CS39002 OPERATING SYSTEMS LABORATORY
SPRING 2025

LAB ASSIGNMENT: 7
DATE: 12–MARCH –2025

Multi-threaded programming with pthreads

Footanical Barden (FooBar) is visited by many visitors. Upon entry to FooBar, a visitor first visits the flower garden, the aquarium, the zoo, the museum, and the food stall. This takes half an hour to two hours. FooBar also has a lake where visitors can enjoy boating. At the end of a visit to FooBar, a visitor catches a boat, and makes a boating tour that runs for 15 minutes to one hour. Each boat can accommodate a single visitor. A visitor, after enjoying the other attractions of FooBar, goes to the jetty to catch a boat. If no boats are available, the visitor waits. On the other hand, if no visitors are ready for boating, the boats wait. After the boat ride, the visitor leaves.

Assume that FooBar has m boats. On some day, it has n visitors. You may assume that $5 \leqslant m \leqslant 10$, and $20 \leqslant n \leqslant 100$. All the m boats are available from the beginning of the day. Also, all the visitors enter at the beginning (no need to simulate random delays between the arrival of two consecutive visitors). Upon entering, each visitor decides a random visit time *vtime* (in the *other* facilities of FooBar) and a random ride time *rtime* on the boat, both integers in the ranges mentioned above. The visitor first waits for *vtime* minutes (*usleep* with one minute scaled down to 100 ms). It then goes to catch a boat. As soon as a boat is available for the visitor, both that boat and that visitor wait for *rtime* minutes (use a proportional *usleep*). Finally, the visitor leaves, and the boat is again ready for the ride of a next visitor. Assume that m and n are known beforehand to all the parties involved. You need to synchronize the boat rides using pthread primitives.

Implement your own counting semaphores

The original pthread standard does not supply the facility of using counting semaphores. However, condition variables support conditional waits in queues. Implement a counting semaphore using a condition variable (and an associated mutex). Define a semaphore data structure as follows.

```
typedef struct {
    int value;
    pthread_mutex_t mtx;
    pthread_cond_t cv;
} semaphore;
```

When you declare a `semaphore` structure, initialize it by:

```
{ init_value, PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER }
```

No other special initialization is needed for this assignment.

Write two functions `P(&s)` and `V(&s)`, where `s` is an (initialized) `semaphore` structure. Use the algorithms given in Section 6.2.2 of the Dragon Book, that are based on using a queue where callers of `P()` wait when the semaphore value is negative. You must not assume that the condition queue is implemented as a FIFO queue. Although this is usually the case, your program should work even if the pthread implementation uses any other data structure for condition queues.

In this assignment, all semaphores you use must be of this type. Do not use System V or POSIX semaphores. Whenever needed, you may use pthread mutexes and barriers.

The threads

The main thread reads *m* and *n* from command-line arguments. It then initializes two semaphores `boat` and `rider`, each to 0, and a standalone mutex `bmtx` to 1. Global variables and arrays are used for sharing data. A few barriers are also used for synchronization. The main thread initializes a barrier `EOS` (end of session) to two.

After this bookkeeping task, the main thread creates *m* boat threads and *n* visitor (or rider) threads, and then wait until the last rider thread completes its boating. This wait will be on the barrier `EOS`. The last boat to carry any visitor will be the second thread to wait on the barrier `EOS`. When both these threads reach the barrier, there is nothing left in the simulation, so the main thread deletes the synchronization and mutual-exclusion resources created at the beginning, and the whole process terminates.

The pseudocodes for each boat thread and for each visitor thread are given below.

Boat thread

```
Mark the boat to be available.
Repeat forever: {
       Send a signal to the rider semaphore.
       Wait on the boat semaphore.
       Get the next rider.
       Mark the boat as not available.
       Read the ride time of the rider from the shared memory.
       Ride with the visitor (usleep).
       Mark the boat as available.
       If there are no more visitors (all of the n visitors have completed riding),
              synchronize with the main thread using the barrier EOS.
}
```

Visitor (or rider) thread

```
Decide a random visit time vtime and a random ride time rtime.
Visit other attractions for vtime (usleep).
Send a signal to the boat semaphore.
Wait on the rider semaphore.
Get an available boat.
Ride for rtime (usleep).
Leave.
```

An issue in synchronization

The above pseudocodes achieve the desired behaviors of the boats and the riders except for one subtle issue. It is possible that several visitors are waiting, and two or more boats become available at the same time. A similar situation arises when multiple boats are waiting, and two or more visitors become ready for boating at the same time. To a waiting visitor, it does not matter which available boat it gets. Likewise, to an available boat, it does not matter which waiting visitor it gets for its next trip. Well, almost so. Each visitor decides a random boat-ride time `rtime` beforehand. It is a necessity for the boat to know which visitor it is taking for the next ride because both that boat and that visitor will simulate the ride by usleep for `rtime`. A handshaking between the two threads (the boat thread and the rider thread) is therefore needed to simulate the ride correctly. One possible way of achieving this is explained now.

When a boat thread is woken up by a signal operation on the `boat` semaphore, it has to do a set of things to catch the next rider (some visitor must be ready for a ride, otherwise who woke up the boat?). Moreover, if multiple visitors are ready to ride and multiple boats are available, distinct visitors must be selected by the available boats. An available boat *i* starts the process by setting a (shared) flag BA[*i*] to true, and its next visitor BC[*i*] to –1. It then uses a barrier BB[*i*] specific to that boat and initialized to 2 beforehand. After storing the willingness to take a rider, the boat thread waits on the barrier BB[*i*].

Let *j* be a waiting visitor woken up by a signal from a boat thread sent to the `rider` semaphore. The visitor thread makes a search in the shared arrays BA[ ] and BC[ ]. If an index *i* is found such that BA[*i*] is true and BC[*i*] is –1, then the visitor sets BC[*i*] = *j* and BT[*i*] = `rtime`. Moreover, the visitor thread *j* joins the barrier BB[*i*].

Since two threads join BB[*i*], the barrier is lifted. Boat *i* sets BA[*i*] to false, and reads `rtime` from BT[*i*], and the visitor *j* and the boat *i* engage in a ride for `rtime` (usleep).

There are quite a few issues involved here.

- Accessing (both reading and writing) the shared arrays BA[ ], BC[ ], and BT[ ] needs mutual exclusion. Use the mutex `bmtx` to achieve that.

- You need to ensure that the barrier BB[*i*] is initialized (to 2) by boat *i* strictly before any visitor plans to join that barrier. An attempt to join an uninitialized barrier results is an error (like floating-point exception, and you know that it is not great, while writing codes, to be exceptional).

- Suppose that BB[*i*] is appropriately initialized. Boat *i*, in an attempt to catch a visitor, would lock `bmtx`, set BA[*i*] and BC[*i*] appropriately, unlock `bmtx`, and join the barrier BB[*i*]. However, it may so happen that before being able to complete all these tasks, the boat thread gets preempted, or is yet to be scheduled after it is woken up by a signal sent to the `boat` semaphore, or fails to lock `bmtx`. But then, no ready visitor can see the availability of boat *i* until that boat thread is scheduled, locks `bmtx`, and finally gets the chance to write its willingness to accept a rider, in the shared memory (joining BB[*i*] is not an issue here).

  The situation is confusing to visitor *j*. It knows that some boat must be available (otherwise who woke it up?). But after making a scan through the entire BA[ ] and BC[ ] arrays, visitor *j* cannot identify an available boat. All it can do is to retry the search to identify an available boat. This leads to a busy wait extending over the period for which boat *i* is not scheduled. It is important to ensure that after each search, visitor *j* must release `bmtx`, otherwise boat *i* will never get a chance to write its availability to the shared arrays.

  This is not a great solution because it involves a busy wait. In any case, the semaphores (`boat` and `rider`) help us in restricting this busy wait to a small duration. That is, a visitor is not starting a busy wait immediately after it is ready for riding. It is making a busy wait only after it is woken up by an available boat. However, starvation is possible if the ride-worthy visitors keep on acquiring `bmtx` again and again, preventing the available boats from writing to the shared memory for an indefinite period of time. If you face this situation, introduce a small (like 1–10 ms) sleep to each visitor before two consecutive searches.

Superficially, this assignment looks similar to LA6. In LA6, a customer *j* is enjoying the service of a waiter *i*. Here too, a visitor *j* needs the service of a boat *i*. In LA6, *i* is fixed from the beginning for each *j* (if that customer is admitted to the restaurant). No search is involved there. Here, boat *i* and visitor *j* can pair up only after a search.

Sample Output

A random output for $m = 5$ boats and $n = 20$ visitors is given below. This transcript shows the events in chronological order (although you cannot see the exact delays). By introducing a shared time variable, you can achieve that (as you did in LA6). For simplicity, you do not have to repeat that exercise of maintaining time, in this assignment. Just use appropriate proportional delays.

```
Boat       2    Ready
Boat       4    Ready
Boat       5    Ready
Boat       1    Ready
Visitor    1    Starts sightseeing for  58 minutes
Boat       3    Ready
Visitor    3    Starts sightseeing for  96 minutes
Visitor    2    Starts sightseeing for  74 minutes
Visitor    4    Starts sightseeing for  36 minutes
Visitor    5    Starts sightseeing for 119 minutes
Visitor    6    Starts sightseeing for  36 minutes
Visitor    8    Starts sightseeing for 118 minutes
Visitor    7    Starts sightseeing for  67 minutes
Visitor    9    Starts sightseeing for  52 minutes
Visitor   10    Starts sightseeing for  51 minutes
Visitor   12    Starts sightseeing for  47 minutes
Visitor   11    Starts sightseeing for  72 minutes
Visitor   13    Starts sightseeing for  47 minutes
Visitor   14    Starts sightseeing for  47 minutes
Visitor   16    Starts sightseeing for  94 minutes
Visitor   15    Starts sightseeing for  77 minutes
Visitor   17    Starts sightseeing for  65 minutes
Visitor   18    Starts sightseeing for  30 minutes
Visitor   20    Starts sightseeing for 106 minutes
Visitor   19    Starts sightseeing for  69 minutes
Visitor   18    Ready to ride a boat (ride time = 46)
Visitor   18    Finds boat  1
Visitor    4    Ready to ride a boat (ride time = 35)
Visitor    4    Finds boat  2
Visitor    6    Ready to ride a boat (ride time = 48)
Visitor    6    Finds boat  3
Boat       2    Start of ride for visitor  4
Visitor   12    Ready to ride a boat (ride time = 24)
Visitor   12    Finds boat  4
Visitor   13    Ready to ride a boat (ride time = 40)
Visitor   13    Finds boat  5
Visitor   14    Ready to ride a boat (ride time = 48)
Boat       4    Start of ride for visitor 12
Boat       1    Start of ride for visitor 18
Boat       3    Start of ride for visitor  6
Boat       5    Start of ride for visitor 13
Visitor   10    Ready to ride a boat (ride time = 34)
Visitor    9    Ready to ride a boat (ride time = 44)
Visitor    1    Ready to ride a boat (ride time = 59)
Visitor   17    Ready to ride a boat (ride time = 60)
Visitor    7    Ready to ride a boat (ride time = 22)
Visitor   19    Ready to ride a boat (ride time = 53)
Boat       2    End of ride for visitor  4 (ride time = 35)
Boat       4    End of ride for visitor 12 (ride time = 24)
Visitor   12    Leaving
Visitor   14    Finds boat  2
Visitor   10    Finds boat  4
Boat       2    Start of ride for visitor 14
Boat       4    Start of ride for visitor 10
Visitor    4    Leaving
Visitor   11    Ready to ride a boat (ride time = 60)
Visitor    2    Ready to ride a boat (ride time = 42)
Visitor   15    Ready to ride a boat (ride time = 51)
Visitor   13    Leaving
Boat       5    End of ride for visitor 13 (ride time = 40)
Visitor    9    Finds boat  5
Boat       5    Start of ride for visitor  9
Visitor   18    Leaving
Boat       1    End of ride for visitor 18 (ride time = 46)
Visitor    1    Finds boat  1
Boat       1    Start of ride for visitor  1
Visitor   16    Ready to ride a boat (ride time = 60)
Visitor    6    Leaving
Boat       3    End of ride for visitor  6 (ride time = 48)
Visitor   17    Finds boat  3
Boat       3    Start of ride for visitor 17
Visitor    3    Ready to ride a boat (ride time = 34)
Boat       4    End of ride for visitor 10 (ride time = 34)
Visitor   10    Leaving
Visitor    7    Finds boat  4
Boat       4    Start of ride for visitor  7
Visitor   20    Ready to ride a boat (ride time = 32)
Visitor    8    Ready to ride a boat (ride time = 59)
Visitor    5    Ready to ride a boat (ride time = 26)
Visitor   14    Leaving
Boat       2    End of ride for visitor 14 (ride time = 48)
```

```
Visitor   19    Finds boat  2
Boat       2    Start of ride for visitor 19
Visitor    7    Leaving
Boat       4    End of ride for visitor  7 (ride time = 22)
Visitor   11    Finds boat  4
Boat       4    Start of ride for visitor 11
Boat       5    End of ride for visitor  9 (ride time = 44)
Visitor    9    Leaving
Visitor    2    Finds boat  5
Boat       5    Start of ride for visitor  2
Visitor    1    Leaving
Boat       1    End of ride for visitor  1 (ride time = 59)
Visitor   15    Finds boat  1
Boat       1    Start of ride for visitor 15
Boat       3    End of ride for visitor 17 (ride time = 60)
Visitor   17    Leaving
Visitor   16    Finds boat  3
Boat       3    Start of ride for visitor 16
Visitor   19    Leaving
Boat       2    End of ride for visitor 19 (ride time = 53)
Visitor    3    Finds boat  2
Boat       2    Start of ride for visitor  3
Boat       5    End of ride for visitor  2 (ride time = 42)
Visitor    2    Leaving
Visitor   20    Finds boat  5
Boat       5    Start of ride for visitor 20
Visitor   11    Leaving
Boat       4    End of ride for visitor 11 (ride time = 60)
Visitor    8    Finds boat  4
Boat       4    Start of ride for visitor  8
Visitor   15    Leaving
Boat       1    End of ride for visitor 15 (ride time = 51)
Visitor    5    Finds boat  1
Boat       1    Start of ride for visitor  5
Boat       5    End of ride for visitor 20 (ride time = 32)
Visitor   20    Leaving
Boat       2    End of ride for visitor  3 (ride time = 34)
Visitor    3    Leaving
Visitor   16    Leaving
Boat       3    End of ride for visitor 16 (ride time = 60)
Visitor    5    Leaving
Boat       1    End of ride for visitor  5 (ride time = 26)
Visitor    8    Leaving
Boat       4    End of ride for visitor  8 (ride time = 59)
```

Submit a single C/C++ source file boating.c(pp).