INX 365 Assignment 1
Airport Simulator

Lukas Elsner
Queensland University of Technology
September 3, 2014

The task was to develop an airport simulator using C and POSIX threads. The airport should have one runway, which is the critical resource, that cannot be used at the same time by multiple threads. The planes should land and take-off randomly based on given probabilities and being parked in the bay of the airport, which can hold op to ten planes. This document covers the implementation report, as well as occurred problems while implementing the simulator.

Contents

1 Statement of Completeness

All functionality was implemented as requested. Some minor additional changes were made. This report will cover the implementation details, as well as the additional changes.

1.1 Basic Functionality

The following basic functionality was implemented:

- Display the banner
- Generate planes with given probability
- Randomly take-off planes with given probability
- Landing simulation
- Take-off simulation
- Display airport state
- Display "airport is empty" on empty airport
- Display "airport is full" on full airport
- Joining threads to gracefully shut down.

In addition to that, the following extras were implemented:

- Running the simulator with one or none probability argument will result in default values of 50 for the non-given probabilities.
- Running the simulator with $-h$ parameter will print the usage help.

1.2 Process Synchronisation and Coordination

To coordinate and synchronize multiple threads, two semaphores and one mutes were used. The landing and take-off threads are acting like a producer/consumer pattern for the parking bays in the airport. Therefore two Semaphores are used to make sure, the threads cannot enter the critical section when their data structure might not allow a proper action, such as landing a plane onto a full airport and parking a plane into a full bay.

1.3 Non-Functionality

Concurrency was implemented with help of the open source pthread library. Three additional threads are spawned from the main thread:

- Landing thread
- Take-off thread
- Monitor thread

To obtain an adequate readability and make it easy to maintain the source code, many object-orientated paradigms were used for this program. Therefore, a lot of dynamic allocation of memory was used. Listing 1 shows that a stress test with valgrind resulted in no possible memory leaks, even after a long runtime of the simulator.
Also, the code is well commented and a doxygen reference manual was created.
There are no known issues regarding to unexpected behavior, such as crashes.

Listing 1: Valgrind analysis

```
1 ==3165== HEAP SUMMARY:
2 ==3165==     in use at exit: 0 bytes in 0 blocks
3 ==3165==   total heap usage: 1,674,733 allocs, 1,674,733 frees, 17,307,187 bytes allocated
4 ==3165==
5 ==3165== All heap blocks were freed -- no leaks are possible
6 ==3165==
7 ==3165== For counts of detected and suppressed errors, rerun with: -v
8 ==3165== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

2 Data structures

2.1 Description of Data Structures

The airport-sim application contains three main data structures:

2.1.1 airport

Listing 2 shows the airport data structure, which is the main data structure in airport-sim. Every airport has the following members.

- name
  The name of the airport
- bays
  The bays, in which the planes can be parked.
- runway
  A virtual runway, to prevent multiple take-off/landing operations at the same time.

- empty
  Semaphore to block take-off thread on empty bay.

- full
  Semaphore to block landing thread on full bay.

Listing 2: Airport structure

```
1 struct airport {
2     char *name; /**< Name of the airport. */
3     bay **bays;/**< Bays in which planes can be parked. Has length NUM_BAYS. */
4     pthread_mutex_t runway;/**< A virtual runway, to prevent multiple
5                             take-off/landing operations at the same time. */
6     sem_t empty;/**< Semaphore to block on empty bay. */
7     sem_t full;/**< Semaphore to block on full bay. */
8 };
```

Listing 3 shows the public available methods for the airport structure. The method `airport_init()` creates a new airport structure and `airport_destroy(airport *)` frees its memory. The methods `airport_land_plane(airport *)` and `airport_takeoff_plane(airport *)` are used for initiating a landing or take-off procedure. Finally, `airport_to_string(airport *)` generates the human readable string which represents the current state of the airport.

Listing 3: Airport methods

```
1 airport *airport_init(char *);
2 void airport_land_plane(airport *);
3 void airport_takeoff_plane(airport *);
4 char *airport_to_string(airport *);
5 void airport_destroy(airport *);
```

2.1.2 bay

Listing 4 shows the bay data structure, which is used to represent a slot for a landed plane. Every bay has the following members.

- plane
  The plane, which is parked in this landing bay, or NULL if there is none.

- parking_time
  Time, the plane was parked or unparked.

Listing 4: Bay structure

```
1 struct bay {
2     plane *plane; /**< Plane parked in bay, null if there is none. */
3     time_t parking_time; /**< Time, the plane was parked or unparked. */
4 };
```

Listing 5 shows the public available methods for the bay structure. The method `bay_init()` creates a new bay structure and `bay_destroy(bay *)` frees its memory. The methods `bay_park_plane(bay *, plane *)` and `bay_unpark_plane(bay *)` are used to park and unpark planes. Finally, `bay_get_plane(bay *)` is being used to get the currently parked plane of the bay.

Listing 5: Bay methods

```
1 bay *bay_init();
2 time_t bay_get_occupation_time(bay *b);
3 void bay_park_plane(bay *b, plane *p);
4 plane *bay_unpark_plane(bay *b);
5 plane *bay_get_plane(bay *b);
6 void bay_destroy(bay *);
```

2.1.3 plane

Listing 6 shows the plane data structure. Every plane has the following members.

- name
  The name of the plane.

Listing 6: Plane structure

```
1 struct plane {
2   char *name; /**< Name of the plane. */
3 };
```

Listing 7 shows the public available methods for the plane structure. The method `plane_init()` creates a new plane structure and `plane_destroy(plane *)` frees its memory. To get the name of a plane, `plane_get_name(plane *)` can be used.

Listing 7: Plane methods

```
1 plane *plane_init();
2 char *plane_get_name(plane *);
3 void plane_destroy(plane *);
```

2.1.4 Time handling

In some parts of the application, it is necessary to block the current thread for a particular amount of milliseconds. Therefore, the method `nanosleep(struct timespec *, struct timespec *)` was used. The `tools.c` file includes a method `msleep(long long)`, which can be used for suspending the current thread for a particular amount of milli seconds. The `msleep(long long)` method creates a `timespec` structure with the required seconds ($m \Uparrow 1000$) and nanoseconds ($1000000 * m \% 1000$) value. This structure is passed to the `nanosleep(struct timespec *, struct timespec *)`.

2.2 Interaction between Threads and Data Structures

To make sure, no race conditions are occurring between the three threads in airport-sim, some thread synchronization patterns were used. The main critical sections are in the methods `airport_land_plane(airport *)` and `airport_takeoff_plane(airport *)`. Both are write accessing the bays data structure. Therefore, a mutex was used to mutual exclude the access to it. Since this airport is used with a producer/consumer pattern, a semaphore are used to let the landing thread block when the airport is full and another semaphore lets the take-off thread block when the airport is empty.

One critical problem which happened in the special situation when the application is going to be shut down while the airport is full or empty was, that one of the threads was waiting forever for the semaphore to change its state. Therefore the application hung on `sem_wait(sem_t *)` and could not perform a clean exit. To resolve this issue, `sem_timedwait(sem_t *, timespec *)` was used with a timeout value of 5 seconds. Thus, the thread will do another loop when it cannot acquire the semaphore after five seconds to check if the application should exit.

Another race condition was found while stress-testing the application. There was a very unlikely situation, where the take-off thread freed the memory of a plan while the human readable output of the airport was created in `airport_to_string(airport *)`. This could lead to a segmentation fault and lets the application crash. To resolve this problem, the string generation is done in a critical section now. One side effect of this fix is, that the user might see a delay after requesting the current airport state and seeing the result.

3 Task 2

One possible solution to support multiple runways (which obviously implies a 2-runway solution) within one airport would be to introduce one additional semaphore which is representing the multiple runways. The value of the semaphore therefore would be the amount of currently used runways. `sem_wait(sem_t *)` would be called right before the `msleep(2000)` call, which simulates landing or take-off. `sem_post(sem_t *)` directly afterwards. The currently used mutex should be renamed and used only for mutual excluding access to the parking bay structure.

The folder `src-multi` includes a prototype for this idea. This is just a proof of concept which might have some flaws. No additional comments were made in the source code.

The following changes have been implemented:

- Renamed mutex `runway` to `baylock`, since this is a more appropriate name.

- Introduced Semaphore `runways` to simulate multiple runways.
- Introduced mutex `runwaylock` to mutual exclude access to runway semaphore. This is necessary to obtain the current value of the semaphore. (This is only needed if we want to know which runway currently is used)
- The methods `airport_takeoff_plane(airport *)` and `airport_land_plane(airport *)` were modified to more complex locking and synchronizing.
- A new member for holding the runway a plane landed on was introduced in the `bay` data structure.
- The method `airport_to_string(airport *)` was adapted to show the runway on which the parked plane was landed.
- The main function is now able to create multiple take-off and landing threads based on NUM_TAKEOFF_THREADS and NUM_LANDING_THREADS.

With this approach, it is possible to scale the number of threads, as well as the size of the airport regarding to parking slots and runways.