Simulated Network-Centric Programming
Mid-Term Exam

Instructions

1. Please use your computer or lab's desktop to complete the Midterm problems. You are NOT allowed to access any Internet resources or in-class code examples during the exam, except for submitting your final answers.
2. Provide the solution to each problem in source file or files.
3. Provide *Makefile* for each programming problem and include a *README* file to describe how to compile/test the program.

Submission Guidelines

- Create a directory named <username>-midterm (replace <username> with your actual username).
- Inside this directory, create subdirectories for each question (e.g., q1, q2, ...,).
- Place all related files (source code, *Makefile*, *README*) inside the corresponding subdirectory.
- Create a tar archive of your directory by using:

```
tar czf <username>-midterm.tar.gz <username>-midterm
```

- Upload `username-midterm.tar.gz` to Canvas.

Questions

Q1 (10 points) In UNIX, file operations can be performed using Standard I/O functions (fopen(), fread(), fwrite()) or System Call I/O functions (open(), read(), write()).

- What is the key difference between Standard I/O and System Call I/O?
- Which one generally provides better performance, and why?
- When would it be better to use system calls instead of Standard I/O?

Q2 (10 points): You have learned about IPv4 vs. IPv6 and IP/host conversions.

- What is the difference between IPv4 and IPv6 in terms of address format?
- What is the purpose of getaddrinfo() and how does it help in IP/host conversion?
- Why is getaddrinfo() preferred over gethostbyname() in modern network programming?.

Q3 (10 points): In the TCP client-server model,

- What are the roles of bind(), listen(), and accept() in a TCP server?
- Why does a TCP client need to call connect() before sending data?
- What happens if a server does not call close() on a client socket after communication is complete?

Q4 (10 points): Process Management – fork() and waitpid()

- What is use of fork() system call?
- Why programs check the return values of fork()?
- What happens if a parent process does not call wait() and the child process terminates?

Q5 (10 points): Threads vs. Processes & Synchronization

- What is the key difference between multithreading and multiprocessing?
- What is a race condition, and how can it occur in a multi-threaded program?
- How does a mutex (pthread_mutex_t) help prevent race conditions?

Q6 (10 points): Write a C program to display all factors of an integer number.

- Interactively read an integer from terminal and display its factors
- Check error conditions (e.g., use enter negative number). If error, print error message and exit.

Q7 (10 points): Write a C program that takes a non-negative integer exponent n from command line as a command-line argument and computes 2^n

- If n is negative, print an error message and exit.
- If the result overflows a 32-bit signed integer (int), detect it and print an error message.
- Otherwise, print the result.

Q8 (10 points): Implement a C program to perform the following tasks using standard I/O library:

1. Open a file (with filename specified in command line)
2. Read 3 lines of text input from terminal and write them to the file.
3. Find file size with fseek() and print
4. Read the content of the file and display.

Q9 (10 points): pre-action: create a text file using command with some text, e.g., echo "2025 midterm simulation" > test1.txt

Write a program to perform the following tasks using system call I/O:

1. Insert a string "<file_name> content:\n" at the begging of the file.
2. Continuously read text input from terminal and write them to the file (until user enter EOF/Ctrl+D).
3. Find file size of the file and print out.
4. Print all content of the file to terminal.

Q10 (20 points): Implement a program that performs the following tasks:

1. Open a file (with the filename specified in the command line).
2. Read text input from the terminal and append it to the file until the user enters EOF (Ctrl+D).
3. Find file size and save the value in variable *fsize*
4. Insert "Content-length: *fsize*\n" at the beginning of the file.
5. Read everything in the file and print out.

Your program must provide two versions of file I/O:

- One version using system calls (open, write, lseek, read, close etc.).
- Another version using standard I/O library functions (fopen, fgets, fprintf, fclose etc.).

If the command line includes the -s (or --system) option, use system calls. Otherwise, use standard I/O library.

Q11 (10 points): Implement a program that uses fork() to create a child process.
In the *parent process*, create a new process and print out parent pid and child pid. The parent should then wait for the child to exit and print the child's exit status (normal exit with code or abnormal exit).

In the child process, print its own pid and parent's pid. Ask use input for exit code: "Enter exit code for child process: ". If the user input is valid (0-255), child exit with the exit code. Otherwise, child process aborts.

If error occurs (like fork failed), report error using perror().

Q12 (10 points): Run a different program in the new process: Implement a program that run command "ls -l" in child process. Parent process creates a new process, and call waitpid() to wait for the child to exit and print the exit status of the child. Child process prints its own pid, sleep for 2 seconds and then run command "ls -l".

Handle error conditions.

Q13 (10 points): Write a program to pass information between parent and child processes. (calculate a^2-b^2)

1. Main process gets 2 integer numbers *a* and *b* from command line (both *a* and *b* are in [-100, 100]).
2. Parent creates two child processes.
3. The first child sleeps 1 second, calculates a^2, passes the result to parent, then exit.
4. The second child sleeps 1 second, calculates b^2, pass the result to parent, and then exit.
5. Parent minus the output of the second child from the output of the first child and print out the result.

❖ You can use any IPC methods (shared memory, pipe, FIFO, message queue) to pass values.

- ❖ Synchronize the processes if necessary.

Q14 (10 points): Implement a multi-threaded C program where the main thread creates a peer thread and exchanges data with it.

- The main thread:
  - Creates a peer thread, passing the string "How to create a thread?" as an argument.
  - Waits for the peer thread to finish.
  - Retrieves the returned string from the peer thread and prints: “Peer thread returns: <return_value>”
- The peer thread function:
  - Receives a string as input and prints: “Thread input: <input_string>”
  - Sleeps for 1 second.
  - Returns the string "Call pthread_create()" to the main thread.
- Thread synchronization: With synchronization to safely retrieve the return value from the peer thread.

Q15 (10 points): Implement a multi-threaded C program that uses 5 peer threads to update a shared counter.

The main thread:
- Defines a global counter initialized to 0.
- Creates 5 threads, each receiving an argument from 100 to 104.
- Waits for all peer threads to terminate.
- Prints the final counter value after all updates.

Each peer thread:
- Receives an integer n as an argument.
- Increments the shared counter exactly n times.

Thread Safety Requirement:
- Use a mutex to prevent race conditions.

Q16 (10 points): Implement a socket client that connects to a server by IP.

The client reads an IP address and a port number from command line. It should use the input IP and port number to set server address. Then it should connect to the address. Once connected, it sends “GET HTTP/1.1\r\nHost: <IP>:<Port>\r\n\r\n” to the server. It then reads and prints the response from the server.

Q17 (10 points): Implement a socket client that connects to a server by hostname.

The client reads a hostname (like www.example.com) from command line. It should use the input hostname to set up server address. Then it should connect to the address. Once connected, it sends “GET HTTP/1.1\r\nHost: <IP>:<Port>\r\n\r\n” to the server. It then reads and prints the response from the server.

Q18 (10 points): Implement a dummy http server.

The server listens on port 8080. When a new client connection comes, it reads and prints the client request. It then sends a dummy response: “HTTP/1.1 200 OK\r\nContent-Type: text/html\r\n\r\n<HTML><BODY>Dummy Web Server </BODY></HTML>\r\n". It then closes the connection.

Q19 (20 points): Implement a basic TCP socket server and client using localhost and port 8088.

Server will listen on the socket, accepts a client connection, and receive a message from the client. It will print the received message, wait for 2 seconds and send response “Response from server”. After closing the connection, it waits for the next client.

Client connects to the server and sends the message: "Request from client". It then reads response from server and print the message.
If errors occur, output error messages. For example, if a socket creation fails, use *perror("Socket Creation Failed");* to display the error message.

Q20 (20 point): Implement concurrent client and server using multi-processes with port 8088 and localhost.

The client will use 2 processes, client 1 and client 2, to open two connections to the server at the same time. Each process handles one connection, sleeps for 2 seconds. The first process sends “Hello from client 1” and the second process sends “Hello from client 2”. Each process then reads response from the server. If the response to client 1 is not ended with “client 1”, client 1 reports error. If the response to client 2 is not ended with "client 2”, client 2 reports error.

The server creates a new process for each new client connection. The child process will read the request message from the socket. It then parses the message and replaces “Hello from” with “Hello to”. It then sends the updated message back to the client. The server should free resources that are no longer needed.

Q21 (20 point): Implement concurrent client and server using multithreads with port 8088 and localhost.

The client will use 2 peer threads, client 1 and client 2, to open two connections to the server at the same time. Each thread handles one connection, sleeps for 2 seconds. The first thread sends “Hello from client 1” and the second thread sends “Hello from client 2”. Each thread then reads response from the server. If the response to client 1 is not ended with “client 1”, client 1 reports error. If the response to client 2 is not ended with "client 2”, client 2 reports error.

The server creates a new thread for each new client connection. The peer thread will read the request message from the socket. It then parses the message and replaces “Hello from” with “Hello to”. It then sends the updated message back to the client. The server should free resources that are no longer needed.

- If multiple threads access a shared resource, ensure that each thread receives a separate copy, or protect it using mutex locks (pthread_mutex_t).