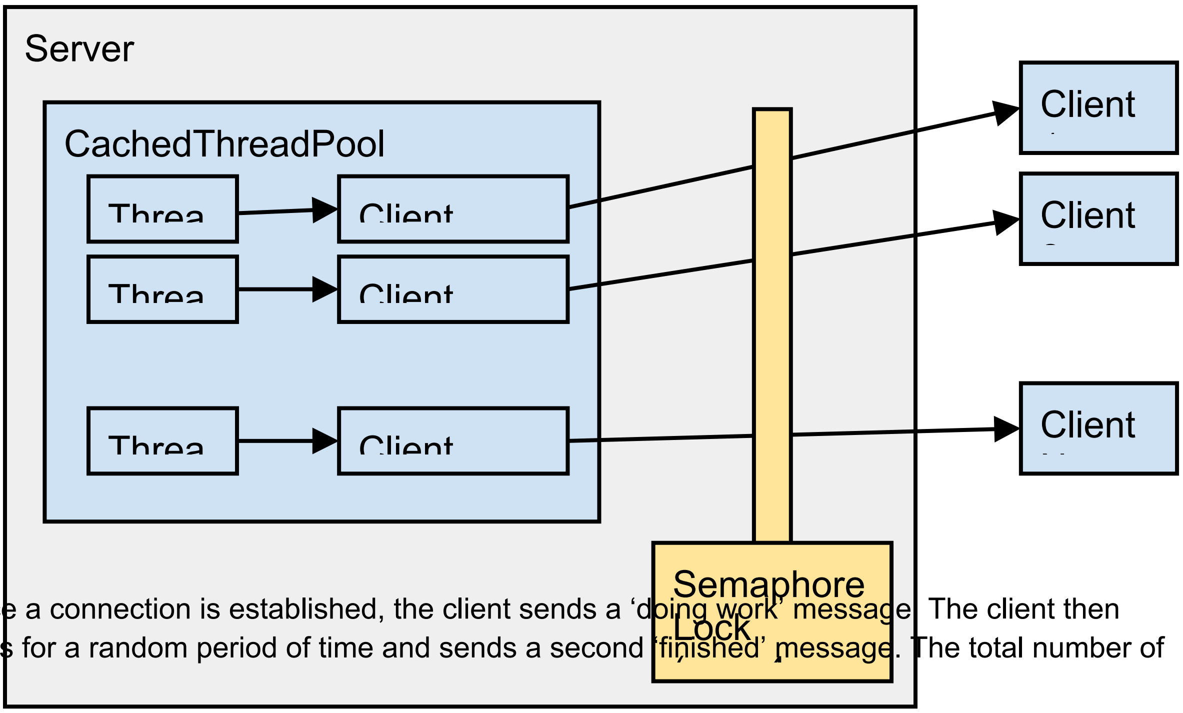
Assignment 3 Report

Damian Polan
Nathan Bosscher

Part 1 - Limited connection Server

The main goal in this section is create a client server model which utilizes a thread pool and a semaphore lock in order to limit the number of client connections.
The advantage of using a thread pool is prevalent when new tasks need to be run. Instead of destroying and recreating new threads, a common thread pool is used to assign each task. This eliminates any extra overhead produced by thread allocation.
The thread pool size is limited through the use of a semaphore lock. In this example, a maximum of 4 threads will be allocated to handle A maximum of 4 client connections. This eliminated the problem of a overly high number of client applications connecting to the server and overloading the thread count.
If the maximum thread count is exceeded, the client must wait to connect to the server.

Below is a high level diagram of the program architecture. A CachedThreadPool is used to maintain every incoming client connection. In order to establish a message send-receive connection, the semaphore must be acquired by the associated ClientConnection.



Once a connection is established, the client sends a ‘doing work’ message. The client then waits for a random period of time and sends a second ‘finished’ message. The total number of

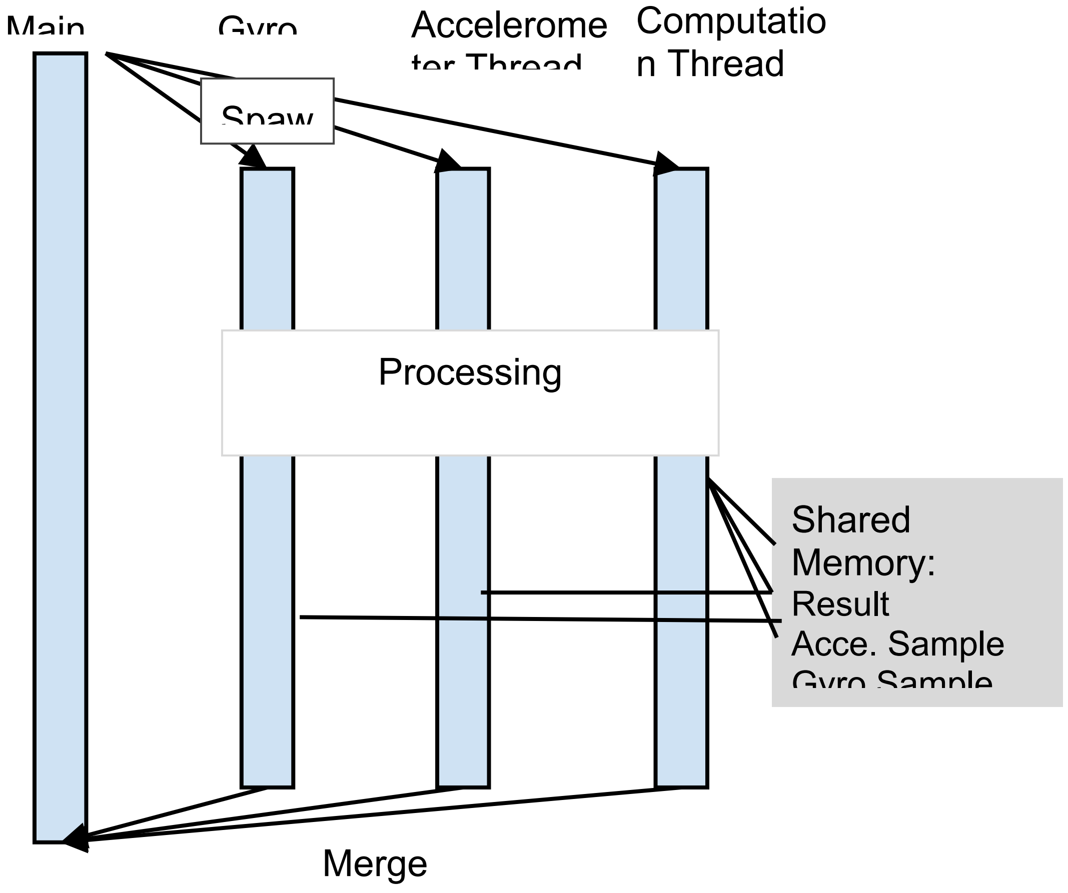
connections at any given moment is tracked and printed at each client connection as proof for not exceeding the maximum of 4.
For testing, one server and ten clients are started.
The following is the output stream of the program:

```
Starting Server...
Starting Client Connection...
Client 1: Hello. I am doing work.
Total Client Connections: 1
Client 3: Hello. I am doing work.
Total Client Connections: 2
Client 0: Hello. I am doing work.
Total Client Connections: 3
Client 2: Hello. I am doing work.
Total Client Connections: 4
Client 1: Bye. I am finished.
Client 4: Hello. I am doing work.
Total Client Connections: 4
Client 2: Bye. I am finished.
Client 5: Hello. I am doing work.
Total Client Connections: 4
Client 3: Bye. I am finished.
Client 6: Hello. I am doing work.
Total Client Connections: 4
Client 0: Bye. I am finished.
Client 7: Hello. I am doing work.
Total Client Connections: 4
Client 4: Bye. I am finished.
Client 8: Hello. I am doing work.
Total Client Connections: 4
Client 5: Bye. I am finished.
Client 9: Hello. I am doing work.
Total Client Connections: 4
Client 6: Bye. I am finished.
Client 7: Bye. I am finished.
Client 8: Bye. I am finished.
Client 9: Bye. I am finished.
```

It is evident that number of connections never exceeds 4 and therefore the program goal is met.
One problem should be noted.
A Future<> data type was attempted to be used for retrieving the total time spent connected to each of the client applications. In this case, a Future<>.get() call is made in order to get the requested value. The problem comes because Future<>.get() is a synchronous call, it blocks the thread. This results in a maximum of one client being able to connect at any given time.
This can be solved by creating wrapper threads for the pooled connections. Unfortunately this would just create more threading issues. A second solution would be to set a global variable and tally all the results once all the connections close.

Part 2 - Xenomai

I am trying to create a monotonic realtime system that runs 3 separate tasks. The tasks will be statically scheduled by the xenomai system and run periodically as separate threads. The threads must access and manipulate shared memory in order to complete the problem in this assignment. There are 3 sections of shared memory that must be protected from corruption.



The system design is simply 4 threads. 1 main thread spawns 3 periodic tasks which run until the program is exited when the 3 tasks merge to the main thread. Each task solved 1 problem posed in the assignment. 1 to monitor the gyro, 1 to monitor the accelerometer and 1 to compute the result. The critical sections are where each task either writes or reads to the shared memory. Since access of data as it's being updated can cause corrupted data to be read, mutexes were used to control access.

Testing was performed by running the program for a fixed length of time that was much longer than the longest task period and observing the output (see assignment files for test output). Validating that the output was indeed periodic and there was no segment faults to show an invalid access of memory. The inputs were not largest concern in testing this program since the problem was the task scheduling. Constant integer values were used to test the system and the expected results were returned. The program does not deadlock. I ran the program for ~2 minutes which produces a large sample of periods to check for deadlock situations. No threads were starved. I tested for this by running the program for ~2 minutes and reviewing the test output for a consistent pattern of outputs.

We have met the goal of creating a multi-task system that relies on several real time period tasks.