**시스템 소프트웨어와 실습**

**실습과제 #3**

**보고서**

**이름: 정여준**

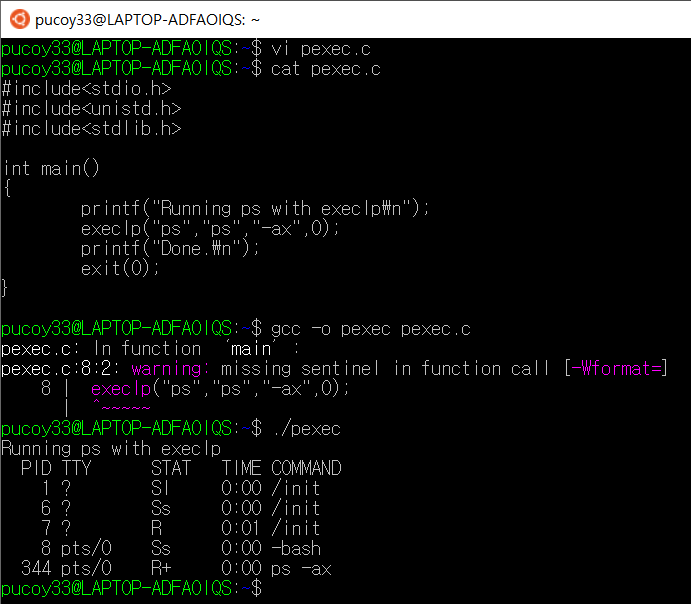
**학번: 2017112138  
학과: 컴퓨터공학과  
교수님: 문봉교 교수님**

**강의명: 시스템소프트웨어와 실습**

**8장 실습**

**1. 첨부된 Process – fork & exec, wait, signal 파일의 내용에 따라 실습을 진행하고 보고서에 포함한다.**

**A. Try It Out – execlp <실습>**

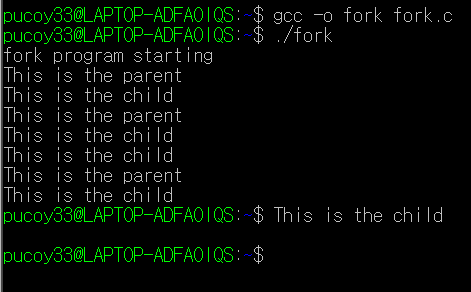


pexec.c파일을 생성하고 이 파일로부터 실행가능한 파일을 만들기 위해서 gcc -o pexec pexec.c를 실행한다. 그 이후 ./pexec 명령어를 통해 실행하면 위와 같은 실행화면이 나온다.

**B. Try It Out – fork <실습>**

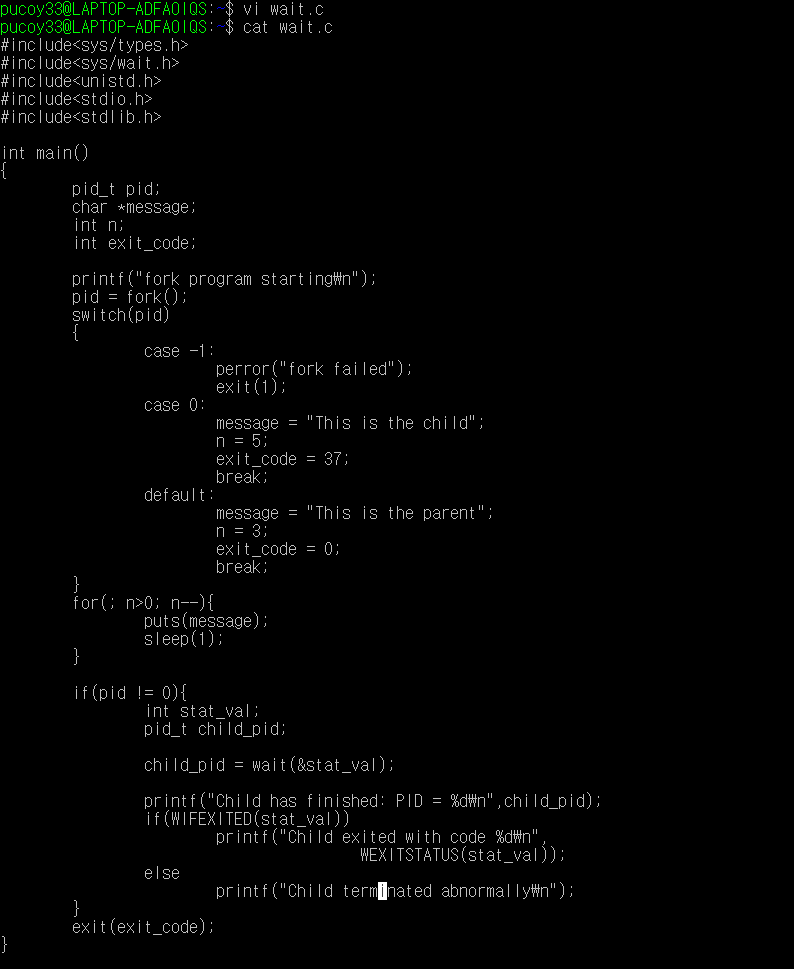


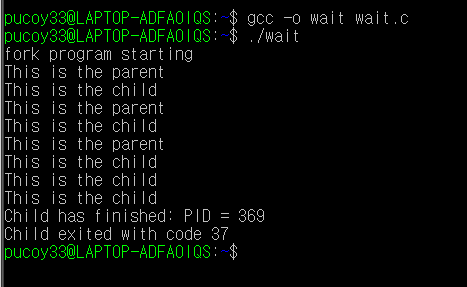
일단 fork.c파일을 생성한다.



gcc -o fork fork.c라는 명령어를 통해서 fork.c파일을 사용해서 실행가능한 파일 fork를 생성하고 ./fork명령어를 통해 파일 fork를 실행한다. 그러면 위와 같은 실행화면이 나오는 것을 볼 수 있다.

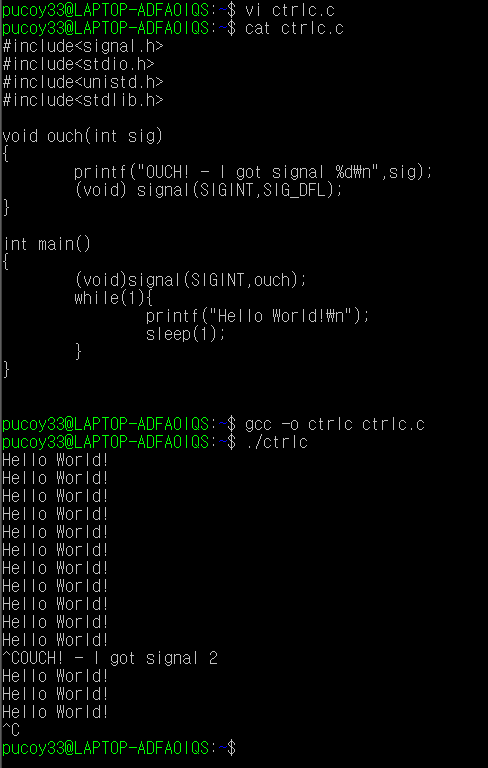
**A. Try It Out – wait <실습>**

  
일단 wait.c 파일을 생성한다.



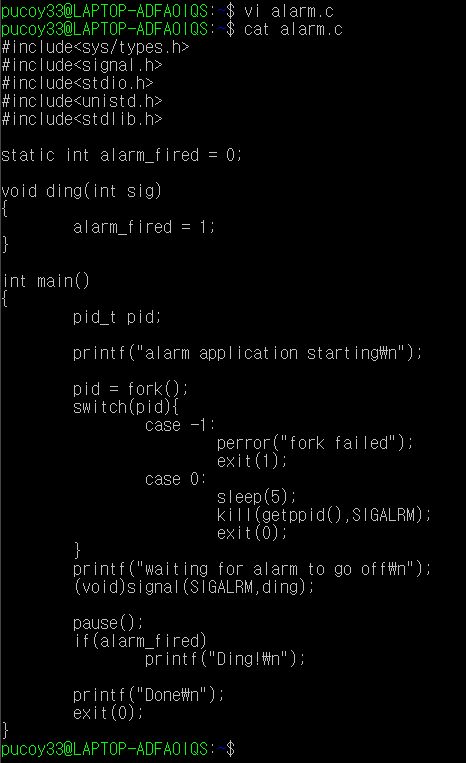
gcc -o wait wait.c 명령어를 통해서 wait.c 파일을 이용해서 실행가능한 wait이라는 파일을 생성한다. 그 이후에 ./wait 명령어를 통해서 파일 wait을 실행한다. 그러면 위와 같은 실행 화면을 볼 수 있다.

**Try It Out – Signal Handling <실습>**

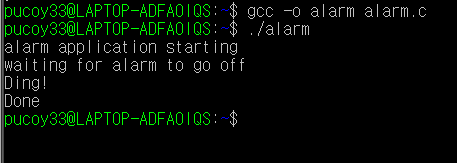


일단 파일 ctrlc.c파일을 생성한다. 이것은 사용자가 Ctr-C를 입력할 때의 신호를 받아서 정해놓은 값을 출력하고 두번 Ctr-C를 입력하면 프로그램이 종료된다. gcc -o ctrlc ctrlc.c 명령어를 통해 ctrlc.c파일을 이용해 실행가능한 파일 ctrcl파일을 생성한다. 그 이후 ./ctrlc 명령어를 통해 실행하면 위와 같은 실행화면이 나오는 것을 볼 수 있다.

**Try It Out – An Alarm Clock <실습>**



일단 alarm.c파일을 생성한다.



gcc -o alarm alarm.c 명령어를 통해서 alarm.c파일을 이용해서 실행가능한 파일 alarm을 생성한다. 그리고 ./alarm 명령어를 통해서 파일 alarm을 실행한다. Alarm.c파일에서 5초간 멈췄다가 Ding!이 출력되도록 프로그램을 짜놨기 때문에 위의 실행화면에서 waiting for alarm to go off가 출력되고 5초 동안 멈춰있다가 Ding!이 출력되는 것을 확인할 수 있다.

**2. 주어진 Practice and Find solutions의 1, 2, 3, 4번의 문제를 직접 실행해보고 그 결과를 분석하고 보고서를 작성한다**

**1.** **How many times will the number 6 be printed in the following program? Explain your answer.**

int main() {

for (int i=0; i<10; i++)

{

Fork();

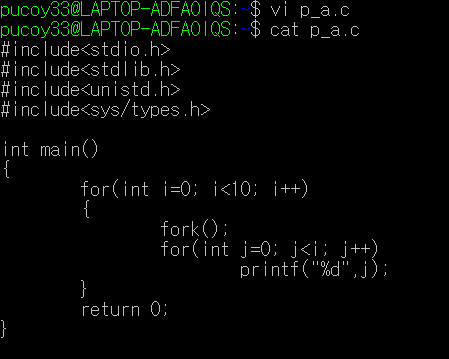
For(int j=0; j<I; j++)

Printf(“%d”,j);

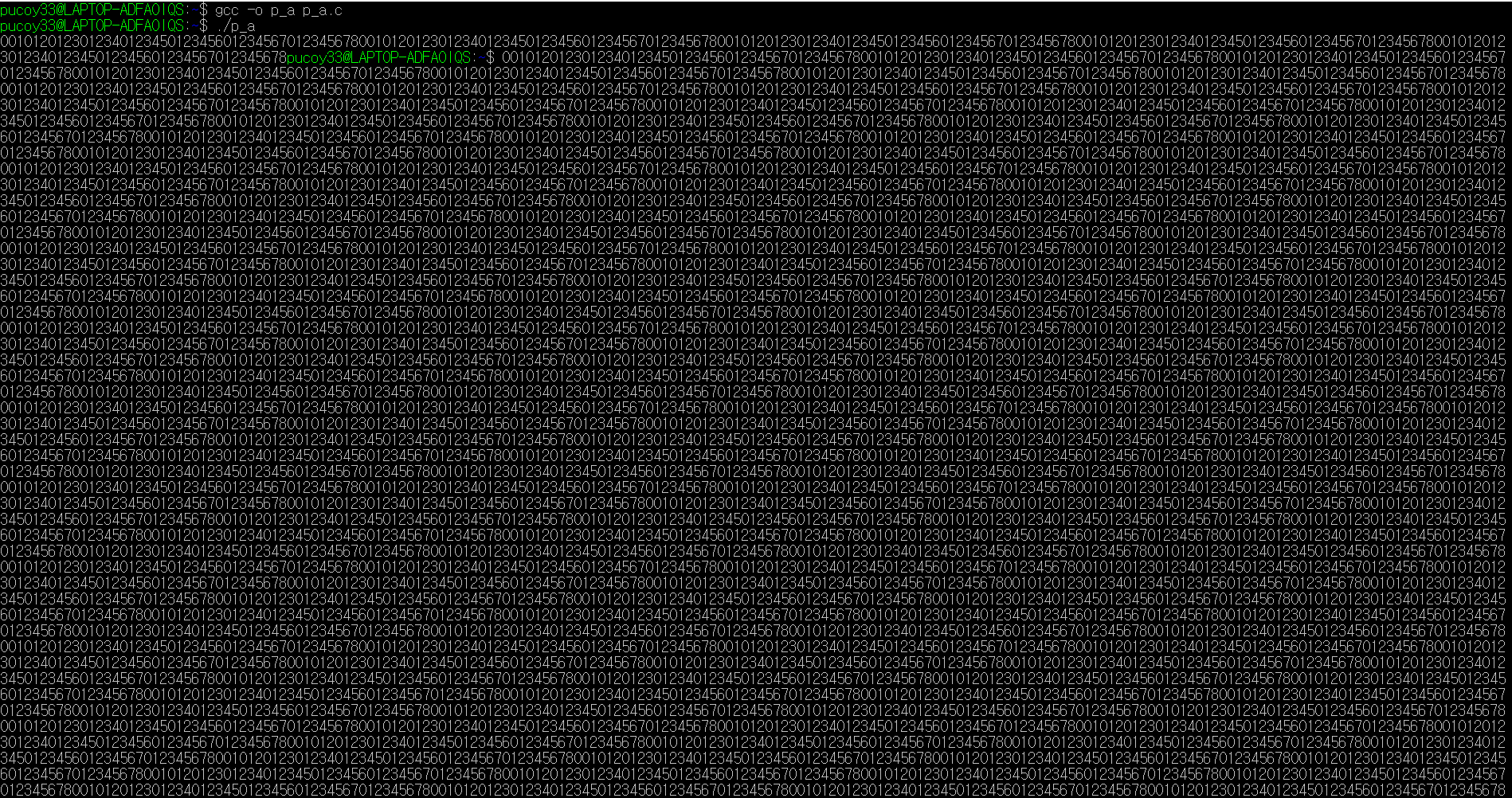
}

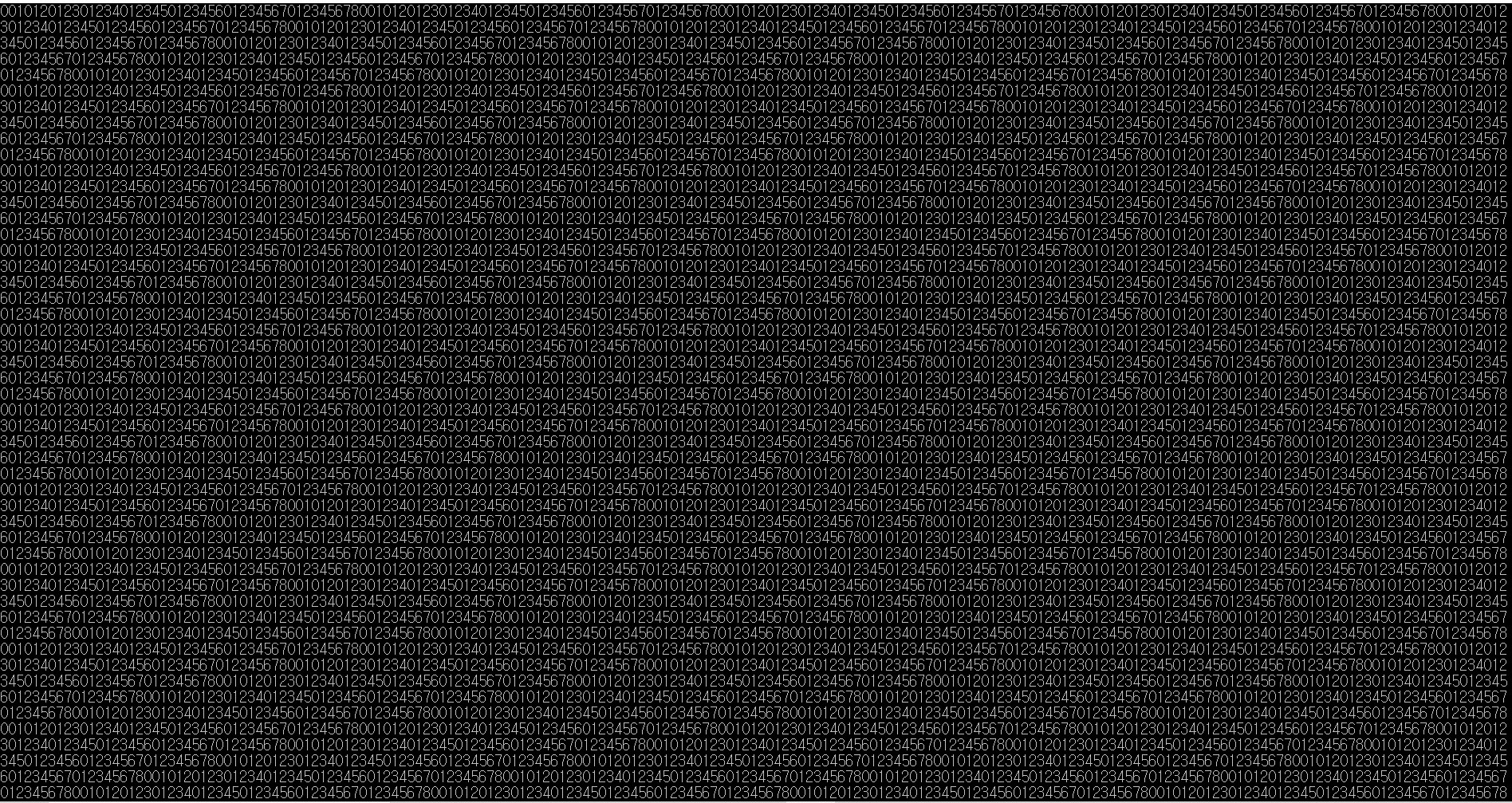
Return 0;

}



파일 p\_a.c를 생성한다. 이는 0~9까지 fork()를 수행하고 fork()를 통해 생겨난 자식 프로세스틀도 반복하면서 fork()를 수행하는 코드이다.





gcc -o p\_a p\_a.c 명령어를 사용하면 p\_a.c파일을 이용해서 실행가능한 파일 p\_a를 생성할 수 있다. 생성된 p\_a파일을 실행하기 위해서 ./p\_a명령어를 수행하면 위와 같은 실행화면을 얻을 수 있다.

0/01/012/0123/01234/012345/0123456/01234567/012345678를 계속 반복한다.

여기서 자식 프로세스는 2^10개 만큼 생성되고 6은 3번이 나오므로 6은 총 3\*2^10번 나오는 것을 알 수 있다. 즉, 3072번 나온다.

**2.** **What are all the possible outputs of the following program? Explain your answer.**

int g1=0, g2=0;

int main()

{

int value;

if (fork() != 0)

{

g2 = g1 + 1;

wait( &value );

}

else

{

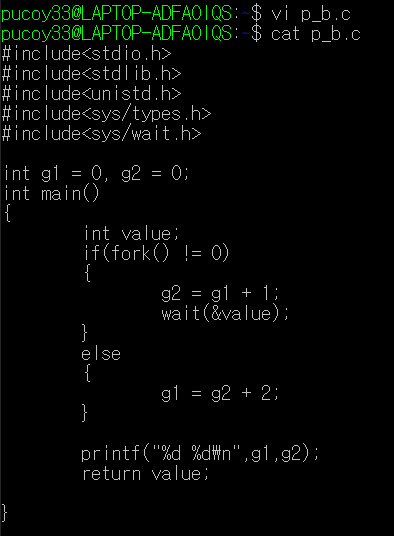
g1 = g2 + 2;

}

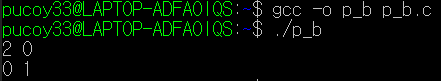
printf("%d %d\n", g1, g2);

return value;

}



일단 p\_b.c파일을 생성한다.



gcc -o p\_b p\_b.c 명령어를 통해서 p\_b.c파일을 사용해서 실행가능한 파일 p\_b를 생성한다. 그 이후에 생성한 파일을 실행하기 위해 ./p\_b 명령어를 사용하면 위와 같은 실행화면이 나오는 것을 확인할 수 있다. 이는 fork()를 통해 생성된 자식 프로세스는 부모 프로세스이 모든 것을 복사해 독립적으로 유지하면서 프로세스를 실행하기 때문에 부모 프로세스의 경우 (fork() != 0) wait을 통해서 자식 프로세스가 종료되기를 기다리므로 자식 프로세스의 printf문이 먼저 실행되는 것이다.

**3.** **In this question we assume that the only process in the system is main, and its process id is 1. In addition, each newly created process gets a process id equal to “the biggest existing process id + 1”. In the following program, write the contents of the array pid, for each process, at the moment this process reaches the line marked by “//end”. Explain your answer.**

int main()

{

int status;

int pid[3];

for (int i=0; i<3; i++)

{

pid[i] = fork();

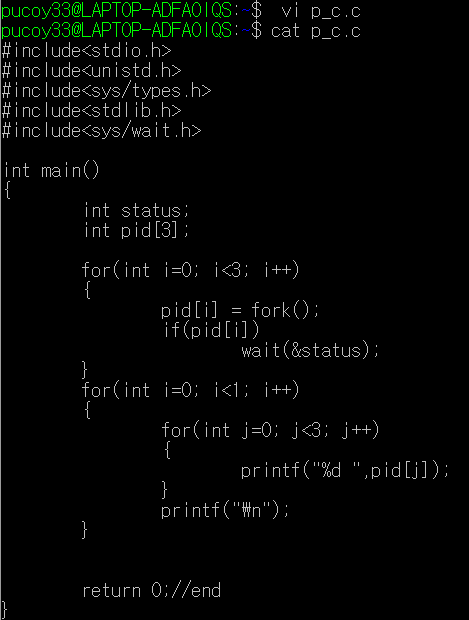
if( pid[i] )

wait(&status);

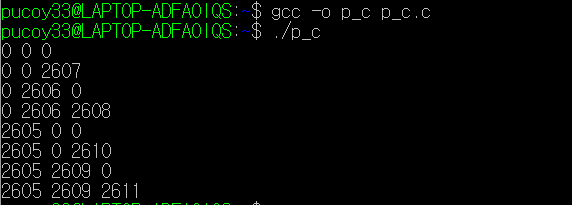
}

return 0; //end

}



일단 p\_c.c파일을 생성한다.



gcc -o p\_c p\_c.c 명령어를 통해서 p\_c.c파일을 사용해 실행가능한 파일 p\_c를 생성한다. 그 이후에 p\_c를 실행하기 위해서 ./p\_c라는 명령어를 실행한다. 그러면 위와 같은 실행결과를 볼 수 있다. 여기서 출력된 값들은 각 프로세스의 프로세스 아이디를 출력한 것이다. 따라서 이러한 각 프로세스 별로 프로세스 아이디를 표현하면 다음과 같다.

프로세스 1) 0 0 0

프로세스 2) 0 0 3

프로세스 3) 0 2 0

프로세스 4) 0 2 4

프로세스 5) 1 0 0

프로세스 6) 1 0 6

프로세스 7) 1 5 0

프로세스 8) 1 5 7

**4. The following program (test.c) was compiled and run as shown below. Assuming the current directory is “/usr/local”, How many times should the sign “-“ be printed when we run the program? Explain your answer.**

#include<stdio.h>

#include<stdlib.h>

int main(char\* argv[], int argc)

{

int num = atoi(argv[1]);

char \*argv2[] = {“test” , “0”, NULL };

int i;

for (i=0; i<num; i++)

{

if (fork() != 0)

{

execv(“/usr/local/test”, argv2);

fork();

}

fork();

execv(“/usr/local/test”, argv2);

}

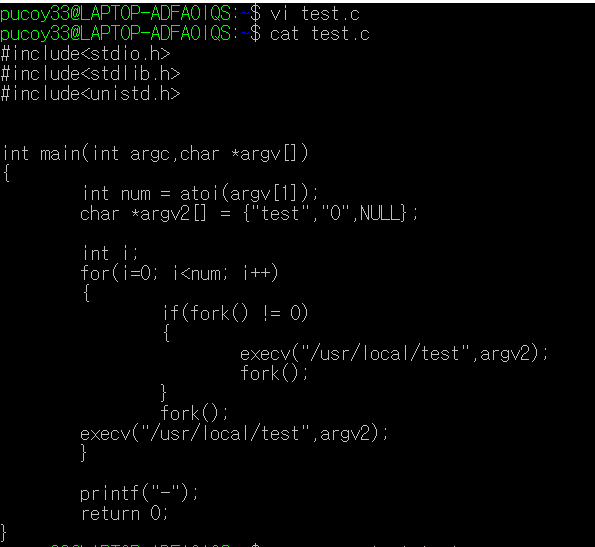
printf(“-“);

return 0;

}

The shell commands executed are:

* gcc –g –o test test.c
* ./test 3



Test.c파일을 생성한다.



Gcc -g -o test test.c 명령어를 통해서 test.c파일을 사용해 실행가능한 파일 test를 생성한다.



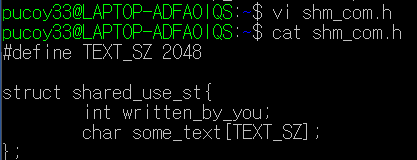
./test 3 명령어를 통해 num에 argv[1]의 값을 저장한다. 이때 argv는 [0] = ./test;이고 [1] = 3이 들어가므로 num에 3이 대입된다. Execv(“usr/local/test”,argv2)는 경로와 인자를 뜻한다. 즉, -는 216회 출력된다.

**3. 해당실습을 진행하고 보고서를 작성하는 동안 느꼈던 개인적인 생각과 느낌을 에세이 형식으로 정리하여 추가한다.**

프로세스에 관련해서 fork()와 exec계열의 함수들 그리고 wait이랑 signal계열의 함수들이 어떤 식으로 작동하는지에 대해서 알 수 있었고 코드를 짜고 실행가능한 파일을 만들어서 실행했을 때 어떤 식으로 값들이 출력되는지에 대해 확인하면서 코드의 의미가 어떤 것인지에 대해서 공부할 수 있어서 좋았습니다.

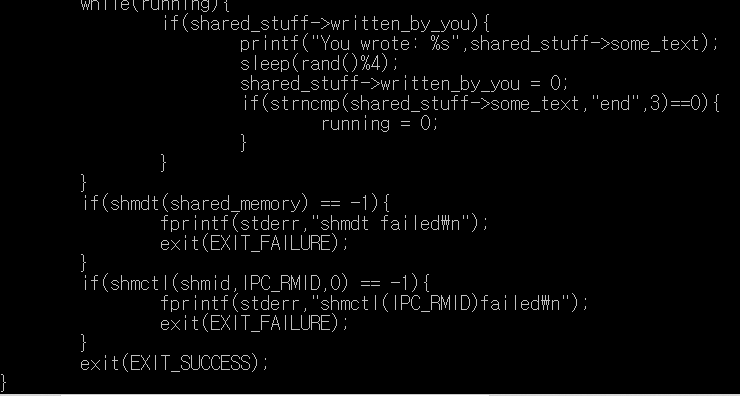
**9장 실습**

**1. 첨부된 Shared Memory & Mapped Memory 파일의 내용에 따라 실습을 진행하고 그 내용을 레포트에 포함한다.**



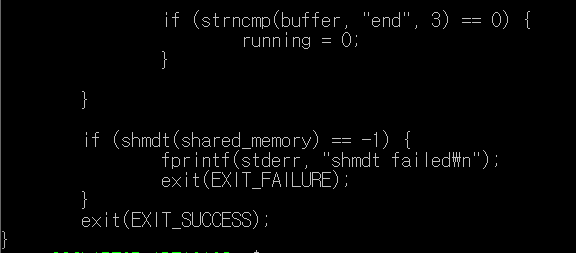
일단 shm\_com.h 파일을 생성한다.





그 이후에 위와 같이 shm1.c파일을 생성한다.

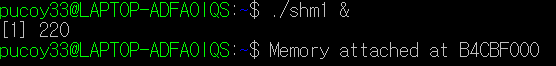




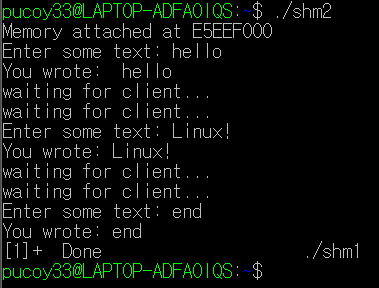
그 이후에 위와 같이 shm2.c파일을 생성한다.







이제 gcc -o shm1 shm1.c 명령어와 gcc -o shm2 shm2.c 명령어를 통해서 shm1.c파일을 사용해서 실행가능한 파일 shm1을 만들고 shm2.c파일을 이용해서 실행가능한 파일 shm2를 생성한다. 그 이후 실습내용과 같이 ./shm1 &을 실행하면 위와 같은 결과화면이 나오는 것을 볼 수 있다.



그 이후에 ./shm2 명령어를 통해 shm2를 실행하면 text를 입력하라고 나오고 실습내용과 같이 텍스트를 입력하면 위와 같은 실행화면이 나오는 것을 볼 수 있다.

**2.Problem A의 주어진 program을 직접 실행해보고 그 결과를 분석하고 보고서를 작성한다.**

**shm\_server.**c

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <stdio.h>

#define SHMSZ 27

main()

{

char c;

int shmid;

key\_t key;

char \*shm, \*s;

/\* We'll name our shared memory segment "5678". \*/

key = 5678;

/\* Create the segment. \*/

if ((shmid = shmget(key, SHMSZ, IPC\_CREAT | 0666)) < 0) {

perror("shmget");

exit(1);

}

/\* Now we attach the segment to our data space. \*/

if ((shm = shmat(shmid, NULL, 0)) == (char \*) -1) {

perror("shmat");

exit(1);

}

/\* Now put some things into the memory for the other process to read. \*/

s = shm;

for (c = 'a'; c <= 'z'; c++)

\*s++ = c;

\*s = NULL;

/\* Finally, we wait until the other process changes the first character of our memory to '\*', indicating that it has read what we put there. \*/

while (\*shm != '\*')

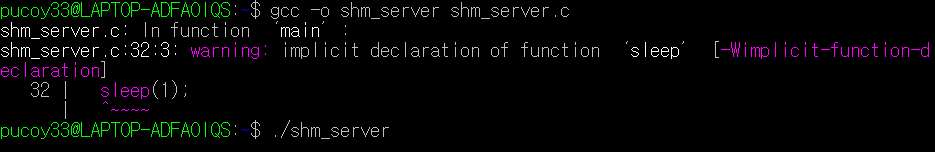
sleep(1);

exit(0);

}



위와 같이 shm\_server.c파일을 생성한다. 이는 메모리에 쓰는 역할을 한다. 따라서 실행가능한 파일을 다음과 같이 만들고 실행을 해도 결과 값이 나오지 않는다.



**shm\_client.c**

/\* \* shm-client - client program to demonstrate shared memory. \*/

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <stdio.h>

#define SHMSZ 27

main()

{

int shmid;

key\_t key;

char \*shm, \*s;

/\* We need to get the segment named "5678", created by the server. \*/

key = 5678;

/\* Locate the segment. \*/

if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {

perror("shmget");

exit(1);

}

/\* Now we attach the segment to our data space. \*/

if ((shm = shmat(shmid, NULL, 0)) == (char \*) -1) {

perror("shmat");

exit(1);

}

/\* Now read what the server put in the memory. \*/

for (s = shm; \*s != NULL; s++)

putchar(\*s);

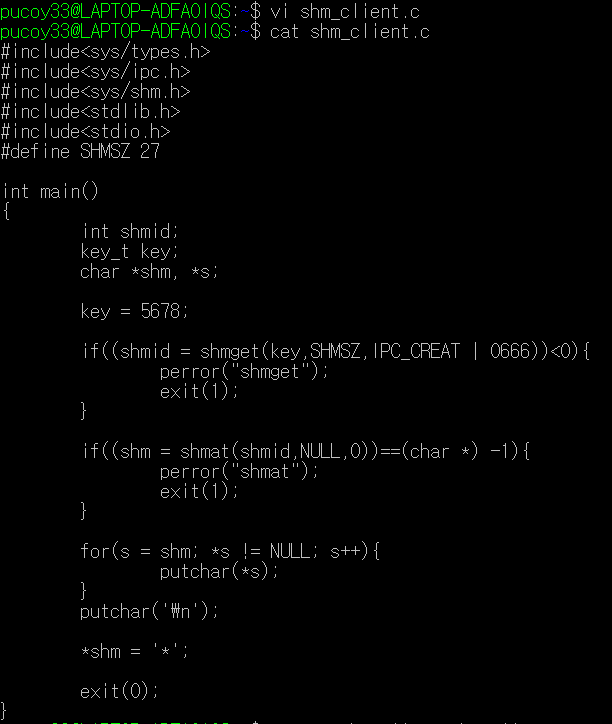
putchar('\n');

/\* Finally, change the first character of the segment to '\*', indicating we have read the segment. \*/

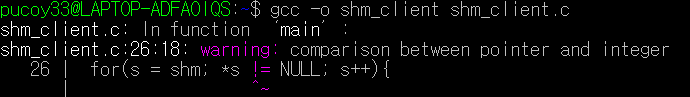
\*shm = '\*';

exit(0);

}



주어진 shm\_client.c파일을 생성한다. 이는 메모리에서 읽는 역할을 한다.



gcc -o shm\_client shm\_client.c 명령어를 통해 shm\_client.c파일을 이용해서 실행가능한 파일 shm\_client를 생성한다.



그 이후 ./shm\_client 명령어를 통해 실행가능한 파일 shm\_client를 실행하면 위와 같은 실행화면이 나오는 것을 확인할 수 있다. 메모리에 있는 문자를 읽어서 출력하는 것이다. 즉 key값만 서로 맞추면 제대로 작동하는 것을 확인할 수 있었다.

**3.해당실습을 진행하고 보고서를 작성하는 동안 느꼈던 개인적인 생각과 느낌을 에세이 형식으로 정리하여 추가한다.**

공유메모리를 생성하고 그에 대해서 메모리를 연결하고 데이터를 생성하고 값을 공유하면서 공유메모리를 다루는 방법에 대해 알 수 있었고 이를 통해서 서버와 클라이언트가 서로 메모리를 공유하면서 공유하는 데이터를 출력하는 것을 실행해 보면서 어떤 식으로 공유메모리를 사용할 수 있는지에 대해서 알 수 있었습니다.

**11장 실습**

**1. 첨부된 Local Socket Connection 파일의 내용에 따라 실습을 진행하고 레포트에 포함한다.**

**A. Try It Out – A simple Local Client <실습>**

#include<sys/type.h>

#include<sys/socket.h>

#include<stdio.h>

#include<sys/un.h>

#include<unistd.h>

int main()

{

int sockfd;

int len;

struct sockaddr\_un address;

int result;

char ch = 'A';

sockfd = socket(AF\_UNIX, SOCK\_STREAM, 0);

address.sun\_family = AF\_UNIX;

strcpy(address.sun\_path, "server\_socket");

len = sizeof(address);

result = connect(sockfd, (struct sockaddr \*)&address, len);

if(result == -1) {

perror("oops: client1");

exit(1);

}

write(sockfd, &ch, 1);

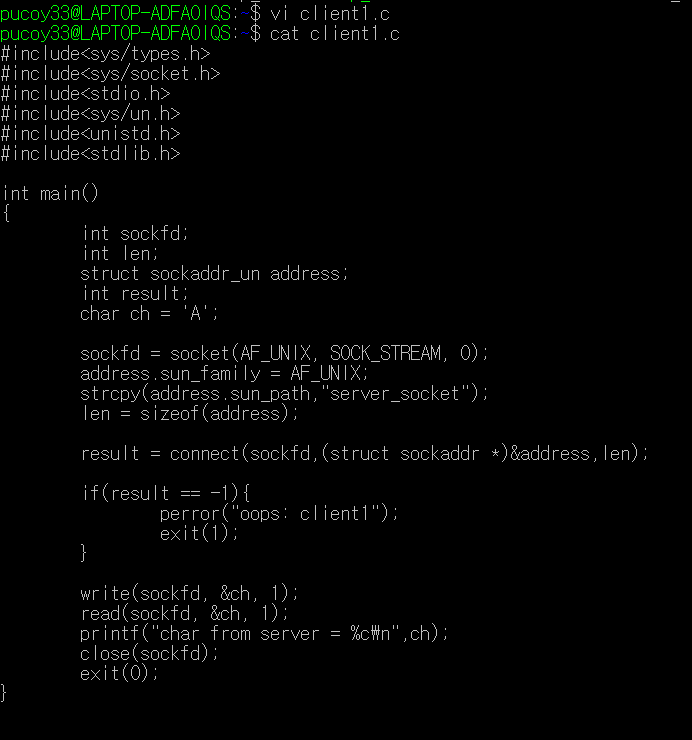
read(sockfd, &ch, 1);

printf("char from server = %c\n", ch);

close(sockfd);

exit(0);

}



일단 client1.c파일을 생성한다.





Gcc -o client1 client1.c 명령어를 통해서 client1.c파일을 이용해서 실행가능한 파일 client1을 생성한다. 그 이후에 client1을 실행하기 위해서 ./client1을 입력하면 아직 server\_socket이 만들어지지 않았기 때문에 연결이 되지 않는다 따라서 oops: client1이 출력되는 것을 확인할 수 있다.

**B. Try It Out – A simple Local Server <실습>**

#include<sys/type.h>

#include<sys/socket.h>

#include<stdio.h>

#include<sys/un.h>

#include<unistd.h>

int main()

{

int server\_sockfd, client\_sockfd;

int server\_len, client\_len;

struct sockaddr\_un server\_address;

struct sockaddr\_un client\_address;

unlink("server\_socket");

server\_sockfd = socket(AF\_UNIX, SOCK\_STREAM, 0);

server\_address.sun\_family = AF\_UNIX;

strcpy(server\_address.sun\_path, "server\_socket");

server\_len = sizeof(server\_address);

bind(server\_sockfd, (struct sockaddr \*)&server\_address, server\_len);

listen(server\_sockfd, 5);

while(1) {

char ch;

printf("server waiting\n");

client\_len = sizeof(client\_address);

client\_sockfd = accept(server\_sockfd, (struct sockaddr \*)&client\_address, &client\_len);

read(client\_sockfd, &ch, 1);

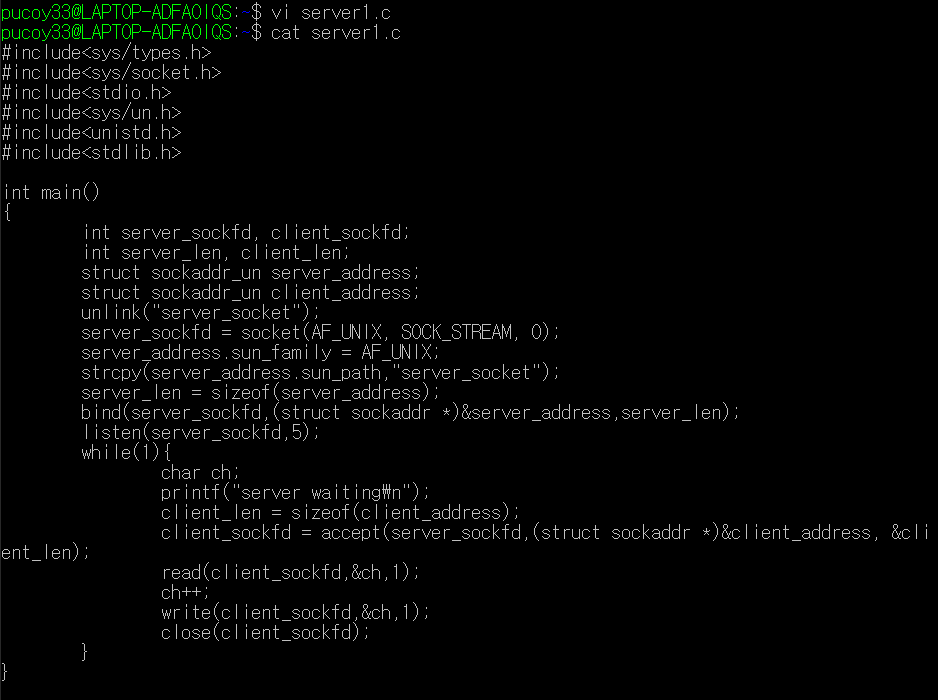
ch++;

write(client\_sockfd, &ch, 1);

close(client\_sockfd);

}

}



일단 sever1.c파일을 생성한다.

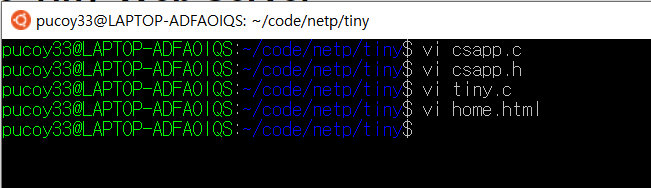


gcc -o server1 server1.c 명령어를 통해서 server1.c파일을 이용해 실행가능한 파일 server1을 생성한다.

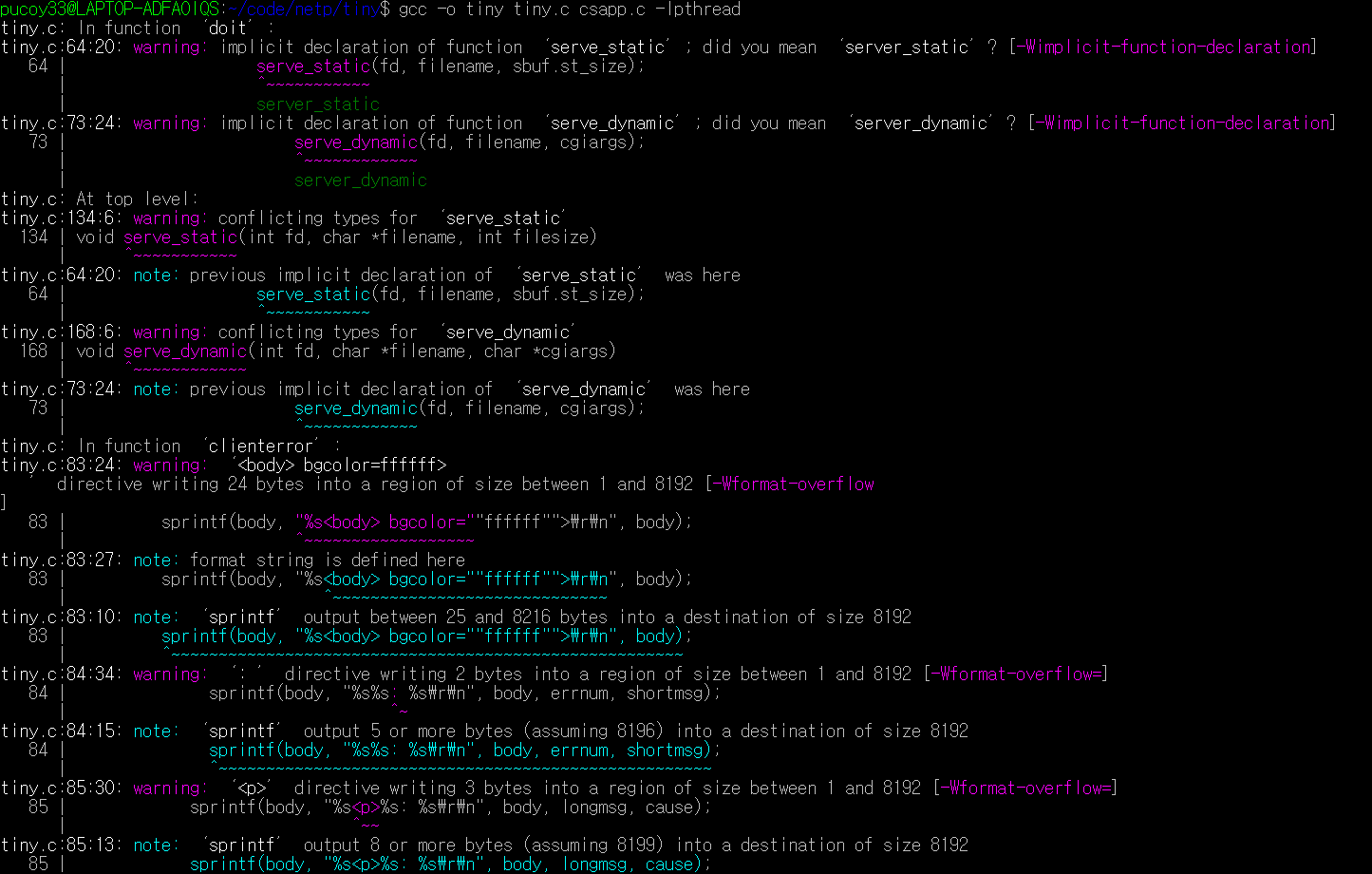


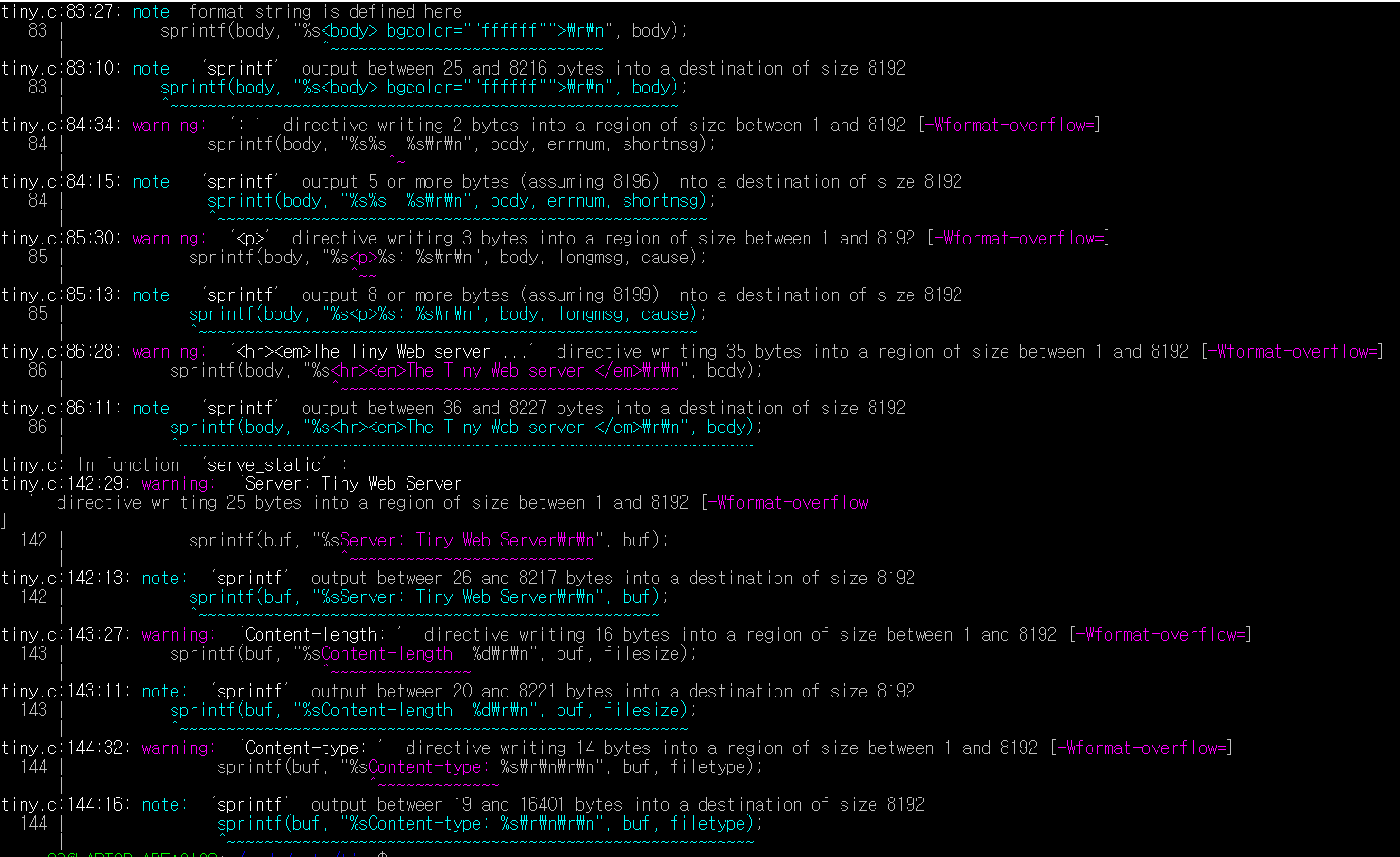
Local Socket Connection파일에 있는 명령어를 동일하게 실행하면 위와 같은 실행화면을 볼 수 있다. ls 명령어를 통해 살펴봤을 때 장치 유형은 socket이고 접근 권한 앞에 s가 붙고 끝에 =이 표시된다. ps명령어를 통해서 백그라운드에서 실행중인 서버를 볼 수 있고 서버가 클라이언트로부터 문자 “A”를 수신해서 이를 증가시키고 B로 반환하는 것을 볼 수 있다. 여러 클라이언트가 같이 실행되면 순차적으로 실행되는 것을 볼 수 있다.

**2. 교재 11.6절 Tiny Web Server를 실행시키고 다양한 테스트를 진행하고 소스를 분석하고 레포트에 포함한다.**



csapp.h, csapp.c, tiny.c, home.html파일을 생성한다.

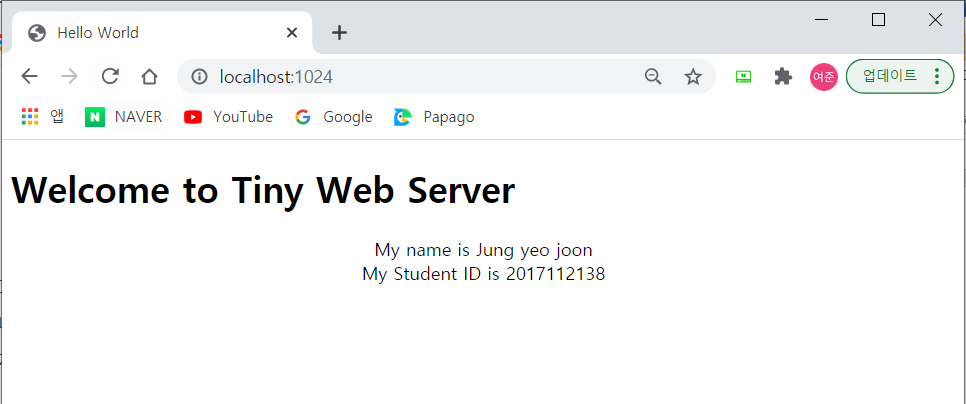




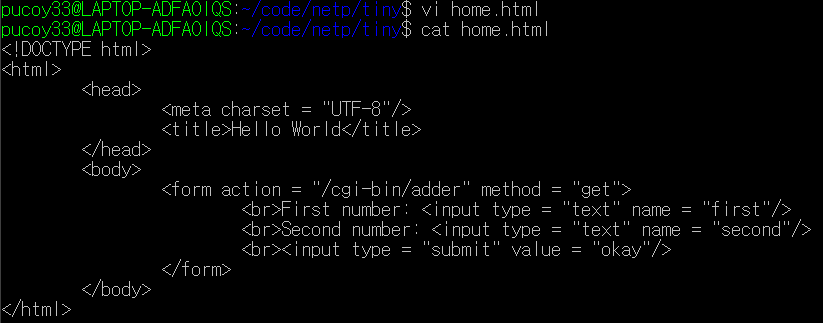
gcc -o tiny tiny.c csapp.c -lpthread명령어를 통해서 tiny.c파일과 csapp.c파일을 링크해서 실행가능한 파일 tiny를 생성한다. 여기서 -lpthread는 csapp.c가 멀티-스레드 라이브러리이므로 추가한 것이다.



그 이후 실행파일 tiny를 실행하기위해서 ./tiny를 하고 그 뒤에 포트번호 1024를 입력하여 실행한다.



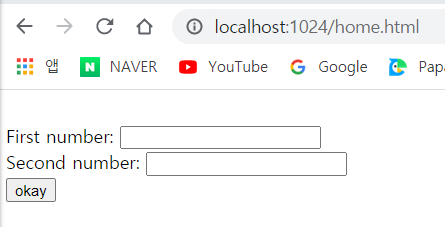
그 이후 localhost:1024를 실행하면 내가 생성한 home.html에 해당하는 내용이 실행되는 것을 확인할 수 있다.



또한 위와 같이 home.html을 수정해준다.

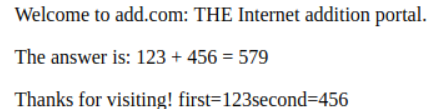
그 이후에 adder.c파일을 생성하고 그 파일을 사용해서 실행가능한 파일 adder을 생성해준다.

그 이후에 ./tiny 1024를 실행하고 <http://localhost:1024/home.html>에 접속하면 다음과 같은 화면을 볼 수 있다.



그 이후에 첫번째 숫자에 123을 입력하고 두번째 수에 456을 입력하고 okay를 클릭하면 url이

<http://localhost:1024/cgi-bin/adder?first=123&second=456>로 이동하고 다음과 같은 화면이 나오는 것을 볼 수 있다.



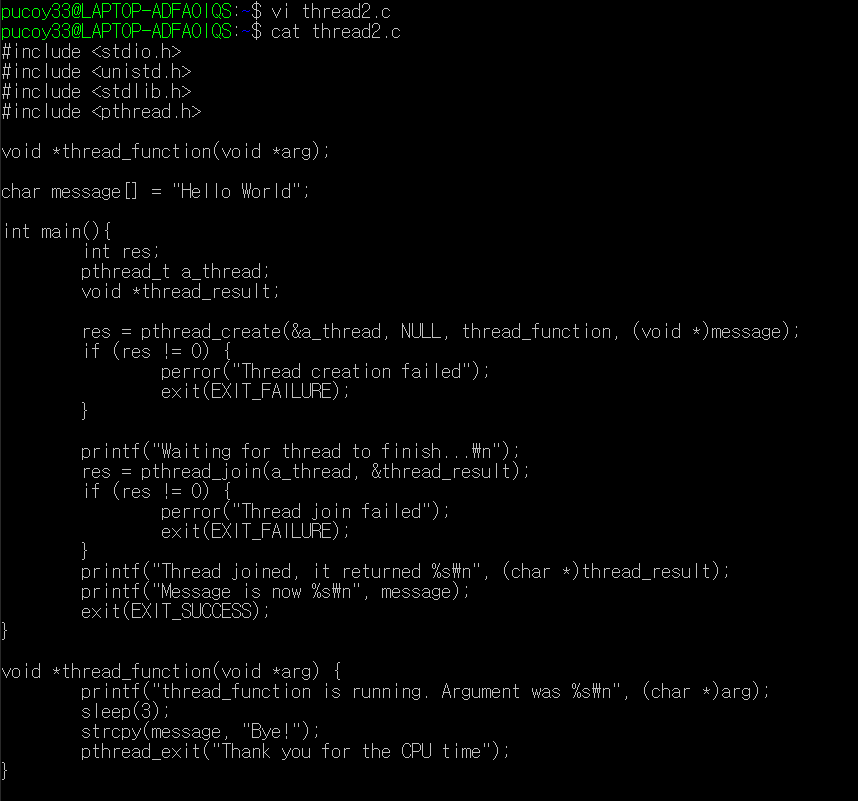
**3. 해당실습을 진행하고 보고서를 작성하는 동안 느꼈던 개인적인 느낌을 에세이 형식으로 정리하여 추가한다.**

Socket client 프로그램을 직접 생성하고 실행해보면서 소켓이 생성되고 연결을 기다리고 연결이 되었을 때 클라이언트가 시작되는 것을 알 수 있었다. 또한 ps -lx명령어를 사용하는 방법에 대해서 알 수 있었고 언제 사용해야 하는지도 배울 수 있었다. 또한 연결이 되기 전까지는 실행이 안되지만 연결이 된 후에는 연동해서 작동하는 것을 볼 수 있었다. 또한 port번호를 입력해서 내가 입력한 port번호에 내가 생성한 tiny web을 실행시키면서 간단한 웹을 만들어 볼 수 있었다. 이를 통해 응용으로 더하기 기능을 수행할 수 있는 웹페이지를 생성하면서 웹페이지를 생성하는 법과 그에 대한 응용도 배울 수 있었다.

**12장 실습**

**1. 첨부된** **Simple Thread Execution 파일의 내용에 따라 실습을 진행하고 레포트에 포함한다.**

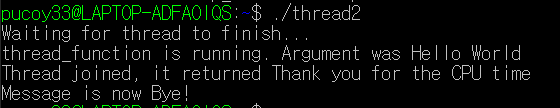
**Try it out - a simple threaded program<실습>**



일단 thread2.c파일을 생성한다.

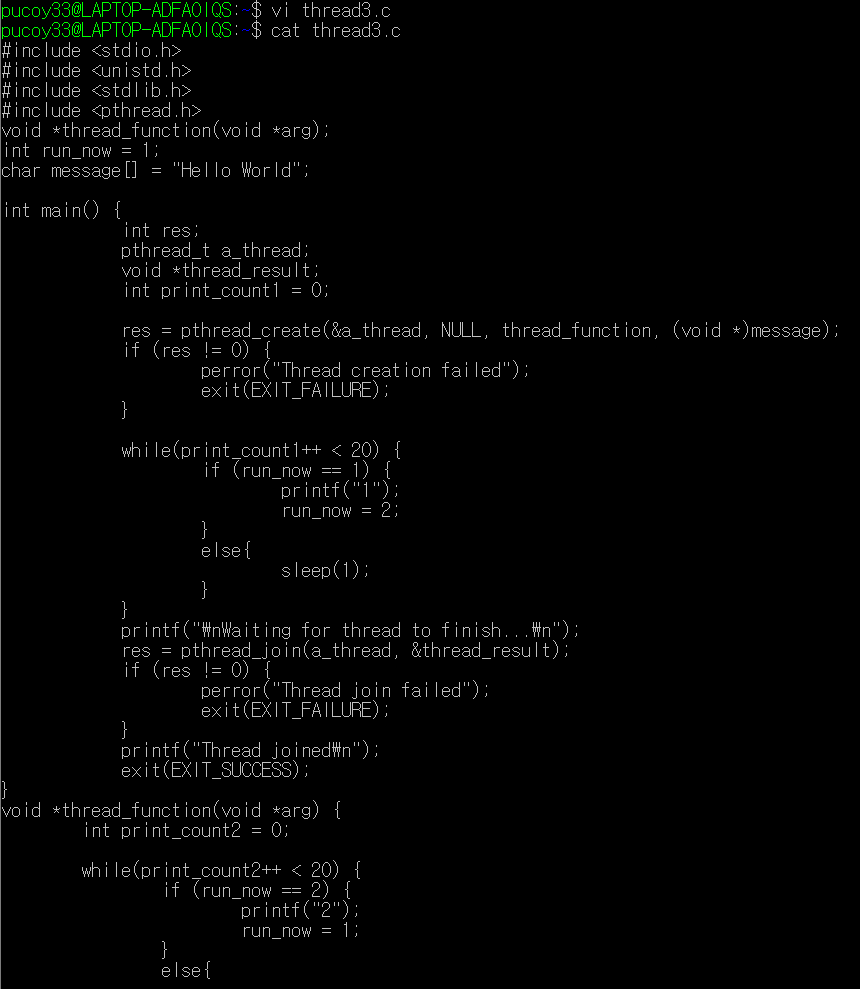


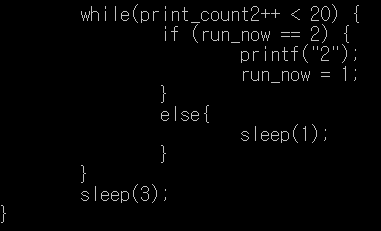
이러한 thread파일을 컴파일하기 위해서는 다음과 같은 명령어가 필요하다. gcc -D\_REENTRAT thread2.c -o thread2 -lpthread를 통해서 thread2.c 파일을 컴파일해서 실행가능한 파일 thread2를 생성한다.



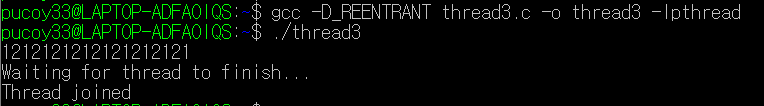
그 이후에 ./thread2 명령어를 통해서 thread2를 실행하면 위와 같은 결과가 나오는 것을 확인할 수 있다.

**Try it out – simultaneous execution of two threads<실습>**



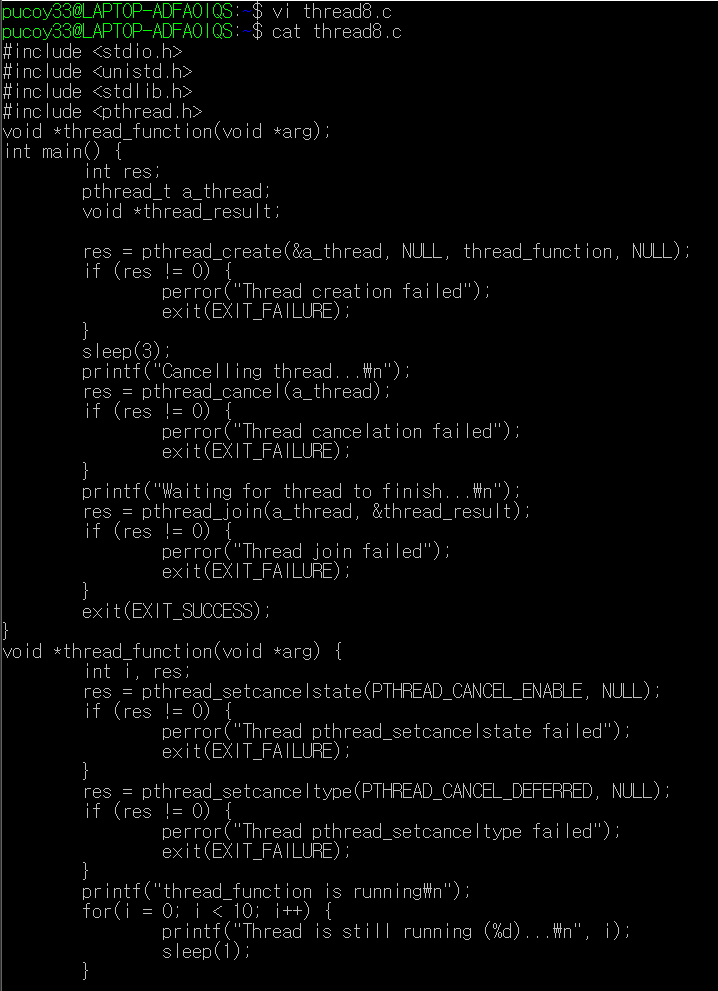


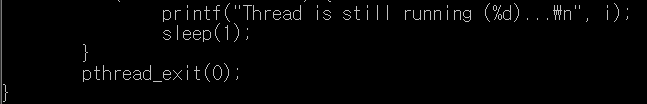
일단 thread2.c에서 실습내용을 추가한 파일 thread3.c파일을 위와 같이 생성한다.



그 이후에 전과 동일하게 gcc -D\_REENTRAT thread3.c -o thread3 -lpthread를 통해서 thread3.c 파일을 컴파일해서 실행가능한 파일 thread3를 생성한다. 그 이후에 ./thread3 명령어를 통해 실행하면 위와 같은 실행화면이 나오는 것을 볼 수 있다.

**Try it out – canceling a threads<실습>**





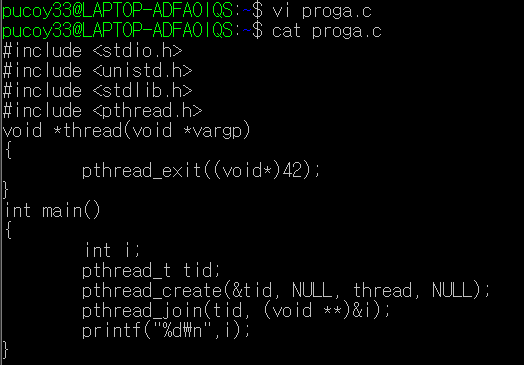
일단 위와 같이 thread8.c파일을 생성한다.



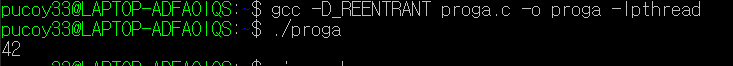
그 이후에 전과 동일하게 gcc -D\_REENTRAT thread8.c -o thread8 -lpthread를 통해서 thread8.c 파일을 컴파일해서 실행가능한 파일 thread8를 생성한다. 그 이후에 ./thread8 명령어를 통해 실행하면 위와 같은 실행화면이 나오는 것을 볼 수 있다.

**2. Problem A의 주어진 program 1, 2, 3, 4, 5를 직접 실행해보고 그 실행 결과를 분석하고 보고서를 작성한다.**

**Program 1**

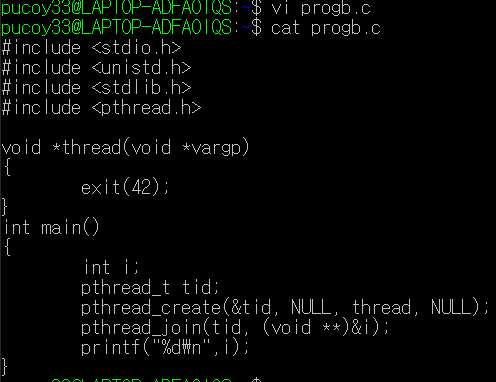


위와 같이 Program 1의 코드 proga.c 파일을 생성한다. 여기서 pthread\_create(&tid, NULL,thread,NULL);을 통해서 아이디가 tid인 thread를 생성한다. 그리고 pthread\_join(tid, (void \*\*)&i);를 통해서 생성했던 스레드를 끝날때까지 기다려준다. i에 반환할 값을 저장해준다. 이 때 i에 42가 저장된다. 따라서 실행하면 42가 출력된다.



위와 같은 명령어를 통해 실행가능한 파일 proga를 생성하고 실행하면 42가 출력되는 것을 확인할 수 있다.

**Program 2**

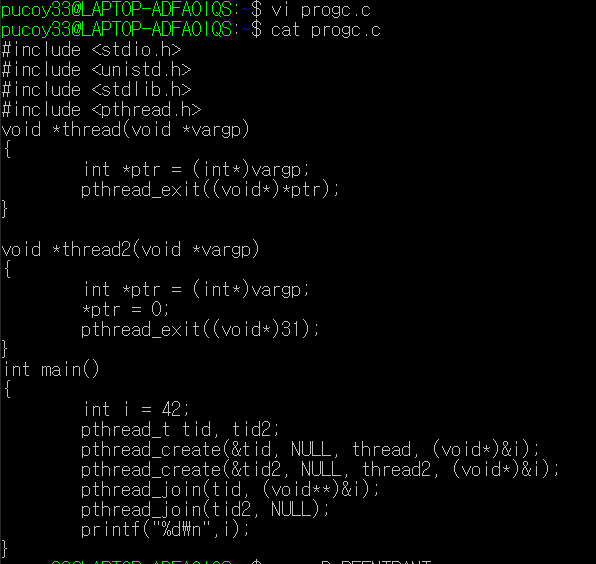


위와 같이 progb.c 파일을 생성한다. Proga.c와 동일하게 메인함수를 만들었지만 void \*thread함수가 다르고 이것 때문에 i에 저장되는 값이 달라진다. 여기서는 exit(42)를 통해서 아무 값도 저장되지 않기 때문에 출력되는 것이 없을 것이다.



위와 같은 명령어를 통해서 실행가능한 파일 progb를 생성하고 실행할 경우에 위와 같이 아무런 값도 출력되지 않는 것을 확인할 수 있다.

**Program 3**



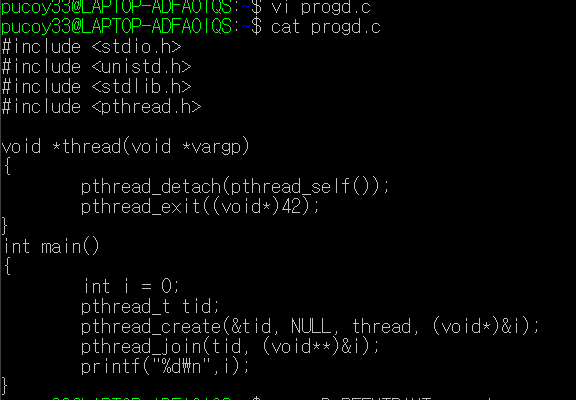
위와 같이 progc.c 파일을 생성한다. 스레드를 두개를 생성하고 i에는 초기에 42의 값으로 초기화 해준다. 이는 pthread\_join(tid, (void \*\*)&i)에 의해서 void \*thread(void \*vargp)함수에서 매개변수 \*vargp에 42가 들어가게 되고 이를 \*ptr에 대입하므로 \*ptr 또한 42가 되고 이를 통해서 program 1과 동일한 결과 값이 나올 것으로 예상된다.





위와 같이 실행가능한 파일 progc를 만들어서 실행하면 42가 출력되는 것을 확인할 수 있다.

**Program 4**

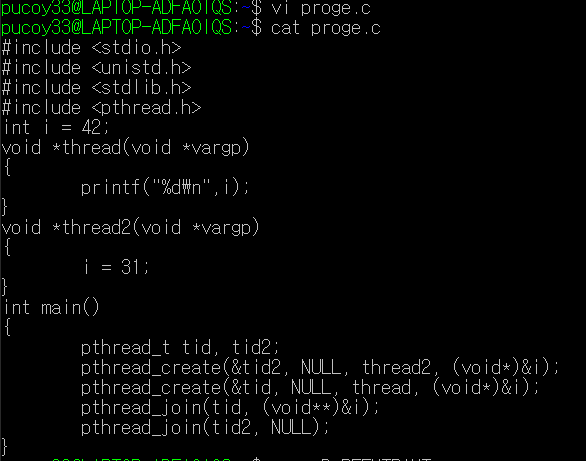


위와 같이 progd.c 파일을 생성한다. 여기서 스레드가 생성될 때 start\_routine에 (void\*)&i값이 전달된다. Pthread\_detatch()함수는 끝나면 알아서 끝내도록 하라는 방식의 함수이다. 즉 독립적인 동작을 하는 대신에 스레드가 끝이나면 반드시 자원을 반환시킨다. Pthread\_create만으로 스레드를 생성하면 루틴이 끝나서도 자원이 반환되지 않는데 이를 해결하는 것이 pthread\_detatch()함수이다. 성공시에 0을 반환한다. pthread\_join함수에 의해서 program 1과 같은 메커니즘으로 42가 출력될 것으로 예상된다.



위와 같은 명령어를 통해서 실행가능한 파일 progd를 생성해주고 이를 실행하면 42가 출력되는 것을 볼 수 있다.

**Program 5**



위와 같이 proge.c 파일을 생성한다. 전역변수로 i를 설정하고 42로 초기화한다. 스레드는 두개를 생성한다. 스레드 1은 스레드 2가 종료되기까지 기다리므로 i에는 31이 대입되고 그렇기 때문에 스레드1이 마지막에 종료되면서 31이 출력될 것으로 예상된다.



위와 같이 명령어를 통해서 실행가능한 파일 proge를 생성하고 이를 실행하여 31이 출력되는 것을 볼 수 있다.

**3. 해당실습을 진행하고 보고서를 작성하는 동안 느꼈던 개인적인 생각과 느낌을 에세이 형식으로 정리하여 추가한다.**

스레드를 생성하고 pthread\_함수들의 사용법과 특징 그리고 정의 등을 알게 되었고 스레드가 먼저 종료되고 기다리고 나중에 종료되는 것과 같은 것을 배워서 실행시에 어떠한 결과 값이 나올지에 대해서 예상하고 해석할 수 있게 되었다. 또한 스레드를 캔슬링하는 것도 배울 수 있어서 좋았습니다.