2018008059 컴퓨터 소프트웨어학부 김은수

운영 체제 HW#3

제출 일자 : 2020/04/09

A. 과제 A

- Named Pipe

1. 프로그램 설명

"namedpipe_version"이라는 디렉토리에 writer_Aprocess.c 파일과 그 실행파일인 writer_Aprocess 그리고 reader_Bprocess.c 파일과 그 실행파일인 reader_Bprocess가 있다. 그리고 컴파일할 때 사용되는 Makefile과 프로그램이 실행할 때 생성되는 named pipe인 named_pipe_file이 있다.

-writer_Aprocess.c

Open(PIPENAME, O_WRONLY)를 통해 PIPE를 쓰기만 가능하도록 열었다. PIPENAME은 #define PIPENAME "./named_pipe_file" 로 설정하였고, named_pipe_file은 reader_Bprocessor를 실행하면 생기는 파이프이다.

예제처럼 open이 성공하면 "Hello, this is A procrss. I'll give the data to B"를 출력하도록 해주었다.

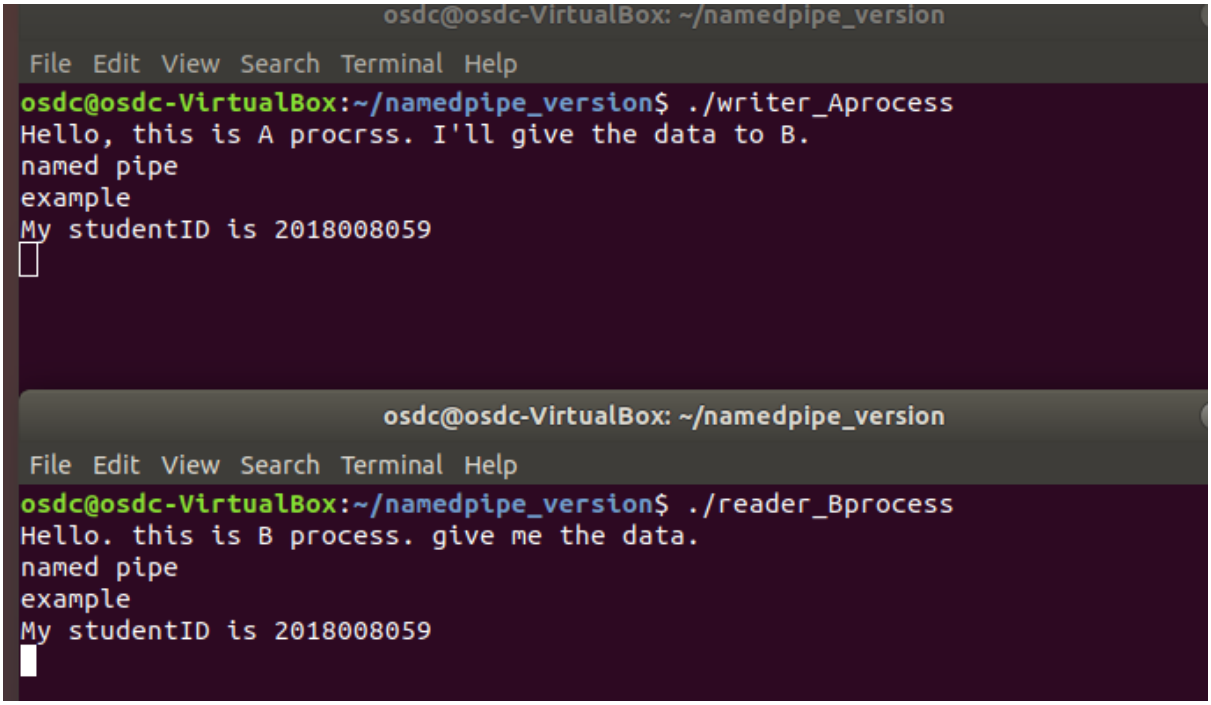
while문을 사용해서 계속 입력을 받도록 했다. String[200]이라는 문자열을 선언하고 fgets를 사용해서 문자열을 받았다. Fgets(string, sizeof(string), stdin)을 통해 3번째 인수로 stdin을 넣어서 키보드에서 입력을 받고, 그 입력을 string에 저장해 주었다. Fgets이 string을 받을 때는 "₩n" 또한 받기 때문에 마지막에 들어오는 "₩n"을 '₩0'로 바꿔주어 개행 문자에 대한 입력을 지웠다. Snprintf 함수를 통해서 string문자열이 msg buffer에 저장하도록 해주었고 write함수를 통해서 데이터 전송영역인 fd에 msg를 msg 크기만큼 쓰도록 해주었고 return값을 ret에 저장했다. Write가 실패하면 -1을 return하기 때문에 ret가 0보다 작다면 "Write failed"을 출력하도록 했다.

```
fd = open(PIPENAME, O_WRONLY);
if(fd<0)
{
    printf("Open failed\n");
    return 0;
}
printf("Hello, this is A procrss. I'll give the data to B.\n");

while(1)
{
    fgets(string,sizeof(string),stdin);
    string[strlen(string)-1] = '\0';
    snprintf(msg,sizeof(msg),"%s",string);
    ret = write(fd, msg, sizeof(msg));
    if(ret<0)
    {
        printf("Write failed\n");
        return 0;
    }
}
return 0;
```

-reader_Bprocess.c

#define PIPENAME "./named_pipe_file" 을 통해 생성할 PIPE의 이름을 named_pipe_file로 정했다. Access를 통해서 name_pipe_file이라는 이름의 Named pipe가 존재하는 지 조사해주고 있다면 그 return 값이 0이므로 access의 return값을 저장한 ret가 0이라면 unlink를 통해 지워주었다. mkfifo를 통해 name_pipe_file이라는 이름의 파이프를 생성했다. Open(PIPENAME, O_RDWR)를 사용해서 생성한 파이프를 읽기/쓰기 모드로 열었고 그 리턴값은 fd에 저장했다. Open이 성공했다면 예제처럼 "Hello. This is B process. Give me the data"를 출력하도록 했다. 그리고 whlie문안에 read함수를 통해서 open함수로 연 fd의 내용을 msg버퍼에 넣었다. read함수가 실행을 실패했으면 -1을 반환하기 때문에 0보다 작을 때는 "Read failed"를 출력하고 정상적이라면 받은 문자열 msg를 출력하도록 해주었다.

```
fd = open(PIPENAME,O_RDWR);

if(fd<0)
{
    printf("Opening of named pipe failed\n");
    return 0;
}
printf("Hello. this is B process. give me the data.\n");

while(1)
{
    ret = read(fd, msg, sizeof(msg));
    if(ret<0)
    {
        printf("Read failed\n");
        return 0;
    }
    printf("%s\n",msg);
}
return 0;
```

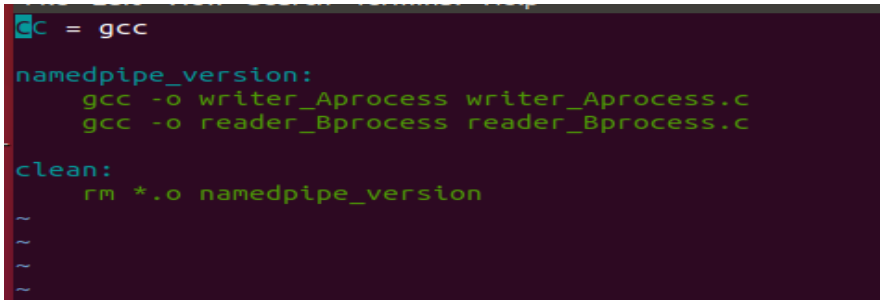


실행한 모습이다. reader_Bprocess에서 pipe를 만들기 때문에 reader_Bprocess를 먼저 실행시키고 writer_Aprocess를 실행해야 한다. 종료는 터미널에서 ctrl-c를 눌러서 종료하면 된다.

2. IPC 메커니즘 설명

단방향 통신으로 writer_Aprocess에서 문자열을 쓰면 reader_Bprocess에서 문자열을 읽는다. Named pipe는 사용할 파이프를 명명할 수 있다. 이 실습에서는 파이프로 사용할 파일을 named_pipe_file라는 이름을 가지도록 해주었다. reader 프로세서에서 Mkfifo() 함수로 생성하여 이후로 이 파이프를 가지고 송수신이 가능하도록 해주었다. File에 writer가 문자열을 쓰면 reader는 그 문자열을 FIFO 방식으로 먼저 들어온 데이터를 읽어 이 프로그램에서는 출력한다.

3. 컴파일 방법 설명



```
osdc@osdc-VirtualBox:~/namedpipe_version$ make
gcc -o writer_Aprocess writer_Aprocess.c
gcc -o reader_Bprocess reader_Bprocess.c
```

목적파일을 namedpipe_version이라 하고 실행해야 할 명령어를

gcc -o writer_Aprocess writer_Aprocess.c

gcc -o reader_Bprocess reader_Bprocess.c

로 두어서 터미널에서 make를 실행했을 때 writer_Aprocess와 reader_Bprocess라는 실행파일이 생기도록 컴파일을 해주었다.

- **Message Queue**

4. 프로그램 설명

Messagequeue_version 디렉토리에 writer_Aprocess.c 파일과 그것의 실행파일인 writer_Aprocess, reader_Bprocess.c 파일과 그것의 실행파일인 reader_Bprocess가 들어있다. 또 이를 컴파일하는 Makefile이 들어있다.

-writer_Aprocess.c

Message queue에 담을 메시지를 msgbuf라는 이름의 구조체로 정의했다. Message queue에 담을 메시지를 구별하기 위해서 사용되는 변수를 msgtype로 두고, 담을 문자열을 이름은 mtext에 크기는 200으로 설정해주었다. msgget함수를 사용하여 message queue를 생성하거나 생성된 객체에 참조를 할 수 있도록 했는데, 메시지큐를 구분하기 위한 key값을 1234로 해주었고 옵션으로는 IPC_CREAT를 써서 1234에 해당하는 큐가 있다면 큐의 식별자를 반환하고 없으면 생성하도록 했다. Msgget이 성공하면 큐의 식별자가 반환되고 이를 key_id에 할당했다. Sndbuf라는 이름에 큐에 투입할 메시지를 선언했다. 이 큐의 msgtype을 3으로 두어서 보낼 메시지가 1234메시지큐에 3번째 메시지라고 설정해주었다. while문안에서 fgets 함수를 사용하고 stdin을 3번째 인자로 넣어서 터미널에서 입력한 문자열을 string이라는 문자열에 저장해주도록 했다. string문자열은 코드초기에 char string[200]으로 선언해주었다. fgets함수는 개행문자도 받으므로 문자열의 마지막을 '₩0'으로 할당해 개행문자를 제거해주었다.

그 다음 strcpy(sndbuf.mtext,string)을 통해 받은 string 문자를 보낼 메시지 sndbuf의 mtext에 복사했다. mgsnd함수에 큐의 식별자와 보내고자 하는 버퍼인 sndbuf의 주소를 넣었다. 또한 메시지 버퍼의 크기와 IPC_NOWAIT를 넣어주었다. msgsnd 함수가 실패하면 -1을 반환하기 때문에 -1일때는 오류 메세지를 출력하도록 했고 정상적으로 실행된다면 예시처럼 "sending nth message is succeed"를 출력하도록 했다. 예시에서 몇 번째

문자인지 출력했으므로 i 라는 변수를 코드 초기에 선언하여 message가 출력할 때마다 i가 1씩 증가하여 몇 번째 메세지인지 확인할 수 있도록 설정했다.

```
struct msgbuf
{
    long msgtype;
    char mtext[200];
};

int main(void)
{
    key_t key_id;
    char string[200];
    int i=0;
    struct msgbuf sndbuf;

    key_id = msgget((key_t)1234,IPC_CREAT|0666);

    if(key_id == -1)
    {
        perror("msgget error : ");
        return 0;
    }

    sndbuf.msgtype =3;
    while(1)
    {

        fgets(string, sizeof(string), stdin);
        string[strlen(string)-1] = '\0';
        strcpy(sndbuf.mtext, string);


        if(msgsnd(key_id, (void*)&sndbuf,sizeof(struct msgbuf),IPC_NOWAIT)==-1)
        {
            perror("msgsne error : ");
        }
        else
        {
            printf("Sending %dth message is succeed\n",i);
            i++;
        }
    }

    return 0;
}
```

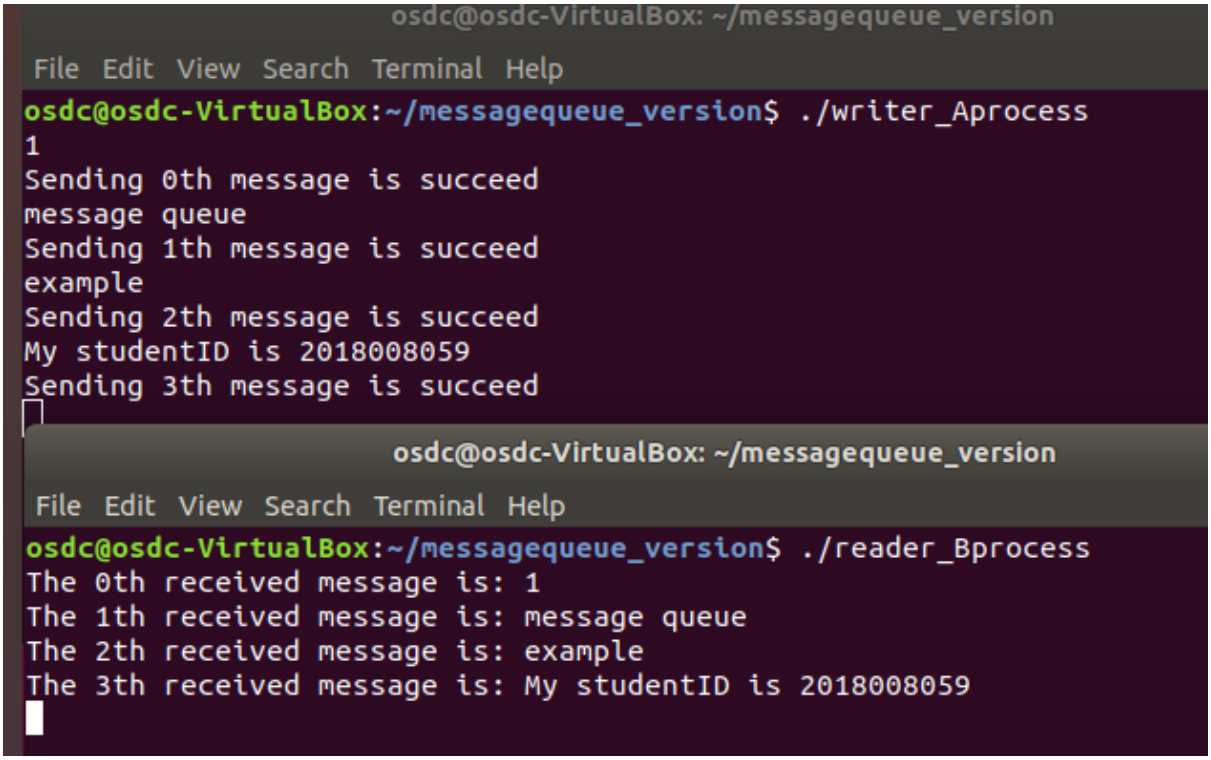
-reader_Bprocess.c

writer에서 보낸 메시지를 얻을 것이므로 message queue 메시지 자료구조를 writer와 동일하게 설정했다. 얻은 메시지가 담길 구조체를 rsvbuf라고 선언해 주었다. Writer_Aprocess.c 코드와 동일하게 msgget((key_t)1234, IPC_CREAT|0666)함수를 사용해서 1234 key 값을 가지는 메시지 큐에 접근하거나 없다면 create해주었다. Msgget이

성공적으로 실행이 되었다면 넣을 메시지 큐의 id가 반환되고 이를 key_id에 할당해주었다. 실패하여 key_id가 -1이면 이 프로그램을 종료하고 성공했다면, while문을 실행해주어 메시지를 계속 읽도록 해주었다. Msgrcv 함수를 사용해서 가져올 메시지가 담긴 message queue id인 key_id를 넣어주었고 메시지를 담을 구조체인 rsvbuf의 주소를 넣어주었다. Writer가 1234메시지큐에 3번째에 메시지를 넣을 것이므로 msgtype을 3으로 설정해주었고 그 msgtype을 msgrcv의 인자로 넣어주었다. Msgrcv가 실패하면 -1을 반환하고 성공하며 읽은 데이터의 크기를 반환하기 때문에 -1이면 에러메시지를 출력하고 성공하면 읽은 메시지를 출력하도록 했다. 예시를 보면 몇 번째 문자열을 받았고 그 문자열이 무엇인지 출력한다. 따라서 "the nth received message is :" 라는 문구와 함께 읽은 문자열을 출력한다. 코드 초기에 i 라는 변수를 선언하였고 출력할 때마다 1씩 증가하여 몇 번째 문자열인지 알 수 있게 해주었다. 읽은 문자열은 rsvbuf의 mtext에 있으므로 printf에 rsvbuf.mtext를 인자로 넣어주었다.

```c
int main(int argc,char **argv)
{
    key_t key_id;
    int i=0;
    struct msgbuf rsvbuf;
    int msgtype = 3;

    key_id = msgget((key_t)1234,IPC_CREAT|0666);

    if( key_id ==-1)
    {
        perror("msgget error : ");
        return 0;
    }

    while(1)
    {
        if(msgrcv(key_id, (void*)&rsvbuf, sizeof(struct msgbuf),msgtype, 0)==-1)
        {
            perror("msgrcv error : ");
        }
        else
        {
            printf("The %dth received message is: %s\n",i,rsvbuf.mtext);
            i++;
        }
    }
    return 0;
```

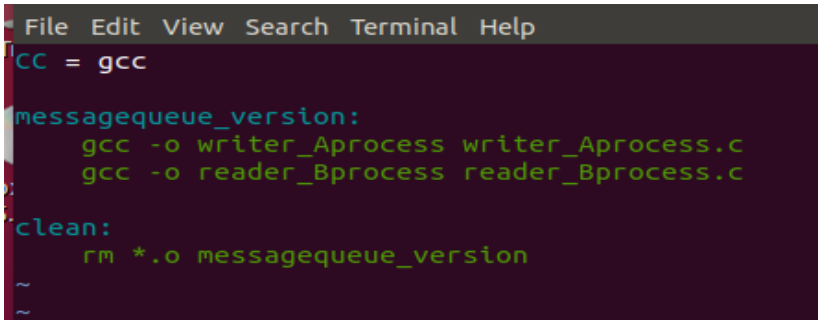
실행한 모습이다. 종료는 ctrl-c를 누르면 된다.



5. IPC 메커니즘 설명

공유하기로 하는 메시지 큐를 만들거나 key값을 통해 찾는다. Writer 프로세스가 키보드로 생성한 메시지가 IPC 메시지 큐에 저장되고 그 메시지를 다른 reader 프로세서가 읽는다. 이때 무슨 메시지 큐인지 확인하는 key뿐만 아니라 그 큐에 어디에 저장할 것인지 하는 msgtype 또한 reader와 writer가 같이 설정해 주어야한다. 여기서는 두 프로세스 모두 3으로 설정해주어서 writer와 reader가 같은 큐 장소 안에서 메시지를 읽고 쓸 수 있다.

6. 컴파일 방법 설명



```
osdc@osdc-VirtualBox:~/messagequeue_version$ make
gcc -o writer_Aprocess writer_Aprocess.c
gcc -o reader_Bprocess reader_Bprocess.c
```

목적파일을 messagequeue_version이라 하고 실행해야 할 명령어를

gcc -o writer_Aprocess writer_Aprocess.c

gcc -o reader_Bprocess reader_Bprocess.c 로 두어서 터미널에서 make를 실행했을 때 writer_Aprocess와 reader_Bprocess라는 실행파일이 생기도록 컴파일을 해주었다.

- **Shared Memory**

7. 프로그램 설명

Messagequeue_version 디렉토리에 writer_Aprocess.c 파일과 그것의 실행파일인 writer_Aprocess, reader_Bprocess.c 파일과 그것의 실행파일인 reader_Bprocess가 들어있다. 또 이를 컴파일하는 Makefile이 들어있다.

-writer_Aprocess.c

Writer가 값을 썼을 때 reader가 이를 알 수 있어야한다. 따라서 #define을 통해 READ_WRITER_FLAG와 READ_READER_FLAG 두 flag를 선언하였다. 이는 나중에 다시 설명하겠다. Shmget을 통해 1234에 해당하는 shared memory를 얻어오고 없다면 생성하였다. Shmget을 통해 리턴되는 shared memory의 id는 shmid에 할당했다. 그 후 프로세서의 메모리 공간에 공유 메모리를 붙여야 하므로 shmat함수를 사용하였다. Shmat 인자에 shmid를 넣었고, 두번째 인자로 0을 넣어서 반환값으로 나오는 메모리 주소를 동적으로 설정하겠다고 했다. 그리고 shmat의 return값으로 나오는 메모리 주소를 shmaddr 포인터 변수가 가리키도록 했다. 앞서 말했듯이 writer가 값을 썼는 지 확인을 해줘야 하는데 이는 공유 메모리주소의 0번지에 줄 것이다. 따라서 shmaddr[0]을 READ_WRITER_FLAG로 설정해준다. 이 flag일 때는 write를 할 수 있다. 따라서 실질적인 문자열은 string으로 설정하고 이 string은 shmaddr +1 지점을 가리킨다.

```c
#define READ_WRITER_FLAG 0
#define READ_READER_FLAG 1

int main(void)
{
    int shmid;
    char *shmaddr;
    char *string;
    int ret;

    shmid =shmget((key_t)1234,1024,IPC_CREAT|0666);
    if(shmid<0)
    {
        perror("shmget");
        return 0;
    }

    shmaddr = shmat(shmid, (void *)0,0);
    if(shmaddr == (char *)-1)
    {
        perror("attach failed\n");
        return 0;
    }

    shmaddr[0] = READ_WRITER_FLAG;

    string = shmaddr +1;
```

While문을 통해 계속 입력을 받도록 해준다. 입력을 받을 때는 shamaddr의 0번지가 READ_WRITER_FLAG여야한다. Fgets(string,200,stdin)을 통해 키보드로부터 입력을 받는데 크기는 200으로 제안해주었고, 이때 string은 shmaddr+1인 것을 잊지 말아야한다. 또한 입력을 받았으면 reader가 이를 알 수 있도록 shmaddr[0]을 READ_READER_FLAG로 바꿔준다. 그리고 named pipe 와 message queue같은 경우에는 while문 다음에 실행해주는 코드가 없어 ctrl-c로 종료해주었지만, shared memory 같은 경우는 뒤에도 실행할 코드가 있기 때문에 quit를 입력하면 종료하도록 했다. (물론 키보드에서 ctrl-c를 눌러도 종료된다.) 받는 string이 quit과 같을 경우 strcmp의 반환 값은 0이기 때문에 !strcmp를 사용해서 같으면 if문안을 실행해서 while문이 끝나도록 해주었다. 그 다음 shmdt(shmaddr)을 사용해서 공유메모리 주소인 shmaddr를 프로세스 메모리 공간에서 떼 주었다.

```c
    while(1){

        if(shmaddr[0] == READ_WRITER_FLAG)
        {
            fgets(string, 100, stdin);
            string[strlen(string)-1]='\0';

            shmaddr[0]=READ_READER_FLAG;
            if(!strcmp(string, "quit"))
            {
                break;
            }
        }

    }

    ret = shmdt(shmaddr);

    if(ret == -1)
    {
        perror("detach failed\n");
        return 0;
    }

    return 0;
}
```

-reader_Bprocess.c

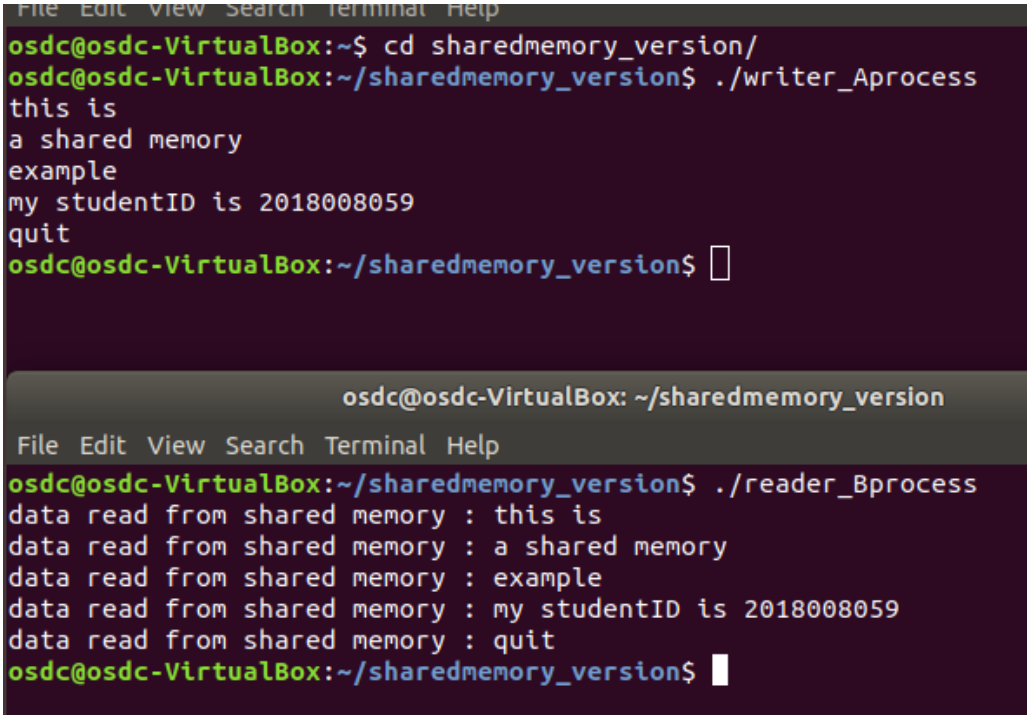
마찬가지로 Writer가 값을 썼을 때 reader가 이를 알 수 있어야한다. 따라서 #define을 통해 READ_WRITER_FLAG와 READ_READER_FLAG 두 flag를 선언하였다. Shmget을 통해 1234에 해당하는 shared memory를 얻어오고 없다면 생성하였다. 그 후 프로세서의 메모리 공간에 공유 메모리를 붙여야 하므로 shmat함수를 사용하였다. Shmat인자에 shmget으로 얻어온 shared memory id인 shmid를 넣었고 그 위치를 shmaddr 포인터 변수가 가리키도록 했다. read함수에서도 shmaddr[0]은 READ_WRITER_FLAG이고 실질적 문자열은 그 다음 번지부터 써진다. 따라서 실질적으로 써지는 문자열 string을 따로 선언해주고 string = shmaddr+1로 선언해주었다.

```c
#define READ_WRITER_FLAG 0
#define READ_READER_FLAG 1

int main(void)
{
    int shmid;
    char *shmaddr;
    int ret;
    char *string;

    shmid = shmget((key_t)1234,1024,IPC_CREAT|0666);
    if(shmid == -1)
    {
        perror("shared memory access is failed\n");
        return 0;
    }

    shmaddr = shmat(shmid, (void *)0,0);
    if(shmaddr ==(char *)-1)
    {

        perror("attach failed\n");
        return 0;
    }


    string = shmaddr +1;

    shmaddr[0] = READ_WRITER_FLAG;
```

Shmaddr가 READ_READER_FLAG라는 얘기는 writer가 글을 썼고 이를 읽으라는 뜻이다. 따라서 string을 출력한다. 그리고 다시 출력하지 않도록 shmaddr[0]을 READ_WRITER_FLAG로 설정해준다. Reader도 마찬가지로 while문 이후에도 실행할 코드가 있기 때문에 quit라는 문자열이 들어오면 break를 통해서 while문을 빠져나오도록 한다. 그다음 shmdt함수를 통해서 shmaddr을 프로세서 메모리에서 떼어내고 shared memory를 제거하기위해 shmctl 함수에 지우려는 shared memory의 id인 shmid와 제거할 것이므로 IPC_RMID를 인자로 넣는다.

```
while(1)
{
    if(shmaddr[0]==READ_READER_FLAG)
    {

        printf("data read from shared memory : %s\n", string);
        shmaddr[0]=READ_WRITER_FLAG;
        if(!strcmp(string, "quit"))
        {
            break;
        }
    }


}

ret = shmdt(shmaddr);
if(ret == -1)
{
    perror("detach failed\n");
    return 0;
}

ret= shmctl(shmid, IPC_RMID,0);
if(ret ==-1)
{
    perror("remove failed\n");
    return 0;
}

return 0;
```

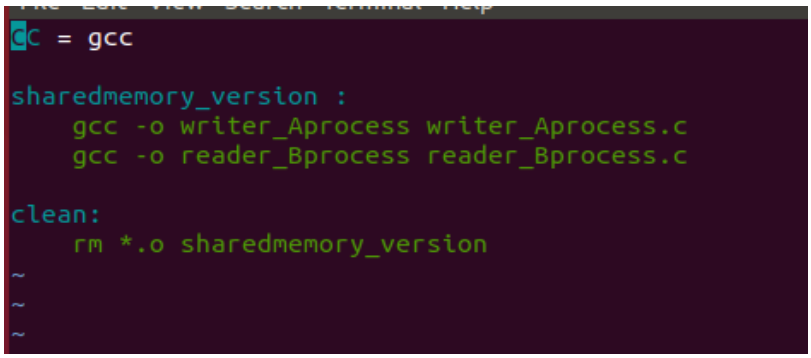
실행 모습이다. 종료는 ctrl-c로도 가능하고 quit를 입력했을 때도 종료된다.

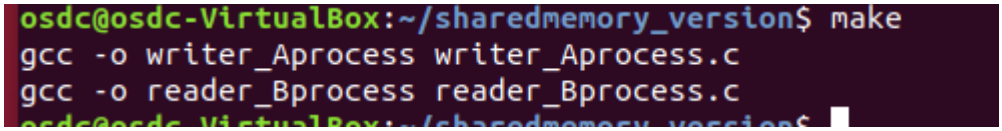


8. IPC 메커니즘 설명

여러 개의 프로세스가 특정 메모리 영역을 공유하도록 하는 방식이다. 이 특정 메모리 영역의 주소를 가져와야 하는데 shmat 함수를 통해서 가져온다. 또한 고유번호가 있어 객체를 식별할 수도 있다. 이 함수를 통해 가져온 메모리 영역을 프로세스에서 두 프로세스가 문자열을 쓰고 읽으며 데이터를 공유한다. 즉, 공유된 메모리 영역을 통해서 통신을 한다

9. 컴파일 방법 설명





목적파일을 sharedmemory_version이라 하고 실행해야 할 명령어를

gcc -o writer_Aprocess writer_Aprocess.c

gcc -o reader_Bprocess reader_Bprocess.c 로 두어서 터미널에서 make를 실행했을 때 writer_Aprocess와 reader_Bprocess라는 실행파일이 생기도록 컴파일을 해주었다.

B. 과제 B

1. 멀티스레딩 내용

멀티 스레딩이란 하나의 프로세스를 다수의 실행 단위로 구분하여 자원을 공유하고 자원의 생성과 관리의 중복성을 최소화하여 수행 능력을 향상시키는 것을 말한다. 즉 하나의 프로그램에 동시에 여러 개의 일을 수행할 수 있도록 해주는 것이다. 멀티 쓰레드를 이용하여 프로세스를 이용하여 동시에 처리하던 일을 쓰레드로 구현할 경우 매모리 공간과 시스템 자원 소모가 줄어들게 된다. 쓰레드 간의 통신이 필요할 경우에도 별도의 자원을 이용할 필요없이 전역 변수의 공간 또는 힙(heap) 영역을 이용하여 데이터를 주고받을 수 있다. 그렇기 때문에 프로세스 간 통신 방법에 비해 쓰레드 간의 통신 방법이 훨씬 간단하다. 즉, 멀티 쓰레딩을 사용하면 프로세스 생성에 들어가는 많은 시간과 자원을 줄일 수 있다. 또한 스택 영역만 비 공유하고 데이터 영역과 힙영역을 공유하므로 데이터 교환에 특별한 기법이 필요 없다. 또한 쓰레드의 생성 및 컨텍스트 스위칭이 프로세스의 생성 및 컨텍스트 스위칭 보다 빠르다. 이 컨텍스트 스위칭은 프로세스 간 혹은, 스레드 간 교체할 때 생기는 것으로 현재까지의 작업상태나 다음 작업에 필요한 데이터를 저장하고 읽어오는 작업을 가리킨다. 하지만 멀티 스레드의 단점으로는 서로 다른 쓰레드가 데이터와 힙 영역을 공유하기 때문에 어떤 쓰레드가 다른 쓰레드에서 사용 중인 변수나 자료 구조에 접근하여 엉뚱한 값을 읽어오거나 수정할 수 있다. 따라서 동기화 작업이 필요하다. 동기화를 통해서 작업 처리 순서를 컨트롤하고 공유 자원에 대한 접근을 컨트롤을 해야 한다. 이렇게 동기화할 때는 필요한 부분에만 synchronized 키워드를 통해서 동기화를 해야 한다.