Pthreads II

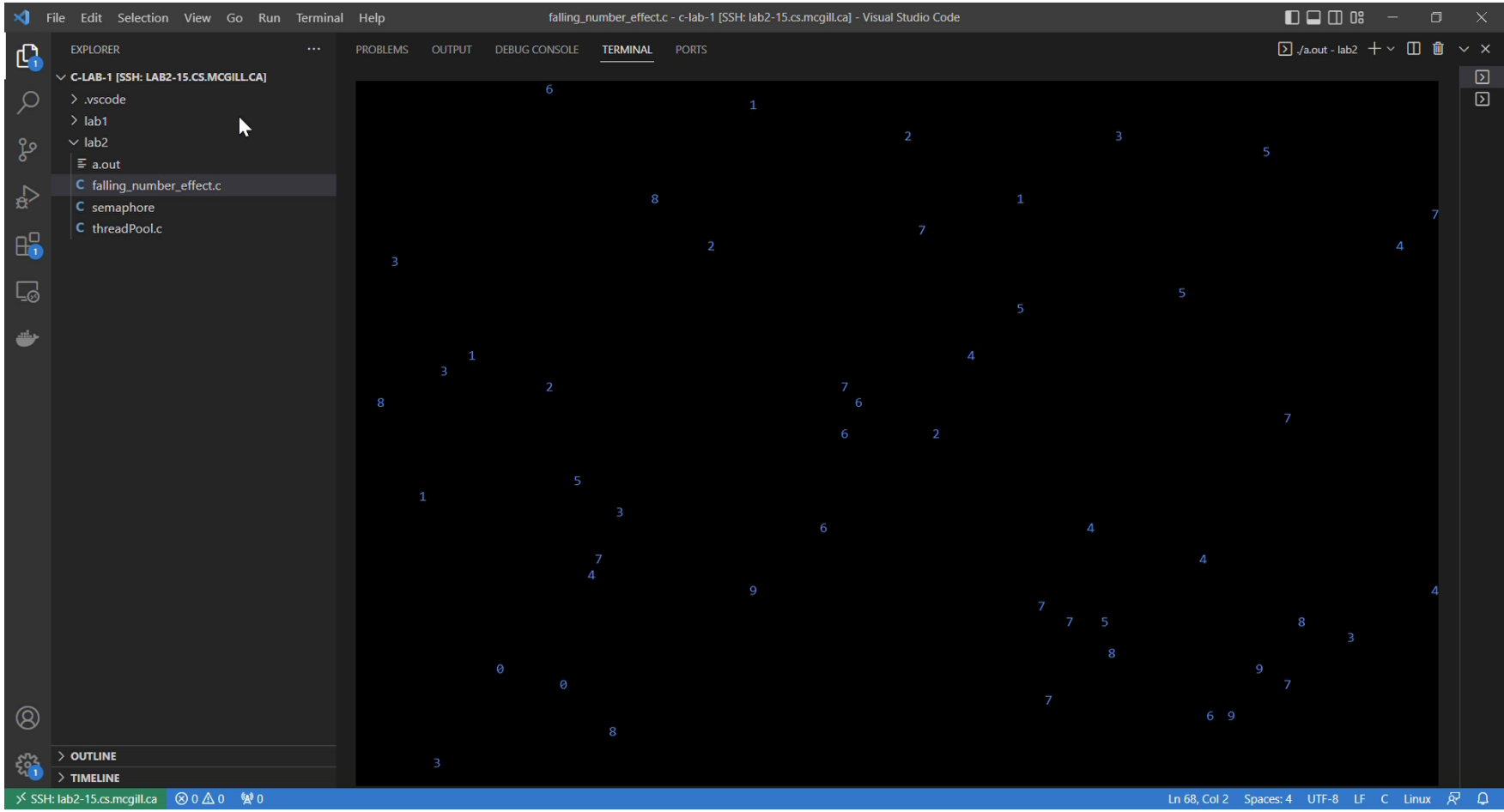
By Zixu Zhou

Revised from Aayush Kapur

Example

We have seen the falling number effect in the matrix movies or as the depiction of hacking in the movies from 90s, 2000s.

Something like in the following video ☺



In this task, we have to make multithreaded implementation of this effect.

Compile the program using - gcc falling_number_effect.c -lpthread –lncurses

We are using two additional libraries here: pthreads and ncurses.

Example

- In the following example, we will try to solve the sleeping barber problem.

The original barbershop problem was proposed by Dijkstra. A variation of it appears in Silberschatz and Galvin's *Operating Systems Concepts* [10].

> A barbershop consists of a waiting room with n chairs, and the barber room containing the barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy, but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber. Write a program to coordinate the barber and the customers.

Elaborating further

- *Each customer, when they arrive, looks to see what the barber is doing. If the barber is sleeping, the customer wakes him up and sits in the cutting room chair. If the barber is cutting hair, the customer stays in the waiting room. If there is a free chair in the waiting room, the customer sits in it and waits their turn. If there is no free chair, the customer leaves.*

- *When the barber finishes cutting a customer's hair, he dismisses the customer and goes to the waiting room to see if there are others waiting. If there are, he brings one of them back to the chair and cuts their hair. If there are none, he returns to the chair and sleeps in it.*

How to handle tasks in a multi threaded manner?

- Have a task queue from where you retrieve the tasks to be executed on threads.
- Have a result queue where you put the results of the tasks you executed.

What is interesting about thread pool?

- It prevents system from overloading.
- We can limit the number of threads being launched.
- Either they will be running or waiting.
- The pool can be scaled up based on need.

Example

- Have say 10,000 iterations to be executed in total.
- Break these iterations into tasks consisting of irregular number of iterations.
- Add these tasks to the task queue.
- Task - estimate the value of pi.
- Collect the values in result array.

Go through the exercise.

```
The value of pi for this division is 3.138828 - 65
The value of pi for this division is 3.143944 - 66
The value of pi for this division is 3.143364 - 67
The value of pi for this division is 3.140688 - 68
The value of pi for this division is 3.143892 - 69
The value of pi for this division is 3.143072 - 70
The value of pi for this division is 3.141560 - 71
The value of pi for this division is 3.142624 - 72
The value of pi for this division is 3.140296 - 73
The value of pi for this division is 3.142592 - 74
The value of pi for this division is 3.140012 - 75
The value of pi for this division is 3.142356 - 76
The value of pi for this division is 3.141696 - 77
The value of pi for this division is 3.141736 - 78
akapur12@lab2-15:~/c-lab-1/lab2$
```

Bonus example

- Implementing a multi threaded linked list.

- Operations to include: insert, member, delete

- Use locks over insert, member and delete functions.
- Fine grain locking used: one mutex per node.
- A nice implementation can be found at https://redirect.cs.umbc.edu/~tsimo1/CMSC483/cs220/code/pth-rw/pth_linked_list_mult_mut.c

Java

- **Thread-Safe Data Structures**
  - **Vector, Hashtable**: Synchronized methods to ensure thread safety.
  - **ConcurrentHashMap**: Segmented locking mechanism for higher concurrency.
  - **CopyOnWriteArrayList, CopyOnWriteArraySet**: Immutable elements on modification to avoid concurrency issues.
  - **BlockingQueue Implementations (e.g., ArrayBlockingQueue, LinkedBlockingQueue)**: Designed for producer-consumer patterns; uses internal locks to manage data safely.
- **Non-Thread-Safe Data Structures**
  - **ArrayList, HashSet, HashMap**: Fast access but no synchronization.
  - **LinkedList**: No internal synchronization; prone to data corruption if modified concurrently.
- **Synchronization Wrappers**
  - **Collections.synchronizedList, Collections.synchronizedSet, Collections.synchronizedMap**: Utility methods to wrap non-thread-safe collections in synchronized wrappers for thread safety.

Golang

- **Thread-Safe Data Structure**
  - **Channel**: Native Go data structure designed for safe communication between goroutines.
- **Non-Thread-Safe Data Structures**
  - **Slice, Map**: No inherent thread safety; prone to race conditions if not properly synchronized.
  - **Arrays**: Like slices, direct access without synchronization can lead to data inconsistencies.
- **Concurrency Management Tools**
  - **sync.Mutex and sync.RWMutex**: Locks to protect shared resources and prevent race conditions.
  - **sync.WaitGroup**: For waiting for a collection of goroutines to finish.
  - **atomic package**: Provides low-level atomic memory primitives useful for managing state without locks.