

8. 프로세스간 통신

- IPC(Inter Process Communication)은 멀티프로세스로 구현된 서버에서 프로세스간에 데이터를 전달하는데 사용
- IPC 기술
 - 파이프, 메시지 큐, 세마포어, 공유메모리

8.1 파이프

파이프(PIPE)

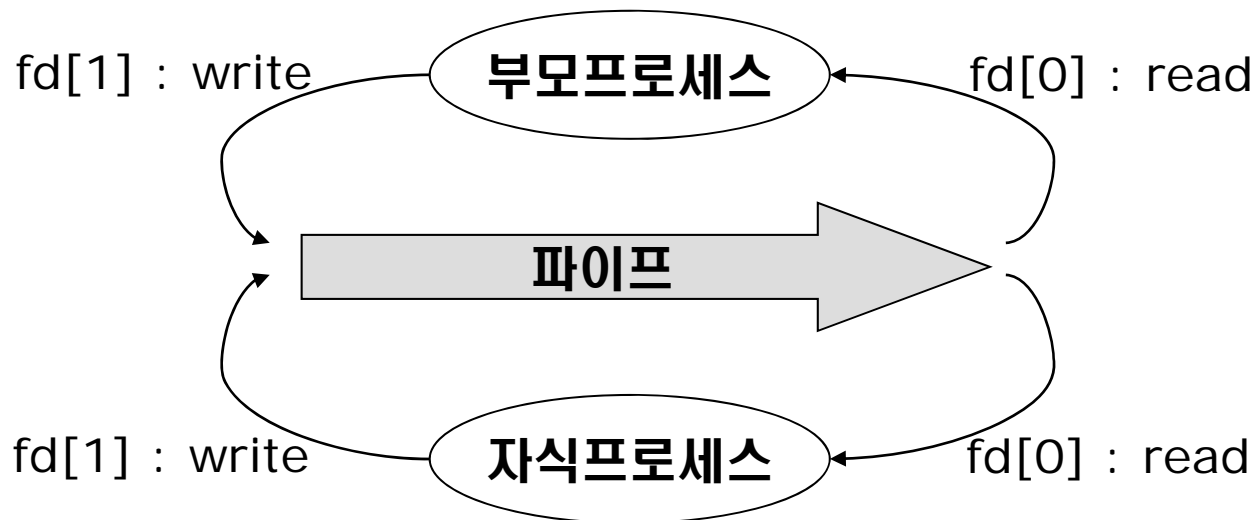
- 운영체제가 제공하는 프로세스간 통신 채널로서 특별한 타입의 파일
 - 일반 파일과 달리 메모리에 저장되지 않고 운영체제가 관리하는 임시 파일
 - 데이터 저장용이 아닌 프로세스간 데이터 전달용으로 사용
- 파이프를 이용한 프로세스간 통신
 - 송신측은 파이프에 데이터를 쓰고 수신측은 파이프에서 데이터를 읽음
 - 스트림 채널을 제공
 - TCP연결은 원격 프로세스간에 스트림 채널을 제공
 - pipe는 같은 컴퓨터내의 프로세스간의 스트림 채널을 제공
 - 송신된 데이터는 바이트 순서를 유지

파이프 생성

- 파이프를 열기 위해서는 시스템 콜 `pipe()`를 사용
 - 하나의 파이프를 생성하면 두 개(읽기, 쓰기)의 파일 디스크립터가 생성됨

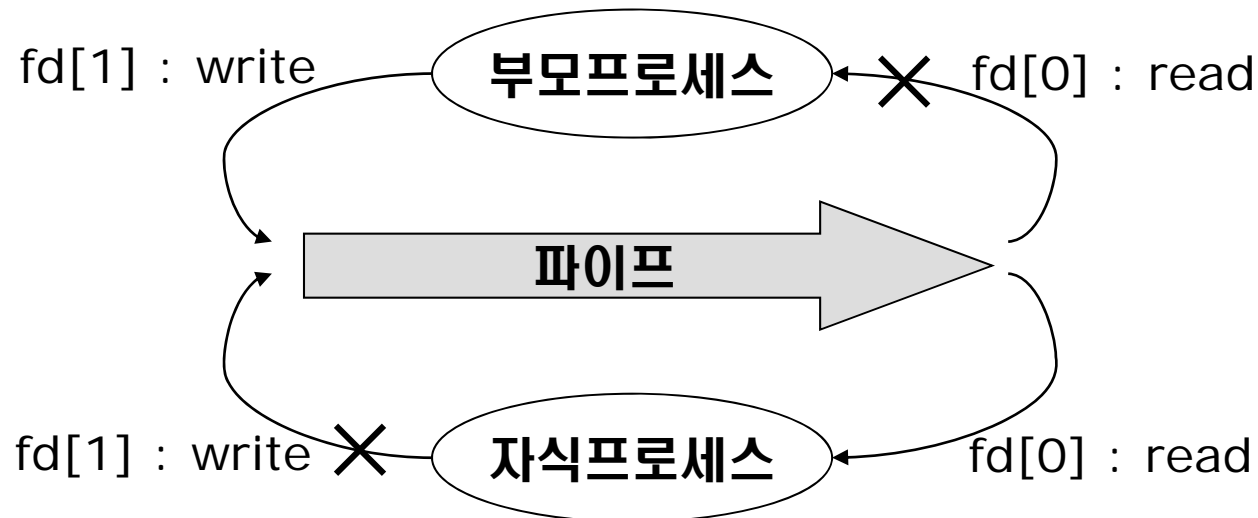
```
#include <unistd.h>  
int pipe(int fd[2]);
```

- `fd[0]`은 읽기용, `fd[1]`은 쓰기용
- 파이프를 생성한 프로세스가 `fork()`를 호출하면 자식 프로세스는 부모 프로세스의 파이프를 공유
 - 부모, 자식 프로세스가 동일 디스크립터(`fd[1]`: 쓰기, `fd[0]`: 읽기)를 사용



파이프 생성

- 부모에서 자식으로 데이터를 보내는 경우



- 사용하지 않는 파일 디스크립터를 닫은 상태
- 파이프를 이용한 부모와 자식 프로세스간 통신



파이프 생성

- 파이프 생성 코드

```
int fd[2];
pid_t pid;
pipe(fd);
if ((pid=fork())<0) {           // 자식 프로세스 생성
    exit(0);
}
else if (pid>0) {
    close(fd[0]);              // 부모 프로세스: 읽기용 파일 디스크립터 제거
}
else if (pid==0) {
    close(fd[1]);              // 자식 프로세스 : 쓰기용 파일 디스크립터 제거
}
```

- 파이프의 특징

- 단방향 스트림 채널만 제공
- 양방향 통신을 위해서는 추가적인 파이프를 생성해야 함

파이프를 이용한 에코 서버, 클라이언트 프로그램 (udpserv_pipeecho.c, udpechocli.c)

- 두 개의 프로세스로 구성된 UDP형 에코 서버 프로그램
 - 부모 프로세스(parent_start)는 클라이언트로부터 받은 메시지를 파이프에 write
 - UDP 서버이므로 부모프로세스는 클라이언트가 보내온 데이터와 함께 클라이언트의 주소를 자식프로세스에게 전달
 - 자식 프로세스(child_start)는 파이프로부터 데이터를 read하여 클라이언트로 에코
 - 파이프를 통해 전달할 데이터의 구조체 정의

// 파이프에 쓰는 데이터 구조

```
typedef struct mseg {  
    struct sockaddr_in addr; // 클라이언트 주소  
    char data[MAX_BUFSZ]; // 에코할 데이터  
} mesg_t;
```

- 클라이언트는 UDP 소켓을 개설, 키보드 입력을 sendto()로 서버에 전달, recvfrom()으로 수신하여 화면에 출력

8.2 FIFO

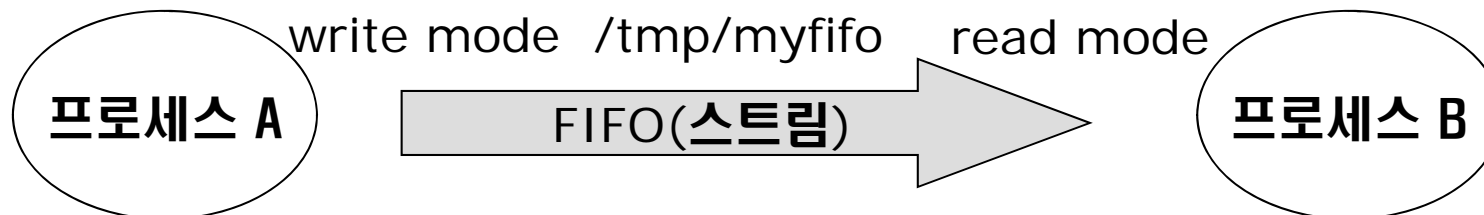
- FIFO(named pipe)는 임의의 프로세스간의 통신을 허용

FIFO 생성

- 파이프는 `fork()`로 만들어진 프로세스들 사이의 통신에만 사용 가능한 제약이 있음
- 제한을 극복하기 위해 파이프에 이름을 지정하고 임의의 다른 프로세스에서 파이프에 접근하도록 한 것을 named pipe(FIFO)라 함
- `mkfifo()`

```
int mkfifo(const char *pathname, mode_t mode);
```

- `pathname` : 파이프의 이름, 경로명이 없으면 현재 디렉토리
- `mode` : FIFO의 파일 접근 권한 설정
- 읽기/쓰기 수행
 - FIFO를 생성한 후 FIFO를 `open()`해야 함
 - FIFO도 파이프의 일종으로 프로세스간 스트림 채널을 제공



FIFO를 이용한 에코 서버 프로그램 (udpserv_fifoecho.c)

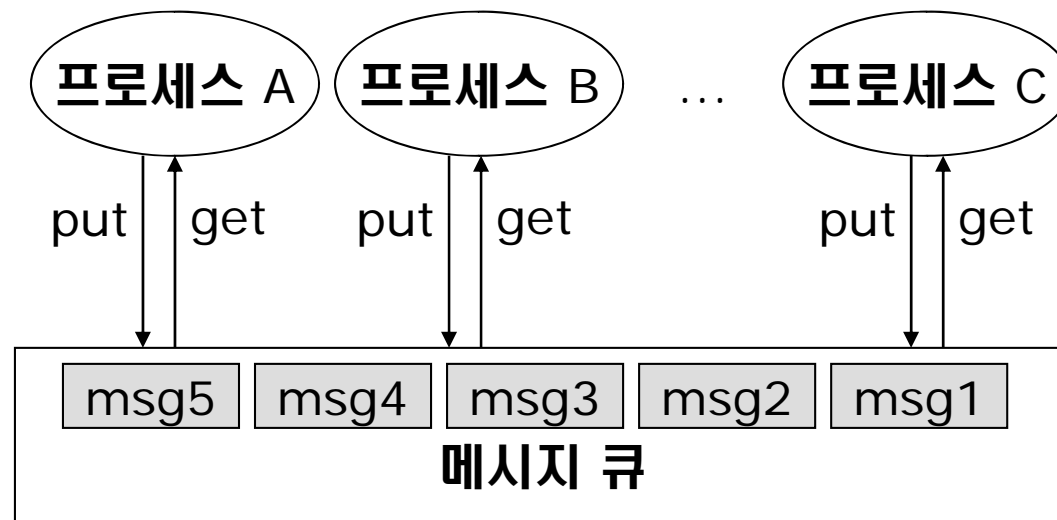
- 파이프를 이용한 에코서버를 FIFO를 이용하여 작성
 - 부모프로세스
 - 클라이언트로부터 받은 메시지를 FIFO에 write
 - 자식프로세스
 - FIFO로부터 데이터를 읽어 클라이언트에 에코
- FIFO에서는 FIFO의 이름만 알면 언제든지 오픈하여 사용
 - 파이프에서는 fork() 전에 파이프를 생성해야 부모, 자식 프로세스가 파일을 공유 할 수 있다.

8.3 메시지큐

- 메시지큐를 사용하여 프로세스사이에 통신
 - 특정 프로세스에게 메시지 단위 전송이 가능
 - 메시지 전송에 우선순위 부여 가능
- 파이프나 FIFO는 프로세스 사이에 제공되는 스트림을 사용하여 통신
 - 메시지 단위로 송신할 때는 주의가 필요

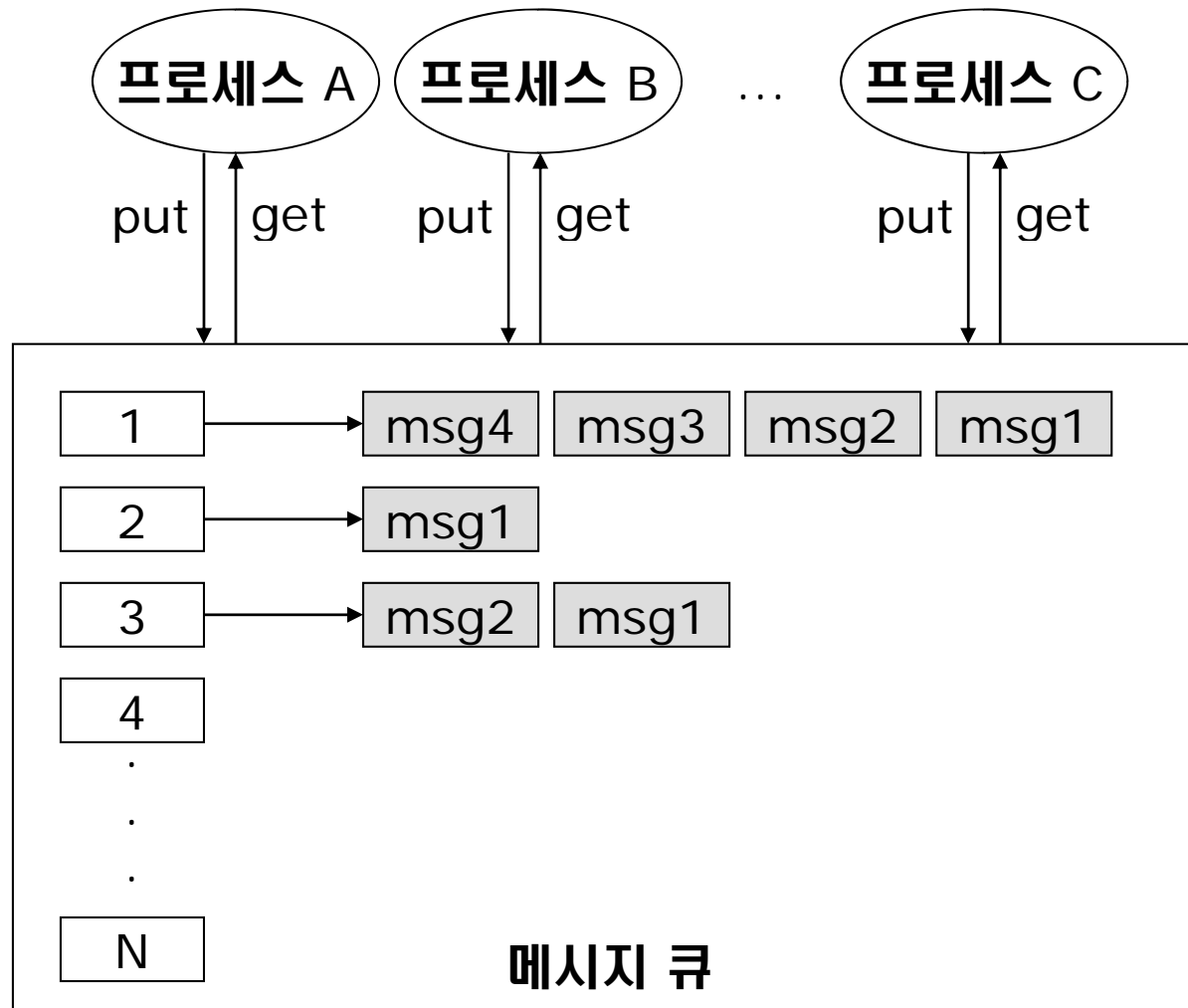
메시지큐 개요

- “메시지 단위”의 송수신용 큐
- 메시지 전송에 우선순위 부여가 가능
- 메시지큐를 이용한 프로세스간 통신
 - put : 메시지 큐에 메시지 쓰기
 - get : 메시지 큐의 메시지 읽기



타입이 있는 메시지 큐

- 2차원 메시지 큐



타입이 있는 메시지 큐(2)

- 메시지 송수신 시에 프로세스는 타입을 지정
 - 큐를 타입 값으로 구분하고, 메시지를 읽고 쓸 때 큐 타입을 지정
 - 특정 프로세스에 메시지 전송이 가능
 - 예 : 메시지를 put할 때 수신할 프로세스의 PID 값을 타입으로 지정하고, 수신 프로세스는 자신의 PID를 타입으로 get
- 메시지 큐의 우선순위 부여 가능
 - 예 : 타입 값을 우선순위로 사용하면 우선순위가 높은 메시지를 먼저 수신하고 우선순위가 높은 메시지가 없으면 우선순위가 낮은 메시지를 수신

메시지큐 생성

- msgget()

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

- key

- 메시지큐를 구분하기 위한 고유 키
 - 메시지 큐를 생성하지 않은 다른 프로세스가 메시지 큐에 접근하려면 key를 알아야 함

- msgflag : 메시지큐 생성시 옵션을 지정(bitmask 형태)

- IPC_CREATE, IPC_EXCL 등의 상수와 파일 접근 권한 지정

- 메시지큐 ID를 반환 : 다른 프로세스는 이 ID를 통해 메시지큐를 사용

- msqid_ds 구조체

- 메시지큐가 생성될 때마다 메시지 큐에 관한 각종 정보를 담은 메시지 큐 객체를 생성

- 마지막으로 송신 또는 수신한 프로세스 PID
 - 송수신 시간
 - 큐의 최대 바이트 수
 - 메시지큐 소유자 정보

메시지큐 객체

- ```
struct msqid_ds {
 struct ipc_perm msg_perm;
 struct msg *msg_first;
 struct msg *msg_last;
 time_t msg_stime;
 time_t msg_rtime;
 time_t msg_ctime;
 struct wait_queue *wwait;
 struct wait_queue *rwait;
 ushort msg_cbytes;
 ushort msg_qnum;
 ushort msg_qbytes;
 ushort msg_lspid;
 ushort msg_lrpid;
};

struct ipc_perm {
 key_t key;
 ushort uid;
 ushort gid;
 ushort cuid;
 ushort cgid;
 ushort mode;
 ushort seq;
};
```

**// 메시지큐 접근권한**  
**// 처음 메시지**  
**// 마지막 메시지**  
**// 마지막 메시지 송신 시각**  
**// 마지막 메시지 수신 시각**  
**// 마지막으로 change가 수행된 시각**  
**// wait 큐**

**// 큐에 있는 모든 메시지들의 바이트 수**  
**// 큐에 있는 메시지 수**  
**// 메시지큐 최대 바이트 수**  
**// 마지막으로 msgsnd를 수행한 PID**  
**// 마지막으로 받은 PID**

**// owner의 euid와 egid**

**// 생성자의 euid와 egid**

**// 접근 모드의 하위 9bits**  
**// 순서번호(sequence number)**



## msgflag 옵션

- IPC\_CREAT
  - 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\_CREAT와 같이 사용해야 함
- IPC\_PRIVATE
  - key가 없는 메시지 큐 생성
  - 명시적으로 키 값을 정의하여 사용할 필요가 없는 경우 이용
    - 예 : 메시지큐 ID를 서로 공유할 수 있는 부모와 자식 프로세스 사이에 사용 가능
  - 외부의 다른 프로세스는 이 메시지큐에 접근 불가
- msgget() 사용 예 : mkq.c
  - 키값을 인자로 주어 메시지큐 객체를 생성한 후 메시지 큐 ID를 출력

- **실행 결과**

\$ mkq 123

created queue id = 229377

key is = 123

opened queue id = 229377

# 메시지 송신

- msgsnd()
  - 메시지큐에 메시지를 넣는 함수

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

struct msgbuf {
 long mtype; // 메시지 타입 > 0
 char mtext[1]; // 메시지 데이터
}
```

- mtype은 1 이상이어야 함
- msgbuf의 첫 4바이트는 반드시 long 타입
- mtext는 문자열, binary 등 임의의 데이터 사용 가능
- msgsz는 mtext만의 크기
- msgflg를 IPC\_NOWAIT로 하면 메시지큐 공간이 부족한 경우 블록되지 않고 EAGAIN 에러코드와 함께 -1을 리턴
  - 0으로 한 경우 메시지큐 공간이 부족하면 블록
- msgsnd 함수는 성공하면 0, 실패하면 -1을 리턴

# 메시지 수신

- msgrcv()
  - 메시지큐로 부터 메시지를 읽는 함수

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

- msqid : 메시지큐 객체 ID
- msgp : 메시지큐에서 읽은 메시지를 저장하는 수신 공간
- msgsz : 수신 공간의 크기
- msgflg : 메시지가 없는 경우 취할 동작, 0이면 대기, IPC\_NOWAIT이면 EAGAIN 에러코드와 -1을 리턴
- 읽은 메시지가 수신 공간 크기보다 크면 E2BIG 에러가 발생
  - msgflg를 MSG\_NOERROR로 설정하면 msgsz 크기만큼만 읽고, 나머지는 잘려진다.
  - msgtype은 메시지큐에서 읽을 메시지 타입을 지정
    - 0으로 하면 타입의 구분 없이 메시지큐에 입력된 순서대로 읽음
    - -10을 지정하면 타입이 10보다 같거나 작은 메시지를 읽음
      - » 타입이 10인 메시지부터 우선순위를 두어 읽음

## 메시지큐 이용 예(qsnd\_pid.c)

- msgtype 값을 특정 프로세스의 PID로 지정하여 지정된 프로세스가 메시지를 수신하는 프로그램
  - 자식 프로세스는 “msgtype=getppid()”로 부모프로세스의 PID를 구하여 이를 큐 타입으로 사용하여 메시지큐로 송신
  - 부모 프로세스는 메시지를 수신 후 메시지큐를 삭제
    - msgctl(qid, IPC\_RMID, 0)
- 실행 결과

```
$ qsnd_pid 1234
Enter message to post :
```

```
10510 프로세스 메시지큐 읽기 대기중..
```

```
Hi! my parent
```

```
<----- 자식 프로세스 입력 데이터
```

```
message posted
```

```
recv = 14 bytes
```

```
<----- 부모 프로세스 출력
```

```
type = 10510
```

```
수신 프로세스 PID= 10510
```

```
value = Hi! my parent
```

# 메시지큐 제어

- msgctl()
  - 메시지큐에 관한 정보 읽기, 동작 허가 권한 변경, 메시지큐 삭제 등을 제어

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

- msqid : 메시지큐 객체 ID
- cmd : 제어 명령 구분
  - IPC\_STAT : 메시지큐 객체에 대한 정보를 얻는 명령
  - IPC\_SET : r/w 권한, euid, egid, msg\_qbytes를 변경하는 명령
  - IPC\_RMID : 메시지큐를 삭제하는 명령
- buf : cmd 명령에 따라 동작
- IPC\_SET
  - r/w권한, euid, egid, msg\_qbytes만 변경이 가능
  - IPC\_STAT 명령으로 메시지큐 객체를 얻은 후 변경시키고 IPC\_SET을 호출
- IPC\_RMID
  - 삭제 명령을 내렸을 때 아직 읽지 않은 메시지가 있어도 즉시 삭제

## msgctl() 사용예 (qctl.c)

- IPC\_STAT를 사용해 메시지큐 객체를 얻어와서 큐에 대한 정보를 출력하고, IPC\_RMID를 사용하여 큐를 삭제
- 실행 결과

```
$mkq 1234
created queue id = 163840
key is = 1234
opened queue id = 163840
$ qctl 1234
큐의 최대 바이트수 : 16384
큐의 유효 사용자 UID : 502
큐의 유효 사용자 GID : 502
큐 접근권한 : 0666
메시지큐 163840 삭제됨
```

# 메시지큐를 이용한 에코서버(msgq\_echoserv.c, res\_send.c)

- **UDP형 에코서버를 메시지큐를 이용하여 멀티프로세스 모델로 구현**
  - REQ\_RECV\_PROC : 에코 요청 메시지를 수신하는 프로세스
    - 에코요청을 메시지큐에 넣고 다음 요청을 기다리는 동작을 반복
    - 한 개의 프로세스만 실행
  - RES\_SEND\_PROC : 에코 메시지를 응답하는 프로세스
    - 메시지큐에서 읽기를 대기하다 메시지가 도착하면 클라이언트에게 응답하는 동작을 반복
    - 여러 개의 프로세스를 실행
      - 한 개의 메시지큐로 부터 메시지를 읽도록 내부적으로 동기화
  - RES\_SEND\_PROC를 생성하는 코드

```
#define RES_SEND_PROC "res_send"
void fork_and_exec(char *key, char *port) {
 pid_t pid = fork();
 if (pid < 0) errquit(" fork fail");
 else if(pid >0)
 return;
 execlp(RES_SEND_PROC, RES_SEND_PROC, key, port, 0);
 perror("execlp fail ");
}
```



## 메시지큐를 이용한 에코서버(계속)

- 메시지큐로 전달할 메시지 구조

```
typedef struct _msg {
 long msg_type; // 메시지큐 타입
 struct sockaddr_in addr; // 클라이언트의 소켓주소
 char msg_text[MAX_BUFSZ]; // 메시지를 저장
} msg_t ;
```

- 메시지큐와 소켓을 생성 후 클라이언트의 메시지가 도착하면  
recvfrom으로 읽고, msgsnd를 사용하여 메시지큐에 넣는다.

```
while(1) {
 nbytes = recvfrom(sock, pmsg.msg_text, MAX_BUFSZ, 0,
 (struct sockaddr *)&pmsg.addr, &len);
 if(nbytes < 0) { perror("recvfrom fail "); continue; }
 pmsg.msg_text[nbytes] = 0;

 // 메시지큐에 쓰기
 if(msgsnd(msqid, &pmsg, size, 0) == -1)
 errquit("msgsnd fail ");
}
```

## RES\_SEND\_PROC(res\_send.c)

- execpl 가 호출될 때 인자로부터 키와 포트번호를 얻는다.
- 메시지큐에서 읽기를 대기하다 메시지가 도착하면 클라이언트에게 응답 메시지를 전송
- 수행결과
  - 여러 프로세스가 순서대로 에코를 처리
  - 커널의 프로세스 스케줄러가 최근에 실행한 프로세스의 우선순위를 낮게 주어 같은 프로세스가 연속적으로 CPU서비스를 받는 것을 금지

```
$ msgq_echoserv 12345 9999
Server starting....
reply process PID = 7228
aaaaaaaaaaaaaaaaaaaaaa
reply process PID = 7229
bbbbbb
reply process PID = 7230
cccccccccccccccccccc
reply process PID = 7228
ddddddddddddddddd
reply process PID = 7229
eeeeeeeeeeeeee
reply process PID = 7230
ffffffffffffffffffffffff
```

## 8.4 공유메모리

- 프로세스간의 통신을 위해서 공유메모리를 사용하는 방법을 소개

# 공유메모리 사용

- 공유메모리
  - 프로세스들이 공통으로 사용할 수 있는 메모리 영역
  - 특정 메모리 영역을 다른 프로세스와 공유하여 프로세스간 통신이 가능
  - 데이터를 한 번 읽어도 데이터가 계속 남아 있음
    - 같은 데이터를 여러 프로세스가 중복하여 읽어야 할 때 효과적
- 공유 메모리를 생성하는 함수 : shmget()

```
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
- int size : 공유 메모리 크기
- shmflg : 공유메모리 생성 옵션을 지정
  - bitmask 형태의 인자
  - IPC\_CREAT, IPC\_EXCL, 파일 접근 권한

# 공유메모리 사용

- shmid\_ds 구조체

```
struct shmid_ds {
 struct ipc_perm shm_perm; // 동작 허가 사항
 int shm_segsz; // 세그먼트의 크기(bytes)
 time_t shm_atime; // 마지막 attach 시각
 time_t shm_dtime; // 마지막 detach 시각
 time_t shm_ctime; // 마지막 change 시각
 unsigned short shm_cpid; // 생성자의 PID
 unsigned short shm_lpid; // 마지막 접근자의 PID
 short shm_nattch; // 현재 attaches 프로세스 수
 // 아래는 private
 unsigned short shm_npages; // 세그먼트의 크기 (pages)
 unsigned long *shm_pages; // array of ptrs to frames -> SHMMAX
 struct vm_area_struct *attaches; // descriptors for attaches
};
```

## Shmflg : 공유메모리 생성 옵션

- **IPC\_CREAT를 설정한 경우**
  - 같은 key값을 사용하는 공유메모리가 존재하면 해당 객체에 대한 ID를 리턴
  - 같은 key값의 공유메모리가 존재하지 않으면 새로운 공유메모리를 생성하고 그 ID를 리턴
- **IPC\_EXCL과 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
    - 메시지큐 ID를 서로 공유할 수 있는 부모와 자식 프로세스 사이에 사용 가능
    - 외부의 다른 프로세스는 이 메시지큐에 접근 불가

# 공유메모리 첨부

- shmat()
  - 공유메모리 생성 후 실제 사용 전에 물리적 주소를 자신의 프로세스의 가상메모리 주소로 맵핑(프로세스에 공유메모리를 첨부한다고 표현)

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

- shmid : 공유메모리 객체 ID
- shmaddr : 첨부시킬 프로세스의 메모리 주소
  - 0 : 커널이 자동으로 빈 공간을 찾아서 처리
- shmflg : 공유메모리 옵션
  - 0 : 읽기/쓰기 모드
  - SHM\_RDONLY : 읽기 전용
- 호출 성공시 첨부된 주소를 리턴하고, 에러 시 -1을 리턴
- 사용예

```
int shmid = shmget(0x1234, 1024, IPC_CREAT | 0600);
char *myaddr = shmat(shmid, 0, 0); //자동으로 빈 공간을 찾고, 읽기/쓰기 가능
//오류 발생 시 -1을 리턴
if (myaddr == (char *)-1) {
 perror("공유메모리를 attach하지 못했습니다.\n");
 exit(0);
}
```

# 공유메모리의 분리

- shmdt()
    - 공유메모리의 사용 종료 시, 자신이 사용하던 메모리 영역에서 **공유메모리를 분리**
- ```
int shmdt(const void *shmaddr);
```
- shmaddr : shmat()가 리턴했던 주소, 현재 프로세스에 첨부된 공유메모리의 시작 주소
 - 공유메모리의 분리가 공유메모리의 삭제를 의미하지는 **않음**
 - 다른 프로세스는 계속 그 공유메모리를 사용할 수 있음
 - shmid_ds 구조체의 shm_nattach **멤버 변수**
 - shmat()로 공유메모리를 첨부하면 1 증가
 - 공유메모리를 분리하면 1 감소

공유메모리 제어

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

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

- shmid : 공유메모리 객체 ID
- cmd : 수행할 명령
 - IPC_STAT : 공유메모리 객체에 대한 정보를 얻어오는 명령
shmctl(shmid, IPC_STAT, buf)
 - IPC_SET : r/w 권한, euid, egid를 변경하는 명령
shmctl(shmid, IPC_SET, &shmds)
 - IPC_RMID : 공유메모리를 삭제하는 명령
shmctl(shmid, IPC_RMID, 0)
- buf : cmd 명령에 따라 의미가 변경
 - 공유메모리 객체 정보를 얻어오는 명령 : 얻어온 객체를 buf에 저장
 - 동작 허가 권한을 변경하는 명령 : 변경할 내용을 buf에 저장

공유메모리의 동기화문제 처리

- 공유메모리는 메모리에 기록한 데이터를 다수의 프로세스가 복사하여 읽는 것이 가능
 - 같은 데이터를 다수의 프로세스에게 전달이 용이
 - 예 : 상태 정보를 공유메모리에 쓰면 여러 프로세스가 읽도록 하는 경우
- 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
 - 해결방안 : 세마포어, 스레드

동기화문제 예(shmbusyaccess.c)

- 3개의 프로세스(부모, 2개의 자식)가 경쟁적으로 공유메모리에 접근하여 동기화 문제를 발생시키는 예
 - 공유메모리 생성하고 두 개의 자식 프로세스를 생성
 - 3개의 프로세스에서 공유메모리를 접근하는 busy()를 수행
 - busy()에서는 access_shm()을 호출하여 공유메모리에 접근
- access_shm()
 - 공유메모리에 자신의 PID를 기록
 - 지연
 - 공유메모리에 남아있는 PID와 자신의 PID를 비교
 - 다르면 access_shm()이 반환되기 전에 다른 프로세스가 공유메모리를 접근한 경우
 - 동일하면 정상적으로 처리
- 실행 결과

```
$ shmbusyaccess 12345
Error(count=9975) : 다른 프로세스도 동시에 공유메모리 접근함
Error(count=31642) : 다른 프로세스도 동시에 공유메모리 접근함
Error(count=43453) : 다른 프로세스도 동시에 공유메모리 접근함
...
```

8.5 세마포어

- 동기화 문제를 소개
- 세마포어를 이용하여 동기화문제를 해결하는 방법

세마포어

- 프로세스간 통신에서 발생할 수 있는 동기화 문제를 해결하기 위해 사용
- 동기화 문제 예

프로세스 A의 작업 :

```
{  
    printf("A: before increase x=%d\n", x);  
    x++;  
    printf("A: after increase x=%d\n", x);  
}
```

프로세스 B의 작업 :

```
{  
    printf("B: before decrease x=%d\n", x);  
    x--;  
    printf("B: after decrease x=%d\n", x);  
}
```

- x의 초기값이 3이고, 프로세스 A, B가 차례로 수행했을 때 결과

A: before increase x=3	
A: after increase x=4	// A는 x=4를 사용
B: before decrease x=4	
B: after decrease x=3	// B는 x=3을 사용

동기화문제 예(계속)

- A가 수행하는 중간($x++$ 를 수행하기 직전)에 B가 CPU 스케줄을 받아 x 를 접근한 경우의 결과

A: before increase	$x=3$	
B: before decrease	$x=3$	// B는 $x=3$ 를 사용
B: after decrease	$x=2$	// B는 $x=2$ 를 사용
A: after decrease	$x=3$	// A는 $x=3$ 을 사용

- A가 $x++$ 를 수행한 직후에 B가 CPU 스케줄을 받아 x 를 접근한 경우의 결과

A: before increase	$x=3$	
B: before decrease	$x=4$	// x 는 $x++$ 직후이므로 4
B: after decrease	$x=3$	// B는 $x=3$ 를 사용
A: after increase	$x=3$	// A는 $x=3$ 을 사용

세마포어의 정의

- 동기화 문제 해결방법
 - 공유데이터를 액세스하는 프로세스 수를 한 순간에 하나로 제한
 - 세마포어를 사용하여 제한할 수 있음
- 세마포어
 - 공유데이터에 대해 현재 사용 가능한 데이터 수를 나타낸다.
- 이진 세마포어 : 공유데이터가 한 개일 경우, 0과 1값을 사용
 - 동작 방법
 - 공유데이터에 접근하기 전에 세마포어 s 의 값을 확인
 - 1이면 s 를 0으로 변경하고 액세스
 - 0이면 대기
 - 액세스가 끝나면 s 값을 1로 다시 변경
- 카운터 세마포어 : 공유할 수 있는 데이터가 둘 이상일 경우
 - 공유데이터가 5개이면 세마포어 값은 최대 5를 가진다.
 - 세마포어가 4이면 4개의 공유데이터가 남아있음을 의미

semget(), 세마포어 생성

- semget()

- 세마포어를 생성할 때 사용하는 함수

```
int semget(key_t key, int nsems, int semflg);
```

- key : 세마포어를 구분하기 위한 키
 - 다른 프로세스는 이 키를 알아야 사용할 수 있음
 - nsems : 세마포어 집합을 구성하는 멤버의 수
 - 3으로 설정하면 3개의 세마포어 집합을 얻음
 - semflg : 세마포어 생성옵션, 세마포어 객체에 대한 접근 권한을 설정
- 세마포어 정보를 갖는 세마포어 객체를 생성하고 세마포어 객체 ID를 반환

- 세마포어 객체 : semid_ds 구조체

```
struct semid_ds {  
    struct ipc_perm sem_perm;           // 접근 허가 내용, ipc.h  
    time_t sem_otime;                   // 최근 세마포어 조작 시각  
    time_t sem_ctime;                   // 최근 변경 시각  
    struct sem *sem_base;               // 첫 세마포어 포인터  
    struct wait_queue *eventn;  
    struct wait_queue *eventz;  
  
    struct sem_undo *undo;              // 배열안에 있는 undo의 수  
    ushort sem_nsems;                  // 세마포어 멤버 수  
};
```


semflg : 세마포어 생성에 관한 옵션, 접근 권한을 설정

- IPC_CREAT
 - 같은 key값을 사용하는 세마포어가 존재하면 해당 객체에 대한 ID를 리턴
 - 같은 key값의 세마포어가 존재하지 않으면 새로운 세마포어를 생성하고 그 ID를 리턴

```
semget(key, nsems, IPC_CREAT | mode);
```

- IPC_EXCL
 - 같은 key값을 사용하는 세마포어가 존재하면 semget() 호출은 실패하고 -1을 리턴
 - IPC_CREAT와 같이 사용해야 함

```
mode = 0660;  
semget(key, nsems, IPC_CREAT | IPC_EXCL | mode);
```

- IPC_PRIVATE
 - key값이 없는 세마포어를 생성
 - 명시적으로 key값을 사용할 필요가 없는 경우에 사용
 - 세마포어 생성 후 fork()를 호출하면 자식 프로세스는 세마포어 객체의 ID를 상속받으므로 접근이 가능
 - 부모와 자식 프로세스간의 통신을 위해 편리하게 사용

세마포어 연산

- 세마포어의 값을 변경(증가 또는 감소하는 것)
- semop() 함수 사용

```
int semop(int semid, struct sembuf *operations, unsigned nsops);
```

- semid : 세마포어 ID
- operations : 세마포어를 변경시킬 값이 있는 구조체
 - sembuf 구조체 사용
- nsops : operations가 몇 개의 리스트를 가지는지 명시
- sembuf 구조체

```
struct sembuf {  
    short sem_num;           // 멤버 세마포어 번호 (0번이 첫 번째 멤버)  
    short sem_op;            // 세마포어 연산 내용  
    short sem_flg;           // 조작 플래그  
};
```

sembuf 구조체

- sem_num
 - 세마포어 집합 중 몇 번째 멤버 세마포어를 연산할지 구분하는 번호
 - 첫 번째 멤버 세마포어일 경우 0을 사용
- sem_op
 - 증가 또는 감소 시킬 값 : 1증가시키려면 +1, 3감소시키려면 -3
- sem_flg
 - 0, IPC_NOWAIT, SEM_UNDO 등을 bitmask 형태로 취함
 - 0
 - 디폴트인 블록킹 모드로 semop()를 수행
 - 어떤 세마포어 값을 N만큼 감소시키려 할 때 현재 값이 N-1이하면 세마포어 값이 N 이상이 될 때까지 블록
 - IPC_NOWAIT
 - 세마포어 값이 부족해도 블록되지 않고 리턴
 - 리턴값은 -1이고 에러코드는 EAGAIN
 - SEM_UNDO
 - 프로세스의 종료 시 커널은 해당 세마포어 연산을 취소
 - 어떤 프로세스가 세마포어 값을 감소한 후 바로 비정상적으로 종료되었을 경우 원래의 값으로 환원할 기회를 얻게 되며 이는 다른 프로세스가 공유데이터를 계속 사용할 수 없게 함

세마포어 사용 예

- 연필 3자루, 노트 3권의 공유 데이터가 있고, 각 프로세스는 연필 1자루와 공책 1권을 동시에 사용하여 작업한다.
 - 현재 10개의 프로세스가 실행 중

```
//세마포어 값을 증감시키기 위한 sembuf 구조체 정의
struct sembuf increase[] = {{0, +1, SEM_UNDO}, {1, +1, SEM_UNDO}};
struct sembuf decrease[] = {{0, -1, SEM_UNDO}, {1, -1, SEM_UNDO}};

//nsems=2 : 연필(0)과 노트(1)에 대해 각각의 세마포어를 생성
semid = semget(mykey, 2, IPC_CREAT | mode);
...
// 세마포어 값 감소
semop(semid, &decrease[0], 1) // 연필
semop(semid, &decrease[1], 1) // 노트
// 연필과 노트를 사용하는 작업 기술
....
// 세마포어 값 증가
semop(semid, &increase[0], 1) // 연필
semop(semid, &increase[1], 1) // 노트
```

세마포어 제어

- 세마포어의 사용 종료, 세마포어 값 읽기 및 설정, 특정 멤버 세마포어를 기다리는 프로세스의 수를 확인할 때 사용하는 함수

```
int semctl(int semid, int member_index, int cmd, union semun semarg);
```

- semid : **제어할 대상의 세마포어 ID**
- member_index : **세마포어 멤버 번호**
- cmd : **수행할 동작**
- semarg : cmd의 종류에 따라 다르게 사용되는 인자 (semun 타입)

```
union semun {  
    int val;                // SETVAL을 위한 값  
    struct semid_ds buf;    // IPC_STAT, IPC_SET을 위한 버퍼  
    unsigned short int *array; // GETALL, SETALL을 위한 배열  
    struct seminfo *__buf;    // IPC_INFO를 위한 버퍼  
    void *__pad;             // dummy  
};
```

- IPC_STAT

- **semid_ds 타입의 세마포어 객체를 얻어오는 명령**
- union semun semarg **인자에 세마포어 객체가 리턴**
- member_index **인자는 사용되지 않음(0)**

```
struct semid_ds semobj;  
union semun semarg;  
semarg.buf = &semobj;  
semctl(semid, 0, IPC_STAT, semarg);
```

- SETVAL

- **공유데이터의 수를 설정해 주는 작업(세마포어 객체 초기화)**
- semarg에 **설정할 값을 지정하여 semarg.val 변수에 입력**

```
union semun semarg;  
unsigned short semvalue = 10;  
semarg.val = semvalue;  
semctl(semid, 2, SETVAL, semarg); // 2번 세마포어 값을 10으로 설정
```

- SETALL
 - 세마포어 집합 내의 모든 세마포어의 값을 초기화하는 명령
 - semarg.array 인자에 초기 값을 배열 형태로 지정
 - member_index는 사용하지 않음

```
unsigned short values = {3, 14, 1, 25};  
union semun semarg;  
semarg.array = values;  
semctl(semid, 0, SETALL, semarg); //4개의 세마포어의 값을 초기화
```

- GETVAL
 - 특정 멤버 세마포어의 현재 값을 얻어오는 명령

```
int n = semctl(semid, 2, GETVAL, 0); // 2번 세마포어 값을 구함
```

- GETALL

- 모든 멤버 세마포어의 현재 값을 읽는데 사용

```
union semun semarg;  
unsigned short semvalues[3]; // 세마포어의 수는 3  
semarg.array = semvalues;  
semctl(semid, 0, GETALL, semarg);  
for (i=0; i<3; i++)  
    printf("%d번 멤버 세마포어의 값 = %d\n", i, semvalue[i]);
```

- GETNCNT

- 세마포어 값이 원하는 값 이상으로 증가되기를 기다리는 프로세스의 수를 얻는데 사용
- 세마포어를 기다리는 프로세스의 수를 반환
- union semun semarg 인자는 사용 안함(0)

```
semctl(semid, member_index, GETNCNT, 0);
```


- GETPID
 - 특정 멤버 세마포어에 대해 **마지막으로 semop() 함수를 수행한 프로세스의 PID를 얻는데 사용**

```
int pid = semctl(semid, member_index, GETPID, 0);
```

- IPC_RMID
 - 세마포어를 **삭제하는 명령**

```
semctl(semid, 0, IPC_RMID, 0);
```

세마포어 예제 (pen_and_note.c)

- 연필 한 자루와 노트 두 권을 4개의 프로세스가 연필 한 자루와 노트 한 권을 동시에 사용하는 프로그램
- 실행 결과

```
$ pen_and_note
[pid: 18178] 연필을 들고
    [pid: 18178] 노트를 들고
    [pid: 18178] 공부를 함
    [pid: 18178] 연필을 내려놓음
    [pid: 18178] 노트를 내려놓음
[pid: 18177] 연필을 들고
    [pid: 18177] 노트를 들고
    [pid: 18177] 공부를 함
    [pid: 18177] 연필을 내려놓음
    [pid: 18177] 노트를 내려놓음
. . .
```

공유메모리의 동기화문제 처리(shmcontrol.c)

- 세마포어를 이용해 공유메모리의 동기화를 해결
 - 공유메모리 사용 시 동기화문제가 있었던 shmbusyaccess.c를 세마포어를 사용하여 해결
 - 공유메모리 접근함수 access_shm()에서 세마포어를 이용하여 critical 영역으로 보호하여 두 프로세스가 access_shm()을 동시에 호출 못하게 한다.
 - access_shm() 호출 전후에 세마포어를 사용하기 위하여 semop()를 호출
- 실행 결과
 - “.”은 정상적으로 공유메모리에 접근하고 있다는 것을 나타낸다.

```
$ shmcontrol 1234 2345
```

```
.....  
.....  
.....  
.....
```