**CS2106 Operating Systems**

**Lab 2 – Fork and Pipe Answer Sheet**

|  |  |
| --- | --- |
| Student ID 1:  A0161868R | Name 1:  Joshua Lee Kai Sheng |
| Student ID 2: | Name 2: |
| Student ID 3: | Name 3 : |

**Question 1.** (5 marks)

Parent’s process ID: **25381**

Child’s process ID: **25382**

Child’s parent’s process ID (as reported by child): **1090**

Parent’s parent’s process ID (as reported by parent): **25354**

*\*\* the PID varies for different runs*

Parent of the parent: **bash**

**Question 2.** (3 marks)

The web browser cannot connect to the web server because …

**When deliverHTTP is called, it will try to read in values from the connfd. However, there is no input to the connection established using telnet. Hence, the execution does not proceed past the read method (blocked). As such the deliverHTTP for the first connection is not fully executed when the Mozilla browser tries to establish another connection. And deliverHTTP for this second connection is not executed (only support one connection at a time).**

**Question 3a.** (5 marks)

The modifications I made, and my explanations for them are:

while(1) {  
 connfd = accept(listenfd, (struct sockaddr \*) NULL, NULL);  
 int result = fork();  
  
 if(result != 0){  
 //parent will write to log and move on to the next loop, ready for next connection  
 writeLog("Connection received.");  
  
 } else {  
 //child will execute deliverHTTP will might be blocked by read()  
 deliverHTTP(connfd);  
 }  
}

**Inside the while loop of startServer(), create a fork after the accept function, which establishes a connection with a client.**

**The parent process will write to log and proceed on to the next loop in order to be ready for connection with the next client, regardless of whether deliverHTTP() function in the child process is blocked.**

**The child process will handle the deliverHTTP() function.**

**Question 3b.** (7 marks)

The maximum number of connections I can make is:

**The maximum number of processes that a LINUX system can execute at the same time. For a particular user, this value can be found by using the “ulimit -u” command. On my system, the max user processes is 7819 and this is also the maximum number of simultaneous connections I can make. This is because each connection will take up a process, because of the added fork.**

*\*\* The maximum number of connections is not limited by the backlog value in the listen method. This is because the backlog value limits the queue size for pending accept requests from clients. Once accepted, these pending requests are removed from the queue. Hence, if connections are continuously accepted, the number of connections that I can make is not limited by the queue size.*

**Question 4a.** (3 marks)

We must close the ends of the pipe we are not using because...

**We want the pipe to provide single directional communication.**

**We first close the read descriptor (input end) in the parent process as we only want the parent to write to the pipe in order not to waste file descriptors and to detect possible dying reader in the child process.**

**We first close the write descriptor (output end) in the child process so that the child can detect the EOF condition and read from the pipe.**

**Question 4b.** (5 marks)

This statement is false because…

**The child and parent processes, after forking, do not share the same data (memory space, arguments, environment variables) as they are in different processes and hence will have different contexts.**

**Hence, the two processes do not have access to the same buffer array. They have, in fact, different copies of buffer arrays.**

**Question 5.** (7 marks)

The modifications I made, with explanation, are:

//global variable for file descriptor   
int fd[2];  
  
int main(int ac, char \*\*av)  
{  
 pipe(fd);//setup the pipe  
  
 if (fork() != 0) {  
  
 int status; //parent will write to the pipe  
  
 close(fd[0]); //close the input end of pipe before writing to pipe  
 startServer(PORTNUM);  
  
 close(fd[1]);  
 wait(&status);  
 } else {  
  
 char buffer[LOG\_BUFFER\_LEN]; //create a buffer  
 FILE\* fptr; //fptr points to the log.txt file  
 fptr = fopen("log.txt", "wb"); //first open the log.txt to clear any previous log  
 fclose(fptr); //close file pointer  
 close(fd[1]); //close output end of pipe before reading  
 int n;  
  
 while (1) { //infinite loop to try to read from pipe  
  
 if ((n = read(fd[0], buffer, LOG\_BUFFER\_LEN)) > 0) {  
 //read from the pipe into buffer  
 //and if pipe is not empty  
 fptr = fopen("log.txt", "a"); //open the log.txt file in append mode  
 fwrite(buffer, 1, n, fptr); //write from the buffer to log.txt  
 fclose(fptr); //close the fptr  
 }  
  
 }  
 close(fd[0]);  
 exit(1);  
 }  
}  
  
void writeLog(const char \*format, ...)  
{  
 char logBuffer[LOG\_BUFFER\_LEN];  
 va\_list args;  
  
 sprintf(logBuffer, "%s: ", getCurrentTime()); //include current time in log  
 va\_start(args, format);  
 vsprintf(logBuffer + strlen(logBuffer), format, args); //append message to log  
 va\_end(args);  
 sprintf(logBuffer + strlen(logBuffer), "\n"); //append a new line character at the end  
  
 write(fd[1], logBuffer, strlen(logBuffer)); //write from buffer to the pipe  
  
}

**Create a fork in the main method where the parent will be responsible calling writeLog() which will write the logs to the pipe.**

**The child process, whereas, will be responsible for reading from the pipe and write the log read to the log.txt. The while loop in the child process ensures that the writing to log file process is executed whenever the pipe is not empty.**

**The file descriptors are closed accordingly at the appropriate places.**