OVERLOOK SYSTEM

OS project report

JANUARY 10, 2021

ARCADIA YOULTEN AND FELIPE PEREZ
ARCADIA.YOULTEN FELIPE.PEREZ

Index

Design

Since Jack, Danny, and Wendy all use sockets, threads, and signals in a similar manner, their general functionality will be described here, and any specific cases will be further explained in their own section.

In each module, we use sockets. These sockets are used to pass information stored within frames between different modules. To connect properly to the sockets, we process config.txt file passed as a command line argument, extracting the port number and IP address from the file.

For example, Danny must connect to Jack and Wendy using sockets in order to send the frames with processed data.

Talk about socket array in jack and Wendy

Another one of the basic project requirements is to have many Danny processes connecting to a Single Jack/Wendy process. So, we chose to use threads in Jack and Wendy, with each thread communicating with one Danny process.

In order to assist us with this implementation, we created a pthread_t array. In each position we have the a pthread_t (pthread type struct variable) of a specific thread. So if, for example, we had multiple Danny processes connected, and one disconnected, we would be able to tell which one it was. Whenever we want to create a new thread, we must pass p_thread_create the following parameters:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

The first is a pointer to the thread we want to create, next is the function we want the thread to run, and finally is any arguments we want to pass to the thread. In our case, our thread creation looks as follows:

```
pthread_create(&tid_array[num_dannys], NULL, ThreadFunctionJack, &thread_data);
```

The main unique thing we do is that we pass a ThreadData type. This struct contains the required variables for the Wendy and Jack threads to work correctly. They have the same name in both Wendy and Jack but are implemented slightly differently. These are further explained in the dedicated Jack and Wendy sections.

Signals are another important aspect that is used in the project. For each module, we must end the program whenever we hit control + C in the terminal, or we use “kill -9” from the command line. In Linux, the signal that CTRL + C sends is SIGINT. Normally, that would just terminate the process, however, a best practice is to free all resources used when the program exits. So, instead of just using CTRL + C to end the program, we reprogram SIGINT so that we free all resources we were using, then a SIGTERM is raised so the program truly ends.

We reprogram by using the signal command, to replace the original purpose of the signal with another function. In our case, we use a function called ksighandler(). This handler is used for each signal reprogramming, and internally, it’s a large switch case to ensure that all signals are handled properly.

The main difference in the signals between each module is what exactly needs to be closed before the process terminates. For example, Wendy only joins all threads, close all file descriptors,

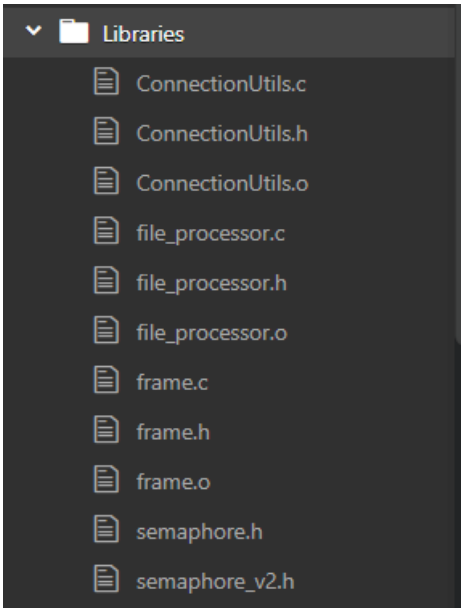
and frees dynamic memory. However, Jack joins all threads, closes all file descriptors, close the semaphores it was using, unlinks the shared memory it was accessing, and tells Lloyd (the child process) it needs to end execution.

The array of sockets and threads is also useful to use in combination with signals, as whenever Jack or Wendy receives a SIGINT, it loops through all the file descriptors, closing them one after another, and calls join for all threads.

Although we could have made the project with a single global makefile, in the end, we chose to give each separate module (Danny, Jack, and Wendy) their own separate one. Having one global makefile means that all different modules need to be present on the same computer, however, if we wanted to just connect a Danny from one computer to a Jack or Wendy on another, we would not be able to. With each independent makefile, we can have many different processes on many different devices.

Shared Library

As we were working on our project, we realized that many of the functionalities of the project are shared across modules. In order to avoid recoding elements that are not strictly unique to each module, we made a shared library folder. We included the following files:



The semaphore libraries are the libraries provided for use on the eStudy. Since every module must connect and process frames or files in some way or another, we chose to re-use these utilities. Due to this, many data structures are reused across modules. Instead of listing these repeated structures in each individual section, they will be listed here instead.

File Processor

File processor contains all the functions needed to process, modify, or analyse a file. Since each module must parse or process a file in some way, file processor was made a shared library. File processor contains many unique data types, to condense and contain the appropriate file information depending on what is needed.

File Processor's data types are shown below:

```
typedef struct {
    char *  weather_station;
    char *  file_directory;
    int     time_to_wait;
    char *  ip_jack;
    int     port_jack;
    char *  ip_wendy;
    int     port_wendy;
}ConfigDanny;

typedef struct {
    char *  ip;
    int     port;
}ConfigGeneral;
```

```
typedef struct {
    char * date;
    char * hour;
    char * temperature;
    char * humidity;
    char * atmosphere_pressure;
    char * precipiataion;
    char * file_name;
}WeatherFormat;

typedef struct {
    unsigned char md5sum[33];
    char * file_name;
    char * size_in_bytes;
    unsigned char * image_data;
}ImageFormat;
```

ConfigDanny is the data structure used to store configuration information that Danny needs. Although Danny does not have its own port/IP, it needs to know both Wendy's and Jack's in order to communicate with both of them. All of this information is generated from the configuration text file given to Danny when its process is started.

ConfigGeneral is the data structure used to store configuration information for both Jack and Wendy. The config files that both Jack and Wendy receive simply contain their own port and IP, so after processing it from the text file, we store it in their respective instances of the data type.

WeatherFormat is the data structure used to store the information for the weather stations. The information from these files is passed between Danny and Jack in a frame format.

ImageFormat is the data structure used to store about the images that Danny receives. The information from these files is passed between Wendy and Danny in a frame format.

Frame

Next is the Frame Module. As made obvious by its name, Frame controls everything having to do with frames in the project.

In Frame, we have a single unique data type, frame. As is specified in the requirements, it has a source which is a string of 14 characters, a type as a char, and data which is a string of 100 characters. Since all frames are sent the same way with the same structure, we thought it would be a good idea to make it a data type in order to maintain consistency.

Although frame does not have many defined Data types, we also defined all the constants that would be needed to process and send or receive frames. This way, if we need to change the type of symbol needed for one of the frames, we would only have to change it here, rather than throughout the project.

Both are shown in the images below:

```c
#define _GNU_SOURCE
#define SOURCE_SIZE 14
#define DATA_SIZE 100
#define FRAME_SIZE 115

#define SOURCE_DANNY "DANNY"
#define SOURCE_JACK "JACK"
#define SOURCE_WENDY "WENDY"
#define DATA_ERROR "ERROR"
#define DATA_CONNECT_OK "CONNECTION OK"
#define DATA_OK "DADES OK"
#define DATA_KO "DADES KO"
#define IMAGE_OK "IMATGE OK"
#define IMAGE_KO "IMATGE KO"
#define DATA_FRAME_ERROR "ERROR DE TRAMA"
#define TYPE_CONNECT 'C'
#define TYPE_DATA 'D'
#define TYPE_IMAGE 'I'
#define TYPE_IMAGE_FRAME 'F'
#define TYPE_DISCONNECT 'Q'
#define TYPE_OK_CONNECT 'O'
#define TYPE_OK_DATA 'B'
#define TYPE_OK_IMAGE 'S'
#define TYPE_ERROR_CONNECT 'E'
#define TYPE_ERROR_DATA 'K'
#define TYPE_ERROR_FRAME 'Z'
#define TYPE_ERROR_IMAGE 'R'
#define NUM_HASHTAGS_DATA 5
#define NUM_HASHTAGS_IMAGE 2
```

```c
typedef struct {
    char    source[SOURCE_SIZE];
    char    type;
    char    data[DATA_SIZE];
}Frame;
```

ConnectionUtils

The last module in our shared library folder is ConnectionUtils. This module is used to set-up sockets, check IP addresses/ports, and determines if connections should be accepted or denied.

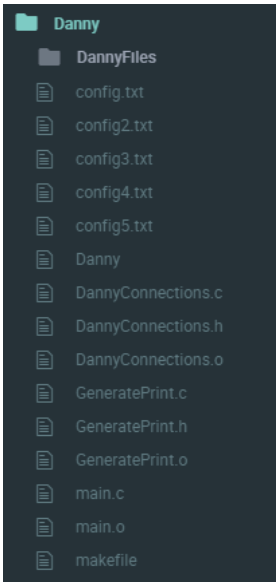
Unlike the other 2 shared libraries, ConnectionUtils contains only a single define:

```c
#define MAX_CONNECTIONS 10
```

This is a self-imposed limit on the number of Dannys that we can have connected at once. In order to avoid overloading the system, we have limited ourselves to 10. Additionally, since it needs the defined frame data type, We, plus functionalities involving sending and receiving frames, we also include frame.h in the ConnectionUtils.h, otherwise, it would be impossible to send back the frames accepting/denying the connection.

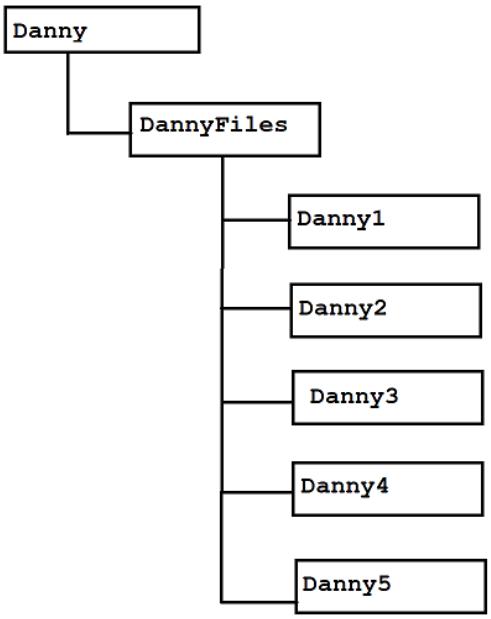
Danny

The Danny module contains the following files:



Danny uses the following system resources: pipes, signals, and forks.

The Danny module also contains a folder called DannyFiles. This is where all the files to be processed are located. In order to ensure that all files are processed and deleted correctly, we make multiple folders with different txt files in them with weather data.



The image above demonstrates the directory listing specifically focusing on how the Danny files are structured. The specific “DannyN” folder that a Danny client will choose is determined by whichever config file it is passed at start-up. This way, we can guarantee that multiple Danny clients will not be accessing the same folder at once.

Aside from sockets, Danny uses signals to scan his directory for new files that may have been uploaded. As mentioned previously, we initially reprogram the signals so that they will function as we want them to with ksighandler. Then, in Danny, we process the config file, and store the time between writing files in the config data type. Finally, we pass alarm the time between writing files so that it knows when to raise the signal and process the files. To avoid complications of reading from a file

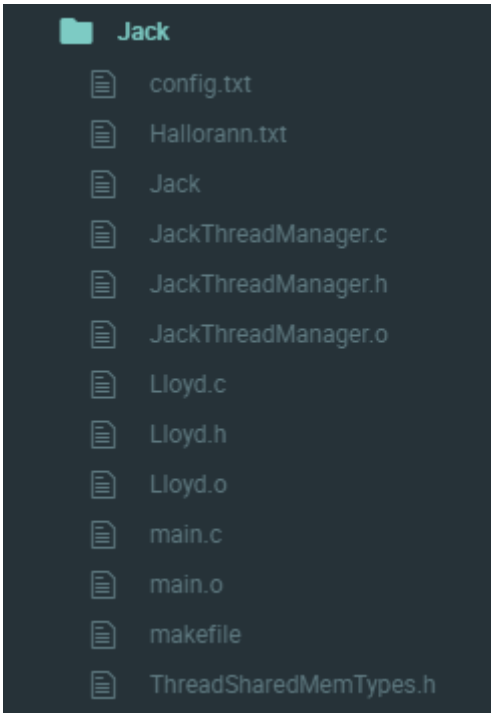
when Jack or Wendy closes, in the alarm function, before doing anything else, we poll the Jack and Wendy sockets to ensure that they are open before processing. This way, absolutely no data is corrupted. If the Jack socket or Wendy socket is closed, then we kill the Danny process by raising a SIGINT signal, which takes care of cleaning up the resources allocated before terminating the program.

So, every n second, a SIGALRM activates, and we process each text or image file that resides in one of the DannyN directories, if any are found.

Lastly, Danny uses forks to get the md5sum of the image files, in order to send it to Wendy. Since the process of performing and getting the md5sum of the file is the same, it is explained in the Wendy process section.

Jack

The Jack module contains the following files:



Jack uses the following system resources: Shared Memory, Semaphores, Signals, and Forks.

Every time Jack receives a connection from a Danny, that is after it has set up the listening socket, it will accept the connection, and create a new thread that will be dedicated to handling the communication with the newly connected Danny. The threads will write to the shared memory once data is received from the connected Danny.

In order to communicate between Jack and Lloyd, we were required to use shared memory. Whenever Jack received information from a Danny, it would process and write that data to a piece of shared memory that both Jack and Lloyd had access to. Then, Lloyd would process the information in the shared memory, and write the result of its calculations in a separate outside file.

Semaphores are used in Jack to ensure data protection of the shared memory. In this project, Jack must write new weather data to an area of shared memory, and Lloyd must process the data in the shared memory. To implement this properly, there are three main issues that need to be taken into consideration:

1) Lloyd must know when there's information in the shared memory
    a. Lloyd can only be told this via synchronization mechanisms
2) Only one Jack process (thread) should be writing to the shared memory at a time
3) Lloyd should be able to read from the shared memory without the data being changed

To meet these basic requirements, we decided that three semaphores were necessary, one for each issue, as shown below:

```
SEM_constructor(&sem_sync_lloyd);
SEM_init(&sem_sync_lloyd, 0);

SEM_constructor(&sem_mutex_shmem);
SEM_init(&sem_mutex_shmem, 1);

SEM_constructor(&sem_lloyd_done);
SEM_init(&sem_lloyd_done, 0);
```

In order to make sure that we could properly pass data to the shared memory, we created 3 semaphores. Sem_sync_lloyd is a synchronization semaphore used to tell the Lloyd process that there's information in the shared memory. sem_mutex_shmem is used to ensure that only one jack process writes to the shared memory at a time. Sem_lloyd_done is a synchronization semaphore used to let a Jack process know that Lloyd has finished processing the data in the shared memory.

By using these semaphores, we also implemented a method to make sure that Lloyd kills itself when needed. Whenever jack writes a "\0" to the shared memory, we signal Lloyd. When Lloyd reads, and it finds that the value of the name is not a valid value, we immediately break, and exit the LloydProcess function.

```
84      SEM_wait(&sem_sync_lloyd);
85
86      if(strlen(sh_data->name) <= 0) {
87        break;
88      }else if(strcmp(sh_data->name, LLOYD_WRITE)){
```

Since we are not allowed to make Lloyd a thread of Jack, and there is no configuration file to setup another process, we instead made it a fork. A fork is a clone of an existing running process. This is beneficial for various reasons. For example, In Jack, the first few things we do in the main is: construct/initialize semaphores and initialize the shared memory region. By forking and creating the Lloyd process immediately after this, there is no need to redo these initialization functions, and Lloyd already knows about 2 of the most important things it needs to function, the semaphores, and the shared memory. From there, Lloyd and Jack can communicate using shared memory and semaphores, as described previously. Jack will signal Lloyd using the sem_sync_lloyd semaphore whenever there was weather data received from a Danny, or when it must write to the "Hallorann.txt" file. Lloyd will check what is in the shared memory, and act accordingly.

Lloyd also uses signal blocking. Every 120 seconds, Lloyd must write to file called "Halloran.txt" with information about the average weather data. However, there could be problems if, for example, Lloyd is reading from the shared memory when the alarm goes off, and it would then have to write incomplete data to Halloran.txt. To avoid this problem, we used sigaddset, sigemptyset and sigprocmask functions.

```
76    sigset_t block_list;
77    //emtpy the set to avoid a valgrind error
78    sigemptyset(&block_list);
79    sigaddset(&block_list, SIGALRM);
```

This is to add the SIGALRM signal to the set of signals (sigset_t block_list) we will be operating with.

```
91        // BLOCK SIGALRM
92        sigprocmask(SIG_BLOCK, &block_list, NULL);
```

```
201       // UNBLOCK SIGALRM
202       sigprocmask(SIG_UNBLOCK, &block_list, NULL);
```

These 2 lines of code block and unblock the signals contained in the block_list sigset_t block_list, which in our case is simply the SIGALRM signal.

Jack also has its own unique data types that it uses to communicate across threads

```
#include "../Libraries/frame.h"

typedef struct {
  int sockfd;
  Frame frame;
  int danny_num;
} ThreadData;

typedef struct {
char name[DATA_SIZE]; //We use a fixed
float temperature;
int humidity;
float atmosphere_pressure;
float precipiataion;
} SharedMemData;
#endif
```

ThreadData is used in the to pass information to the threads for each Danny that connects. Frame is passed to know the name of the weather station, sockfd to know our own socket and be able to close it when the Danny disconnects, and danny_num is to know “which” Danny we are i.e where in the array of threads the pthread_t variable is located that corresponds to this thread.

SharedMemData is used to write information to the shared memory space. Instead of repeatedly writing single bits of different information to the shared data, we save all the different attributes we need to its appropriate space in the SharedMemData variable, and then write. This implementation also allows us to ensure that data is passed with integrity between Jack and Lloyd, as Lloyd can expect information from Jack in the same format, every time.

Wendy

The Wendy module contains the following files:

Wendy uses the following system resources, aside from sockets and threads: Pipes and Forks.

Wendy uses the same technique as Jack for its threads; as soon as a new connection comes in, it creates a thread that handles the communication between Wendy and the Danny that connected. The threads will receive the image data when Danny sends it over. Once all the image data is received, the thread will write it to a file of the same name as the original file (the thread knows this info thanks to the first frame sent by Danny that contains this information, along with the md5sum). Once written to a file, it will perform the md5sum of said file and compare it to the one received from the Danny with the initial frame and send back the result. The process of forking is explained below.

In Wendy, we had to use the fork function so that we could properly get and compare the m5sum of an image. One of the requirements for this project is that we could not implement new code to calculate the MD5SUM of the images. Instead, we had to use the function that was already available on the operating system. Since we are testing and working on LaSalle's Monserrat server, this of course, meant that we had to use Linux's md5sum command. However, we couldn't just call the function in our code, as it is a bash function. Instead, we had to use exec() instead.

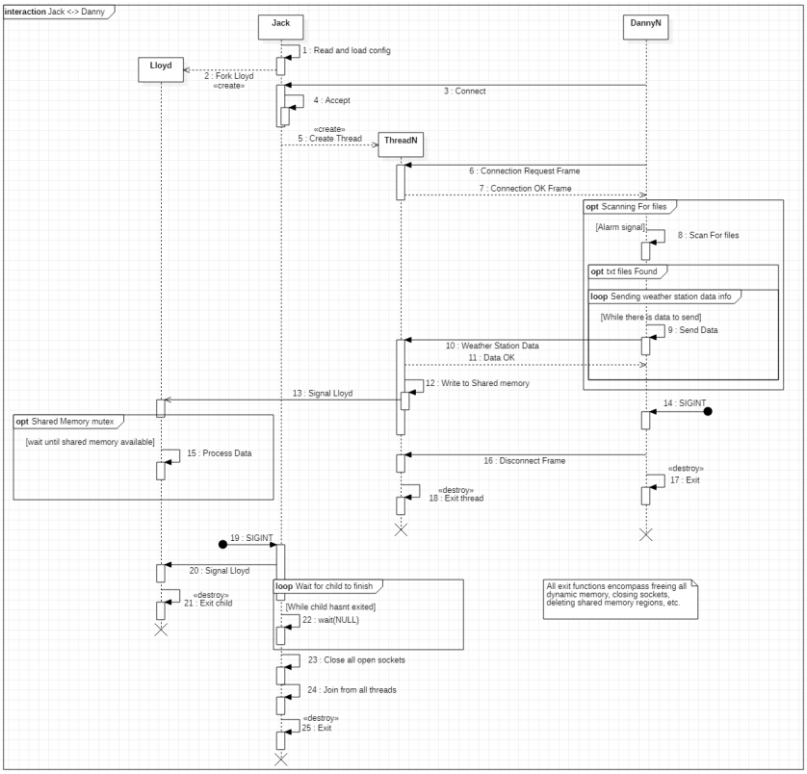
The exec family of functions are interesting, because it completely replaces the current active process with the one specified through arguments (in our case, /usr/bin/md5sum). Since the current process is entirely replaced, we need to fork to be able to continue the execution of the main process and let the child process be replaced by the md5sum process.

However, as explained earlier, if the current process is entirely replaced by the program called in the execl() function, how do we get the md5sum output to the parent Wendy process? In this case, we can use a combination of pipes and the function dup2(). In the parent, we create an array of file descriptors, called pipeAB. These are the read and write pipes (pipeAB[0] for reading and pipeAB[1] for writting). Because the output of the md5sum command is directed to STDOUT, we use dup2 to duplicate the value of the write end (pipeAB[1]) of the pipe to the STDOUT file descriptor (1). That way, whenever md5sum is called, instead of writing the md5sum to the standard output, it will be sent back to Wendy.

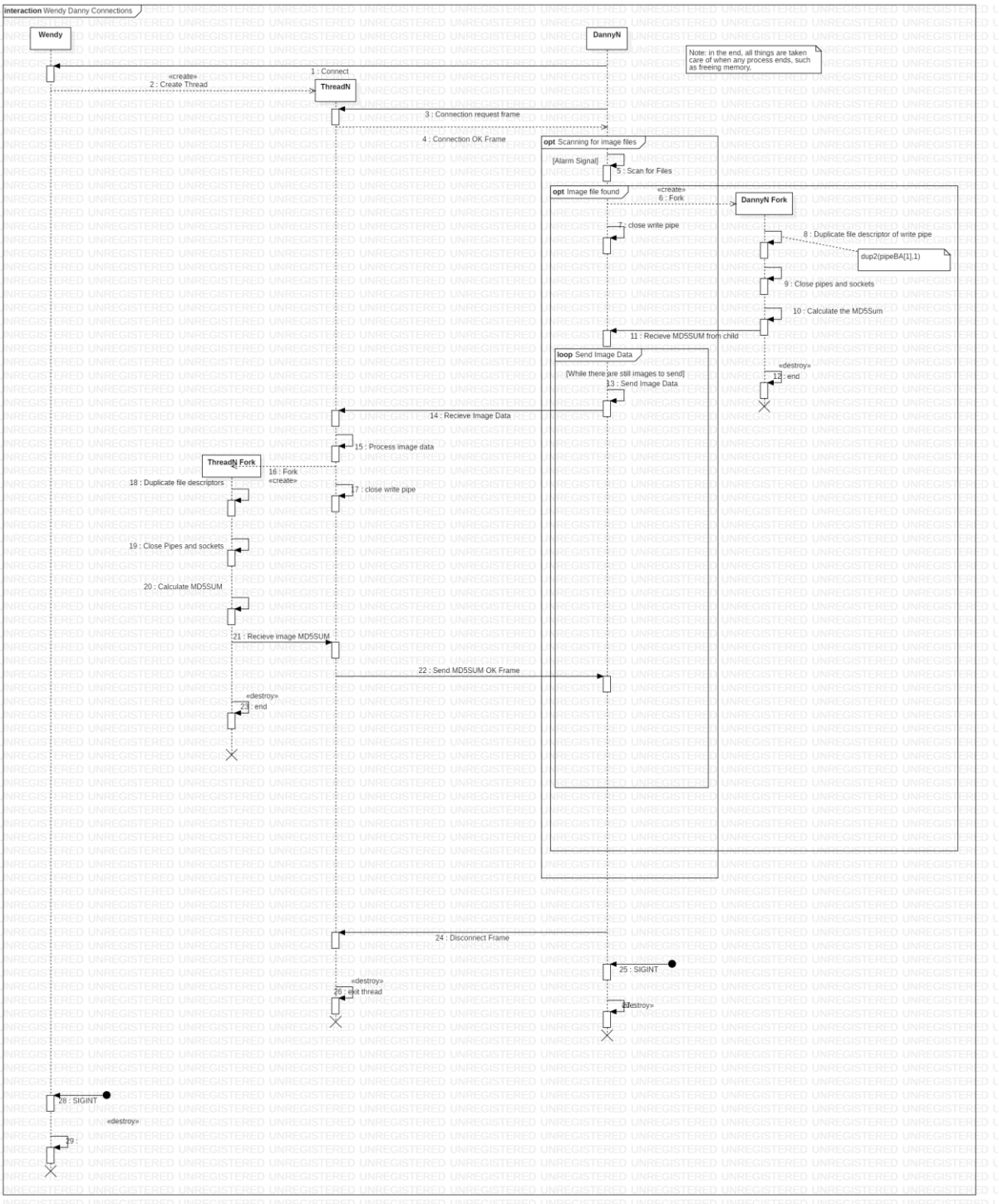
Barry is a file folder where all of the images are stored. If Wendy receives an image, and is able to process it correctly, it is stored in the Barry folder, otherwise, the information is deleted.

Communication Diagrams:

Danny Jack Communication



Wendy Danny Communications



This communication diagram shows how Wendy and Danny Communciate.

Observed Problems

One problem we had with Phase 2 was ensuring that once Jack died, all the Dannys also died. As stated in the Phase 2 requirements, it is necessary for all the Dannys to disconnect and end when Jack receives a CTRL + C signal. At first, we thought it made the most sense to continuously read from the socket. However, after testing, we realized. However, we found that we could continuously poll Jack's socket. If it was open, then we would get back and intelligible response. But if it was closed, then the response to our poll would be POLLRDHUP. By continuously polling the Jack and Wendy sockets, we would only have to wait for the POLLRDHUP response, and once that was received, we disconnected and quit the Dannys.

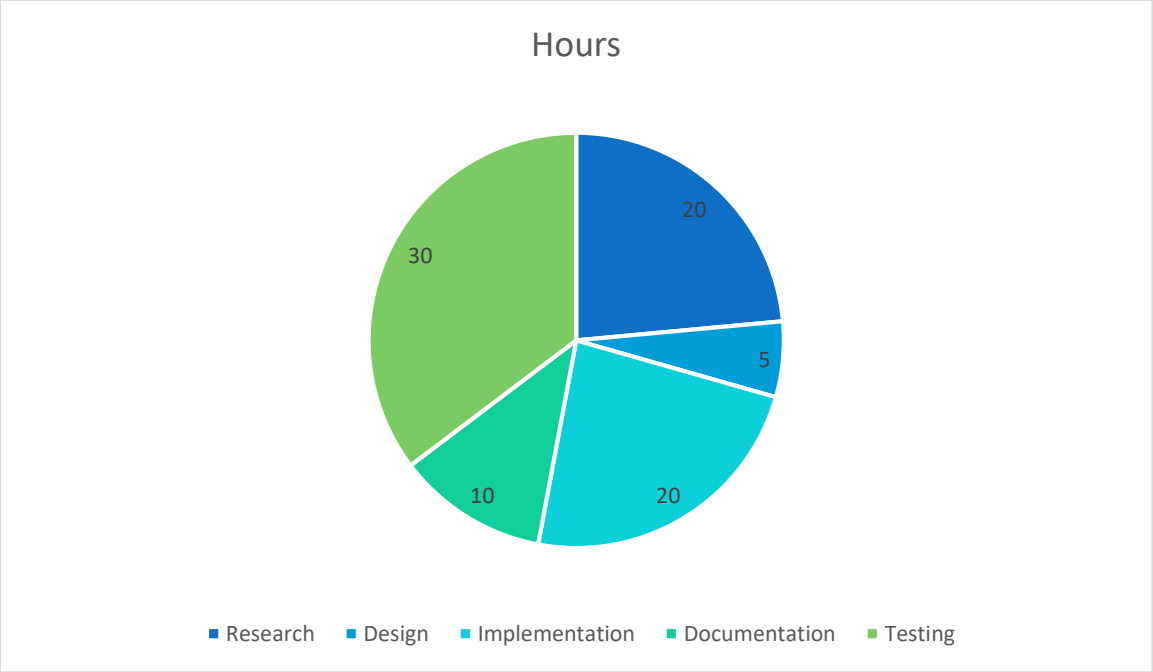
In Phase 3, one problem we had was with ensuring that Lloyd would always be able to properly write to the shared memory without interference. Initially, we thought it would be enough to have 2 semaphores, one to signal Lloyd that he should read from the shared memory, and another to make sure that only one process is writing to the shared process at once. Initially, we thought this setup would be okay, however, we didn't take into consideration that someone might try to write to the shared memory *while* Lloyd was trying to read from the shared memory. Our setup was something like this:

1) Signal that the shared memory is available to be written to
2) THEN tell Lloyd to read the shmem data.

After going back to the drawing board, we realized, that in order to ensure data integrity, we would need another semaphore for Lloyd to tell a Jack process that it has finished reading from the shared memory. This way, we guaranteed that only one process could write to the shared memory at once, Lloyd could properly finish processing the data, and that the next Jack process would easily be able to write to the shared memory.

In Phase 4 one problem we had was making sure all file descriptors closed. In theory, the only thing necessary to fix this problem would be adding a close(); with the appropriate file descriptor inside of it. However, due to previous feedback, whenever we coded an open file descriptor, we also coded its closing as well. So, when we ran valgrind, we were surprised to find that in some cases, file descriptors wouldn't close whenever we ended the program. Even after double checking the code, it seemed like were closing all of the file descriptors. After being able to determine which file descriptor was associated to a specific ID, we were able to tell that we weren't closing one of the pipes in Wendy properly. Originally, we had called execl(); and then we closed the child's write end of the pipe. However, because execl(); completely changes the process, the close(); after the execl() was never reached. By moving the close(); before the execl(); we properly close all file descriptors.

Temporal Estimation



In total, we spent a minimum of 75 hours on this project. Much of this time was spent working over 3-4 hour chunks over the past several months. For each Phase, we would first start by implementing the base of what we would need. This “base” includes minor code aspects, such as things we knew we could do right away any prints, and surface level testing of these elements. Then, once that was done, we would start to research into what options we had to implement the different aspects. For example, for Lloyd, we put time into researching how we could communicate between Jack and itself best using synchronization methods. Then, once we decided on the best method to implement, we would design how it would work with our implemented system. Finally, we would implement it, and do testing.

We ended up spending the most time on testing, because any code we wanted to implement usually required a bit of code rearrangement. This ranged from adding a few lines of code to an if statement, to making another .c/.h file, as we realized that a specific part would be better to reused multiple times, instead of just once. Implementation only fell shortly behind this, as after implementing something, we would either run into an error, or we would do extensive tests to ensure that it worked.

Design ended up taking the least amount of time. At the beginning of each meeting, we would take the time to see how this new project aspect would fit into our design and started implementing things from there. However, because we spent so little time on the design part, we often had to rethink our code, and ended up spending much more time in debugging that we should have.

Conclusion

In conclusion, we became much more familiar with many of the different tools that can be used when designing a system with multiple clients and servers. The main thing we learned was not anything specific about an individual tool, but rather, how we should use pipes, semaphores, sockets, signals, and shared memory in combination with one another. This made us realize that there are more possibilities of things to code with C. The most difficult part of this project was understanding how we should use each tool together. Although we had some small debugging problems, most of our issues stemmed from not properly understanding how each tool would work together. Once we understood this, we were able to fix the problems we had with any given section.

If we were to redo this project, we would probably organize our code from the beginning. We had to go back and put some functions in .c and .h files simply because many of our main files were bloated with code. This would have avoided some bugs that we encountered when trying to re-organize pre-existing code. We would also spend more time on the design phase, as we discovered that our design had to be frequently changed when we tried to implement something new, leading to much time spent in debugging.

Bibliography

user3735849user3735849 73311 gold badge55 silver badges66 bronze badges, et al. “How Do I Use Setsockopt(SO_REUSEADDR)?” *Stack Overflow*, 1 July 1963, stackoverflow.com/questions/24194961/how-do-i-use-setsockoptso-reuseaddr.

user507401user507401 2, et al. “Please Explain the Exec() Function and Its Family.” *Stack Overflow*, 1 Dec. 1959, stackoverflow.com/questions/4204915/please-explain-the-exec-function-and-its-family.

“What Is a Fork?” *Computer Hope*, Computer Hope, 7 Oct. 2019, www.computerhope.com/jargon/f/fork.htm.

“pthread_create(3).” *pthread_create(3) - Linux Manual Page*, Linux Manual Pages, man7.org/linux/man-pages/man3/pthread_create.3.html.