Systems Programming

Queensland University of Technology

Assignment 1

Airport Simulation

Author:
Roland Jäger
n9247220

Due Date:
Tue 9th Sept, 2014

Friday 29th August, 2014

1 Thoughts about the task

1.1 Strings

My greatest concern was the heavy string use in this C project – C is not meant to be used for heavy string usage.

The parallelism of the threads is compromised because of the blocking for the console (single calls to printf() are atomic (and slow), but a series can't be synchronized without an additional mutex). The synchronization is needed for the airport overview and can't be implemented via a message queue without a lot of extra work because of the nature of the C strings. Because of that every call to printf has to be wrapped in a mutex guarded zone.

1.2 Threads

> "... The airport simulation program will have four threads:
>
> 1. The main thread that spawns the following three threads;
> 2. A landing thread (producer) that simulates planes landing at the airport;
> 3. A taking-off thread (consumer) that simulates planes taking off from the airport;
> 4. A monitor thread that accepts input from the keyboard.
>
> The main thread starts the airport simulation program and displays the initial banner (see the example output below). Once the simulation begins it creates the other three threads as listed above and then waits for them to terminate ..."

There may only be 4 threads – one dedicated for keyboard monitoring, one producer, one consumer and the main thread ... doing nothing.

I would generally use the main thread as the dedicated thread for user input – that would be more natural, as the rest of the program could be designed for a later use (for example with a UI). But then again, all messages (currently printed from the different threads) would have to go to a dedicated message queue (thread number 4) which leads to the C string headache ...

1.3 Landing bays

> "... When a plane lands it parks in a randomly allocated landing bay that is not currently occupied (and not just in the next free available space) ..."

Could mean either:

1. Each time a plane lands new memory will be acquired to 'build' a new landing bay that will be destroyed on take off.

   The solution could be a simple list that will be extended every time a plane arrives and decreased on take off. But the additional specification "... (and not just in the next free available space) ..." makes no sense in that context.

   The random selection of a specific landing bay / airplane would be also quite complicated.

2. All memory is acquired on initialization and the landing bay is chosen randomly.

   The solution could be an array. With this there wouldn't be any random allocation "... randomly allocated landing bay ...", but a random selection is possible and the the additional specification "... (and not just in the next free available space) ..." would make sense.

   The random access in an array (for the take off and landing) would lead to misses when a bay is vacant or occupied, which leads to a new generation of a random number aso.

In my opinion none makes any sense!

The most simple solution (and performant) would be two vectors, the occupied landing bays would be in one vector for take off. Vacant landing bays would be placed in the other vector for landing. To prevent sorting the content of the two vectors for the printing of the airport sate, there should also be an array in which all landing bay pointers are stored – sorted by their numbers.

2 Completeness

The program full fills all requirements.

3 Data Structures

3.1 Parameter for the threads

The struct will give the three different threads the needed variables. (One struct will be shared with the monitor thread - no need to have a unique struct only for the monitor.) The two function pointers guarantee a data race free access to the boolean that causes the program to end.

```
1 typedef struct {
2   unsigned char probability;
3   pthread_mutex_t* const print_mutex;
4   bool (*running)();
5   void (*set_running)(bool);
6   airport_t* const airport;
7 } thread_para_t;
```

3.2 Airplane data

```
1 typedef struct {
2   struct timespec landing_time;
3   char* ID;
4 } airplane_t;
```

3.3 Landing bay data

```
1 typedef struct landing_bay {
2   unsigned int nr;
3   airplane_t* plane;
4 } landing_bay_t;
```

3.4 Vector for landing bays

To provide random access for the landing and take off thread I chose a simple vector. The vector is very basic, the array size will increase when a new element is added to the vector – only the integer size of the vector is adjusted when an element is removed.

```
typedef struct {
  landing_bay_t** bays;
  unsigned int size;
  unsigned int bay_size;
} landing_bay_vector_t;
```

3.4.1 Functions

1. `landing_bay_vector_t* create_bay_vector();`
2. `void destroy_bay_vector(landing_bay_vector_t* vec);`
3. `void push_back_vector(landing_bay_vector_t* vec, landing_bay_t* bay);`
4. `void remove_vector(landing_bay_vector_t* vec, unsigned int index);`
5. `laning_bay_t* at_vector(landing_bay_vector_t* vec, unsigned int index);`

3.5 Airport data

The struct contains all information for an airport (multiple airports are possible).
The 'runway_mutex' simulates the one runway that is available. The 'vector_mutex' guards the two vectors (since both of them are modified one mutex is better than two). The 'landing_bay_sem' semaphore is used by the landing thread to know if there are still spaces available. The 'landing_bay _sem' semaphore is used by the take off thread to know if there are planes to take off.
For the printing of the airport state there is the 'bays' array. For the random selection for landing there is the 'free_vector' and for the take off thread there is the 'used_vector'.

```
typedef struct {
  pthread_mutex_t runway_mutex;
  pthread_mutex_t vector_mutex;
  sem_t landing_bay_free_sem;
  sem_t landing_bay_used_sem;
  landing_bay_t** bays;
  landing_bay_vector_t* used_vector;
  landing_bay_vector_t* free_vector;
} airport_t;
```

4 Thread interaction

As required there are four threads:

1. The main thread creates the following threads and joins them when they are done.
2. The landing thread (producer) that simulates planes landing at the airport.

   The thread is basically a while loop with the condition that the boolean (accessed via function pointer - race condition prevention) is true. The thread will block additionally to the mutex bocks, if the 'landing_bay_free_sem' semaphore is 'empty'. That happens when the 'used_vector' is full.
3. The taking-off thread (consumer) that simulates planes taking off from the airport.

   As the landing thread, the thread is basically a while loop with the condition that the boolean(accessed via function pointer - race condition) is true. Also this thread will block additionally to the mutex bocks, if the 'landing_bay_used_sem' semaphore is 'empty'. That happens when the 'free_vector' is full.
4. The monitor thread that accepts input from the keyboard.

   Upon pressing p or P the thread will print out the state of the Airport.
   Upon pressing q or Q the thread will set the boolean (accessed via function pointer) to false and increases the semaphores so waiting threads are unblocked (deadlock prevention).

The data structures that are shared (the two vectors and the boolean) are guarded by mutexes, further more the printf function is guarded by a mutex – for previously stated reasons. Because the monitor thread needs access to landing bays, it block on the mutex for the vectors – because of that, there is a third mutex that simulates the runway that is used only by the landing and take off thread.

5 Remarks

The source code is documented, what each function does and for what they are intended is documented there.