2018008331 컴퓨터소프트웨어학부 박민경

운영 체제 HW#3

제출 일자 : 2020/04/10

A. 과제 A

- Named Pipe

1. 프로그램 설명

```c
#define PIPENAME "./named_pipe_file"

void handler(int sig)
{
    if (sig==SIGINT)
    {
        printf("Program exit\n");
        exit(0);
    }
}

int main()
{
    signal(SIGINT, handler);

    int ret;
    char msg[512];
    int fd;
    pid_t pid;

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

    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", msg);
    }
    return 0;
```

<reader_Aprocess.c>

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

void handler(int sig)
{
    if (sig==SIGINT)
    {
        printf("Program Exit\n");
        exit(0);
    }
}

int main(void)
{
    signal(SIGINT, handler);

    char msg[MSG_SIZE];
    int fd;
    int ret, i;

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

    printf("Hello, this is A process. I'll give the data to B.\n");

    while(1)
    {
        fgets(msg, sizeof(msg), stdin);
        ret = write(fd, msg, sizeof(msg));
        if (ret<0)
        {
            printf("Write failed\n");
            return 0;
        }
    }
    return 0;
}
```

<writer_Bprocess.c>

- 자료구조

  Linux의 Named Pipe IPC 자료구조이며, Pipe 이름을 알 경우 다른 프로세스에서도 접근이 가능하다. FIFO라는 특수 파일을 사용하여 Pipe 역할을 대신한다.

- 함수

  mkfifo 함수를 통해 FIFO라는 특수 파일을 제공한다. FIFO 파일은 Named Pipe의 역할을 하는 파일로, 해당 파일에 읽고 쓰기를 통해 데이터를 주고받는다. 이 프로그램에서는 파일 이름을 [named_pipe_file]로 지정하였다.

  unlink 함수는 해당 파일을 참조하는 count를 1 감소시키고, count 값이 0일 경우 파일을 삭제한다. 이 프로그램에서는 사용하고자 하는 Named Pipe가 존재하는지 조사하고, 존재한다면 pipe를 unlink할 때 사용된다.

- 프로그램 구조

  이 프로그램은 IPC 메커니즘을 사용하여 Writer인 A 프로세스와 Reader인 B 프로세스 간에 통신한다.

  Reader가 [named_pipe_file] 라는 Named Pipe가 존재하는 지 조사한 후 존재한다면 pipe를 unlink한다. 그리고 해당 이름을 가진 Named pipe를 생성하고, 읽고 쓰기가 가능하도록 Pipe 파일을 open한다. 그리고 무한 loop을 돌며 Writer가 write한 메시지를 read하여 print한다.

  Writer는 Reader가 만들어 놓은 Pipe를 쓰기 전용으로 open하고, 마찬가지로 무한 loop을 돌며 사용자에게 엔터로 구분되는 입력을 받고, 그것을 write한다.

  프로그램 종료는 SIGINT(ctrl+c)로 각각 처리된다.

2. IPC 메커니즘 설명

mkfifo 함수를 통해 Named Pipe를 생성하고, 그것을 통해 프로세스 간 통신이 이루어진다. Pipe 역할을 하는 파일에 한쪽에서 쓰면, 다른 한쪽에서 읽음으로써 데이터를 주고받는 단방향 통신 구조이다.

3. 컴파일 방법 설명

Reader_Aprocess.c, Writer_Bprocess.c, Makefile 파일을 하나의 경로에 위치시킨다. make 명령어를 입력하면 실행파일이 생성되고, ./Reader_Aprocess.c 명령어로 reader 먼저 실행한 후, ./Writer_Bprocess.c 명령어로 writer을 실행하여 data를 입력한다.

- **Message Queue**

4. 프로그램 설명

```c
void handler(int sig)
{
    if (sig==SIGINT)
    {
        printf("program exit\n");
    }
    exit(0);
}

struct msgbuf
{
    long msgtype;
    char mtext[20];
};

int main(int argc, char **argv)
{
    signal(SIGINT, handler);

    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", i, rsvbuf.mtext);
            i++;
        }
    }

    return 0;
}
```

<reader_Aprocess.c>

```c
void handler(int sig)
{
    if (sig==SIGINT)
    {
        printf("program exit\n");
        exit(0);
    }
}

struct msgbuf
{
    long msgtype;
    char mtext[20];
};

int main(void)
{
    signal(SIGINT, handler);

    key_t key_id;
    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(sndbuf.mtext, sizeof(sndbuf.mtext), stdin);

        if (msgsnd(key_id, (void *)&sndbuf, sizeof(struct msgbuf), IPC_NOWAIT) =
=-1)
        {
            perror("msgsnd error : ");
        }
        else{
            printf("Sending %dth message is succeed\n", i);
            i++;
        }
    }
    return 0;
}
```

<writer_Bprocess.c>

- 자료구조

  Linux의 Message Queue 자료구조이며, 커널에서 관리하며 Message Queue의 key를 알면 어떤 프로세스에서도 접근이 가능하다. queue 자료구조임에도 불구하고 중간에 있는 데이터를 꺼낼 수 있고, Message Queue에 투입할 데이터에 번호를 붙임으로써 큐 안에 있는 데이터들을 동시에 접근 가능하게 한다.

  Message Queue에 담을 메시지는 사용자가 구조체를 사용하여 정의하는데, 이 프로그램에서는 msgbuf 라는 이름에 msgtype, mtext[20] 라는 변수를 가진 구조체를 사용하였다.

- 함수

  msgget 함수는 Message Queue를 생성하거나, 생성된 객체 참조에 사용된다. 첫번째 인자로는 Message Queue 객체를 식별할 수 있는 고유번호, 두번째 인자로는 Message Queue 옵션을 가지는데, 이 프로그램에서는 고유번호를 "1234"로 지정하고 옵션을 IPC_CREAT 옵션에 모드 값을 or 연산하여 지정하였다.

  msgsnd 함수는 Message Queue에 메시지를 투입할 때 사용된다. 투입하고자 하는 Message Queue의 id, 메시지 구조체 주소, 메시지 구조체의 크기, 동작 옵션을 차례대로 인자로 가진다. 이 프로그램에서는 IPC_NOWAIT를 동작 옵션으로 사용하였는데, 큐에 여유 공간이 없다면 바로 -1을 반환하는 역할을 한다.

  msgrcv 함수는 Message Queue로부터 메시지를 가져올 때 사용된다. 메시지를 가져올 Message Queue id, 메시지를 담을 구조체 주소, 크기, 수신할 메시지 타입, 동작 옵션을 차례대로 인자로 가진다. 이 프로그램에서는 옵션을 0으로 지정했다.

- 프로그램 구조

  이 프로그램은 IPC 메커니즘을 사용하여 Writer인 A 프로세스와 Reader인 B 프로세스 간에 통신한다.

  Reader가 msgget 함수의 첫번째 인자로 key값(1234)를 사용하여 Message queue의 id를 return 받아 key_id에 저장한다. 무한 loop를 돌며 메시지를 성공적으로 queue로부터 가져왔을 경우 출력한다.

  Writer는 msgget 함수의 첫번째 인자로 key값(1234)를 사용하여 Message queue의 id를 return 받아 key_id에 저장한다. 무한 loop를 돌며 사용자로부터 엔터로 구분되는 입력을 받아 sndbuf.mtext에 저장하고, 그것을 성공적으로 queue에 투입했을 경우 해당 메시지를 출력한다.

  프로그램 종료는 SIGINT(ctrl+c)로 각각 처리된다.

5. IPC 메커니즘 설명

프로세스 간의 데이터 송수신을 위해 각 데이터를 메시지 형태로 생성, 전달하고 수신이 가능하도록 queue 구조를 이용한다. Message Queue는 커널에서 관리되

고, 모든 프로세스에서 접근 가능하기 때문에, 하나의 Message Queue가 작성되면 해당 식별자를 아는 모든 프로세스가 그 Message Queue에 접근하여 메시지를 공유할 수 있는 구조이다.

6. 컴파일 방법 설명

Reader_Aprocess.c, Writer_Bprocess.c, Makefile 파일을 하나의 경로에 위치시킨다. make 명령어를 입력하면 실행파일이 생성되고, ./Writer_Bprocess.c 명령어로 writer 먼저 실행한 후, ./Reader_Aprocess.c 명령어로 reader을 실행하고 writer쪽에서 data를 입력한다.

- **Shared Memory**

7. 프로그램 설명

```c
#define READ_FLAG '0'
#define WRITE_FLAG '1'

int ret;
char *shmaddr;
int shmid;

void handler(int sig)
{
    if (sig==SIGINT)
    {
        ret = shmdt(shmaddr);
        if (ret==-1)
        {
            perror("detach failed\n");
            exit(0);
        }
        ret = shmctl(shmid, IPC_RMID, 0);
        if (ret ==-1)
        {
            perror("remove failed\n");
            exit(0);
        }
        printf("Program exit\n");
        exit(0);
    }
}

int main(void)
{
    signal(SIGINT, handler);
    char *msg;

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

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

```
    msg = shmaddr+1;

    while(1)
    {

        if (shmaddr[0]==READ_FLAG)
        {
            printf("data read form shared memory: %s", msg);
            shmaddr[0]=WRITE_FLAG;
        }
    }

    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;
}
```

<reader_Aprocess.c>

```
#define READ_FLAG '0'
#define WRITE_FLAG '1'

int ret;
char* shmaddr;

void handler(int sig)
{
    if (sig==SIGINT)
    {
        ret = shmdt(shmaddr);
        if (ret==-1)
        {
            perror("detach failed\n");
            exit(0);
        }
        printf("Program exit\n");
        exit(0);
    }
}

int main(void)
{
    signal(SIGINT, handler);

    int shmid;
    int i;
    char *msg;
    char tmp[50];

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

    shmaddr = shmat(shmid, NULL, 0);

    if (shmaddr == (char *)-1)
    {
        perror("attch failed\n");
        return 0;
    }
    shmaddr[0] = WRITE_FLAG;
    msg = shmaddr+1;
```

```
    while (1)
    {
        if (shmaddr[0]==WRITE_FLAG)
        {
            fgets(tmp, sizeof(tmp), stdin);
            strcpy(msg, (char *)tmp);
            shmaddr[0] = READ_FLAG;
        }
    }

    return 0;
}
```

<writer_Bprocess.c>

- 자료구조

  Linux의 Shared Memory 자료구조이다. 일반적으로 메모리공간은 메모리를 요청한 프로세스만이 접근가능하도록 되어있기 때문에, 여러 개의 프로세스가 특정 메모리 공간을 동시에 접근하기 위해 사용된다.

- 함수

  shmget 함수는 key 값을 통해 shared memory의 id를 얻어오기 위해 사용된다. shared memory 객체의 고유번호, 가져올 shared memory의 크기, 사용할 flag를 차례대로 인자로 가지며, 이 프로그램에서는 key를 1234, size는 1024, flag는 IPC_CREAT 옵션에 or 값을 연산하여 지정하였다.

  shmat 함수는 프로세스의 메모리 공간에 공유 메모리를 붙이는 함수이다. 아직 메모리 공간에 붙여지지 않은 shared memory의 id, 공유메모리 주소, 동작 옵션을 차례대로 인자로 갖는다. 이 프로그램에서는 shmget의 return 값을 첫번째 인자로 넘기고, NULL 값을 두번째 인자로 넘김으로써 반환값으로 나오는 공유 메모리 주소를 dynamic하게 설정하였다.

  shmdt 함수는 프로세스의 메모리 공간에서 공유 메모리를 떼는 함수로, 공유메모리 주소를 인자로 갖는다.

  shmctl 함수는 shared memory 정보를 확인/변경/제거하는 함수로, 정보를 확인/변경/제거하려는 shared memory의 id, 제어 명령, 공유 메모리 정보를 가져오기 위한 구조체를 차례대로 인자로 갖는다. 이 프로그램에서는 shared memory를 제거하기 위한 IPC_RMID 명령을 두번째 인자로, 구조체 정보를 가져올 필요가 없기 때문에 0을 세번째 인자로 넘겼다.

- 프로그램 구조

이 프로그램은 IPC 메커니즘을 사용하여 Writer인 A 프로세스와 Reader인 B 프로세스 간에 통신한다.

우선 같은 data를 불필요하게 읽는 것을 방지하기 위해서, FLAG를 설정해주었다. FLAG를 프로세스 간에 공유해야하므로 Shared Memory의 첫번째 주소에 FLAG를 식별하기 위한 공간을 마련하고, 실제 data가 저장될 위치는 msg라는 새로운 변수에 shmaddr+1 값으로 할당하였다. 그리고 READ_FLAG와 WRITE_FLAG를 통해 현재 새로운 값이 write 되었는지, 그 값이 read 되었는지를 체크하여 writer가 data를 쓸 때만 reader가 읽어 가서 출력하도록 한다.

Reader가 shmget 함수를 통해 shmid에 Shared Memory의 id를 저장한다. 그리고 shmat 함수를 통해 프로세스의 메모리 공간에 공유 메모리를 붙이고, return된 공유 메모리 주소를 shmaddr에 저장한다. 무한 loop를 돌며 FLAG가 READ_FLAG일 경우, 즉 현재 read 할 차례일 경우 해당 data와 메시지를 출력하고 FLAG를 WRITE_FLAG로 변경한다.

Writer는 shmget 함수를 통해 shmid에 Shared Memory의 id를 저장한다. 그리고 shmat 함수를 통해 프로세스의 메모리 공간에 공유 메모리를 붙이고, return된 공유 메모리 주소를 shmaddr에 저장한다. 여기까지 Reader와 동일하다. Writer가 먼저 실행될 것이기 때문에, FLAG를 WRITE_FLAG로 초기화한다. 그리고 무한 loop를 돌며 사용자에게 엔터로 구분되는 입력을 받고 tmp에 저장한 후, char* 로 형변환을 하여 데이터를 Shared Memory 영역인 msg에 복사한다. 그리고 FLAG를 READ_FLAG로 변경한다.

프로그램 종료는 SIGINT(ctrl+c)로 각각 처리된다.

8. IPC 메커니즘 설명

Shared Memory는 서로 다른 프로세스가 동시에 접근할 수 있는 메모리를 만든다. 같은 Shared Memory key 값을 통해 id를 얻어오고, 그것을 프로세스 메모리 공간에 attach함으로써 사용 가능하다. 하나의 프로세스가 공유 메모리에 data를 쓰면, 다른 프로세스가 그 메모리 영역에 접근함으로써 읽을 수 있는 방식이다.

9. 컴파일 방법 설명

Reader_Aprocess.c, Writer_Bprocess.c, Makefile 파일을 하나의 경로에 위치시킨다. make 명령어를 입력하면 실행파일이 생성되고, ./Writer_Bprocess.c 명령어로 writer 먼저 실행한 후, ./Reader_Aprocess.c 명령어로 reader을 실행하고 writer쪽

에서 data를 입력한다.

B. 과제 B

1. 멀티스레딩 내용

Multi-thread

: 하나의 프로세스를 다수의 실행 단위로 구분하여 자원 공유, 수행능력 향상

- Multi-thread vs Multi-process

Multi-thread: data, heap, stack 중 stack만 비공유

Multi-process: data, heap, stack 모두 비공유

- thread 구현 방법

1. User Threads (user-level)

: user-level 에서 thread 만들어 실행 (thread_create API)
장점: Kernel Thread 에 비해 빠름 (왜? thread 를 create 하고 scheduling 할 때 kernel 의 갭이 필요하지 X, 모두 user-level 에서 발생)

단점: user-mode 에서는 1 개의 thread 라도 waiting 으로 가면, 나머지 thread 들 모두가 scheduling 대상에서 제외됨. user-level thread 안에 여러개가 있더라도 kernel 은 그것을 single process 로만 보기 때문에.
ex) Pthreads, C-threads

2. Kernel Threads (kernel-level)

: kernel 을 create 하고 scheduling 하는 모든 행위가 kernel 에서 이루어짐
ex) windows, solaris, linux

- Multithreading Models

1. Many-to-One: user-level-> 여러 thread, kernel->단일 threaded라고 인식
2. One-to-One: kernel에서 만드는 만큼 user에게 보임
3. Many-to-Many: kernel-thread-> virtual CPU