Operating System 실습
[6주차]
IPC(InterProcess Communication)

한양대학교 HANYANG UNIVERSITY
Data-Centric Systems Lab.

InterProcess Communication

- Basic mechanisms that unix systems offer to allow interprocess communication

- Goals of IPC
  - Data transmission
  - Data sharing
  - Event alarm
  - Resource sharing
  - Process management - debgger

InterProcess Communication

- Signal
- Pipes and Named Pipe(FIFO)
- Message Queue
- Shared memory

Signal

- Signal은 프로세스에게 어떤 사건의 발생을 알릴 때 사용
- 각 신호는 관련 default 동작이 있음 : exit, core, stop, ignore
- 신호이름은 파일 <signal.h>에 매크로로 정의되어 있음
- 목록 확인 방법

  $kill -l

```
lee@lee-VirtualBox:~$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Signal

- Process의 정상 종료
  - exit() 시스템콜 사용
- Process의 비정상 종료
  - abort() 시스템 호출 사용
  - 이 시스템 호출은 자기 프로세스에게 SIGABRT 신호를 보냄

Signal	Description
SIGHUP (1)	터미널의 연결이 끊어졌을 때 발생
SIGINT (2)	Ctrl-C 나 Ctrl-Break 키를 입력했을 때 발생
SIGABRT (6)	프로그램의 비정상 종료 시 발생
SIGKILL (9)	프로세스를 죽이기 위해서
SIGUSR1 (10)	사용자를 위해 정의된 시그널
SIGSEGV (11)	잘못된 메모리 참조에 의해 발생
SIGPIPE (13)	단절된 파이프에 write한 경우 발생
SIGSTOP (19)	프로세스의 일시 중단(Ctrl-Z)시 발생
SIGIO (29)	비 동기적인 입출력이 발생했을 때 발생

Signal

- 프로세스가 signal을 받았을 때의 행동
  - 종료 (exit, abort)
    * 대부분의 signal은 signal 수신 시 프로세스가 정상 종료 (normal termination)한다
    * SIGABRT, SIGBUS, SIGSEGV, SIGQUIT, SIGILL, SIGTRAP, SIGSYS, SIGXPU, SIGXFSZ, SIGFPE 시그널은 비정상종료 (abnormal termination)을 발생시킴
  - 시그널 무시 (ignore)
  - 수행 중지 (stop)
  - 사용자 수준 signal 처리 함수 (user level catch function) 수행

Signal

- 터미널에서 특수키를 누르는 경우
  - Ex) Crtl-C -> SIGINT 발생
- 하드웨어 오류
  - 0으로 나눈 경우, 잘못된 메모리를 참조하는 경우 등
- kill 함수의 호출
  - 특정 프로세스나 프로세스 그룹에게 원하는 시그널을 발생하거나 전달한다.
  - 대상 프로세스에 대한 권한이 있어야 한다.
- kill 명령의 실행
  - 내부적으로 kill 함수를 호출한다.
- 소프트웨어적조건
  - 네트워크에서의 데이터오류(SIGURG), 파이프 작업에서의 오류(SIGPIPE), 알람 시계의 종류(SIGALRM) 등

signal 집합

- sigset_t 를 사용하여 정의
- 사용하고자하는 시그널의 리스트를 지정
- 시그널을 다루는 시스템 호출에 전달되는 주요 인수 중의 하나

```
#include <signal.h>
<초기화>
int sigemptyset(sigset_t *set); // 빈 집합 생성
int sigfillset(sigset_t *set) // 풀 집합 생성

<조작>
int sigaddset(sigset_t *set, int signo) // 추가
int sigdelset(sigset_t *set, int signo) // 삭제
```

signal 집합 만들기

```
sigset_t set1, set2;

//빈집합 생성
sigemptyset(&set1);

//시그널 추가
sigaddset(&set1, SIGINT); //SIGINT 추가
sigaddset(&set1, SIGQUIT); //SIGOUT 추가
-----------------------------------------------------
//완전히 차있는 집합생성
sigfillset(&set2);

//시그널제거
sigdelset(&set2, SIGCHLD);
```

sigaction

```
#include <signal.h>

int sigaction(int signo, const struct sigaction *act, struct sigaction *oact);
```

특정 시그널의 수신에 대해 취할 액션을 설정하거나 변경하기 위해서 사용된다

signo
- 행동을 지정하고자 하는 개개 시그널을 식별
- SIGSTOP, SIGKILL을 제외한 모든 시그널 이름 지정가능

act : signo에 대해 지정하고 싶은행동
oact: 현재 설정 값을 채운다.

Signal 실습

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void signal_handler();

void main (int argc, char *argv[])
{
    struct sigaction sact;
    char c;

    printf("(1) Ignore SIGINT signal \n");
    sact.sa_handler = SIG_IGN;
    sigaction(SIGINT, &sact, NULL);
    while((c=getchar()) != '\n');

    printf("(2) User defined Handler \n");
    sact.sa_handler = signal_handler;
    sact.sa_flags = SA_SIGINFO;
    sigaction(SIGINT, &sact, NULL);
    while((c=getchar()) != '\n');

    printf("(3) default SIGINT signal processing \n");
    sact.sa_handler = SIG_DFL;
    sigaction(SIGINT, &sact, NULL);
    while((c=getchar()) != '\n');

}

void signal_handler()
{
    printf("received SIGINT\n");
}
```

실행 후 Ctrl + c 로 SIGINT 발생

SIG_IGN : 시그널을 무시한다
SIG_DFL : 기본행동을 하도록 한다

Pipe

- 운영체제가 제공하는 프로세스간 통신 채널로서 특별한 타입의 파일
  - 일반 파일과 달리 메모리에 저장되지 않고 운영체제가 관리하는 임시 파일
  - 데이터 저장용잉 아닌 프로세스간 데이터 전달용으로 사용

- 파이프를 이용한 프로세스간 통신
  - 송신측은 파이프에 데이터를 쓰고 수신측은 파이프에서 데이터를 읽음
  - 스트림 채널을 제공
  - 송신된 데이터는 바이트 순서가 유지

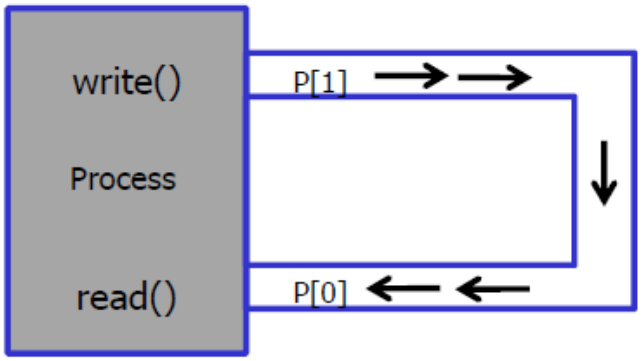
Pipe

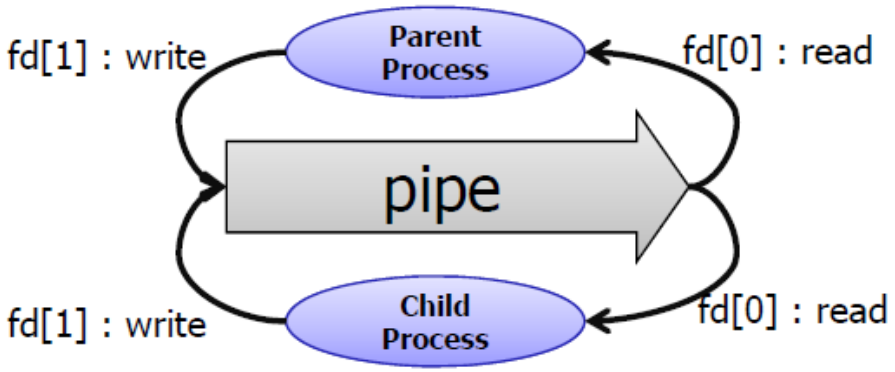
#include <signal.h>

int pipe(int fd[2])

파이프를 열기 위해서는 pipe() 함수를 사용

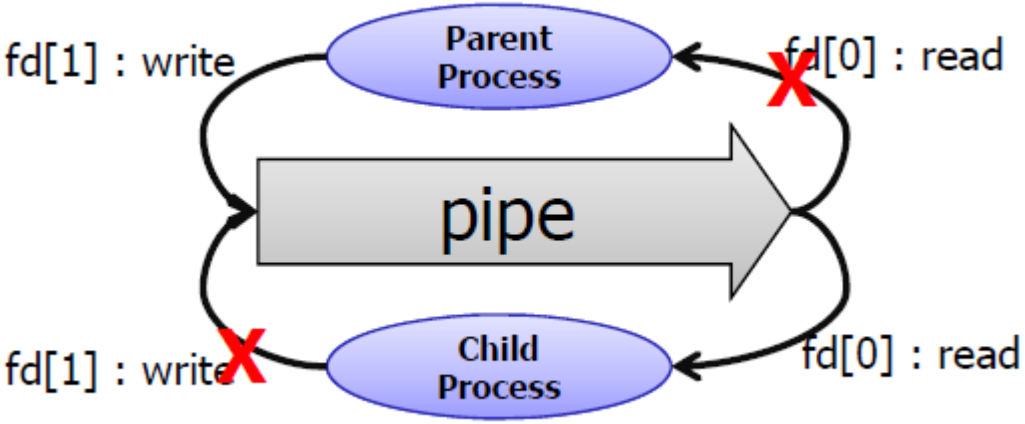
- 하나의 파이프를 생성하면 두 개 (읽기, 쓰기)의 파일 디스크립터가 생성됨
- fd[0]은 읽기용, fd[1]은 쓰기용
- 파이프를 생성한 프로세스가 fork()를 호출하면 자식 프로세스는 부모 프로세스의 파이프를 공유



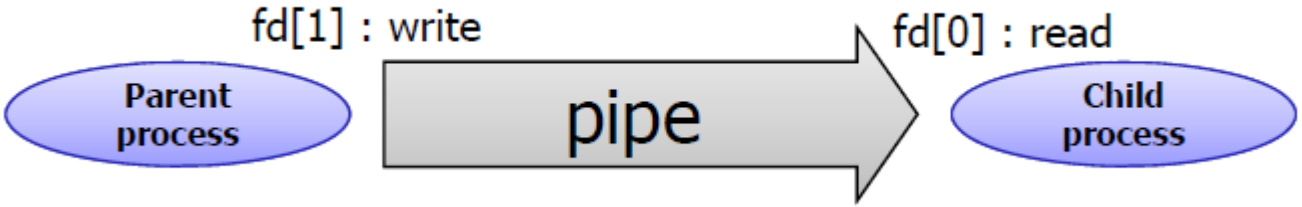


Pipe

- 사용하지 않는 파일 디스크립터를 닫은 상태



- 파이프를 이용한 부모와 자식 프로세스간 통신



Pipe

- 파이프의 크기와 프로세스 제어
  - 파이프의 크기는 유한하다
  - POSIX에는 최소 512바이트로 정의되어 있다
  - 파이프가 full인데 write하면 write하는 프로세스의 수행이 잠시 중단된다
  - 파이프가 empty인데 read하면 read하는 프로세스가 block된다

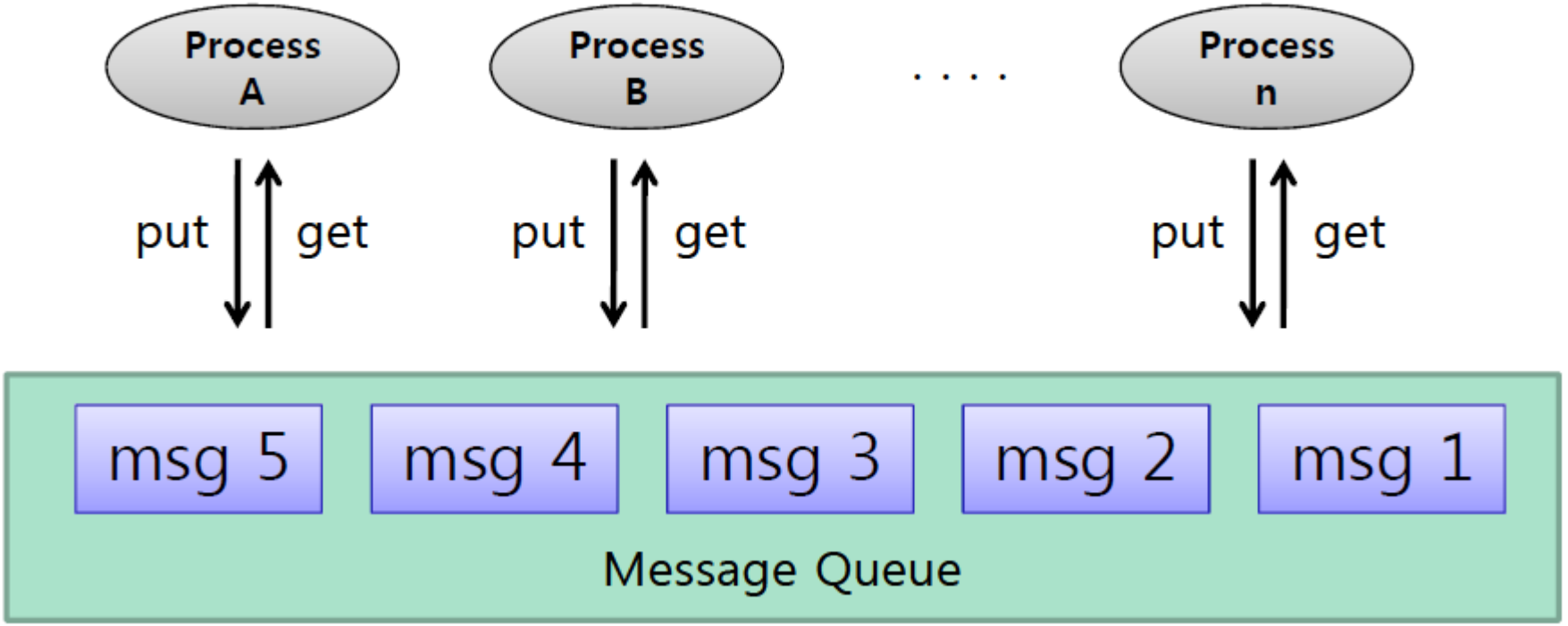
- 파이프 닫기
  - 파이프에 write하는 모든 프로세스가 이 파이프를 닫으면 이 파이프를 read 하는 프로세스의 read 시스템 콜은 0을 반환한다
  - 파이프에 read 하는 모든 프로세스가 이 파이프를 받으면 이 파이프에 write하는 프로세스는 신호 SIGPIPE를 받는다

Pipe 실습

```c
#include <stdio.h>
#include <stdlib.h>

#define MAX_BUF 32

void main(int argc, char *argv[])
{
    char put_msg[MAX_BUF];
    char get_msg[MAX_BUF];
    int pipe_desc[2];

    int child_pid;

    if (pipe(pipe_desc) == -1)
    {
        perror("Create pipe failed\n");
        exit(EXIT_FAILURE);
    }

    if ((child_pid = fork()) == 0)
    {
        printf("< child process >\n");

        while(1)
        {
            read(pipe_desc[0], get_msg, MAX_BUF);

            printf("Get Message is : %s\n", get_msg);

            if (!strncmp(get_msg, "quit", 4))
            {
                exit(EXIT_SUCCESS);
            }
        }
    }
    else if (child_pid > 0)
    {
        printf("< parent process >\n");

        while(1)
        {
            printf("Input pipe : ");

            fgets(put_msg, MAX_BUF, stdin);
            write(pipe_desc[1], put_msg, MAX_BUF);
            if (!strncmp(put_msg, "quit", 4))
            {
                exit(EXIT_SUCCESS);
            }
            sleep(1);
        }
    }
    else
    {
        perror("Failed to create process\n");
    }
}
```

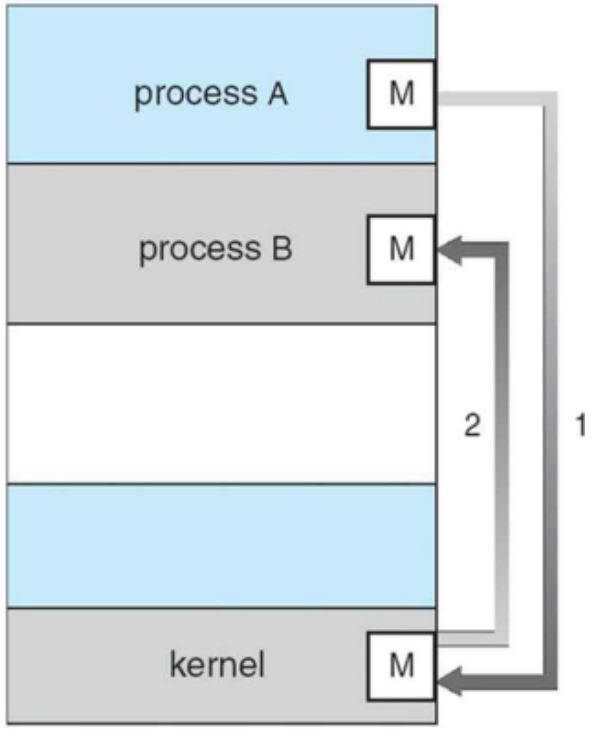
Message queue

- 스트림 채널 외에 “메시지 단위”의 송수신용 큐
- 메시지 전송에 우선순위 부여가 가능
- 메시지큐를 이용한 프로세스간 통신



Message queue

- 프로세스가 생성한 각 메시지는 IPC 메시지 큐로 전송되며, 다른 프로세스가 읽을 때까지 메시지 큐에 유지된다.



Message queue

- 메시지는 고정된 크기의 헤더(Header)와 가변 크기 텍스트(Text)로 이루어진다.
  - 메시지에는 메시지 유형(message type)을 나타내는 정수 값이 붙을 수 있으며, 이러한 정수 값을 이용하여 선택적으로 메시지를 읽을 수 있다
  - 메시지 큐에서 프로세스가 메시지를 읽으면 커널은 메시지를 제거한다
    * 따라서, 단 하나의 프로세스만이 주어진 메시지를 받을 수 있다

Message queue

```
#include <sys/ipc.h>

int msgget(key_t key, int msgflg);
```

key : 메시지큐를 구분하기 위한 고유 키
msgflag : 메시지큐 생성시 옵션을 지정(bitmask 형태)

IPC_CREATE, IPC_EXCL 등의 상수와 파일 접근 권한 지정

msqid_ds 구조체 : 메시지큐가 생성될 때마다 메시지큐에 관한 정보를 담는 메시지큐 객체가 생성

마지막으로 송신 또는 수신한 프로세스 PID, 송수신 시간, 큐의 최대 바이트 수, 메시지큐 소유자 정보 등이 저장

Message queue

Message Queue 객체

```
struct msqid_ds {
        struct ipc_perm msg_perm;       // 메시지큐 접근권한
        struct msg *msg_first;          // 처음 메시지
        struct msg *msg_last;           // 마지막 메시지
        time_t msg_stime;     // 마지막 메시지 송신시각
        time_t msg_rtime;     // 마지막 메시지 수신시각
        time_t msg_ctime;     // 마지막으로 change가 수행된 시각
        struct wait_queue *wwait;
        struct wait_queue *rwait;
        ushort msg_cytes;
        ushort msg_qnum;
        ushort msg_qbytes;    // 메시지큐 최대 바이트수
        ushort msg_lspid;     // 마지막으로 msgsnd를 수행한 PID
        ushort msg_lrpid;     // 마지막으로 받은 PID
};
struct ipc_perm {
        key_t key;            // owner의 euid와egid
        ushort udi;
        ushort gid;
        ushort cuid;          // 생성자의 euid와 egid
        ushort cgid;
        ushort mode;          // 접근모드의 하위 9bits
        ushort seq;           // 순서번호(sequence number)
};
```

Message queue

msgflag 옵션

IPC_CREATE
- 동일한 key를 사용하는 메시지 큐가 존재하면 그 객체에 대한 ID를 정상적으로 리턴
- 존재하지 않는다면 메시지큐 객체를 생성하고 ID를 리턴

• IPC_EXCL
- 동일한 key를 사용하는 메시지 큐가 존재하면 -1을 리턴
- 단독으로 사용하지 못하고 IPC_CREATE와 같이 사용해야 함

• IPC_PRIVATE
- key가 없는 메시지큐 생성
- 명시적으로 key 값을 정의하여 사용할 필요가 없는 경우 이용
- 메시지큐 ID를 서로 공유할 수 있는 부모와 자식 프로세스 사이에 사용 가능
- 외부의 다른 프로세스는 이 메시지큐에 접근 불가

Message queue

```
int msgsnd(int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg);

struct msgbuf {
  long mtype; // 메시지타입> 0
  chat mtext[1]; // 메시지데이터
}
```

- mytype은 1 이상이어야 함
- msgbuf의 첫 4바이트는 반드시 long 타입
- mtext는 문자열, binary 등 임의의 데이터 사용가능
- msgflg는 0으로 한 경우 메시지큐 공간이 부족하면 블록됨
- msgflg를 IPC_NOWAIT로 하면 메시지큐 공간이 부족한 경우 블록되지 않고 EAGAIN 에러코드와 함께 -1을 리턴
- Return value
  [성공시:0]
  [실패시:-1]

Message queue

ssize_t **msgrcv**(int msqid, struct msgbuf *msgp, size_t msgsz, long msgtype, int msgflg);

메시지큐로부터 메시지를 읽는 함수
- msqid : 메시지큐 객체ID
- msgp : 메시지큐에서 읽은 메시지를 저장하는 수신공간
- msgsz : 수신공간의 크기
- msgflg : 메시지가 없는 경우 취할 동작
  - 0이면 대기
  - IPC_NOWAIT이면 EAGAIN 에러코드와 -1을 리턴
  - MSG_NOERROR로 설정하면 msgsz 크기만큼만 읽음
    (읽은 메시지가 수신공간 크기보다 크면 E2BIG 에러가 발생)
- msgtype
  - 0: 타입의 구분없이 메시지큐에 입력된 순서대로 읽음
  - 양수: 그 값을 갖는 메시지를 읽음
  - 음수: 절대값보다 작거나 같은 것 중에서 최소값부터 순서대로 읽음

Message queue

int **msgctl**(int msqid, int cmd, struct msqid_ds *buf);

메시지큐에 관한 정보 읽기, 동작 허가 권한 변경, 메시지큐 삭제 등을 제어

- msqid: 메시지큐 객체ID
- cmd: 제어 명령 구분
  - IPC_STAT :메시지큐 객체에 대한 정보를 buf에 넣도록 시스템에 지시
  - IPC_SET : r/w 권한, euid, egid, msg_qbyte를 변경하는 명령
  - IPC_RMID : 메시지큐를 삭제하는 명령
- buf: cmd 명령에 따라 동작

IPC_SET
: r/w권한, euid, egid, msg_qbyte만 변경이 가능
: IPC_STAT 명령으로 메시지큐의 객체를 얻은 후 변경시키고 IPC_SET call

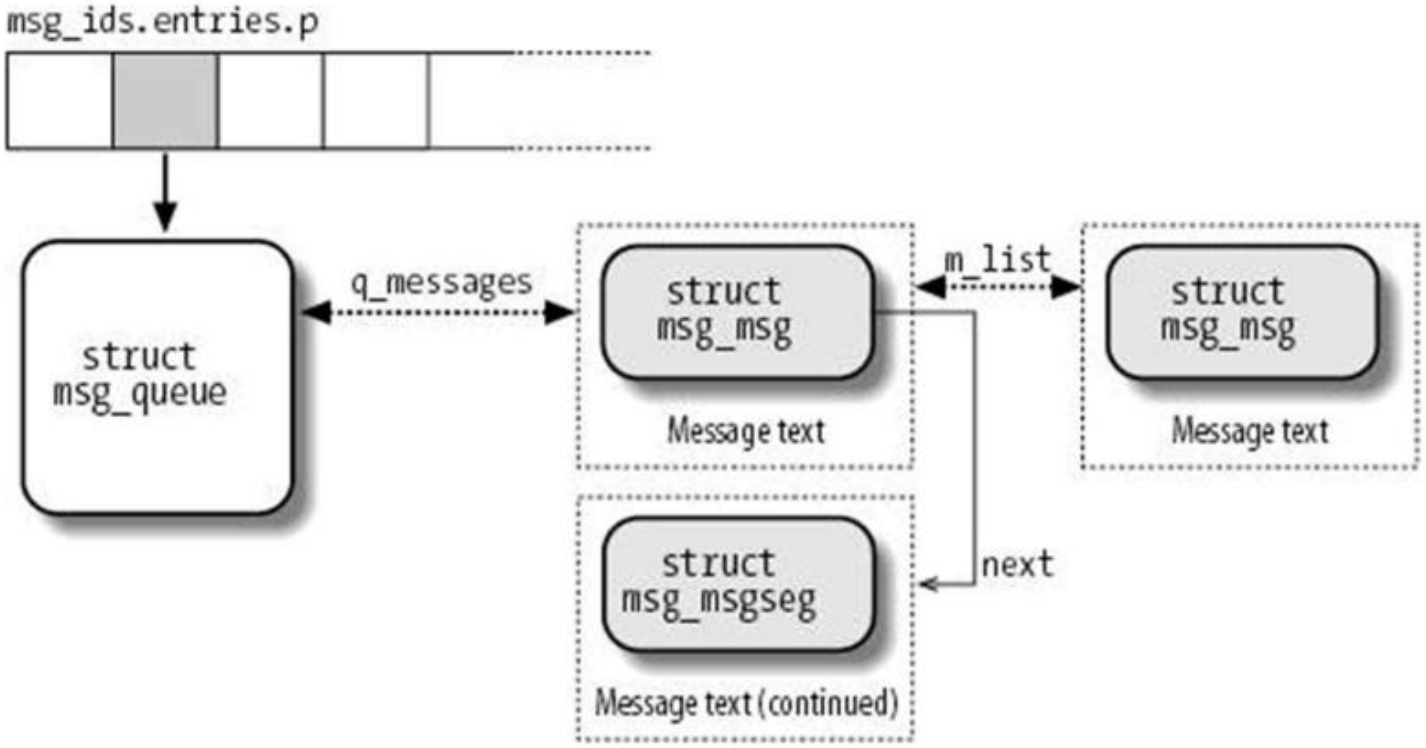
IPC_RMID
: 삭제 명령을 내렸을 때 아직 읽지 않은 메시지가 있어도 즉시 삭제

Message queue

- 자원의 소진을 피하기 위해 제한이 존재한다.
  - IPC 메시지 큐의 자원 수
    * /proc/sys/kernel/msgmni
  - 각 메시지의 크기 (기본 8192)
    * /proc/sys/kernel/msgmax
  - 큐에 있는 메시지의 총 크기 (기본 16,384)
    * /proc/sys/kernel/msgmnb

Message queue

- 메시지 큐 자료구조



Message queue 실습

- Sender source

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/msg.h>

#define  BUFF_SIZE   1024

typedef struct {
    long  data_type;
    int   data_num;
    char  data_buff[BUFF_SIZE];
} t_data;

int main(int argc, char *argv[])
{
    int msqid;
    int i = 0;
    t_data data;

    if ((msqid = msgget((key_t)1111, IPC_CREAT | 0666)) == -1)
    {
        perror("msgget() is failed.\n");
        exit(EXIT_FAILURE);
    }

    for (i=0; i<3; i++)
    {
        data.data_type = 1;
        data.data_num  = i;
        sprintf(data.data_buff, "type=%ld, idx=%d", data.data_type, i);

        if (msgsnd(msqid, &data, sizeof(t_data) - sizeof(long), 0) == -1)
        {
            perror("msgsnd() is failed.\n");
            break;
        }
    }

    // msgctl( msqid, IPC_RMID, 0);     // Delete message queue
}
```

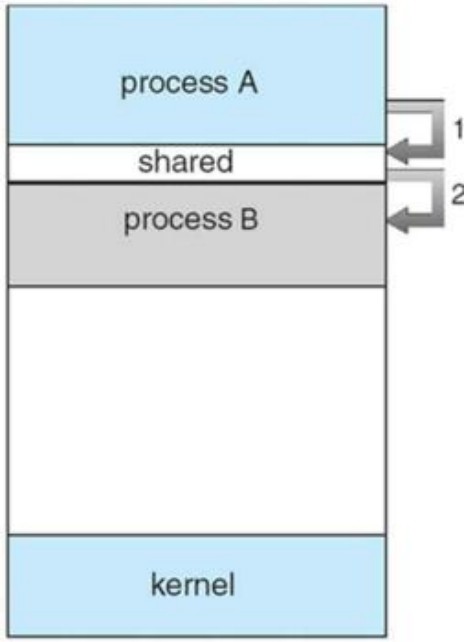
Message queue 실습

- Receiver source

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/msg.h>

#define  BUFF_SIZE   1024

typedef struct {
    long  data_type;
    int   data_num;
    char  data_buff[BUFF_SIZE];
} t_data;

int main( void)
{
    int      msqid;
    t_data   data;
    struct   msqid_ds msqstat;
    int      i;

    if ((msqid = msgget((key_t)1111, IPC_CREAT | 0666)) == -1)
    {
        perror( "msgget() is failed.\n");
        exit(EXIT_FAILURE);
    }

    if (msgctl(msqid, IPC_STAT, &msqstat) == -1)
    {
        perror( "msgctl() is failed.\n");
        exit(EXIT_FAILURE);
    }

    printf("The total number of the message is %ld.\n", msqstat.msg_qnum);

    for (i=0; i<msqstat.msg_qnum; i++)
    {
        if (msgrcv(msqid, &data, sizeof(t_data) - sizeof(long), 0, 0) == -1)
        {
            perror("msgrcv() is failed.\n");
            exit(EXIT_FAILURE);
        }
        printf("%d - %s\n", data.data_num, data.data_buff);
    }

    return 0;
}
```

Shared memory

- 프로세스들이 공통으로 사용할 수 있는 메모리 영역
- 특정 메모리 영역을 다른 프로세스와 공유하여 프로세스 간 통신이 가능
- 데이터를 한 번 읽어도 데이터가 계속 남아 있음
- 같은 데이터를 여러 프로세스가 중복하여 읽어야 할 때 효과적



Shared memory

```
#include <sys/ipc.h>

int shmget(key_t key, int size, int shmflg);
```

int 타입의 공유메모리 ID를 리턴
- struct shmid_ds 구조체에 정보를 저장

key : 새로 생성될 공유메모리를 식별하기 위한 값
- 다른 프로세스가 접근하기 위해서는 이 키 값을 알아야 함

shmflg : 공유메모리 생성 옵션
- bitmask 형태의 인자
- IPC_CREAT, IPC_EXCL, 파일접근권한

Shared memory

shmid_ds 구조체

```
sturct shmid_ds {
        struct ipc_perm shm_perm;       // 동작허가사항
        int shm_segsz;                  // 세그먼트의 크기
        (bytes)time_t shm_atime;        // 마지막attach 시각
        time_t shm_dtime;               // 마지막detach 시각
        time_t shm_ctime;               // 마지막change 시각
        unsigned short shm_cpid;        // 생성자의 PID
        unsigned short shm_lpid;        // 마지막접근자의 PID
        short shm_nattch;               // 현재attaches no.

        // 아래는 private
        unsigned short shm_npages;      // 세그먼트의크기(pages)
        unsigned long *shm_pages;       // array of ptrs to frames ->SHMMAX
        sturct vm_area_struct *attaches;// descriptors for attaches
};
```

Shared memory

shmflg 옵션

- IPC_CREAT를 설정한 경우
  - 같은 key값을 사용하는 공유메모리가 존재하면 해당 객체에 대한 ID를 리턴
  - 같은 key값의 공유메모리가 존재하지 않으면 새로운 공유메모리를 생성 하고 그 ID를 리턴

- IPC_EXCL 과 IPC_CREAT를 같이 설정한 경우
  - 같은 key값을 사용하는 공유메모리가 존재하면 shmget() 호출은 실패하고 -1을 리턴
  - IPC_CREAT와 같이 사용해야 함

- IPC_PRIVATE
  - key값이 없는 공유메모리를 생성
    - 명시적으로 key값을 사용할 필요가 없는 경우에 사용
    - 공유메모리 ID를 서로 공유할 수 있는 부모와 자식 프로세스 사이에 사용가능
    - 외부의 다른 프로세스는 이 공유메모리에 접근 불가

Shared memory

void ***shmat**(int shmid, const void *shmadr, int shmflg);

공유메모리 생성 후 실제 사용 전 물리적 주소를 자신의 프로세스의 가상메모리 주소로 매핑

- shmid : 공유메모리 객체 ID
- shmaddr : 첨부시킬 프로세스의 메모리주소
  - 0 : 커널이 자동으로 빈 공간을 찾아서 처리
- shmflg : 공유메모리옵션
  - 0 : 읽기/쓰기모드
  - SHM_RDONLY : 읽기전용
- 호출 성공 시 첨부된 주소를 리턴, error 시 NULL 포인터를 리턴

```
int shmid = shmget(0x1234, 1023, IPC_CREAT | 0600);
(void *)myaddr = shmat(shmid, (void *)0, 0);
if (myaddr = (void *)-1) {
    perror("공유메모리를 attach하지 못했습니다.₩n");
    exit(0);
}
```

Shared memory

int **shmdt**(const void *shmaddr);

자신이 사용하던 메모리 영역에서 공유메모리를 분리

- shmaddr : shmat()가 리턴했던 주소, 현재 프로세스에 첨부된 공유메모리의 시작주소

- 공유메모리의 분리가 공유메모리의 삭제를 의미하지는 않음
  - 다른 프로세스는 계속 그 공유메모리를 사용할 수 있음

- shmid_ds 구조체의 shm_nattach 멤버변수
  - shmat()로 공유메모리를 첨부하면 1 증가
  - 공유메모리를 분리하면 1 감소

Shared memory

int **shmctl**(int shmid, int cmd, struct shmid_ds *buf);

• 공유메모리의 정보 읽기, 동작 허가 권한 변경, 공유메모리 삭제 등의 공유메모리 제어

- shmid : 공유메모리객체ID
- cmd : 수행할 명령
  - IPC_STAT : 공유메모리 객체에 대한 정보를 얻어 오는 명령
  - IPC_SET : r/w 권한, euid, eguid를 변경하는 명령
  - IPC_RMID : 공유메모리를 삭제하는 명령
- buf : cmd 명령에 따라 의미가 변경
  - 공유메모리 객체 정보를 얻어오는 명령: 얻어온 객체를 buf에 저장
  - 동작 허가 권한을 변경하는 명령: 변경할 내용을 저장

• 여러 프로세스가 병행하여 쓰기/읽기 작업을 수행하면 동기화 문제가 발생할 수 있음

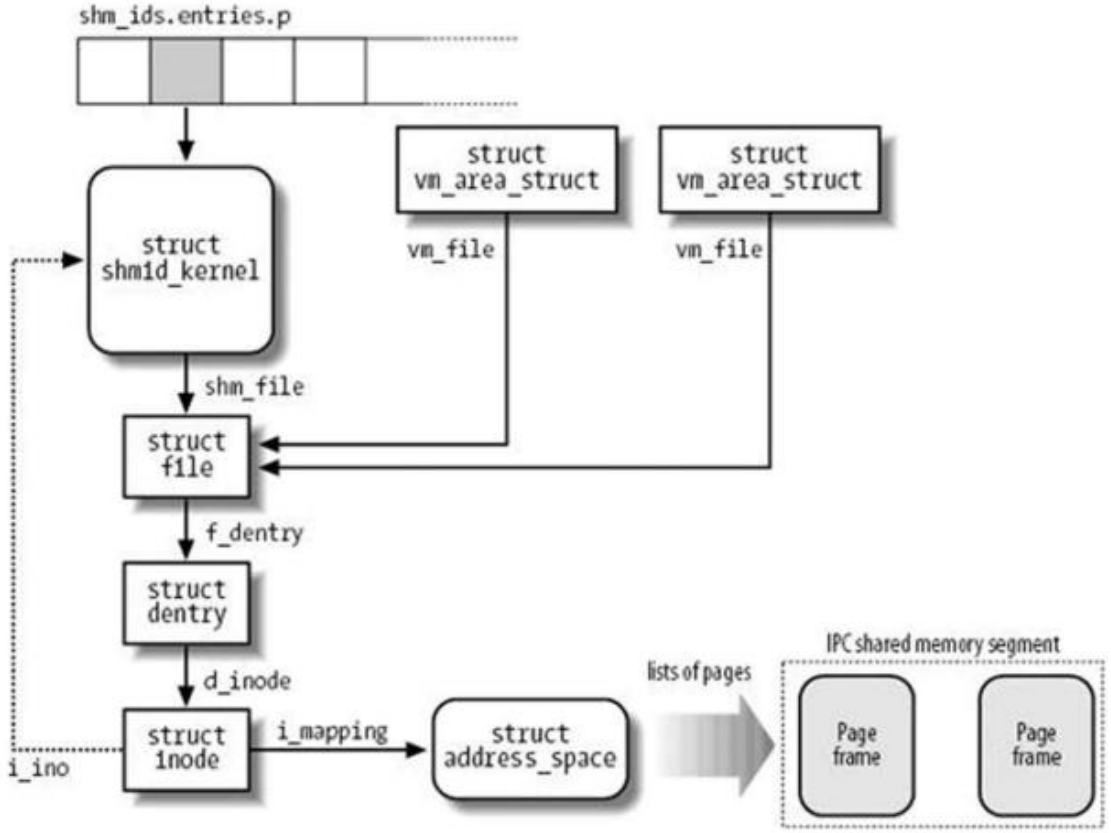
- 하나의 공유데이터를 둘 이상의 프로세스가 동시에 접근함으로써 발생할 수 있는 문제
- 데이터의 값이 부정확하게 사용되는 문제가 있음

Shared memory

- 사용자 모드 프로세스가 공유 메모리를 너무 많이 사용하는 것을 방지하기 위해 제한이 존재한다
  - IPC 공유 영역의 수
    - /proc/sys/kernel/shmmni
  - 각 세그먼트의 크기
    - /proc/sys/kernel/shmall
  - 모든 세그먼트의 최대 총 사용량
    - /proc/sys/kernel/shmmax

Shared memory

- IPC 공유 메모리 자료 구조



실습 문제

- 2개의 쓰레드 생성
- SIGUSR1, SIGUSR2 사용
- Thread 1은 SIGUSR1 받도록 설정
- Thread 2은 SIGUSR2 받도록 설정
- 각 signal에 따른 handler 실행

  - Ex)
  - thread ➔ printf(“Thread [%d] is received\n”, thread_num);
  - signal handler ➔printf(“SIGUSR1 signal handler is executed\n”);

- 자식 프로세스에서는 부모 프로세스에 SIGUSR1, SIGUSR2를 보낸다.

실습 문제

```
#include <sys/types.h>
#include <signal.h>

int kill (pid_t pid, int sig);
```

- pid> 0 : pid를 가진 프로세스에게 시그널을 보낸다
- pid== 0 : 시그널은 보내는 프로세스와 같은 프로세스 그룹에 속하는 모든 프로세스에 보내짐
- pid== -1 & not superuser: 프로세스의 유효사용자 식별번호와 같은 모든 프로세스에 보내짐
- pid== -1 & superuser: 특수한 시스템 프로세스를 제외한 모든 프로세스에 보내짐
- pid< -1 : 프로세스의 그룹 식별번호가 pid의 절대값과 같은 모든 프로세스에 보내짐

실습 문제

```
#include <unistd.h>

int pause (void);
```

- 신호를 받을 때까지 호출 프로세스를 중지시킨다

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
void sig_handler(int signo)
{
        printf("SIGINT occur ₩n");
}
int main()
{
        printf("hello world!₩n");
        signal(SIGINT, (void *)sig_handler);
        pause();
        printf("Interrupt₩n");
}
```