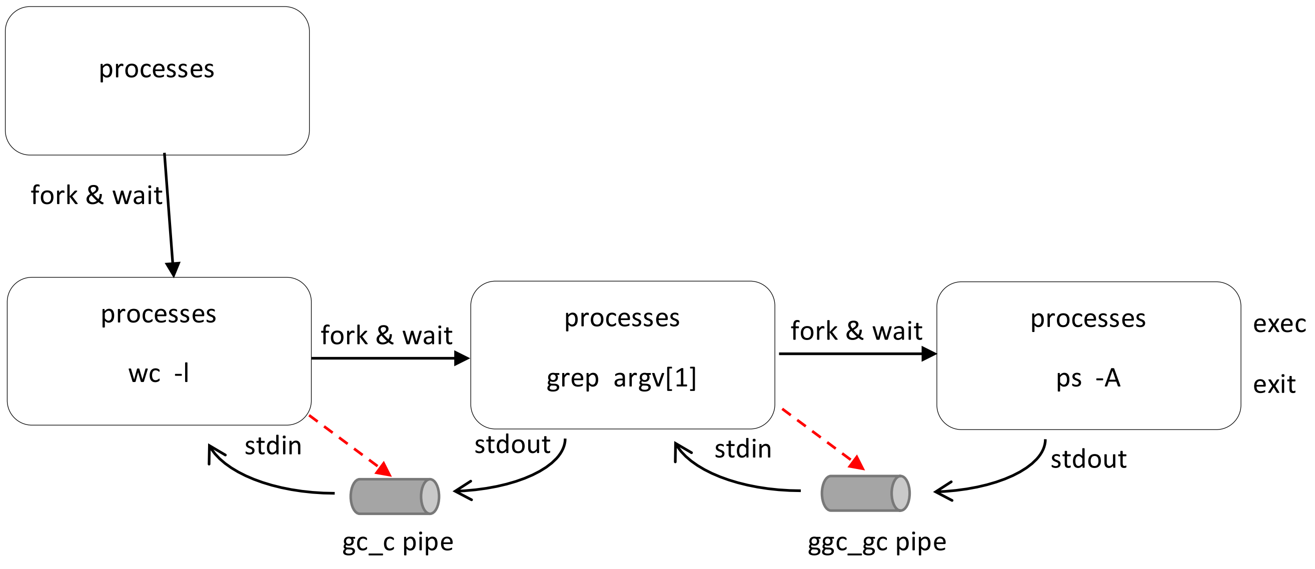
Hyeon Hong
Professor Michael Panitz
CSS 430

# Thread OS Process Forking & Piping

## Algorithm

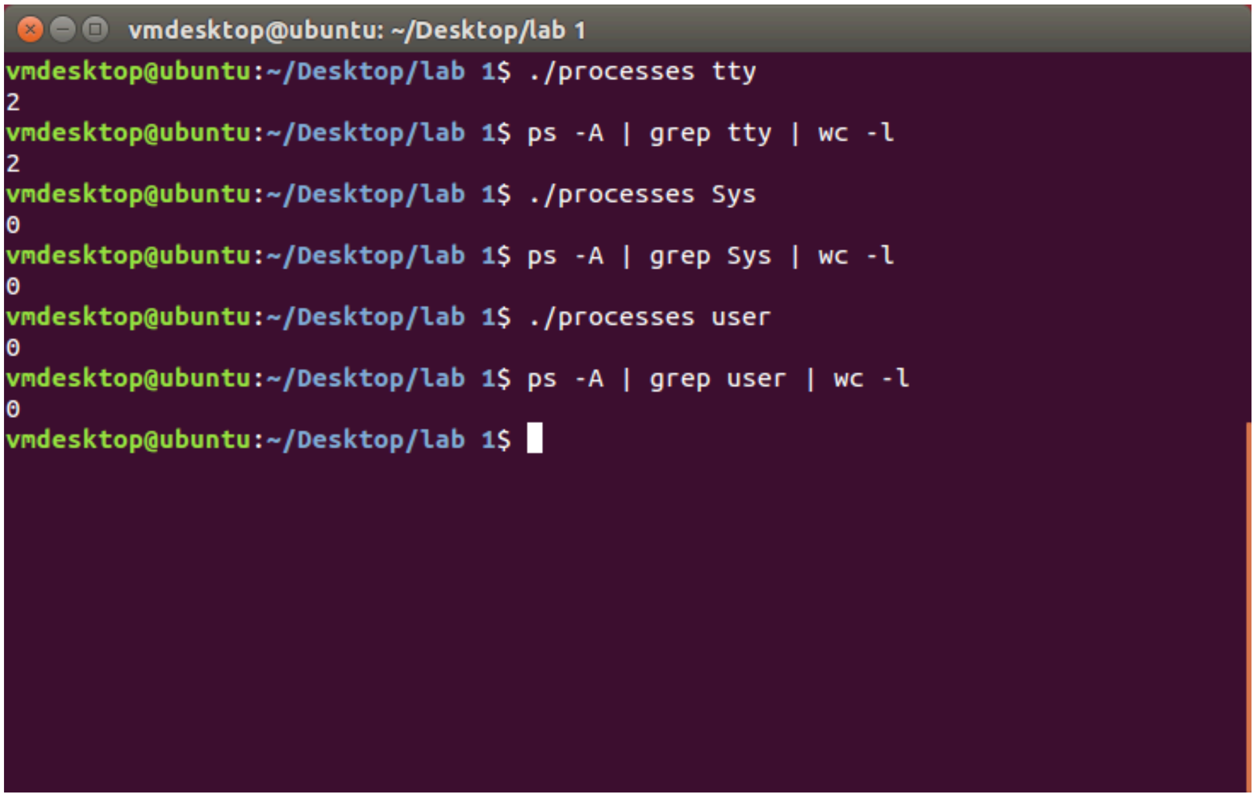
1. This program first creates two pipes:
   - pipe between great-grand-child and grand-child
   - pipe between grand-child and child
2. The parent (original) process calls the fork() and the child process is created.
3. The child **calls the pipe()** and then calls the fork(), which creates the grand-child.
4. The grand-child **calls the pipe()** and then calls the fork(), which creates the great-grand-child.
5. The great-grand-child closes unnecessary file descriptor, and connects stdout to write-end pipe. Then, it closes the used file descriptor and executes 'ps -A' command.
6. The great-grand-child is finished and terminated.
7. The grand-child closes unnecessary file descriptors and connects stdin to read-end pipe. Then, separate pipe between grand-child and child is used for connecting stdout to write-end. Afterward, it closes the used file descriptors and executes 'grep argv[1]' command.
8. The grand-child is finished and terminated.
9. The child closes unnecessary file descriptor, and connects stdin to read-end pipe. Then, it closes the used file descriptor and executes 'wc -l' command.
10. The child is finished and terminated.
11. The parent exits the program.

The overall process returns the same result as 'ps -A | grep argv[1] | wc -l' command in Unix.



Pipe is created before the forking.

## Testing



```
vmdesktop@ubuntu:~/Desktop/lab 1$ ./processes tty
2
vmdesktop@ubuntu:~/Desktop/lab 1$ ps -A | grep tty | wc -l
2
vmdesktop@ubuntu:~/Desktop/lab 1$ ./processes Sys
0
vmdesktop@ubuntu:~/Desktop/lab 1$ ps -A | grep Sys | wc -l
0
vmdesktop@ubuntu:~/Desktop/lab 1$ ./processes user
0
vmdesktop@ubuntu:~/Desktop/lab 1$ ps -A | grep user | wc -l
0
vmdesktop@ubuntu:~/Desktop/lab 1$
```

Testing confirms that the program demonstrates the same behavior as the following command.

ps -A | grep argv[1] | wc -l

# Thread OS Shell

## Algorithm

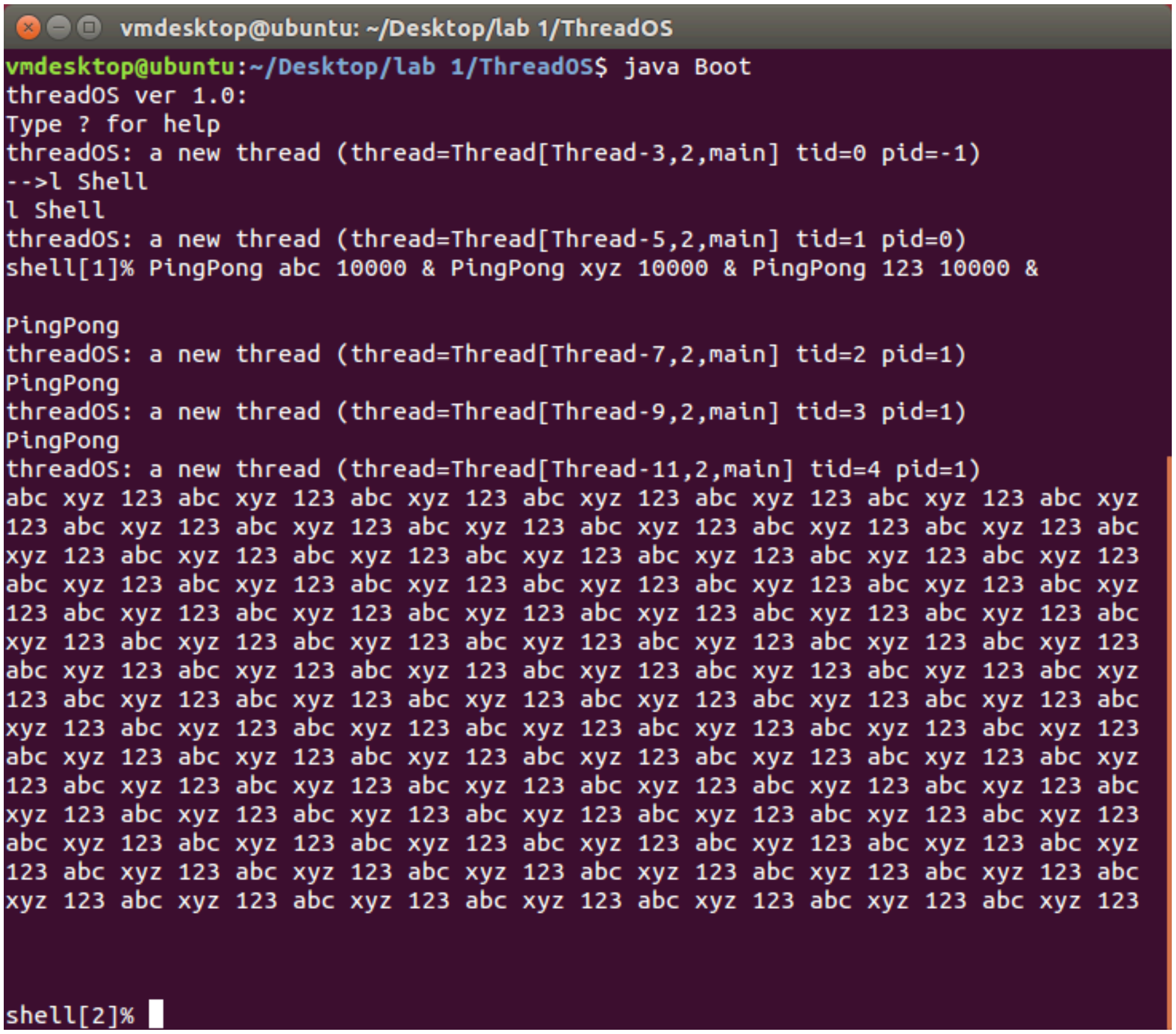
1. This program first reads from user input and stores it as a StringBuffer object.
2. The object is converted and stored as a string (cmdLine).
3. The string is split into words and stored in an array (args).
4. Finds out how many arguments one command passes in by traversing the array.
5. Once numOfArg is found, the array is traversed in set increments (numOfArg + 2).
6. Another array (args2) is created and filled with only the elements of one command.
7. exec() is executed with the array (args2) passed in as argument.
8. The delimiter part is processed.

```
        - If there’s a delimiter,
                if ‘;’,
                        first, join all of the finished concurrent threads from previous calls
                        then, execute join()
                - if ‘&’,
                        first, keep track of concurrent threads that are running
                        then, move on to next command
        - If there’s no delimiter, it will be treated the same as ‘;’
```

9. Repeat the process from step 5 to 8 until the end of command line is reached.
10. Prints out the new command line with next number (e.g., shell[2]) and waits for user input.
11. If the word “exit” is entered, the program terminates; Otherwise, it repeats from step 1.

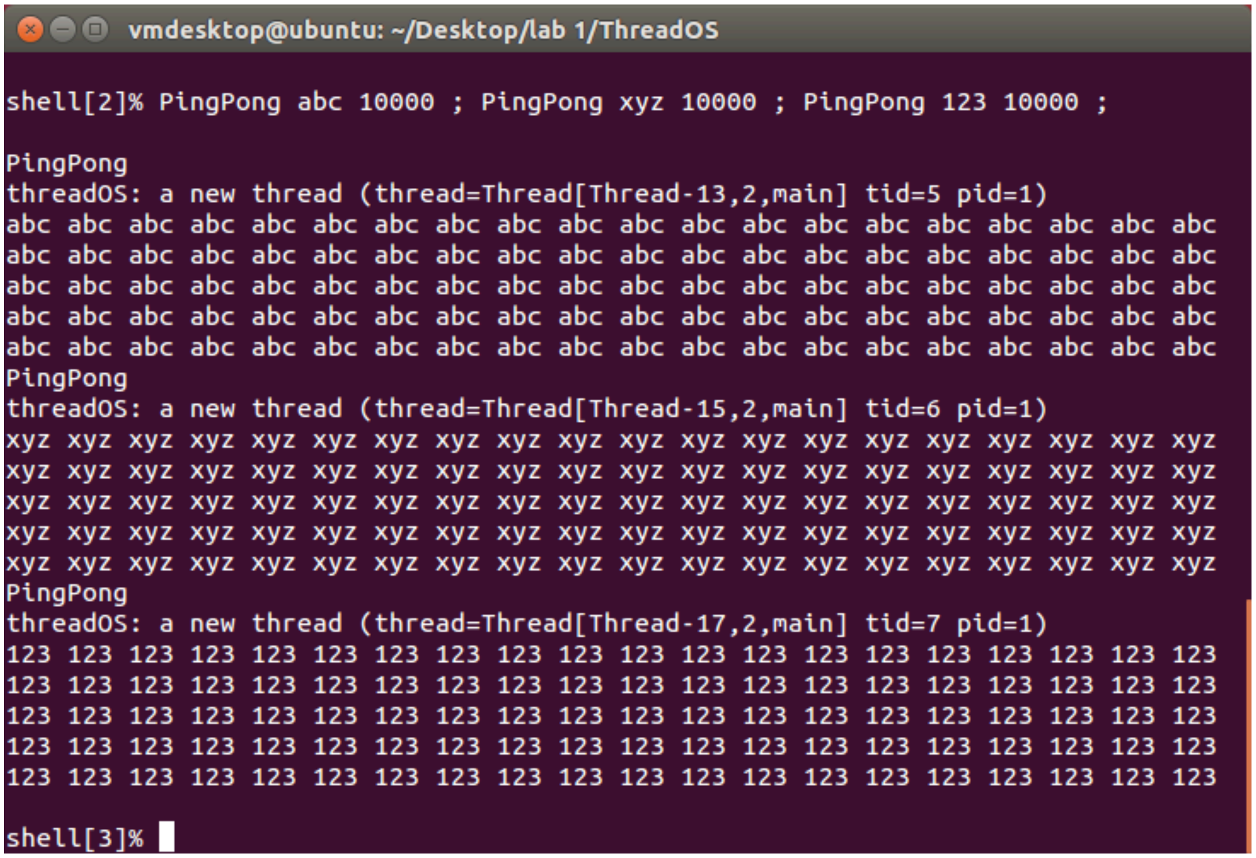
## Testing

PingPong abc 10000 & PingPong xyz 10000 & PingPong 123 10000 &



```
vmdesktop@ubuntu:~/Desktop/lab 1/ThreadOS$ java Boot
threadOS ver 1.0:
Type ? for help
threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)
-->l Shell
l Shell
threadOS: a new thread (thread=Thread[Thread-5,2,main] tid=1 pid=0)
shell[1]% PingPong abc 10000 & PingPong xyz 10000 & PingPong 123 10000 &

PingPong
threadOS: a new thread (thread=Thread[Thread-7,2,main] tid=2 pid=1)
PingPong
threadOS: a new thread (thread=Thread[Thread-9,2,main] tid=3 pid=1)
PingPong
threadOS: a new thread (thread=Thread[Thread-11,2,main] tid=4 pid=1)
abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz
123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc
xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123
abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz
123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc
xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123
abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz
123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc
xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123
abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz
123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc
xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123
abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz
123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc
xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123 abc xyz 123



shell[2]%
```

PingPong abc 10000 ; PingPong xyz 10000 ; PingPong 123 10000 ;



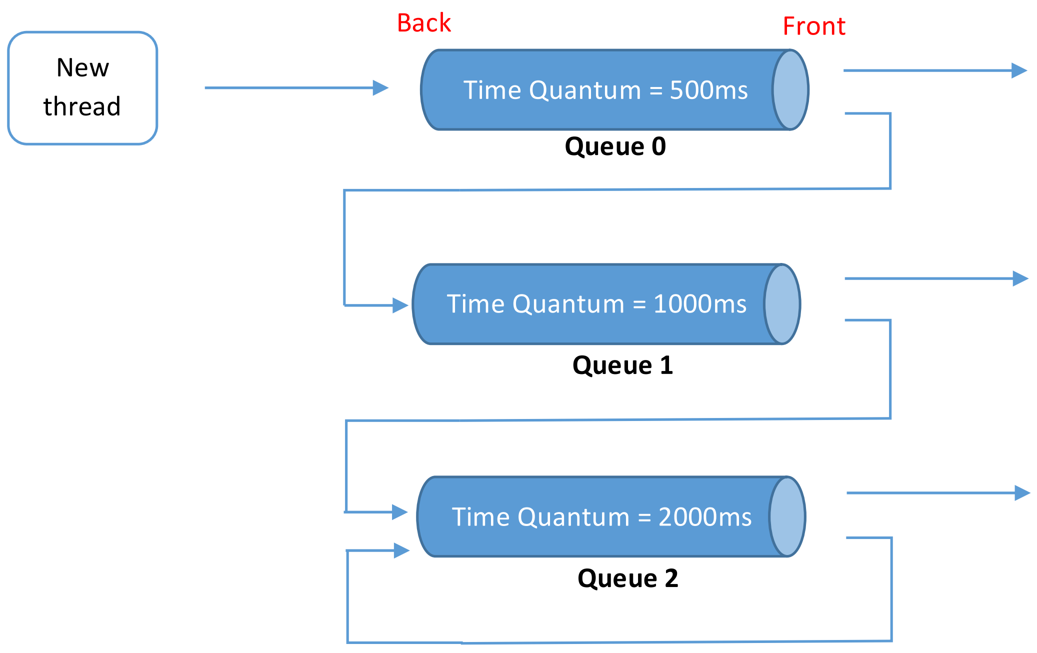
```
shell[2]% PingPong abc 10000 ; PingPong xyz 10000 ; PingPong 123 10000 ;

PingPong
threadOS: a new thread (thread=Thread[Thread-13,2,main] tid=5 pid=1)
abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc
abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc
abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc
abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc
abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc
PingPong
threadOS: a new thread (thread=Thread[Thread-15,2,main] tid=6 pid=1)
xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz
xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz
xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz
xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz
xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz xyz
PingPong
threadOS: a new thread (thread=Thread[Thread-17,2,main] tid=7 pid=1)
123 123 123 123 123 123 123 123 123 123 123 123 123 123 123 123 123 123 123 123
123 123 123 123 123 123 123 123 123 123 123 123 123 123 123 123 123 123 123 123
123 123 123 123 123 123 123 123 123 123 123 123 123 123 123 123 123 123 123 123
123 123 123 123 123 123 123 123 123 123 123 123 123 123 123 123 123 123 123 123
123 123 123 123 123 123 123 123 123 123 123 123 123 123 123 123 123 123 123 123

shell[3]%
```

# Thread OS Scheduler

## MFQS Algorithm

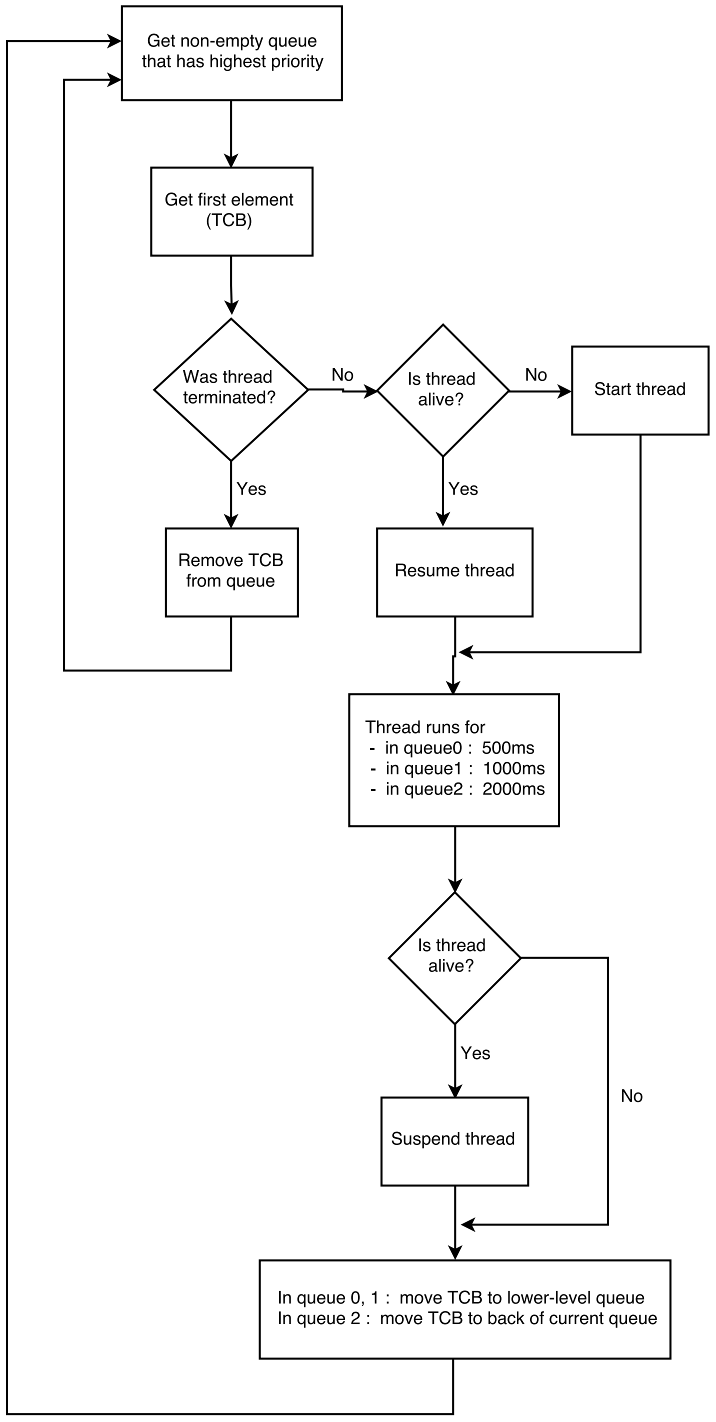
1. It has 3 queues with a time slice of 500, 1000, 2000 ms respectively.
2. Each queue in itself is scheduled via round robin.
3. New threads are added to the highest priority queue, which is queue0 (500ms = timeSlice/2).
4. If the thread in queue0 doesn't complete the execution within the time quantum, it gets suspended and moved to the lower level queue, which is queue1 (1000ms = timeSlice).
5. If queue0 is empty, the scheduler executes the first thread in queue1.
6. After one execution, the scheduler first checks queue0 to see if it's not empty or not.
7. If queue0 is not empty, then threads in it are processed first.
8. If the thread in queue1 doesn't complete the execution within the time quantum, it gets suspended and moved to the lower level queue, which is queue2 (2000ms = 2 * timeSlice).
9. If both queue0 and queue1 are empty, the scheduler executes the first thread in queue2.
10. After one execution, the scheduler first checks queue0 and queue1 to see if they're empty or not.
11. If they are not, then queue0 is processed first, and then queue1 is processed.
12. If the thread in queue2 doesn't complete the execution within the time quantum, it gets suspended and moved to the tail of queue2.

The overall process is illustrated in the following diagram.



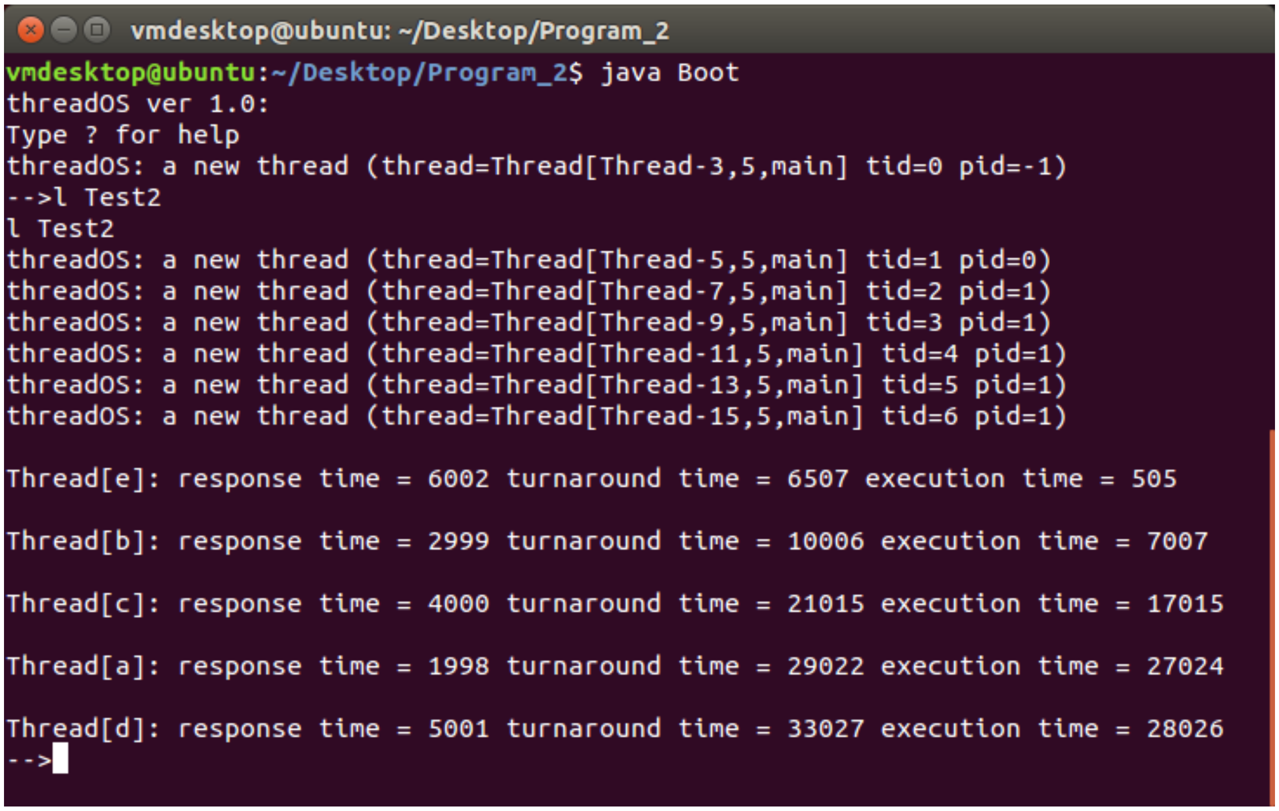
## Algorithm Flowchart

### Flowchart



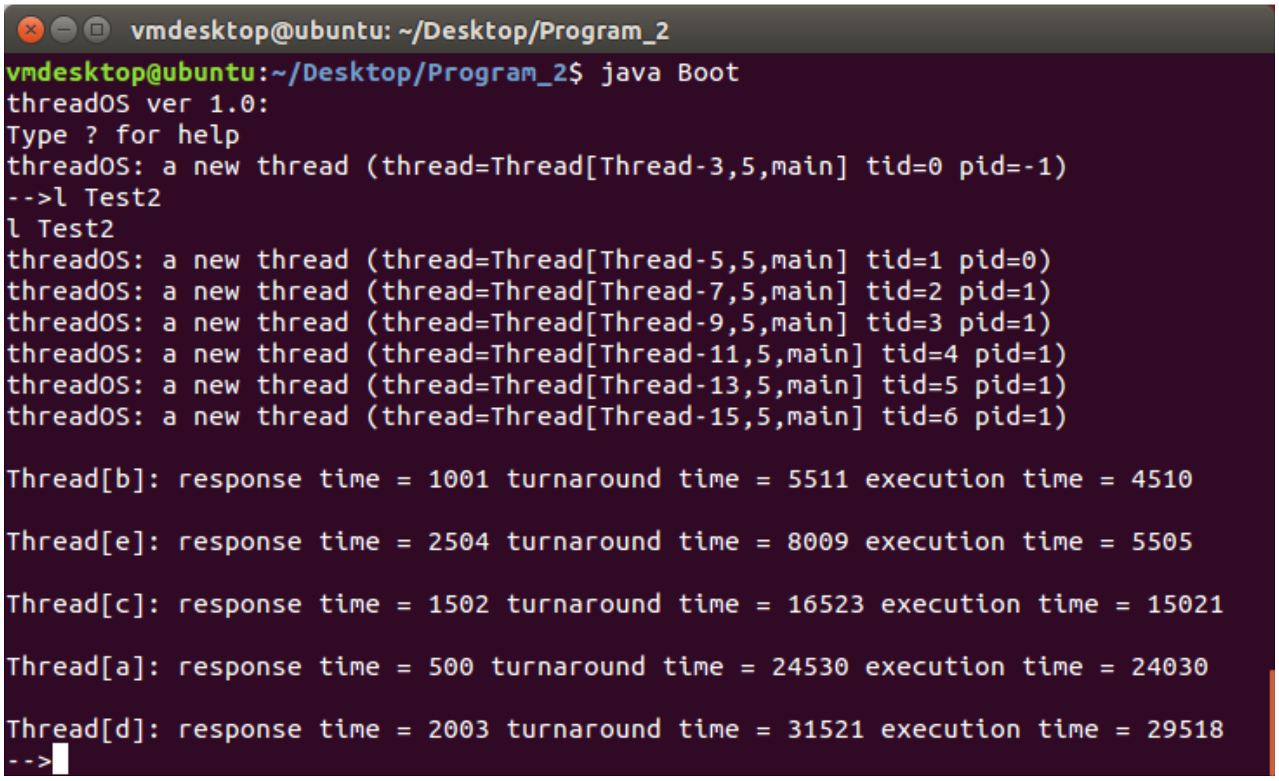
## Output

Round Robin Scheduler (Part 1)



```
vmdesktop@ubuntu: ~/Desktop/Program_2
vmdesktop@ubuntu:~/Desktop/Program_2$ java Boot
threadOS ver 1.0:
Type ? for help
threadOS: a new thread (thread=Thread[Thread-3,5,main] tid=0 pid=-1)
-->l Test2
l Test2
threadOS: a new thread (thread=Thread[Thread-5,5,main] tid=1 pid=0)
threadOS: a new thread (thread=Thread[Thread-7,5,main] tid=2 pid=1)
threadOS: a new thread (thread=Thread[Thread-9,5,main] tid=3 pid=1)
threadOS: a new thread (thread=Thread[Thread-11,5,main] tid=4 pid=1)
threadOS: a new thread (thread=Thread[Thread-13,5,main] tid=5 pid=1)
threadOS: a new thread (thread=Thread[Thread-15,5,main] tid=6 pid=1)

Thread[e]: response time = 6002 turnaround time = 6507 execution time = 505

Thread[b]: response time = 2999 turnaround time = 10006 execution time = 7007

Thread[c]: response time = 4000 turnaround time = 21015 execution time = 17015

Thread[a]: response time = 1998 turnaround time = 29022 execution time = 27024

Thread[d]: response time = 5001 turnaround time = 33027 execution time = 28026
-->
```

Multilevel Feedback Queue Scheduler (Part 2)



```
vmdesktop@ubuntu: ~/Desktop/Program_2
vmdesktop@ubuntu:~/Desktop/Program_2$ java Boot
threadOS ver 1.0:
Type ? for help
threadOS: a new thread (thread=Thread[Thread-3,5,main] tid=0 pid=-1)
-->l Test2
l Test2
threadOS: a new thread (thread=Thread[Thread-5,5,main] tid=1 pid=0)
threadOS: a new thread (thread=Thread[Thread-7,5,main] tid=2 pid=1)
threadOS: a new thread (thread=Thread[Thread-9,5,main] tid=3 pid=1)
threadOS: a new thread (thread=Thread[Thread-11,5,main] tid=4 pid=1)
threadOS: a new thread (thread=Thread[Thread-13,5,main] tid=5 pid=1)
threadOS: a new thread (thread=Thread[Thread-15,5,main] tid=6 pid=1)

Thread[b]: response time = 1001 turnaround time = 5511 execution time = 4510

Thread[e]: response time = 2504 turnaround time = 8009 execution time = 5505

Thread[c]: response time = 1502 turnaround time = 16523 execution time = 15021

Thread[a]: response time = 500 turnaround time = 24530 execution time = 24030

Thread[d]: response time = 2003 turnaround time = 31521 execution time = 29518
-->
```

## Comparison

| | Thread | Response time | Turnaround time | Execution time |
|---|---|---|---|---|
| Round Robin | a (5000) | 1998 | 29022 | 27024 |
| | b (1000) | 2999 | 10006 | 7007 |
| | c (3000) | 4000 | 21015 | 17015 |
| | d (6000) | 5001 | 33027 | 28026 |
| | e (500) | 6002 | 6507 | 505 |
| Multilevel Feedback Queue | a (5000) | 500 | 24530 | 24030 |
| | b (1000) | 1001 | 5511 | 4510 |
| | c (3000) | 1502 | 16523 | 15021 |
| | d (6000) | 2003 | 31521 | 29518 |
| | e (500) | 2504 | 8009 | 5505 |

Unit: ms

1. There is additional 2 time quantums added in for every 5 thread cycle, which can be explained by two additional threads (Loader, Test2) in the scheduler.

2. Based on the above output, MFQ shows a slightly better performance overall.

3. MFC algorithm seems to excel in cases where the burst time varies greatly among threads.

## Implementing Queue2 as FCFS

1. Compared to the RR implementation, FCFS spends much less time in switching and this will likely result in better turnaround time.

2. If there's a thread with large remaining burst in queue2, it will hog the CPU time. This will cause the rest of threads to wait for a long time. In this case, the turnaround time will be longer than RR.

3. The response time won't be affected at all, since all the new threads are checked in first at queue0.

# Thread OS Synchronization

# Part A – Investigating Java Synchronization

## 1. synchronized / wait / notify / notifyAll

By adding the ‘synchronized’ keyword to method’s declaration, lock is enforced in the boundary of method. The program executes only one thread at a time within a monitor, and critical section is wrapped inside a method. wait() is used to give up the lock and wait. notify() is used to signal the waiting thread to wake up. notifyAll() is used to wake up all waiting threads.
In the dining philosopher’s problem, ‘takeForks’, ‘returnForks’, and ‘test’ method can be implemented with ‘synchronized’. wait() can be implemented inside the ‘takeForks’ method, right after the ‘test’ method returns non-eating state. notify() can be implemented inside the test method, after passing the check for both chopsticks’ availability.

Advantage:
- Mutual exclusion in monitor is automatic, preventing the programmers’ improper use of lock.
- Unlike semaphores, there’s no waste of resources due to busy waiting
- Semaphores are never reentrant, while Java monitors are always reentrant.

Disadvantage:
- It doesn’t allow multiple threads to access to a shared resource, since only one thread can acquire the lock.

## 2. Counting semaphores

Counting semaphores can allow more than one lock to be available for threads. This allows multiple threads to enter the critical section as long as the lock is available.
In the dining philosopher’s problem, counting semaphores can be used as a way to enforce the maximum number of philosophers allowed to pick up the chopsticks. In this scenario, the program will only allow at most N - 1 philosophers to be sitting simultaneously. Therefore, the semaphore is initialized to ‘N - 1’, in addition to the mutex semaphore.

Advantage:
- It is more flexible to implement, and can be used anywhere in a program.
- Lock can allow keeping track of multiple resources by using an initial value > 1

Disadvantage:
- Spinlock (Busy waiting)
- It is hard to test and verify whether the lock was correctly implemented in a program.
- Scalability – i.e. it does not scale up well.

## 3. Reentrant locks

Since Java 1.5, the native support for reentrant locks is available in Java API as the ReentrantLock class. It extends the functionality of ‘synchronized’ keyword.

Advantage:
- Unlike ‘synchronized’ constructs, it is unstructured, i.e. there is no need for block structure when locking and unlocking.

Disadvantage:
- More added complexity for programmers

# Part B – How My Solution Works

## Algorithm

1. For monitor, ReentrantLock class in Java API is implemented, which extends the functionality of 'synchronized'. The takeForks, returnForks, test method is wrapped with lock, so that the mutual exclusion in the critical section is enforced.

2. My solution imposes the restriction that a philosopher is allowed to pick up her chopsticks only if both chopsticks are available.

3. There are three states for philosophers.
   - HUNGRY : When a philosopher tries to eat, her state becomes hungry.
   - EATING : If the both chopsticks are available, she can eat and her state becomes eating.
   - THINKING : When a philosopher is not hungry or eating, her state becomes thinking.

4. First, the constructor sets every philosopher to THINKING, and creates the condition variable.

5. The philosopher, denoted by integer number, attempts to takeForks and her state is set to HUNGRY. Then, test method is called to see if she can eat.

6. In the test method, the philosopher can set her state to EATING only if both neighbors are not eating and she is hungry.
   - If she can eat, her state is set to EATING and call signal(), which doesn't have any effect in this case because the current philosopher is not in a waiting state.
   - If she can't eat, her state is set to WAITING and waits on condition variable self[i] until it will be signaled in the future.

7. When a philosopher finishes eating, she returnForks and her state is set to THINKING. Now that she has returned both chopsticks, her neighbors may have a chance to eat. To make sure this happens, she calls test method for both neighbors and release them if the condition is met.

8. This algorithm does not create a deadlock since there is no circular waiting.

# Thread OS Paging

## Algorithm

1. Cache class implements a buffer cache that stores frequently accessed disk blocks in memory.

2. Page table is implemented as an array of Entry objects. Entry type is a private inner class and it holds four attributes: data, blockId, referenceBit, and dirtyBit.

3. First, the constructor initializes the pageTable with Entry objects and sets the victimIndex to the last entry of the pageTable.

4. For reading:
   - First, check blockId to see if it's valid or not, and return false if it's a negative value.
   - Look for the specified block, denoted by blockId, in the pageTable. If found, read the block's data into the passed-in buffer and set the referenceBit to true.
   - If the specified block is not found in the pageTable:
     - First, search for the free page in the page table.
     - If there's no free page, increment the victimIndex and check to see if it was recently used or not. (i.e., if the referenceBit is true or false.)
     - If the referenceBit is false then return the index, otherwise flip the referenceBit and keep looping until it reaches the condition in which the referenceBit is false.
     - After having obtained the index for victim page, check to see if the data had been modified or not. (i.e., if the dirtyBit is true or false.)
     - If the dirtyBit is true, write the data back to the disk and flip the dirtyBit.
     - Now, fetch the block's data from the disk and store into the passed-in buffer using rawwrite.
     - Copy the buffer array and store into the block's data in the pageTable.
     - Finally, save the blockId and set the referenceBit to true.

5. For writing:
   - First, check blockId to see if it's valid or not, and return false if it's a negative value.
   - Look for the specified block, denoted by blockId, in the pageTable. If found, copy the passed-in buffer array and store into the block's data in the pageTable.
   - If the specified block is not found in the pageTable:
     - First, search for the free page in the page table.
     - If there's no free page, increment the victimIndex and check to see if it was recently used or not. (i.e., if the referenceBit is true or false.)
     - If the referenceBit is false then return the index, otherwise flip the referenceBit and keep looping until it reaches the condition in which the referenceBit is false.
     - After having obtained the index for victim page, check to see if the data had been modified or not. (i.e., if the dirtyBit is true or false.)

- If the dirtyBit is true, write the data back to the disk and flip the dirtyBit.
- Copy the passed-in buffer array and store into the block's data in the pageTable.
- Finally, save the blockId and set the referenceBit & dirtyBit to true.

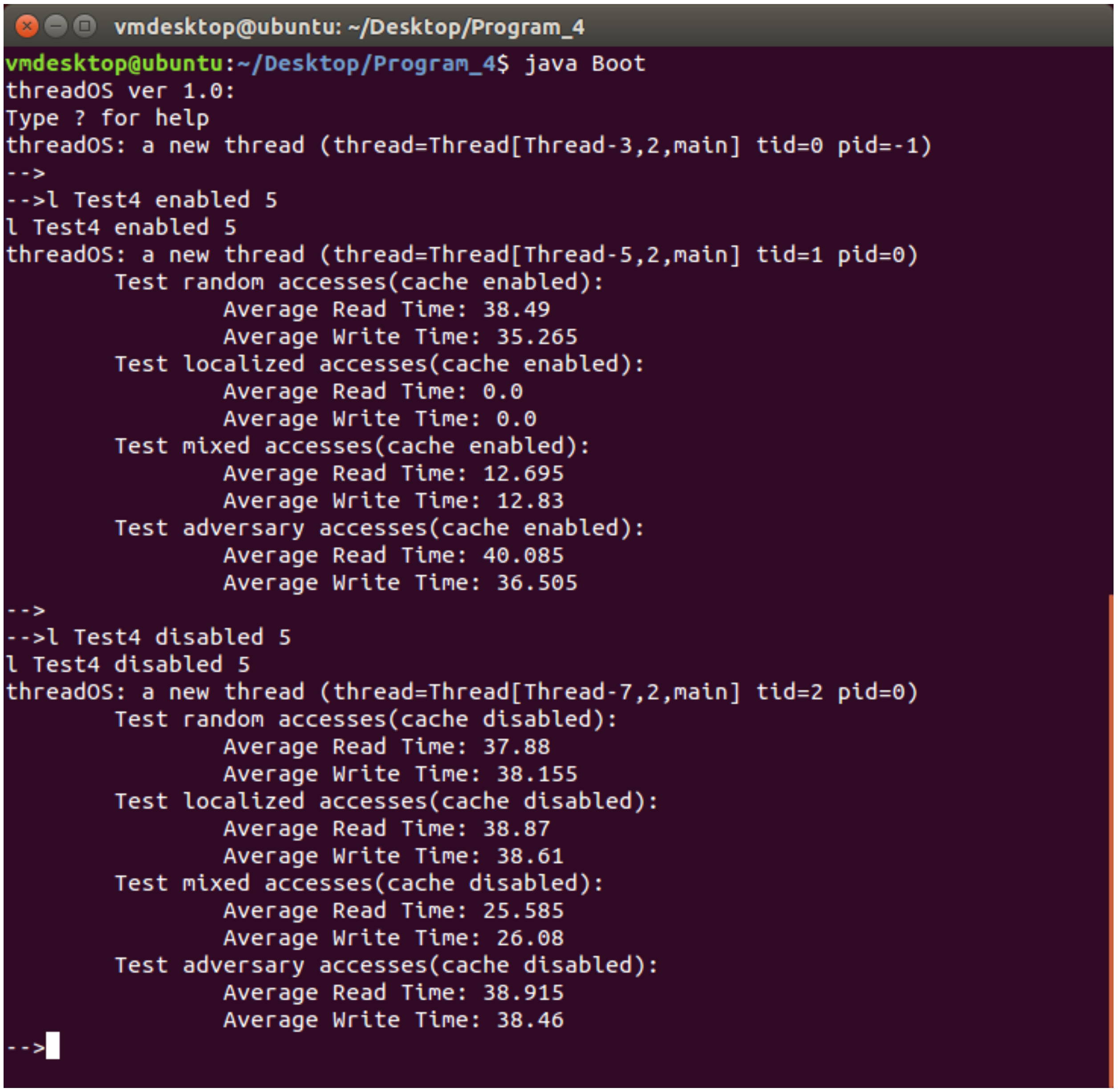
6. If there's a need to clear the cache in memory, flush() method is called.
   - First, it traverses the pageTable and writes any modified data back to the disk.
   - To clean up the pageTable, the blockId, referenceBit, and dirtyBit are reset to the initial values. (-1, false, false)
   - SysLib.sync() is called.

7. Before shutting down ThreadOS, the Kernel class calls sync() method.
   - It traverses the pageTable and writes any modified data back to the disk.
   - SysLib.sync() is called.

## Specification

| methods | descriptions |
|---|---|
| Cache( int blockSize, int cacheBlocks ) | The constructor initializes the pageTable with Entry objects and set the victimIndex to the last page of pageTable. The blockSize is set to the passed-in parameter blockSizeParm. |
| private class Entry{ | Each Entry object corresponds to each page in the pageTable. |
| private int findFreePage() | Finds a free page in the pageTable and returns its index. If not found, returns -1 |
| private int nextVictim() | Chooses the next victim page in the pageTable. It increments the victimIndex and checks to see if the page had been recently used. If the referenceBit is true, then flip the referenceBit and keep looping until it reaches the condition in which the referenceBit is false. |
| private void writeBack(int victimEntry) | Writes back all the pages that have been modified. If the dirtyBit is true, write the data back to the disk and flip the dirtyBit. |
| private byte[] copyArray(byte original[]) | Copies the original array to the newArray and returns it. |
| private void readIntoBuffer(byte original[], byte buffer[]) | Copies the original array into the passed-in buffer array |
| public synchronized boolean read(int blockId, byte buffer[]) | Reads into the buffer array the cache block specified by blockId from the disk cache if it is in cache. Otherwise, reads the corresponding disk block from the disk device. |
| public synchronized boolean write(int blockId, byte buffer[]) | Writes the buffer array contents to the cache block specified by blockId from the disk cache if it is in cache. Otherwise, finds a free cache block and writes the buffer contents on it. No write through. |
| public synchronized void sync() | Writes back all dirty blocks to Disk.java and thereafter forces Disk.java to write back all contents to the DISK file. It doesn't reset the contents in the pageTable. |
| public synchronized void flush() | rites back all dirty blocks to Disk.java and thereafter forces Disk.java to write back all contents to the DISK file. Resets all the entries in the pageTable. |

## Results



vmdesktop@ubuntu: ~/Desktop/Program_4

```
vmdesktop@ubuntu:~/Desktop/Program_4$ java Boot
threadOS ver 1.0:
Type ? for help
threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)
-->
-->l Test4 enabled 5
l Test4 enabled 5
threadOS: a new thread (thread=Thread[Thread-5,2,main] tid=1 pid=0)
        Test random accesses(cache enabled):
                Average Read Time: 38.49
                Average Write Time: 35.265
        Test localized accesses(cache enabled):
                Average Read Time: 0.0
                Average Write Time: 0.0
        Test mixed accesses(cache enabled):
                Average Read Time: 12.695
                Average Write Time: 12.83
        Test adversary accesses(cache enabled):
                Average Read Time: 40.085
                Average Write Time: 36.505
-->
-->l Test4 disabled 5
l Test4 disabled 5
threadOS: a new thread (thread=Thread[Thread-7,2,main] tid=2 pid=0)
        Test random accesses(cache disabled):
                Average Read Time: 37.88
                Average Write Time: 38.155
        Test localized accesses(cache disabled):
                Average Read Time: 38.87
                Average Write Time: 38.61
        Test mixed accesses(cache disabled):
                Average Read Time: 25.585
                Average Write Time: 26.08
        Test adversary accesses(cache disabled):
                Average Read Time: 38.915
                Average Write Time: 38.46
-->
```

| Avg. Read / Write | Random | Localized | Mixed | Adversary |
|---|---|---|---|---|
| **enabled** | 38.49 / 35.26 | 0.0 / 0.0 | 12.69 / 12.83 | 40.08 / 36.50 |
| **disabled** | 37.88 / 38.15 | 38.87 / 38.61 | 25.58 / 26.08 | 38.91 / 38.46 |

## Random Access

Random access for cache enabled and disabled both gives similar results. This is expected because the 200 blocks being read and 200 blocks being written are random, there is higher chance of having to go to disk for fetching the blocks, and making it unlikely to have any benefits of using cache.

## Localized Access

With the cache enabled, reading and writing the small selection of blocks gets really fast. This is because the same small working set in cache gets accessed repeatedly and there is almost no cost of going to disk, except when the blocks are read from disk for the first time. With no cache, on the other hand, reading and writing the same blocks from the disk does not benefit from locality.

## Mixed Access

Mixed access is faster than random access but slower than localized access. This is expected from 90% of accesses being localized and only 10% being randomized. Without cache, mixed access is actually faster than localized access, random access, and adversary access.

## Adversary Access

Adversary access test runs at similar speeds with and without cache, because this test was specifically designed to access different disk blocks as much as possible and never access the same blocks more than once, which eliminates the benefit of a cache, therefore it is expected to run slower than random access.