Homework 2: Dispatcher/Worker model with Linux pthreads

Nizar Khalaila - ID: 212683932, Alisa Gumerova - ID: 956711378

This program implements the dispatcher/worker model in Linux using threads to offload work. The dispatcher spawns a configurable number of worker threads and processes commands specified in a command file. The program reads commands from a command file and parses and executes them using multiple threads.

Components of the code

Constants and Definitions:

- NUM_THREADS: Represents the maximum number of worker threads.
- NUM_COUNTERS: Represents the maximum number of counter files.
- MAX_LINE: Maximum length of a line in the command file.
- MAX_COMMANDS: Maximum number of commands in a job.
- MAX_COMMAND_LENGTHZ: Maximum length of a command string.
- NUM_COMMANDS: Number of supported commands.
- COMMAND DEFINITIONS: Constants representing command indices for different operations.

```
#define NUM_THREADS 4096
#define NUM_COUNTERS 100
#define MAX_LINE 1024
#define MAX_COMMANDS 128
#define MAX_COMMAND_LENGTHZ 18
#define NUM_COMMANDS 6

// COMMAND DEFINITIONS
#define MSLEEP 0
#define INCREMENT 1
#define DECREMENT 2
#define DISPATCHER_WAIT 3
#define DISPATCHER_MSLEEP 4
#define REPEAT 5
```

Structures:

- Job: Represents a job with an array of commands and arguments, job index, length, and start time.

```
typedef struct Job{
  int commands[MAX_COMMANDS];//command indices
  int args[MAX_COMMANDS];//command arguments
  int index;//job line index
  int length;//length of commands in a job
  clock_t startTime;//starTime of job for turnaround calculation
}Job;
```

- Thread: Represents a worker thread with a thread ID, working status, and worker number. It is used to keep track of the progress of the thread and identify it easily.

```
typedef struct Thread{
  pthread_t threadId;
  int bWorking;
  int workerNumber;
}Thread;
```

- funcArgument: Structure used to pass arguments to the thread function to end the thread and process the job throughout.

```
typedef struct funcArgument{
  Thread* thread;
  Job* job;
}funcArgument;
```

Function Declarations:

- parseJob: Parses a line of commands and arguments and stores them in a Job structure.
- loadLines: Reads the command file and stores each line in a buffer.
- loadJobs: Parses each job using parseJob and stores them in the jobs array.
- executeJobs: Executes the jobs using threading and handles dispatcher calls on the main thread.
- getNumLines: Returns the number of lines in the command file.
- printStats: Creates a stats.txt file to store program runtime calculations.
- initCounterFiles: Creates counter files and initializes them with 0.
- incrementToFile: Increments a counter file by a given value.
- startThreadLog: Creates a log file for a thread and writes a start log line.
- endThreadLog: Writes an end log line to a thread's log file.
- threadFunc: The main function executed by worker threads to process commands.
- getWorker: Returns the index of an available worker thread.
- startThread: Sets up the argument to be passed to the thread function for a specific job.
- waitForThreads: Waits until all worker threads have finished.

```
void parseJob(char line[], Job* job);
void loadLines();
void loadJobs();
void executeJobs();
int getNumLines();
void printStats();
void initCounterFiles();
void incrementToFile(int increment, int counterFileIndex);
void startThreadLog(FILE* fp,const int workerNumber, const int jobIndex);
void endThreadLog(FILE* fp, const int workerNumber, const int jobIndex);
void* threadFunc(void* ptrVoidArgs);
int getWorker();
void startThread(int workerNumber,Job* job);
void waitForThreads();
```

Global Variables:

- commandList: Array of command strings used for indexing commands in parseJob.
- currentTime: Stores the current time.
- filePath: Path of the command file.

- numLines: Number of lines in the command file.
- numThreads: Number of worker threads.
- numCounters: Number of counter files.
- bLog: Flag indicating whether logging is enabled.
- threads: Array of Thread structures representing worker threads.
- jobs: Array of Job structures representing the jobs to be executed.
- lines: Array of strings storing the lines read from the command file.

```
char commandList[NUM_COMMANDS][MAX_COMMAND_LENGTHZ]={"msleep", "increment", "decrement","repeat","dispatcher_msleep", "dispatcher_wait
clock_t currentTime;
char* filePath;
int numLines = 0;
int numThreads, numCounters, bLog;
Thread* threads;
Job* jobs;
char** lines;
```

Functions

- `parseJob(char line[], Job* job)` - This function takes a line of commands as a string input and a pointer to a `Job` struct and parses the commands and arguments from the input string. Then, it stores the indices of the commands and their arguments in the `commands` and `args` arrays of the `Job` struct, respectively. The `index` and `length` fields of the `Job` struct are also set in this function.
- `loadLines()` - This function reads the command file, specified by the `filePath` global variable, and stores each line in a dynamically allocated array of strings named `lines`.
- `loadJobs()` - This function calls the `parseJob()` function for each line of commands stored in the `lines` array. It runs through the global array jobs which is an array of `Job` structs, and populates each struct with the parsed commands and arguments.
- `executeJobs()` - This function is responsible for executing the jobs using threads. It first finds an available thread worker index. Then it starts each job on the available worker threads using the `startThread()`**. F**or dispatcher calls, it uses the main thread to handle the `DISPATCHER_WAIT` and `DISPATCHER_MSLEEP` commands.
- `getNumLines()` - This function returns the number of lines in the command file.
- `printStats()` - This function writes the statistics of the program to a file named `stats.txt`. These statistics are attained by calculating the runtime of the program, and the average turnaround time for each job. It also calculates the min, the max, and the sum.
- `initCounterFiles()` - This function initializes the counter files by creating empty files named `counter_i.txt` where `i` is the index of the counter.
- `incrementToFile(int increment, int counterFileIndex)` - This function increments the counter file specified by `counterFileIndex` by `increment`.
- `startThreadLog(FILE* fp,const int workerNumber, const int jobIndex)` - This function writes the start time of a job to the log file specified by `fp`. It takes the worker number and job index as input.
- `endThreadLog(FILE* fp, const int workerNumber, const int jobIndex)` - This function writes the end time of a job to the log file specified by `fp`. It takes the worker number and job index as input.
- `threadFunc(void* ptrVoidArgs)` - This function is the main function executed by each worker thread. It takes a pointer to a `funcArgument` struct as input. The `threadFunc()` function executes the commands in the job, by accessing the `commands` and `args` arrays of the `Job` struct passed to it. It writes the start time and end time of the job to the log file and increments the corresponding counters using the `startThreadLog()`, `endThreadLog()`, and `incrementToFile()` functions.
- `getWorker()` - This function returns the index of the first available worker thread.
- `startThread(int workerNumber,Job* job)` - This function starts a new worker thread with the specified `workerNumber` and `job`.
- `waitForThreads()` - This function waits for all worker threads to complete.

The flow of the program

The program's execution flow can be summarized as follows:

1. The main function reads command line arguments, initializes variables, and calls loadLines to read the command file.
2. The loadLines function parses the command file line by line and stores each line in the lines array.
3. The loadJobs function parses each line using parseJob and stores the resulting jobs in the jobs array.
4. The executeJobs function is called to execute the jobs using threads. It also handles dispatcher calls on the main thread.
5. The threadFunc function is the entry point for worker threads. It executes the commands of a job, such as sleeping, incrementing/decrementing counters, or repeating commands.
6. The program waits for all worker threads to finish using the waitForThreads function.
7. Finally, statistics are printed by calling the printStats function.