Shufelt, Jake A | Koebeles, Charles | Kuzko, Joseph | Abdul, Anuoluwapo

St. CLOUD STATE UNIVERSITY  FALL 2022

CSCI 312

Project 5

**Program Design Summary**

**Forming The Network and Coordinator Election**

The program is designed such that five instances of the program must be opened with differing command line arguments to connect to each other and form a ring. Each process is provided arguments to specify their philosopher number, and the number of their successor in the ring. These numbers correspond to port numbers for establishing a TCP connection to their successor. As the connection is made, a connection message is relayed down the chain ending at the first process to be opened. Once the first process sees that it has four predecessors, it will establish a connection to the newest process forming a ring topology.

Now that each process is connected in a ring, the program to complete the ring will prompt an election while simultaneously selecting randomized IDs for each process. This way the IDs may be unique since only one process may generate them. Election messages are received from predecessors, interpreted, and then forwarded to successors. The election message may take two forms, an “A” or “B” election message. If the message is prefaced with an A, then it symbolizes that an election is taking place and that the coordinator/election winner has not yet seen the message. Once the coordinator has seen the message, they will modify the message to a B message and forward it along. This is important since the coordinator will act as a centralized server to all other philosopher processes. When receiving a B message, a philosopher will know it’s safe to connect to the coordinator because the coordinator has seen the message and has started hosting. Once the B message has gone full circle and the coordinator receives a B message, they will know everyone has seen the message. This is used as trigger to begin the “Dining Philosopher” routine.

Diagram

Description automatically generated

**Dining Philosopher Routine**

During the coordinator connection process, the process acting as the coordinator will create an additional four threads to handle each process TCP connection, as well as an additional thread to act as its philosopher, while the main thread will act as the coordinator. Each philosopher handling thread will listen to fork requests coming from their corresponding philosopher process, upon receiving these requests, they are written to a pipe that is used as a queue. The coordinator’s own philosopher thread does not require a handling thread since it is a part of the coordinator process and shares the same memory, so it directly writes its requests to the pipe queue. The coordinator main thread of execution reads from the pipe and handles requests one at a time. As the coordinator reads the request from the pipe, it decodes it to determine the requesting philosopher, the forks requested, and the socket descriptor from which the message was received. If the forks are available, the coordinator marks them as allocated, and sends a message across socket directly to the philosopher, allowing them to begin eating. In the case of the coordinator granting forks to its internal philosopher thread, a thread condition is signaled instead. If the requested forks are unavailable, the coordinator waits until they are available and promptly grants the request. Once philosophers are done eating, they send a deallocation message to their handling thread which marks the forks as available. Only the coordinator may allocate forks according to the queueing, but deallocation happens promptly by handling threads.

For the program to work logically, a mutex is used to guard any changes made to the forks whether allocation or deallocation. Likewise, the coordinator process uses a mutex for standard output so that its many threads do not overwrite each other, and messages may be seen as they happen in order.

Diagram

Description automatically generated

**Manual Pages**

**NAME**

Philosopher (executable)

**COMPILATION**

This executable may be compiled using the following compilation method in a UNIX environment

*g++ philosopher.cpp philosopher.h -pthread -std=c++11 -o Philosopher*

**EXECUTION**

In order for the executable to function properly, it must be executed in five instances with varying command line arguments. Each instance must be executed with the following arguments in the following order.

1. ./Philosopher 0
2. ./Philosopher 1 0
3. ./Philosopher 2 1
4. ./Philosopher 3 2
5. ./Philosopher 4 3

The first argument is the philosopher number, and the second argument is the successor philosopher. This execution pattern allows the processes to connect and form a ring topography to facilitate coordinator election. The ordering is important, because the processes must be running with their philosopher number before another philosopher attempts connection.

**DESCRIPTION**

Upon successful execution (see **EXECUTION** above), there are many outputs to be witnessed by each Philosopher.

**OUTPUTS**

***Arguments = X***

Outputs number of arguments provided to program at execution.

***Connection Status = 0***

Indicates that connection to successor was successful

***Connection was established to port X***

Indicates that connection to successor was successful

***Message received: X***

Printout of message received from predecessor

***Received Election Message, and forwarding***

Displayed when received message was an election request

***Philosopher ID: X***

Displaying ID as determined by election request

***Received First Election Request… X***

Displayed if first time seeing election request

***I am the coordinator : )***

Displayed if the process has the highest ID, as they are elected coordinator

***Coordinator Accept Thread started.***

Coordinating process has started to accept connections from other philosopher processes

***All predecessor philosophers connected.***

Displayed by philosopher 0 when four predecessors have connected

***Attempting to complete chain…***

Philosopher 0 is attempting to connect to philosopher 4 to form a ring topology

***|COORDINATOR| Connection handling subthread created…***

A philosopher process has connected to the coordinator process and a subthread has been created to handle its fork requests.

***Philosopher ID: X has connected***

This philosopher has connected to the coordinator and is being handled by the subthread.

***|COORDINATOR| RECEIVED REQUEST: X***

A philosopher has requested forks through TCP socket to a coordinator subthread, which has forwarded the request to the main coordinator thread.

***|COORDINATOR| HANDLING REQUEST: X***

The coordinator is handling the following request, attempting to allocate forks to the requester as soon as possible.

***PHIL ID: X, may begin eating.***

The philosopher with corresponding ID has been granted the forks and given permission to eat by the coordinator.

***ALLOCATING LEFT FORK: X RIGHT FORK: X***

Message describing recent fork allocation

***|COORDINATOR| Deallocating, LEFT: X RIGHT: X***

Message received by handling threads indicating that the following forks have been deallocated by a philosopher

***Begin thinking for X seconds***

Message sent from philosopher indicating it has begun thinking.

***Done thinking***

Message sent from philosopher indicating it has completed thinking.

***Requesting left fork***

Philosopher is sending request for its left fork.

***Requesting right fork***

Philosopher is sending request for its right fork.

***Left fork acquired***

Philosopher has received message indicating fork has been allocated.

***Right fork acquired***

Philosopher has received message indicating fork has been allocated.

***Begin eating for X seconds***

Philosopher has begun eating.

***Done eating***

Philosopher has finished eating.