Implementing Load Balancing

Summary: In this homework, you will be implementing the main multi-threaded logic for doing batch-based server load balancing using mutexes.

1 Background

In this assignment, you will write a batch-based load balancer. Consider a server that handles data processing based on user requests. In general, a server has only a fixed set of hardware resources that it can use to satisfy user requests. This is problematic since servers can become overloaded. One common scaling feature that is implemented is load batching (a form of *load balancing*). On modern platforms, it is possible to spin up a virtual *instance* server using a cloud environment (e.g., AWS, Microsoft Azure, Google Cloud) to handle specific needs. The idea of cloud computing has gained traction as a way to deal with scalability and maintenance issues. Examples include internet-based storage, computing, or services, which are accessed by client-based devices or by web browsers. As more requests are made, more servers instance are spawned. This gives the ability to seamlessly scale up to peaks in user requests. Although this provides a good way to scale resources, the cost of spinning up a new server is non-trivial. Thus, *gateway* servers tend to collect a *batch* of requests before starting a server instance. Your goal in this assignment is to implement the multi-threaded load balancer server logic that will mimic the process of creating server instances to process batches of requests as they come in. *Note that while this system is conceptually a network situation, we will be modeling it as a set of functions spread across three files.*

At a high level, the load balancer server waits for a specific number of requests to be made before spawning an instance server in a cloud-like environment, and initializing the instance with the appropriate work. We call the number of requests that should be serviced the *batch size*, and assume that the server instances are designed so they can handle exactly *batch size* number of requests at a time. The load balancer starts to form a batch as soon as it receives as least one request and continues to add requests to the batch until the batch is full (i.e., the batch size is reached). At that point, the load balancer will spawn a new server instance using the instance host to begin processing the batch. The load balancer makes sure that server instances are only initialized when enough requests have been made. Internally, the load balancer will add requests to a list which is then forwarded to an instance server once it reaches the appropriate size. As the requests are handled by the individual server instances, each instance will store the result into an output location given by the creator of the request.

This document is separated into five sections: Background, System Architecture, Requirements, Include Files, and Submission. You have almost finished reading the Background section already. In System Architecture, we discuss the conceptual view of the system as the interactions between users, a load balancer, and an instance host. In Requirements, we will discuss what is expected of you in this homework. In Include Files, we discuss the the various libraries and function calls you may find useful in the assignment. Lastly, Submission discusses how your source code should be submitted on Canvas.

2 System Architecture

Conceptually, the system is divided into three main components: users, the load balancing server, and the server instance host. Figure 1 shows these components and how they are related.

1. **User:** A user is a person who utilizes the system to get a job done. Users make requests to the load balancer server. A request consists of the id of the user, some data payload, and a place to store the result of processing. A request is also called a *job*. On completion of the request from the server, the result (which will be computed by some instance) will be stored in the location specified in the request.

2. **Load Balancer Server:** The load balancer server acts as a gateway that is a middle layer between server instances and the users, it is used to distribute incoming traffic to different servers capable of fulfilling those requests in a manner that maximizes throughput, and ensures that no one server is overworked.
3. **Server Instance Host:** The server instance host is a cloud-like environment that can dynamically create server instances to deliver resources (e.g., compute time) on demand. Changes to one server instance do not have any effect on other instances. Unlimited server instances can be run within a host environment at any given time. Each server instance takes a batch of requests, and processes it.

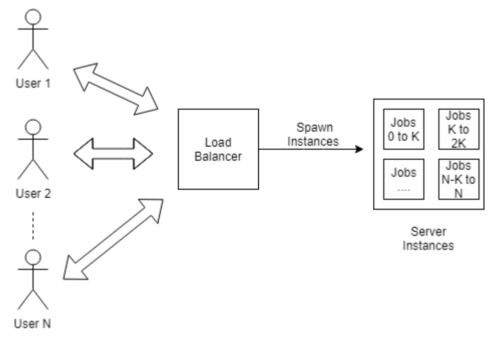

Figure 1: Structural overview of the system containing users, a load balancer, and a server host.

Figure 2 shows an overview of the system flow. Behaviorally, the load balancer first receives requests from users. Once the load balancer has received enough requests, it adds the requests to a batch, instantiates a new server instance, and sends the batch to it. After a batch has been received, it is processed, and the results are saved where the user specified.

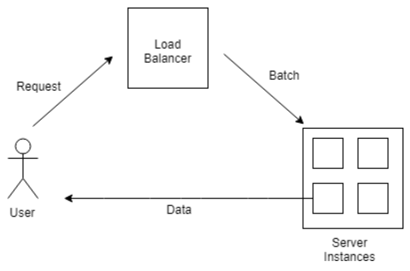

Figure 2: Behavioral overview depicting how system elements communicate.

2.1 System Model

For this scenario, we will *not* be implementing a full networked solution of different programs running across multiple computers, but rather model it as a set of functions spread across different files. The implementation will be done using three files that provide the functionality needed by each of the system components. For example, we will have a file called InstanceHost.c which contains a function called host_request_instance(). When the load balancer in our system needs to create an instance, it will simply call host_request_instance() from InstanceHost.c. It will not make any sort of network connection. The goal is simply to use C functions that fit into the overall conceptual framework given by the problem.

The network will be modeled using three files. User.c will receive input for the amount of users making requests, and the batch size. During execution, a thread will be generated to represent each user request. Each request will be passed to LoadBalancer.c. LoadBalancer.c, after receiving requests from User.c, will construct a batch (a linked list). The batch will be passed to InstanceHost.c which will remove the requests from the batch, and process it. Batches will be represented as a linked list of nodes, with nodes containing the pieces of information (id, data, result address) required by each request. Note that a job will simply consist of squaring a number.

3 Requirements [40 points]

For your submission will you create two new files: LoadBalancer.c (which includes LoadBalancer.h, and InstanceHost.h), and InstanceHost.c (which includes InstanceHost.h). You will not need to make changes to User.c, LoadBalancer.h, or InstanceHost.h. Your task is to implement the functionality as defined in LoadBalancer.h and Instance.h in their respective .c files. **As a base requirement, your program must compile and run under Xubuntu (or another variant of Ubuntu) 22.04.**

The number of requests (N) and batch size (K) will be provided as an input during execution. Each request will run on an individual thread. Requests will be received by the load balancer which will instantiate new server instances depending on batch size. The LoadBalancer will create a linked list (up to size K) as requests are made and which will be protected by a mutex to ensure only one request is added at a time. After a batch has been filled, the batch is sent to a new server instance. Once in a thread on the host, the linked list's requests are processed and the results are given as an output. Throughout the lifetime of the program, create only one LoadBalancer and one InstanceHost object. Global variables not permitted for LoadBalancer or InstanceHost.

1. **LoadBalancer.c:** A file containing a set of functions to simulate a load balancer. [21 points total]
   (a) *balancer* balancer_create(int batch_size)*: Initializes the load balancer. Takes batch size as parameter. [3 points]
   (b) *void balancer_destroy(balancer** lb):* Shuts down the load balancer. Ensures any outstanding batches have completed. [3 points]
      - If there are leftover jobs (too few to have spawned an instance host normally), then create an instance host to handle them. [5 points]

   (c) *void balancer_add_job(balancer* lb, int user_id, int data, int* data_return):* Adds a job to the load balancer. The linked list of jobs must be protected by a mutex. [6 points]
      - If enough jobs have been added to fill a batch, the load balancer will request a new instance from InstanceHost. [5 points]

2. **InstanceHost.c:** A file containing a set of functions to simulate a cloud-like server instance host. [17 points total]
   (a) *host* host_create():* Initializes the host environment. [4 points]
   (b) *void host_destroy(host** h):* Shuts down the host environment. Ensures any outstanding batches have completed. [3 points]
   (c) *void host_request_instance(host* h, struct job_node* batch):* Creates a new server instance (i.e., thread) to handle processing the items contained in a batch (i.e., a listed list of job_node). Server instances are modeled as threads that process a list of jobs. [6 points]
      - Data must be processed (the data value squared) and returned to the user (using the result pass-by-reference). [5 points]

You may add other helper functions as needed.

Sample Output

```
User #1: Wants to process data=77 and store it at 0xb2a00470.
LoadBalancer: Received new job from user #1 to process data=77 and store it at 0xb2a00470.
User #4: Wants to process data=49 and store it at 0xb2c00480.
LoadBalancer: Received new job from user #4 to process data=49 and store it at 0xb2c00480.
User #5: Wants to process data=62 and store it at 0x95f89c0.
LoadBalancer: Received new job from user #5 to process data=62 and store it at 0x95f89c0.
```

```
User #8: Wants to process data=40 and store it at 0x95f89d0.
LoadBalancer: Received new job from user #8 to process data=40 and store it at 0x95f89d0.
User #3: Wants to process data=93 and store it at 0x95f89b0.
LoadBalancer: Received new job from user #3 to process data=93 and store it at 0x95f89b0.
LoadBalancer: Received batch and spinning up new instance.
User #9: Wants to process data=72 and store it at 0xb2a004a0.
LoadBalancer: Received new job from user #9 to process data=72 and store it at 0xb2a004a0.
User #7: Wants to process data=63 and store it at 0xb2c00490.
LoadBalancer: Received new job from user #7 to process data=63 and store it at 0xb2c00490.
User #6: Wants to process data=90 and store it at 0xb2a00490.
LoadBalancer: Received new job from user #6 to process data=90 and store it at 0xb2a00490.
User #0: Wants to process data=83 and store it at 0xb2c00470.
LoadBalancer: Received new job from user #0 to process data=83 and store it at 0xb2c00470.
User #2: Wants to process data=86 and store it at 0xb2a00480.
LoadBalancer: Received new job from user #2 to process data=86 and store it at 0xb2a00480.
LoadBalancer: Received batch and spinning up new instance.
User #0: Received result from data=83 as result=6889.
User #3: Received result from data=93 as result=8649.
User #6: Received result from data=90 as result=8100.
User #1: Received result from data=77 as result=5929.
User #8: Received result from data=40 as result=1600.
User #5: Received result from data=62 as result=3844.
User #4: Received result from data=49 as result=2401.
User #7: Received result from data=63 as result=3969.
User #2: Received result from data=86 as result=7396.
User #9: Received result from data=72 as result=5184.
```

This output was generated with 10 user requests and a batch size of 5. Since the users make requests after waiting a random amount of time your output will differ. To get this output, you will need to add two lines to your lines:

```
//as first line of balancer_add_job:
printf("LoadBalancer: Received new job from user #%d to process data=%d and
store it at %p.\n", user_id, data, data_return);

//as first line of host_request_instance:
printf("LoadBalancer: Received batch and spinning up new instance.\n");
```

3.1 Basecode

For this assignment, three base files are provided: User.c, LoadBalancer.h, and InstanceHost.h. These files together make up the logic of our system.

1. **User.c:** A program to simulate multiple users simultaneously requesting work (a "job") to be carried by a load balancing server and returned to the user. A job is to compute the square of a number.
   (a) *int main():* Entry point to simulation.
   (b) *void* simulate_user_request(void* user_id):* simulates a user requesting work to be done a server.
2. **LoadBalancer.h:** A file containing definitions for a set of functions to simulate a load balancer.
   (a) *void balancer_init(int batch_size)*: Initializes the load balancer.
   (b) *void balancer_shutdown():* Shuts down the load balancer.
   (c) *void balancer_add_job(int user_id, int data, int* data_return):* Adds a job to the load balancer.
3. **InstanceHost.h:** A file containing definitions for a set of functions to simulate a cloud-like server instance host.
   (a) *void host_init():* Initializes the host environment.
   (b) *void host_shutdown():* Shuts down the host environment.
   (c) *void host_request_instance(struct job_node* batch):* Creates a new server instance to handle processing the items contained in a batch.

4 Include Files

To complete this assignment, you may find the following include files useful:

- pthread.h: Defines functionality for manipulating mutexes.
  - Useful functions
    - int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr): function initializes the mutex referenced by mutex with attributes specified by attr.
    - int pthread_mutex_lock(pthread_mutex_t *mutex): The mutex object referenced by mutex is locked by calling pthread_mutex_lock(). If the mutex is already locked, the calling thread blocks until the mutex becomes available.
    - int pthread_mutex_unlock(pthread_mutex_t *mutex): The pthread_mutex_unlock() function releases the mutex object referenced by mutex.
    - int pthread_mutex_destroy(pthread_mutex_t *mutex): The pthread_mutex_destroy() function shall destroy the mutex object referenced by mutex; the mutex object becomes, in effect, uninitialized.

5 Submission

The submission for this assignment has one part: a source code submission. The file should be attached to the homework submission link on Canvas.

Writeup: For this assignment, no write up is required.

Source Code: Please name your classes as "LastnameLoadBalancer.c", and “LastnameInstanceHost.c” (e.g., "KhanLoadBalancer.c", and “KhanInstanceHost.c”). (If you require additional .c or .h files, please check with the instructor or TA first.)