**고급프로그래밍 설계과제**



**IPC**

|  |  |  |
| --- | --- | --- |
| **12조** | | |
| 이름 | 학번 | 담당 PC기법 |
| **이길형** | **2017150048** | **Named pipe** |
| **정수경** | **2017156037** | **Shared Memory** |
| **정하림** | **2017152049** | **Message Passing** |
| **제출 날짜 : 2019.12.09** | | |

목차

1. 문제정의 및 요구사항 분석 4

1.1 문제 정의 및 설계과제 목표 4

1.2 요구사항 분석 4

2. 시스템 설계 6

2.1 Named Pipe 6

2.1.1 **소프트웨어 설계도** 6

**2.1.2** **쓰레드 설계** 7

2.2 Shared Memory 8

**2.2.1** **소프트웨어 설계도** 8

**2.2.2** **쓰레드 설계** 9

2.3 Message Passing 11

**2.3.1** **소프트웨어 설계도** 11

**2.3.2** **쓰레드 설계** 12

3. 시스템 구현 및 데모 13

3.1 소스코드 주요부분 설명 13

**3.1.1** **Named Pipe** 13

**3.1.2** **Shared Memory** 20

**3.1.3** **Message Passing** 36

3.2 데모 시나리오 43

3.3 프로그램 실행결과 캡쳐 및 설명 43

3.4 현실적 제한요소 반영 45

**3.4.1** **경쟁상태** 46

**3.4.2** **성능을 높일 수 있는 방안** 47

**3.4.3** **메모리 사용량을 줄일 수 있는 방안** 47

**3.4.4** **UNIX API 활용** 48

**3.4.5** **Secure coding 기법 활용** 50

4. 팀워크 52

4.1 역할 및 작업내용 52

5. 결론 및 총평 52

5.1 정수경 52

5.2 정하림 53

5.3 이길형 53

6. 부록 54

6.1 코드 54

**6.1.1** **Named Pipe** 54

**6.1.2** **Shared Memory** 62

**6.1.3** **Message Passing** 76

# 문제정의 및 요구사항 분석

## **문제 정의 및 설계과제 목표**

프로세스간 통신(IPC: Inter Process Communication) 기법을 달리하여 채팅 기능을 베이스로 하고 인물 맞추기 게임을 추가적으로 구현한다. 사용하는 IPC 기법으로는 Message Passing, Shared Memory, Named pipe를 이용하여 서버와 클라이언트 구조의 프로그램을 구현하고 병행적으로 프로그램이 실행될 수 있도록 쓰레드를 이용한다. 프로그램 내에서 각 필요한 기능들이 병행적으로 실행될 수 있게 프로그램이 실행되면 자동적으로 생성되는 메인 쓰레드를 제외한 2개 이상의 쓰레드를 생성하여 총 3개 이상의 쓰레드를 구현한다. 프로그램 동작 구조는 서버를 하나로 설정하고 클라이언트는 2명으로 하여 일대일 채팅으로 설정한다. 각 클라이언트는 서로 간 메시지를 입력하고 서버에 메시지를 보내 서버는 각 클라이언트에게 메시지를 전송한다.   
 IPC 기법마다 프로그램 설계가 다르기 때문에 수행하는 기능에 따라 사용하는 프로세스 간 통신 기법에 맞춰 필요한 쓰레드를 생성하고 병행적으로 프로그램이 실행되도록 한다. 이 때 쓰레드가 공유자원에 동시에 접근하는 경쟁상태(race condition)가 발생될 수 있으므로 프로그램 구현할 때 이러한 경쟁상태가 발생되지 않게 세마포어나 뮤텍스와 같은 동기화 기법(synchronization tool)을 이용하여 경쟁상태가 발생되지 않도록 한다.  
 각자 서로 다른 IPC 기법을 이용해 서버와 클라이언트 통신구조를 완성하고 서버와 클라이언트 간에 데이터를 주고받을 때 걸리는 시간을 측정하여 IPC 기법마다 성능 차이를 분석한다.  
 설계 환경은 pthread 관련 함수를 사용하기 때문에 POSIX 라이브러리를 포함하고 있는 Linux 기반 운영체제인 우분투(Ubuntu)와 학교 리눅스 서버(computer.kpu.ac.kr)에서 실행될 수 있게 설정한다. 컴파일러는 Linux의 기본 컴파일러인 GCC를 이용하고 실행파일은 Make 유틸리티를 이용하여 생성한다. 그리고 코드를 작성할 때는 시큐어 코딩 기준에 맞춰서 코딩하도록 한다.  
 Advanced IPC 기법인 Shared Memory, Message Passing을 사용할 때, key 번호는 60130~60139를 사용하고 작성된 소스 파일, 헤더 파일, 최종 보고서, 발표 자료, 설계 제안서, readME.txt 파일을 포함하여 Linux 압축 포맷인 tar형식으로 압축하여 최종적으로 제출할 수 있도록 한다.

## 요구사항 분석

**조원이 3명인 경우 Advanced IPC 기법인 Shared Memory, Message passing, Pipe 세 가지를 구현한다.**

* 정수경(Shared Memory), 정하림(Message passing), 이길형(Named pipe) 세명에서 각자 IPC기법을 나눠서 프로그램을 작성한다.

**서버와 클라이언트 구조를 설계하고 클라이언트는 2개 이상 실행되도록 한다.**

* 하나의 서버와 2개의 클라이언트를 이용해 채팅 기능을 베이스로 하고, 이미지를 아스키코드(ASCII)로 변환하여 클라이언트들은 정답을 맞히고 정답인지 오답인지 판단하는 기능을 가진 서버를 구현한다.

**프로그램을 멀티 쓰레드를 사용해 병행적으로 실행할 수 있도록 한다.**

* POSIX 라이브러리를 이용해 pthread를 이용하여 멀티 쓰레드를 구현하고 프로그램이 병행적으로 작동될 수 있도록 한다.

**경쟁상태(Race condition)가 발생하지 않도록 한다.**

* 쓰레드가 같은 공유자원에 접근하게 될 경우, 동기화 도구(Synchronization tool)인 mutex와 semaphore 이용하여 병행적으로 실행되더라도 경쟁상태가 발생하지 않도록 한다.

**Advanced IPC 기법 사용시 Key값 설정 범위**

* computer.kpu.ac.kr 학교 리눅스 서버에서 프로그램 구동 테스트를 할 예정이기 때문에 key 값이 중복될 수 있다. 그렇기 때문에 13조인 우리 조는 60130 ~ 60139번 key 값을 사용하도록 한다.

**코드 작성시 Secure Coding 기법을 사용하도록 한다.**

* scanf와 같은 사용할 영역을 지정해주지 않는 함수 사용시 미할당 메모리 영역을 건드릴 수 있으므로 fgets와 같은 사이즈를 지정해주는 함수를 사용하고, 오류 처리를 하는 코딩기법을 이용해 시큐어 코딩기법을 적용한다.

**실행 파일 생성시 Linux Utility 인 Make를 이용한다.**

* 3명에서 작성한 소스코드와 헤더파일을 Make 유틸리티를 통해 실행파일을 생성하며, 명령어에 따라 다른 IPC 기법을 사용한 프로그램을 컴파일한다.

**오류 발생시 Linux Utility 인 GDB 디버거를 이용하여 디버깅을 시행한다.**

* 프로그램 소스코드 작성 중 원인을 알 수 없는 오류가 발생했을 경우 Linux 디버깅 툴인 GDB 디버거를 이용하여, 해당 코드의 문제와 오류의 원인을 파악하여 문제를 해결하도록 한다.

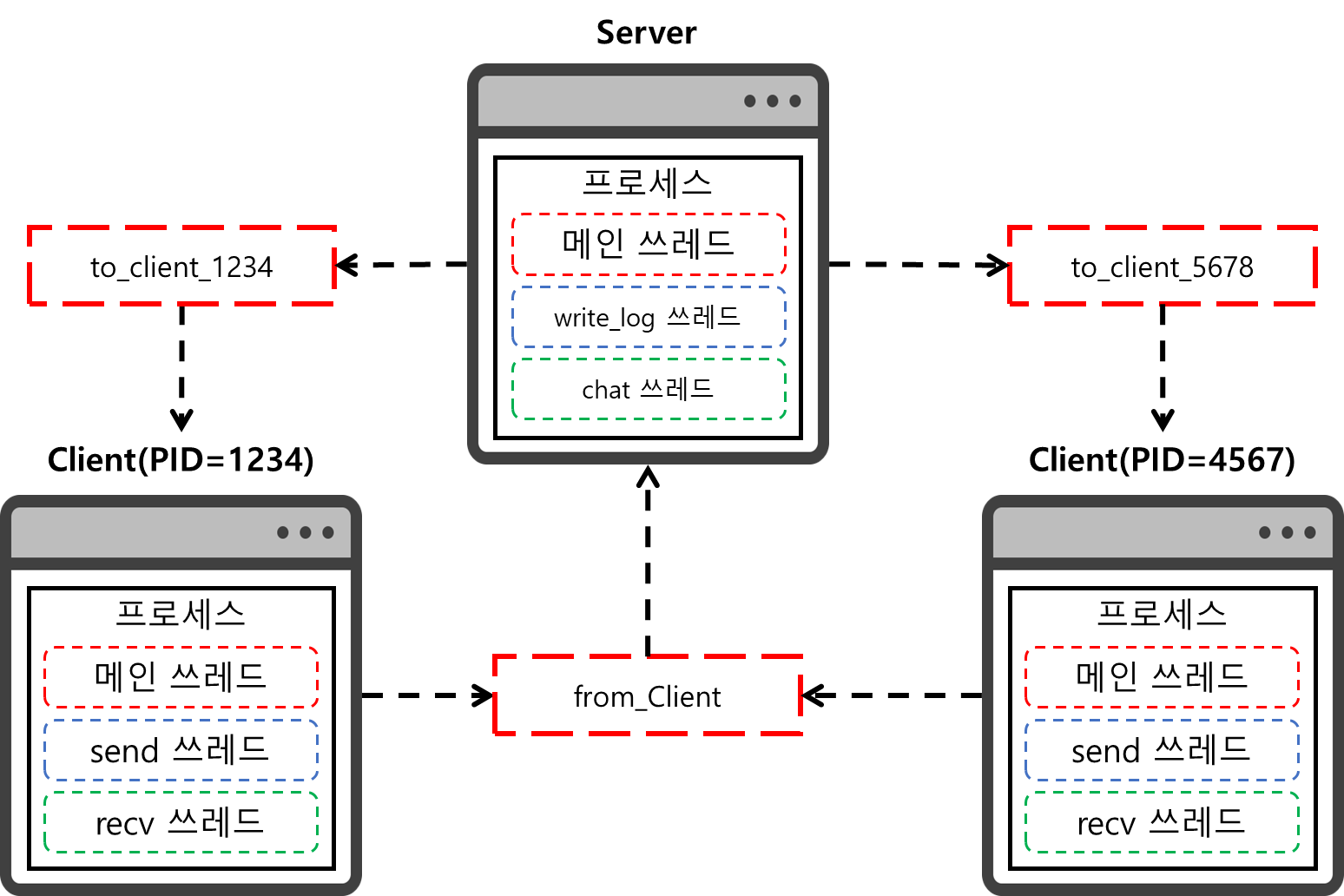
# 시스템 설계

## **Named Pipe**

### **소프트웨어 설계도**스크린샷이(가) 표시된 사진 자동 생성된 설명

1. 이미지 파일을 ASCII 코드로 변환시켜 1.txt, 2.txt 과 같이 숫자.txt 파일로 상대경로의 quiz폴더에 저장해 놓는다.
2. 서버 프로그램을 실행한다.
3. mkfifo( ) 함수로 fifo 파일을 생성한다. To\_client\_[PID] 와 From\_client fifo 파일을 생성한다.
4. 클라이언트 프로그램을 실행한다.
5. 클라이언트 프로그램은 실행되면 서버에 접속했음을 알리는 메시지를 전송한다.
6. 클라이언트로부터 접속했음을 알리는 메시지를 받은 서버는 구조체 메시지를 저장하고 클라이언트 수를 카운트한다.
7. 접속한 클라이언트가 2명이 되면 채팅을 시작한다.
8. 서버는 클라이언트가 2명이되면 이미지를 아스키코드로 변환하고 txt형태로 저장한 파일을 불러와서 클라이언트에게 전송한다.
9. 클라이언트는 정답을 입력하고 서버에 전달한다.
10. 서버는 일반 채팅인지 정답인지 확인한다.
11. 정해진 문제를 모두 실행하면 자동으로 서버 프로그램은 종료된다.

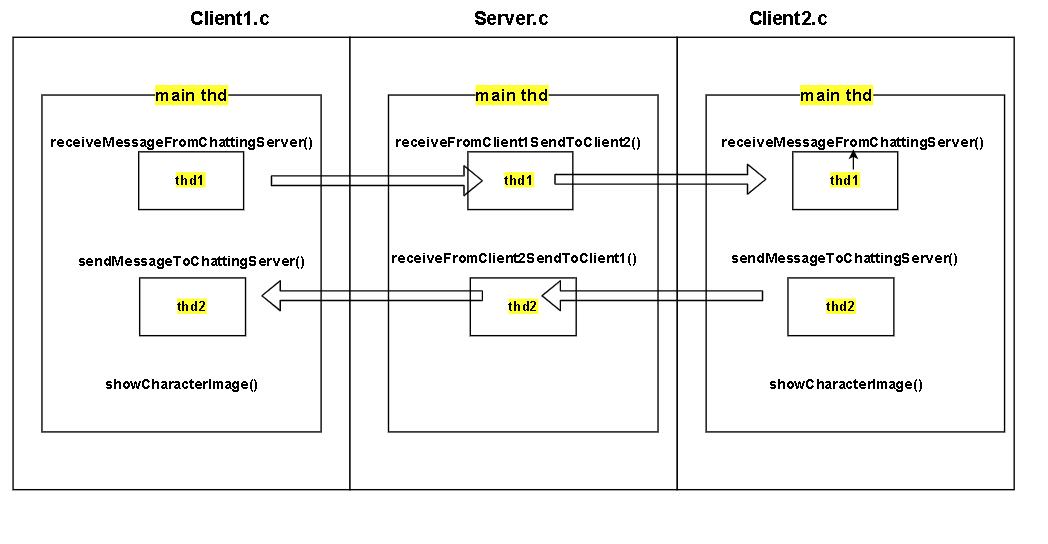
### **쓰레드 설계**



위의 그림은 IPC 기법 중 Named Pipe를 사용한 프로세스와 쓰레드 설계도이다. 서버 프로세스는 실행하면 메인 쓰레드를 실행하고 log.txt 파일에 로그를 기록하는 write\_log 쓰레드와 채팅과 정답을 체크하는 chat 쓰레드가 실행된다. 클라이언트 프로세스는 실행하면 메인 쓰레드가 실행되고 사용자가 메시지를 입력하면 from\_client 파이프를 통해 서버에 전달하는 send 쓰레드와 서버 프로세스로부터 메시지를 계속해서 읽어오는 receive 쓰레드가 실행된다. 프로세스간 통신하는 데이터는 구조체로 정의하고 fifo 파일로 데이터를 전송, 수신한다.

## **Shared Memory**

### **소프트웨어 설계도**



Client1과 Client2가 서버를 통하여 네트워크 소켓 프로그래밍 없이도 Shared Memory IPC 기법을 이용해 메시지를 주고받도록 설계하였다. 인물 사진을 터미널 창에 출력하는 부분은 잦은 에러 발생으로 cleint1과 client2 각자의 프로세스에서 main 쓰레드가 시작하면 showCharacterImage()를 통해 띄우도록 하였다. 클라이언트1, 2, 서버 프로세스에는 각자 쓰레드가 총 3개씩 존재한다. 우선 client1과 client2 프로세스에는 동일하게 receiveMessageFromChattingServer 함수와 sendMessageToChattingServer 함수가 메인 쓰레드에서 두개의 쓰레드로 분기하면서 총 3개의 쓰레드가 동작하게 된다. 이 두 함수를 통해 서버에게 메시지를 전달해주고 서버에서 메시지를 전달받는다.

Server 프로세스에서는 클라이언트1으로부터 전달받은 메시지를 클라이언트2에게 전달해주는 receiveFromClient1SendToClient2 함수가 메인 쓰레드에서 쓰레드로 분기한다. 또한 클라이언트2로부터 전달받은 메시지를 클라이언트1에게 전달해주는 receiveFromClient2SendToClient1 함수가 쓰레드로서 작동한다. 따라서 서버 프로세스에는 총 3개의 쓰레드가 존재하게 된다.

다음과 같이 설계하기 위해서는 공유메모리(shared memory) 2개와 동기화 툴(synchronization tool)로서 세마포어 2개가 필요한데 그에 대한 자세한 설명은 아래 쓰레드 설계에서 자세하게 설명하도록 하겠다.

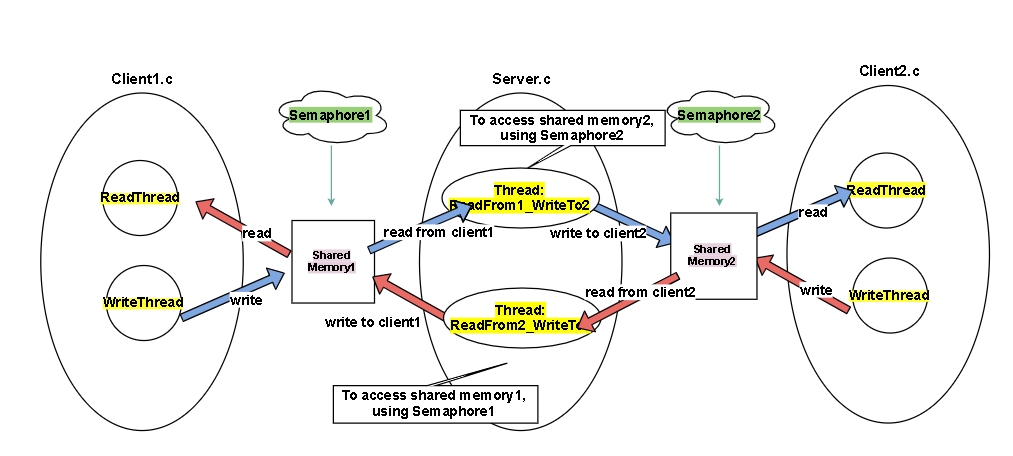
### **쓰레드 설계**

[그림2]

[그림1]

클라이언트1과 클라이언트2, 서버 3개의 프로세스가 통신을 하기 위해서 공유메모리를 생성하는 부분에서 프로그램을 짜는 초기에 많은 고민이 있었다. 원래는 [그림1]과 같이 Client2 ∩ Server ∩ Client2가 같이 공유하고 있는 교집합처럼 세 프로세스가 모두 같은 공유메모리를 공유하는 구조로 설계를 하여 코딩하려 했었다.

하지만 이렇게 되는 순간 Client1과 Client2가 함께 공유하는 메모리가 존재하게 된다. 서버의 역할은 두 프로세스가 통신할 수 있도록 이어주는 역할인데 세 프로세스가 공유하는 메모리가 생기는 순간 두 프로세스가 서버 없이도 통신할 수 있게 되어 서버가 서버로서의 역할을 하지 못하는 모순이 생긴다고 생각을 하였다. 따라서 [그림2]와 같이 Client1과 Client2 프로세스가 함께 공유하는 메모리가 없게 하여 오직 서버를 통해서만 통신 가능하게 설계를 하게 되었다.



**[그림3]**

* 참고

함수명을 좀더 가독성 있고 개발자가 아닌 사람도 한눈에 알아볼 수 있게 짓다 보니 함수명이 길어지게 되었다. 따라서 코드에서 사용한 함수명을 설계도의 그림에 담기에는 너무 길기 때문에 이번 설계도 그림에서만 짧게 줄여 부르기로 하겠다.

receiveFromClient1sendToClient2 -> **ReadFrom1\_WriteTo2**

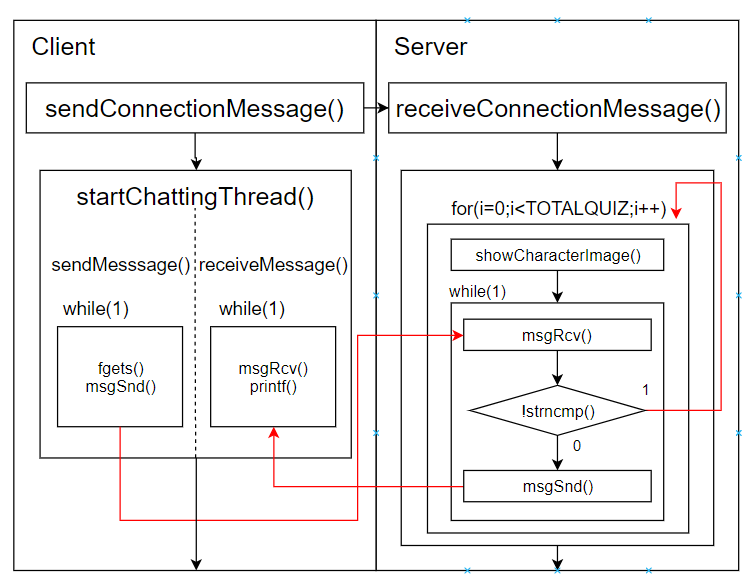
receiveFromClient2sendToClient1 -> **ReadFrom2\_WriteTo1**

[그림2]와 같은 구조로 클라이언트와 서버의 관계를 설계한 결과 [그림3]과 같이 쓰레드 구조를 설계할 수 있었다. 우선 이 설계도에서는 **공유메모리가 총 2개**가 필요하다. 그림에서 확인할 수 있듯이 main에서 client1과 client2 간에 오고 가는 메시지를 전달하려면 우선 클라이언트1 프로세스와 main 프로세스간 공유메모리를 생성할 필요가 있다. 다음으로 클라이언트2와 main 프로세스간 공유메모리를 생성해야 한다. Client1에서 쓰기함수인 sendMessageToChattingServer()를 통해 공유메모리에 쓴 데이터를 서버의 receiveFromClient1SendToClient2 함수(receive message From client1 and then send to client2)를 통해 서버는 ‘클라이언트1과 자신’과의 공유메모리에서 메시지를 읽어 ‘클라이언트2와 자신’과의 공유메모리에 쓰기작업을 한다. 따라서 서버는 클라이언트1이 작성한 메시지를 읽어와 클라이언트2와의 공유메로리에 전달을해준다. 그 반대 방향인 receiveFromClient2SendToClient1 함수도 receiveFromClient1SendToClient2 함수와 같은 원리로 작동하게 된다. 따라서 클라이언트1, 서버, 클라이언트2 세 프로세스를 관통하여 메시지가 전달되는 흐름이 ->(왼쪽에서 오른쪽)와 <-(오른쪽에서 왼쪽) 뱡향으로 크게 2가지 전달 흐름이 있다고 할 수 있다.

더불어 이 설계도에서는 **세마포어 총 2개**가 필요하다. 맨 처음 공유 메모리 2개를 공유메모리1에 접근하는 쓰레드가 공유메모리2에도 접근을 하여 2개의 공유메모리에 모두 같은 세마포어 키를 사용하여 동기화를 하였다. 하지만 서로 다른 공유메모리는 서로 다른 세마포어 키가 필요하기 때문에 2개의 세마포어 키를 가지고 서로 다른 공유메모리를 동기화할 필요가 있다. 결론적으로 세마포어 2개를 서로 다른 공유메모리에 사용함으로써 공유메모리에 접근하는 여러 쓰레드가 경쟁상태(race condition)에 있는 것을 막아 서버가 프로세스 사이에 오고 가는 메시지를 원활하게 전달할 수 있다.

## **Message Passing**

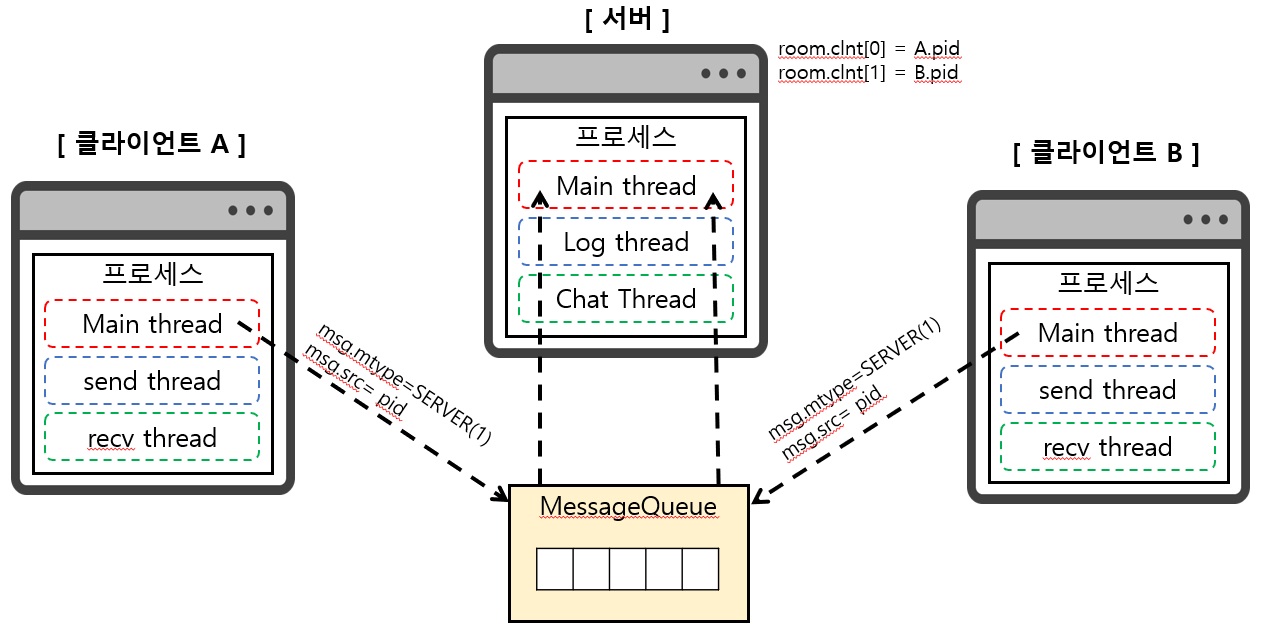
### **소프트웨어 설계도**



주요 기능을 하는 함수들의 흐름을 표기한 흐름도이다. 먼저 서버는 프로그램이 실행되면 receiveConnectionMessage()에서 블록상태가 되어 클라이언트의 접속을 기다린다. 클라이언트는 실행 시 sendConnectionMessage()를 통해 자신의 고유번호인 pid를 서버측으로 메시지 큐를 통해 전달하고 sendMessage, receiveMessage 쓰레드를 통해 반복하며 읽고 쓸 준비를 끝마친다. 클라이언트에서 모든 메시지는 서버를 향해 전송한다. 서버는 연결 시 저장한 클라이언트들의 pid와 메시지의 출발지정보를 비교하여 메시지 전달을 수행한다. 클라이언트들이 모두 접속하면 게임이 실행되고 문제의 총 개수만큼 반복하며 서버에서 미리 저장되어 있는 아스키 코드 이미지를 각각의 클라이언트들에게 전달해준다. 이후에는 클라이언트의 메시지들을 정답과 비교하며 채팅기능을 수행한다. 정답을 모두 맞추면 게임이 종료된다.

### **쓰레드 설계**

소프트웨어 설계의 내용을 쓰레드와 메시지 큐 사이의 관계에 중점을 두어 표현하면 아래와 같은두개의 흐름으로 나누어진다.

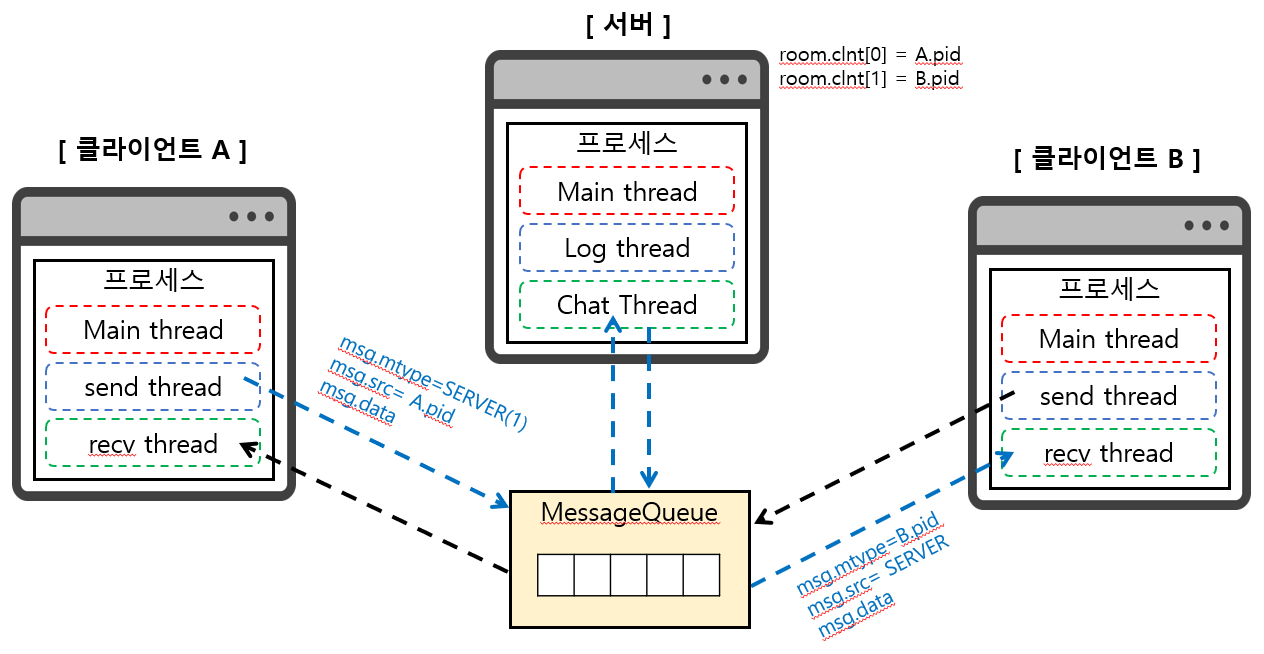


실행된 서버의 메인 쓰레드는 메시지큐를 초기화 시키고, 로그 쓰레드를 생성하여 익명파이프를

통해 상세한 내용을 log.txt파일로 기록하고 메시지 큐를 읽어(자신의 번호인 1번 메시지를 읽음)

room 구조체의 손님배열에 해당하는 클라이언트의 pid를 집어넣는다.

원하는 클라이언트를 전부 채워 넣은 후에 Chat 쓰레드를 실행하여 게임을 진행한다.



이후의 실행흐름 중 클라이언트 A가 B에게 메시지를 전송하는 상황을 나타낸 것이다.

먼저 클라이언트 A의 send thread에 fgets문을 통해 사용자에게 메시지를 입력받고 해당 내용을서버 방향으로 메시지 큐에 올려 전달한다. 서버의 chat thread에서는 먼저 자신에게 온 메시지를 확인하여 인물 퀴즈의 정답과 일치하는지 확인하고 msg.src라는 출발지 pid 정보와 미리 받아 놓은 room 구조체의 사용자 pid를 비교하여 메시지의 목적지를 재지정 하여 메시지 큐에 다시 적재 한다. 메시지큐에 들어간 메시지는 클라이언트 B의 recv 쓰레드에 의해 읽혀지고 출력된다.

요약하자면 클라이언트 A의 send Thread를 통해 메시지 큐에 입력한 내용이 서버의 Chat 쓰레드를 통해 B의 recv 쓰레드에게 전달되어 출력된다. 이러한 과정을 통해 채팅기반 인물 퀴즈가 진행된다.

# 시스템 구현 및 데모

## **소스코드 주요부분 설명**

### **Named Pipe**

* **Fifo\_server.c 파일**

**(구조체)**

**typedef struct msgtype {**

**long mtype;**

**char data[BUF\_SIZE];**

**long src;**

**}msg\_t;**

**typedef struct Room {**

**long clnt[2];**

**} Room;**

서버 프로그램에서 사용하는 구조체이다. msg\_t는 클라이언트와 통신하기 위한 메시지 프로토콜이다. mtype 멤버는 데이터가 어떤 기능을 하는지를 나타낸다. data[BUF\_SIZE] 데이터를 담고 있는 멤버이다. 클라이언트와 서버 둘 모두 BUF\_SIZE를 매크로로 크기를 고정되게 하여 통신한다. src 멤버는 메시지를 보낸 클라이언트의 PID를 의미한다.

* **(메인함수)**

**pipe(log\_fds);**

**pthread\_create(&write\_log\_thread, NULL, writeLogToTextFile, NULL);**

**receiveConnectionMessage(&room);**

**pthread\_create(&chat\_thread, NULL, startUserChatting, &room);**

**pthread\_join(write\_log\_thread, NULL);**

**pthread\_join(chat\_thread, &chat\_thread\_return);**

**프로그램의 전체 실행을 담당하는 main( ) 함수**의 흐름이다. 가장 먼저 fifo\_log.txt에 server의 로그를 기록하기 위해 파이프를 생성한다. 이후 fifo\_log.txt에 기록하기위해 실행하는 write\_log\_thread 쓰레드를 pthread\_create( )함수로 쓰레드를 생성하고, 클라이언트가 접속했다는 메시지를 전송 받기 위해 대기하는 함수 receiveConnectionMessage( ) 함수를 실행한다. 그 후 전반적인 클라이언트의 채팅을 관리하고 클라이언트들에게 문제를 출력하게 하는 chat\_thread를 생성한다. 쓰레드가 종료 될 때까지 기다리고 종료하면 사용한 자원을 반납하게 하는 pthread\_join( )함수를 통해 쓰레드 종료를 대기한다.

* **(함수)**

**void receiveConnectionMessage(Room\* room);**

**void startUserChatting(Room\* room);**

**void** **showCharacterImage(Room\* room, int num);**

**void\* writeLogToTextFile();**

receiveConnectionMessage( ) 클라이언트가 접속했음을 알리는 메시지를 전송 받고 클라이언트 2명 모두 접속될 때까지 실행을 대기하는 함수이다.

startUserChatting( ) 클라이언트 2명이 모두 접속하고 채팅이 시작되면 채팅 메시지를 전송받고 다른 클라이언트에게 전송하는 채팅 쓰레드의 핸들러 함수이다.

showCharacterImage( ) 클라이언트에게 이미지 Quiz를 전송하는 함수이다. 미리 저장된 파일을 불러오고 불러온 데이터를 클라이언트들에게 전송한다.

writeLogToTextFile( ) 서버의 내용을 기록하는 로그파일에 로그내역을 저장하는 쓰레드의 핸들러 함수이다.

* **(pipe 경로 설정)**

**#define FROM\_CLIENT\_FILE "./fifo/from\_client"**

**#define TO\_CLIENT\_FILE "./fifo/to\_client\_"**

**mkfifo(FROM\_CLIENT\_FILE, 0666);**

**sprintf(path, "%s%ld", TO\_CLIENT\_FILE, room->clnt[i]);**

**mkfifo(path, 0666);**

mkfifo( ) 함수를 이용해 Named Pipe 파일인 fifo 파일을 생성한다. 클라이언트로부터 read( )하는 파일은 고정이기 때문에 매크로로 #define FROM\_CLIENT\_FILE “./fifo/from\_client”로 선언한다. 클라이언트에게 write( )해서 데이터를 전송하기 위한 파일은 접속하는 클라이언트마다 따로 열어줘야하기 때문에 고정으로 “to\_client\_” 를 매크로 #define TO\_CLIENT\_FILE “./fifo/to\_client\_”로 선언하여 사용한다. 가변되는 부분은 sprintf( )함수를 이용해 고정 파일명 + PID를 붙여 생성한다.

* **(pipe read)**

read\_fd = open(FROM\_CLIENT\_FILE, O\_RDWR)

read(read\_fd, &msg, sizeof(msg)

클라이언트가 보내주는 데이터를 읽기위한 코드이다. fifo파일을 open( )함수를 이용해서 read\_fd 변수에 file descriptor의 값을 저장하고 read( ) 함수에 read\_fd에서 읽어온 데이터를 msg 변수에 넣어준다. O\_RDWR는 읽기 쓰기 권한을 모두 부여하는 것을 말한다.

* **(pipe write)**

**sprintf(path, "%s%ld", TO\_CLIENT\_FILE, room->clnt[j]);**

**write\_fd = open(path, O\_RDWR)**

**msg.mtype = room->clnt[j];**

**write(write\_fd, &msg, sizeof(msg))**

**서버가 클라이언트에게 데이터를 전송할 때 사용하는 코드 부분**이다. open( ) 함수를 이용해서 클라이언트에게 데이터를 전송하기 위해 사용할 fifo 파일을 오픈하고 write\_fd에 해당 파일의 file descriptor를 저장한다. room->clnt[j]는 서버에 접속한 클라이언트의 PID를 가지고 있는다. msg.mtype은 데이터를 받았을 때 누구에게 전달해야 할 지 판별하기 위해 사용하는 message type이다. sprintf( )함수로 path를 정해준다. 여기서 room->clnt[i]는 접속한 클라이언트들의 PID를 가지고 있다. TO\_CLIENT\_FILE은 default 경로이고 뒤에 room->clnt[i]를 붙여주면 클라이언트에게 데이터를 전송할 독립적인 fifo 파일의 경로를 설정해준다. write\_fd에 저장된 file descriptor를 이용해 fifo 파일에 msg 데이터를 전송한다.

* **(클라이언트 채팅 전송)**

**if (msg.src == room->clnt[0])**

**{**

**msg.mtype = room->clnt[1];**

**memset(path, 0, PATH\_SIZE);**

**sprintf(path, "%s%ld", TO\_CLIENT\_FILE, room->clnt[1]);**

**if ((write\_fd = open(path, O\_RDWR)) == -1**

**{**

**perror("[SYSTEM] Open Error!!!\n"); // 오류 처리**

**}**

**}**

msg.src로 구분하여 어떤 클라이언트에게 메시지를 전송할지 판단하고 경로를 정해주는 코드 부분이다. 마찬가지로 sprintf( )와 open( )을 이용한다.  
msg.src == room->clnt[0] 이면 0번 클라이언트이기 때문에 메시지를 받을 클라이언트를 1번 클라이언트로 설정한다. 1번 클라이언트와 통신하기 위한 fifo를 open한다.

* **Fifo\_client.c 파일**
* **(구조체)**

**typedef struct msgtype {**

**long mtype;**

**char data[BUF\_SIZE**

**long src;**

**}msg\_t;**

서버와 클라이언트 간의 데이터를 통신할 때 사용하는 데이터의 구조체이다. mtype은 해당 데이터가 무엇을 의미하는지 서버에 알려주기 위한 값을 저장하는 멤버이다. CONNECT일 경우 처음 클라이언트가 서버와 연결됨을 알려주고, SERVER일 경우 서버에게 일반적으로 보내는 메시지를 의미한다. data[BUF\_SIZE]는 서버에게 메시지를 보내기위해 클라이언트가 입력한 메시지를 저장하는 멤버이다. src는 클라이언트의 PID를 저장하는 멤버이다. 메시지를 보낸 클라이언트가 무엇인지 판별하고 서버가 보내는 데이터를 read하기 위해 경로를 설정할 때 사용하는 멤버이다.

* **(전역변수)**

**long id=0;**

**int read\_fd=0;**

**int write\_fd=0;**

**char read\_path[PATH\_SIZE];**

변수 **id**는 클라이언트의 Process ID를 저장한다.

변수 **read\_fd**는 서버가 보내는 메시지를 수신하기 위한 fifo파일의 descriptor를 저장한다.

변수 **write\_fd**는 서버에게 메시지를 전송하기 위한 fifo파일의 descriptor 정보를 저장한다.

배열 **read\_path**는 서버가 보내는 메시지를 수신하기 위해 오픈하는 fifo의 경로를 저장하는 변수이다.

* **(File descriptor 지정)**

**id = (long)getpid(); // getpid()**

**if ((write\_fd = open(TO\_SERVER\_FILE, O\_RDWR)) == -1)**

**{**

**perror("[SYSTEM] mkfifo 파일 열기 실패(send to server)\n");**

**printf("[SYSTEM] 프로그램이 3초 후 종료됩니다.\n");**

**sleep(3);**

**exit(1);**

**}**

**sprintf(read\_path, "%s%ld", FROM\_SERVER\_FILE, id);**

**if ((read\_fd = open(read\_path, O\_RDWR)) == -1)**

**{**

**perror("[SYSTEM] mkfifo 파일 열기 실패(receive from server)\n");**

**printf("[SYSTEM] mkfifo 파일 생성 중(receive from server)\n");**

**mkfifo(read\_path, 0666);**

**printf("[SYSTEM] mkfifo 파일 생성 완료(receive from server)\n");**

**if ((read\_fd = open(read\_path, O\_RDWR)) == -1)**

**{**

**perror("[SYSTEM] mkfifo 열기 실패(receive from server)\n");**

**printf("[SYSTEM] 프로그램이 3초 후 종료됩니다.\n");**

**sleep(3);**

**exit(1);**

**}**

**}**

**서버와 통신하기 위한 fifo 파일을 open하는 함수**이다. 서버에게 write하여 데이터를 전송하는 fifo 파일은 모든 클라이언트가 고정이기 때문에 매크로로 정의한 고정 경로로 파일은 open( ) 한다. 서버로부터 데이터를 전송 받기위한 클라이언트와 서버의 개별적 fifo 파일을 생성하고 open( ) 한다. getpid로 실행하고 있는 해당 클라이언트의 pid를 얻고 pid를 default 경로에 pid를 sprintf( )를 이용해 to\_client\_(pid)의 형태로 생성한다. 그리고 생성이 완료되면 open( ) 함수로 연결한다.  
 write\_fd 는 서버에게 데이터를 write하여 전송하기 위해 fifo file의 file descriptor를 저장하는 변수이다. read\_fd 는 서버가 보내는 데이터를 송신하기 위한 fifo file의 file descriptor를 저장하는 변수이다.

* **(채팅 전송)**

**while (1)**

**{**

**memset(send\_buf, '\0', sizeof(send\_buf));**

**fgets(send\_buf, sizeof(send\_buf), stdin**

**send\_buf[strlen(send\_buf)] = '\0';**

**strncpy(msg.data, send\_buf,sizeof(send\_buf));**

**write(write\_fd, &msg, sizeof(msg));**

**}**

데이터를 서버와 연결되어 있는 pipe인 fifo파일에 write하는 쓰레드에 들어가는 핸들러 함수이다. while(1)로 반복적으로 수행하며 fgets( ) 함수로 클라이언트 사용자에게 메시지를 입력 받고 서버에게 데이터를 전송하기 위해 연결한 fifo 파일과 연결된 file descriptor의 값을 write\_fd에 저장하고, 데이터를 write( ) 함수를 이용해 fifo 파일에 write( ) 한다.

* **(채팅 수신)**

**while (1)**

**{**

**if (read(read\_fd, &msg, sizeof(msg)) > 0)**

**{**

**printf("%s", msg.data);**

**}**

**}**

서버가 보내주는 데이터를 read하는 쓰레드에 들어가는 핸들러 함수이다. 연결되어 있는 pipe의 fifo파일의 file descriptor의 값을 저장하고 있는 read\_fd를 이용하여 메시지를 받고 클라이언트에게 메시지를 보여주기 위해 printf( )로 해당 데이터의 메시지를 출력한다.

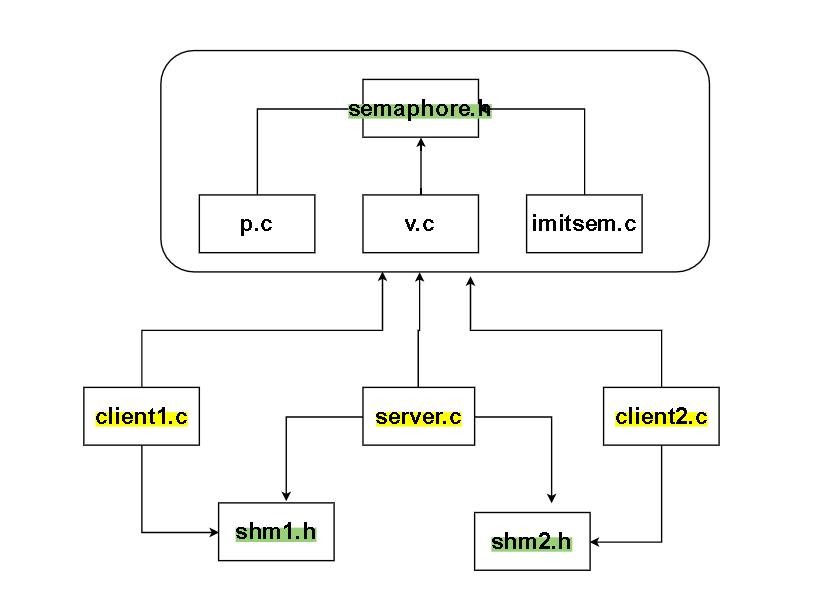
* **(총 정리)**

공통적으로 서버와 클라이언트 프로그램은 통신을 위한 fifo파일을 open하고, fifo 파일을 통해 서로의 데이터를 송수신한다. 서버는 클라이언트가 보낸 메시지를 읽고 해당 데이터를 처리하여 다른 클라이언트에게 채팅을 전송하고 문제를 출력한다. 클라이언트는 데이터를 서버에 전송하고 서버로부터 데이터를 수신한다.

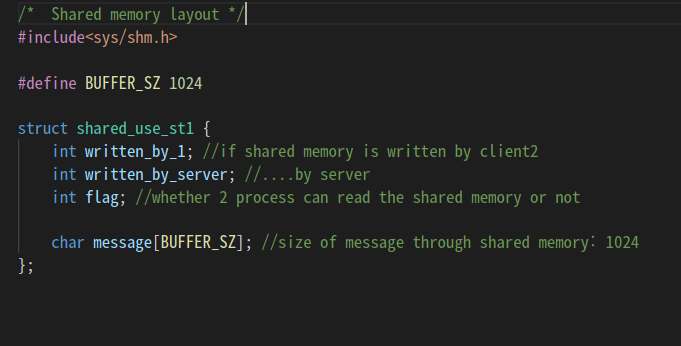
공통되는 데이터 송수신 과정은 fifo파일을 open( )함수로 열고 반환되는 file descriptor 값을 저장하고 저장된 file descriptor를 이용해 데이터를 수신할 때는 read( )함수를 통해 fifo에 저장된 데이터를 읽고, 데이터를 송신할 때는 write( )함수를 이용해 fifo에 데이터를 쓴다.

### **Shared Memory**

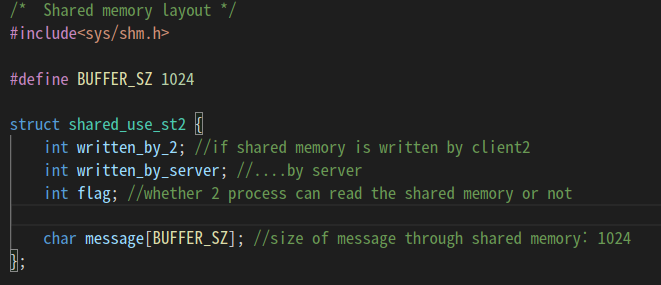
* **소스파일의 포함관계 및 구조**



* **client1.c, server.c, client2.c**: main문이 담긴 파일로 이 3파일들을 gcc 컴파일러로 돌리면 된다. 결국 이 세 메인 파일에서 세마포어와 공유메모리를 사용하므로 아래에서 기술한 모든 파일을 헤더로 include하고 사용해야 한다.
* **shm1.h**: client1.c 파일의 프로세스와 server.c 파일의 프로세스가 공유하는 메모리 구조를 선언해 놓은 파일이다. 따라서 client1.c와 server.c 양쪽에서 이 헤더파일을 include 해주어야 한다.
* **shm2.h**: shm1.c와 같이, client2.c 파일의 프로세스와 server.c 파일의 프로세스가 공유하는 메모리 구조를 선언해 놓은 파일이다. 따라서 client2.c와 server.c 양쪽에서 이 헤더파일을 include 해주어야 한다.
* **semaphore.h**: 프로그램에서 사용하는 공유 메모리 2개에 대해 여러 프로세의 여러 쓰레드가 접근하고 읽고 쓰는 작업을 시도한다. 따라서 동기화 툴인 세마포어를 사용하고 세마포어의 구조를 이 헤더파일에 선언해 놓았다.
* **p.c, v.c**: p.c파일과 v.c 파일은 세마포어가 수행하는 -1 혹은 +1 연산작업을 기술한 파일이다. 이 파일에서는 미리 선언해 놓은 세마포어 구조를 사용해야 하므로 semaphore.h 파일을 include해야 한다.
* **initsem.c**: initsem.c 파일은 사용할 세마포어를 초기화하는 부분이다. 따라서 이 파일에서는 미리 선언해 놓은 세마포어 구조를 사용해야 하므로 semaphore.h 파일을 include해야 한다.
* **shm1.h 파일**

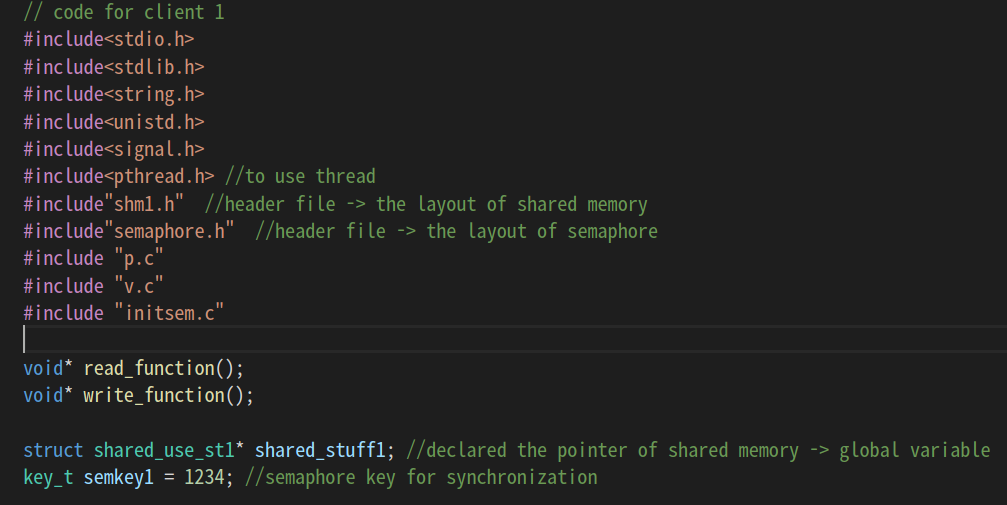


* **message**: 1024 크기의 char형 배열로 공유메모리 구조체에서 실제로 읽히고 쓰여지는 부분이라고 할 수 있다.
* **written\_by\_1**: 프로세스 client1에서 공유메모리의 message 배열에 ‘쓰기’ 작업을 하였다면 이 변수를 1로 설정한다. 만약에 서버에 의해 message 부분이 읽혔다면 0으로 다시 초기화해주어 공유메모리가 비었음을 표시한다. 공유메모리가 비었다고 표시가 되면 서버 혹은 클라이언트1에서 공유메모리에 쓰기 작업을 시도할 것이다.
* **written\_by\_server**: 서버 프로세스에서 공유메모리의 message 배열에 ‘쓰기’ 작업을 하였다면 이 변수를 1로 설정한다. 만약에 서버에 의해 message 부분이 읽혔다면 0으로 다시 초기화해주어 공유메모리가 비었음을 표시한다. 공유메모리가 비었다고 표시가 되면 서버 혹은 클라이언트1에서 공유메모리에 쓰기 작업을 시도할 것이다.
* **flag**: 공유메모리가 비었는지 특정 쓰레드에 의해 쓰여서 꽉 차 있는지는 written\_by\_1과 written\_by\_server로도 할 수 있지만 이는 어떤 쓰레드가 썼는지를 표시하기 위해 만든 변수이다. 따라서 어떤 쓰레드가 썼는지 여부와 상관없이 공유메모리가 비었는지 꽉 찼는지만을 확인할 때 이 flag 변수를 확인할 수 있다. flag가 1이면 공유메모리는 특정 쓰레드에 의해 쓰여져 누군가에 의해 빨리 읽혀져야 하는 상황임을 알린다. flag가 0이면 공유메모리가 누군가에 의해 읽혀지어 비었기 때문에 특정 쓰레드가 쓰기 작업을 해도 된다는 상황을 의미한다.
* **shm2.c 파일**



* **written\_by\_2:** 이 변수는 프로세스 client2에서 서버와 공유하는 공유메모리의 변수임을 나타내기 위해 by 뒤에 숫자 2를 표기하였다. shm1.h와 같이 역시나 공유메모리인 message 배열에 ‘쓰기’ 작업을 하였다면 이 변수를 1로 설정한다.
* shm1.h와 구조가 거의 같으므로 자세한 설명은 생략한다.
* **client1.c 파일** -> (client2.c 파일은 client1.c파일과 변수명을 제외하고 구조가 거의 같으므로 설명은 생략하고 사진 첨부는 맨 마지막 전체 코드에 첨부하도록 하겠다. )

**헤더파일, 함수, 전역변수**



**(헤더파일)**

* **#include <shm1.h>** : 클라이언트1과 서버가 공유할 공유메모리를 정의한 헤더파일 포함
* **#include <semaphore.h>**: 클라이언트1과 서버가 공유메모리에 접근할 때, 동기화가 필요하므로 세마포어 구조를 정의한 헤더파일을 포함 그 외 세마포어 연산 및 초기화에 필요한 헤더파일들 #include “p.c”, #include “v.c”, #include “initsem.c” 포함

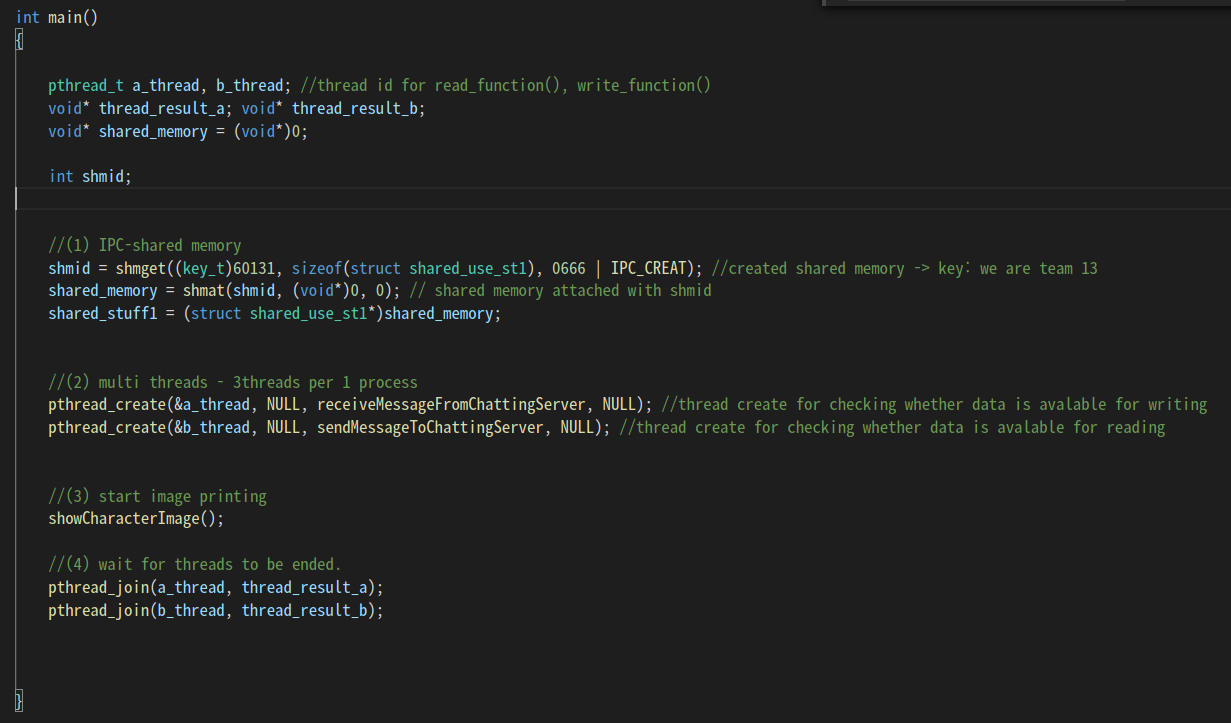
**(함수)**

* **receiveMessageFromChattingServer():** 클라이언트1과 서버간 공유되는 메모리를 읽어서 client1 터미널에 출력한다.
* **sendMessageToChattingServer():** 클라이언트1과 서버간 공유되는 메모리를 읽어서 client1 터미널에 출력한다.
* **showCharacterImage():** 클라이언트1의 터미널에서 인물 사진을 출력하는 함수이다.

**(전역변수)**

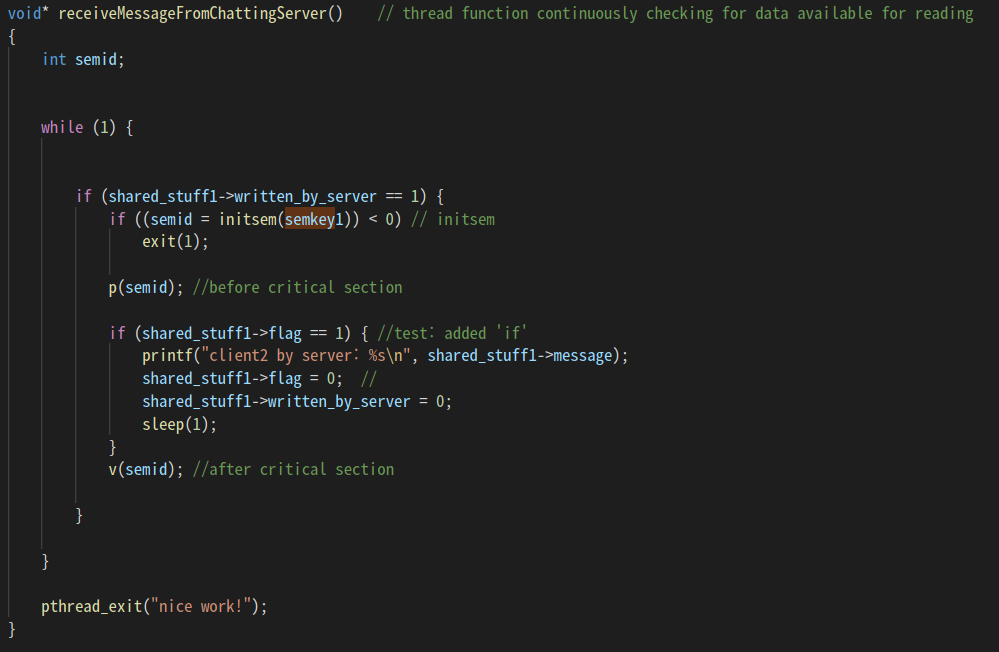
* **shared\_stuff1:** 클라이언트1과 서버 간에 사용할 공유메모리를 가리킬 포인터로 struff 뒤에 1을 붙여 클라이언트2와 서버 간에 공유하는 메모리와 구분하였다.
* **semkey1:** 클라이언트1과 서버가 사용하는 세마포어의 키로서 1234로 임의로 설정함

**main함수**



* **주석 (1):** 공유메모리를 생성하고 client1 프로세스에 부착하는 과정을 담았다.
* **주석 (2):** 이 부분은 메인문의 메인 쓰레드에서 receiveMessageFromChattingServer()과 sendMessageToChattingServer()으로 쓰레드를 생성하게 된다. 따라서 한 프로세스당 총 3개의 쓰레드가 동시에 병렬적으로 실행되게 된다.
* **주석 (3):** showCharacterImage() 함수를 호출해 클라이언트1에서 인물사진이 출력되기 시작한다.
* **주석 (4):** 이 부분은 쓰레드가 종료되길 기다리는 코드이다.

**receiveMessageFromChattingServer 함수**



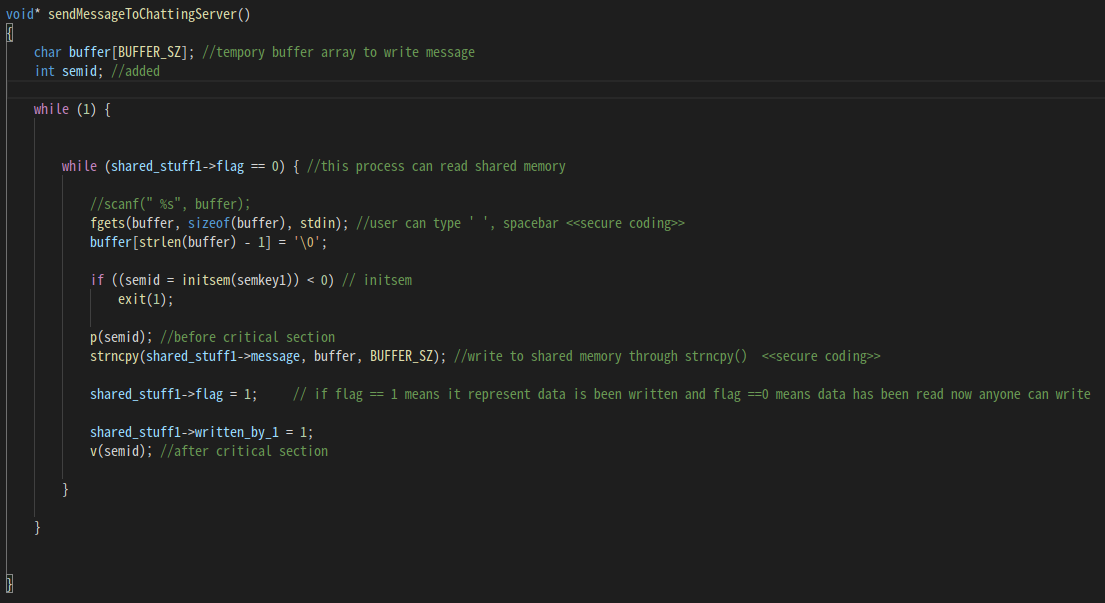
* main 문에서 receiveMessageFromChattingServer() 함수로 쓰레드가 분기하게 된다.
* **while(1):**  shm1.h 부분에서 설명하였듯이, 서버가 공유메모리에 ‘쓰기’ 작업을 하였으면 written\_by\_server 값을 1로 변경해 놓기 때문에 while문 안에서 written\_by\_server 값이 1인지 끊임없이 검사한다.
* **if(shared\_stuff1 -> written\_by\_server == 1):** 쓰레드가 동작하면서 while문 안에서 실시간으로 계속 클라이언트1 프로세스와 서버간에 공유하는 메모리에 서버가 ‘쓰기’를 하였는지 검사를 한다.
* **if(shared\_stuff1 -> flag == 1):** 이 때, written\_by\_server 값이 1이어서 서버가 공유메모리에 ‘쓰기’ 작업을 하였더라도 client1이 ‘읽기’ 작업을 이미 하였을 수도 있다. 따라서 서버가 ‘쓰기’ 작업을 하였지만 ‘읽기’ 작업을 하여 공유메모리가 비어 있는지 확인을 할 필요가 있다. 따라서 flag 값이 1인지 확인을 한다. (flag가 1이면 공유메모리에 ‘쓰기’ 작업이 되어있지만 ‘읽기’작업은 하지 않아서 공유메모리가 차 있다는 뜻이다. )
* **p(semid), v(semid):** 공유메모리에 접근하기 전에 세마포어와 동기화 툴을 사용해 공유메모리에 다른 쓰레드가 동시에 접근하여 읽어가거나 쓰지 못하게 막을 필요가 있다. 따라서 임계영역인 아래 빨간 코드를 앞 뒤로 세마포어의 p연산과 v연산을 해줄 필요가 있다.
* **printf("client2 by server: %s\n", shared\_stuff1->message); //임계영역**

공유메모리의 메시지에 접근하는 임계영역(critical section) 부분이다. 공유메모리의 message 부분을 읽어 client1 터미널 창에 출력한다.

* **shared\_stuff1->flag = 0; shared\_stuff1->written\_by\_server = 0; //임계영역**

서버가 공유메모리에 작성한 메시지를 클라이언트1이 읽어갔기 때문에 다시 written\_by\_server = 0으로 초기화해주어 ‘서버는 공유메모리에 쓰기 작업을 하지 않았다’라고 상태를 바꿔줘야 한다. 또한 flag = 0 으로 초기화해주어 ‘공유메모리는 더 이상 읽을 것이 없으므로 비었다’라고 공유메모리의 상태를 바꿔준다.

**sendMessageToChattingServer 함수**



* main 문에서 sendMessageFromChattingServer() 함수로 쓰레드가 분기하게 된다.
* **while(1):** shm1.h 부분에서 설명하였듯이, 클라이언트1에서 공유메모리에 ‘읽기’ 작업을 하여 공유메모리가 비게 되면 flag 값은 0으로 설정된다. 따라서 공유메모리가 비어있을 경우에만 ‘쓰기’ 작업을 시도해야 하므로 while문 안에서 flag 값이 0인지 끊임없이 검사한다.
* **fgets(buffer, sizeof(buffer), stdin); buffer[strlen(buffer) - 1] = '\0'; //시큐어코딩 준수**

사용작(client1)이 쓰기 작업을 하는 터미널에 입력을 하여 쓰기 작업을 하는 부분이다.

원래는 scanf(" %s", buffer);로 사용자(client1)에게 입력을 받도록 하였지만 secure coding 가이드에 따라 fgets함수를 통해 입력 받는 사이즈를 인자로 주어 버퍼 오버플로우를 방지하였다.

* **p(semid), v(semid):** 공유메모리에 접근하기 전에 세마포어와 동기화 툴을 사용해 공유메모리에 다른 쓰레드가 동시에 접근하여 읽어가거나 쓰지 못하게 막을 필요가 있다. 따라서 임계영역인 아래 빨간 코드를 앞 뒤로 세마포어의 p연산과 v연산을 해줄 필요가 있다.
* **strncpy(shared\_stuff1->message, buffer, BUFFER\_SZ); //임계영역, 시큐어 코딩 준수**

공유메모리의 메시지에 접근하는 임계영역(critical section) 부분이다. 사용자에게 fgets함수를 이용해 입력 받은 값을 공유메모리의 message 부분에 복사하여 쓰는 부분이다. 평소 코딩할 때, strcpy 함수를 즐겨 썼지만 시큐어 코딩 가이드를 따라 strncpy 함수를 사용하여 복사하는 메시지의 크기를 인자로 주도록 하여 버퍼 오버플로우를 방지하였다.

* **shared\_stuff1->flag = 1; shared\_stuff1->written\_by\_1 = 1; //임계영역**

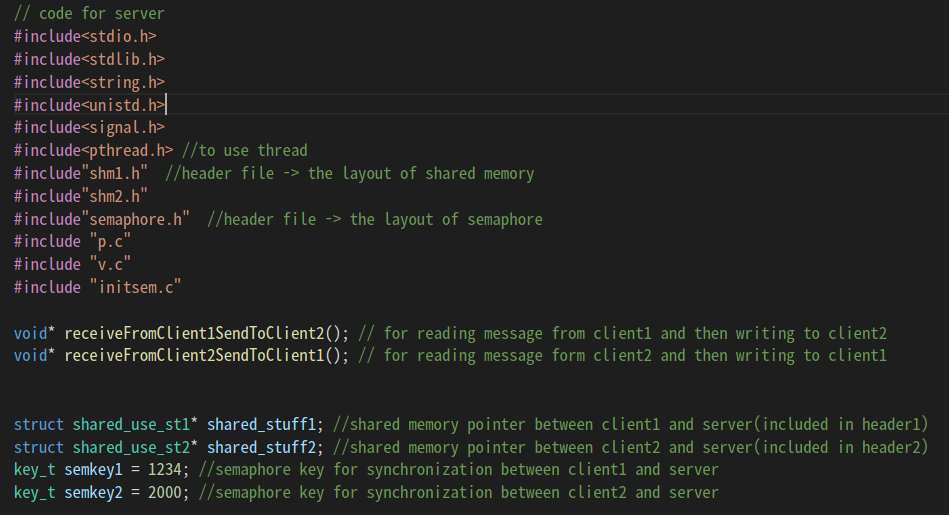
client1이 비어 있는 공유메모리에 쓰기 작업을 하였으므로 다시 flag = 1로 초기화해주어 ‘공유메모리는 차 있다. 누군가 읽기 작업을 해 줄 필요가 있다’라고 공유메모리의 상태를 바꿔준다. 또한 written\_by\_1 = 1로 값을 변경해주어 ‘client1이 공유메모리에 쓰기 작업을 하였다’ 라고 공유메모리의 상태를 바꿔줘야 한다.

**showCharacterImage 함수**



* **for문:** 이 부분은 클라이언트1의 메인 쓰레드에서 quiz 디렉토리에 미리 저장해 놓은 아스키코드형으로 변환된 이미지 파일을 차례로 읽어서 클라이언트1의 터미널에 출력되는 부분이다. BUFFER\_SZ는 미리 헤더에서 1024바이트로 선언해 놓았다. 따라서 아스키코드로 변환된 이미지 파일 .txt를 1024 단위로 끊어서 읽어 들여 터미널에 쓰게 된다.
* **server.c 파일**

**헤더파일, 함수, 전역변수**



**(헤더파일)**

* **#include <shm1.h>, #include <shm2.h>**: 클라이언트1과 공유할 공유메모리 헤더파일과 동시에 클라이언트2와 공유할 공유메모리 헤더파일을 include 한다. 즉 server 프로세스는 공유메모리 2개를 가지게 된다.
* **#include <semaphore.h>**: 클라이언트1과 서버가 공유메모리에 접근할 때, 동기화가 필요하므로 세마포어 구조를 정의한 헤더파일을 포함 그 외 세마포어 연산 및 초기화에 필요한 헤더파일들 p.c 파일, v.c 파일, initsem.c 파일 포함

**(함수)**

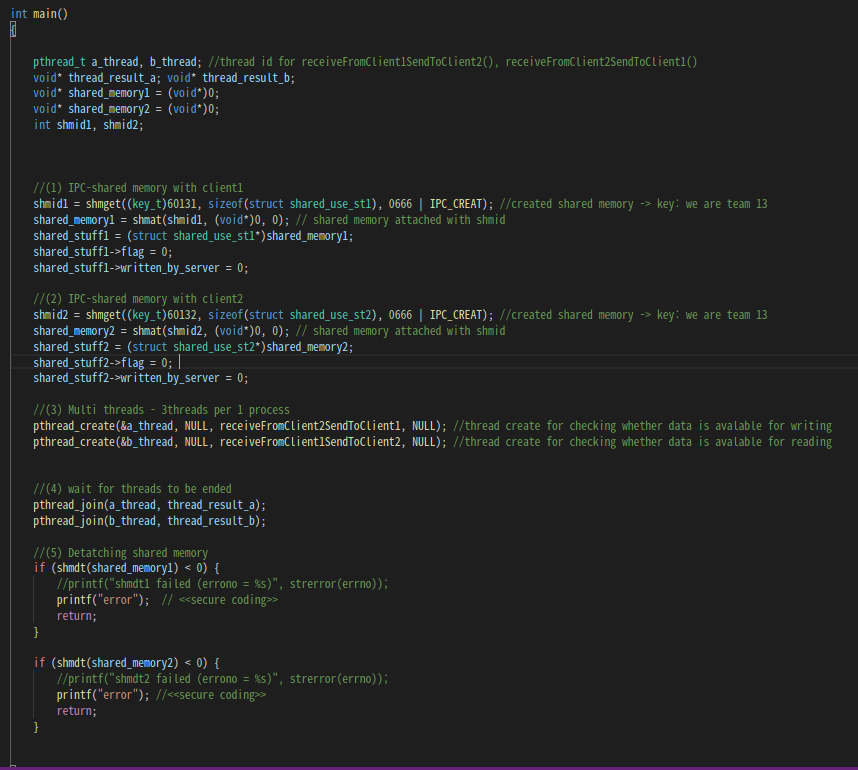
* **receiveFromClient1SendToClient2():** 클라이언트1에서 서버의 공유메모리1에 ‘쓰기’ 작업된 메시지를 읽어 서버의 공유메모리2에 ‘쓰기’ 작업하는 함수
* **receiveFromClient2SendToClient1():** 클라이언트2에서 서버의 공유메모리2에 ‘쓰기’ 작업된 메시지를 읽어 서버의 공유메모리2에 ‘쓰기’ 작업하는 함수

**(전역변수)**

* **shared\_stuff1:** 서버가 클라이언트1과 사용할 공유메모리를 가리키는 포인터
* **shared\_stuff2:** 서버가 클라이언트2와 사용할 공유메모리를 가리키는 포인터
* **semkey1:** 서버가 클라이언트1과 사용하는 세마포어의 키로서 1234으로 임의로 설정하였다. 서버의 receiveFromClient1SendToClient2 함수와 clinet1의 메시지를 받는 함수와 쓰는 함수 총 3개가 공통적으로 semkey1 값을 가지고 공유메모리1(shared\_stuff1)을 접근하는데 있어 동기화하는데 사용할 것이다.

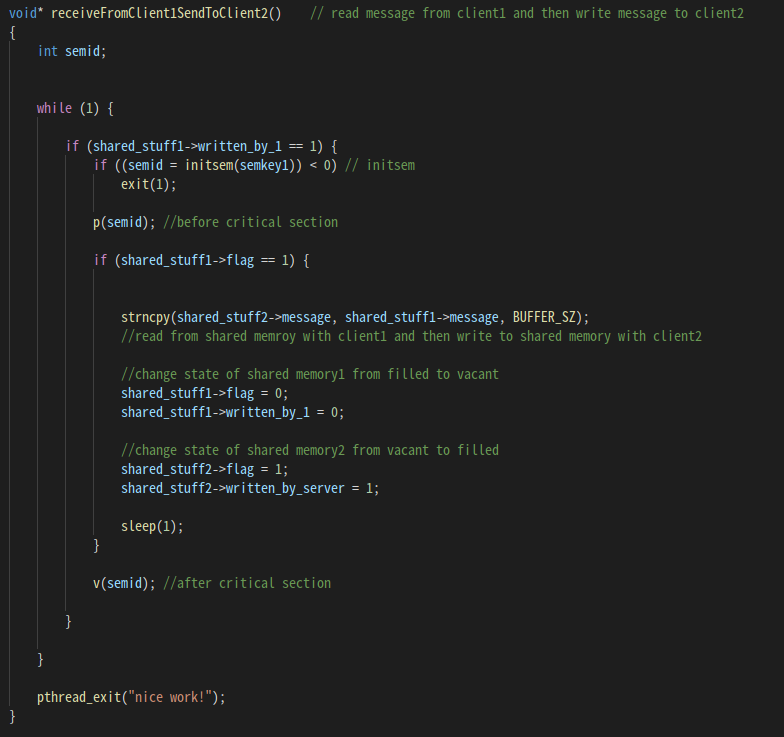
- **semkey2:** 서버가 클라이언트2와 사용하는 세마포어의 키로서 2000으로 임의로 설정하였다. 서버의 receiveFromClient2SendToClient1 함수와 clinet2의 메시지를 받는 함수와 쓰는 함수 총 3개가 공통적으로 semkey2 값을 가지고 공유메모리2(shared\_stuff2)을 접근하는데 있어 동기화하는데 사용할 것이다.

**main함수**



* **주석 (1):** 서버가 클라이언트1과 사용할 공유메모리를 shmget() 함수를 통해 생성하고, shmat() 함수를 통해 자신의 프로세스에 부착한다. 여기서 처음 클라이언트1과 서버가 접속했을 때는 공유메모리에 어떤 쓰레드로 ‘쓰기’ 작업을 하지 않았으므로 공유메모리가 비어있다. 따라서 shared\_stuff1->flag = 0 으로 서버 프로세스에서 초기화해주고 시작한다. 또한 서버에서도 클라이언트1과의 공유메모리에 메시지를 전달해준 것이 없으므로 shared\_stuff1->written\_by\_server = 0으로 초기화해준다.
* **주석 (2):** 서버가 클라이언트2와 사용할 공유메모리를 shmget() 함수를 통해 생성하고, shmat() 함수를 통해 자신의 프로세스에 부착한다. 여기서 처음 클라이언트2와 서버가 접속했을 때는 공유메모리에 어떤 쓰레드로 ‘쓰기’ 작업을 하지 않았으므로 공유메모리가 비어있다. 따라서 shared\_stuff2->flag = 0 으로 서버 프로세스에서 초기화해주고 시작한다. 또한 서버에서도 클라이언트2와의 공유메모리에 메시지를 전달해준 것이 없으므로 shared\_stuff2->written\_by\_server = 0으로 초기화해준다.
* **주석 (3):** 이 부분은 메인문의 메인 쓰레드에서 receiveFromClient1SendToClient2()과 receiveFromClient2SendToClient1()으로 쓰레드를 생성하게 된다. 따라서 한 프로세스당 총 3개의 쓰레드가 동시에 병렬적으로 실행되게 된다.
* **주석 (4):** 이 부분은 쓰레드 2개가 종료되길 기다리는 코드이다.
* **주석 (5):** 서버 프로세스에 부착되었던 2개의 공유메모리를 더 이상 사용하지 않으므로 shmdt() 함수를 통해 떼어낸다.

**receiveFromClient1SendToClient2 함수**



* main 문에서 receiveFromClient1SendToClient2 () 함수로 쓰레드가 분기하게 된다.
* **while(1):** 클라이언트1이 서버와 공유하는 공유메모리에 ‘쓰기’작업을 하면 written\_by\_1 값이 1로 변경되므로 while문 안에서 shared\_struff -> written\_by\_1 값이 1인지 끊임없이 검사한다.
* **if(shared\_stuff1 -> written\_by\_1 == 1):** 쓰레드가 동작하면서 while문 안에서 실시간으로 계속 클라이언트1 프로세스와 서버간에 공유하는 메모리에 클라이언트1이 ‘쓰기’를 하였는지 검사를 한다.
* **if(shared\_stuff1 -> flag == 1):** 이 때, written\_by\_1 값이 1이어서 서버가 공유메모리에 ‘쓰기’ 작업을 하였더라도 server가 ‘읽기’ 작업을 이미 하였을 수도 있다. 따라서 클라이언트가 ‘쓰기’ 작업을 하였지만 서버가 ‘읽기’ 작업을 하여 공유메모리가 비어 있는지 확인을 할 필요가 있다. 따라서 flag 값이 1인지 확인을 한다. (flag가 1이면 공유메모리에 ‘쓰기’ 작업이 되어있지만 ‘읽기’작업은 하지 않아서 공유메모리가 차 있다는 뜻이다. )
* **p(semid), v(semid):** 공유메모리에 접근하기 전에 세마포어와 동기화 툴을 사용해 공유메모리에 다른 쓰레드가 동시에 접근하여 읽어가거나 쓰지 못하게 막을 필요가 있다. 따라서 임계영역인 아래 빨간 코드를 앞 뒤로 세마포어의 p연산과 v연산을 해줄 필요가 있다.
* **strncpy(shared\_stuff2->message, shared\_stuff1->message, BUFFER\_SZ); //임계영역, 시큐어 코딩 준수**

공유메모리의 메시지에 접근하는 임계영역(critical section) 부분이다. 공유메모리1의 배열 message에 ‘쓰기’ 작업 되어있는 문자열을 복사하여 공유메모리2의 배열 message에 복사한다. 시큐어 코딩 가이드를 준수하여 복사하는 크기를 BUFFER\_SZ 1024바이트로 지정해주어 버퍼 오버플로우를 방지한다.

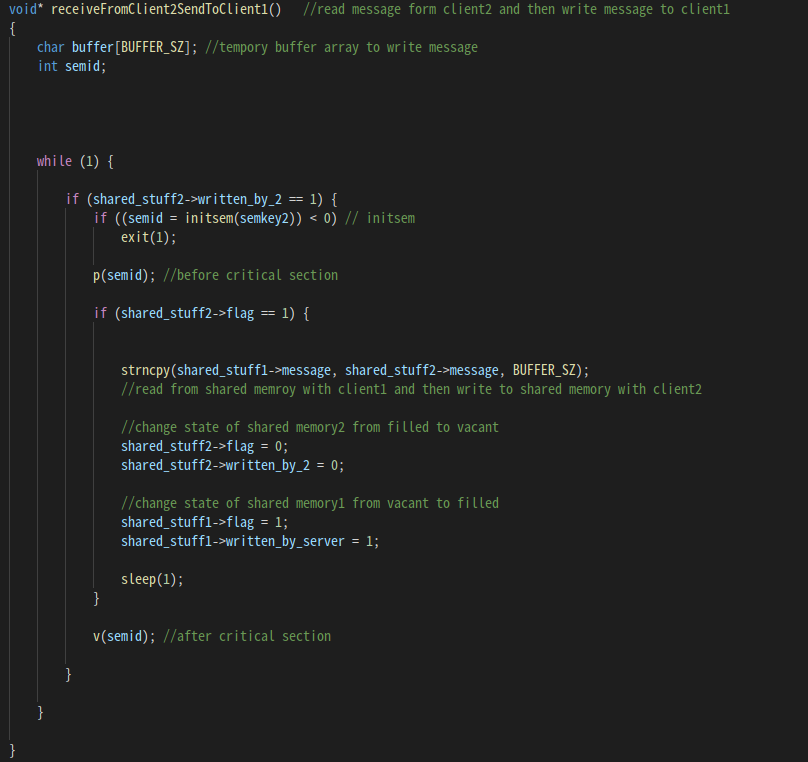
* **shared\_stuff1->flag = 0; shared\_stuff1->written\_by\_1 = 0; //임계영역**

클라이언트1이 공유메모리에 작성한 메시지를 서버가 읽어갔기 때문에 다시 written\_by\_1 = 0으로 초기화해주어 ‘클라이언트1은 공유메모리에 쓰기 작업을 하지 않았다’라고 상태를 바꿔줘야 한다. 또한 flag = 0 으로 초기화해주어 ‘공유메모리는 더 이상 읽을 것이 없으므로 비었다’라고 공유메모리의 상태를 바꿔준다.

* **shared\_stuff2->flag = 1; shared\_stuff2->written\_by\_server = 1; //임계영역**

서버가 클라이언트2와의 공유메모리에 클라이언트1의 공유메모리에서 읽은 것을 복사하여 쓰기 작업을 하였기 때문에 written\_by\_server = 1로 값을 변경해주다. 따라서 ‘서버가 공유메모리2에 쓰기 작업을 하였으니 빨리 누군가 읽어줘라’라고 공유메모리의 상태를 바꿔주게 된다. 또한 서버가 쓰기 작업을 함으로써 공유메모리는 비어있는 상태에서 찬 상태가 되므로 flag = 1 으로 값을 변경해주어 ‘공유메모리는 현재 차 있다’라고 상태를 바꿔준다.

**receiveFromClient2SendToClient1 함수**



* main 문에서 receiveFromClient2SendToClient1 () 함수로 쓰레드가 분기하게 된다.
* **while(1):** 클라이언트2와 서버가 공유하는 공유메모리에 ‘쓰기’작업을 하면 written\_by\_2 값이 1로 변경되므로 while문 안에서 shared\_struff -> written\_by\_2 값이 1인지 끊임없이 검사한다.
* **if(shared\_stuff2 -> written\_by\_2 == 1):** 쓰레드가 동작하면서 while문 안에서 실시간으로 계속 클라이언트2 프로세스와 서버간에 공유하는 메모리에 클라이언트2가 ‘쓰기’를 하였는지 검사를 한다.
* **if(shared\_stuff2 -> flag == 1):** 이 때, written\_by\_2 값이 1이어서 클라이언트2가 공유메모리에 ‘쓰기’ 작업을 하였더라도 server가 ‘읽기’ 작업을 이미 하였을 수도 있다. 따라서 클라이언트가 ‘쓰기’ 작업을 하였지만 서버가 ‘읽기’ 작업을 하여 공유메모리가 비어 있는지 확인을 할 필요가 있다. 따라서 flag 값이 1인지 확인을 한다. (flag가 1이면 공유메모리에 ‘쓰기’ 작업이 되어있지만 ‘읽기’작업은 하지 않아서 공유메모리가 차 있다는 뜻이다. )
* **p(semid), v(semid):** 공유메모리에 접근하기 전에 세마포어와 동기화 툴을 사용해 공유메모리에 다른 쓰레드가 동시에 접근하여 읽어가거나 쓰지 못하게 막을 필요가 있다. 따라서 임계영역인 아래 빨간 코드를 앞 뒤로 세마포어의 p연산과 v연산을 해줄 필요가 있다.
* **strncpy(shared\_stuff1->message, shared\_stuff2->message, BUFFER\_SZ); //임계영역, 시큐어 코딩 준수**

공유메모리의 메시지에 접근하는 임계영역(critical section) 부분이다. 공유메모리2의 배열 message에 ‘쓰기’ 작업 되어있는 문자열을 복사하여 공유메모리1의 배열 message에 복사한다. 시큐어 코딩 가이드를 준수하여 복사하는 크기를 BUFFER\_SZ 1024바이트로 지정해주어 버퍼 오버플로우를 방지한다.

* **shared\_stuff2->flag = 0; shared\_stuff2->written\_by\_2 = 0; //임계영역**

클라이언트2가 공유메모리에 작성한 메시지를 서버가 읽어갔기 때문에 다시 written\_by\_2 = 0으로 초기화해주어 ‘클라이언트2는 공유메모리에 쓰기 작업을 하지 않았다’라고 상태를 바꿔줘야 한다. 또한 flag = 0 으로 초기화해주어 ‘공유메모리는 더 이상 읽을 것이 없으므로 비었다’라고 공유메모리의 상태를 바꿔준다.

* **shared\_stuff1->flag = 1; shared\_stuff1->written\_by\_server = 1; //임계영역**

서버가 클라이언트1와의 공유메모리에 클라이언트2의 공유메모리에서 읽은 것을 복사하여 쓰기 작업을 하였기 때문에 written\_by\_server = 1로 값을 변경해주다. 따라서 ‘서버가 공유메모리1에 쓰기 작업을 하였으니 빨리 누군가 읽어줘라’라고 공유메모리의 상태를 바꿔주게 된다. 또한 서버가 쓰기 작업을 함으로써 공유메모리는 비어있는 상태에서 찬 상태가 되므로 flag = 1으로 값을 변경해주어 ‘공유메모리는 현재 차 있다’라고 상태를 바꿔준다.

### **Message Passing**

**사용할 주요 구조체와 변수**

**#define BUFFER\_SIZE 1024 // 총 버퍼의 크기**

**#define SERVER 1 // 서버의 기본 번호**

**#define CONNECT 32768 +1 // 최대 pid수 이상인 CONNECT 번호**

메시지큐를 통해 교환될 메시지들을 담을 버퍼의 총 사이즈는 1024이고 서버의 고유번호는 사용자의 pid의 범위(2~32768)에서 벗어난 1번으로 지정한다. 또한 클라이언트의 연결메시지를 뜻하는 CONNECT 번호도 32768+1로 중복되지 않도록 설정한다.

**typedef struct MessageType**

**{**

**long mtype;**

**char data[BUFFER\_SIZE];**

**long source;**

**} Message\_t; // 메세지 큐에 저장할 값**

**메시지큐를 통해 교환될 메시지의 형태를 담은 구조체**이다.

먼저 mtype으로 메시지타입을 long으로 지정하여 메시지들을 구분하여 메시지큐에 넣도록 하였고 char형 배열인 data를 BUFFER\_SIZE만큼 만들어 데이터를 교환할 수 있게 하였다.

그리고 source에 출발지의 pid를 담아 서버에서 source값을 확인하여 처리해줄 수 있게 설정했다.

**typedef struct Room {**

**long clients[2];**

**} Room; // 클라이언트들을 저장할 구조체**

서버에서 클라이언트들을 관리하기위해서 만든 구조체로 클라이언트의 pid들을 clients배열에 담아 Room으로 묶어 보관한다.

아래는 **서버에서 전역변수**들이다.

**key\_t key\_id; // 메시지큐의 키를 저장할 변수**

**int log\_fds[2]; // 로그의 파일 디스크립터(익명파이프 사용)**

**char log\_message[100];**

key\_t 인 key\_id에는 메시지큐의 아이디 반환값이 들어갈 변수이고, log\_fds[] 배열에는 로그를 담을 파일의 파일 디스크립터가 들어간다. 마지막으로 log\_message[] 배열은 로그에 담길 String을 담을 버퍼이다.

* **msgQ\_server.c**

**메시지큐의 초기화 부분**

**key\_id = msgget((key\_t)60139, IPC\_CREAT | 0666);**

**if (key\_id == -1)**

**{**

**perror("msgget error : ");**

**exit(0);**

**}**

전역변수로 미리 선언해준 key\_id에 메시지큐를 초기화하고 그 id값을 대입해주는 부분이다.

메시지큐의 값은 60XXY형식에 맞게 지정하였고 IPC\_CREAT 플래그를 통해 새로운 메시지큐를 생성하거나 이미 있는 메시지큐의 확인자를 반환하도록 하였다. 그리고 메시지큐의 접근권한은 666으로 생성하였다.

메시지큐를 생성한 후 발생하는 반환값인 Key\_id의 값이 -1로 생성에 실패했을 때의 예외를 if문과 perror, exit함수를 활용하여 예외처리 하였다.

**서버에서 클라이언트들의 접속을 대기하는 함수**

**void receiveConnectionMessage(Room\* room)**

**{**

**int i;**

**Message\_t message;**

**memset(message.data, 0, BUFFER\_SIZE);**

**for (i = 0; i < 2; i++) {**

**if ((msgrcv(key\_id, &message, sizeof(message) - sizeof(long), CONNECT, 0)) < 0) {**

**printf("error\n");**

**}**

**else {**

**room->clients[i] = message.source;**

**printf("%d 번째 손님 id : %ld\n", i + 1, room->clients[i]);**

**snprintf(log\_message, strlen("%d 번째 손님 접속 id : %ld\n") ,"%d 번째 손님 접속 id : %ld\n", i + 1, room->clients[i]);**

**write(log\_fds[1], log\_message, strlen(log\_message));**

**}**

**}**

**sprintf(log\_message, "접속 완료\n");**

**write(log\_fds[1], log\_message, strlen(log\_message));**

**}**

클라이언트의 접속을 대기하고 room 구조체에 클라이언트들의 pid를 받아 넣어주는 역할을 수행한다. 미리 지정한 프로토콜에 따라 클라이언트가 접속하면 sendConnectionMessage() 함수를 통해 자신의 pid를 담은 메시지를 전송하고 receiveConnectionMessage()메서드에서 해당 메시지를 읽어 구조체 Room에 넣어준다. 현재 사용자의 수는 2명으로 2명이 접속하면 사용자 모두를 서버가 Room 구조체로 관리하게 되고 다음 흐름으로 넘어가게 된다.

**유저간 채팅기능의 주요 내용**

유저간 채팅기능의 주요 내용이다. 이 반복문은 문제를 출제하는 for문안에서 동작하여 문제를 맞출 때 까지 채팅기능을 수행해주도록 한다.

**void startUserChatting(Room\* room)**

**showCharacterImage(room, i + 1);**

**while (1) {**

**if ((msgrcv(key\_id, &message, sizeof(message) - sizeof(long), SERVER, 0)) < 0) {**

**printf("error\n");**

**}**

**//printf("\n\ntest------ msg.data= %s, answer = %s\n\n",msg.data, answer[i]);**

**if (!strncmp(message.data, answer[i],strlen(answer[i]))) {**

**sprintf(feedback,"\n%ld 유저의 정답입니다!!\n", message.source);**

**printf("%s", feedback);**

**strncpy(message.data, feedback, strlen(feedback));**

**for (j = 0; j < 2; j++) {**

**message.mtype = room->clients[j];**

**msgsnd(key\_id, &message, sizeof(message) - sizeof(long), 0);**

**}**

**break;**

**}**

**if (message.source == room->clients[0]) {**

**message.mtype = room->clients[1];**

**}**

**else {**

**message.mtype = room->clients[0];**

**}**

**msgsnd(key\_id, &message, sizeof(message) - sizeof(long), 0);**

**}**

먼저 showCharacterImage() 메서드를 호출하여 room 구조체안에 들어있는 각 클라이언트들에게 i번째 문제의 사진을 출력해준다. showCharacterImage() 메서드는 아래에서 추가 설명할것이다.

이후에는 채팅기능의 주요 반복문 흐름으로 들어간다.

메시지 큐에서 SERVER(1)로 들어온 메시지를 읽어 message구조체의 data배열에 저장한다. 이후strncmp함수를 활용하여 해당 메시지 내용을 answer 배열과 비교해 정답인지 확인하고 정답이라면 반복을 끝내 다음 문제로 넘어가거나 종료되도록 한다. 만약 정답이 아니라면 메시지의 source값을 확인하여 같은 방의 상대편 클라이언트에게 메시지가 전해지도록 if문을 활용해 message구조체의 mtype을 지정한 후 메시지를 전송한다. 그리고 다시 while문으로 돌아간다.

**이전에 채팅기능에서 호출된 이미지를 출력하는 함수의 주요부분이다.**

**void showCharacterImage(Room\* room, int num)**

**char imagepath[20];**

**snprintf(imagepath, strlen("quiz/%d.txt"),"quiz/%d.txt", num);**

**FILE\* fp = fopen(imagepath, "r"); // 파일을 읽기 모드(r)로 열기.**

**if (fp == NULL)**

**{**

**printf("error\n");**

**return;**

**}**

**memset(message.data, 0, BUFFER\_SIZE);**

**while (feof(fp) == 0) // 파일 포인터가 파일의 끝이 아닐 때 계속 반복**

**{**

**count = fread(message.data, sizeof(char), BUFFER\_SIZE - 1, fp); // 1바이트씩 1023번 읽기**

**printf("%s", message.data);**

**for (i = 0; i < 2; i++) {**

**message.mtype = room->clients[i];**

**msgsnd(key\_id, &message, sizeof(message) - sizeof(long), 0);**

**}**

**memset(message.data, 0, BUFFER\_SIZE); // 버퍼를 0으로 초기화**

**total += count; // 읽은 크기 누적**

**}**

이 메서드에서는 먼저 fopen함수를 통해 imagepath에 해당하는 경로(./quiz/\*.txt)의 텍스트 파일을 읽기모드로 읽어 fp 포인터에 저장한다. 이후 fread함수를 활용하여 버퍼 사이즈만큼 파일을 읽어 메시지 구조체의 데이터부분에 담고 해당 내용을 각 클라이언트들에게 송신해준다. 이러한 작업을 EOF가 나올 때 까지 반복해주어 아스키코드로 된 이미지 텍스트를 모든 사용자가 출력받을 수 있도록 해주는 함수이다.

**로그 작성 파이프 함수**

**void\* writeLogToTextFile() // 로그 작성 파이프함수**

**{**

**int length;**

**FILE\* fp = NULL;**

**pthread\_mutex\_t log\_mutex;**

**while (1) {**

**char buffer[100] = { 0, };**

**length = read(log\_fds[0], buffer, sizeof(buffer));**

**if(length == -1) {**

**printf("log read error");**

**}**

**pthread\_mutex\_lock(&log\_mutex);**

**if ((fp = fopen("log.txt", "a")) != NULL) {**

**fprintf(fp, "%s", buffer);**

**fclose(fp);**

**}**

**else {**

**printf("파일 열기 오류\n");**

**}**

**pthread\_mutex\_unlock(&log\_mutex);**

**}**

**return NULL;**

**}**

익명파이프를 활용해 로그를 작성하여 텍스트파일에 로그를 남겨주는 함수이다.

Log\_fds[1]와 log\_message를 통해 서버의 각 코드에서 파일로 로그를 입력시켜주고 쓰레드로 실행되는 이 메서드에서 입력받은 로그를 받아 파일에 출력시켜 로그를 기록하게 된다.

하나의 로그파일에 한꺼번에 접속하여 수정하는 경쟁상태가 발생하지 않도록 뮤텍스락을 활용하여 파일을 쓰는 동작은 하나의 실행흐름만이 원자적으로 진행시킬 수 있도록 하였다.

* **msgQ\_client.c**

**클라이언트가 켜지면 가장 먼저 수행되는 연결메시지 전송부분**이다.

**void sendConnectionMessage(MultipleArg\* mArg) {**

**Message\_t message;**

**message.mtype = CONNECT;**

**message.source = mArg->id;**

**msgsnd(mArg->fd, &message, sizeof(message) - sizeof(long), 0); // 연결 메시지 전송**

**}**

미리 define해놓은 CONNECT 번호를 통해 메시지큐에 자신의 pid를 담은 연결 메시지를 등록시킨다. 앞서 설명한 서버의 receiveConnectionMessage에서 해당 메시지를 읽어 클라이언트의 pid를 저장하게 된다. 이러한 프로토콜을 통해 일정 수의 사용자가 모두 방에 접속하였을 때 게임을 시작할 수 있도록 하였다.

**메시지를 수신하는 주요 메서드의 주요** **부분**이다.

**void\* sendMessageToChattingServer(void\* multiple\_argment)**

**memset(message.data, '\0', BUFFER\_SIZE);**

**message.mtype = SERVER;**

**message.source = multipleArgment.id;**

**while (quit==0) {**

**memset(send\_buffer, '\0', sizeof(send\_buffer));**

**fgets(send\_buffer, sizeof(send\_buffer), stdin);**

**send\_buffer[strlen(send\_buffer)] = '\0';**

**strncpy(message.data, send\_buffer, strlen(send\_buffer));**

**msgsnd(multipleArgment.fd, &message, sizeof(message) - sizeof(long), 0);**

**}**

먼저 메시지 구조체를 초기화 시키고 메시지 전송방향을 서버로, 메시지의 출발지를 자신의 pid로 설정해준다.

버퍼를 초기화 하고 표준입력(키보드)으로부터 내용을 입력 받아 send버퍼에 저장한 후 해당 내용을 메시지 구조체의 data 배열에 담아 메시지 큐로 메시지를 전송해준다.

다음은 **메시지를 받는 함수의 주요 부분**이다.

**void\* receiveMessageFromChattingServer(void\* arg)**

**while (quit==0) {**

**if ((msgrcv(multipleArgment.fd, &message, sizeof(message) - sizeof(long), multipleArgment.id, 0)) > 0) {**

**if (!strncmp(message.data, "clear",strlen("clear"))) {**

**system("clear");**

**}**

**else if(!strncmp(message.data, "\_end\_",strlen("\_end\_"))){**

**quit=1;**

**}**

**else {**

**printf("%s", message.data);**

**}**

**}**

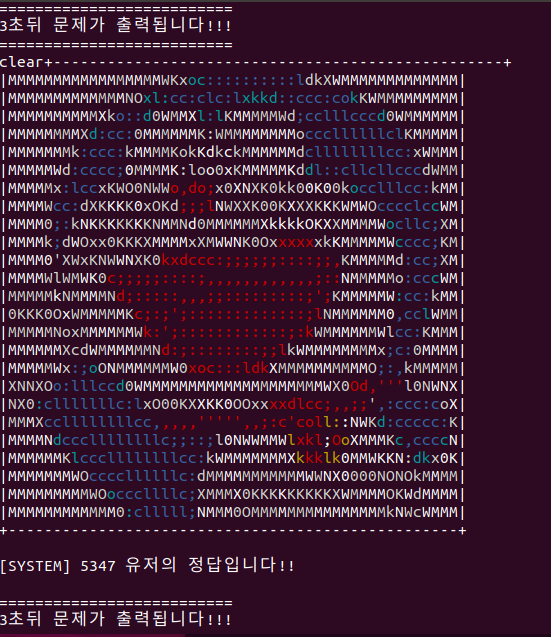
**}**

쓰레드를 통해 병행적으로 반복하며 메시지를 수신 받으면 해당 메시지가 서버에서 준 명령어인지, 사용자 간의 채팅 또는 출력될 이미지인지 판별하여 만약 명령어라면 clear(화면지우기)나 end(쓰레드 종료)를 수행하고 채팅메시지나 이미지라면 그것을 클라이언트의 화면에 출력하도록 하는 역할을 하는 쓰레드이다.

## **데모 시나리오**

1. Make 명령어를 통해 각각 FIFO, MesssagePassing, SharedMemory의 서버, 클라이언트 실행파일을 생성한다.
2. 오브젝트 파일이 저장되어 있는 디렉토리에 mkdir 명령어를 사용하여 fifo 디렉토리를 생성한다. fifo 디렉토리에는 프로그램 중 mkfifo를 통해 생성되는 fifo 파일을 저장한다.
3. 오브젝트 파일이 저장되어 있는 디렉토리에 mkdir 명령어를 사용하여 Quiz 디렉토리를 생성한다. Quiz 디렉토리에는 클라이언트에게 서버가 보낼 아스키 코드로 변환된 이미지 txt파일을 저장할 디렉토리이다.
4. ./”서버 실행파일” 을 먼저 실행한다.
5. 이후 ./”클라이언트 실행파일”을 실행한다.
6. 클라이언트가 2명이 되면 채팅이 가능하고 서버는 아스키코드로 변환된 이미지를 클라이언트에 전송한다.
7. 클라이언트는 전송받은 이미지를 화면에 출력한다.
8. 정답을 입력한다.
9. 설정되어 있는 Quiz를 모두 수행하였을 경우 server는 프로그램을 종료한다.

## **프로그램 실행결과 캡쳐 및 설명**

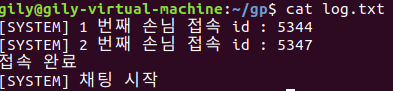


서버가 해당 문제를 보내주고 클라이언트의 화면에 전송되는 아스키코드로 변환된 이미지이다.



서버는 서버가 클라이언트에게 전송한 데이터를 기록한다. total은 서버가 클라이언트에게 보낸 데이터의 바이트의 크기를 의미한다.

채팅을 할 경우 채팅 기록을 출력하고, 이미지를 전송했을 경우 이미지를 출력한다.



서버가 기록하는 log.txt의 기록이다. 퀴즈내는 것을 제외하고 클라이언트의 상태를 기록한다.

## **현실적 제한요소 반영**

### **경쟁상태**

* **FIFO, MessageQueue**

서버에서 클라이언트의 접속이나 메시지 전송, 정답여부 등의 여러 정보들을 기록하는 로그파일을 생성하려 한다. 로그 파일에 정보를 출력해주는 것은 로그쓰기 쓰레드와 익명파이프를 통해 진행하는데 이때 로그쓰기 작업에 다른 여러 쓰레드들이 동시에 접근해서 특정로그의 쓰기 작업이 누락되는 것을 막기위해 동기화 툴을 사용한다.

사용한 동기화 툴은 **뮤텍스락**으로 특정 범위의 소스코드를 원자적으로 수행할 수 있도록 하여 동시에 쓰고 저장해서 발생하는 데이터의 누락이나 교착상태를 방지할 수 있다. 현재의 소스코드에서는 로그 쓰레드에서 log.txt에 fprintf해주는 부분에 뮤텍스락을 주어 실제로 파일에 내용을 기록할 때 한번에 한 쓰레드만 접근하여 로그를 기록하도록 하였다.

* **Shared Memory(공유 메모리)**

클라이언트1이 클라이언트2에게 메시지를 전달하려면 서버 프로세스가 중간에서 클라이언트1 프로세스와 클라이언트2 프로세스가 서로 전달하려는 메시지를 받아서 전달해줘야 한다. 그러기 위해서는 총 2개의 공유 메모리가 필요하고 이 공유메모리에 클라이언트의 쓰레드들과 서버의 쓰레드들이 동시에 접근하려고 한다.

따라서 이로 인한 경쟁상태(race condition)을 막기 위해서는 **‘세마포어’로 동기화** 하는 것이 필수였다. 여러 개의 쓰레드들이 공유메모리에 접근하여 ‘읽기 작업’ 또는 ‘쓰기 작업’을 할 때, 임계 영역(critical section)의 앞 뒤에 세마포어 p연산과 v연산을 해주었다. 둘 이상의 쓰레드가 입력이나 조작이 동시에 일어나 의도하지 않은 결과를 가져오는 경우를 막았다. 또한 공유메모리 안에 **flag**라는 변수에 쓰기가 필요한 경우는 값을 0, 읽기가 필요한 경우는 값을 1로 표시하여 공유메모리로 접근하는 쓰레드들의 순서를 조정했다.

### **성능을 높일 수 있는 방안**

1. **블록 단위 파일 읽기**

파일을 읽고 쓸 때 가장 효과적이라고 알려진 2의 배수이며 clock소요 시간을 감안했을 때 크기가 가장 무난한 1024크기의 버퍼를 사용했다. IPC로 통신할 때 해당 단위를 통해 한번에 1024바이트 블록 단위로 이미지를 아스키코드로 바꾼 값을 전송하도록 하였다.

1. **루프 안에서 최대한 많은 작업 처리 -> 루프가 도는 횟수 최소화함**

하나의 루프에서 누락된 작업이 있어 루프가 1번 더 도는 것보단 루프 안에서 처리해줘야 할 작업을 최대한 다하여 성능을 조금이라도 높이는 것이 좋다. 따라서 코드를 짤 때 루프 안에서 누락되는 처리 없도록 고민하면서 짬으로써 루프가 도는 횟수를 최소화하였다.

1. **함수의 인자로 구조체 포인터를 넘겨주기 -> 데이터 복사시간 단축시킴**

함수의 인자로 구조체를 통째로 넘겨주면 구조체 안의 모든 데이터를 포함한 전체 데이터를 복사해야 한다. 하지만 구조체를 가리키는 포인터를 넘겨주면 단순히 4바이트의 주소 값만 복사하면 되고 포인터를 통해서 구조체의 멤버 변수에 쉽게 접근 가능하다. 따라서 구조체 포인터를 함수의 인자로 넘겨주는 것이 성능이 더 좋고 효율적인 코드가 될 수 있으므로 코드를 작성함에 있어 구조체 포인터를 함수의 인자로 넘겨주도록 짜려고 노력하였다.

### **메모리 사용량을 줄일 수 있는 방안**

1. **배열의 사이즈를 적절한 사이즈로 설정한다.**

버퍼와 같이 배열로 관리하는 부분의 배열 사이즈를 너무 크지 않게 설정하여 사용하지 않는 부분까지 과잉으로 사용하지 않는다.

1. **포인터에 동적으로 메모리 영역을 할당 시 -> free( ) 함수로 반드시 해제**

malloc과 같은 메모리를 동적으로 할당하는 함수로 포인터 영역을 동적으로 예약해 줄 경우 반드시 free( ) 함수를 통해 사용한 메모리 영역을 반환해준다.

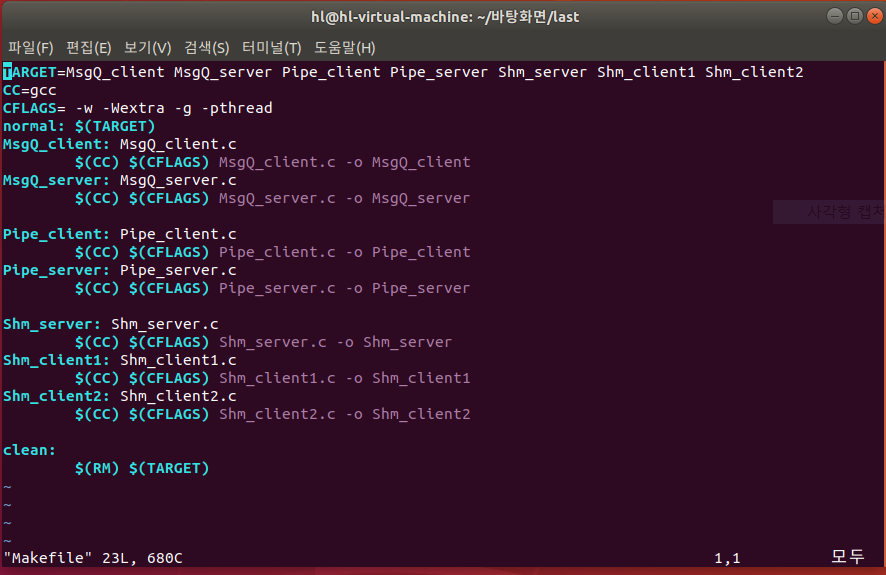
1. **pthread 사용 시 반드시 자원을 반환하는 함수 사용**

pthread\_create( )함수를 사용할 경우 반드시 pthread\_join( )과 pthread\_detach( )와 같은 함수를 이용하여 할당 받은 메모리를 반환한다.

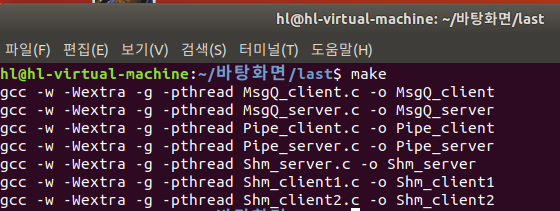
### **UNIX API 활용**

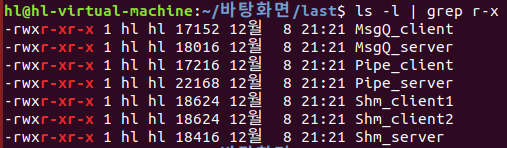
컴파일에 gcc 컴파일러를 사용하였고, gdb를 이용하여 core파일을 확인해보며 오류를 디버깅했다. 그리고 make 유틸리티를 사용하여 각기 다른 ipc를 사용한 파일을을 하나의 명령어를 통해 컴파일하여 실행파일을 만들 수 있게 하였다.

아래 사진은 make 유틸리티 사용을 위한 Makefile의 세부 내용이다.



컴파일러로는 gcc컴파일러를 사용하였다. 그리고 플래그들을 통해 원하는 정보들만을 출력하도록 하였으며 각 c파일에 대응하는 확장자가 없는 서버 / 클라이언트 실행파일을 nomal값인 make만 입력하여도 전부 컴파일하도록 만들었다.





Make를 통한 컴파일으로 입력된 명령어와 생성한 실행파일들이다.



실행파일을 제거할 때는 make clean으로 한꺼번에 생성한 실행파일들을 지울 수 있다.

### **Secure coding 기법 활용**

이번 프로젝트의 최종 보고서를 제출하기까지 끊임없이 코드의 효율성 및 보안 측면에서 수정 및 보완하면서 행정안전부에서 제시하는 **‘C 시큐어 코딩 가이드’**의 다음 부분을 참고하였다.



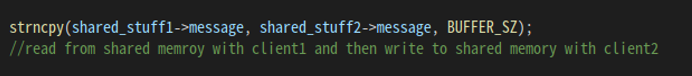
**제 1절** **입력데이터 검증 및 표현**

**8. 스택에 할당된 버퍼 오버플로우 …p44**

**제 4절 에러처리**

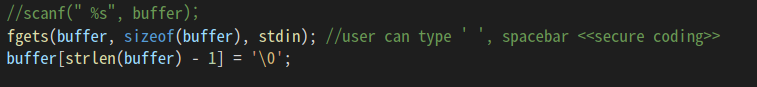
1. **오류 메시지를 통한 정보 노출 …p152**
2. **오류상황 대응 부재 …p156**
3. **적절하지 않은 예외처리 …p159**
4. **버퍼 오버플로우 방지(제1절 입력데이터 검증 및 표현)**

**(공유 메모리)**



* 공유메모리에 복사한 문자열을 복사하는 과정으로 원래는 **strcpy()**를 사용하였지만 **strncpy()**로 바꾸어 크기를 선언해주어 버퍼 오버 플로우를 방지하였다.

**(FIFO, 메시지큐, 공유메모리)**



* 클라이언트 터미널에서 사용자에게 입력받는 부분으로 원래는 **scanf()**를 이용해 사용자에게 입력 받도록 구현하였지만 **fgets()**를 사용해 입력 받는 크기를 명시해주었다.

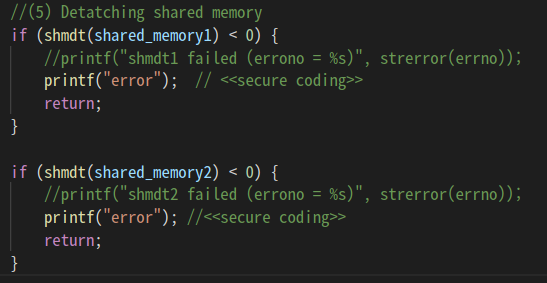
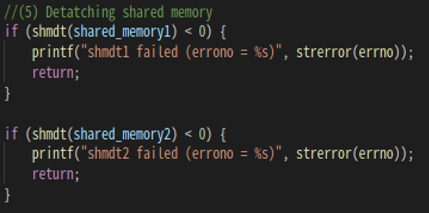
**(FIFO, 메시지큐, 공유메모리)**



* 파일 경로를 path라는 변수에 저장하는 부분으로 원래는 **sprintf()**를 사용하였지만 **snprintf()**를 사용해 크기를 저장하는 문자열의 크기를 함수에 전달해주었다.

1. **오류 메시지를 통한 정보 노출 방지(제4절 에러처리)**

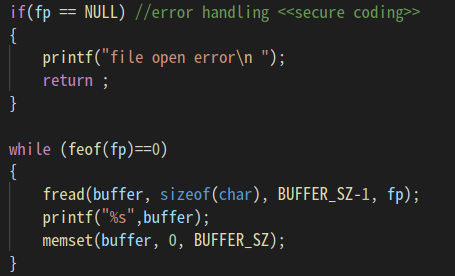
**공유메모리 (수정 전)** **(수정 후)**



서버 프로세스에서 생성한 공유메모리 2개를 다 사용한 후 detach하는 부분으로 실패할 경우 에러 처리를 하였다. 원래는 sterror(errno)을 터미널 창에 출력함으로써 사용자도 에러의 구체적 원인을 볼 수 있게 하였지만 간단한 에러 메시지만 출력되도록 함으로써 정보 노출을 방지하였다.

1. **오류상황에 대한 대응(제4절 에러처리)**

**(공유 메모리, FIF0, 메시지큐)**



원래는 파일을 open하는 부분에서 에러처리를 하지 않았지만 시큐어 코딩 가이드를 참고하여 발생할 수 있는 모든 오류 상황에 적절한 처리를 해주도록 바꾸었다. 따라서 파일 열기에 실패하게 되면 “file open error”가 출력된다.

# 팀워크

## **역할 및 작업내용**

우리 13조는 팀장 정하림과 2명의 팀원 정수경, 이길형으로 총 3명으로 구성되어있다.

Shared Memory는 정수경 학생, Message Passing은 정하림 학생, Named Pipe는 이길형 학생이 전담하여 프로그램을 작성하였다. 각자 담당하는 IPC 기법에 대해 학습하고 Linux와 우분투 사용법을 학습하도록 하였다. 담당하는 기법에 따라 코드를 작성하는 부분은 각자 수행하였고 공통적으로 사용할 수 있는 함수는 같이 모여 수행역할을 작성하였다.

유닉스 프로그래밍 경험이 있는 정하림 팀장이 리눅스 유닉스 사용법과 명령어에 대해 알려주며 과제를 진행하였다. 만약 모두가 모르는 부분이 나오는 경우 개별조사를 수행하고 팀원들끼리 대화를 통해 알아낸 정보를 공유하는 방법으로 문제해결을 하였다.

Make 유틸리티와 쓰레드 환경 코드의 전반적 오류를 검토하고 사전 설계 환경에 대해 조사하는 것은 정수경 팀원이 담당하였다. 정수경 팀원은 셋 중에서 코드를 볼 때 세밀한 부분까지 생각하기 때문에 다른 팀원이 놓친 부분이나 오류 발생부분을 잘 찾기 때문에 이 역할을 수행하였다.

보고서와 PPT의 전반적인 틀은 이길형 팀원이 담당하였다. 보고서와 PPT 디자인 프레임을 만들었다.

과제는 가능하면 개별적으로 진행하지 않고 모두가 과제를 하고 있을 때 진행하는 것으로 목표로 삼고 진행하였고, 가급적이면 온라인으로 팀 프로젝트를 진행하는 것이 아닌 모여서 팀 프로젝트를 진행할 수 있도록 하였다.

프로그램 소스코드 작성 중에 오류가 발생하는 경우 해당 프로그램 작성자와 나머지 팀원이 같이 코드를 보며 토의하는 방식으로 수정할 부분과 오류를 발생시킬 수 있는 부분을 찾아가며 프로그램을 만들었다. 혼자서 오류 발생 부분을 찾는 것보다 다양한 관점에서 코드를 보며 보완하고 수정하는 과정이 더 효율적이었으며, 코드를 작성할 때 주의할 점을 상기할 수 있었기 때문에 팀 프로젝트를 한층 더 능률적으로 할 수 있었다.

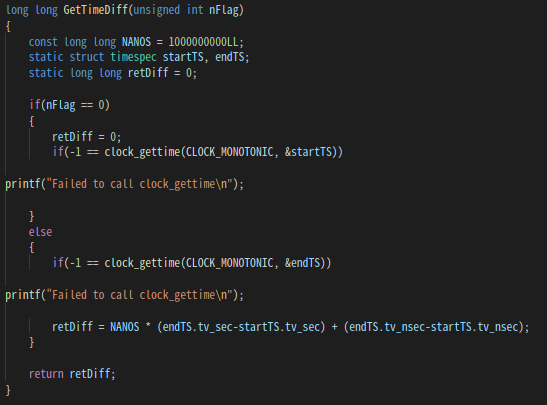
# 결론 및 총평

## **결론**

지금까지 “채팅 기반 인물 맞추기 게임”을 3가지 서로 다른 IPC 기법으로 구현을 해보았다. 이번 설계 프로젝트를 여기서 마치는 것이 아니라 3가지 IPC기법을 사용했을 때 성능분석을 하여 각IPC기법의 장단점을 분석해 보도록 하였다.

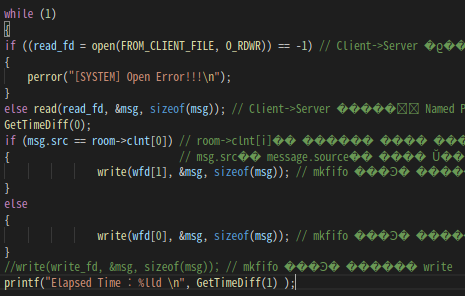
**GetTimeOff 함수**

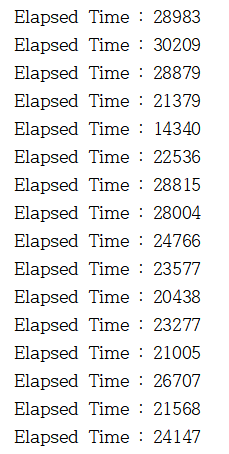
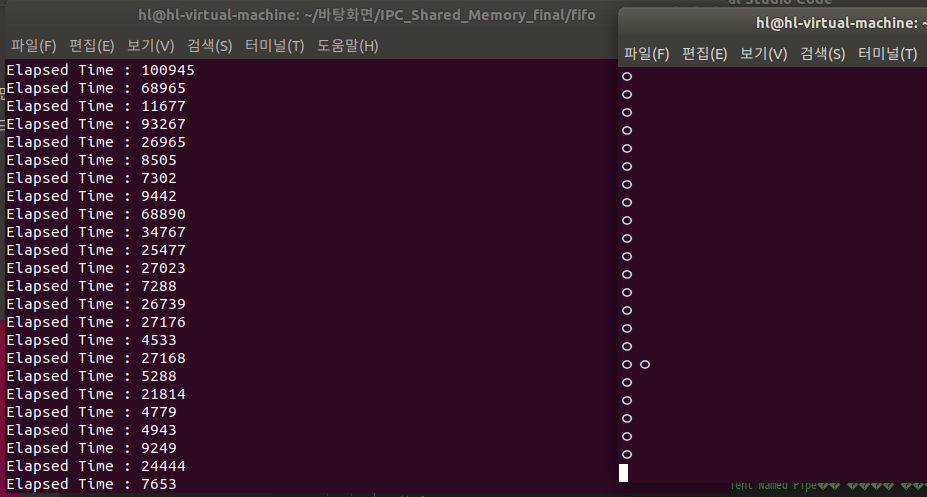
우선 실행시간을 측정하는 GetTimeOff 함수를 작성하였다. 사용방법으로는 시간을 측정할 코드부분의 앞과 뒤에 GetTimeOff(0)과 GetTimeOff(1)을 배치한 한 후 코드를 돌린다.



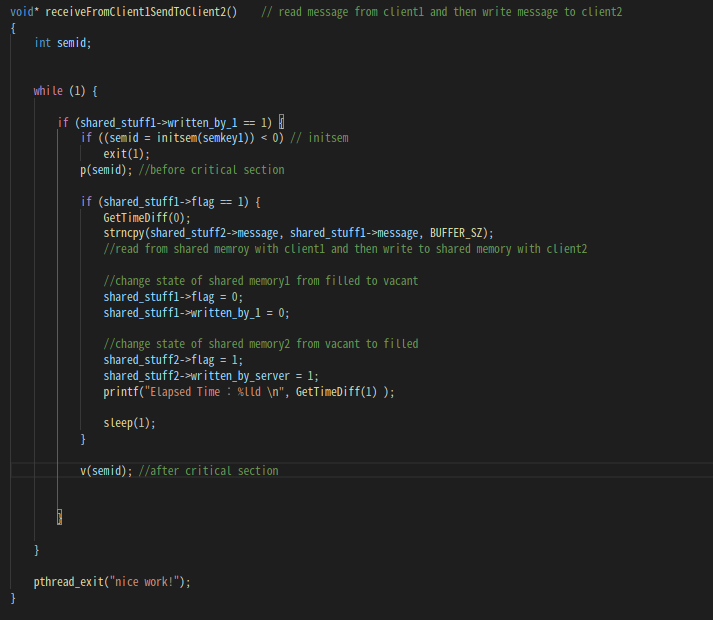
우리의 프로그램에서는 IPC기법의성능을 분석하기 위해, 3가지 버전의 프로그램에서 클라이언트1이 클라이언트2에게 서버를 통해 메시지를 보낸다고 했을 때, 클라이언트2에 메시지가 도달하는 시간을 분석하도록 하였다. GetTimeOff 함수를 각 프로그램마다100번씩 시행하여 평균값을 구해보았다. GetTimeOff 함수 코드를 삽입한 코드와 그 결과는 아래와 같다.

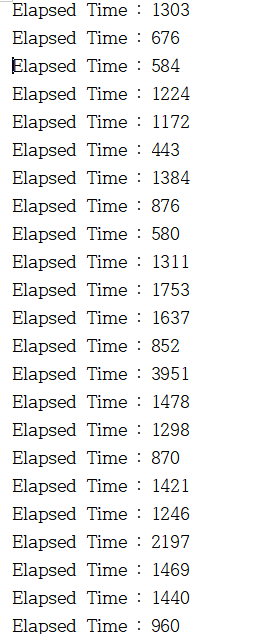
**Named Pipe 성능 측정**

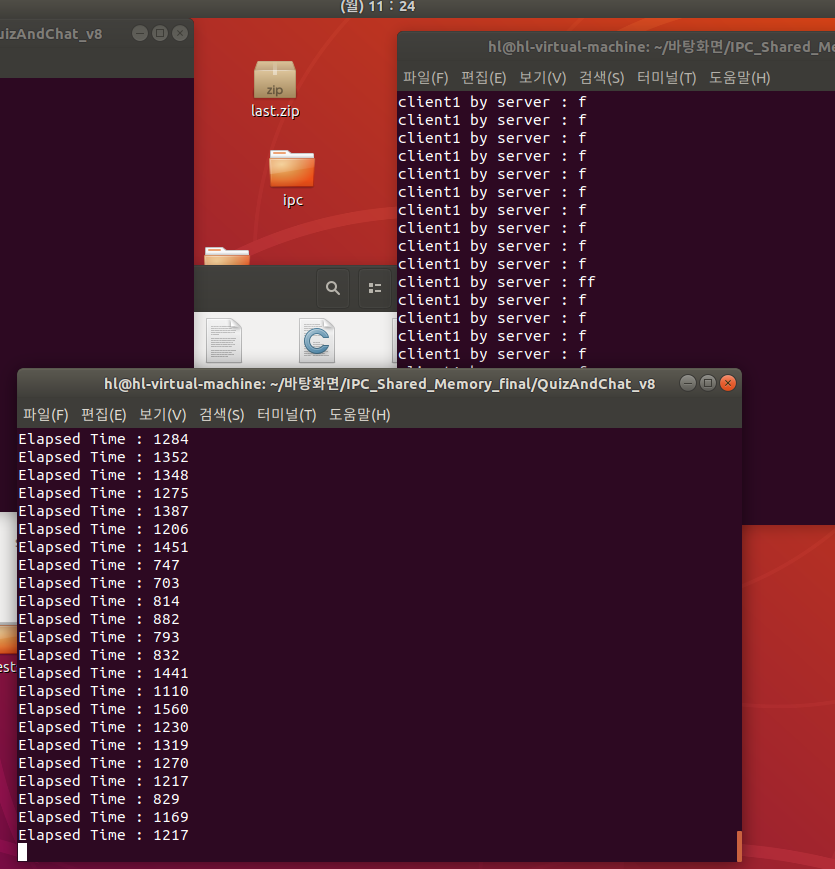




**Shared Memory 성능 측정**

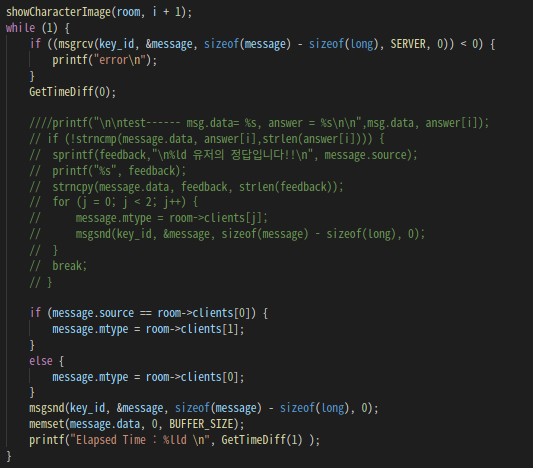


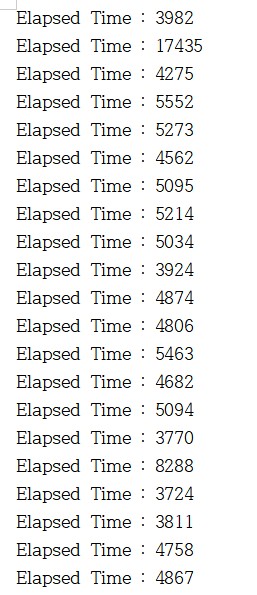


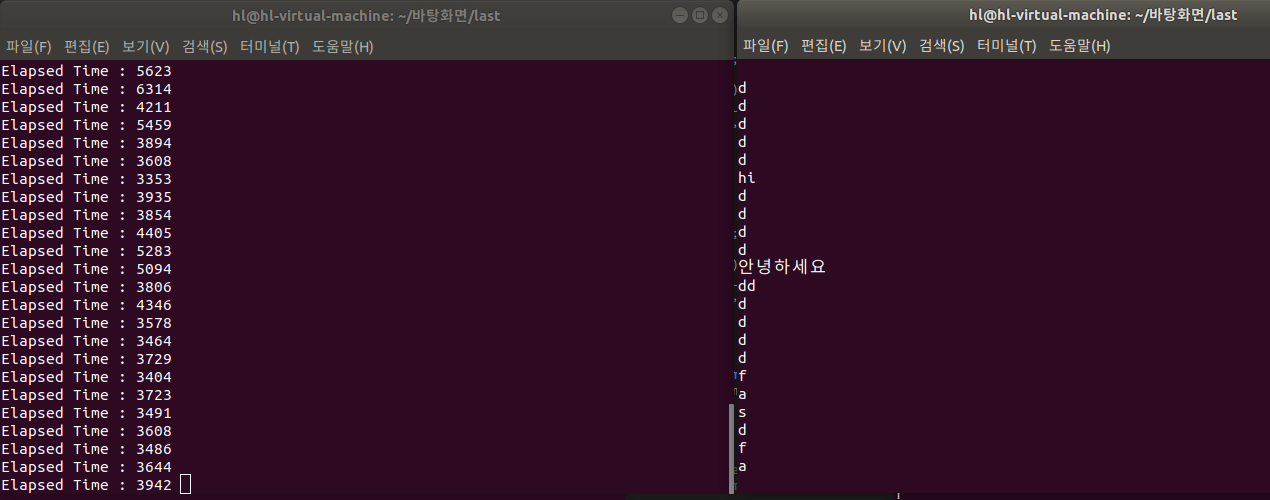


**서버의 터미널 창에 클라이언트1 용 터미널 창에서 글자 하나를 입력해서 서버를 통해 클라이언트2에 메시지를 보낼 때의 소요시간이 “Elapsed Time: 소요시간”로 출력되도록 하였다.**

**Message Passing 성능 측정**







**100번 측정 후 속도 평균값 비교**



**100번의 측정결과를 엑셀에 담아 보았다. 각 IPC기법을 통해 클라이언트1에서 서버를 통해 클라이언트2에게 메시지가 전달되는 평균 속도를 계산한 결과 파이프는 28914.34 ns, 메시지큐는 8444.25 ns, 공유메모리는 1226.62ns가 나왔다. 따라서 성능 측면에서 Shared Memory, Message Passing, FIFO 순서로 좋다는 결론을 얻었다.**

**FIFO < Message Passing < Shared Memory**

이러한 결론에 대한 이유에 대해 팀원들이 조사를 해본 결과, FIFO는 파일에 직접 접근해야 되기때문에 가장 느리고 Message Passing은 커널 모드와 유저 모드를 번갈아 바꾸기 때문에 느리다. 공유메모리가 가장 빠른 이유는 공유메모리만 다른 IPC기법들과달리시스템콜을 사용하지 않고 포인터로 메모리에 직접 접근함으로써 오버헤드가 최소화된다.

## **총평**

**정수경**

이번 텀 프로젝트를 통해 많은 것을 배운 것 같다. 그동안 과제를 할 때는 조건과 요구사항에 맞춰 프로그램을 짜고 끝내는 게 대부분이었다. 하지만 이번 텀 프로젝트에서 처음으로 요구사항과 조건에 맞춰 코드를 짜는 거에서 그치지 않고 시큐어 코딩을 준수해서 코드를 짜보았다. 또한 어떻게 하면 성능을 높이고 메모리를 효율적으로 쓸지 고민을 하면서 코드를 짜보았다. 그러다 보니 처음 짠 코드를 계속해서 끊임없이 수정하고 최적의 코드로 바꾸려고 노력했다. 이번 텀 프로젝트는 멀티쓰레드, 동기화, IPC 기법을 통한 클라이언트와 서버 구조 크게 이 3가지의 요구사항을 만족하여 설계를 하고 또 3명 모두 다 구현해내야 돼서 많이 힘들고 고생도 많이 하였다. 하지만 우리 팀이 처음 설계했던 프로그램을 요구사항에 맞춰 짜내고 나니 정말 뿌듯하고 보람찼으며 내가 맡은 IPC 기법에 대해서는 누구보다 제대로 알고 넘어갈 수 있어서 좋았다. 또한 프로그램을 만들면서 리눅스 환경과 API 툴에 익숙해지고 코딩 실력 또한 올라간 것 같아 정말 뜻깊은 텀프로젝트였다.

**정하림**

설계과제물로 다른 팀들의 수준에 비해 비교적 쉬운 주제를 정했다고 생각했는데 IPC 기법을 활용하는 것이 처음이라 많이 해메였다. 그리고 생각보다 리눅스의 gcc환경에서 디버깅을 하거나 파일을 옮겨가며 작업하는 것에 많은 시행착오를 겪었고 때문에 정말 제때 내고 싶었는데 지연제출까지 하게 되었다. 이번 프로젝트를 진행하며 혼자 작성할 때도 해당되지만 여러명이서 함께 작작업할 때 특히 더 설계부분이 중요하다는 것을 깨닳았다. 또한 과제가 미리 주어지면 마감기한이 다가오기 전에 미리 끝내놓아야한다는 것을 뼈저리게 느꼈다. 이러한 후회도 많지만 클라이언트와 서버구조를 이해하고 여러 쓰레드를 사용해보고 ipc기법을 통해 다른 프로세스와 통신해보면서 여러 함수나 구조적 지식들을 자연스럽게 익히게 되었다. 또한 gdb나 make같이 이전에 전혀 사용해보지 않았던 gnu툴들을 사용해보면서 리눅스 환경에 더 친숙해 졌던 과제였던 것 같다.

물론 제 시간 안에 끝내지 못한 것은 아쉽지만 팀원들 또한 모두 열정을 가지고 항상 같이 모여서 고민하고 막힌 부분에 대해 질문하고 소스코드나 보고서를 작성하면서 협업의 즐거움 또한 느낄 수 있었다. 비록 결과물의 완성도는 약간 떨어지더라도 작성과정에서 느낀 것들은 다음 차례의 문제를 해결하는 과정에서 나를 더 성숙하게 해줄 것이라 생각하고 앞으로도 이러한 고민거리 많은 문제들을 해결해보고 싶다.

**이길형**

Linux에서 GCC 컴파일을 이용한 C프로그램 작성은 처음이기 때문에 과제를 시작하며 많은 어려움이 있었다. 기존의 Windows에서 GUI(Graphic User Interface)만을 사용하며 프로그램을 작성하다가 리눅스의 CLI(Command Line Interface) 방식을 처음 접하다 보니 많은 부분이 처음이었다. Visual Studio에서는 없는 POSIX와 같은 함수들을 다루기 위해 Linux에서 구동해보는 방식이 어색하였기 때문에 익숙해지는데 많은 시간이 걸렸다.

내가 담당하기로 한 Named Pipe부분에 대해서 코드를 작성하며 많은 부분을 이해할 수 있었다. 기존의 이론으로만 이해하고 있던 Named Pipe를 이번 과제를 수행하며 많이 알게 된 것 같다. 기본적으로 Named Pipe의 기본 원리는 fifo 파일로 다른 프로세스와 연결하고 fifo 파일을 open하고 read와 write하는 것을 기본으로 동작하는 것을 코드로 직접 작성하며 많은 부분을 이해하였다.

해당 과제를 통해 Make 유틸리티의 사용법도 익히고 프로세스 간 통신도 직접 눈으로 확인하며 배울 수 있어서 실력향상에 많은 도움이 되었다고 생각한다. 처음에는 막막할 정도로 어려운 과제였지만 수행하고 보니 서버 클라이언트 구조에 대한 이해도도 높아지고 UNIX & Linux의 사용법도 익힐 수 있었고 멀티 쓰레드를 직접 사용하며 쓰레드가 프로그램에서 어떤 식으로 흐름을 형성하고 병행적으로 수행되는지 알았다. 병렬적 프로그램과 병행적 프로그램에 대한 차이도 정확하게 알 수 있었다. Windows에서만 코딩을 하였더라면 저수준 입출력을 사용할 일이 없었을 텐데 로우레벨 함수를 직접 사용하며 프로그래밍적 이해가 한층 더 높아졌다고 생각한다.

프로그램을 설계할 때 file descriptor와 메모리 구조를 이해하며 코드를 작성하고, 멀티 프로세스와 멀티 쓰레드 환경을 구성하는 프로그램이라면 공유자원과 동기화를 생각하며 경쟁상태를 고려하며 프로그램을 만들어야 함을 느꼈다. 프로세스 간 통신인 IPC 기법을 이해하며 프로세스와 쓰레드 그리고 서버와 클라이언트 구조에 대해 이해하며 학습할 수 있는 어렵지만 좋은 과제였다고 생각한다.

# 부록

## **코드**

### **Named Pipe**

**[ Fifo\_server.c ]**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <unistd.h>**

**#include <string.h>**

**#include <pthread.h>**

**#include <sys/msg.h>**

**#include <fcntl.h>**

**#define FROM\_CLIENT\_FILE "./fifo/from\_client" // Clinet -> Server 데이터를 전달하는 Named pipe 경로**

**#define TO\_CLIENT\_FILE "./fifo/to\_client\_" // Server -> Client 데이터를 전달하는 Named pipe 경로**

**#define LOG\_FILE "fifo\_log.txt" // 로그내용을 기록하는 파일의 경로**

**#define USER\_ID\_SIZE 21**

**#define BUF\_SIZE 1024 // 버퍼 사이즈**

**#define SERVER 1**

**#define CONNECT 32768 +1**

**#define TOTAL\_QUIZ 3 //문제 수**

**#define PATH\_SIZE 50 // 경로명의 최대 사이즈**

**typedef struct msgtype {**

**long mtype;**

**char data[BUF\_SIZE]; // 데이터의 Data**

**long src;**

**}msg\_t; // mkfifo를 통해 전달할 데이터 구조체**

**typedef struct Room {**

**long clnt[2];**

**} Room; // 접속한 Client를 수를 확인하고 Client의 PID를 저장하는 구조체**

**void receiveConnectionMessage(Room\* room); // Client가 Server에 접속을 확인하는 함수**

**void startUserChatting(Room\* room); // 채팅과 게임을 관리하는 함수**

**void showCharacterImage(Room\* room, int num); // 문제를 출력해주는 함수**

**void\* writeLogToTextFile(); // 로그를 기록하는 쓰레드의 handler 함수**

**int log\_fds[2];**

**char log\_msg[100];**

**int main()**

**{**

**pthread\_t chat\_thread, write\_log\_thread;**

**Room room;**

**void\* chat\_thread\_return;**

**int i;**

**int fd;**

**pipe(log\_fds); // 로그기록 파이프 생성**

**pthread\_create(&write\_log\_thread, NULL, writeLogToTextFile, NULL); // 로그를 기록하기 위한 쓰레드 생성**

**receiveConnectionMessage(&room); //Client의 연결을 기다리고 Client가 처음 접속 했을 때 정보 데이터를 전달받을 함수**

**pthread\_create(&chat\_thread, NULL, startUserChatting, &room); // 채팅과 게임을 시행하고 관리하는 쓰레드를 생성한다.**

**pthread\_join(write\_log\_thread, NULL);**

**pthread\_join(chat\_thread, &chat\_thread\_return);**

**return 0;**

**}**

**void receiveConnectionMessage(Room\* room) // Client가 Server에 접속을 확인하는 함수**

**{**

**int i;**

**int read\_fd, write\_fd;**

**msg\_t msg;**

**char path[PATH\_SIZE];**

**system("clear");**

**mkfifo(FROM\_CLIENT\_FILE, 0666); // Client로부터 데이터를 받기 위한 Named Pipe file 생성**

**for (i = 0; i < 2; i++) // 2명의 유저 모두 참여했는지 체크**

**{**

**memset(msg.data, 0, BUF\_SIZE); // 버퍼 초기화**

**if ((read\_fd = open(FROM\_CLIENT\_FILE, O\_RDWR)) == -1) // Client로부터 데이터를 받기 위한 Named Pipe file 오픈**

**{**

**perror("[SYSTEM] Open Error!!!\n"); // 에러처리**

**}**

**if (read(read\_fd, &msg, sizeof(msg)) == -1) // Client가 Server에 접속을 체크하기 위해 Named Pipe로부터 read**

**{**

**perror("[SYSTEM] Read Error!!!\n"); // 에러처리**

**}**

**room->clnt[i] = msg.src;**

**printf("[SYSTEM] %d 번째 손님 id : %ld\n", i + 1, room->clnt[i]);**

**sprintf(log\_msg, "[SYSTEM] %d 번째 손님 접속 id : %ld\n", i + 1, room->clnt[i]);**

**write(log\_fds[1], log\_msg, strlen(log\_msg)); // 로그기록**

**sprintf(path, "%s%ld", TO\_CLIENT\_FILE, room->clnt[i]); // Server -> Client를 위한 Named pipe 파일인 mkfifo 파일명 설정**

**mkfifo(path, 0666); // Server -> Client를 위한 Named pipe 파일 생성**

**}**

**sprintf(log\_msg, "접속 완료\n");**

**write(log\_fds[1], log\_msg, strlen(log\_msg)); // 로그기록**

**}**

**void startUserChatting(Room\* room) // 채팅과 게임을 관리하는 함수**

**{**

**int read\_fd, write\_fd;**

**msg\_t msg;**

**int i, j;**

**char path[PATH\_SIZE];**

**char feedback[50];**

**char answer[3][20] = { "강호동\n","도라에몽\n","아인슈타인\n" }; // 정답순서**

**sprintf(log\_msg, "[SYSTEM] 채팅 시작\n");**

**write(log\_fds[1], log\_msg, strlen(log\_msg)); // 로그기록**

**for (i = 0; i < TOTAL\_QUIZ; i++)**

**{**

**strncpy(msg.data, "\n==========================\n3초뒤 문제가 출력됩니다!!!\n==========================\n",sizeof("\n==========================\n3초뒤 문제가 출력됩니다!!!\n==========================\n"));**

**for (j = 0; j < 2; j++)**

**{**

**memset(path, 0, PATH\_SIZE); // 경로명 path 변수 초기화**

**sprintf(path, "%s%ld", TO\_CLIENT\_FILE, room->clnt[j]); // Server->Client 하기위한 Named Pipe 파일명 변경**

**if ((write\_fd = open(path, O\_RDWR)) == -1) // Server->Client 하기위한 Named Pipe 파일 오픈**

**{**

**perror("[SYSTEM] Open Error!!!\n"); //오류처리**

**}**

**msg.mtype = room->clnt[j];**

**write(write\_fd, &msg, sizeof(msg)); // Server->Client Named Pipe를 통해 데이터 전송**

**}**

**sleep(3);**

**strncpy(msg.data, "clear",sizeof("clear")); // strcpy->strncpy 시큐리티 코딩 적용**

**for (j = 0; j < 2; j++)**

**{**

**memset(path, 0, PATH\_SIZE); // 경로명 path 변수 초기화**

**sprintf(path, "%s%ld", TO\_CLIENT\_FILE, room->clnt[j]); // Server->Client 하기위한 Named Pipe 파일명 변경**

**if ((write\_fd = open(path, O\_RDWR)) == -1) // Server->Client 하기위한 Named Pipe 파일 오픈**

**{**

**perror("[SYSTEM] Open Error!!!\n"); //오류처리**

**}**

**msg.mtype = room->clnt[j];**

**write(write\_fd, &msg, sizeof(msg)); // Server->Client Named Pipe를 통해 데이터 전송**

**}**

**showCharacterImage(room, i + 1); // 문제출력**

**while (1)**

**{**

**if ((read\_fd = open(FROM\_CLIENT\_FILE, O\_RDWR)) == -1) // Client->Server 하기위한 Named Pipe 파일 오픈**

**{**

**perror("[SYSTEM] Open Error!!!\n");**

**}**

**else read(read\_fd, &msg, sizeof(msg)); // Client->Server 데이터를 Named Pipe를 통해 읽는다.**

**//printf("\n\ntest------ msg.data= %s, answer = %s\n\n",msg.data, answer[i]);**

**if (!strcmp(msg.data, answer[i]))**

**{**

**sprintf(feedback, "\n[SYSTEM] %ld 유저의 정답입니다!!\n", msg.src);**

**printf("%s", feedback);**

**strncpy(msg.data, feedback,sizeof(feedback)); //(strcpy -> strncpy) 시큐리티 코딩 사용**

**for (j = 0; j < 2; j++)**

**{**

**memset(path, 0, PATH\_SIZE);**

**sprintf(path, "%s%ld", TO\_CLIENT\_FILE, room->clnt[j]);**

**if ((write\_fd = open(path, O\_RDWR)) == -1) // Server->Client 하기위한 Named Pipe 파일 오픈**

**{**

**perror("[SYSTEM] Open Error!!!\n"); // 오류 처리**

**}**

**msg.mtype = room->clnt[j];**

**write(write\_fd, &msg, sizeof(msg)); // Server->Client Named Pipe를 통해 데이터 전송**

**}**

**break;**

**}**

**if (msg.src == room->clnt[0]) // room->clnt[i]는 서버에 들어온 순서대로 Client의 PID를 저장한다.**

**{ // msg.src는 message.source이다 보낸 클라이언트의 PID를 저장한다.**

**msg.mtype = room->clnt[1]; // 보낸 클라이언트가 clnt[0]이면 데이터 전달 대상을 clnt[1]로 정한다.**

**memset(path, 0, PATH\_SIZE); // path 버퍼 초기화**

**sprintf(path, "%s%ld", TO\_CLIENT\_FILE, room->clnt[1]); // client Named Pipe 경로 설정**

**if ((write\_fd = open(path, O\_RDWR)) == -1) // Named Pipe 파일 오픈**

**{**

**perror("[SYSTEM] Open Error!!!\n"); // 오류 처리**

**}**

**}**

**else**

**{**

**msg.mtype = room->clnt[0]; // 보낸 클라이언트가 clnt[1]이면 데이터 전달 대상을 clnt[0]으로 정한다.**

**memset(path, 0, PATH\_SIZE); // path 버퍼 초기화**

**sprintf(path, "%s%ld", TO\_CLIENT\_FILE, room->clnt[0]); // client Named Pipe 경로 설정**

**if ((write\_fd = open(path, O\_RDWR)) == -1) // mkfifo 파일 오픈**

**{**

**perror("[SYSTEM] Open Error!!!\n"); // 오류 처리**

**}**

**}**

**write(write\_fd, &msg, sizeof(msg)); // mkfifo 파일에 데이터 write**

**}**

**}**

**close(read\_fd); // Client->Server Named pipe 닫기**

**close(write\_fd); // Server->Client Named pipe 닫기**

**}**

**void showCharacterImage(Room\* room, int num) // 문제를 출력해주는 함수**

**{**

**char buffer[BUF\_SIZE] = { 0, };**

**char quiz\_path[PATH\_SIZE];**

**char path[PATH\_SIZE];**

**int count = 0;**

**int total = 0;**

**int i;**

**int write\_fd;**

**msg\_t msg;**

**sprintf(quiz\_path, "quiz/%d.txt", num); //quiz 파일 대상설정**

**FILE\* fp = fopen(quiz\_path, "r"); // 파일을 읽기 모드(r)로 열기.**

**// 파일 포인터를 반환**

**memset(msg.data, 0, BUF\_SIZE);**

**while (feof(fp) == 0) // 파일 포인터가 파일의 끝이 아닐 때 계속 반복**

**{**

**count = fread(msg.data, sizeof(char), BUF\_SIZE - 1, fp); // 1바이트씩 1023번 읽기**

**printf("%s", msg.data);**

**for (i = 0; i < 2; i++) //**

**{**

**msg.mtype = room->clnt[i];**

**memset(path, 0, PATH\_SIZE); // 경로 path 버퍼 초기화**

**sprintf(path, "%s%ld", TO\_CLIENT\_FILE, room->clnt[i]); // open할 client mkfifo 경로 설정**

**if ((write\_fd = open(path, O\_RDWR)) == -1) // client에게 데이터 전송을 위해 mkfifo 파일 오픈**

**{**

**perror("[SYSTEM] Open Error!!!\n"); // 오류처리**

**}**

**write(write\_fd, &msg, sizeof(msg)); // client에게 message(Quiz) mkfifo 파일을 통해 전송**

**}**

**memset(msg.data, 0, BUF\_SIZE); // 버퍼를 0으로 초기화**

**total += count; // 읽은 크기 누적**

**}**

**printf("\ntotal: %d\n", total); // total: 파일을 읽은 전체 크기 출력**

**fclose(fp); // 파일 포인터 닫기**

**}**

**void\* writeLogToTextFile() // 로그를 기록하는 쓰레드의 handler 함수**

**{**

**int len;**

**FILE\* fp = NULL;**

**while (1)**

**{**

**char buf[100] = { 0, };**

**len = read(log\_fds[0], buf, sizeof(buf));**

**if ((fp = fopen(LOG\_FILE, "a")) != NULL) // 로그파일 open**

**{**

**fprintf(fp, "%s", buf); // log.txt에 작성**

**fclose(fp); // File pointer close**

**}**

**else**

**{**

**perror("Fail to create log.txt"); // 오류처리**

**}**

**}**

**return NULL;**

**} [ Fifo\_client.c ]**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <unistd.h>**

**#include <string.h>**

**#include <pthread.h>**

**#include <sys/msg.h>**

**#include <fcntl.h>**

**#define TO\_SERVER\_FILE "./fifo/from\_client" // Client가 Server로 데이터를 보내기 위한 Named Pipe 경로**

**#define FROM\_SERVER\_FILE "./fifo/to\_client\_" // Client가 Server에서 데이터를 받기 위한 Named Pipe 경로**

**#define USER\_ID\_SIZE 21**

**#define BUF\_SIZE 1024 // 버퍼 최대 사이즈**

**#define SERVER 1**

**#define CONNECT 32768 +1**

**#define PATH\_SIZE 50 // 경로명의 최대 사이즈**

**typedef struct msgtype { // 서버와 클라이언트가 통신하기 위해 사용하는 데이터 구조체**

**long mtype; // 메시지 형태이다. 전달 대상을 설정한다.**

**char data[BUF\_SIZE]; // 클라이언트가 입력한 메시지**

**long src; // 서버에게 데이터를 보낸 클라이언트의 PID 정보**

**}msg\_t;**

**long id=0; // 실행하고 있는 해당 프로세스(client)의 PID를 저장할 변수**

**int read\_fd=0; // Server로부터 메세지를 읽어올 Named Pipe의 file descriptor**

**int write\_fd=0; // Server에게 데이터를 전달할 Named Pipe의 file descriptor**

**char read\_path[PATH\_SIZE]; // 서버와 통신하기위해 사용하는 mkfifo의 경로(client의 PID에 따라 변경)**

**void sendConnectionMessage (void);**

**void startChattingThread(void);**

**void\* sendMessageToChattingServer(void\* data);**

**void\* receiveMessageFromChattingServer(void\* data);**

**int main()**

**{**

**id = (long)getpid(); // getpid()**

**if ((write\_fd = open(TO\_SERVER\_FILE, O\_RDWR)) == -1) // Server에 데이터를 전송할 때 사용할 Named Pipe file 열기**

**{**

**perror("[SYSTEM] mkfifo 파일 열기 실패(send to server)\n"); // 오류처리**

**printf("[SYSTEM] 프로그램이 3초 후 종료됩니다.\n");**

**sleep(3);**

**exit(1);**

**}**

**sprintf(read\_path, "%s%ld", FROM\_SERVER\_FILE, id); // Server에서 데이터를 전송받을 때 사용할 mkfifo 경로 설정(./fifo/to\_clinet\_[pid])**

**if ((read\_fd = open(read\_path, O\_RDWR)) == -1) // mkfifo file(./fifo/to\_clinet\_[pid]) 오픈**

**{**

**perror("[SYSTEM] mkfifo 파일 열기 실패(receive from server)\n"); // 오류처리**

**printf("[SYSTEM] mkfifo 파일 생성 중(receive from server)\n");**

**mkfifo(read\_path, 0666); // Server에서 데이터를 받을 Named Pipe이 없을 경우 mkfifo()를 이용해 생성한다.**

**printf("[SYSTEM] mkfifo 파일 생성 완료(receive from server)\n");**

**if ((read\_fd = open(read\_path, O\_RDWR)) == -1)**

**{**

**perror("[SYSTEM] mkfifo 열기 실패(receive from server)\n"); //2차 오류처리**

**printf("[SYSTEM] 프로그램이 3초 후 종료됩니다.\n");**

**sleep(3);**

**exit(1);**

**}**

**}**

**sendConnectionMessage (); // Server에 접속했다고 알리기 위한 데이터를 전송하는 함수**

**startChattingThread(); // 채팅에 관련된 쓰레드를 생성하고 관리하는 함수**

**return 0;**

**}**

**void\* sendMessageToChattingServer(void\* data) // Server에 데이터를 보낼 때 사용할 쓰레드의 handler function**

**{**

**msg\_t msg;**

**char send\_buf[BUF\_SIZE];**

**memset(msg.data, '\0', BUF\_SIZE);**

**msg.mtype = SERVER;**

**msg.src = id;**

**while (1)**

**{**

**memset(send\_buf, '\0', sizeof(send\_buf)); //send\_buf 변수 널문자로 초기화**

**fgets(send\_buf, sizeof(send\_buf), stdin); // (gets -> fgets) 시큐어 코딩 적용**

**send\_buf[strlen(send\_buf)] = '\0';**

**strncpy(msg.data, send\_buf,sizeof(send\_buf)); // (strcpy -> strncpy) 시큐어 코딩 적용**

**write(write\_fd, &msg, sizeof(msg)); // Server에 전송할 mkfifo 파일에 데이터를 전송**

**}**

**return NULL;**

**}**

**void\* receiveMessageFromChattingServer(void\* data) // 서버로부터 데이터를 받을 때 사용할 쓰레드의 handler function**

**{**

**msg\_t msg;**

**while (1) // while loop로 계속해서 받는다.**

**{**

**if (read(read\_fd, &msg, sizeof(msg)) > 0)**

**{**

**printf("%s", msg.data);**

**}**

**}**

**return NULL;**

**}**

**void sendConnectionMessage (void) //처음 Client가 Server에 연결될 때 입장을 알리기 위한 데이터 전송 함수**

**{**

**msg\_t msg;**

**msg.mtype = CONNECT;**

**msg.src = id;**

**write(write\_fd, &msg, sizeof(msg));**

**}**

**void startChattingThread(void) // Client의 기본적인 채팅 함수**

**{**

**pthread\_t send\_thread, recv\_thread; // send, receive 쓰레드 선언**

**void\* send\_thread\_return, \*recv\_thread\_return;**

**printf("채팅시작\n");**

**pthread\_create(&send\_thread, NULL, sendMessageToChattingServer, NULL); // Server에 데이터를 보내는 쓰레드 생성**

**pthread\_create(&recv\_thread, NULL, receiveMessageFromChattingServer, NULL); // Server로부터 데이터를 받는 쓰레드 생성**

**pthread\_join(send\_thread, &send\_thread\_return);**

**pthread\_join(recv\_thread, &recv\_thread\_return);**

**close(read\_fd);**

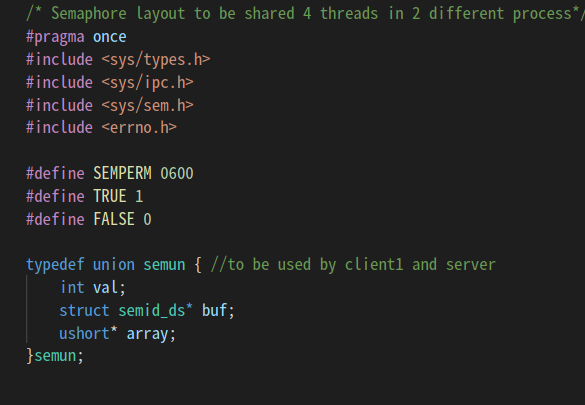
**close(write\_fd);**

**printf("채팅종료\n");**

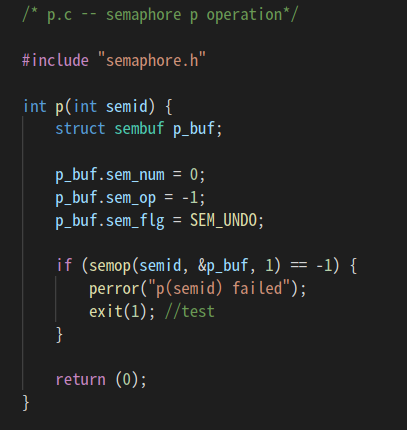
**}**

### **Shared Memory**

**semaphore.h**

****

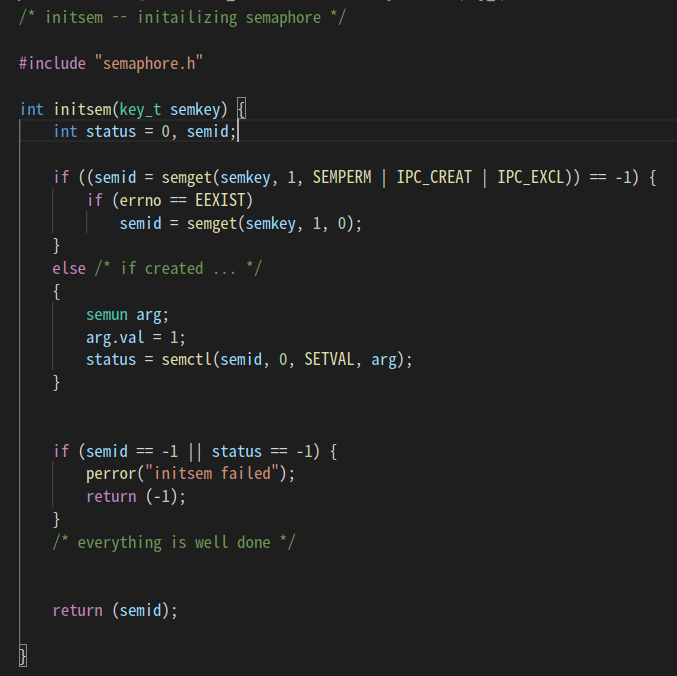
**p.c**

****

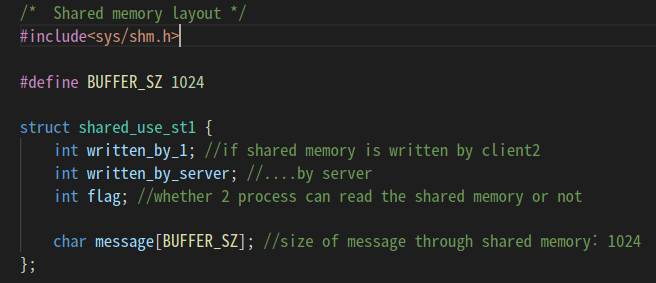
**v.c**

****

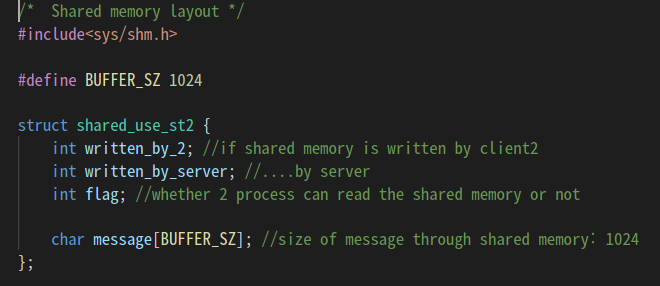
**initsem.c**

****

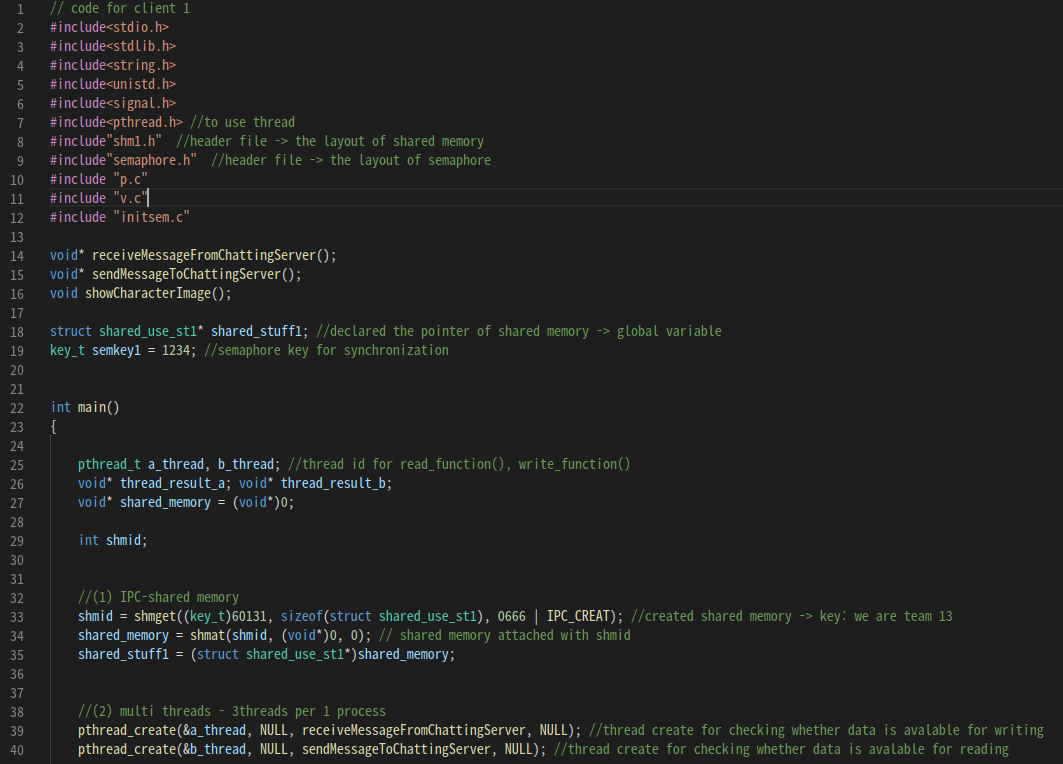
**shm1.h**

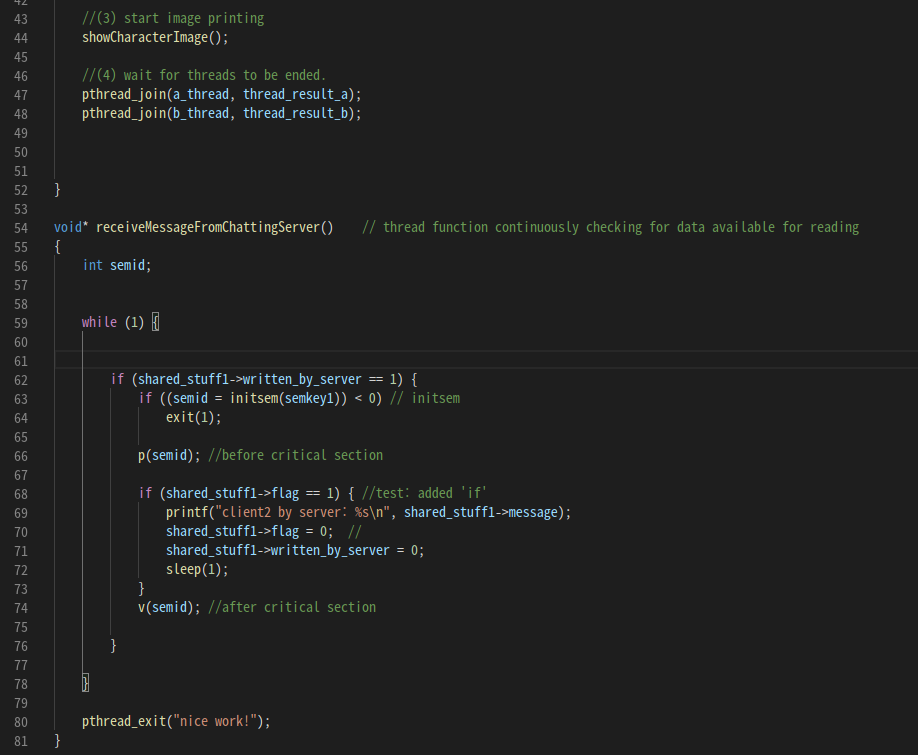
****

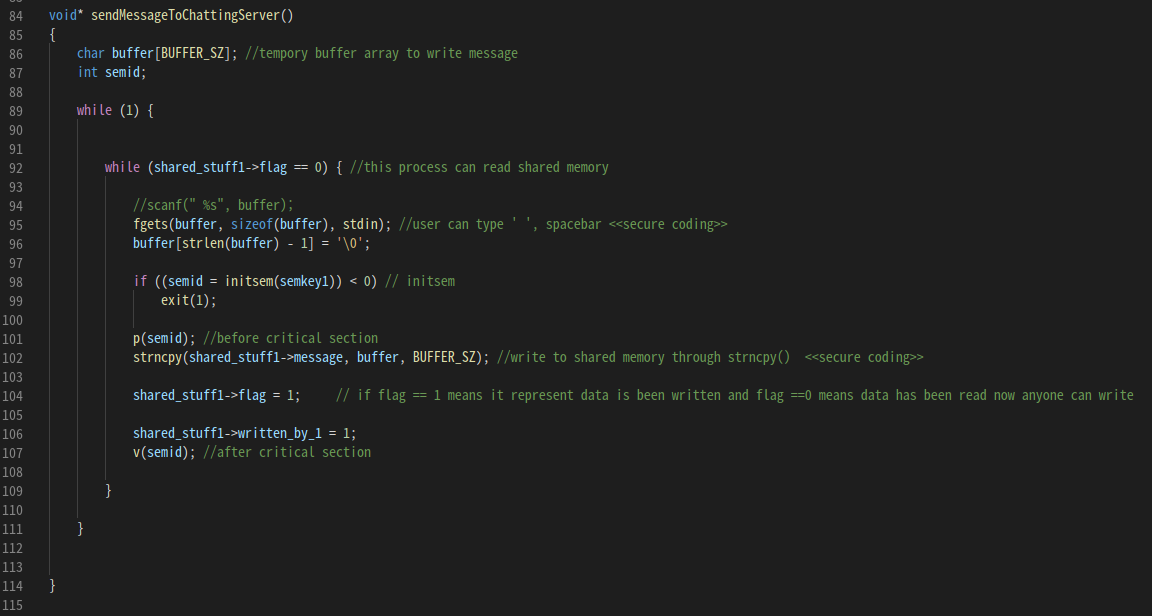
**shm2.h**

****

**client1.c**

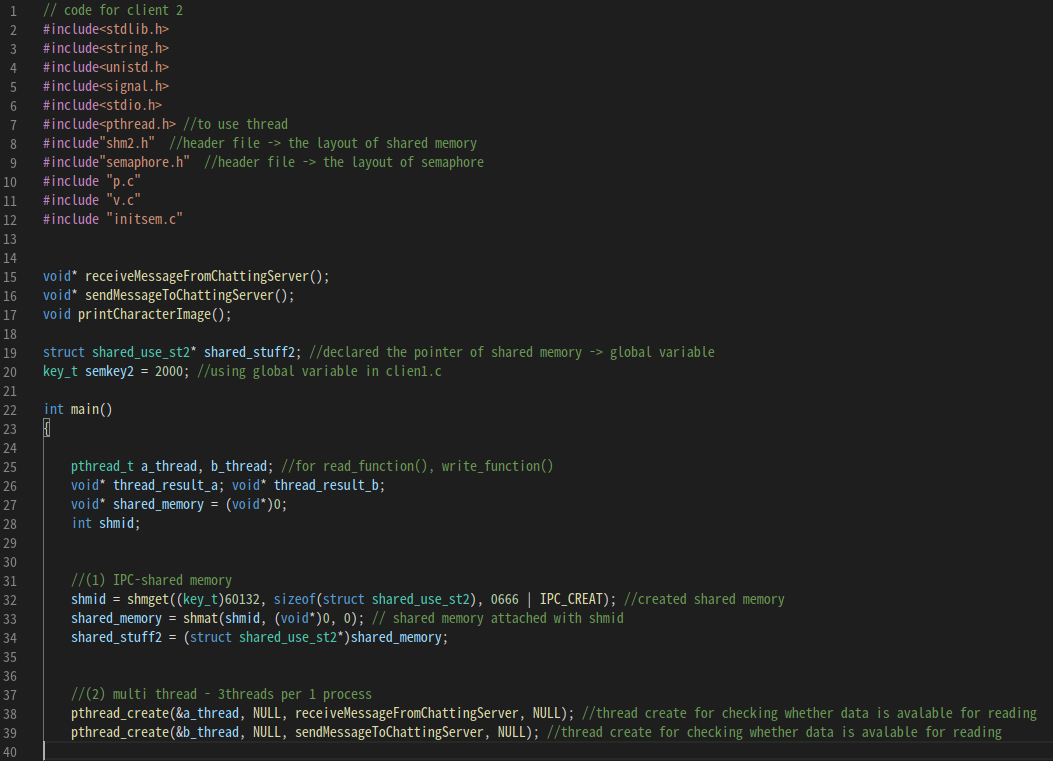
****

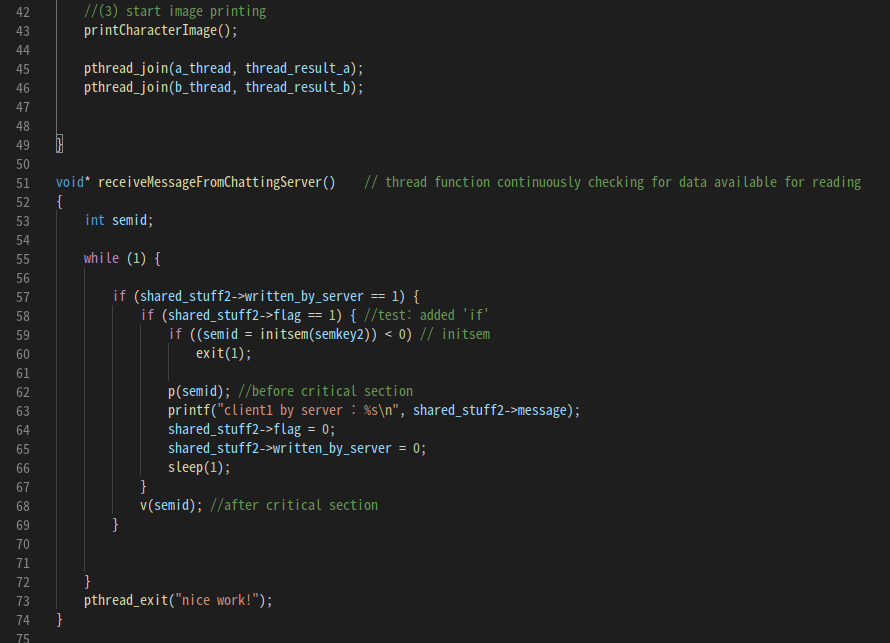
****

****

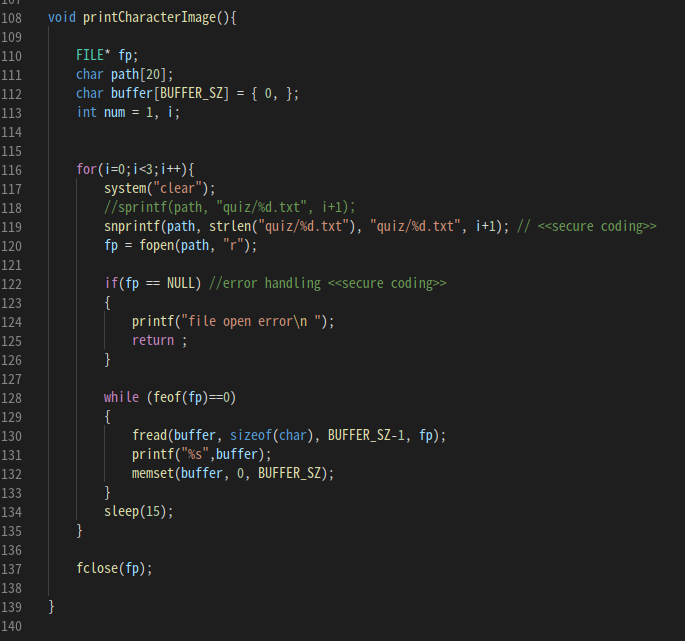
****

**cient2.c**

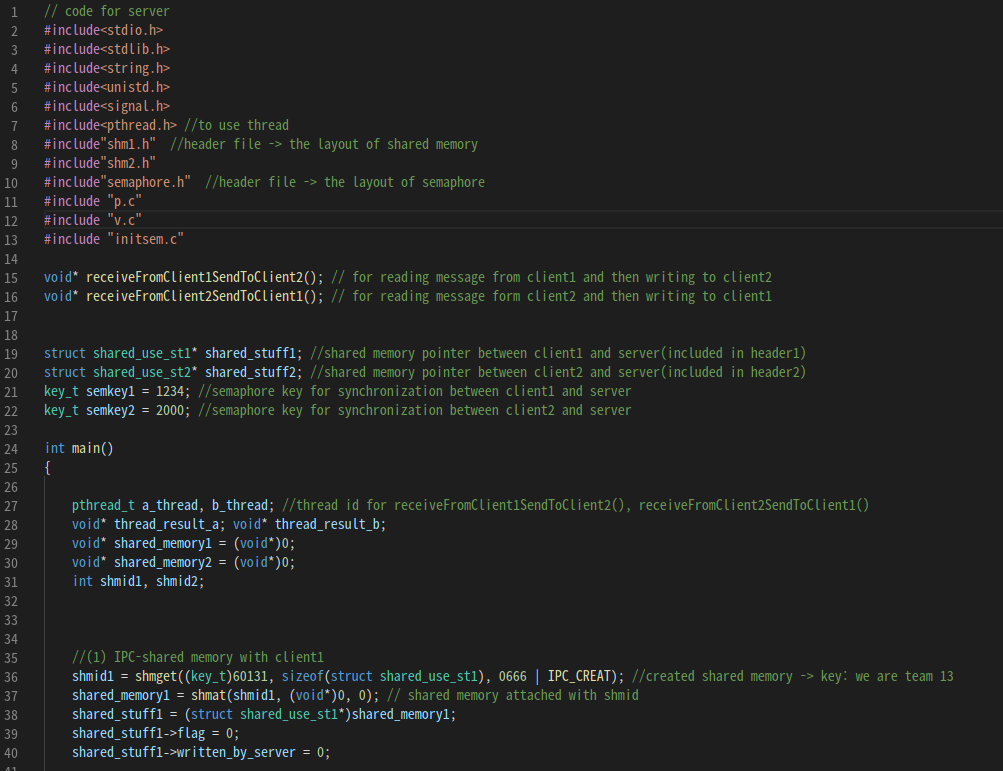
****

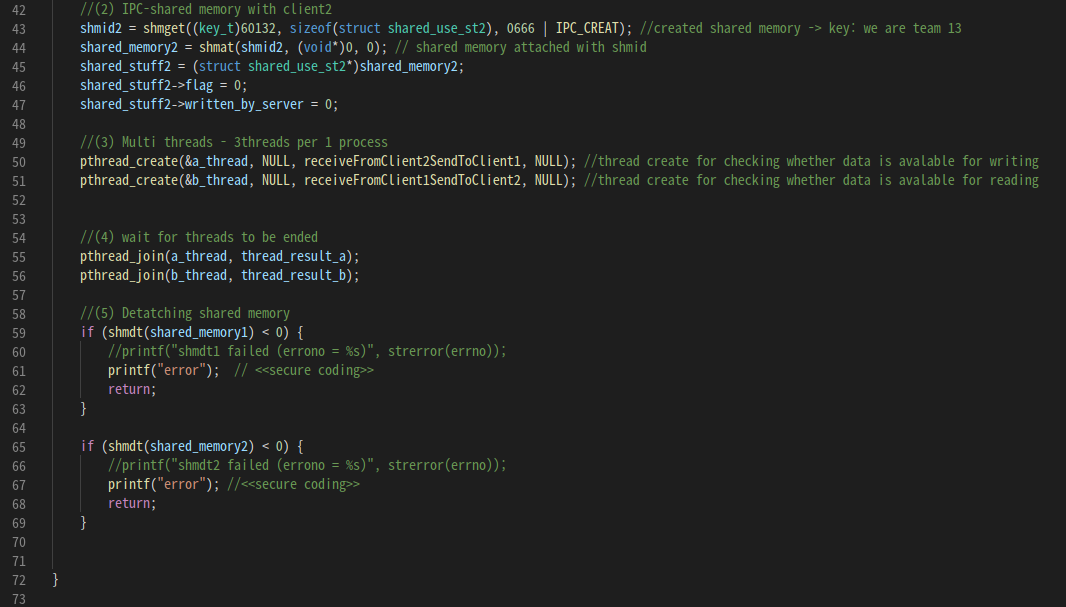
****

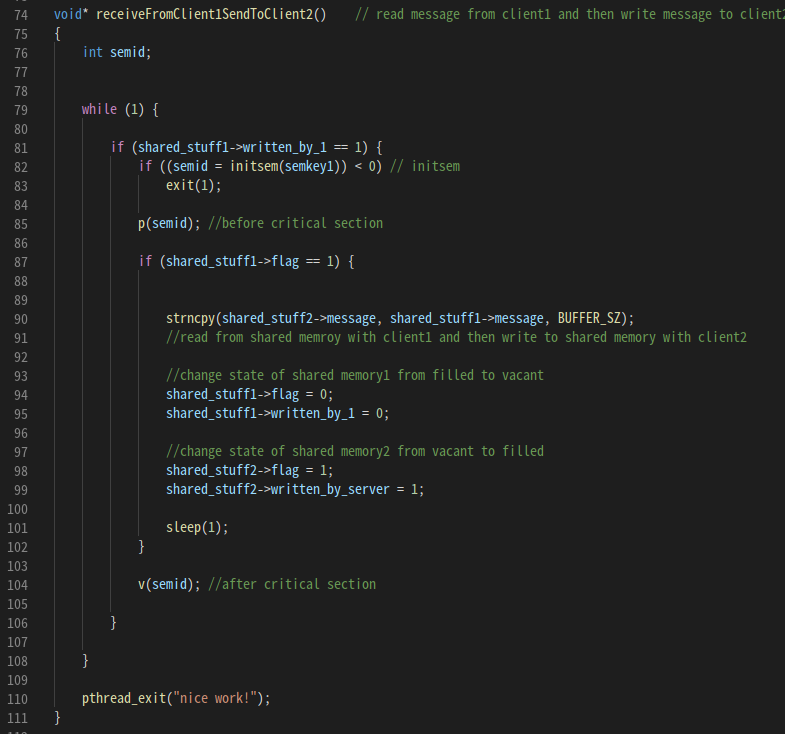
****

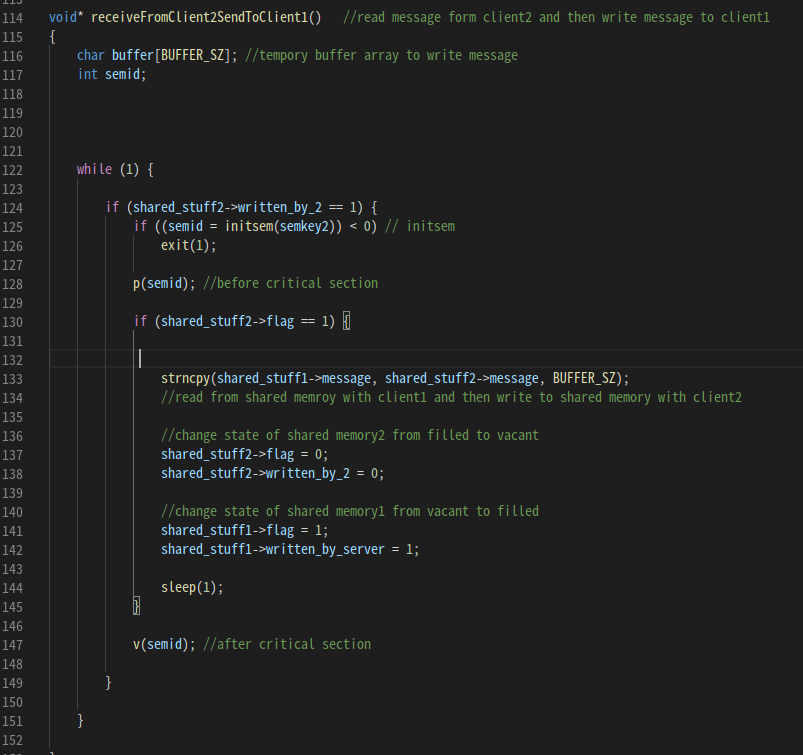
****

**server.c**









### **Message Passing**

**[ MsgQ\_server.c]**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <unistd.h>**

**#include <string.h>**

**#include <pthread.h>**

**#include <sys/msg.h>**

**#include <fcntl.h>**

**#define BUFFER\_SIZE 1024 // 총 버퍼의 크기**

**#define SERVER 1 // 서버의 기본 번호**

**#define CONNECT 32768 +1 // 최대 pid수 이상인 CONNECT 번호**

**#define TOTAL\_QUIZ 3 // 문제의 갯수**

**typedef struct MessageType { long mtype; char data[BUFFER\_SIZE]; long source; } Message\_t;**

**// 메세지 큐에 저장할 값**

**typedef struct Room {**

**long clients[2];**

**} Room; // 클라이언트들을 저장할 구조체**

**void receiveConnectionMessage(Room\* room); // 서버가 켜지고 난 후 클라이언트의 연결을 받는 메서드**

**void startUserChatting(Room\* room); // 실질적으로 게임이 수행되는 메서드**

**void showCharacterImage(Room\* room, int num); // 사용자가 맞출 인물 퀴즈를 출력해줄 메서드**

**void\* writeLogToTextFile(); // 로그 저장 메서드**

**key\_t key\_id; // 메시지큐의 키를 저장할 변수**

**int log\_fds[2]; // 로그의 파일 디스크립터(익명파이프 사용)**

**char log\_message[100];**

**int main() {**

**pthread\_t chat\_thread; // 채팅 쓰레드의 아이디를 받을 변수**

**Room room; // 클라이언트들의 pid를 저장할 room 변수**

**pthread\_t t\_id; // 쓰레드의 반환을 맡을 t\_id 함수**

**void\* chat\_thread\_return; // 쓰레드의 반환값을 저장할 변수**

**//IPC 기법 초기화**

**key\_id = msgget((key\_t)60139, IPC\_CREAT | 0666); // 메시지큐를 초기화시켜줌**

**if (key\_id == -1)**

**{**

**printf("error\n");**

**exit(0);**

**}**

**// 로그 부분**

**pipe(log\_fds); // 로그기록 파이프 생성**

**pthread\_create(&t\_id, NULL, writeLogToTextFile, NULL); // 로그 쓰레드를 동작시킴**

**receiveConnectionMessage(&room);**

**pthread\_create(&chat\_thread, source == room->clients[0]) {**

**message.mtype = room->clients[1];**

**}**

**else {**

**message.mtype = room->clients[0];**

**}**

**msgsnd(key\_id, &message, sizeof(message) - sizeof(long), 0);**

**}**

**}**

**strncpy(message.data, "\_end\_", strlen("\_end\_"));**

**for (j = 0; j < 2; j++) {**

**message.mtype = room->clients[j];**

**msgsnd(key\_id, &message, sizeof(message) - sizeof(long), 0);**

**}**

**sleep(3);**

**msgctl(key\_id,IPC\_RMID,0);**

**}**

**void showCharacterImage(Room\* room, int num) {**

**int count = 0;**

**int total = 0;**

**Message\_t message;**

**int i;**

**char imagepath[20];**

**snprintf(imagepath, strlen("quiz/%d.txt"),"quiz/%d.txt", num);**

**FILE\* fp = fopen(imagepath, "r"); // 파일을 읽기 모드(r)로 열기.**

**if (fp == NULL)**

**{**

**printf("error\n");**

**return;**

**}**

**memset(message.data, 0, BUFFER\_SIZE);**

**while (feof(fp) == 0) // 파일 포인터가 파일의 끝이 아닐 때 계속 반복**

**{**

**count = fread(message.data, sizeof(char), BUFFER\_SIZE - 1, fp);**

**// 1바이트씩 1023번 읽기**

**printf("%s", message.data);**

**for (i = 0; i < 2; i++) {**

**message.mtype = room->clients[i];**

**msgsnd(key\_id, &message, sizeof(message) - sizeof(long), 0);**

**}**

**memset(message.data, 0, BUFFER\_SIZE); // 버퍼를 0으로 초기화**

**total += count; // 읽은 크기 누적**

**}**

**printf("\ntotal: %d\n", total); // total: 파일을 읽은 전체 크기 출력**

**fclose(fp); // 파일 포인터 닫기**

**}**

**void\* writeLogToTextFile() { // 로그 작성 파이프함수**

**int length;**

**FILE\* fp = NULL;**

**pthread\_mutex\_t log\_mutex;**

**while (1) {**

**char buffer[100] = { 0, };**

**length = read(log\_fds[0], buffer, sizeof(buffer));**

**if(length == -1) {**

**printf("로그 오류");**

**}**

**pthread\_mutex\_lock(&log\_mutex);**

**if ((fp = fopen("log.txt", "a")) != NULL) {**

**fprintf(fp, "%s", buffer);**

**fclose(fp);**

**}**

**else {**

**printf("로그 오류\n");**

**}**

**pthread\_mutex\_unlock(&log\_mutex);**

**}**

**return NULL;**

**}**

**[ MsgQ\_client.c ]**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <unistd.h>**

**#include <string.h>**

**#include <pthread.h>**

**#include <sys/msg.h>**

**#define BUFFER\_SIZE 1024 // 총 버퍼의 크기**

**#define SERVER 1 // 서버의 기본 번호**

**#define CONNECT 32768 +1 // 최대 pid수 이상인 CONNECT 번호**

**typedef struct MessageType { long mtype; char data[BUFFER\_SIZE]; long source; } Message\_t; // 메세지 큐에 저장할 값**

**typedef struct MultipleArg {**

**int fd;**

**int id;**

**} MultipleArg;**

**void sendConnectionMessage(MultipleArg\* mArg); // 클라이언트가 켜지고 서버에게 연결메시지 전송**

**void startChattingThread(MultipleArg\* mArg); // 채팅을 위한 쓰레드들을 시작해주는 함수**

**void\* sendMessageToChattingServer(void\* multiple\_arg); // 채팅서버로 메시지큐에 입력하는 함수**

**void\* receiveMessageFromChattingServer(void\* arg); // 메시지큐에서 자신으로 입력을 받는 함수**

**int quit = 0; // 쓰레드들의 종료를 위한 flag 값**

**int main() {**

**MultipleArg\* mArg;**

**mArg = (MultipleArg\*)malloc(sizeof(MultipleArg)); // init**

**mArg->fd = msgget((key\_t)60139, IPC\_CREAT | 0666);**

**if (mArg->fd == -1)**

**{**

**printf("error\n");**

**exit(0);**

**}**

**mArg->id = (long)getpid();**

**sendConnectionMessage(mArg);**

**startChattingThread(mArg);**

**return 0;**

**}**

**void sendConnectionMessage(MultipleArg\* mArg) {**

**Message\_t message;**

**message.mtype = CONNECT;**

**message.source = mArg->id;**

**msgsnd(mArg->fd, &message, sizeof(message) - sizeof(long), 0); // 연결 메시지 전송**

**}**

**void\* sendMessageToChattingServer(void\* multiple\_argment)**

**{**

**MultipleArg multipleArgment = \*((MultipleArg\*)multiple\_argment);**

**Message\_t message;**

**char send\_buffer[BUFFER\_SIZE];**

**memset(message.data, '\0', BUFFER\_SIZE);**

**message.mtype = SERVER;**

**message.source = multipleArgment.id;**

**//sprintf(msg.data, "%d님이 입장하였습니다.\n", mArg.id);**

**//msgsnd(mArg.fd, &msg, sizeof(msg) - sizeof(long), 0);**

**while (quit==0) {**

**memset(send\_buffer, '\0', sizeof(send\_buffer));**

**fgets(send\_buffer, sizeof(send\_buffer), stdin);**

**send\_buffer[strlen(send\_buffer)] = '\0';**

**strncpy(message.data, send\_buffer, strlen(send\_buffer));**

**msgsnd(multipleArgment.fd, &message, sizeof(message) - sizeof(long), 0);**

**}**

**return NULL;**

**}**

**void\* receiveMessageFromChattingServer(void\* arg)**

**{**

**MultipleArg multipleArgment = \*((MultipleArg\*)arg);**

**Message\_t message;**

**while (quit==0) {**

**if ((msgrcv(multipleArgment.fd, &message, sizeof(message) - sizeof(long), multipleArgment.id, 0)) > 0) {**

**if (!strncmp(message.data, "clear",strlen("clear"))) {**

**system("clear");**

**}**

**else if(!strncmp(message.data, "\_end\_",strlen("\_end\_"))){**

**quit=1;**

**}**

**else {**

**printf("%s", message.data);**

**}**

**}**

**}**

**return NULL;**

**}**

**void startChattingThread(MultipleArg\* mArg) {**

**pthread\_t send\_thread, receive\_thread;**

**void\* send\_thread\_return, \* recv\_thread\_return;**

**printf("채팅시작\n");**

**pthread\_create(&send\_thread, NULL, sendMessageToChattingServer, (void\*)mArg);**

**pthread\_create(&receive\_thread, NULL, receiveMessageFromChattingServer, (void\*)mArg);**

**pthread\_join(send\_thread, &send\_thread\_return);**

**pthread\_join(receive\_thread, &recv\_thread\_return);**

**printf("채팅종료\n");**

**}**