2017029589 컴퓨터소프트웨어학부 류지범

운영 체제 HW#3

제출 일자: 2021/03/29

A. 과제 A

- Named Pipe

1. 프로그램 설명

reader_Bprocess.c (~/OS-2021/HW3/namedpipe_version) - VI

파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

#define PIPENAME "./named_pipe_file"

int main(){
    int ret;
    char msg[80];
    int fd;
    pid_t pid;

    //open the named pipe
    fd = open(PIPENAME, O_RDWR);
    if(fd < 0){
        printf("Opening of named pipe failed\n");
        return 0;
    }

    while(1){
        ret = read(fd, msg, sizeof(msg));
        if(ret < 0){
            printf("Read failed\n");
            return 0;
        }
        //if received msg is exit program quit
        if(strcmp(msg, "exit") == 0){
            printf("-exit this program-\n");
            break;
        }
        printf("the received message is : %s\n", msg);
    }
    return 0;
}
```

writer_Aprocess.c (~/OS-2021/HW3/namedpipe_version) - VIM

파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>

#define MSG_SIZE 80
#define PIPENAME "./named_pipe_file"

int main(void){
    char msg[MSG_SIZE];
    char input[MSG_SIZE];
    int fd;
    int ret, i;

    ret = access(PIPENAME, F_OK);
    if(ret == 0){
        //delete the pip
        unlink(PIPENAME);
    }

    // create a named pipe
    ret = mkfifo(PIPENAME, 0666);
    if(ret < 0){
        printf("Creation of named pipe failed\n");
        return 0;
    }

    //open the named pipe
    fd = open(PIPENAME, O_WRONLY);
    if(fd < 0){
        printf("Open failed\n");
        return 0;
    }

    while(1){
        //get msg from keyboard about 1 line
        fgets(input, sizeof(input), stdin);
        input[strlen(input) - 1] = '\0';
        snprintf(msg, sizeof(msg), "%s", input);
        ret = write(fd, msg, sizeof(msg));
        if(ret < 0){
            printf("Write failed\n");
            return 0;
        }
        //if you enter 'exit' program quit
        if(strcmp(msg, "exit") == 0){
            printf("-exit this program-\n");
            return 0;
        }
    }
    return 0;
}
```

Writer는 pipe를 통해 메시지를 전달하고, reader는 pipe로부터 메시지를 읽어들인다. writer가 처음에 pipe를 생성한다. 이 때 pipe가 이미 존재할 경우, unlink 함수를 통

해 파일을 삭제해준다. Unlink는 해당 파일을 참조하는 count를 1 감소시키며, count 값이 0이 되면 파일을 삭제한다. 이후 mkfifo 함수를 통해 named pipe의 역할을 하는 특수 파일 FIFO를 만든다. 이후 reader는 named pipe 파일을 열고, 메시지를 받을 준비를 한다.

다음으로 writer에 대해 설명해보겠다. Writer는 reader가 생성해놓은 named pipe를 연다. 이후 while문을 통해서 사용자로부터 입력 받은 메시지를 pipe에 쓰는 과정을 반복한다. Snprintf를 통해서 입력받은 값을 옮겨 담고, write 함수를 통해서 pipe 파일에 메시지를 쓴다. 이 때 사용자로부터 'exit' 라는 입력이 들어오면 프로그램을 종료한다. Reader는 read 함수를 통해 Pipe 파일로부터 메시지를 읽어와서 msg[]에 담는다. 이후 이 메시지를 화면으로 출력한다. 이 때 읽어 온 메시지가 'exit'이면 프로그램을 정상 종료한다.

2. IPC 메커니즘 설명

Pipe 이름을 알면 다른 프로세스에서도 접근이 가능하기 때문에 named pipe를 사용한다. FIFO라는 특수 파일을 사용하여 pipe 역할을 대신한다. Reader와 writer는 pipe를 통해 파일을 읽고 쓴다.

3. 컴파일 방법 설명

```
1 all: writer_Aprocess reader_Bprocess
2
3 CC = gcc
4
5 writer_Aprocess: writer_Aprocess.c
6         $(CC) -o $@ $<
7
8 reader_Bproess: reader_Bprocess.c
9         $(CC) -o $@ $<
~
```

컴파일은 makefile을 통해 진행했고, 위와 같다.

- **Message Queue**

4. 프로그램 설명

reader_Aprocess.c (~/OS-2021/HW3/messagequeue_version) ...

파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/stat.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define MSG_SIZE 100

//contain msg & msg type
struct msgbuf{
    long msgtype;
    char mtext[MSG_SIZE];
};

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

    //get the msg object which key is 1234
    key_id = msgget((key_t)1234, IPC_CREAT|0666);

    if(key_id == -1){
        perror("msgget error :");
        return 0;
    }
    while(1){
        //receive msg
        if(msgrcv(key_id, (void *)&rsvbuf, sizeof(struct msgbuf), ms
gtype, 0) == -1){
            perror("msgrcv error :");
        }else{
            //received msg is 'exit' program quit
            if(strcmp(rsvbuf.mtext, "exit") == 0){
                printf("-exit this program-\n");
                return 0;
            }
            printf("The received message is: %s\n", rsvbuf.mtext);
        }
    }
    return 0;
}
```

writer_Bprocess.c (~/OS-2021/HW3/messagequeue_version) - ...

파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/stat.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define MSG_SIZE 80

//contain msg & msg type
struct msgbuf{
    long msgtype;
    char mtext[MSG_SIZE];
};

int main(void){
    key_t key_id;
    struct msgbuf sndbuf;
    char input[MSG_SIZE];

    //get msg object which key is 1234
    key_id = msgget((key_t)1234, IPC_CREAT|0666);

    if(key_id == -1){
        perror("msgget error :");
        return 0;
    }
    while(1){
        //send msg from keyboard about 1 line
        fgets(input, sizeof(input), stdin);
        input[strlen(input) - 1] = '\0';
        //define msg type
        sndbuf.msgtype = 3;
        strcpy(sndbuf.mtext, input);
        if(msgsnd(key_id, (void *)&sndbuf, sizeof(struct msgbuf), 0)
== -1){
            perror("msgsnd error :");
        }
        //if entered msg is 'exit', program quit
        if(strcmp(sndbuf.mtext, "exit") == 0){
            printf("-exit this program-\n");
            break;
        }
    }

    return 0;
}
```

우선 메시지 큐에 담을 메시지를 구조체를 통해 정의했다. 메시지를 저장하는 char type 변수와, 메시지의 타입을 저장할 long type의 변수 두 개를 가지고 있다. 이 구조체는 reader와 writer 둘 다 가지고 있어야 한다.

우선 reader에 대해 설명해보겠다. Reader는 메시지 구조체를 선언하고, 받을 메

시지의 타입은 3이라고 정한다. 그 후 msgget 함수를 통해 key값이 1234인 메시지 큐 객체를 생성 또는 참조한다. 이후 while문을 통해 메시지 큐에서 메시지를 계속 받는다. 이 때 msgrcv함수를 통해 메시지를 받는다. 이 함수는 메시지 큐로부터 메시지를 가져올 때 사용되며, 메시지 큐의 key값, 메시지를 담을 구조체(msgbuf)주소, 구조체 크기, 수신할 메시지 타입(==3), msgflg를 인자로 받는다.

받은 메시지 구조체의 메시지를 msgbuf.mtext를 통해 확인하며 이 때 전달 받은 메시지가 'exit'일 경우 프로그램을 정상종료하고, 아닐 경우 화면에 띄운다.

다음으로 writer에 대해 설명해보면, 마찬가지로 메시지 구조체를 선언해주고, 사용자로부터 입력 받은 메시지를 담을 char input을 선언해준다. 이후 msgget 함수를 통해 key가 1234인 메시지 큐 객체를 생성 또는 참조한다. 이후 while문을 통해 사용자로부터 메시지를 계속 입력 받고 메시지 큐에 쓰는 과정을 반복한다. 이 때 사용자로부터 ₩n가 입력되기 전까지 메시지를 입력받으며, 입력 받은 값을 메시지 구조체의 mtext에 담고, 메시지 타입은 3으로 정의하고 msgsnd 함수를 통해 메시지 큐에 메시지를 투입한다. 사용자로부터 입력받은 값이 'exit'일 경우 프로그램을 정상종료한다.

5. IPC 메커니즘 설명

리눅스의 message queue 자료구조는 커널에서 관리하며 key를 알면 어떤 프로세스에서도 접근 가능하다. 큐 자료구조임에도 불구하고 중간에 있는 데이터를 꺼낼 수 있는 장점이 있다. 이 프로그램에서는 중간에 있는 데이터를 빼진 않았다. 메시지 타입을 지정하고 해당 타입에 해당하는 메시지를 빼오도록 구성하였다. Reader와 writer 모두 똑같은 타입의 메시지를 쓰고 읽도록 했다.

6. 컴파일 방법 설명

Make로 컴파일, Makefile은 위의 파일과 동일하다.

- **Shared Memory**

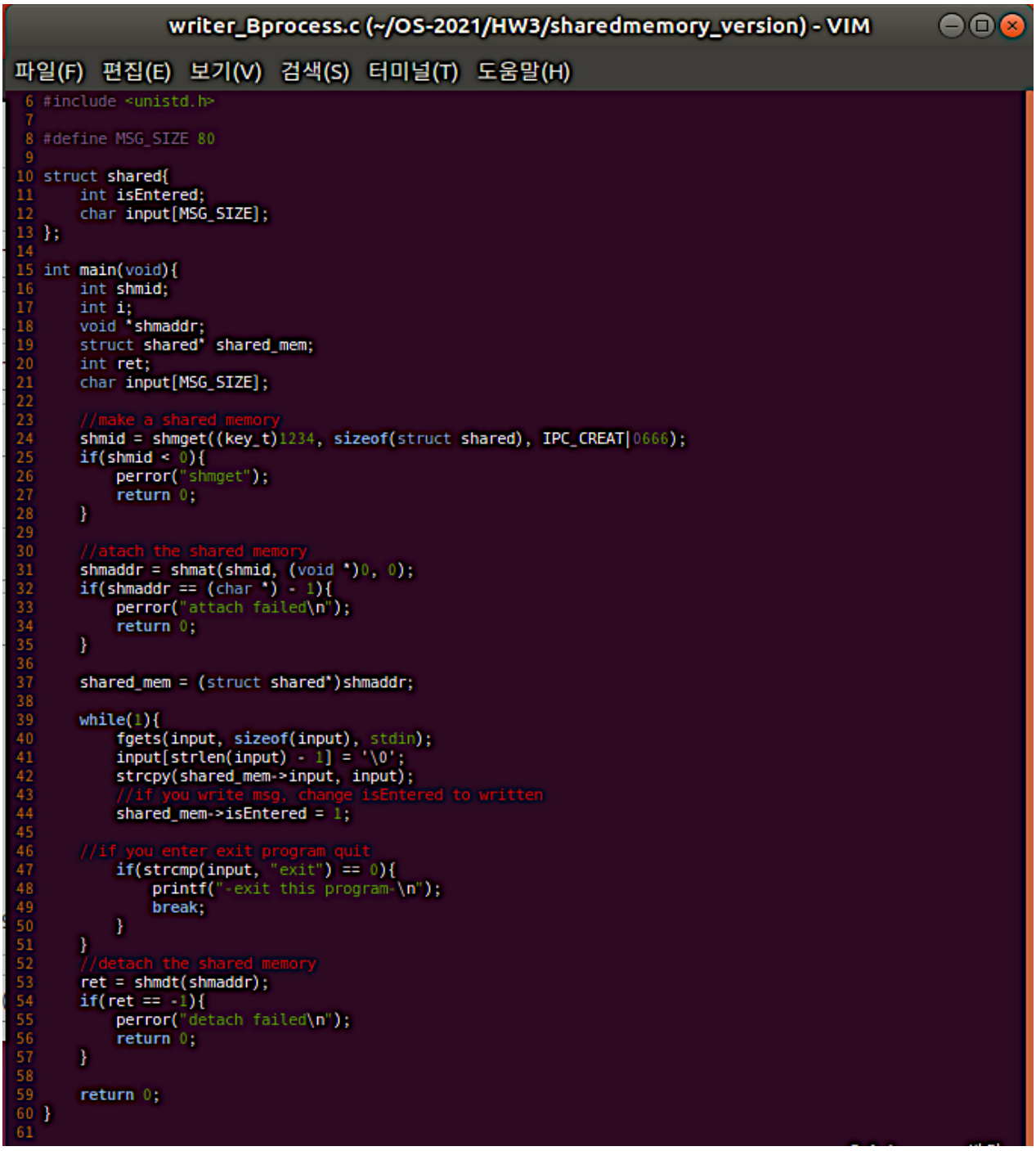
7. 프로그램 설명

reader_Aprocess.c + (~/OS-2021/HW3/sharedmemory_version) - VIM

파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define MSG_SIZE 80

struct shared{
    int isEntered;
    char input[MSG_SIZE];
};

int main(void){
    int shmid;
    void *shmaddr;
    struct shared* shared_mem;
    int ret;
    //get shared memory id
    shmid = shmget((key_t)1234, sizeof(struct shared), IPC_CREAT|0666);
    if(shmid == -1){
        perror("shared memory access is failed\n");
        return 0;
        }

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

    //check msg is written
    shared_mem->isEntered = 0;

    while(1){
    //if msg is written, print msg
        if(shared_mem->isEntered){
    //if received msg is exit program quit
            if(strcmp(shared_mem->input, "exit") == 0){
                printf("-exit this program-\n");
                break;
            }
            printf("data read from shared memory: %s\n", shared_mem->input);
            shared_mem->isEntered = 0;
        }
    }
    //detach the shared memory
    ret = shmdt(shmaddr);
    if(ret == -1){
        perror("detach failed\n");
        return 0;
    }

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

-- 끼워넣기 -- 20,10-13 모두




```c
#include <unistd.h>

#define MSG_SIZE 80

struct shared{
    int isEntered;
    char input[MSG_SIZE];
};

int main(void){
    int shmid;
    int i;
    void *shmaddr;
    struct shared* shared_mem;
    int ret;
    char input[MSG_SIZE];

    //make a shared memory
    shmid = shmget((key_t)1234, sizeof(struct shared), IPC_CREAT|0666);
    if(shmid < 0){
        perror("shmget");
        return 0;
    }

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

    shared_mem = (struct shared*)shmaddr;

    while(1){
        fgets(input, sizeof(input), stdin);
        input[strlen(input) - 1] = '\0';
        strcpy(shared_mem->input, input);
        //if you write msg, change isEntered to written
        shared_mem->isEntered = 1;

    //if you enter exit program quit
        if(strcmp(input, "exit") == 0){
            printf("-exit this program-\n");
            break;
        }
    }
    //detach the shared memory
    ret = shmdt(shmaddr);
    if(ret == -1){
        perror("detach failed\n");
        return 0;
    }

    return 0;
}
```

우선 공통적인 부분에 대해서 설명을 하자면, reader와 writer 모두 shared라는 구조체를 가진다. 이 구조체에 메시지를 담고, 메시지가 입력됐는지 여부를 기록한다. 메시지를 읽었는지는 isEntered 변수를 통해 확인한다.

우선 reader에 대해서 설명해보겠다. 처음으로 공유메모리를 가리킬 void pointer를 선언해주고, 메시지에 대한 구조체를 선언해준다. 이후 shmget 함수를 통해 key 값이 1234인 shared memory의 id를 얻어온다. 이 때 shared memory가 없으면 생성하

도록 한다. 다음으로 shmat 함수를 통해 프로세스의 메모리 공간에 공유 메모리를 붙인다. Shdaddr이 이 메모리의 주소를 가리키도록 했다. 다음으로 shared_mem 구조체에 shdaddr을 타입 변환을 통해 담도록 한다. 우리는 이 메모리에 구조체를 담고, 읽고 쓰는 과정을 반복하도록 할 것이다. 구조체의 isEntered 변수의 값을 0으로 바꿔주어 아직 메시지를 읽지 않은 상태임을 알려준다. 이후 while문을 통해서 메시지를 받아오는데 inEntered의 값이 1일 경우(메시지가 입력됐을 경우)에만 구조체에서 메시지를 읽어와서 화면에 출력하도록 한다. 메시지를 읽고 난 후에는 isEntered 값을 0으로 바꾸어 메시지를 읽었음을 알린다. 이때 읽어온 메시지가 'exit'이면 while 루프에서 탈출한다. 탈출 이후엔 붙여놓았던 메모리를 shmdt를 통해 떼어내고 프로그램을 종료한다.

다음을 writer에 대해서 설명해보겠다. Writer도 마찬가지로 처음으로 공유메모리를 가리킬 void pointer를 선언해주고, 메시지에 대한 구조체를 선언해준다. 이후 shmget 함수를 통해 key 값이 1234인 shared memory의 id를 얻어온다. 그리고 사용자로부터 메시지를 입력받은 값을 저장할 input[]을 선언한다. 이후의 과정도 reader와 동일하게 shmat 함수를 통해 프로세스의 메모리 공간에 공유 메모리를 붙인다. Shdaddr이 이 메모리의 주소를 가리키도록 하고, shared_mem 구조체에 shdaddr를 타입 변환을 통해 담도록 한다. 이후 while루프를 통해 사용자로부터 값을 입력 받고, 메시지 구조체에 메시지를 담는다. 메시지를 담은 이후에는 구조체의 isEntered 값을 1로 바꾸어 메시지가 입력되었음을 알린다. 이때 입력된 메시지가 'exit'이면 루프에서 탈출하고, shmdt를 통해 붙여놨던 메모리를 떼어내고 프로그램을 정상 종료한다.

8. IPC 메커니즘 설명

Shared memory란 공유 메모리고, 여러 프로세스가 공동으로 사용하는 메모리를 의미한다. 여러 프로세스가 공유 메모리에 attach해서 메모리를 공유한다. 공유 과정이 끝나면 dettach를 통해서 메모리에서 떼어낸다. 원래 여러 프로세스가 이렇게 공동으로 이용할 때는 lock과 같은 복잡한 과정을 통해서 충돌이 일어나지 않도록 해주어야하는데, 우리의 경우에는 두 프로세스만 가지고 하나는 읽고 하나는 쓰는 과정을 하기 때문에 lock의 경우는 복잡하게 따지지 않고, 단지 읽을 메시지가 있으면 읽도록 프로그램을 구성했다.

9. 컴파일 방법 설명

컴파일은 처음과 동일하게 make를 통해 진행했고, Makefile도 동일하다.

B. 과제 B

1. 멀티스레딩 내용

스레드는 실행의 한 단위고, 적은 리소스를 필요로 하므로 가벼운 프로세스로 불린다. 프로세스는 address space와 processor context를 가지고 있지만 스레드는 독자적은 stack space와 processor context를 가지며, 프로세스와 다른 자원들을 공유한다. 스레드는 running, ready, waiting의 실행 상태를 가지고, context switching을 필요로 한다. 스레드는 자원을 공유하는데, 한 스레드가 전역 변수를 바꾸면, 다른 스레드도 바뀐 것을 알아볼 수 있고, 한 스레드가 연 파일은 다른 스레드에서도 접근가능하다. 스택에 저장된 변수는 스레드의 private data이다. Thread-local 변수는 각 개인 스레드에서만 공유 가능하다.

스레드의 장점

1. 특정 부분이 block되도 프로그램이 계속 진행될 수 있다.
2. 자원과 메모리 공유에 대한 이점이 존재한다.
3. Context switch와 creation에서 경제적이다.
4. Concurrency를 높일 수 있다.

User threads & kernel threads

1. User threads를 이용하면, 커널의 도움없이 User-level에서 thread library를 통해 관리가 가능하고, 빠르게 만들고 관리할 수 있다는 이점이 있다. 하지만 커널이 single threaded일 경우, user-level thread가 시스템 콜을 blocking할 경우 전체 프로세스가 block될 수 있다.
2. Kernel thread는 커널에 의해 직접 관리되며, 커널이 kernel space에서 스레드의 생성, 스케줄링, 관리를 담당한다. 이 때문에 user thread보다 생성과 관리가 느리다.

Multithreading Models

1. Many-to-one; 많은 user-level 스레드가 single kernel thread에 매핑됨. 커널 스레드의 도움이 필요 없는 시스템에서 사용된다.
2. One-to-one; 각각의 user-level 스레드가 kernel thread와 매핑됨

3. Many-to-many; 많은 수의 user-level 스레드가 많은 수의 kernel thread와 매핑됨. OS가 충분한 수의 kernel thread를 생성할 수 있다.