Programming Problems HW#4

Chapter 7

7.15

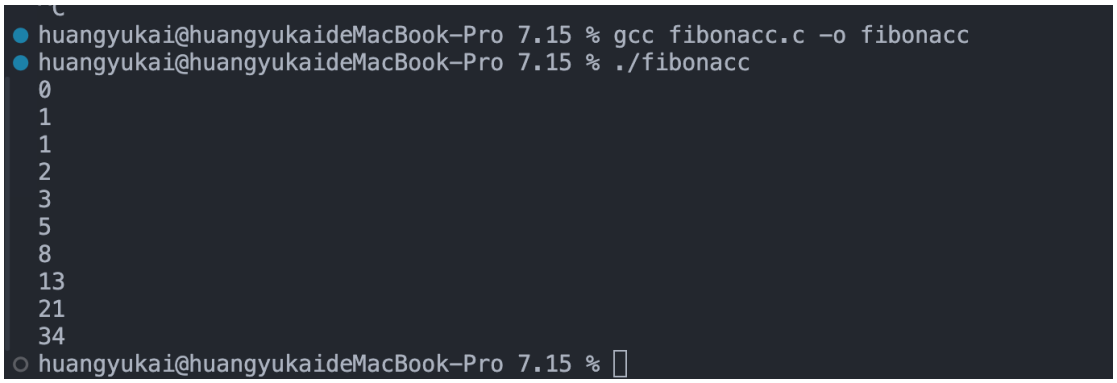
In this code, the child thread computes each Fibonacci number and then signals the parent thread that a number has been computed. The parent thread waits for this signal before printing each number. This ensures that the parent thread only prints a number after it has been computed by the child thread.

To create a compile file, command:

```
gcc fibonacc.c -o fibonacc.o -lm
```

And then, type:

```
./monte.o
```



7.17 (Optional)

In this implementation, I use four threads to generate a total of 10 million random points, if you want to change the random points just modify the (NUM_POINTS), with each thread generating 2.5 million points. The count variable is protected by a mutex to prevent data races and ensure thread safety.

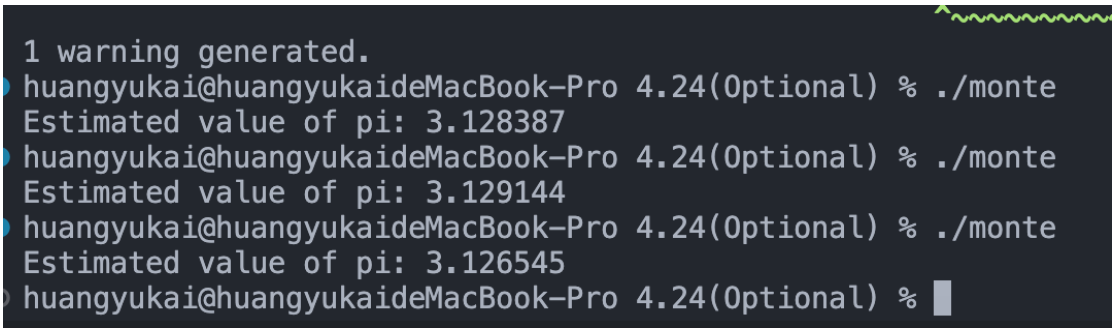
When a thread finishes, it acquires the lock on the mutex, updates the global count variable with its local count, and then releases the lock. After all threads have finished, the main thread calculates the estimated value of pi using the formula, and outputs the result to the console.

Note that the random number generator is initialized with a seed that is unique to each thread, based on the current time and the thread ID. This ensures that each thread generates a different sequence of random numbers. To create a compile file, command:

```
gcc monte.c -o monte.o -lm
```

And then, type:

```
./monte.o
```



Chapter 8

8.32

I have implemented a C++ program to help us understand a solution to this problem preventing deadlock and starvation.

To compile:

```
g++ SingleLaneBridgeProblem.cpp -lpthread
```

To run:

1. ./a.out
   - ✓ Creates random number of villagers in North and South.
   - ✓ Allows a villager to starve for count = 3, at most.
2. ./a.out s
   - ✓ Creates random number of villagers in North and South
   - ✓ Allows a villager to starve for count = s (integer), at most.
3. ./a.out m n
   - ✓ Creates m (integer) & n (integer) number of North & South villagers respectively and randomly.
   - ✓ Allows a villager to starve for count = 3, at most.
4. ./a.out m n s
   - ✓ Creates m (integer) & n (integer) number of North & South villagers respectively and randomly.
   - ✓Allows a villager to starve for count = s (integer), at most.

Mutex initialization

`pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr)`

Creates a mutex, referenced by mutex, with attributes specified by attr. If attr is NULL, the default mutex attribute (NONRECURSIVE) is used.

Returned value

If successful, `pthread_mutex_init()` returns 0, and the state of the mutex becomes initialized and unlocked. If unsuccessful, `pthread_mutex_init()` returns -1.

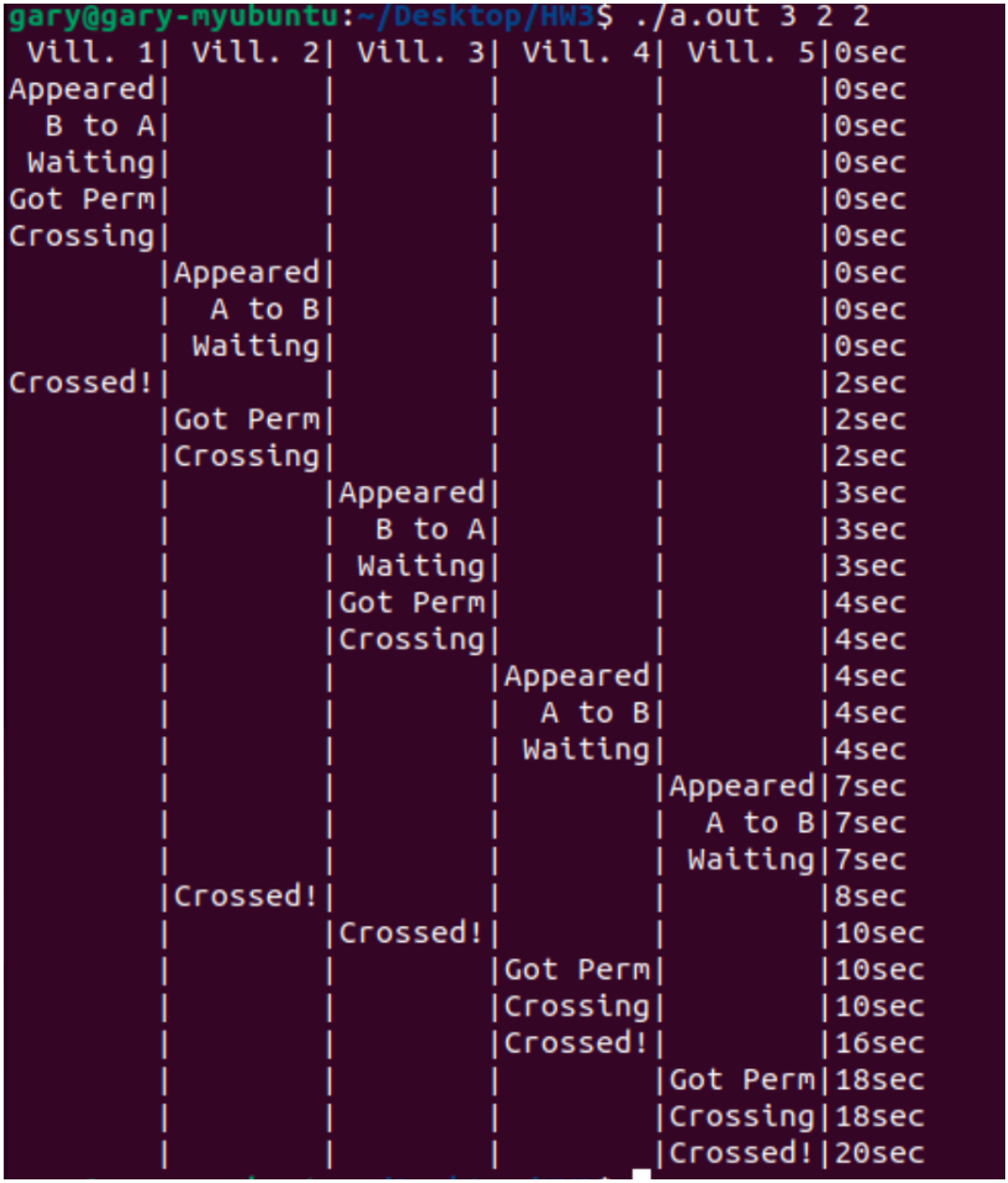
Mutex lock

`int pthread_mutex_lock(pthread_mutex_t *mutex)`

Mutex Unlock

`int pthread_mutex_unlock(pthread_mutex_t *mutex)`In figure below, given the three arguments m, n & s as 3, 2 & 2 respectively and

thus we get the output with 3 villagers going from A to B, 2 villagers going from B to A and a villager can starve for a maximum count of 2.



Chapter 9

9.32

✓ Page Number = quotient of address / 4KB

✓ offset = remainder of address / 4KB

Since page size is 4KB = 2^12 bytes, thus 12 bits holding the virtual address.

To compile:

```
gcc PageNumber.c -o PageNumber.o
```

To run:

```
./ PageNumber.o
```

```
huangyukai@huangyukaideMacBook-Pro 9.28 % gcc PageNumber.c -o PageNumber
huangyukai@huangyukaideMacBook-Pro 9.28 % ./PageNumber
Enter the address: ^C
huangyukai@huangyukaideMacBook-Pro 9.28 % ./PageNumber 19986
The address=19986
Page number = 4
offset = 3602
huangyukai@huangyukaideMacBook-Pro 9.28 % ./PageNumber 22222
The address=22222
Page number = 5
offset = 1742
huangyukai@huangyukaideMacBook-Pro 9.28 % ./PageNumber 22222
The address=22222
Page number = 5
offset = 1742
huangyukai@huangyukaideMacBook-Pro 9.28 % ./PageNumber 33333
The address=33333
Page number = 8
offset = 565
huangyukai@huangyukaideMacBook-Pro 9.28 %
```

Group Projects

Chapter 7

- Mutex

Only one thread to access a resource at once. For example, when student is accessing a file, no one else should have access the same file at the same time.

A lock of mutex (from mutual exclusion) is a synchronization mechanism for enforcing limits on access to a resource in an environment where there are many threads of execution. A lock is designed to enforce a mutual exclusion concurrency control policy.

Mutex Lock used:
To lock and unlock variable ChairsCount to increment and decrement its value.

- Semaphore

A semaphore contains access to a shared resource using a counter.

If the counter is greater than zero, than access is allowed. If it is zero, then access is denied. What the counter is counting are permits that allow access to the shared resource. Thus, to access the resource, a thread must be granted a permit from the semaphore.

- Semaphores used:

1. A semaphore to signal and wait TA's sleep.
2. An array of 3 semaphores to signal and wait chair to wait for the TA.
3. A semaphore to signal and wait for TA's next student.

Working of Semaphore In general, to use a semaphore, the thread that wants access to the shared resource tries to acquire a permit.

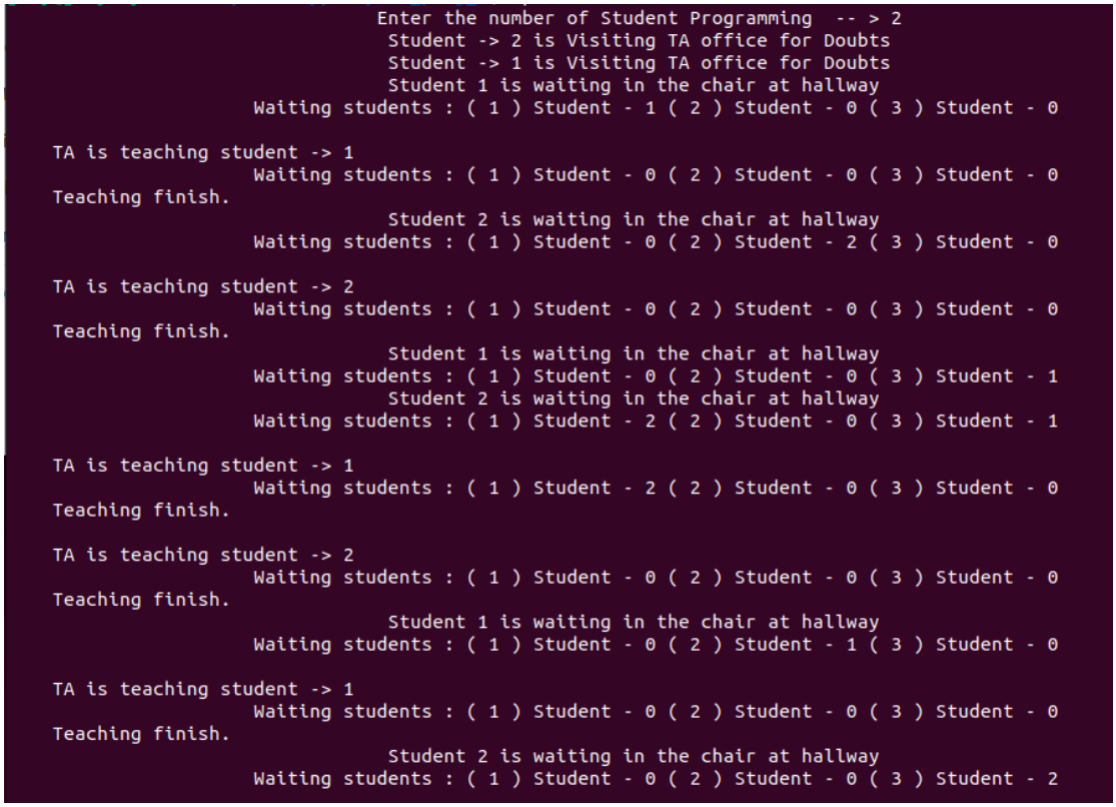
If the semaphore’s count is greater than zero, then the thread acquires a permit, which causes the semaphore’s count to be decremented.

Otherwise, the thread will be blocked until a permit can be acquired. • When the thread no longer needs an access to the shared resource, it releases the permit, which causes the semaphore’s count to be incremented.

If there is another thread waiting for a permit, then that thread will acquire a permit at that time.

- Pthread

Pthreads are a simple and effective way of creating a multi-threaded application. This introduction to pthreads shows the basic functionality – executing two tasks in parallel and merging back into a single thread in this case ‘mutex’ when the work has been done. It’s just like making two function calls at the same time.



Chapter 8

In the context of the Banker's Algorithm, NUMBER_OF_RESOURCES is likely a constant that represents the total number of different types of resources in the system. available is likely an array that represents the currently available amount of each resource.

The provided code does the following:

- It prints a string to the console. This string could be a message or a label for the data that is about to be printed.
- It enters a loop that runs once for each type of resource.
- Inside the loop, it prints the currently available amount of a resource. The resources are printed in the order they are stored in the available array.

Without more context, it's hard to provide a more detailed explanation. However, in the context of the Banker's Algorithm, this code could be used to print the current state of available resources, which could be useful for debugging or understanding the state of the system.

```
huangyukai@huangyukaideMacBook-Pro Chapter8 % ./banker_algo 10 5 7
Total system resources are:
A B C
10 5 7

Processes (maximum resources):
   A B C
P1 10 2 0
P2 1 0 5
P3 0 2 4
P4 6 5 2
P5 8 1 3
Processes (currently allocated resources):
   A B C
P1 0 0 0
P2 0 0 0
P3 0 0 0
P4 0 0 0
P5 0 0 0
Processes (possibly needed resources):
   A B C
P1 10 2 0
P2 1 0 5
P3 0 2 4
P4 6 5 2
P5 8 1 3
Available system resources are:
A B C
10 5 7

P1 requests for 7 1 0
```

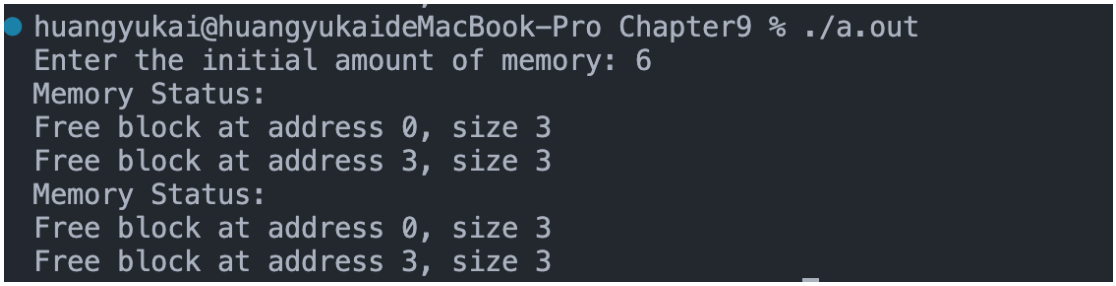
Chapter 9

It uses an array of MemoryBlock structures to represent each block of memory. Each MemoryBlock has an address, a size, and a flag is_allocated to indicate whether it's currently allocated or free.

The loop iterates over each block of memory. If it finds a block that has the given address and is currently allocated, it marks that block as free by setting is_allocated to 0.

Iterates over each block of memory. If it finds a free block, it checks if it's the start of a new free region. If it is, it sets free_start and free_end to the index of the current block. If it's not the start of a new free region, it increments free_end to include the current block in the free region.

If it finds an allocated block and there's a free region before it (free_start != -1), it compacts the free region by setting the size of the block at free_start to the size of the free region. Then it resets free_start to -1 to indicate that it's not currently tracking a free region.



Summary:

Team members

工管四乙 黃鈺凱 109370210 50%

資工三 林兆玄 110590021 50%