Train Journey Manager

Sara Francavilla
Registration Number: 235951
email: sara.francavilla@studenti.unitn.it

July 30, 2024

1 Instructions for compiling and running

In order to correctly compile the program please access the directory in which you have saved the program files (main.c, journey.c, controller.c, lock.c). Once you have accessed it use the g++ compiler linking the pthread library, in particular write:

```
g++ main.c journey.c controller.c lock.c -o train_simulation -lpthread
```

To run the program you can now digit:

```
./train_simulation chosen_map
```

where the chosen map can either be MAP1 or MAP2.

2 Target System: Software and Hardware Features

- **Software**
  - **Linux Environment:** Windows Subsystem for Linux (WSL), tested with Ubuntu 22.04. The program should work on any standard Linux distribution with the necessary tools installed.
  - **Compiler:** GCC (GNU Compiler Collection) installed via WSL.
  - **Required Libraries:** pthread library for multi-threading.
- **Hardware**
  - **CPU Architecture:** x86-64 (common architecture for PCs and laptops).
  - **Memory Requirements:** The program does not require much memory. It has been successfully tested on systems with as little as 2 GB of RAM. WSL typically allocates sufficient memory dynamically based on your system's resources.
  - **Disk Space:** Minimal disk space is required, likely under 100 MB, including installation and temporary files.

Since the development was done using WSL, users should ensure their WSL environment is properly configured with the necessary development tools. The default WSL installation is generally sufficient. While tested on WSL, the program is expected to run on native Linux environments as well, provided that the required software dependencies are installed.

3 Design and Implementation

The main challenge addressed in this project is ensuring that each track segment is occupied by only one train at a time, thus preventing collisions and deadlocks.

To achieve this, shared memory is used to represent the state of each track segment, with semaphores managing access and ensuring mutual exclusion. Each train requests permission to enter a track segment, updating the shared memory state to reflect its occupancy and preventing access by other trains. The journey process maintains predefined routes (MAP1 and MAP2) and communicates these itineraries to the trains via pipes. Once a train reaches its destination station, it terminates, signaling the completion of its journey.

Let's analyse everything one step at a time.

3.1 Inter-Process Communication with Pipes

As previously anticipated, the trains are communicating with the journey process using different pipes. In particular they all send requests through the same pipe, but each one of them reads the journey response from a specific one. Because the processes do not have a child-parent relationship, named pipes were used.

The initial ideas were multiple, from using signals for the trains-journey requests and pipes for the journey-trains responses, to the usage of only two pipes, one for requests and one for resposes. But if on one hand this choices would have reduced the context switching, on the other they would have needed a more complex code solution.

As for the signal choice the journey would have had to keep track of different signals coming from different processes, just to then create pipes or messages to answer them back. A lot of work to do just to then have to answer back using pipes. Using only one pipe to which all trains have access makes the reading part much easier and faster.

As for the two pipes choice, the problem comes when the trains have to read from the response pipe. In fact, when doing that they delete information that are needed by the rest of the trains. In order to avoid this negative effects, an easy and fast solution is to implement the journey-trains communication with specific pipes.

It now comes natural that the option of combining signals and an unique pipe was discarded for the same reasons listed above, whereas using only five pipes for both reading and writing would have needed the journey process to continuously check multiple things at a time, making the program doing much more parallel work.

3.1.1 Code analysis

The communication between the train processes and the journey process is done using the function:

```
char* askJourney(int i);
```

The function leverages two distinct FIFOs for communication: a single request FIFO and multiple response FIFOs. The request FIFO is used for sending requests from trains to the journey process, while each train has its own dedicated response FIFO. This separation ensures that responses are routed directly to the requesting train without interference.

```
 ...
if ((request_fd = open(REQUEST_FIFO, O_WRONLY)) < 0) {
     perror("open request FIFO");
    exit(EXIT_FAILURE);
}

char fifo_path[BUFFER_SIZE];
snprintf(fifo_path, sizeof(fifo_path), "%s%d", RESPONSE_FIFO, i + 1);

if ((response_fd = open(fifo_path, O_RDONLY | O_NONBLOCK)) < 0) {
    perror("open response FIFO");
    exit(EXIT_FAILURE);
}
...
```

The response FIFO path is dynamically constructed based on the train's identifier. This approach ensures that each train has a unique communication channel for receiving its itinerary, avoiding conflicts between multiple trains. The naming convention (RESPONSE FIFO$< i >$) facilitates organized and scalable response handling.

The response FIFO is opened in non-blocking mode, which allows the train process to attempt reading without blocking. This design ensures that the train can periodically check for responses and handle cases where no data is immediately available, thus improving responsiveness.

The train process writes its request, which includes its identifier (i + 1), to the request FIFO and starts reading from the response FIFO in a loop, checking for responses corresponding to the train's identifier.

```
snprintf(write_buffer, sizeof(write_buffer), "%d\n", i + 1);
while (write(request_fd, write_buffer, strlen(write_buffer)) < 0) {
    if (errno != EAGAIN) {
        perror("write request FIFO");
        break;
    }
    usleep(100000);
}

while (1) {
    ssize_t bytes_read = read(response_fd, read_buffer, sizeof(read_buffer) - 1);
    if (bytes_read > 0) {
        read_buffer[bytes_read] = '\0';
        if ((read_buffer[1] - '0') == i + 1) {
            break;
        }
    } else if (bytes_read < 0) {
        if (errno != EAGAIN) {
            perror("read response FIFO");
            break;
        }
        usleep(100000);
    }
}
```

Using usleep mitigates the busy-waiting problem. The function includes robust error handling for file operations and dynamic memory allocation, enhancing the reliability of the communication mechanism.

On the other side of the communication, the journey process reads all incoming requests from the common request FIFO and handles them one by one. This sequential processing prevents race conditions and ensures that each request is addressed in the order it is received. The journey process sends the requested path information to the appropriate train via its dedicated response FIFO, ensuring accurate delivery of itineraries.

```
if (bytes_read > 0) {
  read_buffer[bytes_read] = '\0';
  char *line = strtok(read_buffer, "\n");

  if (strcmp(line, "STOP") == 0) {
      break;
  }

  while (line != NULL) {
    int train_num = atoi(line);
    snprintf(write_buffer, sizeof(write_buffer), "%s\n", journeys[train_num - 1]);
    while (write(response_fd[train_num - 1], write_buffer, strlen(write_buffer)) < 0){
        if (errno != EAGAIN){
            perror("write response FIFO");
            break;
        }
        usleep(100000);
    }
    line = strtok(NULL, "\n");
  }
}
```

3.2 Shared Memory implementation

If the shared memory is initialized in the controller through the functions *createSharedMemory* and *cleanupSharedMemory*, its management is mostly done by the function *modifySharedMemory* using the lock.c file.

In the first two functions a shared memory segment is created, initialized, retrieved, detached and removed. The most important role is taken by the last function, which works with semaphores and current and next position of the train.

As previously anticipated, the *modifySharedMemory* function works together with the lock file. This is necessary to ensure data consistency and prevent race conditions when accessing or modifying shared memory in a concurrent environment as this one. In fact when multiple processes or threads access and modify shared memory concurrently, race conditions can occur. Locks help ensure that operations on shared memory are atomic, without them concurrent modifications could lead to data corruption or deadlocks, where processes are stuck waiting for each other to release resources.

In this function locks are used around the shared memory operations, to ensure that the modifications are performed safely and consistently.

```
bool modifySharedMemory(int current_index, int next_index) {

lock();

// Access shared memory and modify the specified segment
bool modified = false;

// If next_index is occupied return false
if (next_index > 0 && shared_memory[next_index - 1] == '1') {
    modified = false;
}
// If next_index is 0, and current_index is non-zero
else if (next_index == 0 && current_index > 0) {
    ...
}
// If current_index is 0, and next_index is non-zero
else if (current_index == 0 && next_index > 0) {
    if (shared_memory[next_index - 1] == '0') {
        ...
    }
}
// If both current_index and next_index are non-zero
else if (current_index > 0 && next_index > 0) {
    if (shared_memory[next_index - 1] == '0') {
        ...
    }
}else if (current_index == 0 && next_index == 0) {
    modified = true;
}

unlock();

return modified;
```

The current and next indexes computed by the *extractindex* function are here the conditions that determine in which way we want to modify memory. In fact, based on their value different action will be taken, just as commented in the code above. Notice that if one of the indexes is 0 it means that we are working with a station.

In the case in which we have to move to the next segment a new check is done. This check verifies that the segment is free before taking any action.

4 Execution

When the program is executed the train processes will create 5 different log files (one per train) in which they will record information about the mission using the function *logTrainActivity*.

From these files it is possible to check the right functioning of the program. In fact in both the journey maps there are some cases in which the trains will ask to go to the same segment. If the program works correctly the trains will access the segment at different times (visible on the files), mainly at 2 seconds of difference one from another given the fact that each train sleeps for 2 seconds after having completed a loop.

Let's assume that we have run the program using MAP2. T4 and T5 will generate these files:

```
[Current: S6], [Next: MA8], 2024-07-29 21:44:14
[Current: MA8], [Next: MA3], 2024-07-29 21:44:16
[Current: MA3], [Next: MA2], 2024-07-29 21:44:20
[Current: MA2], [Next: MA1], 2024-07-29 21:44:22
[Current: MA1], [Next: S1], 2024-07-29 21:44:24
[Current: S1], [Next: ], 2024-07-29 21:44:26
```

T4

```
[Current: S5], [Next: MA4], 2024-07-29 21:44:14
[Current: MA4], [Next: MA3], 2024-07-29 21:44:16
[Current: MA3], [Next: MA2], 2024-07-29 21:44:18
[Current: MA2], [Next: MA1], 2024-07-29 21:44:20
[Current: MA1], [Next: S1], 2024-07-29 21:44:22
[Current: S1], [Next: ], 2024-07-29 21:44:24
```

T5

Both the trains need to access the segment MA3 but only one of them can, in this case T4 will wait and give priority to T5. If rerunning the program T5 happens to be slower the opposite will happen.

Let's see in detail how does this happen.

4.1 Code analysis

In the train process we first create the variables needed to keep track of the train position and we the first element in position (the Tx in the MAP files).

```
char *position = strtok(journey, " ");
char previous_destination[10] = "";
char next_destination[10] = "";
bool modified = false;
position = strtok(NULL, " ");
```

We now go inside the while loop, where after having updated the train position we evaluate the position indexes using the *extracIndex* function.

```
while (position != NULL) {

    strncpy(previous_destination, position, sizeof(previous_destination) - 1);
        previous_destination[sizeof(previous_destination) - 1] = '\0';

        position = strtok(NULL, " ");

        if (position != NULL) {
            strncpy(next_destination, position, sizeof(next_destination) - 1);
            next_destination[sizeof(next_destination) - 1] = '\0';
        } else {
            next_destination[0] = '\0'; // No more destinations
        }

        int index_destination = extractIndex(next_destination);
        int index_current_position = extractIndex(previous_destination);
        ...
}
```

Once we have obtained the indexes we can upload the log file with the time in which we have reached a certain position and try to get to the next position using the *modifySharedMemory* function.

```
...
        logTrainActivity(i+1, previous_destination, next_destination);

        while (!modified){
            modified = true;
            modified = modifySharedMemory(index_current_position, index_destination);
        }
        modified = false;
        sleep(2);

    }
```

At the end of the loop, the modified variable is restored and the train waits for 2 seconds before asking to move again.

5 Conclusion

This project manages track segments in a railway simulation to prevent collisions and deadlocks. Shared memory and semaphores ensure that each track segment is occupied by only one train at a time, maintaining data integrity and preventing concurrent access issues. Inter-process communication via named pipes (FIFOs) allows for clear and organized communication between trains and the journey process, with each train receiving responses through its dedicated FIFO.

The system's design, including dynamic FIFO path construction and non-blocking reads, enhances responsiveness and reliability. Comprehensive logging verifies correct operation and concurrent access handling. Overall, the solution ensures safe and efficient track management, meeting the project's objectives and demonstrating robustness in a concurrent environment.