Design Document: Mini Project Two

Courtney Gosselin - 60186160
Benjamin Tisserand - 37615168
Alex Qin - 35732156
COSC 315 - Operating System
Dr. Apurva Narayan
March 4, 2020

Table of Contents

1.0 Introduction:

The goal of this project is to implement a multi-threaded request scheduler. Our design for this is a bounded buffer producer-consumer framework. Assume that the request queue is a circular buffer of size N (i.e., an array of size N). The method involves a single producer, which is the master thread, and N consumers, which are the slave threads. The master thread will sleep for a random short duration and produce a request. Each request has a sequentially increasing request ID and a randomly chosen request length (assign each new request a random length between 1 and M seconds). The master thread then inserts the request into the queue and goes back to sleep for a random short duration before it produces another request. Of course, if the request queue, which is a bounded buffer, is full, the master thread must wait before it can insert the request into the queue. Each slave thread can be idle or busy. When a slave thread is idle, it acts as a consumer waiting for a new request in the request queue. After it consumes a request from the queue, the slave thread will be busy for a duration that is equal to the request length for that request. The busy state of a slave thread is emulated by making the thread sleep for that duration. Upon completing the request, the slave thread goes back to idle thread and attempts to consume a pending request from the request queue. If the queue is empty, the slave thread must wait, like a consumer in the producer consumer problem.

In part 1: we implement the above problem using Java and Monitors.

In part 2: we implement the above problem using C and semaphores.

2.0 Prioritization/Planning:

The following is a breakdown of the project into a list of measurable and timed checkpoints

Product functionality:

- Use Java and Monitors to implement producer-consumer framework (Feb, 24th)

- Use C and semaphores to implement producer-consumer framework (Feb, 29th)

- Manage code structures and put comments on (Mar, 1st)

- Design document (Mar, 3rd)

- README.md (Mar, 4th)

3.0 Contributions:

Benjamin:

- Java and C code with Logic Design
- Planning
- Documentation (Design document)
- Document Editing

Courtney:

- Base code in C
- Debugging/implementation of C
- Scrum master
- Documentation (Design document, README)
- Planning

Alex:

- Planning
- Feedback and Testing

- Syntax research for C and pthreads
- Documentation (Design document)

4.0 Code Outline:

4.1 Java outline

Code in Java: First, we construct our classes into: Job, Buffer, MasterThread, SlaveThread, and the main class which we called Run.

Job class: Wrapper class for the Job object, which specifies the job (or request) ID and length.

Buffer class: Defines the shared buffer for the threads. Has private variables startPointer and endPointer to specify the next index to be filled and the currently filled index that has existed the longest, respectively. It also tracks the max length of the buffer and, of course, the buffer data, which is an array of Job objects. The remaining functions are public, synchronized, and used to add or remove things from the buffer data array. This class acts as a monitor and as such the array data and pointers to it are edited atomically.

MasterThread class: The master thread class extends from thread class. Master thread has its own buffer (described above and shared with the slave threads), and keeps track of maximum request creation delay, maximum job duration, and next job ID. In the constructor, we initialized the next job ID to zero, and the rest are inputted upon the creation of the thread (in our case, decided by user input). We override the Thread's, or more accurately, the Runnable interface's run method to define this thread's functionality. This thread waits for an empty spot on the buffer, then accesses the buffer's add function to create and place jobs into it. Along the way it produces informative output such as the ID, length, and creation time of a job it produces and puts on the buffer. It then outputs that it will sleep for some amount of time, which is random but has a determined maximum, and does so.

SlaveThread class: As with the master thread, the slave thread extends from the thread class. Slave threads have a buffer shared with the master thread, and a unique slave ID. The slaves act as consumers, consuming and 'processing' the Job objects on the buffer. In the slaves overridden run method, we wait for a Job to be placed on the buffer, then use the buffer's remove function to get the Job and remove it from the buffer. This job is then used to create an output notifying the user of the job ID, length and the current time. Finally, the thread sleeps to simulate processing of the job and outputs that this request has been satisfied.

Run class: This is our main class and the one used to run the program. First, the user is asked to input the value for buffer size, numbers of slave threads, maximum request length and maximum request creation delay. Then we construct objects from parameters: by using buffer size to generate our buffer; by using this buffer, maximum request length and maximum request creation delay to generate our master thread; and by using numbers of slave threads to generate an array of slave thread type and define how many threads need to be created. We create these slaves in a loop and assign them unique IDs and the buffer, shared with the master. At the end we run the master thread, and then use a for-each loop to run slave thread.

4.2 C outline

Code in C: In C, we created structs for data that needed to be grouped, such as Job, threadArgs, and Buffer. For the threads, we created two dedicated methods for the threads to follow, explained below.

Structure Job: Struct that defines data relevant to a specific job: ID and length.

Structure Buffer: As with the java buffer, our buffer in C defines a start and end pointer to keep track of what stretch of our buffer array is in use, length to keep track of its max length, and an array of Job structs.

Structure ThreadArgs: This struct is used to pass in all relevant information to a consumer thread upon creation. This includes a unique consumer ID and a pointer to the shared buffer.

printTime method: This is a quick helper method using the time.h library to print out the current time in the format of “HH:MM:SS”.

Producer method: This method is used by the producer thread to continually create jobs to add the the buffer array. It first waits for the semaphore denoting the amount of empty slots in the buffer, ensuring that it will only create a job if there is space on the buffer for one. Once there is free space, it creates a Job with a unique incrementing ID and a random length, the maximum of which is defined by the user in a global variable. After that, it waits for the mutual exclusion lock, then prints job details and adds it to the buffer in the endpointer index. It then releases a space in the full semaphore, indicating that there is another job on the buffer for a consumer to take, and potentially waking up a waiting consumer. Finally, it sleeps for a random period of time, the max of which is also defined by the user, and prints out that time.

Consumer method: This method is used by the consumer threads to continually remove jobs from the buffer and simulate satisfying the requests. First, we define a ‘nullJob’ which is used to denote an empty space in the buffer. Then, inside an infinite loop, we use our semaphore denoting the amount of full spaces in the buffer array to wait for there to be a request in the buffer. It then waits for the mutual exclusion lock, then saves the job at the startpointer index to memory and sets that index to nullJob (simulating an empty space). After that, it rotates the startpointer to the next index and releases the mutex, then prints the consumer ID with current time, job ID and its length. Next, the thread sleeps for the specified request length, simulating time for processing the request. In the end, it prints out the completed job ID with the current time.

The main: We start by asking the user to input the value for buffer size, number of consumer threads, maximum request length, and maximum request creation delay. Then we construct objects from parameters: by using buffer size to generate our buffer, using pthread to initialize our producer, using pthread and numbers of consumer threads to initialize an array of consumer threads and define how many threads need to be created. After that, initialize the ‘full’ semaphore to zero to ensure the consumer threads don’t attempt to retrieve a request from the initially empty buffer. We also initialize the ‘empty’ semaphore to the size of the buffer to ensure the producer may create that many requests before it is full. Next, we use a loop to create and assign thread arguments into an array and assign pointers to those array indices to our new consumers, which are also created in this loop. Lastly, we create the producer using a pointer to the shared buffer, and sleep for 30 seconds to ensure the program ends after that time.

5.0 Code structure/Folder layout:

6.0 Build, Compilation, and Run:

The following contains the build instructions stating how to compile the code on a Linux OS for both our Java and C programs. It also contains Run instructions stating how to execute the compiled code on Linux for both Java and C program implementations.

6.1 Java Build, Compilation, and Run

The Java program has already been compiled with javac *.java inside the project folder of Part1-Java. To run the program you need to be outside the project folder in the src folder and run the command "java project.Run" to start the program where you will then be prompted for your information.

1. Clone the repository to your local machine
2. Use Javac for compilation of the program using the code below. This needs to be ran inside the the folder with the .java files for our repository that is the Part1-Java/Miniproject2/src/project
    - Javac *.java

3. To run the program go to the src/ file in this case and type the following code into the terminal seen below. This will start the program which will begin prompting you for information to run the threads and synchronization file **(start here if you are using our compilation)**
    - java project.Run
4. You will be prompted to input the information as seen below. The program is expecting the individual to input integers.
    - "Input the buffer size: "
    - "Input the amount of slave threads: "
    - "Input the max request creation delay: "
    - "Input the max request length: "

5. After taking these inputs the program will run for 30 seconds - the current ending time we have set. However, this can be changed inside the source code to be run indefinitely. See `7.0 Code output example` for an example of the program executing.

6.2 C Build, Compilation, and Run

The C program has already been compiled and is part of the GitHub repository under Part2-C of the folder tree. However if you want to compile and run the program yourself you can navigate to the file and within that section of the folder tree run the following:

1. Clone the repository to your local machine
2. Use the pthread flag with gcc to compile this code:
    - gcc -pthread producer-consumer.c -o producer-consumer
3. To run the program you need to type the following into the terminal **(start here if you are using our compilation)**
    - ./producer-consumer
4. You will be prompted to input the information as seen below. The program is expecting the individual to input integers.
    - “Input number of consumer threads: “
    - “Input size of buffer: “
    - “Input max request length: “
    - “Input max producer wait time: “
5. After taking these inputs the program will run for 30 seconds - the current ending time we have set. However, this can be changed inside the source code to be run indefinitely. See `7.0 Code output example` for an example of the program executing.

7.0 Code output examples:

The following is sample output from both our Java and C programs in their terminals.

7.1 Examples in Java:

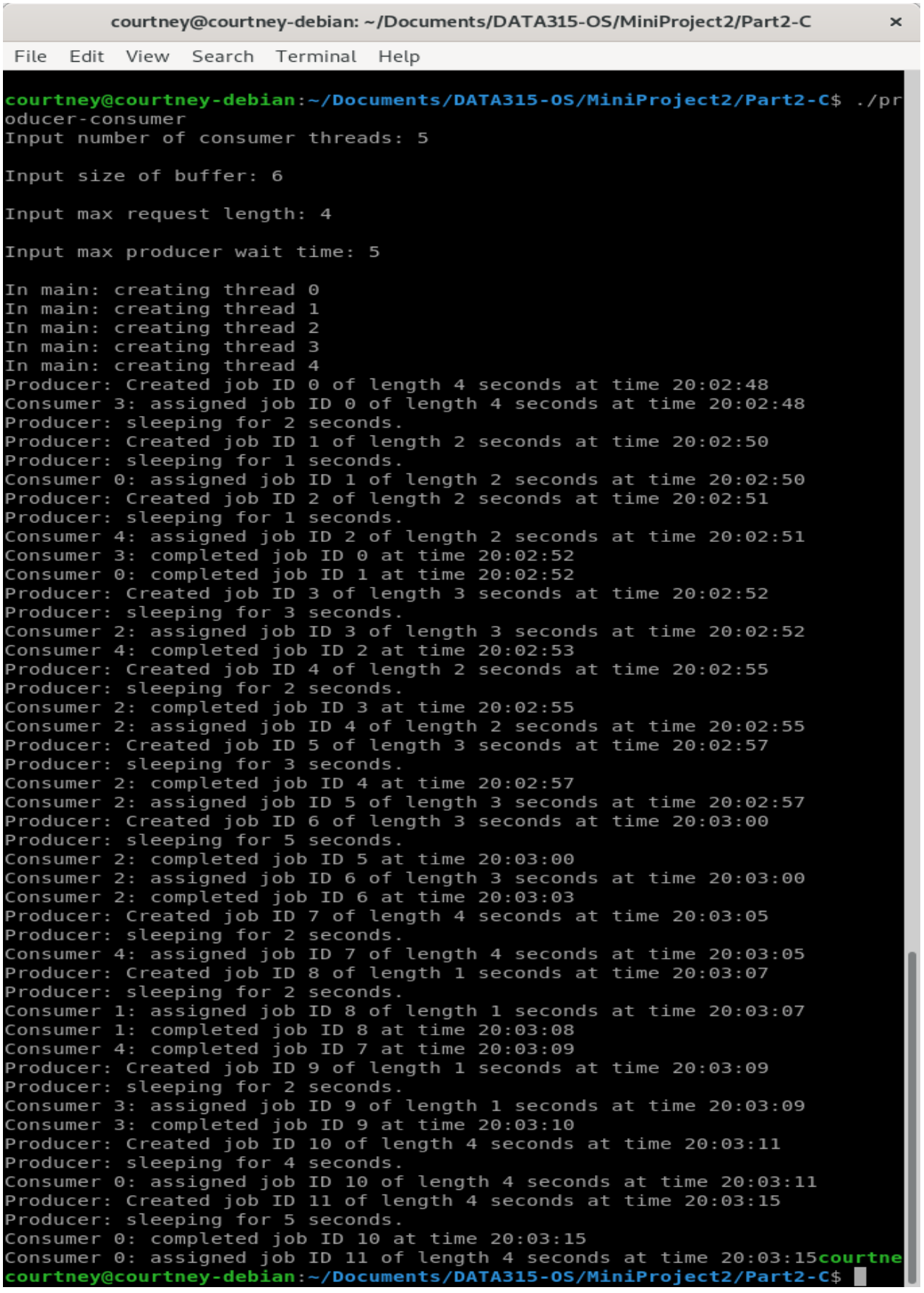
```
coolc@DESKTOP-ARN85S7 MINGW64 ~/OneDrive/Desktop/UBC/Semester2/COSC315 OS/Projec
ts/MiniProject2/Part1-Java/src (master)
$ java project.Run
Input the buffer size: 5

Input the amount of slave threads: 6

Input the max request creation delay: 7

Input the max request length: 4
Producer: Produced request ID 0, length 3 seconds at time 20:38:14
Producer: sleeping for 2 seconds
Consumer 1: assigned request ID 0, processing request for the next 3 seconds, cu
rrent time is 20:38:14
Producer: Produced request ID 1, length 1 seconds at time 20:38:16
Producer: sleeping for 3 seconds
Consumer 2: assigned request ID 1, processing request for the next 1 seconds, cu
rrent time is 20:38:16
Consumer 2: completed request ID 1 at time 20:38:17
Consumer 1: completed request ID 0 at time 20:38:17
Producer: Produced request ID 2, length 3 seconds at time 20:38:19
Producer: sleeping for 2 seconds
Consumer 0: assigned request ID 2, processing request for the next 3 seconds, cu
rrent time is 20:38:19
Producer: Produced request ID 3, length 4 seconds at time 20:38:21
Producer: sleeping for 5 seconds
Consumer 1: assigned request ID 3, processing request for the next 4 seconds, cu
rrent time is 20:38:21
Consumer 0: completed request ID 2 at time 20:38:22
Consumer 1: completed request ID 3 at time 20:38:25
Producer: Produced request ID 4, length 2 seconds at time 20:38:26
Producer: sleeping for 4 seconds
Consumer 4: assigned request ID 4, processing request for the next 2 seconds, current time is 20:38:26
Consumer 4: completed request ID 4 at time 20:38:28
Producer: Produced request ID 5, length 4 seconds at time 20:38:30
Producer: sleeping for 1 seconds
Consumer 1: assigned request ID 5, processing request for the next 4 seconds, current time is 20:38:30
Producer: Produced request ID 6, length 4 seconds at time 20:38:31
Producer: sleeping for 6 seconds
Consumer 4: assigned request ID 6, processing request for the next 4 seconds, current time is 20:38:31
Consumer 1: completed request ID 5 at time 20:38:34
Consumer 4: completed request ID 6 at time 20:38:35
Producer: Produced request ID 7, length 4 seconds at time 20:38:37
Producer: sleeping for 4 seconds
Consumer 0: assigned request ID 7, processing request for the next 4 seconds, current time is 20:38:37
Consumer 0: completed request ID 7 at time 20:38:41
Producer: Produced request ID 8, length 1 seconds at time 20:38:41
Producer: sleeping for 4 seconds
Consumer 4: assigned request ID 8, processing request for the next 1 seconds, current time is 20:38:41
Consumer 4: completed request ID 8 at time 20:38:42
```

7.2 Examples in C:



8.0 Implementation Experience

The following is a brief discussion describing the implementation of the project in both Java and C, as well as comments on the amount of effort and the ease of coding the problem we had as a team programming in both Java and C.

8.1 Implementation Experience for Java

As something we were familiar with, the Java implementation was far easier to complete than the C one. Structuring the project and making use of what the Java language and libraries had to offer was extremely straightforward and we ran into very few problems.

8.2 Implementation Experience for C

Creating this program in C was far more difficult. The struggles we faced affected each other and the difficulty increased exponentially with each layer of unfamiliarity. Compared to coding Java in an IDE with an easy to use thread library and a synchronized keyword, moving on to coding in VIM in a low level language that none of us have used much was a nightmare. We started with a solid skeleton of partial C code and some pseudo-code, and moved on to fill it with logic from our Java implementation and fixed errors from there. The semaphores library was very helpful, but using C pointers and data types like structs made logic very hard to translate into code. In the end we made a program that we are proud of, but the fact remains that it was at least four times longer to create than the Java part.