KING ABDULAZIZ UNIVERSITY

THE COLLEGE OF ENGINEERING

Operating Systems

EE463 – Spring 2023

Term Project Design Document

Name	ID
SAAD ALI SADAGAH AL JEHANI	1935151
KHALED MAMNDOUH NASSER AL DAHASI	1935129
NAWAF SAMI MUALLA Al-HARBI	1936576

Tuesday, June 6, 2023

Table of Contents

Project Definition

This project aims to develop a network application program that uses message-passing in a client server setup, shared memory, signals, multi-threading, and synchronization mechanisms to create a working web server. The basic architecture of the web server is provided, and the task is to make it more responsive by changing the way it handles the received requests. The new web server will use a thread-pool architecture and a shared scheduling list of buffers to handle the requests in a concurrent and efficient manner. To ensure the correct execution of the system, synchronization mechanisms such as mutual exclusion and semaphores will be used to manage the shared resources among the worker threads. Additionally, the web server will have a self-monitoring component to maintain a constant number of live worker threads in the thread-pool, report any abnormal conditions as they occur, handle a termination signal (Ctrl-c), and clean-up before normal termination.

The server will also have two different strategies to handle overload situations as the buffers become full, which will require careful consideration of synchronization and mutual exclusion to maintain correct operation. Overall, this project presents an opportunity to implement a complex system that involves different programming paradigms and requires careful management of shared resources to ensure correct and efficient execution.

When there are no empty buffers, the server can block, drop the new request immediately, or drop the oldest request in the queue that is not currently being processed by a thread. The default policy is to block until a buffer becomes available. The first alternative is to drop the new request immediately (DRPT), and the second alternative is to drop the oldest request in the queue (DRPH).

Project Schedule

The project schedule is divided into two parts: the first part is the development phase, and the second part is the testing phase. The development phase is further divided into the following tasks:

1. Understand the basic architecture of the provided web server.
2. Modify the web server to use a thread-pool architecture and a shared scheduling list of buffers (queue).
3. Develop the self-monitoring component (self-monitoring thread).
4. Implement two different strategies to handle overload situations.

5. Test the developed web server for its functionality.
    a. Conduct performance testing.
    b. Compare the server's performance against the original server.
    c. Evaluate the server for scalability.

The team will work together on all tasks. Each member of the team will be assigned a specific task to complete and will work collaboratively to ensure the project's successful completion. The following is the timeline of tasks and their assigned members working on them where M1 is Saad, M2 is Khaled, M3 is Nawaf, and the leader is assigned to be M2.

Table 1 Team Members' names

Name	Member
SAAD ALI SADAGAH AL JEHANI	M1
KHALED MAMNDOUH NASSER AL DAHASI	M2
NAWAF SAMI MUALLA Al-HARBI	M3

The Schedule

10/5/2023 - 13/5/2023: **(Done)**

- Discuss project scope and goals with team members (all members)
- Assign specific tasks to team members (M2)
- Begin work on Task 1: Understand the basic architecture of the provided web server:
    - Write a comprehensive project definition (M1).
    - Divide the work into tasks and assign them dates and members (M2).
    - Write the early algorithms for the project, overall and non-trivial (M2).
    - Early data structures, Classes, and methods (M2).
    - Memo of this task and report (M3)

14/5/2023: **(Done)**

- Submit Progress Report 1 detailing completed work and any challenges faced. (M2)

15/5/2023 - 18/5/2023: **(Done)**

- Begin work on Task 2: Modify the web server to use a thread-pool architecture and a shared scheduling list of buffers (queue):
    - Implementation of main thread (M1).
    - Implementation of Thread-pool for worker threads (M2).
    - Implementation of shared Queue (M3).

19/5/2023 - 22/5/2023: **(Done)**

- Continue work on Task 2: Modify the web server to use a thread-pool architecture and a shared scheduling list of buffers (queue):
    - Update report with Implementation of main thread (M1).
    - Update report with Implementation of Thread-pool for worker threads (M3).
    - Update report with Implementation of shared Queue (M3).
- Begin work on Task 3: Develop the self-monitoring component (self-monitoring thread) (M2)
- Write a new memo for current progress (M1).

23/5/2023: **(Done)**

- Submit Progress Report 2 detailing completed work and any challenges faced (M3)

24/5/2023 - 30/5/2023: **(Done)**

- Continue work on Task 3: Develop the self-monitoring component (self-monitoring thread) (M2).
- Begin work on Task 4: Implement two different strategies to handle overload situations:
    - Implement DRPT (M1).
    - Implement DRPH (M2).

31/5/2023 - 2/6/2023: **(Done)**

- Continue work on Task 4: Implement two different strategies to handle overload situations:
    - Implement DRPT (M1).
    - Implement DRPH (M2).
- Begin work on Task 5: Test the developed web server for its functionality (M1)

3/6/2023 - 5/6/2023: **(Done)**

- Continuing work on Task 5: Test the developed web server for its functionality (M1).
- Prepare final report for submission (M3).
- Prepare final Presentation for submission (M1).

6/6/2023: **(Done)**

- Submit final report after revision (M2).
- Complete any necessary cleanup or final testing after revision (M1).
- Prepare for presentation (all team).

8/6/2023: **(Not Started Yet)**

- Give presentation on completed project (all team).

The following is the Gannt Chart for our project:

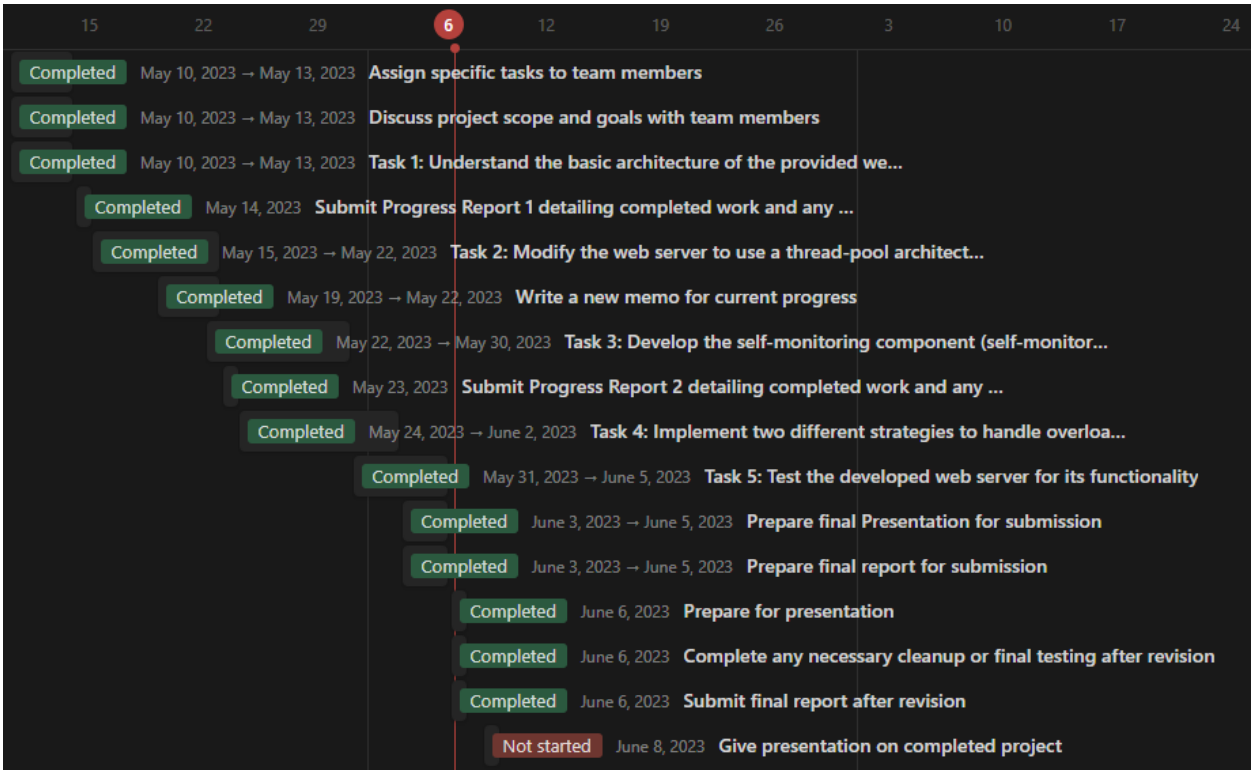

Figure 1 Project Gannt Chart

Server Algorithm

The overall server algorithm consists of the following steps:

1. Accept, validate, and use command-line parameters.

2. Create a fixed and sustained number of worker threads.
3. Continuously receive new requests and insert them into a shared scheduling list of buffers (queue).
4. A worker thread will repeatedly check the shared list for waiting requests.
5. If the list is not empty, the worker thread will remove and serve exactly one request from the list.
6. If a worker thread finds the list empty, it will block.
7. If the main thread finds the list full, it will block or use one of the overload handling policies mentioned in the project definition.
8. The self-monitoring component will maintain a constant number of live worker threads in the thread-pool, report any abnormal conditions as they occur, handle a termination signal (Ctrl-c), and clean-up before normal termination.

Non-trivial algorithms and/or formulas

Scheduling algorithm using semaphores.

The scheduling algorithm in the web server is responsible for assigning incoming requests to the worker threads. To facilitate this process, the web server utilizes a shared scheduling list of buffers, where each buffer can hold one request. The scheduling algorithm employs two semaphores, `produce` and `consume`, to manage access to the scheduling list and ensure that multiple threads do not concurrently access the same buffer.

When a new request arrives, the scheduling algorithm waits on the produce semaphore. This semaphore allows the algorithm to know if there is an available buffer in the scheduling list to accommodate the new request. Once a buffer becomes available, the algorithm inserts the new request into the buffer indicated by the tail pointer and increments the tail pointer. By using the produce semaphore, the algorithm ensures that it does not add a request to a full scheduling list.

On the other hand, when a worker thread is prepared to process a request, it waits on the `consume` semaphore. This semaphore guarantees that a buffer containing a request is available in the scheduling list for the worker thread to consume. Once a buffer becomes available, the algorithm retrieves the request from the buffer pointed to by the head pointer and increments the head pointer.

Algorithms for handling overload situations

When the web server is overloaded with requests and the scheduling list is full, the server must handle the overload situation to prevent the server from crashing or becoming unresponsive. The project definition mentions two overload handling policies:

a. Drop-newest policy: In this policy, when the scheduling list is full, the server drops the newest request and continues to serve the requests that are already in the list. The algorithm for this policy is straightforward. When the scheduling list is full, the server discards the request that arrived last and does not insert it into the scheduling list.
b. Drop-oldest policy: In this policy, when the scheduling list is full, the server drops the oldest request and inserts the newest request into the list. The algorithm for this policy is more complex. When the scheduling list is full, the server removes the request from the head of the list (the oldest request) and discards it. It then inserts the newest request into the tail of the list.

Required Data Structures

Scheduling List: A data structure that holds incoming requests and manages the order in which they are processed by the worker threads. This list can be implemented as a FIFO queue using a linked list.

QueueElement: A structure to pass different variables and information to the shared List.

Classes and Methods

QueueElement:

Author: Nawaf Alharbi

Date: 2023/5/21

The `QueueElement` class represents an element that can be enqueued in the queue. It contains an integer count, and a socket. It is used to encapsulate a request's socket along with its count.

Methods:

1. getSocket(); returns the socket.
2. getCount(); returns the count.

Interface:

The `QueueElement` class provides methods to retrieve the encapsulated `ServeWebRequest`, the count, and the socket.

Inputs:

1. `int count`: The count value to be encapsulated.
2. `Socket connect`: The socket to be encapsulated.

Outputs:

1. `int`: The encapsulated count value.
2. `Socket`: The encapsulated socket.

Preconditions: None.

Postconditions: None.

Queue:

Author: Nawaf Alharbi

Date: 2023/5/20

The `Queue` class implements a queue data structure using a linked list. It allows elements of type `QueueElement` to be enqueued at the rear and dequeued from the front. The queue has a maximum size defined during initialization and uses a semaphore to handle concurrency.

Methods:

- enQueue(QueueElement item): Enqueues an item at the rear of the queue.
- deQueue(): Dequeues an item from the front of the queue.
- size(): Returns the current size of the queue.
- isEmpty(): Checks if the queue is empty.
- displayQueue(): Displays the elements of the queue and their positions, this is for debugging.
- clear(): Clears the queue by setting the front, rear, and length to null or zero.

Interface:

The `Queue` class provides methods to interact with the queue, such as enqueuing and dequeuing items, checking the size and emptiness of the queue, displaying the queue contents, and clearing the queue.

Inputs:

QueueElement item (enQueue): The item to be enqueued in the queue.

Outputs:

QueueElement (deQueue): The item dequeued from the front of the queue.

Preconditions:

1. The maximum size of the queue is defined during initialization.
2. The queue must have been initialized before using its methods.

Postconditions:

1. The item is enqueued at the rear of the queue (enQueue).
2. The item is dequeued from the front of the queue (deQueue).
3. The size of the queue is updated accordingly (enQueue, deQueue).

The queue is cleared, and all elements and references are set to null or zero (clear).

WorkerThreads:

Author: Khaled Dahhasi

Date: 2023/5/21

The `WorkerThreads` class represents a worker thread that handles incoming requests from a shared scheduling list of buffers. It dequeues requests from the scheduling list, performs the necessary actions, and serves the web request using an instance of `socket` and count. The worker thread is controlled by a semaphore and can be stopped or checked for its status.

Methods:

- run(): Implements the main logic of the worker thread, continuously dequeuing requests from the scheduling list and serving the web requests.
- stop(): Stops the execution of the worker thread.

Interface:

The `WorkerThreads` class provides methods to control and check the status of the worker thread, such as starting, stopping, and checking if it is alive.

Inputs: None.

Outputs: None.

Preconditions:

1. The worker thread must be started using the `run()` method.

Postconditions:

1. The worker thread dequeues requests from the scheduling list and serves web requests until it is stopped.

WebServer:

Author: Saad Al Jehani

Date: 2023/5/20

The `WebServer` class represents the main thread of execution in the web server. It initializes the shared scheduling list (queue) and semaphore, creates, and starts multiple worker threads, and accepts incoming socket connections. It also enqueues web requests into the scheduling list for the worker threads to handle.

Methods:

- `main()`: The entry point of the web server application. It initializes the server settings, creates worker threads, starts the monitor thread, and accepts incoming socket connections.
- `getWorkerThreads()`: Returns the list of worker threads created by the main thread.

Interface:

The `WebServer` class provides methods to start and handle incoming socket connections for the web server. It also provides a method to retrieve the list of worker threads.

Inputs: None.

Outputs: None.

Preconditions:

1. A shared scheduling list (queue) and a semaphore must be provided during initialization.

Postconditions:

1. The main thread accepts incoming socket connections and enqueues web requests into the scheduling list until the server is terminated.

MonitorThread:

Author: Khaled Dahhasi

Date: 2023/5/22

This class represents the monitor thread of the web server responsible for maintaining a constant number of live worker threads in the thread-pool, reporting any abnormal conditions as they occur in the server, handling a termination signal (Ctrl-c) when it comes from the keyboard, and cleaning up before normal termination.

Methods:

- `run()`: Implements the main logic of the monitor thread, continuously collecting and logging server statistics.
- `checkWorkerThreads()`: Checks the status of worker threads, removes any terminated threads, and starts new worker threads if needed.
- `startWorkerThread()`: Starts a new worker thread.
- `terminate()`: Signals the monitor thread to terminate.
- `cleanup()`: Performs cleanup tasks before the normal termination of the monitor thread.

Interface:

> The `MonitorThread` class provides methods to control and check the status of the monitor thread, such as starting, stopping, and checking if it is alive. It also includes methods to check worker thread status, start new worker threads, and perform cleanup tasks.

Inputs: None.

Outputs: None.

Preconditions:

1. The monitor thread must be started using the `run()` method.

Postconditions:

1. The monitor thread continuously collects and logs server statistics until it is stopped.

Discussion of Classes and methods

Overall, the collaboration between the `Queue`, `WorkerThreads`, `WebServer`, `MonitorThread`, and `QueueElement` classes forms the foundation of a robust and scalable multithreaded web server. The `Queue` class provides a thread-safe data structure for managing the scheduling of incoming web requests, ensuring proper synchronization and concurrent access. The `WorkerThreads` class handles the actual processing and serving of web requests, leveraging the shared scheduling list provided by the `Queue` class. The `WebServer` class orchestrates the server's operations, accepting incoming socket connections and enqueuing web requests into the scheduling list. The MonitorThread class is in charge of making sure that there's a set number of worker threads working at all times. It checks the status of the worker threads and the list itself, reporting any issues that come up. It also takes care of cleaning up when a termination signal is

sent from the keyboard. The `QueueElement` class encapsulates sockets for web requests and their count within the scheduling list, facilitating organized storage and retrieval. By working together, these classes enable the server to efficiently handle multiple concurrent web requests. The main thread accepts incoming connections and enqueues requests, while the worker threads dequeue requests from the scheduling list and processes them independently. This parallel processing enhances the server's responsiveness and throughput.

Conclusion:

In conclusion, this project aims to develop a network application program that creates a responsive web server using various programming paradigms and synchronization mechanisms. The project focuses on modifying the provided web server architecture to use a thread-pool and a shared scheduling list of buffers, allowing concurrent and efficient handling of requests. Mutual exclusion and semaphores are utilized to manage shared resources among worker threads, ensuring correct execution.

The project schedule is divided into two phases: the development phase and the testing phase. The development phase includes understanding the provided architecture, implementing the thread-pool and shared scheduling list, developing the self-monitoring component, and implementing two overload handling strategies. The testing phase involves evaluating the server's functionality, performance, and scalability.

The server algorithm follows a set of steps, including accepting and validating command-line parameters, creating worker threads, continuously receiving and inserting requests into the scheduling list, and serving requests by worker threads. Overload situations are handled using drop-newest or drop-oldest policies.

The project requires specific data structures such as the scheduling list implemented as a FIFO queue using a linked list. Additionally, classes and methods are defined for QueueElement, Queue, WorkerThreads, MonitorThread, and WebServer, each serving specific functionalities in the server implementation.

Overall, this project provides an opportunity to implement a complex system that combines multiple programming paradigms and synchronization mechanisms to create an efficient and

responsive web server. The team has followed a well-defined schedule and assigned tasks to team members for successful project completion.