figure 13 is closer to the real problem. Let's verify this with a theoretical graph as well. For example, if Figure 13 is the initial state, Process R executes the following tasks in order.

1. Release the yellow crossing
2. Lock the blue crossing
3. Lock the yellow crossing
4. Release the blue crossing

Process B similarly executes the following tasks in order:

1. Release the blue crossing
2. Release the yellow crossing
3. Lock the yellow crossing
4. Lock the blue crossing

Because Process R and Process B have a loop structure, these tasks are executed repeatedly. Because these processes include repetition, there are virtually an infinite number of combinations of steps that can occur before a deadlock occurs, but here we will only consider the step that will cause a deadlock in the shortest time. This can be reproduced by the following steps.

1. Process R executes task 1
2. Process B executes tasks 1 to 3
3. Process R executes task 2

What is happening in this situation? The next task of process R (task 3) requires passing through the yellow crossing, which is locked by process B. The next task of process B (task 4) requires passing through the blue crossing, which is locked by process R. This is a genuine deadlock, and can be represented by the following graph. B stands for the blue crossing, Y for the yellow crossing, and 0 and 1 stand for the locked and unlocked states, respectively. For example, B = 0 means that the blue train is locked. Also, I use expressions such as R2 and B3 to represent task numbers. For example, R2 is the second task of process R, and B3 is the third task of process B. Please forgive me for not showing all the diagrams due to space limitations, but only showing an example of the shortest process that leads to a deadlock.

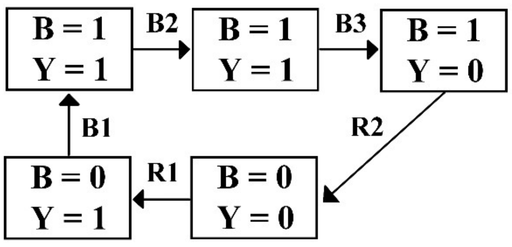

Fig. 15. Graph showing how the crossing state changes each time a task is executed.

In Figure 15, starting from state B = 0, Y = 0 (i.e. both crossings are locked), by executing tasks in a specific order, we can return to state B = 0, Y = 0. But in that situation, the next tasks that process R and process B should execute are R3 and B4, respectively, which cannot be executed because they require crossing resources. This graph is circular, which means that a deadlock can occur. However, it does not mean that it will definitely occur. For example, even in the stage of Figure 13, a deadlock will not occur depending on the execution order of certain processes. In this way, we can see that the results of the detection algorithm are consistent with the theoretical answer.

4. Conclusion

In this research, we added the ability to detect crashes and deadlocks to an educational game for parallel programming. Crashes can be detected statistically, and deadlocks can be detected reliably. For deadlock detection, we also implemented an auxiliary function that can remember the path on which a deadlock occurred and reproduce it graphically. One of the challenges with deadlock detection is that the number of game elements supported is small. We would like to continue improving the model so that we can solve these problems in the future.

References

[1] M. Purgina, M. Mozgovoy, "Designing Interactive Visualizations for Teaching Concurrent Programming", Proceedings of the 14th International Congress on Advanced Applied Informatics, Koriyama, Japan, 2023.
[2] Zhu, Jichen, et al. "Programming in game space: how to represent parallel programming concepts in an educational game." Proceedings of the 14th International Conference on the Foundations of Digital Games. 2019.
[3] Dijkstra, E. W. (2002). Cooperating sequential processes. In The origin of concurrent programming: from semaphores to remote procedure calls (pp. 65-138). New York, NY: Springer New York.
[4] Merz, Stephan. "Model checking: A tutorial overview." Summer School on Modeling and Verification of Parallel Processes (2000): 3-38.