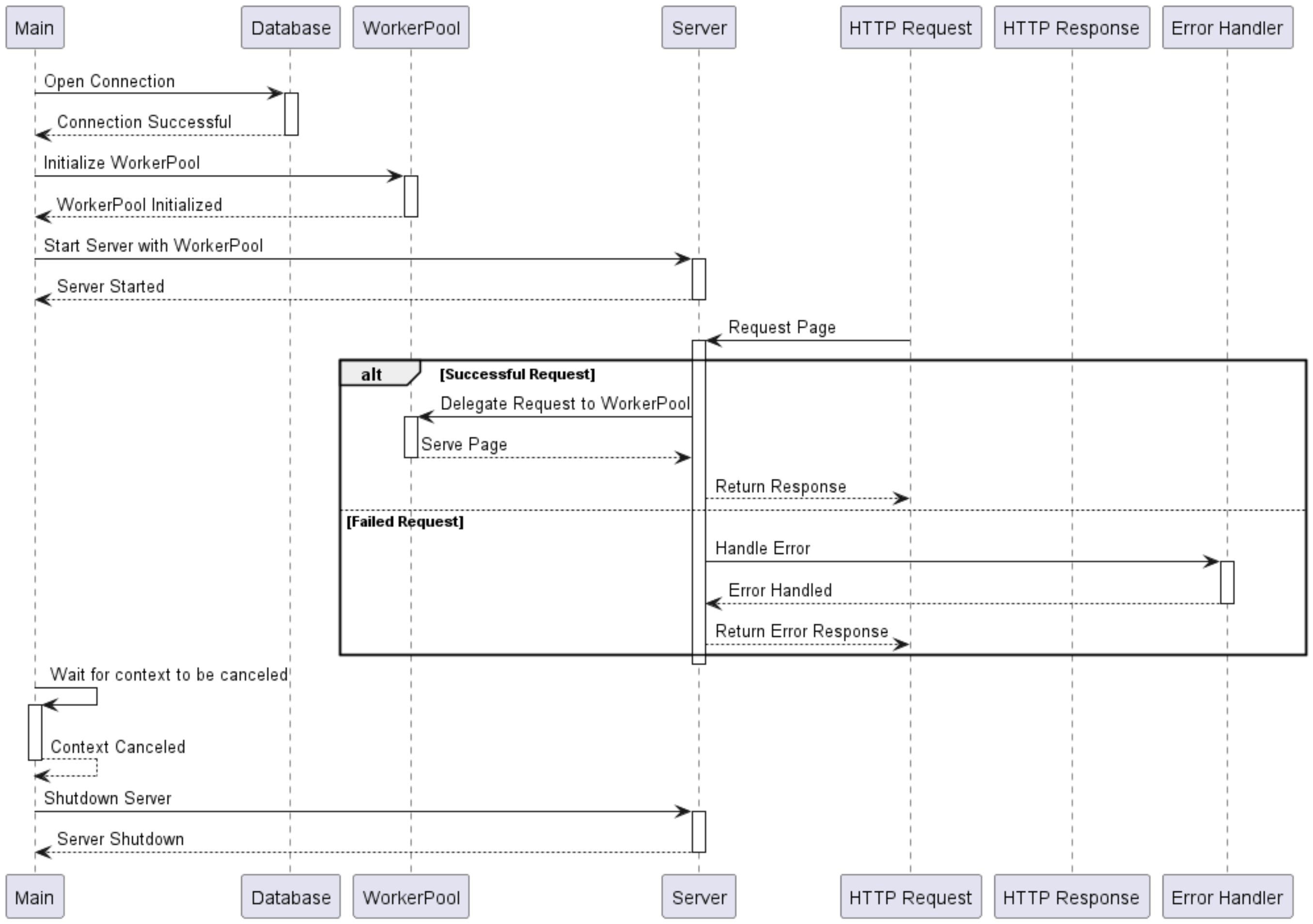
ECE 1170 Final Report

Part A: Threads & Processes



The diagram in `ThreadsProcess.puml` represents the process of creating and returning threads to the worker pool in the context of handling HTTP requests. The code that corresponds to this process is primarily located in the `server.go` file.

Opening a Database Connection: The main function opens a connection to the database. This is represented in the diagram by the interaction between "Main" and "Database".

```
// Connect to the database
cfg := mysql.Config{
        User:   os.Getenv("DBUSER"),
        Passwd: os.Getenv("DBPASS"),
        Net:    "tcp",
        Addr:   "127.0.0.1:3306",
        DBName: "formoutput",
}
```

```
// Get a database handle.
db, err = sql.Open("mysql", cfg.FormatDSN())
if err != nil {
        log.Fatal(err)
}

// Check if the connection is successful
pingErr := db.Ping()
if pingErr != nil {
        log.Fatal(pingErr)
}
fmt.Println("Connected to the Database!")
```

Initializing the Worker Pool: The main function initializes a worker pool with a specified maximum number of workers. This is represented in the diagram by the interaction between "Main" and "WorkerPool".

```
// Initialize the worker pool with a maximum of 10 workers
pool := NewWorkerPool(config.ThreadPoolSize)
```

Starting the Server with the Worker Pool: The main function starts the server, passing the worker pool as the handler for incoming HTTP requests. This is represented in the diagram by the interaction between "Main" and "Server".

```
// Start the server using the worker pool
server := &http.Server{
        Addr: config.ServerPort,
        Handler: http.HandlerFunc(func(w http.ResponseWriter, r *http.Request)
{
                // Serve the request within the context with the worker pool
                pool.ServeHTTPWithContext(ctx, w, r, http.DefaultServeMux)
        }),
}
```

Handling HTTP Requests: When an HTTP request is received, the server delegates the request to the worker pool. If a worker is available, it serves the request and then is returned to the pool. If a worker is not available, the request is not served. This is represented in the diagram by the interactions between "HTTP Request", "Server", "WorkerPool", and "HTTP Response".

```
// ServeHTTPWithContext serves HTTP requests within the given context
func (wp *WorkerPool) ServeHTTPWithContext(ctx context.Context, w
http.ResponseWriter, r *http.Request, handler http.Handler) {
```

```go
		// Try to acquire a worker from the pool
		select {
		case wp.workerQueue <- struct{}{}:
			// A worker is available, handle the request
			fmt.Println("Worker locked.")
			// Describe the request
			fmt.Printf("Request: %s %s\n", r.Method, r.URL.Path)
			if r.URL.Path == "/image" {
				// Serve the image
				imageHandler(w, r)
			} else if r.URL.Path == "/" {
				// Serve index if no path is provided
				r.URL.Path = "/index.html"
				handler.ServeHTTP(w, r)
			} else {
				// Serve other requests
				handler.ServeHTTP(w, r)
			}
			fmt.Println("Worker released.")
			// Release the worker back to the pool
			<-wp.workerQueue
		case <-ctx.Done():
			// Context is done, return without serving the request
			return
		}
}
```

Shutting Down the Server: When the context is canceled, the main function shuts down the server. This is represented in the diagram by the interaction between "Main" and "Server".

```go
// Wait for the context to be canceled (after the timeout)
<-ctx.Done()

// Shutdown the server gracefully
fmt.Println("Shutting down the server...")
if err := server.Shutdown(context.Background()); err != nil {
	// Handle error
	fmt.Println("Error shutting down the server:", err)
}
```

In summary, the `server.go` file is the main entry point of the application. It starts by defining several types and functions that are used throughout the application.

The `Config` struct is used to hold the configuration of the server, which is read from a JSON file. The `CustomFileServer` struct is a custom file server that serves files with a custom buffer size. It has a `ServeHTTP` method that serves HTTP requests. The `WorkerPool` struct is used to manage a pool of goroutines that can serve HTTP requests concurrently. It has a `ServeHTTPWithContext` method that serves HTTP requests within a given context.

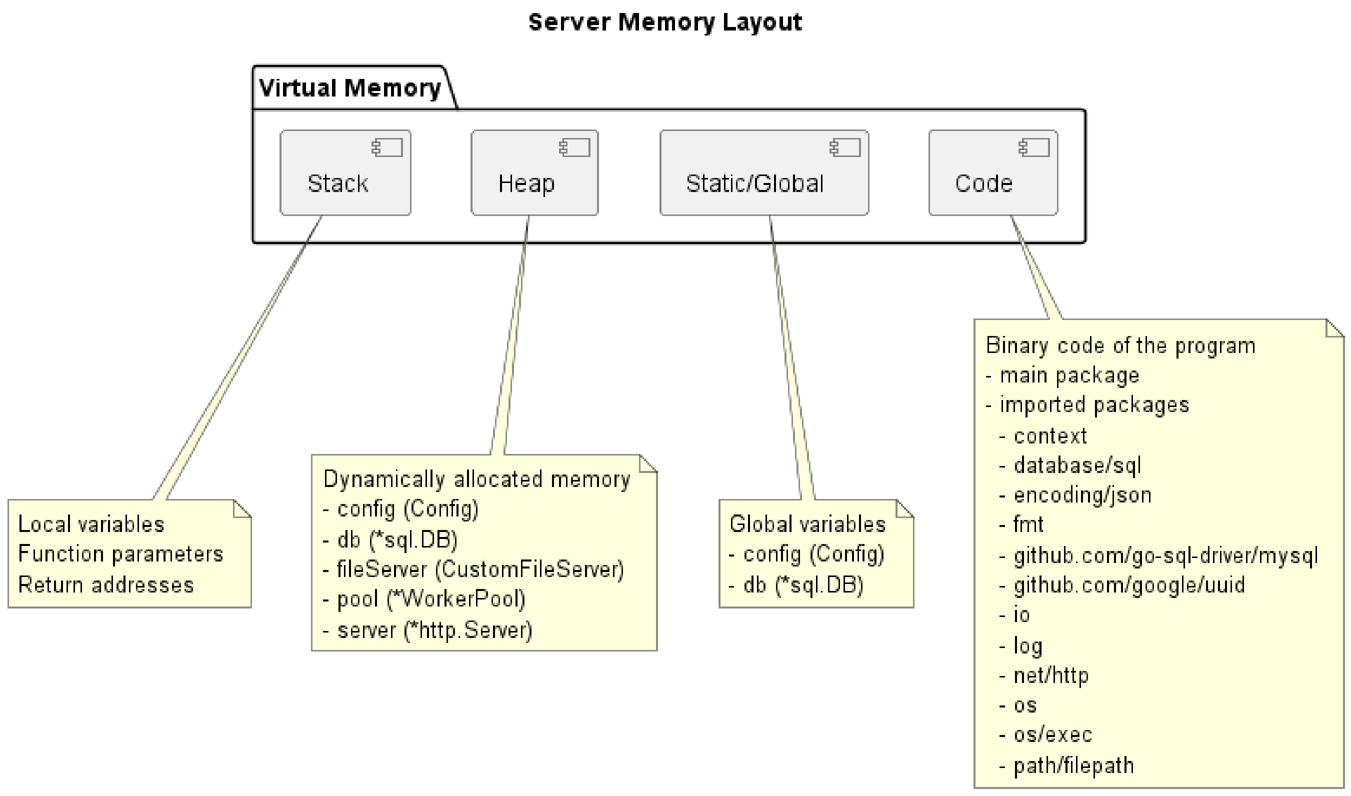
The `main` function is the entry point of the application. It starts by reading the configuration file and parsing it into the `Config` struct. It then starts a `CustomFileServer` and registers several handler functions for different paths.

The `main` function then opens a connection to the database and initializes a `WorkerPool` with a maximum number of workers specified in the configuration. The `main` function then starts the server with the `WorkerPool` as the handler for incoming HTTP requests.

When an HTTP request is received, the server delegates the request to the `WorkerPool`. If a worker is available, it serves the request and then is returned to the pool. If a worker is not available, the request is not served.

Finally, the `main` function waits for the context to be canceled and then shuts down the server gracefully.

Part B: Memory Layout



In the Go memory model describes how Go programs read from and write to shared memory. It is a specification that defines the behavior of concurrent Go programs, and it is crucial for understanding how to write correct concurrent code in Go. In Go, concurrency is achieved through goroutines, which are lightweight threads managed by the Go runtime. Each goroutine has its own stack, which is used for storing local variables, function parameters, and return addresses.

The Stack is used for storing local variables, function parameters, and return addresses. Each goroutine in Go has its own stack, which grows and shrinks as needed. The stack is very efficient for storing and retrieving data because it follows the Last-In-First-Out (LIFO) principle. However, the data stored in the stack is only available within its own scope, which means the data is not accessible outside the function it's declared in. Once the function execution is completed, all the variables in the stack will be wiped out.

The Heap is used for storing dynamically allocated variables. Unlike the stack, data in the heap can be accessed globally, and it remains in memory until either the program ends or the memory is explicitly freed (in Go, this is done by the garbage collector). This makes it suitable for data that needs to persist outside of the function it was created in. However, allocating and deallocating memory from the heap is generally slower than stack operations.

In `server.go`, the `config`, `db`, `fileServer`, `pool`, and `server` variables are examples of data that would be stored in the heap. These are complex data structures that are created dynamically and need to be accessed globally across different functions.

In the Go memory model, synchronization mechanisms like locks and channels play a crucial role in coordinating the execution of goroutines and ensuring the consistency of shared data.

Locks, provided by the `sync` package in Go, are used to protect shared data from concurrent access, thus preventing data races. When a goroutine acquires a lock (using `Lock()` method), no other goroutine can access the locked data until the lock is released (using `Unlock()` method). This ensures that only one goroutine can access the shared data at a time, providing a way to enforce mutual exclusion.

In `server.go`, the `WorkerPool` struct uses a channel as a semaphore to limit the number of concurrent goroutines. This is a form of lock, where acquiring a worker from the pool is equivalent to acquiring a lock, and releasing a worker back to the pool is equivalent to releasing a lock.

Channels in Go are used for communication between goroutines. They provide a way for one goroutine to send data to another, and this communication is synchronized. When a goroutine sends data on a channel, it will block until another goroutine receives the data from the channel. Similarly, when a goroutine tries to receive data from a channel, it will block until another goroutine sends data on that channel.

In addition to data transfer, channels can also be used for signaling between goroutines. For example, closing a channel can be used as a signal to indicate that no more data will be sent on the channel.

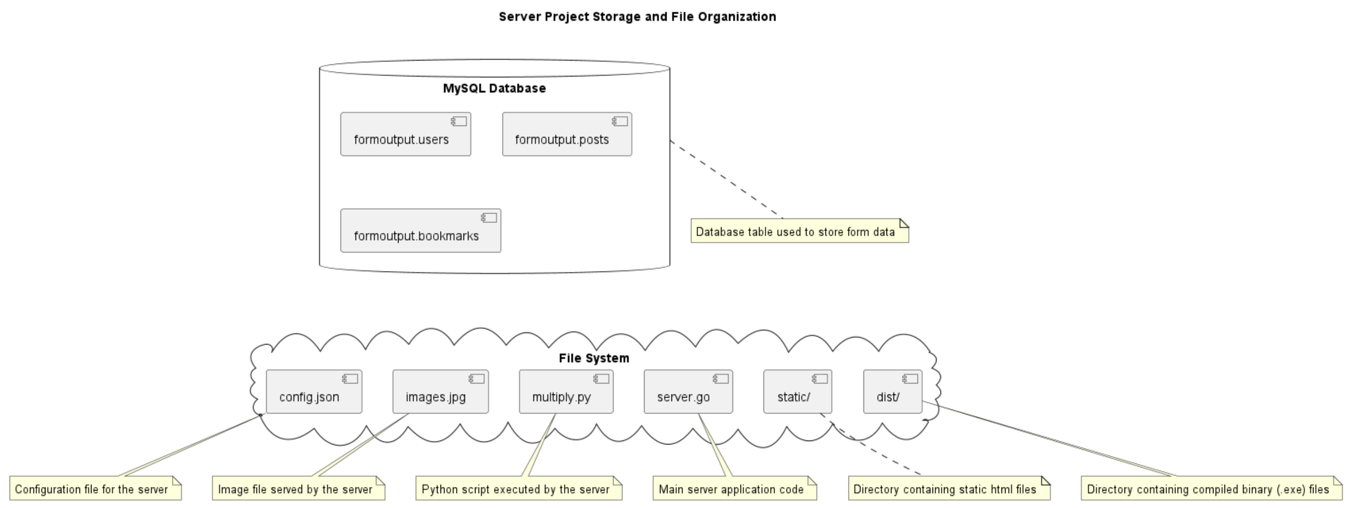
In `server.go`, the `WorkerPool` struct uses a channel (`workerQueue`) to manage the pool of workers. The `ServeHTTPWithContext` method tries to send a value on the `workerQueue` channel to acquire a worker from the pool. If a worker is available (i.e., the channel is not full), the method proceeds to handle the request. If not, it blocks until a worker becomes available or the context is done.

These synchronization mechanisms are essential for writing correct concurrent programs in Go. They allow goroutines to safely and predictably share data, coordinate execution, and communicate with each other.

The Static/Global region is used for storing global variables and static data. These are variables that are declared outside any function and are accessible from any part of the code. They are initialized only once, at the start of the program, and they remain in memory until the program ends. In the provided `server.go` code, the `config` and `db` variables are examples of global variables that would be stored in the Static/Global region.

The Code region is used for storing the binary code of the program. This includes the compiled instructions of the program and the imported packages. The code is read-only and is loaded into memory when the program starts. In the provided `server.go` code, the main package and the imported packages (such as `context`, `database/sql`, `encoding/json`, `fmt`, `github.com/go-sql-driver/mysql`, `github.com/google/uuid`, `io`, `log`, `net/http`, `os`, `os/exec`, `path/filepath`) would be stored in the Code region.

Part C: Storage and File Organization


Server Project Storage and File Organization
MySQL Database
formoutput.users
formoutput.posts
formoutput.bookmarks
Database table used to store form data
File System
config.json
images.jpg
multiply.py
server.go
static/
dist/
Configuration file for the server
Image file served by the server
Python script executed by the server
Main server application code
Directory containing static html files
Directory containing compiled binary (.exe) files

`server.go` interacts with a MySQL database and serves files from a file system. It also executes Python scripts and serves images. The server uses a configuration file `config.json` to set up various parameters such as the root directory, the default file to serve, the server port, the buffer size, and the number of buffers. The server serves static HTML files from a directory specified in the configuration file. It also contains a directory `dist/` that holds compiled binary (.exe) files.

The server interacts with a MySQL database that contains tables `formoutput.users`, `formoutput.posts`, and `formoutput.bookmarks`. The server inserts data into these tables and queries data from them.

The server uses buffering when serving files. The buffer size is specified in the configuration file. The server reads files in chunks of this size and writes these chunks to the HTTP response. This buffering is done by the application itself.

The server also uses a worker pool to handle HTTP requests. The number of workers in the pool is specified in the configuration file. This is a form of buffering done by the application, where incoming requests are buffered in a queue until a worker is available to handle them.

The server executes a Python script `multiply.py` and serves an image file `images.jpg`. These files are part of the file system that the server interacts with. The server uses the `os`, `io`, and `http` packages from the Go standard library, and the `github.com/go-sql-driver/mysql` package to interact with the MySQL database.