Project 2 Documentation

John Carlo Sumabat

The following program is a simple threaded implementation of a file server in C. The implementation level is Level 4. The entirety of the program is enclosed in `file_server.c`. You may access the **documentation video** by opening this link.

1 Introduction

1.1 Implementation Overview

This section provides a summary of the mechanism with which the program works. Future sections will elaborate on the aspects discussed here.

The system mainly relies on two arrays to work: one that keeps track of "active" files (named `files`), and a corresponding array that keeps data on each entry in `files` (named `jobs`). Concurrency is achieved via a thread pool (an array named `tid`) and a corresponding array that keeps track of which threads are active (named `threads`).

At any point during the progam's execution, at most `N` files may receive operations and at most `M` threads may be active at one time, where `N` and `M` are constants that can be changed prior to compilation. These two constants reflect the sizes of `files` and `tid`, respectively.

User input is parsed by the master thread. Upon receiving a command, it performs various checks to determine the correct data to provide an available worker thread from `tid[]` (elaborated in Section 2.7). This data is stored in a `struct Jobdata` whose pointer is passed as a parameter to `pthread_create()` which dispatches an available worker thread from the thread pool. The worker thread performs its assigned operation, allowing the master thread to immediately receive input again. Since the execution of an operation is done on a different thread, the master thread can continuously receive requests.

One such check that the master thread performs is identifying whether the file specified by the user-supplied filepath is "active". A file is deemed *active* if at least one worker thread will perform a read or write operation on it. If a file is active, then its filepath is stored in `files` and it has its own lock independent of other files. An active file has each of its worker threads assigned a job ID. A thread's job ID is determined by their order of arrival, such that they work on a file in a "First In, First Out" (FIFO) manner. That is, a thread can only do its assigned operation if its job ID is equal to the current job that the file is accepting. This makes it so that sequential reads or writes to the same file happen in the order that they are requested.

Such enforcement of order is made possible by a file's corresponding `struct TJob` which records an active file's activity (elaborated in Sections 2.2 and 2.6). The pointer to this struct is located in the `jobs` array, at the same index where a file's pointer is located in the `files` array.

Whereas multiple requests to the same file are executed sequentially, multiple requests to different files are handled concurrently. This is because any two files with different filepaths are tracked via different indices in the `files` array. Since each active file has its own lock, locking one file does not interrupt operations on other files. As such, operations on different files happen simultaneously (but no order is enforced).

Deadlocks and race conditions are obstacles that are present with this implementation. Details on how they are prevented are found throughout future sections.

1.2 Rationale

The goal of this implementation is to allow concurrent operations on multiple files. For ease of development, dynamic management of jobs via nodes and linked lists was not pursued. Rather, a static approach was taken via arrays whose size can be tweaked based on anticipated workload. By allowing each active file to occupy a single space in `files` (and therefore `jobs`), the problem boils down to properly managing locks and job logic.

2 Implementation Details

2.1 Macros

Three constants are defined (Code Block 1). `N` and `M` are the maximum sizes of the `files` and `tid` arrays, respectively, while `MAXLEN` stands for the maximum length of any user input.

Code Block 1 User-defined constants

```
#define N 500           // max number of files that can be used at a time
#define M 1000          // max number of threads that can run at a time
#define MAXLEN 120      // max length of any file server command
```

2.2 Structs

Two structs are defined. The first one is `struct TJob` (Code Block 2). As mentioned in Section 1.1, this holds data about each active file. The `workers` member records the number of threads pending execution. Once this value becomes zero, the `files` and corresponding `jobs` entries can be safely cleared. The `count` member increments every time a new worker thread wishes to work on a file. The resulting value is assigned to the worker thread's `struct Jobdata` as its `job_ID`. The `curr_job` member indicates the `job_ID` of the worker thread whose operation it is currently accepting.

The struct also has a condition variable and two mutexes. The condition variable `done` is broadcast every time a worker thread finishes an operation to wake up other waiting threads. The mutex `wlock` is used when updating the `worker` count, while the mutex `global_lock` is used whenever a worker thread actually performs its operation. Both of these mutexes are required to prevent race conditions since more than one thread may want to interact with a file's `struct TJob` at the same time.

The second struct is `struct Jobdata` (Code Block 3). This holds the data that each worker thread uses to work on a job, namely:

- `cmdidx`: the assigned command (i.e., read, write, or empty)
- `fileidx`: the index of the assigned file in the `files` array
- `job_ID`: its position in a file's "queue" of workers
- `worker_ID`: its index in the `tid` array
- `string`: information to write to the file, if any

Code Block 2 `struct TJob` definition

```c
typedef struct TJob {
  int workers;
  int count;
  int curr_job;
  pthread_cond_t done;          // condition variable for finished jobs
  pthread_mutex_t wlock;        // lock for updating worker count
  pthread_mutex_t global_lock; // lock for reading/writing
} tjob;
```

Code Block 3 `Struct Jobdata` definition

```c
typedef struct Jobdata {
  int cmdidx;
  int fileidx;
  int job_ID;
  int worker_ID;
  char string[51];
} jobdata;
```

2.3 Global Variables

Code Block 4 shows the global variables used in the program. Note that the first four arrays are the arrays mentioned in Section 1.1. The three file pointers are used for the log files `commands.txt`, `read.txt`, and `empty.txt`.

The mutex array `tlock` provides a lock for every entry in `files` and `jobs`, while `qlock` provides a single lock for the thread pool. One mutex is required per entry in `files` to enable concurrent operations (since having only one lock would cause blocking, defeating the purpose of threads). On the other hand, the act of dispatching and exiting threads does not need to be concurrent (only thread-safe) so one lock is sufficient. Lastly, `rlock` and `elock` are mutexes for `read.txt` and `empty.txt`, respectively. Since more than one thread may want to write to the said log files at the same time, mutexes are necessary to prevent write errors.

Next, the `cmds` array is used to help check the validity of user input. The `space` array is used as a parameter to the `strtok_r()` function to tokenize user input. Lastly, `testmode` is a flag that allows the program to print test statements when set to `1`.

2.4 Test Functions

In order to verify the correctness of the progam, several test functions were created.

- The function `pfiles()` (Code Block 5) prints all active files. An entry in `files` has an active file if the content at an index `i` is not `" "`.
- The function `pft()` (Code Block 6) prints all active worker threads. An entry in `threads` has an active worker if the content at an index `i` equals `1`.
- Every time a worker thread is dispatched, `pwork()` (Code Block 7) may be called to print the members from its corresponding `struct Jobdata`.
- Lastly, `pjob()` (Code Block 8) prints the members of a file's corresponding `struct TJob` given the file's index in the `files` array.

These functions are called via `test_print()` (Code Block 9) after every user request if the

Code Block 4 Global variables

```
char files[N][MAXLEN];          // keeps track of currently active jobs
tjob *jobs[N] = { 0 };          // contains data of jobs
pthread_t tid[M];               // worker threads
int threads[M] = { 0 };         // keeps track of currently working threads
FILE *cmdfile;                  // log file for commands
FILE *readfile;                 // log file for reads
FILE *emptyfile;                // log file for empty
pthread_mutex_t tlock[N];       // lock for accessing jobs
pthread_mutex_t qlock;          // lock for accessing working threads
pthread_mutex_t rlock;          // lock for read.txt
pthread_mutex_t elock;          // lock for empty.txt

char cmds[][MAXLEN] = { "read", "write", "empty", };
char space[2] = " ";
int test_mode;
```

global variable `test_mode` equals `1`. It takes in the index of a file in the `files` array and a thread's `struct Jobdata` as parameters.

Code Block 5 `pfiles` function

```
void pfiles() {
  for ( int i = 0; i < N; i++ ) {
    if ( strcmp( files[i], " " ) != 0 ) {
      printf( "files[%d]: %s\n", i, files[i] );
    }
  }
}
```

Code Block 6 `pft` function

```
void pft() {
  printf( "active threads: " );
  for( int i = 0; i < M; i++ ) {
    if( threads[i] == 1 ) {
      printf( "%d, ", i );
    }
  }
  printf( "\n" );
}
```

Code Block 7 `pwork` function

```
void pwork( jobdata *data ) {
  printf( "thread working on fileidx = %d with:\n"
      "\tcommand = %s\n"
      "\tjob_ID = %d\n"
      "\tworker_ID = %d\n"
      "\tstring = %s\n"
      ,
      data->fileidx,
      cmds[data->cmdidx],
      data->job_ID,
      data->worker_ID,
      data->string );
}
```

Code Block 8 `pjob` function

```
void pjob( int idx ) {
  tjob *job = jobs[idx];
  printf( "file <<%s>> currently has:\n"
      "\tworkers = %d\n"
      "\tcount = %d\n"
      "\tcurr_job = %d\n"
      ,
      files[idx],
      job->workers,
      job->count,
      job->curr_job );
}
```

Code Block 9 `test_print` function

```
void test_print( int fileidx, jobdata *data ) {
  pfiles();
  pft();
  pjob( fileidx );
  pwork( data );
}
```

2.5 Helper Functions

To help with repetitive tasks, some helper functions were created.

- The function `getcmd()` (Code Block 10) accepts a buffer and a buffer size. It sets the buffer to zero and places user input onto it.
- Every time a valid command is entered, `log_command()` (Code Block 11) takes the current time as a string, concatenates it with the user request, and appends it to `commands.txt`.
- The function `strindex()` (Code Block 12) searches the index of a string in a string array (the size of the array is also a parameter). It returns `-1` if the string is not found.

Code Block 10 `getcmd` function

```c
int getcmd( char *buf, int nbuf ) {
  printf( "> " );
  memset( buf, 0, nbuf );
  if( fgets( buf, nbuf, stdin ) != NULL ) {
    return 0;
  }
  return 1;
}
```

Code Block 11 `log_command` function

```c
void log_command( char *cmd ) {
  time_t curr_time;
  char log[150] = { 0 };
  curr_time = time( NULL );

  // add colon and space after timestamp
  strcpy( log, ctime( &curr_time ));
  log[strlen( log ) - 1] = ':';  // turn '\n' to ':'
  strcat(log, " ");

  cmdfile = fopen( "commands.txt", "a" );
  fputs(strcat(log, cmd), cmdfile);
  fclose( cmdfile );
}
```

Code Block 12 `strindex` function

```c
int strindex( char *str, char arr[][MAXLEN], int narr ) {
  for( int i = 0; i < narr; i++ ) {
    if( strcmp( str, arr[i] ) == 0 ) {
      return i;
    }
  }
  return -1;
}
```

2.6 Worker Thread Functions

When worker threads are dispatched, they call a few functions to ensure that proper order is enforced and race conditions are prevented. Recall that each thread has its own `struct Jobdata`, and a file that a thread is assigned to has a corresponding `struct TJob` for managing worker threads.

- The function `dispatch_worker()` (Code Block 14) is where all worker threads begin. Using the provided `struct Jobdata`, a thread calls `enqueue_worker()` to add itself to the `workers` member of the assigned file's `struct TJob`. This allows the file to stay active as long as at least one thread is still pending execution.

  Afterwards, the worker thread is put to sleep via `xsleep()`. The `global_lock` of the file's `struct TJob` is then acquired to prevent concurrent operations on the same file. While the thread's job ID is not equal to the `struct TJob`'s current job, the thread goes to sleep.

  When it is the worker thread's turn to operate on a file, its `struct Jobdata`'s `cmdidx` is checked via a `switch` statement to identify which operation it will execute (and thus which function to call). After doing its operation, the worker thread releases `global_lock`, removes itself from `worker` via `dequeue_worker()`, and calls `finish_job()`.

- Before a thread carries out an operation, `enqueue_worker()` (Code Block 15) is called to increment the `workers` member of the assigned file's `struct TJob`. Since multiple threads may want to call this function at one time, the struct's `wlock` is used to prevent race conditions.

- After an operation, `dequeue_worker()` (Code Block 16) is called to reverse the effect of `enqueue_worker()`.

- The function `xsleep()` (Code Block 13) generates a random number between 0–99. If the generated number is less than 80, the worker thread sleeps for 1 second. Otherwise, the sleep duration is 6 seconds. This way, the chances of a thread to sleep 1 and 6 seconds is 80% and 20%, respectively. If testing is enabled, the sleep duration is printed.

When performing an assigned operation, a worker thread calls one of three functions: `xread()`, `xwrite()`, or `xempty()` (corresponding to "read", "write", and "empty" commands). Each of these functions accepts a `struct Jobdata` which allows the thread to identify which file to open and what string to append, if any. None of these functions acquire or release `global_lock` when reading from or writing to the assigned file since the calling function `dispatch_worker()` does this beforehand.

- The function `xread()` (Code Block 17) writes the contents of a file to `read.txt`. It first acquires the global mutex `rlock` to prevent write errors, since multiple threads may call `xread()` at the same time. The function first tries to open the thread's assigned file. If the assigned file does not exist, it writes this fact in `read.txt`. Otherwise, the contents of the assigned file are copied to `read.txt`, character by character. Afterwards, both the assigned file and `read.txt` are closed and `rlock` is released. If testing is enabled, then the name of the read file is printed before returning.

- The function `xwrite()` (Code Block 18) writes a user-specified string to the thread's assigned file. First, it sleeps for $25 \times n$ milliseconds, where n is the size of the string. It then attempts to open the assigned file. If the file does not exist, it will create it. However, if the directory of the file does not exist, it will *not* proceed. Otherwise, the string is appended to the file all at once via `fputs()`. Again, there is no need to lock the file within the function. If testing is enabled, then the name of the file and the sleep duration are printed before returning.

- The function `xempty()` (Code Block 19) writes the contents of a file to `empty.txt` then clears its contents. Similar to `xread()`, the global mutex `elock` is acquired to prevent write errors from simultaneous writes. If the assigned file does not exist or is already empty, it writes this fact in `empty.txt`. Otherwise, the contents of the assigned file are copied to `empty.txt`, character by character, then it is emptied via an `fopen()` call. Afterwards (or if the file exists but was found already empty), the file is closed and `elock` is released. It then goes to sleep for a random amount of time between 7 and 10 seconds via a `rand()` call. Since $(n \bmod 4) \in \{0, \ldots, 3\}$ for any integer n, the expression $(r \bmod 4) + 7$ will provide a random number between 7 and 10 for a random integer r. This number is then used as a parameter to `sleep()` to make the thread sleep. If testing is enabled, then the name of the file and the sleep duration are printed before returning.

The last function that a worker thread calls is `finish_job()` (Code Block 20) which takes in the thread's `struct Jobdata` as a parameter. Since multiple threads may complete a job at the same time, the lock `tlock[i]` is acquired, where `i` is the index of the file in the `files` array.

The function first checks if there are worker threads pending execution by checking the `struct TJob`'s `workers` member. If it is equal to zero, then the `struct TJob` is freed, its entry in `jobs` is set to `NULL`, and the corresponding entry in `files` is set to `" "`. Otherwise, the `curr_job` member is incremented and the remaining worker threads are woken up via `pthread_cond_broadcast()`. Afterwards, `tlock[i]` is released.

In any case, the specific worker thread that called `finish_job()` will retire at this point. In order to prevent race conditions (e.g., the same thread retiring and being dispatched at the same time), `qlock` is acquired. The thread's entry in `threads` is set to `0`, and its corresponding `struct Jobdata` is freed.

Code Block 13 `xsleep` function

```
void xsleep( int worker_ID ) {
  int s;

  srand(time(0));
        int r = rand()%100;
  s = ( r < 80 ) ? 1 : 6;
  sleep(s);

  if( test_mode ) {
    printf( "thread %d slept for %d seconds.\n", worker_ID, s );
  }
}
```

Code Block 14 `dispatch_worker` function

```
void *dispatch_worker( void *arg ) {
  jobdata *data = (jobdata *) arg;
  tjob *job = jobs[data->fileidx];

  // begin operation
  enqueue_worker( job );
  xsleep( data->worker_ID );

  pthread_mutex_lock( &( job->global_lock ) );

  while( job->curr_job != data->job_ID ) {
    pthread_cond_wait( &( job->done ), &( job->global_lock ) );
  }

  // execute command
  switch( data->cmdidx ) {
    case 0:
      xread( data );
      break;
    case 1:
      xwrite( data );
      break;
    case 2:
      xempty( data );
      break;
  }

  pthread_mutex_unlock( &( job->global_lock ) );

  // end operation
  dequeue_worker( job );
  finish_job( data );
}
```

Code Block 15 `enqueue_worker` function

```
void enqueue_worker( tjob *job ) {
  pthread_mutex_lock( &( job->wlock ) );
  job->workers++;
  pthread_mutex_unlock( &( job->wlock ) );
}
```

Code Block 16 `dequeue_worker` function

```
void dequeue_worker( tjob *job ) {
  pthread_mutex_lock( &( job->wlock ) );
  job->workers--;
  pthread_mutex_unlock( &( job->wlock ) );
}
```

Code Block 17 `xread` function

```
void xread( jobdata *data ) {
  FILE *file;
  char *fname;

  pthread_mutex_lock( &rlock );

  fname = files[data->fileidx];
  readfile = fopen( "read.txt", "a" );
  file = fopen( fname, "r" );

  fprintf( readfile, "read %s: ", fname );

  if( file == NULL ) {
    fputs( "FILE DNE\n", readfile );
  }
  else {
    char ch;
    while(( ch = fgetc( file )) != EOF ) {
      fputc( ch, readfile );
    }
      fputc( '\n', readfile );
    fclose( file );
  }

  fclose( readfile );
  pthread_mutex_unlock( &rlock );

  if( test_mode ) {
    printf( "read %s complete.\n", fname);
  }
}
```

Code Block 18 `xwrite` function

```
void xwrite( jobdata *data ) {
  FILE *file;
  char *fname;
  char *string = data->string;
  int ms = 25 * strlen( string );

  usleep( ( unsigned int )( 1000 * ms ));

  fname = files[data->fileidx];
  file = fopen( fname, "a" );

  if( file == NULL ) {
    printf( "Target directory %s does not exist.\n> ", fname );
  }
  else {
    fputs( string, file );
    fclose( file );
  }

  if( test_mode ) {
    printf( "write %s complete. slept for %d milliseconds.\n", fname, ms);
  }
}
```

Code Block 19 `xempty` function

```
void xempty( jobdata *data ) {
  FILE *file;
  char *fname;
  char ch;

  pthread_mutex_lock( &elock );

  fname = files[data->fileidx];
  emptyfile = fopen( "empty.txt", "a" );
  file = fopen( fname, "r" );

  fprintf( emptyfile, "empty %s: ", fname );

  if( file == NULL || ( ch = fgetc( file )) == EOF ) {
    fputs( "FILE ALREADY EMPTY\n", emptyfile );
    if ( ch == EOF ) { fclose( file ); }
  }
  else {
    // appending to empty.txt
    do {
      fputc( ch, emptyfile );
    } while (( ch = fgetc( file )) != EOF );

    fputc( '\n', emptyfile );
    fclose( file );

    // emptying file
    file = fopen( fname, "w" );
    fclose( file );
  }

  fclose( emptyfile );
  pthread_mutex_unlock( &elock );

  // sleep
  srand(time(0));
  int r = rand()%4 + 7;
  sleep(r);

  if( test_mode ) {
    printf( "empty %s complete. slept for %d seconds.\n", fname, r);
  }
}
```

Code Block 20 `finish_job` function

```
void finish_job( jobdata *data ) {
  int i = data->fileidx;

  // finish job
  pthread_mutex_lock( &tlock[i] );
  tjob *job = jobs[i];
  if( job->workers == 0 ) {
    free( job );
    jobs[i] = NULL;
    strcpy( files[i], " " );
  }
  else {
    job->curr_job++;
    pthread_cond_broadcast( &(job->done) );
  }
  pthread_mutex_unlock( &tlock[i] );

  // retire worker
  pthread_mutex_lock( &qlock );
  threads[data->worker_ID] = 0;
  free( data );
  pthread_mutex_unlock( &qlock );
}
```

2.7 Master Thread Functions

The master thread is in charge of initializing the program, determining the correct data for a job, continuously receiving user requests, and dispatching worker threads.

- The `main()` function (Code Block 21) is where the master thread and the program as a whole begins. It declares the following variables:
  - `buf`: stores the user input as a string
  - `cpy`: stores a copy of user input for logging
  - `command`, `filepath`, and `string`: store the requested file operation, user-specified filepath, and string to be written (if any), respectively
  - `context`: a helper variable used by `strtok_r()` in tokenizing the string
  - `cmdidx`: the index of the command in the `cmds` array
  - `fileidx`: the index of the filepath in the `files` and `jobs` arrays
  - `job_ID`: represents a worker thread's order of operation on a file
  - `worker_ID`: represents a worker thread's location in the thread pool `tid`

  To begin setting up the program, the `init()` function is called with command line arguments as parameters. Then, a `while` loop is entered where user input is continuously received via `getcmd()` and stored in `buf`. For every iteration in the `while` loop, the master thread performs the following, in order:
  - The contents of `buf` are copied to `cpy`, then the last character of `buf` is changed from `‘n’` to `‘ ’` to enable tokenization.
  - The array `string` is initialized to `" "`.
  - Using `strtok_r()`, `buf` is tokenized based on the delimiter `" "` (stored in `space`), with `context` used to mark the location of the next token. The first token is stored in `command`. If this token is `"quit"`, the `while` loop exits and the program ends.
  - The command is validated by checking whether it exists in the `cmds` array via `strindex()`. If the command is invalid, the `while` loop iterates. Otherwise, the command is logged to `commands.txt` via `log_command()`.
  - The next token which is the user-specified filepath is stored in `filepath`. If the command is "write", then another token which is the write data is stored in `string`.
  - Using `strindex()`, the master thread checks whether the specified file is already active (i.e., already exists in `files`). If it is, its index in the `files` array is stored in `fileidx`. Otherwise, the file is instead registered via `register_job()` whose return value `fileidx` stores.
  - Since every operation requires one thread, an available worker thread is sought via `register_worker()` and its index is stored in `worker_ID`.
  - Now that the file has been registered in the `files` array, the corresponding `struct TJob` is prepared via `init_worker()`. The `count` member of this struct is stored in `job_ID` to function as the worker thread's job ID.
  - The data obtained by the previous steps are stored in `data` as a `struct Jobdata` via `init_worker()`.
  - If testing is enabled, then `test_print()` is called.

  - Lastly, the worker thread is dispatched via `pthread_create()`, with `data` passed as a parameter. In order to let the OS know that it can immediately reclaim memory once a worker thread finishes an operation, the thread is detached.

  Since the user may decide to quit while worker threads are not yet finished, the program exits with a call to `pthread_exit()`. This allows all worker threads to finish operation before terminating the program, preventing data loss.

- The function `init()` (Code Block 22) takes in the command line arguments as parameters. If `"test"` was used as an argument to the program, `test_mode` is set to `1`, enabling testing.

  All locks in `tlock` are initialized, as well as the global locks `qlock`, `rlock`, and `elock`. Then, all entries in `files` are initialized to `" "` (rather than `NULL`). This is necessary since `files` is passed to `strindex()` in `main()`, and `strindex()` calls `strcmp()` which has undefined behavior when dealing with `NULL` strings. Lastly, the log files `commands.txt`, `read.txt`, and `empty.txt`, are emptied via `fopen()` (then `fclose()`) calls.

- The function `register_job()` (Code Block 25) seeks a free slot in `files` given a filepath by finding the first entry stored as `" "`. If a slot is found, the filepath is stored in `files` and its index is returned. Otherwise, the `files` array is full (i.e., there are currently `N` active files). In this case, the program terminates via `pthread_exit()`.

- The function `init_job()` (Code Block 24) takes in the index `idx` of a file in `files` as well as its filepath as parameters. Since a worker thread may call `finish_job()` at the same time that the master thread calls `init_job()` and since both functions modify `files` and `jobs`, `tlock[idx]` is acquired to prevent race conditions.

  Note that at the start of the function (line 275), the filepath is stored again in `files` even though an earlier call from `main()` to `register_job()` already does this. This is necessary since `finish_job()` clears `files[idx]` when all worker threads are done. In between `register_job()` and `init_job()` calls is a nonzero amount of time wherein a lone worker thread may possibly call `finish_job()`. This would result in an error in the program where an active file is not stored in `files` but its `struct TJob` exists. This would cause the next worker thread to try to modify a file named `" "`, which is not intended. To prevent this very error, `strcpy()` is again called to redundantly save the file in `files`.

  Afterwards, the function simply checks whether `jobs[idx]` already has a `struct TJob`. if it does, its `count` member is incremented. otherwise, memory is allocated for a `struct TJob`. first, its locks are initialized. the `workers` member is initialized to `0` since no worker threads have enqueued themselves yet. since the job id of the first worker thread would be `1`, both `count` and `curr_job` are initialized to `1`. this is necessary to prevent a deadlock in `dispatch_worker()` where the first worker never wakes up. Lastly, the struct is stored in `jobs`.

  Afterwards, `tlock[idx]` is released, and `count` is returned as the worker thread's job ID.

- The function `register_worker()` (Code Block 25) seeks an available thread in `tid` by finding the first entry in `threads` equal to `0`. If a thread is found, its index `i` is returned and `threads[i]` is set to `1`. Since a thread may call `finish_job()` at the same time that the master thread calls `register_worker()` and since both functions modify `threads`, `qlock` is acquired to prevent race conditions.

- The function `init_worker()` (Code Block 26) takes in the data from `main()` equivalent to all of a `struct Jobdata`'s parameters. The function simply allocates memory for the struct, initializes each of its members with its corresponding parameter, and returns the struct's pointer.

Code Block 21 `main` function

```
int main( int argc, char *argv[] ) {
  char buf[MAXLEN]; // buffer where user input is stored
  char cpy[MAXLEN]; // copy of user input
  char command[10], filepath[51], string[51];
  char *context;    // used by strtok to save state
  int cmdidx;       // index of command in cmds array
  int fileidx;      // index of file in files/jobs array
  int job_ID;       // position of worker in job queue
  int worker_ID;    // position of worker in thread pool

  init( argc, argv );

  while( getcmd( buf, sizeof( buf ) ) == 0 ) {
    strcpy( cpy, buf );            // reset string
    buf[strlen( buf ) - 1] = ' ';  // turn '\n' to space
    strcpy( string, " " );         // reset string

    strcpy( command, strtok_r( buf, space, &context ));

    if( strcmp( command, "quit" ) == 0 ) { break; }

    // validating command
    if(( cmdidx = strindex( command, cmds, 3 )) == -1 ) {
      printf( "Invalid command.\n" ); continue;
    }
    log_command( cpy );

    // save filepath and/or string
    strcpy( filepath, strtok_r( NULL, space, &context ));
    if( cmdidx == 1 ) {
      strcpy( string, strtok_r( NULL, space, &context ));
    }

    // check if job already exists, else save it
    if(( fileidx = strindex( filepath, files, N )) == -1 ) {
      fileidx = register_job( filepath );
    }

    // find an available thread in thread pool
    worker_ID = register_worker();

    // ready jobs and worker data
    job_ID = init_job( fileidx, filepath );
    jobdata *data = init_worker( cmdidx, fileidx, job_ID, worker_ID, string );

    // print job and worker status for testing
    if( test_mode ) { test_print( fileidx, data ); }

    // dispatch worker
    pthread_create( &tid[worker_ID], NULL, dispatch_worker, ( void * )data );
    pthread_detach( tid[worker_ID] );
  }

  pthread_exit(NULL);
}
```

Code Block 22 `init` function

```
void init( int argc, char *argv[] ) {

  if( argc == 2 ) {
    test_mode = strcmp( argv[1], "test" ) ? 0 : 1;
  }

  for( int i = 0; i < N; i++ ) {
    if( pthread_mutex_init( &tlock[i], NULL ) != 0 ) {
      printf( "mutex init has failed\n" );
    }
  }

  if( pthread_mutex_init( &qlock, NULL ) != 0  ||
      pthread_mutex_init( &rlock, NULL ) != 0  ||
      pthread_mutex_init( &elock, NULL ) != 0 ) {
    printf( "mutex init has failed\n" );
  }

  for( int i = 0; i < N; i++ ) {
    strcpy( files[i], " " );
  }

  cmdfile = fopen( "commands.txt", "w" );
  readfile = fopen( "read.txt", "w" );
  emptyfile = fopen( "empty.txt", "w" );
  fclose( cmdfile );
  fclose( readfile );
  fclose( emptyfile );

  printf( "Initialization complete. Type \"quit\" to exit the program.\n" );
}
```

Code Block 23 `register_job` function

```
int register_job( char *filepath ) {
  for( int i = 0; i < N; i++ ) {
    if( strcmp( files[i], " " ) == 0 ) {
      strcpy( files[i], filepath );
      return i;
    }
  }
  // no available slots
  printf( "Too many jobs at once. Program will exit after all operations "
      "are finished.\n" );
  pthread_exit(NULL);
}
```

Code Block 24 `init_job` function

```
int init_job( int idx, char *filepath ) {
  pthread_mutex_lock( &tlock[idx] );

  strcpy( files[idx], filepath );
  tjob* job = jobs[idx];

  // update job
  if( job != NULL ) {
    job->count++;
  }
  // create job
  else {
    job = malloc( sizeof( tjob ) );

    if( pthread_mutex_init( &( job->wlock ), NULL ) != 0 ||
        pthread_mutex_init( &( job->global_lock ), NULL ) != 0 ) {
      printf( "mutex init has failed\n" );
    }

    if(pthread_cond_init( &( job->done ), NULL ) != 0) {
      printf( "cond init has failed" );
    }

    job->workers = 0;
    job->count = 1;
    job->curr_job = 1;
    jobs[idx] = job;
  }

  pthread_mutex_unlock( &tlock[idx] );
  return job->count;
}
```

Code Block 25 `register_worker` function

```
int register_worker() {
  pthread_mutex_lock( &qlock );
  for( int i = 0; i < M; i++ ) {
    if( threads[i] == 0 ) {
      threads[i] = 1;
      pthread_mutex_unlock( &qlock );
      return i;
    }
  }
  pthread_mutex_unlock( &qlock );
  // no available threads
  printf( "No more available threads. Program will exit after all "
      "operations are finished.\n" );
  pthread_exit(NULL);
}
```

Code Block 26 `init_worker` function

```
jobdata *init_worker( int cmdidx, int fileidx, int j_ID, int w_ID, char *str ) {
  jobdata *data = malloc( sizeof(jobdata) );
  data->cmdidx = cmdidx;
  data->fileidx = fileidx;
  data->job_ID = j_ID;
  data->worker_ID = w_ID;
  strcpy( data->string, str );
  return data;
}
```

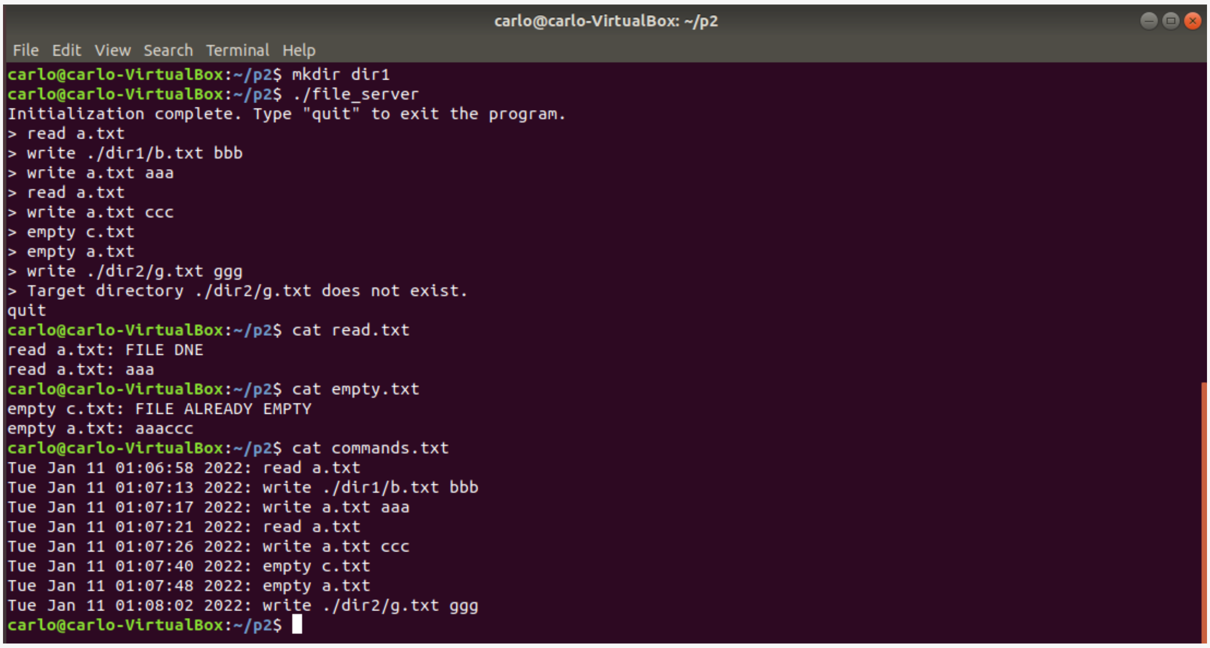
3 Testing the System

3.1 Correctness of Commands

Figure 1 shows a run of the program with testing disabled. The subdirectory `dir1` is created beforehand, but no files exist yet. Notice that the command `write ./dir2/g.txt ggg` is not carried out. This is because the subdirectory does not exist, and the program is not designed to handle this situation.

As shown in the log files, we can see that all commands work as intended.

Figure 1. Sample program run



3.2 Sequential Execution of Requests to the Same File

Figure 2 shows a run of the program with testing enabled. Recall that when testing is enabled, the following information is printed per request:

- All active files and all active worker threads
- The specified file's `struct TJob` members
- The worker thread's `struct Jobdata` members

The first command is `"read a.txt"` (1), which creates a new entry in `files`. Note that this file does not exist yet. Thus, a `struct TJob` is stored in `jobs`, with the expected initial values of its members. Since the worker thread's job ID is the same as the job's `curr_job`, it immediately begins operation.

As this is happening, a second command `"write a.txt fasjd"` is entered (2). The `count` member of the job is updated, and the job ID of the new worker thread is `2`.

A third "read" command to `a.txt` is supplied (3). Meanwhile,we find that the thread with a `worker_ID` of `1` (the second worker thread) is done sleeping (4). Since the first worker thread is not yet done reading, this thread sleeps again. Notice that in (5), the thread with a `worker_ID` of `0` has finished sleeping and reading. Only then does the second thread continue execution.

Lastly, an "empty" command (6) is entered. Notice that it is not carried out until the preceding read is finished (7) since, again, the worker thread's `job_ID` (which is `4`) and the job's `curr_job` (which is `3`) are not equal at the time that the command is issued.

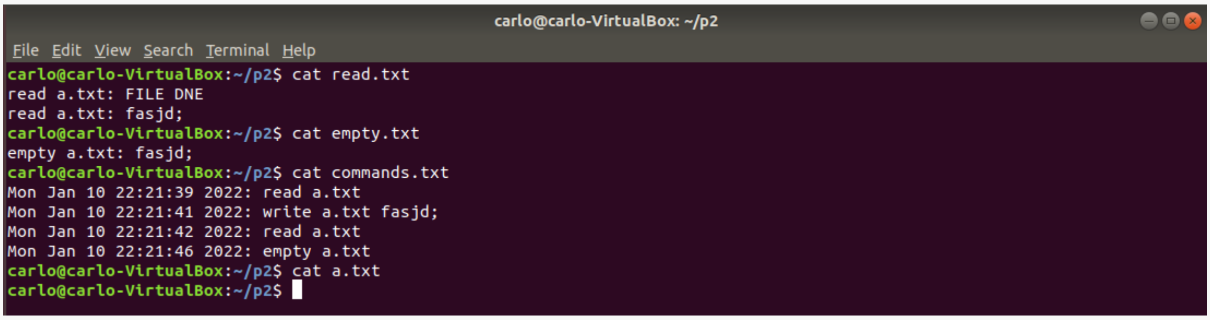
Figure 3 shows that the output of the log files are as expected. Notice that the first read to `a.txt` logs `FILE DNE`. After the write, the log is now `"fasjd"`. Since the last operation on `a.txt` was empty, running `cat a.txt` produces nothing.

With this test, we have shown the following:

- Reqests to the same file are carried out sequentially despite worker threads exiting `xsleep()` at unpredictable times, which prevents simultaneous operations to the same file.
- The master thread can continuously receive requests without blocking.

Figure 2. Submitting multiple requests to one file

Figure 3. Execution results



3.3 Concurrent Execution of Requests to Different Files

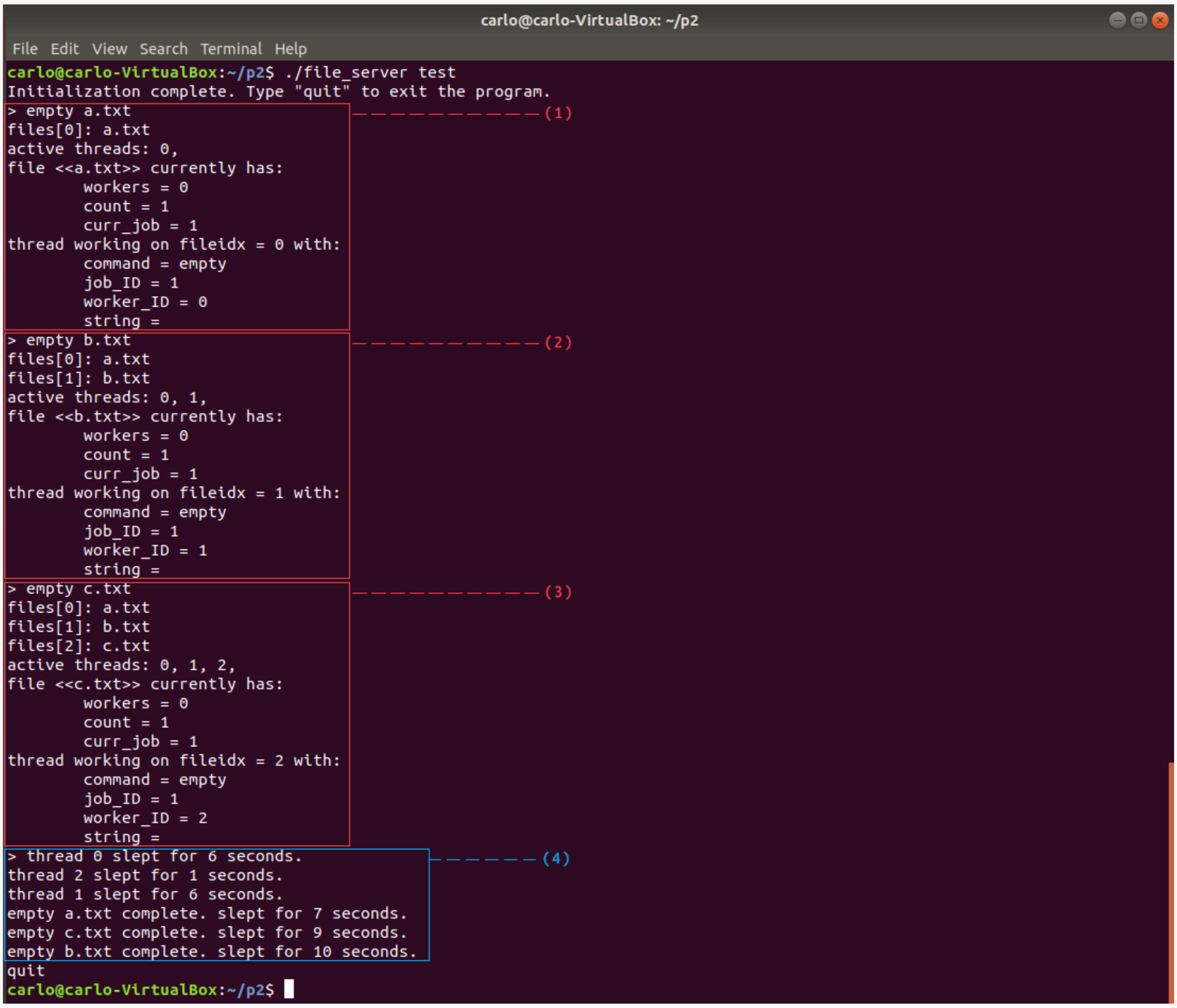
Figure 4 shows a run of the program with testing enabled. Three “empty” commands are issued (1–3). Note that for each of these commands, the list of active files is printed, and that each command registers a new file. This allows the program to perform these operations concurrently.

From (4) and Figure 5, we can see that the following are all different:

- The order in which the commands were issued
- The order in which the worker threads finished sleeping
- The order in which the operations were logged to `empty.txt`

This shows that since no order is enforced, the end result of concurrent operations is non-deterministic.

Figure 4. Submitting multiple requests to different files



```
carlo@carlo-VirtualBox: ~/p2
File Edit View Search Terminal Help
carlo@carlo-VirtualBox:~/p2$ ./file_server test
Initialization complete. Type "quit" to exit the program.
> empty a.txt                                    (1)
files[0]: a.txt
active threads: 0,
file <<a.txt>> currently has:
        workers = 0
        count = 1
        curr_job = 1
thread working on fileidx = 0 with:
        command = empty
        job_ID = 1
        worker_ID = 0
        string =
> empty b.txt                                    (2)
files[0]: a.txt
files[1]: b.txt
active threads: 0, 1,
file <<b.txt>> currently has:
        workers = 0
        count = 1
        curr_job = 1
thread working on fileidx = 1 with:
        command = empty
        job_ID = 1
        worker_ID = 1
        string =
> empty c.txt                                    (3)
files[0]: a.txt
files[1]: b.txt
files[2]: c.txt
active threads: 0, 1, 2,
file <<c.txt>> currently has:
        workers = 0
        count = 1
        curr_job = 1
thread working on fileidx = 2 with:
        command = empty
        job_ID = 1
        worker_ID = 2
        string =
> thread 0 slept for 6 seconds.                  (4)
thread 2 slept for 1 seconds.
thread 1 slept for 6 seconds.
empty a.txt complete. slept for 7 seconds.
empty c.txt complete. slept for 9 seconds.
empty b.txt complete. slept for 10 seconds.
quit
carlo@carlo-VirtualBox:~/p2$
```

Figure 5. Execution results

```
carlo@carlo-VirtualBox: ~/p2
File Edit View Search Terminal Help
carlo@carlo-VirtualBox:~/p2$ cat empty.txt
empty a.txt: FILE ALREADY EMPTY
empty c.txt: FILE ALREADY EMPTY
empty b.txt: FILE ALREADY EMPTY
carlo@carlo-VirtualBox:~/p2$ cat commands.txt
Tue Jan 11 02:09:46 2022: empty a.txt
Tue Jan 11 02:09:48 2022: empty b.txt
Tue Jan 11 02:09:51 2022: empty c.txt
carlo@carlo-VirtualBox:~/p2$
```