운영체제 Report 3: 프로세스 생성

2019312014 박병준

2023-2 학기 류은석 교수님 강의

과제: Shared Memory 와 Message Passing 방식을 사용한 IPC 코드를 작성한 프로그램을 구현

IPC(Inter-Process Communication)이란 프로세스 간 통신을 의미한다. 다른 프로세스와 영향을 주고받고 데이터를 공유하는 협력적 프로세스들은 데이터를 교환하기 위해 IPC 기법을 필요로 한다. IPC 기법에는 Shared Memory 와 Message Passing 의 두 가지 모델이 있다. Shared Memory 방식을 이용하면 프로세스 A 와 B 사이에 공유되는 메모리 영역을 통해 정보를 교환할 수 있다. Message Passing 방식을 이용하면 커널의 메시지 큐를 통해 프로세스 간 정보 교환이 가능하다. (사진) IPC 의 장점으로는 정보 공유, 병렬 처리로 인한 계산 가속화, 모듈화 등이 있다. Shared Memory 방식은 공유 메모리 영역을 구축할 때만 시스템 콜이 필요할 뿐, 통상 시스템 콜에 의한 오버헤드가 발생하지 않아 Message Passing 방식보다 빠르다. 일단 공유 메모리 영역이 구축되고 나면 모든 접근은 일반적인 메모리 접근으로 취급되어 커널의 도움을 받지 않아도 된다. 반면 Message Passing 방식은 Shared Memory 와 달리 충돌의 위험을 회피하지 않아도 되므로 적은 양의 데이터를 교환하는 데 유용하고, 분산 시스템에서 Shared Memory 보다 구현하기 쉽다. 일반적으로 시스템의 코어 수가 늘어날수록 캐시 일관성 문제로 인해 Shared Memory 보다 Message Passing 이 더 나은 성능을 보인다는 연구 결과가 있다.

Shared Memory 에서, 공유 메모리 영역을 이용해 통신하고자 하는 프로세스들은 공유 메모리 segment 를 자신의 주소 공간에 추가해야 한다. 일반적으로 운영체제는 한 프로세스가 다른 프로세스의 메모리에 접근하는 것을 금지하기 때문에, Shared Memory 에서는 프로세스들이 이 제약 조건의 제거를 따르도록 한다. 프로세스들이 동시에 동일한 위치에 쓰지 않도록 관리하는

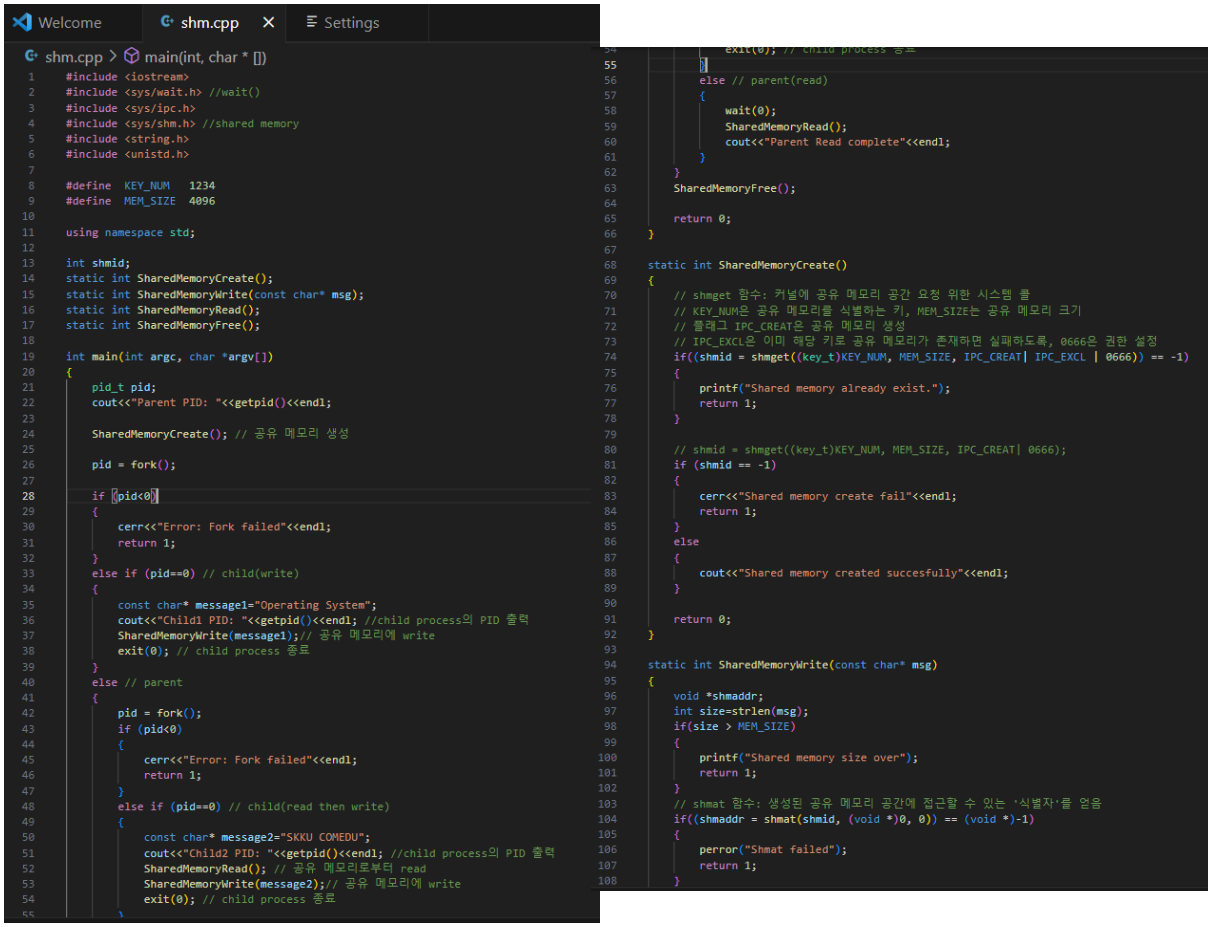
것이 필요하다. Shared Memory 에 참여하는 프로세스를 생산자 프로세스와 소비자 프로세스로 나눌 수 있다. 생산자가 정보를 채워 넣고 소비자가 소모하려면 공유 메모리 영역의 buffer 가 반드시 사용 가능해야 한다.

Message Passing 방식에선 동일한 주소 공간을 공유하지 않고도 프로세스들이 통신을 할 수 있다. Message Passing 방식은 기본적으로 send 와 receive 두 가지 연산을 제공한다. 프로세스 P 와 Q 의 통신을 위해서는 communication link(통신 연결)가 설정되어야 한다. 이를 구현하기 위해 1 직접 또는 간접 통신을 사용하거나, 2 동기식 또는 비동기식 통신을 사용하거나, 3 자동 또는 명시적 버퍼링을 사용할 수 있다. 1 직접 통신의 경우, 각 프로세스는 통신의 수신자 또는 송신자의 이름을 명시해야 한다. 프로세스 식별자를 바꾸면 이와 관련된 모든 부분을 바꿔야 하는 단점이 있다. 간접 통신은 직접 통신과 달리, 메시지들이 메일박스나 포트로 송수신된다. 프로세스들이 한 메일박스를 공유할 때, 메시지를 수신할 수 있는 프로세스가 여럿이면 라운드 로빈 등 설계한 알고리즘에 따라 누가 메시지를 수신할 지 정할 수 있다. 메일박스는 한 프로세스 또는 운영체제에 의해 소유될 수 있다. 2 동기식 송수신의 경우, 메시지 송수신이 완료될 때까지 작업이 봉쇄되어 프로세스가 기다리게 되고, 비동기식 송수신의 경우, 메시지 송수신 시 봉쇄되지 않는다. 3 교환되는 메시지는 임시 큐에 존재하게 되는데, 이러한 큐의 구현 방식은 무용량, 유한 용량, 무한 용량의 3 가지 방식으로 나뉜다. 무용량의 경우, 링크는 대기하는 메시지들을 가질 수 없어 수신자가 메시지를 수신할 때까지 송신자는 기다려야 한다. 유한 용량의 경우, 유한한 길이의 큐에 메시지가 존재할 수 있으며, 새로운 메시지가 전송될 때 큐가 full 이 아니라면 메시지는 큐에 놓여 송신자는 기다리지 않고 계속 송신할 수 있다. 대신 큐가 full 이면 송신자는 기다려야 하는데, 무한 용량의 경우 송신자는 기다리지 않아도 된다.

1. Shared Memory 방식 C++ 코드

https://coding-chobo.tistory.com/16

이 웹사이트의 코드를 기반으로 Shared Memory 방식을 구성했다. 코드에 많은 변형을 가하고자 하였다. 코드를 두 개로 쪼개어 2 개의 프로세스에서 동작하도록 한 기존 코드와 달리, 지난 시간 배웠던 부모-자식 프로세스를 코드에 활용하여 하나의 코드 안에 프로세스들이 동작하도록 했다. 그리고 공유 메모리에 값을 써주는 작업을 1 의 값을 써주는 작업에서 const char 형 메시지를 쓰는 방식으로 변형하였다. 코드의 흐름은 다음과 같다. 공유 메모리를 생성하고, 부모 프로세스가 자식 프로세스 1 을 생성한다. 생성된 자식 프로세스 1 은 공유 메모리에 메시지를 write 하고 종료한다. 이후 부모 프로세스는 자식 프로세스 2 를 만들고, 생성된 자식 프로세스 2 는 공유 메모리에서 read 한 후 새로운 값을 write 하고 종료한다. 부모 프로세스는 자식 프로세스의 종료를 기다렸다가 공유 메모리에서 값을 read 한다. 최종적으로 SharedMemoryFree() 함수를 통해 공유메모리가 삭제된다.



```cpp
#include <iostream>
#include <sys/wait.h> //wait()
#include <sys/ipc.h>
#include <sys/shm.h> //shared memory
#include <string.h>
#include <unistd.h>

#define  KEY_NUM   1234
#define  MEM_SIZE  4096

using namespace std;

int shmid;
static int SharedMemoryCreate();
static int SharedMemoryWrite(const char* msg);
static int SharedMemoryRead();
static int SharedMemoryFree();

int main(int argc, char *argv[])
{
    pid_t pid;
    cout<<"Parent PID: "<<getpid()<<endl;

    SharedMemoryCreate(); // 공유 메모리 생성

    pid = fork();

    if (pid<0)
    {
        cerr<<"Error: Fork failed"<<endl;
        return 1;
    }
    else if (pid==0) // child(write)
    {
        const char* message1="Operating System";
        cout<<"Child1 PID: "<<getpid()<<endl; //child process의 PID 출력
        SharedMemoryWrite(message1);// 공유 메모리에 write
        exit(0); // child process 종료
    }
    else // parent
    {
        pid = fork();
        if (pid<0)
        {
            cerr<<"Error: Fork failed"<<endl;
            return 1;
        }
        else if (pid==0) // child(read then write)
        {
            const char* message2="SKKU COMEDU";
            cout<<"Child2 PID: "<<getpid()<<endl; //child process의 PID 출력
            SharedMemoryRead(); // 공유 메모리로부터 read
            SharedMemoryWrite(message2);// 공유 메모리에 write
            exit(0); // child process 종료
        }
        else // parent(read)
        {
            wait(0);
            SharedMemoryRead();
            cout<<"Parent Read complete"<<endl;
        }
    }
    SharedMemoryFree();

    return 0;
}

static int SharedMemoryCreate()
{
    // shmget 함수: 커널에 공유 메모리 공간 요청 위한 시스템 콜
    // KEY_NUM은 공유 메모리를 식별하는 키, MEM_SIZE는 공유 메모리 크기
    // 플래그 IPC_CREAT은 공유 메모리 생성
    // IPC_EXCL은 이미 해당 키로 공유 메모리가 존재하면 실패하도록, 0666은 권한 설정
    if((shmid = shmget((key_t)KEY_NUM, MEM_SIZE, IPC_CREAT| IPC_EXCL | 0666)) == -1)
    {
        printf("Shared memory already exist.");
        return 1;
    }

    // shmid = shmget((key_t)KEY_NUM, MEM_SIZE, IPC_CREAT| 0666);
    if (shmid == -1)
    {
        cerr<<"Shared memory create fail"<<endl;
        return 1;
    }
    else
    {
        cout<<"Shared memory created succesfully"<<endl;
    }

    return 0;
}

static int SharedMemoryWrite(const char* msg)
{
    void *shmaddr;
    int size=strlen(msg);
    if(size > MEM_SIZE)
    {
        printf("Shared memory size over");
        return 1;
    }
    // shmat 함수: 생성된 공유 메모리 공간에 접근할 수 있는 '식별자'를 얻음
    if((shmaddr = shmat(shmid, (void *)0, 0)) == (void *)-1)
    {
        perror("Shmat failed");
        return 1;
    }
```

```
108         }
109
110         strcpy((char *)shmaddr, msg);
111
112         // shmdt 함수: 공유 메모리를 호출 프로세스의 메모리 영역에서 분리
113         if(shmdt(shmaddr) == -1)
114         {
115             perror("Shmdt failed");
116             return 1;
117         }
118         return 0;
119     }
120
121     static int SharedMemoryRead()
122     {
123         void *shmaddr;
124
125         if((shmaddr = shmat(shmid, (void *)0, 0)) == (void *)-1)
126         {
127             perror("Shmat failed");
128             return 1;
129         }
130
131         cout<<"Message from shared memory: "<<(char*)shmaddr<<endl;
132
133         if(shmdt(shmaddr) == -1)
134         {
135             perror("Shmdt failed");
136             return 1;
137         }
138         return 0;
139     }
140
141     static int SharedMemoryFree()
142     {
143         // shmctl 함수: 공유 메모리 삭제 등 작업
144         shmctl(shmid, IPC_RMID, (struct shmid_ds *)NULL);
145
146         cout<<"Shared memory end"<<endl;
147         return 0;
148     }
```

실행 결과

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS 1

fork  fork.cpp  hello  hello.cpp  shm  shm.cpp
pbj@LAPTOP-4I8AU5BK:~/codes$ ./shm
Parent PID: 13981
Shared memory created succesfully
Child1 PID: 13982
Child2 PID: 13983
Message from shared memory: Operating System
Message from shared memory: SKKU COMEDU
Parent Read complete
Shared memory end
pbj@LAPTOP-4I8AU5BK:~/codes$
```

2. Message Passing 방식 코드

https://velog.io/@jhlee508/%EC%9A%B4%EC%98%81%EC%B2%B4%EC%A0%9C-Message-Queue-IPC

이 웹사이트의 코드를 기반으로 Message Passing 방식을 구현했다. Int 값만 메시지로 전달했던 것과 달리, 문자열도 추가로 전달하고자 했다. 기존 코드에는 없는 msgctl 함수를 이용하여

message queue 상태 정보를 획득하고 queue 를 삭제하는 코드를 작성했다. 앞선 Shared Memory 방식과는 달리 sender 와 receiver 를 다른 코드에 작성하여, sender 코드가 끝난 후와 receiver 코드가 끝난 후 ipcs 명령어를 이용해 차이를 비교할 수 있었다.

sender.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
// for Message queue
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define KEY_NUM 2357
#define BUFFER_SIZE 1024

void msgqueue_print(int msqid);

typedef struct { // Message buffer structure
  long msgtype; // Message type, must be > 0 with 'long' data type
  int value;
  char buf[BUFFER_SIZE]; // Message data to push in queue
} msgbuf;

int main() {
    int cnt = 10;
    int msqid; // Message queue ID
    msgbuf msg;
    msg.msgtype = 1;

    msqid = msgget((key_t) KEY_NUM, IPC_CREAT|0666); // Create Message (message queue key, message flag)

    if (msqid == -1) {
        printf("Message Get Failed!\n");
        exit(0);
    }

    printf("Before Message Sending\n");
    msgqueue_print(msqid);

    for (int i=0;i<cnt;i++)
    {
        msg.value = i;
        strcpy(msg.buf, "I love OS");

        if (msgsnd(msqid, &msg, sizeof(msg), IPC_NOWAIT) == -1) { // IPC_NOWAIT flag: if no more queue space, fail instead of blocking
            printf("Message Sending Failed!\n");
            exit(0);
        }

        printf("value: %d     message: %s\n", msg.value, msg.buf);
        sleep(1);
    }

    printf("After Message Sending\n");
    msgqueue_print(msqid);
    exit(0);
}

void msgqueue_print(int msqid)
{
    struct msqid_ds m_stat;
    printf("========== messege queue info ============\n");
    //msgctl 함수: message queue 상태 정보 획득, 제한 변경, queue 제거 용도
    if(msgctl(msqid,IPC_STAT,&m_stat)==-1)
    {
            printf("msgctl failed");
            exit(0);
    }
    printf(" message queue info \n");
    printf(" msg_qnum : %ld\n",m_stat.msg_qnum); // queue number
    // lspid: 마지막으로 송신한 pid, stime: message send time
    printf(" Last send by process %d at %ld\n",m_stat.msg_lspid, m_stat.msg_stime);
    // lrpid: 마지막으로 수신한 pid, rtime: message receive time
    printf(" Receiving list by %d at %ld\n",m_stat.msg_lrpid, m_stat.msg_rtime); //

    printf("========== messege queue info end ============\n");
}
```

실행 결과

receiver.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
// for Message queue
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define KEY_NUM 2357
#define BUFFER_SIZE 1024

void msgqueue_print(int msqid);

typedef struct {
    long msgtype;
    int value;
    char buf[BUFFER_SIZE];
} msgbuf;

int main() {
    int msqid;
    msgbuf msg;
    msg.msgtype = 1;

    msqid = msgget((key_t) KEY_NUM, IPC_CREAT|0666); // Create Message (message queue key, message flag)

    if (msqid == -1) {
        printf("Message Get Failed!\n");
        exit(0);
    }

    printf("Before Message Receiving\n");
    msgqueue_print(msqid);

    while (1) {
        if (msgrcv(msqid, &msg, sizeof(msg), 1, 0) == -1) {
            printf("Message Receiving Finished!\n");
            break;
        }

        printf("value: %d     message: %s\n", msg.value, msg.buf);
    }

    printf("Message Receiving Finished!\n");
    printf("After Message Sending\n");
    msgqueue_print(msqid);

    //메시지 큐 삭제
    if(msgctl(msqid,IPC_RMID,NULL)==-1)
    {
        printf("msgctl failed\n");
        exit(0);
    }
    exit(0);
}
```

```
pbj@LAPTOP-4I8AU5BK:~/codes$ ls
fork  fork.cpp  hello  hello.cpp  mp-receiver  mp-receiver.c  mp-sender  mp-sender.c  shm  shm.cpp
pbj@LAPTOP-4I8AU5BK:~/codes$ ./mp-sender
Before Message Sending
========== messege queue info ============
 message queue info
 msg_qnum : 0
 Last send by process 0 at 0
 Receiving list by 0 at 0
========== messege queue info end ============
value: 0     message: I love OS
value: 1     message: I love OS
value: 2     message: I love OS
value: 3     message: I love OS
value: 4     message: I love OS
value: 5     message: I love OS
value: 6     message: I love OS
value: 7     message: I love OS
value: 8     message: I love OS
value: 9     message: I love OS
After Message Sending
========== messege queue info ============
 message queue info
 msg_qnum : 10
 Last send by process 17067 at 1695998618
 Receiving list by 0 at 0
========== messege queue info end ============
pbj@LAPTOP-4I8AU5BK:~/codes$ ipcs -q

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x00000935 6          pbj        666        10400        10
```

```
pbj@LAPTOP-4I8AU5BK:~/codes$ ./mp-receiver
Before Message Receiving
========== messege queue info ============
 message queue info
 msg_qnum : 10
 Last send by process 17067 at 1695998618
 Receiving list by 0 at 0
========== messege queue info end ============
value: 0     message: I love OS
value: 1     message: I love OS
value: 2     message: I love OS
value: 3     message: I love OS
value: 4     message: I love OS
value: 5     message: I love OS
value: 6     message: I love OS
value: 7     message: I love OS
value: 8     message: I love OS
value: 9     message: I love OS
^C
pbj@LAPTOP-4I8AU5BK:~/codes$ ipcs -q

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x00000935 6          pbj        666        0            0

pbj@LAPTOP-4I8AU5BK:~/codes$
```