Group 13 – Phase 4 Remote Multitasking CLI Shell with Scheduling Capabilities

Jun & Sonya

Overview

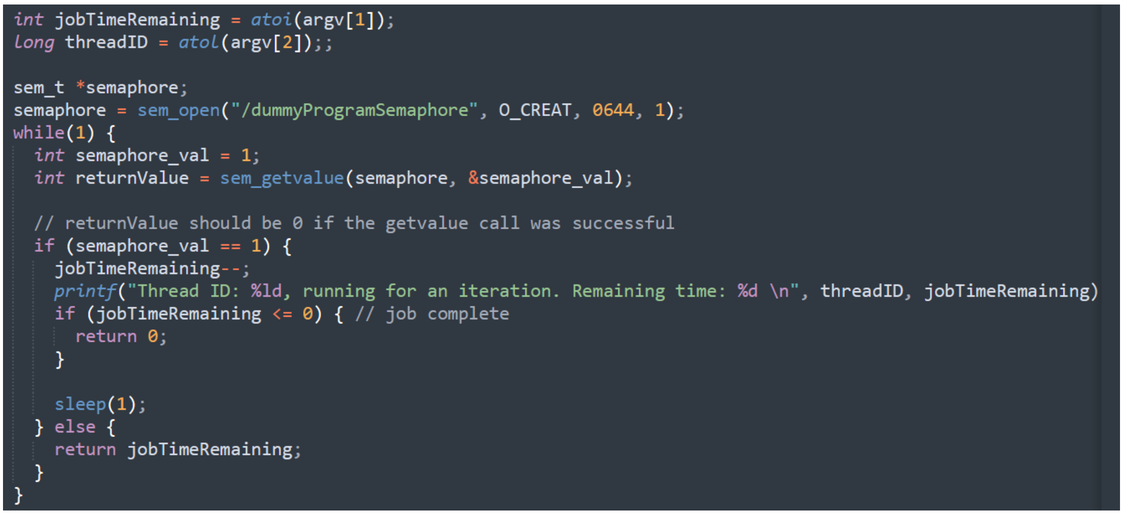
In Phase4, the program was upgraded from Phase3 with scheduling capabilities, Round Robin with different quantum based on the rounds and Shortest Job Remaining First. The server serves multiple concurrent clients using threads. A dummy program is created with a parameter representing the job remaining time as a simulation of process entering CPU. When a dummy program is called, it is recorded as a node and enter the waiting queue built with a linked list. When a new process/request arrive, the scheduler function will select the next process to execute based on SJRF controlled by round numbers. A counting semaphore is used to ensure mutually exclusive execution of different process.

Usage

1) Open the directory in TWO separate UNIX-based terminals
2) Compile the files required for running by using the Makefile. Run 'make all' to compile the client, server, and dummyProgram.c file.
3) In one of the terminals, run the server by running the command "./Phase3-server.o"
4) In the other terminals, run the client by running the command "./Phase3-client.o"
5) The order is important! The server should be run first before the client, otherwise the clients will report "Connection error" as there is no server to connect to.
6) Once the connection is created, the client terminal will report the successful connection. You can start entering commands listed on the CLI including command "./dummyProgram.o X" (X represents an integer) to send to the server, and the server will run the commands or put the program in the waiting queue before sending back the output of the commands to the client.
7) You can also connect with more than 1 client by repeating steps 4-6 to create more clients.
8) If the input commands are not among the listed ones, then the client will get an error message specifying "commands are not available".
9) To exit the program, input "exit" on the client end.

Implementation Details

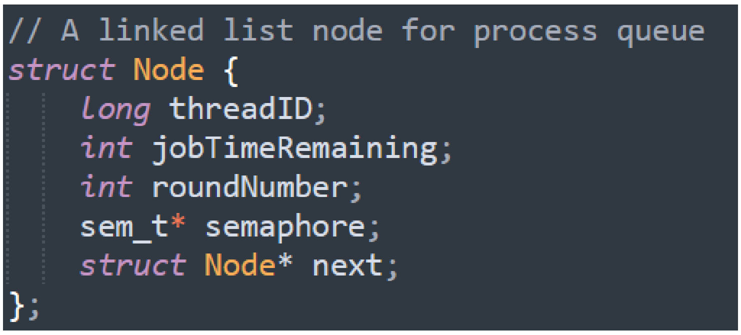
Dummy Program –



```c
int jobTimeRemaining = atoi(argv[1]);
long threadID = atol(argv[2]);;

sem_t *semaphore;
semaphore = sem_open("/dummyProgramSemaphore", O_CREAT, 0644, 1);
while(1) {
  int semaphore_val = 1;
  int returnValue = sem_getvalue(semaphore, &semaphore_val);

  // returnValue should be 0 if the getvalue call was successful
  if (semaphore_val == 1) {
    jobTimeRemaining--;
    printf("Thread ID: %ld, running for an iteration. Remaining time: %d \n", threadID, jobTimeRemaining)
    if (jobTimeRemaining <= 0) { // job complete
      return 0;
    }

    sleep(1);
  } else {
    return jobTimeRemaining;
  }
}
```

The dummy program takes a parameter as job remaining time and contains a loop with sleep function. The dummy program is compiled in MakeFile. After calling, it return an integer of its remaining job time. A named semaphore is used to allow it is only executed when being selected by the scheduler function. A named semaphore is used so that the semaphore is accessible in multiple files, the dummyProgram.c and the server.c files. The dummy program does not ‘wait’ for the semaphore, it only reads the value from it. This is so that the state of the semaphore is only controlled by the scheduler, and also prevents the possibility of a deadlock happening.

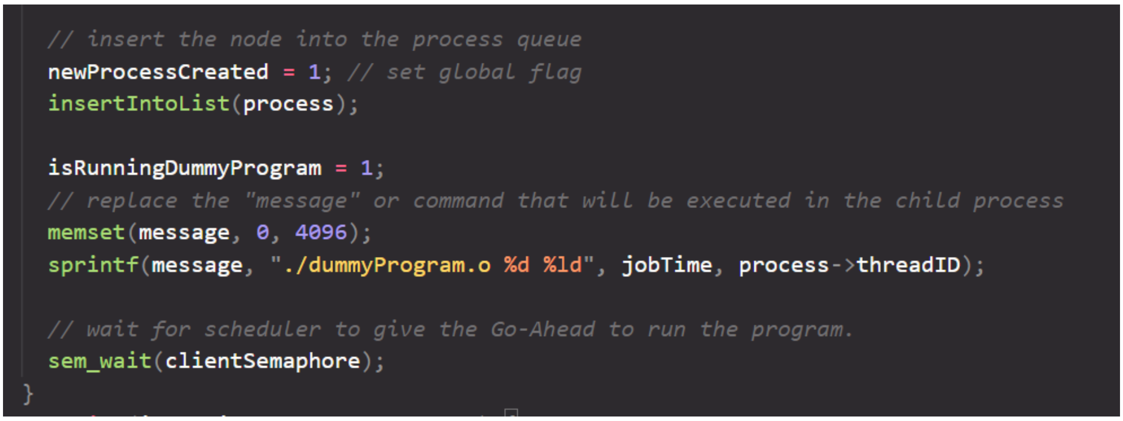
Node for holding process / Process Queue



```c
// A linked list node for process queue
struct Node {
    long threadID;
    int jobTimeRemaining;
    int roundNumber;
    sem_t* semaphore;
    struct Node* next;
};
```

A linked list structure is used to act as the process queue. Each node represents one process running on one thread and we record its threadID, remaining job time, round number, and a pointer to the thread semaphore.

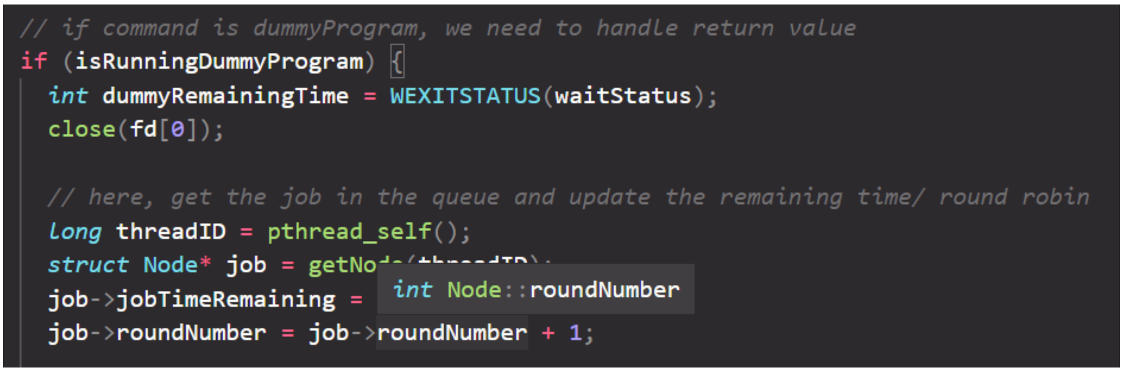
Client waiting for scheduler



```
  // insert the node into the process queue
  newProcessCreated = 1; // set global flag
  insertIntoList(process);

  isRunningDummyProgram = 1;
  // replace the "message" or command that will be executed in the child process
  memset(message, 0, 4096);
  sprintf(message, "./dummyProgram.o %d %ld", jobTime, process->threadID);

  // wait for scheduler to give the Go-Ahead to run the program.
  sem_wait(clientSemaphore);
}
```

When a client requests to run the dummyProgram, the client creates a ‘process’ object, and inserts it into the Process Queue. It then waits for the Scheduler to release the semaphore before actually executing the command.

“Pausing” the execution of dummyProgram

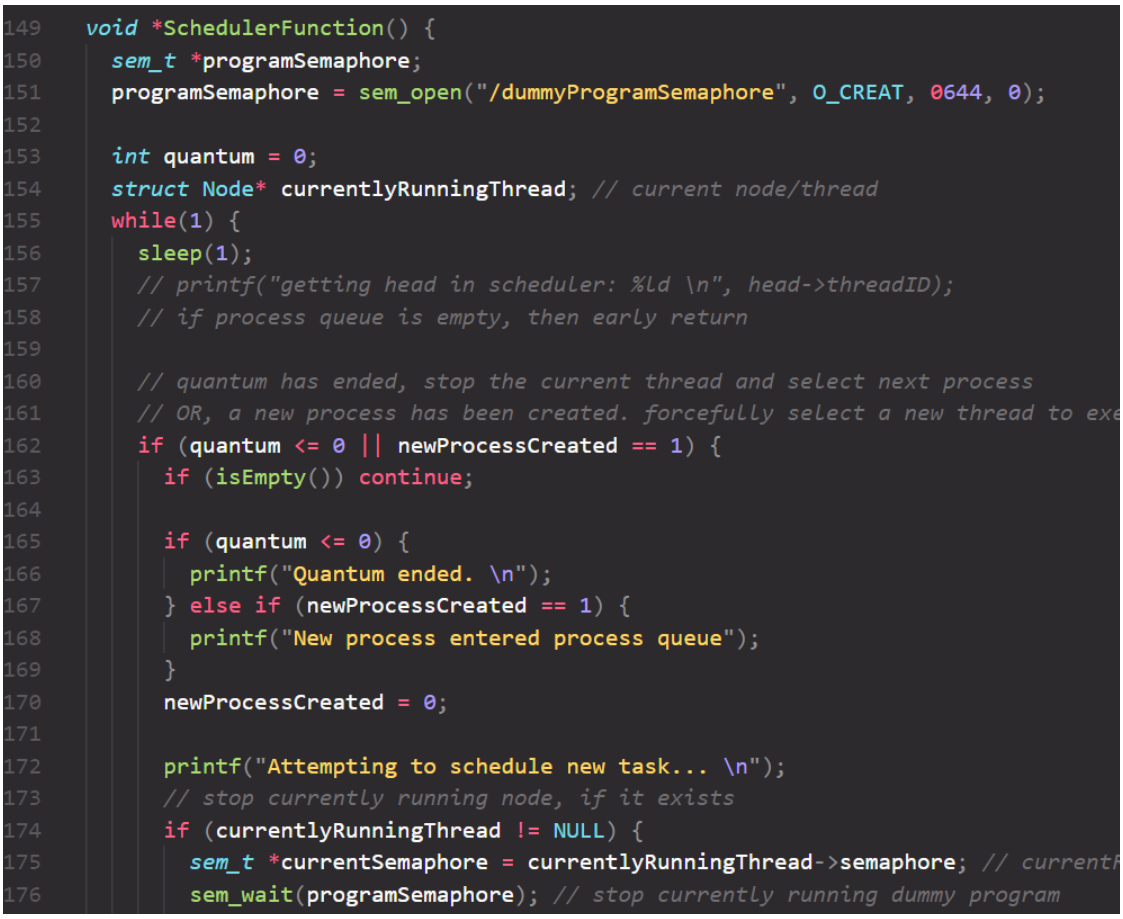


```
// if command is dummyProgram, we need to handle return value
if (isRunningDummyProgram) {
  int dummyRemainingTime = WEXITSTATUS(waitStatus);
  close(fd[0]);

  // here, get the job in the queue and update the remaining time/ round robin
  long threadID = pthread_self();
  struct Node* job = getNode(threadID);
  job->jobTimeRemaining =
  job->roundNumber = job->roundNumber + 1;
```

As seen in the screenshot above with the code of the dummyProgram, the execution is not actually paused. Instead, it returns early and returns the remaining execution time left after it’s current execution.

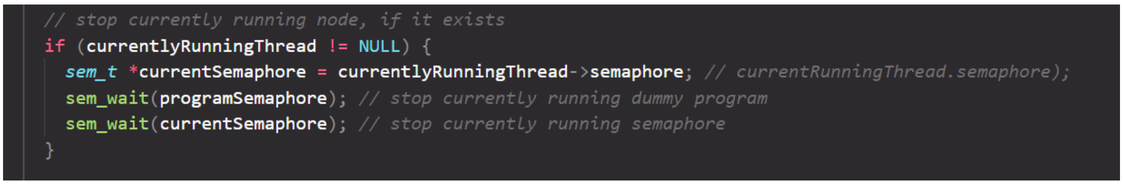
On the client thread, the parent process will get the return value from the dummyProgram execution after it is done executing (either return early because scheduler locked the named semaphore, or return when it has completed execution, jobTimeRemaining = 0). It then updates the details of the job in the Process Queue, and the Scheduler can select this job again in the future.

Scheduler function -



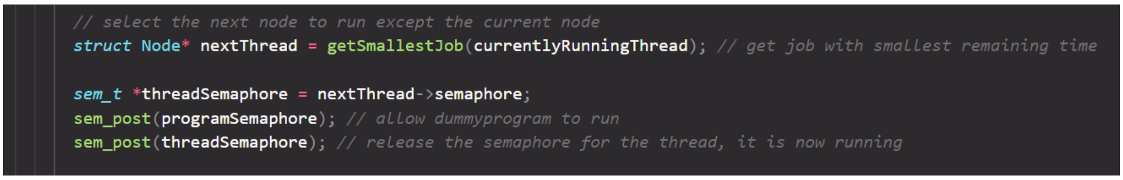
```c
void *SchedulerFunction() {
  sem_t *programSemaphore;
  programSemaphore = sem_open("/dummyProgramSemaphore", O_CREAT, 0644, 0);

  int quantum = 0;
  struct Node* currentlyRunningThread; // current node/thread
  while(1) {
    sleep(1);
    // printf("getting head in scheduler: %ld \n", head->threadID);
    // if process queue is empty, then early return

    // quantum has ended, stop the current thread and select next process
    // OR, a new process has been created. forcefully select a new thread to exe
    if (quantum <= 0 || newProcessCreated == 1) {
      if (isEmpty()) continue;

      if (quantum <= 0) {
        printf("Quantum ended. \n");
      } else if (newProcessCreated == 1) {
        printf("New process entered process queue");
      }
      newProcessCreated = 0;

      printf("Attempting to schedule new task... \n");
      // stop currently running node, if it exists
      if (currentlyRunningThread != NULL) {
        sem_t *currentSemaphore = currentlyRunningThread->semaphore; // currentR
        sem_wait(programSemaphore); // stop currently running dummy program
```

In the main function of the server, a thread is created to run the SchedulerFunction. The SchedulerFunction initializes the named semaphore used to control the execution of dummyPrograms. The function runs in a loop, with a sleep(1) every iteration to simulate a CPU cycle. Whenever the quantum for the currently running program has finished, or a new process has entered the queue, the scheduler will schedule a new process to run.



```c
// stop currently running node, if it exists
if (currentlyRunningThread != NULL) {
  sem_t *currentSemaphore = currentlyRunningThread->semaphore; // currentRunningThread.semaphore);
  sem_wait(programSemaphore); // stop currently running dummy program
  sem_wait(currentSemaphore); // stop currently running semaphore
}
```

It first stops the execution of any currently running thread by holding the semaphore for the threads, if any.

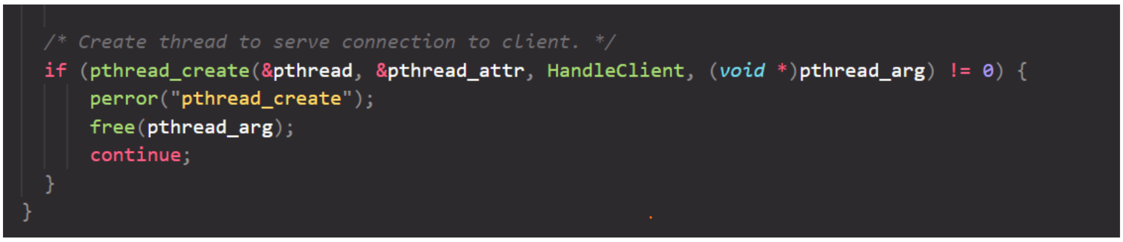


```c
// select the next node to run except the current node
struct Node* nextThread = getSmallestJob(currentlyRunningThread); // get job with smallest remaining time

sem_t *threadSemaphore = nextThread->semaphore;
sem_post(programSemaphore); // allow dummyprogram to run
sem_post(threadSemaphore); // release the semaphore for the thread, it is now running
```

Then, it selects a new thread to be run, and releases the semaphores required by the program to run for a time quantum based on that thread’s roundNumber. Once the time quantum finishes, this process repeats.

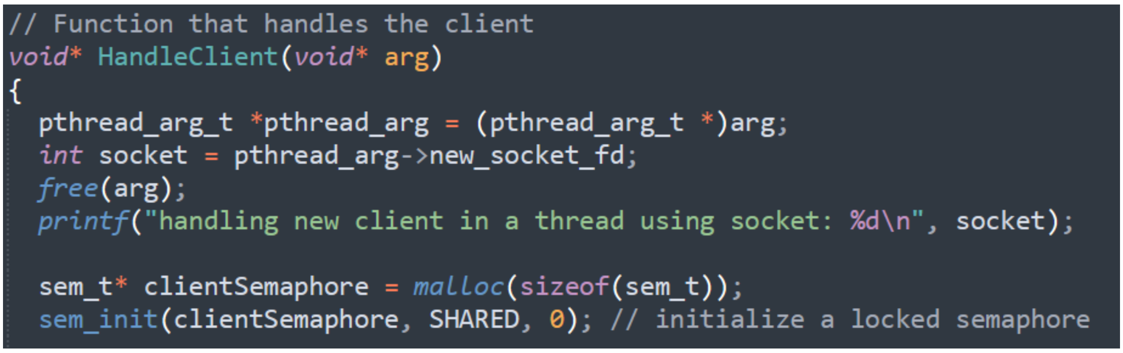
Multiclient capability -

After the server begins listening to connections, there is a while loop where the server will allocate new memory for the arguments needed for a new client, such as the socket fd and thread arguments. Once a new client connects, the server calls pthread_create on HandleClient() for each client, so there will be 1 thread running HandleClient() for each client that is connected to the server.



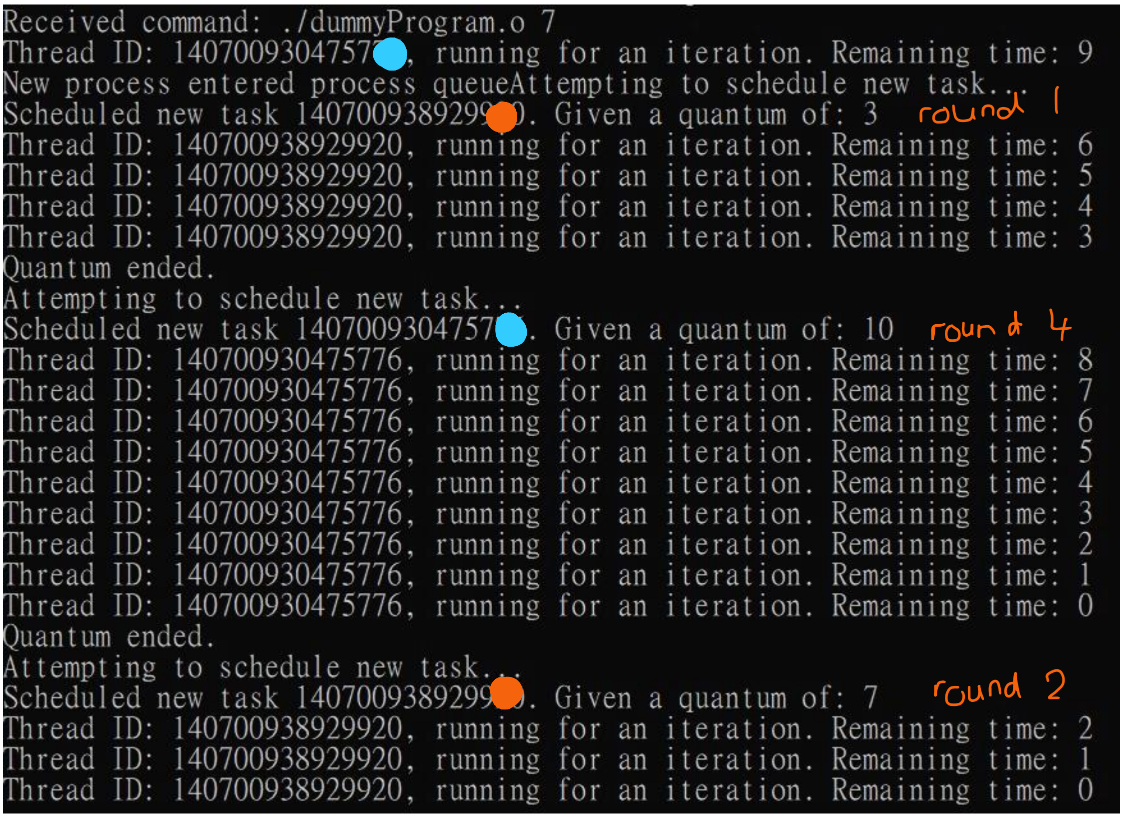
```
    /* Create thread to serve connection to client. */
    if (pthread_create(&pthread, &pthread_attr, HandleClient, (void *)pthread_arg) != 0) {
        perror("pthread_create");
        free(pthread_arg);
        continue;
    }
}
```

In HandleClient(), the thread is detached from the main process, so that the main process does not need to wait for the termination of this thread. After receiving the socket variable as an argument from the thread creator, the thread frees the argument and starts listening to user inputs from the client using the socket as previously implemented in Phase3. A semaphore is initialized for every client.



```
// Function that handles the client
void* HandleClient(void* arg)
{
  pthread_arg_t *pthread_arg = (pthread_arg_t *)arg;
  int socket = pthread_arg->new_socket_fd;
  free(arg);
  printf("handling new client in a thread using socket: %d\n", socket);

  sem_t* clientSemaphore = malloc(sizeof(sem_t));
  sem_init(clientSemaphore, SHARED, 0); // initialize a locked semaphore
```

Example execution log in server



```
Received command: ./dummyProgram.o 7
Thread ID: 1407009304757●, running for an iteration. Remaining time: 9
New process entered process queueAttempting to schedule new task...
Scheduled new task 1407009389299●0. Given a quantum of: 3   round 1
Thread ID: 140700938929920, running for an iteration. Remaining time: 6
Thread ID: 140700938929920, running for an iteration. Remaining time: 5
Thread ID: 140700938929920, running for an iteration. Remaining time: 4
Thread ID: 140700938929920, running for an iteration. Remaining time: 3
Quantum ended.
Attempting to schedule new task...
Scheduled new task 1407009304757●. Given a quantum of: 10   round 4
Thread ID: 140700930475776, running for an iteration. Remaining time: 8
Thread ID: 140700930475776, running for an iteration. Remaining time: 7
Thread ID: 140700930475776, running for an iteration. Remaining time: 6
Thread ID: 140700930475776, running for an iteration. Remaining time: 5
Thread ID: 140700930475776, running for an iteration. Remaining time: 4
Thread ID: 140700930475776, running for an iteration. Remaining time: 3
Thread ID: 140700930475776, running for an iteration. Remaining time: 2
Thread ID: 140700930475776, running for an iteration. Remaining time: 1
Thread ID: 140700930475776, running for an iteration. Remaining time: 0
Quantum ended.
Attempting to schedule new task...
Scheduled new task 1407009389299●0. Given a quantum of: 7   round 2
Thread ID: 140700938929920, running for an iteration. Remaining time: 2
Thread ID: 140700938929920, running for an iteration. Remaining time: 1
Thread ID: 140700938929920, running for an iteration. Remaining time: 0
```

Group 13 – Phase 3 Remote Multitasking CLI Shell

Jun & Sonya

Overview

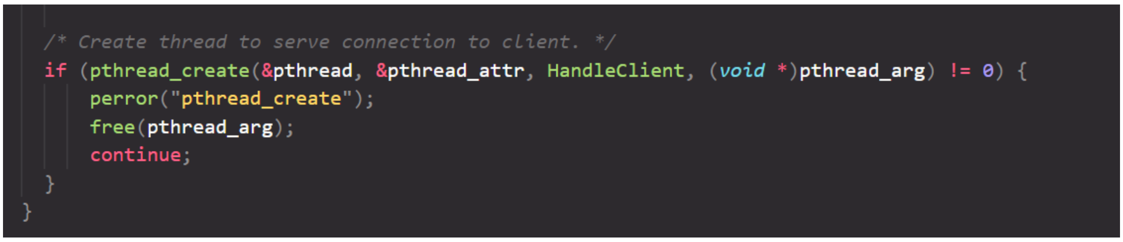
In Phase3, the program was upgraded from Phase2 with multiclient capability. The Phase3 server is able to serve multiple concurrent clients using threads. Some issues from Phase2 has also been fixed in this version, such as CTRL-C signals on the client now properly being sent to the server for proper termination. Commands that have no output are also working now. Executions of invalid commands no longer offset future user inputs.

Usage

1) Open the directory in TWO separate UNIX-based terminals
2) Compile the files required for running by using the Makefile. Run 'make all' to compile the client, server, and test .c file.
3) In one of the terminals, run the server by running the command "./Phase3-server.o"
4) In the other terminal, run the client by running the command "./Phase3-client.o"
5) The order is important! The server should be run first before the client, otherwise the client will report "Connection error" as there is no server to connect to.
6) Once the connection is created, the client terminal will report the successful connection. You can start entering commands listed on the CLI to send to the server, and the server will run the commands before sending back the output of the commands to the client.
7) You can also connect with more than 1 client by repeating steps 4-6 to create more clients.
8) If the input commands are not among the listed ones, then the client will get an error message specifying "commands are not available".
9) To exit the program, input "exit" on the client end.

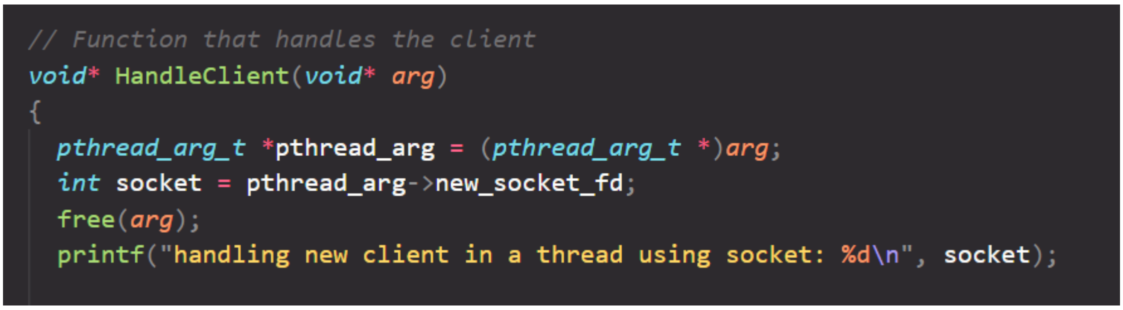
Implementation Details

Multiclient capability -
After the server begins listening to connections, there is a while loop where the server will allocate new memory for the arguments needed for a new client, such as the socket fd and thread arguments. Once a new client connects, the server calls pthread_create on HandleClient() for each client, so there will be 1 thread running HandleClient() for each client that is connected to the server.



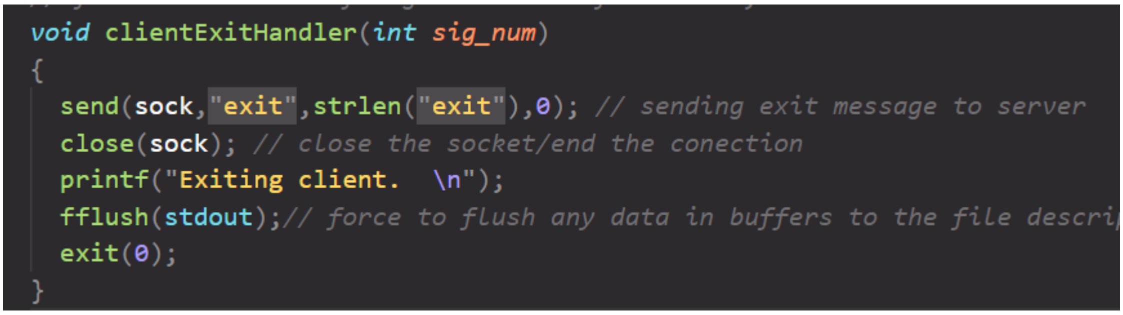
```
    /* Create thread to serve connection to client. */
    if (pthread_create(&pthread, &pthread_attr, HandleClient, (void *)pthread_arg) != 0) {
        perror("pthread_create");
        free(pthread_arg);
        continue;
    }
}
```

In HandleClient(), the code is mostly the same as in the single-client case as previously implemented in Phase2. The thread is detached from the main process, so that the main process does not need to wait for the termination of this thread. After receiving the socket variable as an argument from the thread creator, the thread frees the argument and starts listening to user inputs from the client using the socket as previously implemented in Phase2.

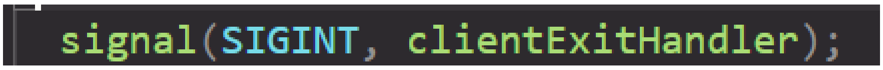


```
// Function that handles the client
void* HandleClient(void* arg)
{
    pthread_arg_t *pthread_arg = (pthread_arg_t *)arg;
    int socket = pthread_arg->new_socket_fd;
    free(arg);
    printf("handling new client in a thread using socket: %d\n", socket);
```

Fixing CTRL-C not working



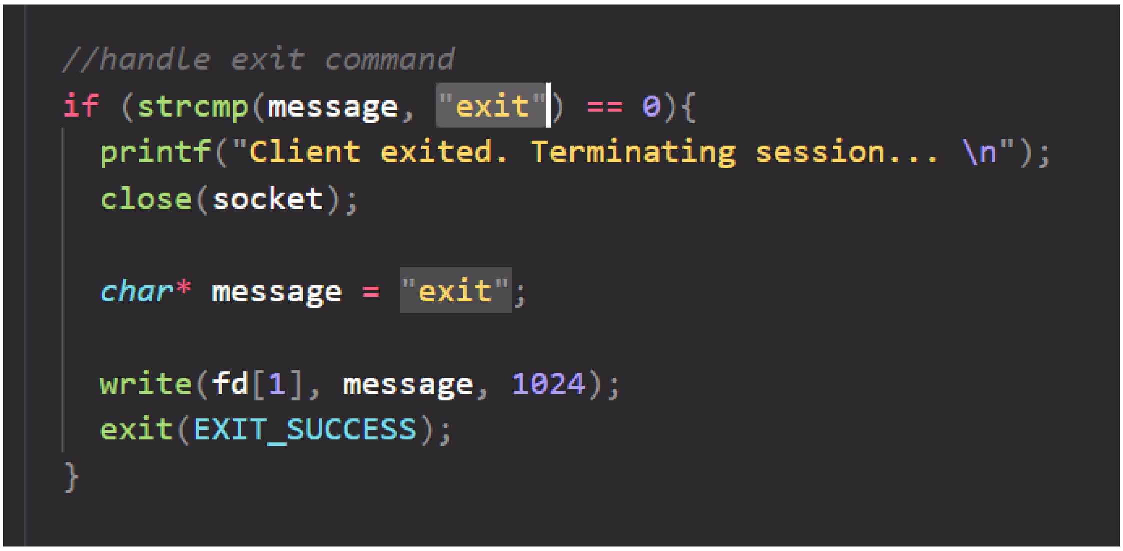
```
void clientExitHandler(int sig_num)
{
    send(sock,"exit",strlen("exit"),0); // sending exit message to server
    close(sock); // close the socket/end the conection
    printf("Exiting client.  \n");
    fflush(stdout);// force to flush any data in buffers to the file descri
    exit(0);
}
```



```
    signal(SIGINT, clientExitHandler);
```

In the client, we listen to the SIGINT command (CTRL+C) from the user. If the user tries to exit the client using CTRL-C, we catch that and gracefully exit by sending one last message to the server, “exit_client” which signals the server to terminate this client session.

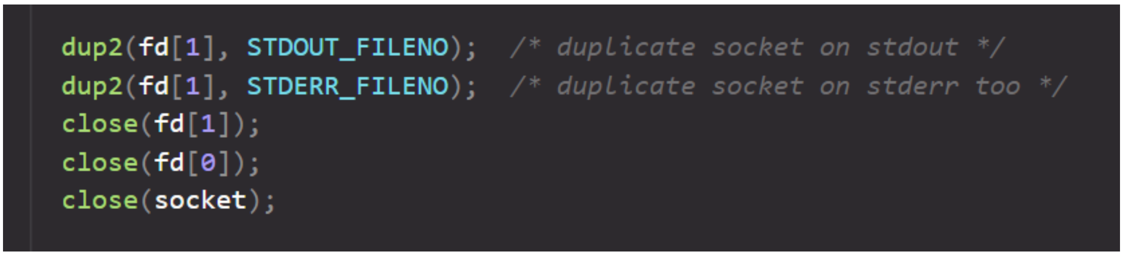
On the server:



```
//handle exit command
if (strcmp(message, "exit") == 0){
  printf("Client exited. Terminating session... \n");
  close(socket);

  char* message = "exit";

  write(fd[1], message, 1024);
  exit(EXIT_SUCCESS);
}
```

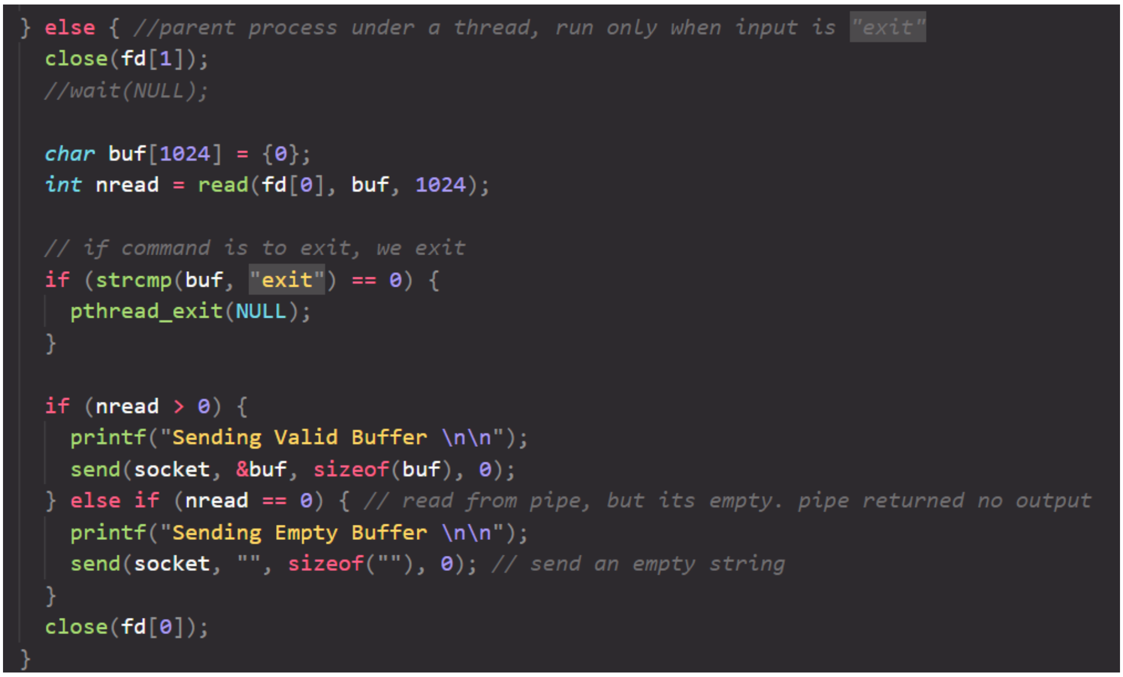
Fixing commands with no output

In the previous versions of the shell, the ‘execvp’ functions were piped directly to the socket. This caused some issues when the execvp functions was one that did not return anything, as the client expects a response message for every command it sends, as it is listening with recvp(). How do we send an empty message if the command has no output?



```
dup2(fd[1], STDOUT_FILENO);  /* duplicate socket on stdout */
dup2(fd[1], STDERR_FILENO);  /* duplicate socket on stderr too */
close(fd[1]);
close(fd[0]);
close(socket);
```

We fixed the problem by creating a new pipe and piped the return values from execvp to it. Thus, the output of execvp is captured in our pipe, fd. For every command to be run in the server, we fork the HandleClient thread, and run the execvp command in the child process. In our child process, we call execvp to get an output in the FD pipe. Then in the parent process, we read from FD to determine if there is an output or not.

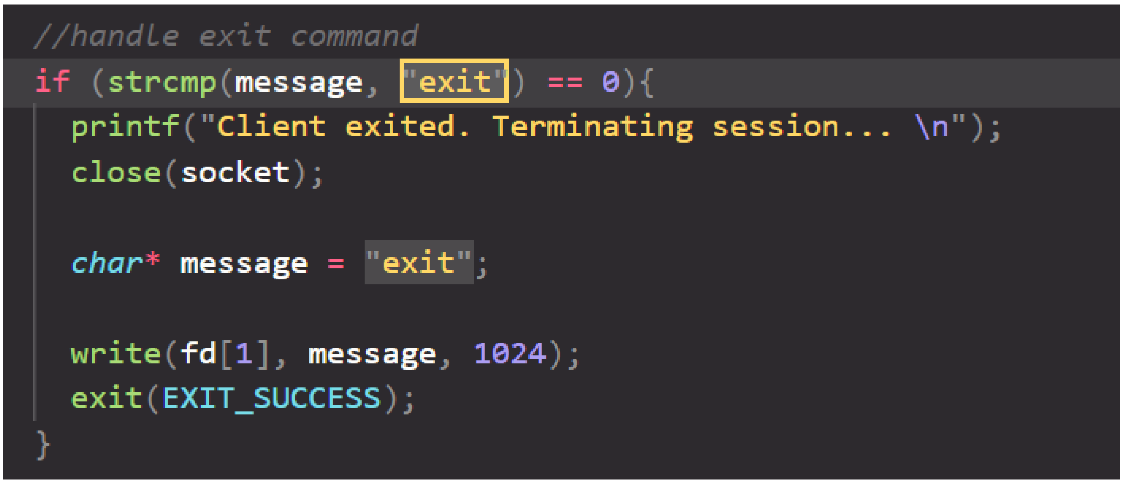


```
} else { //parent process under a thread, run only when input is "exit"
  close(fd[1]);
  //wait(NULL);

  char buf[1024] = {0};
  int nread = read(fd[0], buf, 1024);

  // if command is to exit, we exit
  if (strcmp(buf, "exit") == 0) {
    pthread_exit(NULL);
  }

  if (nread > 0) {
    printf("Sending Valid Buffer \n\n");
    send(socket, &buf, sizeof(buf), 0);
  } else if (nread == 0) { // read from pipe, but its empty. pipe returned no output
    printf("Sending Empty Buffer \n\n");
    send(socket, "", sizeof(""), 0); // send an empty string
  }
  close(fd[0]);
}
```

As the pipes are properly closed in the child process, the read() function will return “0” if the pipe is empty. It will return an integer > 0 if the read was successful, and the pipe contained data. Thus, if the pipe contains a message, we send that message back to the client. If it does not, that meant that the command has no output, e.g “grep ‘zzzzzzzzzzzz’ when there are no files named ‘zzzzzzzzzzz’. In that case, we simply send an empty buffer so that the client’s recv() waiting call will receive a response and to signal that the client can proceed to send the next command.

Terminating the thread after user exit

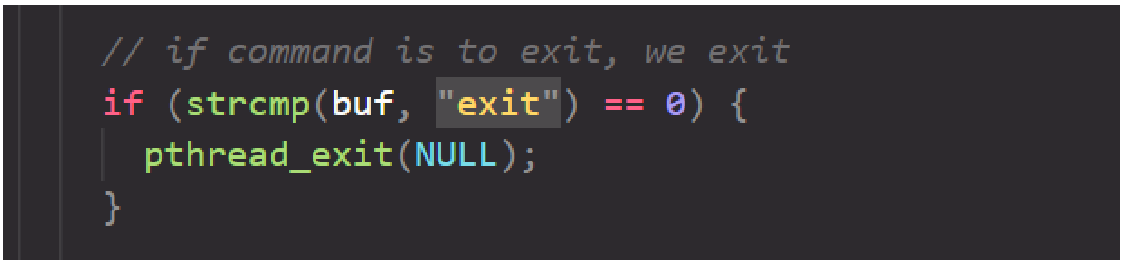
After the client exits for any reason, be it the client sending the “exit” command or SIGINT (CTRL+C) signal, the server will process it and terminate the thread running the session for that particular client.



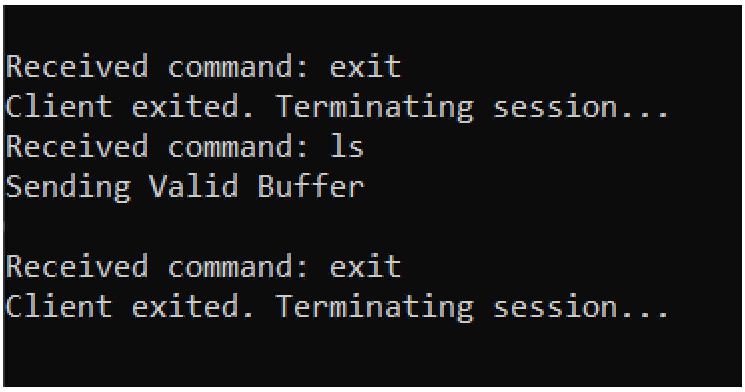
```
//handle exit command
if (strcmp(message, "exit") == 0){
  printf("Client exited. Terminating session... \n");
  close(socket);

  char* message = "exit";

  write(fd[1], message, 1024);
  exit(EXIT_SUCCESS);
}
```

As there is 2 processes in the thread for every command, (The child, running the execvp and the parent processing the results from the execvp), we have to terminate the thread in the parent process instead of the child process. We write a message to our pipe in the child process, so that the parent process will receive an “exit” message.

In the parent process, we simply pthread_exit() if we receive an “exit” message in the FD pipe. Thus, the thread is terminated, but other threads and other clients being served from the server will keep running as the thread is detached.



```
    // if command is to exit, we exit
    if (strcmp(buf, "exit") == 0) {
      pthread_exit(NULL);
    }
```



```
Received command: exit
Client exited. Terminating session...
Received command: ls
Sending Valid Buffer

Received command: exit
Client exited. Terminating session...
```

Group 13 – Phase 2: Remote CLI Shell

Jun & Sonya

Usage of the program

We upgraded Phase1 with socket communication. The client takes inputs from the user and sends requests to the server. The client will then receive the output/result from the server and print it on the screen.

To run the client/server

1) Open the directory in TWO separate UNIX-based terminals
2) In one of the terminals, run the server by running the command “gcc Phase2-server.c -o Phase2-server.o && ./Phase2-server.o”
3) In the other terminal, run the client by running the command “gcc Phase2-client.c -o Phase2-client.o && ./Phase2-client.o”
4) The order is important! The server should be run first before the client, otherwise the client will report “Connection error” as there is no server to connect to.
5) Once the connection is created, the client terminal will report the successful connection. You can start entering commands listed on the CLI to send to the server, and the server will run the commands before sending back the output of the commands to the client.
6) If the input commands are not among the listed ones, then the client will get an error message specifying “commands are not available”.
7) To exit the program, input “exit” on the client end. Do not enter ctrl+c on the client because it will not close the server properly. **Example test cases and results** valid test cases

--- *3 pipes* ---

```
"cat words.txt | grep yasin | tee output1.txt | wc -l"

  6

"cat words.txt | uniq | sort | head -10"

  Android

  Banana

  Cat
```

```
    Coffee

    Edamame

    Hello

    Hersheys

    Mcdonalds

    Starbucks

    Sun
```

```
  "sort alphabets.txt | head -10 | tail -n 5 | tee
    output3.txt" f
    g
    h
    i

    k
```

--- *2 pipes* ---

```
  "sort words.txt | head -10 | grep 'a'"

    Banana

    Cat

    Edamame

    Mcdonalds

    Starbucks

 "cat words.txt | grep yasin | wc -l"

    6
```

--- *1 pipe* ---

```
  "cat alphabets.txt | tail -10"

    l
```

```
w

q

r

z

f

u
g
n
h
```

"cat words.txt | uniq"

```
Hello

Mcdo
nalds
Coffe
e
yasin
hey
Starbu
cks
Banan
a yasin
Wat
er
Test
fish
yasi
n yo
brea
d
pota
to
Cat
Eda
ma
me
yasin
more
```

```
Sun
burg
er
Hersheys

Swimming

Android
```

"df | tee disk_usage.txt"

```
Filesystem      1K-blocks       Used Available Use% Mounted
on rootfs         249467900 217039252  32428648  88% / none
249467900   217039252       32428648       88%    /dev    none
249467900   217039252       32428648       88%    /run    none
249467900  217039252     32428648     88%   /run/lock   none
249467900  217039252     32428648     88%   /run/shm   none
249467900  217039252     32428648     88%   /run/user  tmpfs
249467900 217039252  32428648  88% /sys/fs/cgroup
C:\            249467900 217039252  32428648  88% /mnt/c

D:\            249470972  46119964 203351008  19% /mnt/d
```

--- *0 pipes* ---

"cat alphabets.txt"

```
b
d
k
e
m

a
c
p
i
t
y

l

w

q
```

```
   r

   z

   f

   u
   g
   n
   h
"ls -l"

   -rwxrwxrwx 1 yaya1721 yaya1721  2550 Apr  5 18:32 Phase2-client.c

   -rwxrwxrwx 1 yaya1721 yaya1721 17328 Apr  4 23:17 Phase2-client.o

   -rwxrwxrwx 1 yaya1721 yaya1721 15541 Apr  5 18:52 Phase2-server.c

   -rwxrwxrwx 1 yaya1721 yaya1721 22048 Apr  4 23:17 Phase2-server.o

   -rwxrwxrwx 1 yaya1721 yaya1721 12665 Apr  4 23:17 Phase2Report.docx

   -rwxrwxrwx 1 yaya1721 yaya1721 13024 Apr  5 18:51 a.out

   -rwxrwxrwx 1 yaya1721 yaya1721    63 Apr  3 11:25 alphabets.txt

   -rwxrwxrwx 1 yaya1721 yaya1721   581 Apr  6 02:40 disk_usage.txt

   -rwxrwxrwx 1 yaya1721 yaya1721    54 Apr  6 02:37 output1.txt

   -rwxrwxrwx 1 yaya1721 yaya1721    15 Apr  6 02:38 output3.txt

   -rwxrwxrwx 1 yaya1721 yaya1721   189 Apr  3 11:25 words.txt

"man"

   What manual page do you want?

"pwd"

   /mnt/c/Users/Sonya/Documents/GitHub/OS-Project/Phase2

"echo
   hello"
   hello

"ps"
```

```
    PID TTY          TIME CMD

      9 tty1     00:00:01 bash

    27334 tty1     00:00:00 a.out

    27433 tty1     00:00:00 ps

  "whoami"
    yaya1721
  "exit"

    Closing socket from clientside

    Exited
```

invalid test cases

```
  "cd"

Command is currently unavailable, change one...
```

Group 13 – Phase 1

Jun & Sonya

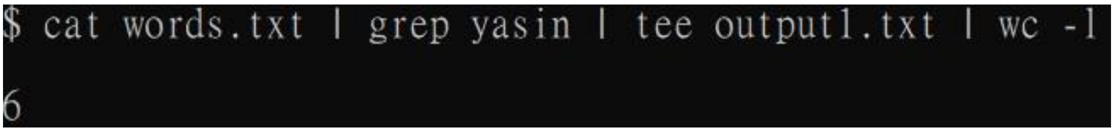
Usage of the program

We developed our own shell in C that replicates feature from the Linux commands or a program to execute including its name. We implemented single and composed commands using 0 to 3 pipes.

Example test cases and results

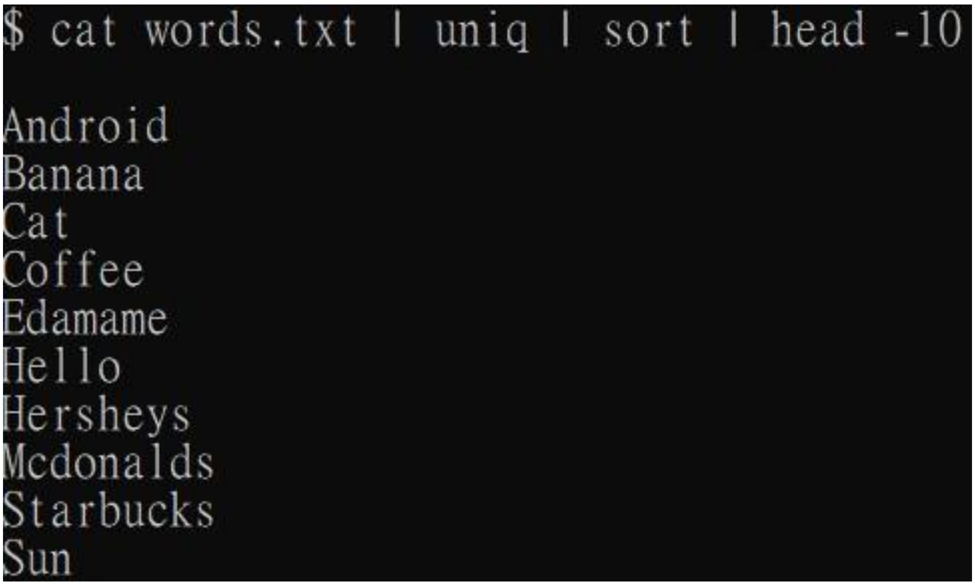
--- *3 pipes* ---

cat words.txt | grep yasin | tee output1.txt | wc -l



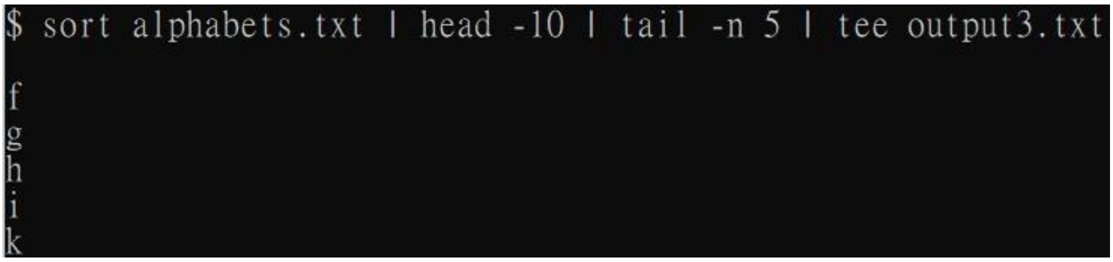
```
$ cat words.txt | grep yasin | tee output1.txt | wc -l

6
```

cat words.txt | uniq | sort | head -10



```
$ cat words.txt | uniq | sort | head -10

Android
Banana
Cat
Coffee
Edamame
Hello
Hersheys
Mcdonalds
Starbucks
Sun
```

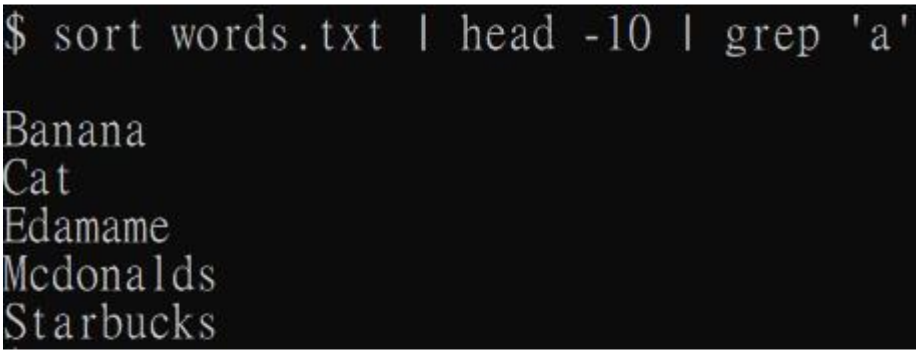
sort alphabets.txt | head -10 | tail -n 5 | tee output3.txt



```
$ sort alphabets.txt | head -10 | tail -n 5 | tee output3.txt

f
g
h
i
k
```

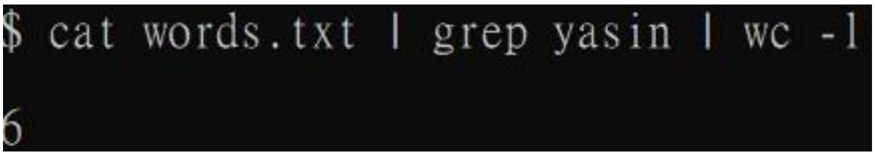
--- *2 pipes* ---

sort words.txt | head -10 | grep 'a'



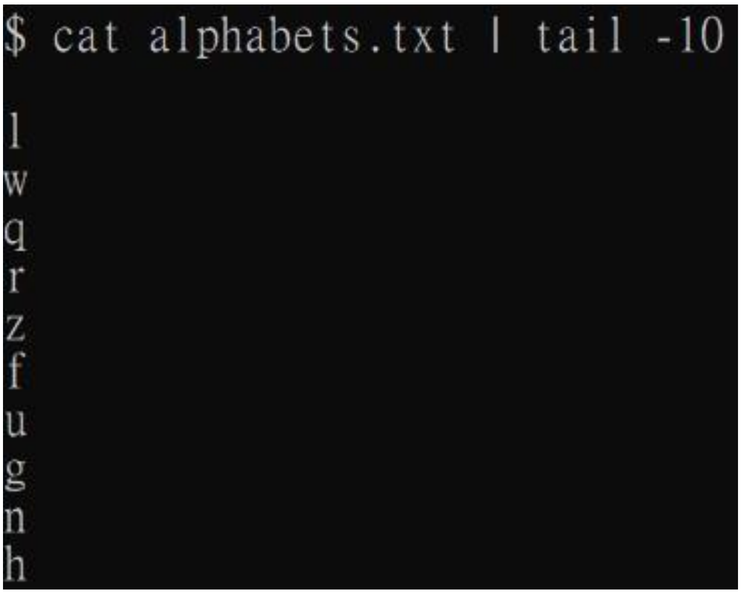
```
$ sort words.txt | head -10 | grep 'a'

Banana
Cat
Edamame
Mcdonalds
Starbucks
```

cat words.txt | grep yasin | wc -l



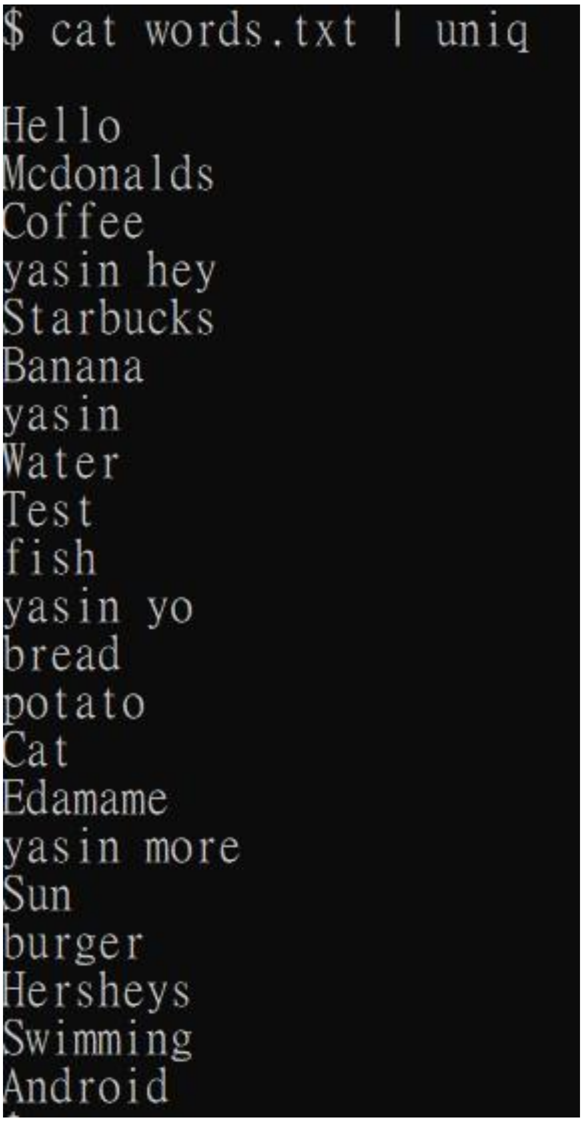
```
$ cat words.txt | grep yasin | wc -l

6
```

--- *1 pipe* --- cat alphabets.txt | tail -10



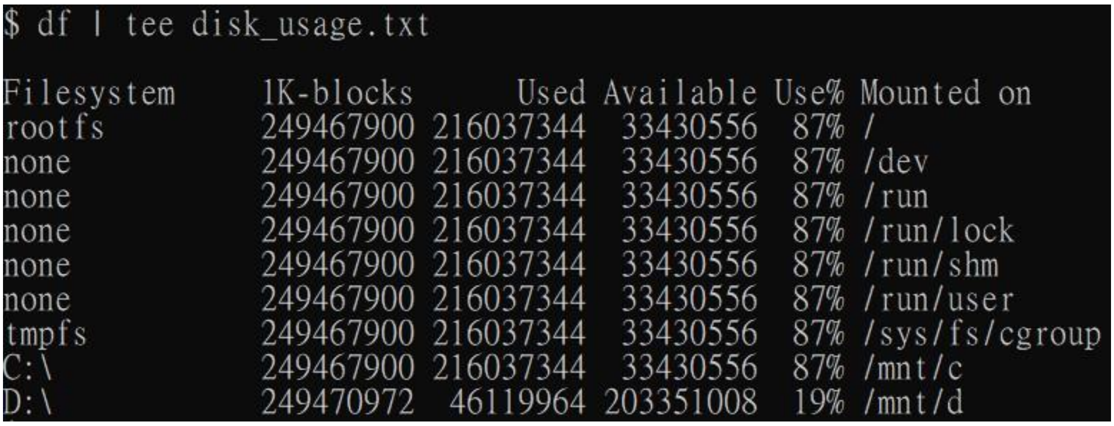
```
$ cat alphabets.txt | tail -10

l
w
q
r
z
f
u
g
n
h
```

cat words.txt | uniq



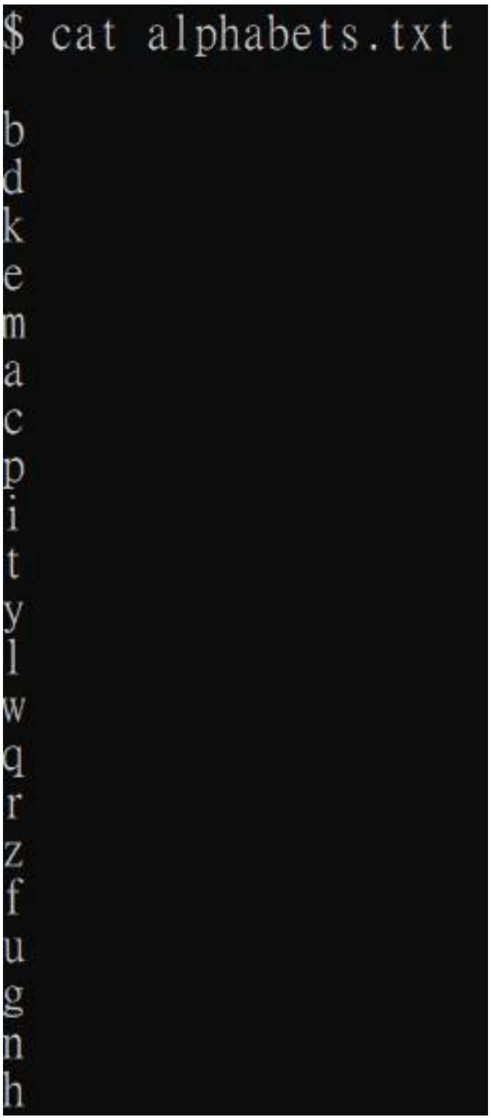
```
$ cat words.txt | uniq

Hello
Mcdonalds
Coffee
yasin hey
Starbucks
Banana
yasin
Water
Test
fish
yasin yo
bread
potato
Cat
Edamame
yasin more
Sun
burger
Hersheys
Swimming
Android
```

df | tee disk_usage.txt



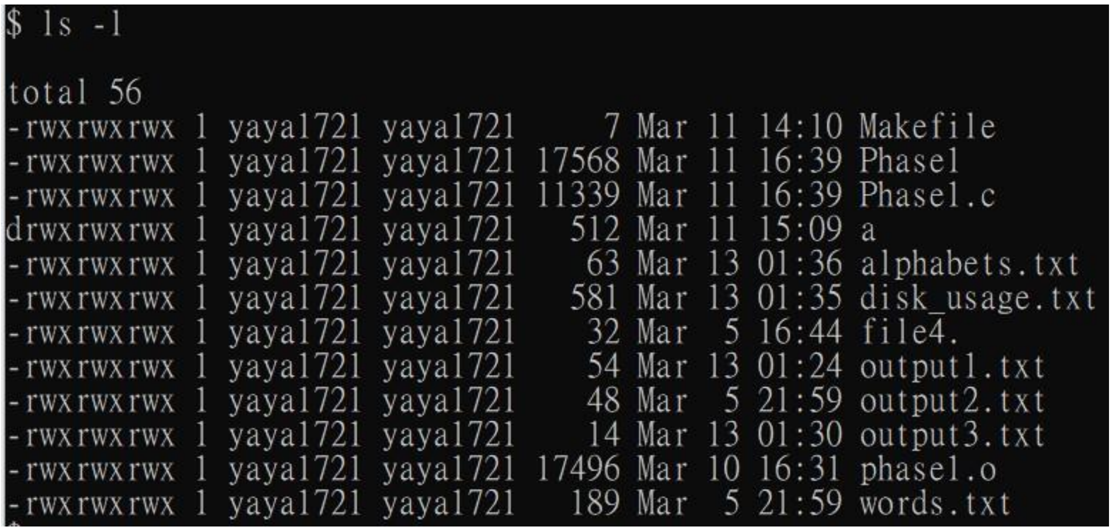
```
$ df | tee disk_usage.txt

Filesystem      1K-blocks      Used Available Use% Mounted on
rootfs          249467900 216037344  33430556  87% /
none            249467900 216037344  33430556  87% /dev
none            249467900 216037344  33430556  87% /run
none            249467900 216037344  33430556  87% /run/lock
none            249467900 216037344  33430556  87% /run/shm
none            249467900 216037344  33430556  87% /run/user
tmpfs           249467900 216037344  33430556  87% /sys/fs/cgroup
C:\             249467900 216037344  33430556  87% /mnt/c
D:\             249470972  46119964 203351008  19% /mnt/d
```

--- *0 pipes* --- cat
alphabets.txt



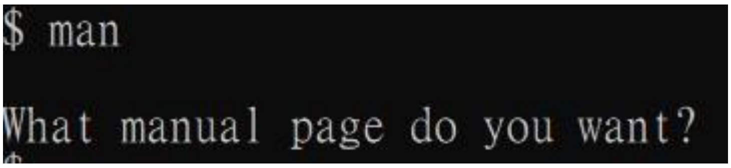
```
$ cat alphabets.txt

b
d
k
e
m
a
c
p
i
t
y
l
w
q
r
z
f
u
g
n
h
```

ls -l



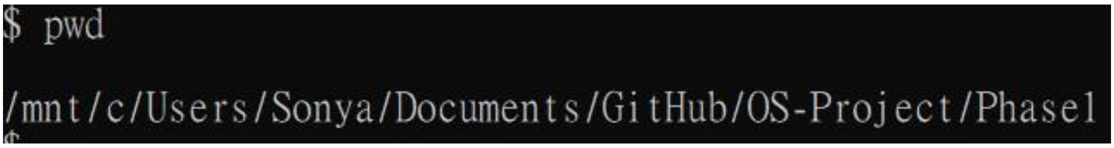
```
$ ls -l

total 56
-rwxrwxrwx 1 yaya1721 yaya1721     7 Mar 11 14:10 Makefile
-rwxrwxrwx 1 yaya1721 yaya1721 17568 Mar 11 16:39 Phase1
-rwxrwxrwx 1 yaya1721 yaya1721 11339 Mar 11 16:39 Phase1.c
drwxrwxrwx 1 yaya1721 yaya1721   512 Mar 11 15:09 a
-rwxrwxrwx 1 yaya1721 yaya1721    63 Mar 13 01:36 alphabets.txt
-rwxrwxrwx 1 yaya1721 yaya1721   581 Mar 13 01:35 disk_usage.txt
-rwxrwxrwx 1 yaya1721 yaya1721    32 Mar  5 16:44 file4.
-rwxrwxrwx 1 yaya1721 yaya1721    54 Mar 13 01:24 output1.txt
-rwxrwxrwx 1 yaya1721 yaya1721    48 Mar  5 21:59 output2.txt
-rwxrwxrwx 1 yaya1721 yaya1721    14 Mar 13 01:30 output3.txt
-rwxrwxrwx 1 yaya1721 yaya1721 17496 Mar 10 16:31 phase1.o
-rwxrwxrwx 1 yaya1721 yaya1721   189 Mar  5 21:59 words.txt
```

man



```
$ man

What manual page do you want?
```

pwd

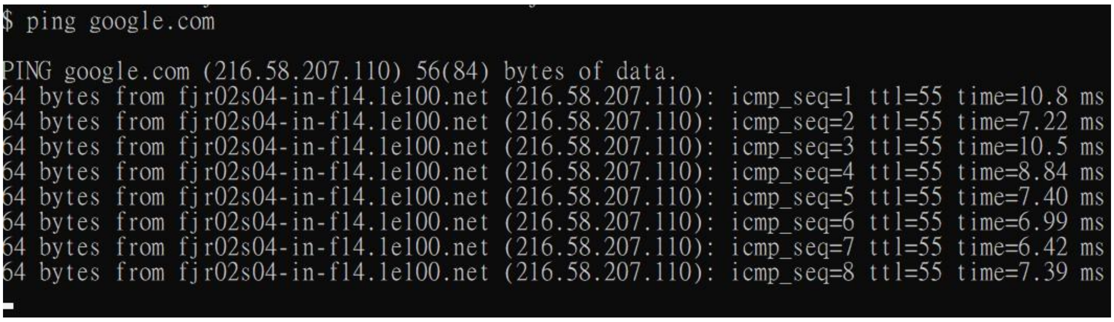


```
$ pwd

/mnt/c/Users/Sonya/Documents/GitHub/OS-Project/Phase1
```

touch dummy.txt

Create a file in the same directory called dummy.txt rm
dummy.txt

Remove the file, dummy.txt ping
google.com



```
$ ping google.com

PING google.com (216.58.207.110) 56(84) bytes of data.
64 bytes from fjr02s04-in-f14.1e100.net (216.58.207.110): icmp_seq=1 ttl=55 time=10.8 ms
64 bytes from fjr02s04-in-f14.1e100.net (216.58.207.110): icmp_seq=2 ttl=55 time=7.22 ms
64 bytes from fjr02s04-in-f14.1e100.net (216.58.207.110): icmp_seq=3 ttl=55 time=10.5 ms
64 bytes from fjr02s04-in-f14.1e100.net (216.58.207.110): icmp_seq=4 ttl=55 time=8.84 ms
64 bytes from fjr02s04-in-f14.1e100.net (216.58.207.110): icmp_seq=5 ttl=55 time=7.40 ms
64 bytes from fjr02s04-in-f14.1e100.net (216.58.207.110): icmp_seq=6 ttl=55 time=6.99 ms
64 bytes from fjr02s04-in-f14.1e100.net (216.58.207.110): icmp_seq=7 ttl=55 time=6.42 ms
64 bytes from fjr02s04-in-f14.1e100.net (216.58.207.110): icmp_seq=8 ttl=55 time=7.39 ms
```

Description of implementation

We developed our own shell in C programming language. After parsing the input from the command line, we determined how many pipes should be used in each command and called the corresponding function. Within each function, we forked child process and created pipes. Parent process should execute the last command after its child process executed the previous commands if the number of commands is larger than one. User is able to exit the program by inputting “exit” on the command line interface.