한림대학교 | 소프트웨어융합대학
HALLYM UNIVERSITY

Operating Systems LAB 9

Wonpyo Kim
skykwp@gmail.com

Outline

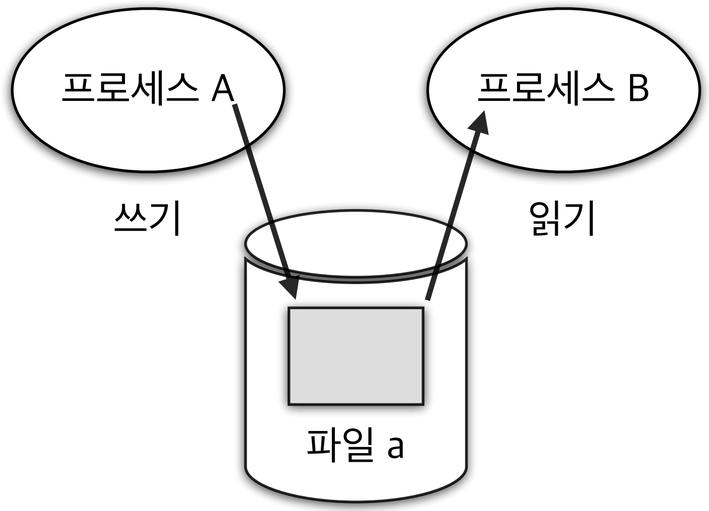
- Substance
  - **File Communication and PIPE**
  - **Message Queue**

Record Locking

- 여러 프로그램이 동시에 같은 파일을 이용하면 문제가 발생할 수 있는데, 이는 파일을 이용한 통신 기법을 통해 해결할 수 있다.
- 파이프를 이용해 여러 프로세스가 데이터를 주고 받을 수 있게 하면 더 효율적인 결과를 얻을 수 있다.
- 파일을 이용한 통신이 제대로 이루어지기 위해서는 레코드 잠금이 필요하다.

Record Locking

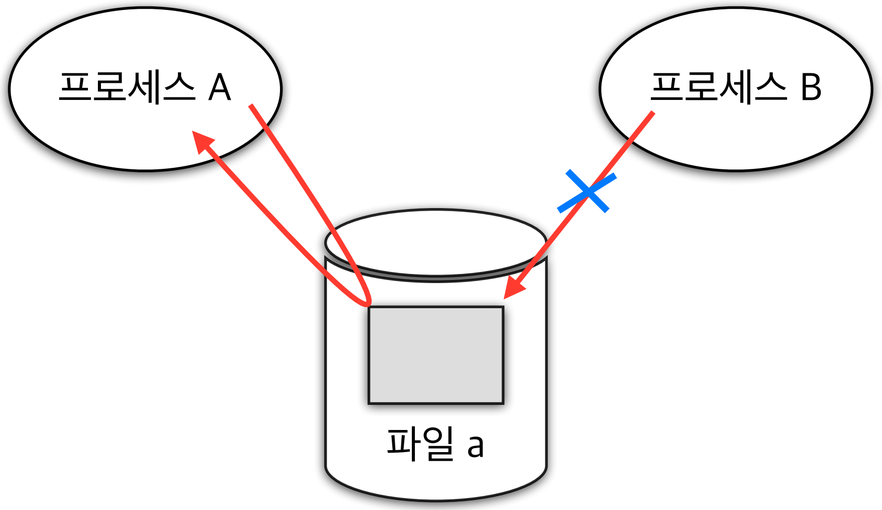
- 프로세스 간에 정보를 주고받는 기본적인 방법은 파일을 이용하는 것이다. 프로세스 A가 파일 a에 저장한 데이터를 프로세스 B가 읽으면 통신(communication)이 이루어진 것이다.



- 이 때, 여러 프로세스가 같은 파일을 동시에 이용하면 문제가 발생할 수 있는데, 이를 해결하는 대표적인 방법이 "레코드 잠금"이다.

Record Locking

- "레코드 잠금(record locking)" 은 두 개 이상의 프로세스가 하나의 파일(또는 파일의 일부)을 일정 시점을 기준으로 오직 하나의 프로세스만 파일을 이용할 수 있게 하는 것으로 fcntl( ) 과 lockf( ) 함수로 구현된다.



Record Locking

- lockf( ) 함수는 fcntl( ) 함수보다 기능이 떨어져 최근에는 사용되지 않는다. 따라서, fcntl( ) 함수를 살펴본다.

```
fcntl 함수
기능
        파일을 제어한다
기본형
        int fcntl(int fd, int cmd, struct flock *lock);
        fd: 제어할 파일의 파일 식별자
        cmd: 동작 지정
        lock: 레코드 잠금에서 사용되는 인수
반환값
        성공: cmd에 따라 달라짐
        실패: -1
헤더파일
        <unistd.h>
        <fcntl.h>
```

cmd	의미
F_GETLK	레코드 잠금 정보를 얻는다. 세 번째 인수인 lock에 레코드 잠금 정보가 저장된다.
F_SETLK	레코드 잠금을 설정한다. 다른 프로세스에 의해 레코드가 잠겨 동작이 불가능하면 즉시 실패로 -1을 반환한다.
F_SETLKW	레코드 잠금을 설정한다. 다른 프로세스에 의해 레코드가 잠겨 동작이 불가능하면 가능할 때까지 기다린다.

Record Locking

- 세 번째 인수인 lock 의 데이터형은 struct flock 로 <fcntl.h> 에서 다음과 같이 정의한다.

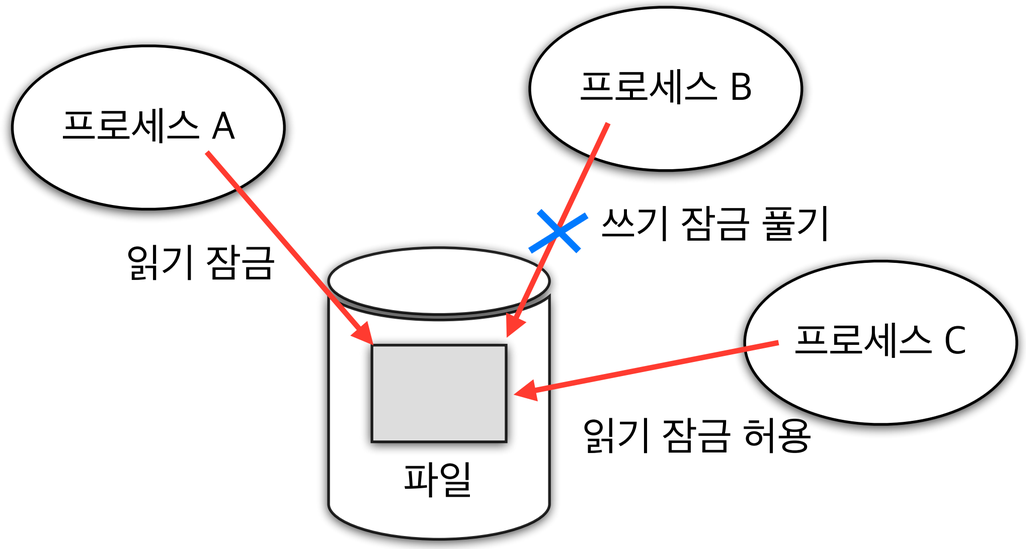
```
struct flock
  {
    short int l_type;   /* Type of lock: F_RDLCK, F_WRLCK, or F_UNLCK.  */
    short int l_whence; /* Where `l_start' is relative to (like `lseek').  */
#ifndef __USE_FILE_OFFSET64
    __off_t l_start;    /* Offset where the lock begins.  */
    __off_t l_len;  /* Size of the locked area; zero means until EOF.  */
#else
    __off64_t l_start;  /* Offset where the lock begins.  */
    __off64_t l_len;    /* Size of the locked area; zero means until EOF.  */
#endif
    __pid_t l_pid;  /* Process holding the lock.  */
  };
```

- l_type 은 잠금의 유형을 나타내며, <fcntl.h>에 정의된 다음과 같은 값을 취할 수 있다.

l_type	의미
F_RDLCK	읽기 잠금을 설정한다.
F_WRLCK	쓰기 잠금을 설정한다.
F_UNLCK	잠금을 해제한다.

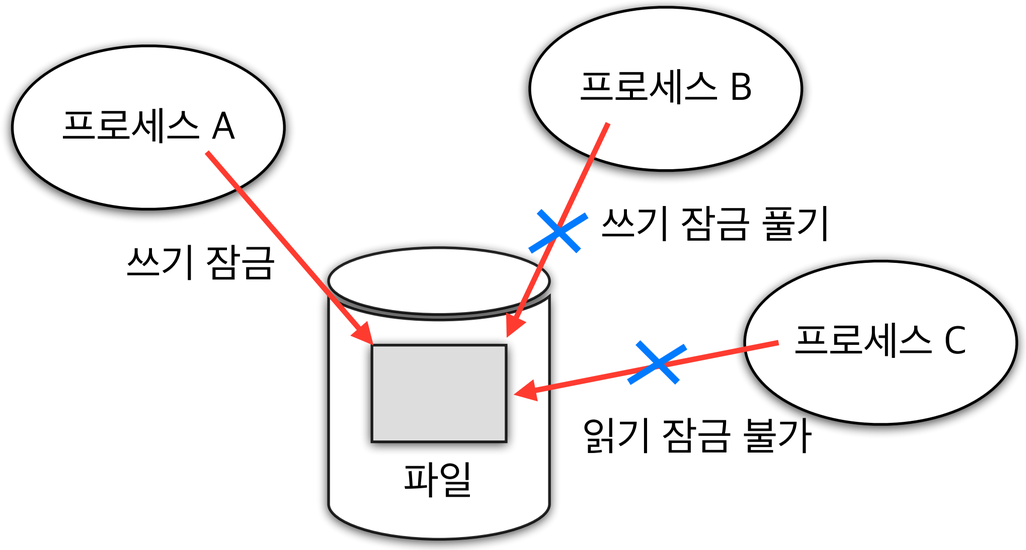
Record Locking

- 잠금은 읽기 잠금과 쓰기 잠금으로 구분되며, 읽기 잠금을 다른 프로세스의 쓰기 잠금은 막고 읽기 잠금은 허용한다. 즉, 다른 프로세스가 파일 내용을 볼 수는 있지만 갱신하지 못한다.



Record Locking

- 쓰기 잠금은 다른 프로세스의 모든 잠금을 막아 다른 프로세스가 파일을 읽고 쓰지 못하게 한다.



Record Locking

- l_whence, l_start, l_len 을 통해 잠금을 설정하거나 해제할 영역을 지정하는데, l_whence 는 lseek 에서의 whence 와 동일하게 어디를 기준으로 할지를 나타내는 것으로 다음과 같은 값이 있다.

l_whence	의미
SEEK_SET	파일의 시작을 기준으로 한다.
SEEK_CUR	현재 읽기/쓰기 포인터가 가리키는 부분을 기준으로 한다.
SEEK_END	파일의 끝을 기준으로 한다.

- l_start 는 영역의 시작 위치를 나타내는 것으로 l_whence 의 상대 위치다. l_len 은 바이트 단위로 표시한 영역의 길이로 이 값을 0으로 하면 파일의 끝까지를 나타낸다.

Record Locking

- 다음 예제는 프로세스 A가 특정 파일에 읽기 또는 쓰기 잠금을 설정하고 프로세스 B가 동일한 파일에 대해 읽기 또는 쓰기 잠금을 시도하는 과정이다.

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>

main(int argc, char *argv[])
{
    int fd;
    struct flock filelock;

    filelock.l_type = (!strcmp(argv[1], "r")) ? F_RDLCK:F_WRLCK;
    filelock.l_whence = SEEK_SET;
    filelock.l_start = 0;
    filelock.l_len = 0;

    fd = open(argv[2], O_RDWR | O_CREAT, 0666);

    if (fcntl(fd, F_SETLK, &filelock) == -1) {
        perror("fcntl failed");
        exit(1);
    }

    printf("locked %s\n", argv[2]);
    sleep(30);
    printf("unlocked %s\n", argv[2]);

    exit(0);
}
```

- 파일명은 proc_a.c 로 하고 다음과 같이 백그라운드로 실행한다.

  $ proc_a r file &

Record Locking

- 프로세스 B를 다음과 같이 생성한다.

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>

main(int argc, char *argv[])
{
    int fd;
    struct flock filelock;

    filelock.l_type =  (!strcmp(argv[1], "r")) ? F_RDLCK:F_WRLCK;
    filelock.l_whence = SEEK_SET;
    filelock.l_start = 0;
    filelock.l_len = 0;

    fd = open(argv[2], O_RDWR);

    if (fcntl(fd, F_SETLK, &filelock) == -1) {
        perror("fcntl failed");
        exit(1);
    }

    printf("success\n");

    exit(0);
}
```

- 파일명은 proc_b.c 로 하고 다음과 같이 실행한다.

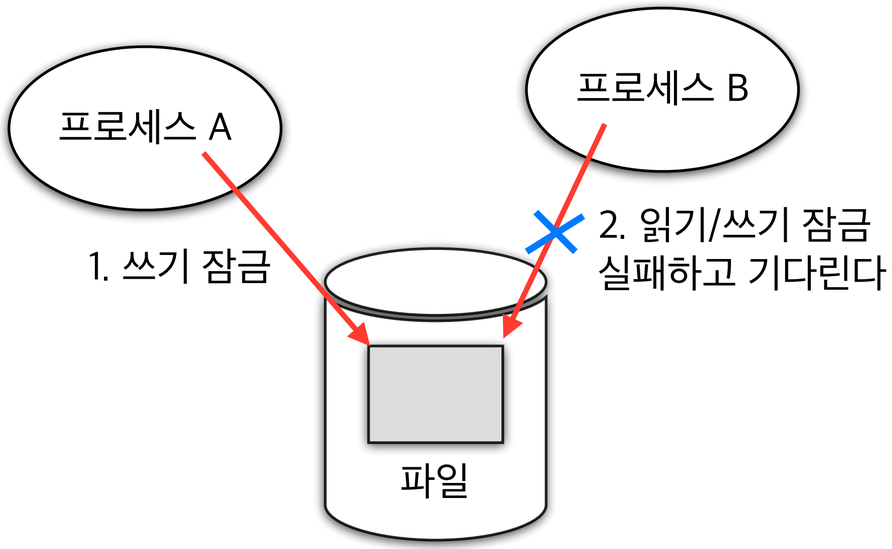
  $ proc_b r file

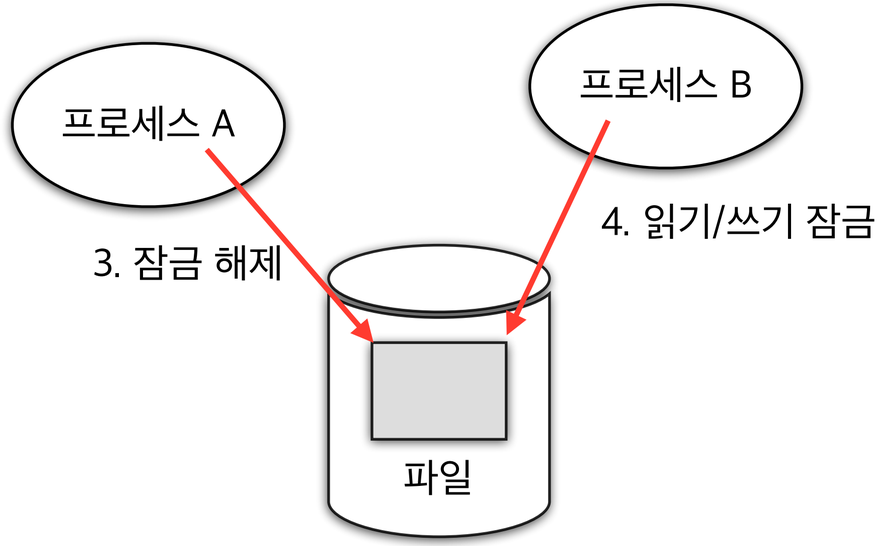
- 실행하면, 결과로 프로세스 A에 의해 읽기 잠금으로 설정되어 있는 file 에 대해 읽기 잠금이 성공했음을 의미하는 success 가 출력된다.

Record Locking

- fcntl( ) 함수에서 레코드 잠금을 설정하는 cmd 값은 F_SETLK 와 F_SETLKW 두 가지가 있는데, F_SETLK 는 레코드가 다른 프로세스에 의해 이미 잠겨 실패했을 때 실패를 반환하는 반면 F_SETLKW 는 성공할 때까지 기다린다.

- F_SETLK 와 F_SETLKW 의 사용법에는 큰 차이가 없다. 다음은 F_SETLKW 의 예제이다.





Record Locking

- fcntl( ) 함수에서 레코드 잠금을 설정하는 cmd 값은 F_SETLK 와 F_SETLKW 두 가지가 있는데, F_SETLK 는 레코드가 다른 프로세스에 의해 이미 잠겨 실패했을 때 실패를 반환하는 반면 F_SETLKW 는 성공할 때까지 기다린다.

- F_SETLK 와 F_SETLKW 의 사용법에는 큰 차이가 없다. 다음은 F_SETLKW 의 예제이다.

```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>

main(int argc, char *argv[])
{
    int fd;
    struct flock filelock;

    filelock.l_type = F_WRLCK;
    filelock.l_whence = SEEK_SET;
    filelock.l_start = 0;

    fd = open(argv[1], O_RDWR);

    switch (fork()) {
        case -1:
            perror("fork failed");
            exit(1);
        case 0:
            filelock.l_len = 5;
            if (fcntl(fd, F_SETLKW, &filelock) == -1) {
                perror("fcntl failed");
                exit(1);
            }
            printf("Child process: locked\n");
            sleep(3);
            filelock.l_type = F_UNLCK;
            if (fcntl(fd, F_SETLK, &filelock) == -1) {
                perror("fcntl failed");
                exit(1);
            }
            printf("Child process: unlocked\n");
            break;
        default:
            filelock.l_len = 10;
            if (fcntl(fd, F_SETLKW, &filelock) == -1) {
                perror("fcntl failed");
                exit(1);
            }
            printf("Parent process: locked\n");
            sleep(3);
            filelock.l_type = F_UNLCK;
            if (fcntl(fd, F_SETLK, &filelock) == -1) {
                perror("fcntl failed");
                exit(1);
            }
            printf("Parent process: unlocked\n");
            wait(NULL);
    }
    exit(0);
}
```

- 실행은 매개변수로 file 을 입력한다. 부모와 자식 프로세스가 각각 락을 거는 것을 확인할 수 있다.

  ex) $ a.out file

Outline

- Substance
  - **File Communication and PIPE**
  - **Message Queue**

Pipe

- 리눅스 쉘에서 다음과 같이 명령어와 명령어를 연결하는 파이프(|)를 볼 수 있었을 것이다.

  $ cat file | wc

- 이 명령의 의미를 cat file 의 출력 내용을 화면으로 출력하지 않고 wc 의 입력으로 보내 wc 를 실행하는 것이다.

- 결국 파이프(pipe)는 한 프로세스의 출력을 다른 프로세스의 입력으로 연결시킴으로써 통신이 이루어지게 한다.

- 이러한 기능을 하는 함수가 pipe( ), popen( ) / pclose( ) 함수이다.

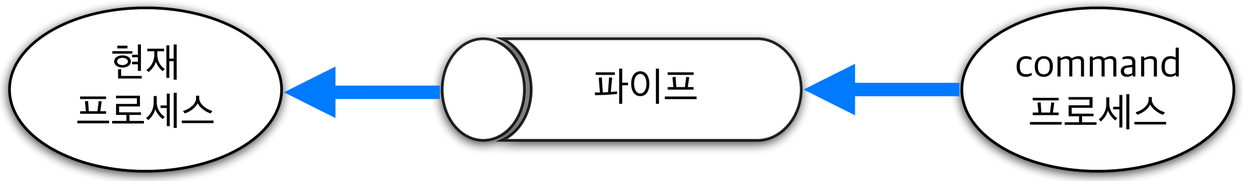
Pipe - popen( ), pclose( )

- popen( ) 과 pclose( ) 는 파이프를 간단하게 구현하는 함수로, popen( ) 함수는 파이프와 command 에 해당하는 새로운 프로세스를 생성하고 파이프를 이용해 데이터를 보내거나 받는다.

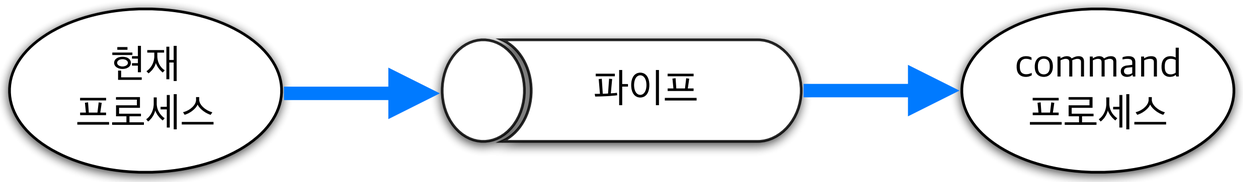
```
popen 함수
기능
        파이프를 이용해 명령어를 실행한다.
기본형
        FILE *popen(const char *command, const char *type);
        command: 실행할 명령어
        type: 통신 형태
반환값
        성공: 파일 포인터
        실패: NULL
헤더파일
        <stdio.h>
```

Pipe - popen( ), pclose( )

- 두 번째 인수인 type 은 "r" 또는 "w" 가 될 수 있는데, "r" 로 설정하면 command 의 출력을 파이프를 통해 입력받는다.



- "w" 로 설정하고 데이터를 파이프에 출력하면 command 프로세스의 입력으로 전달된다.



Pipe - popen( ), pclose( )

- popen( ) 함수에 의해 생성된 파이프의 사용이 끝났다면 닫는 것이 바람직한데, 이는 pclose( ) 함수가 담당하며 pclose( ) 호출에 성공하면 종료되는 프로세스의 종료 상태를 반환한다.

```
pclose 함수
기능
        파이프를 닫는다.
기본형
        int pclose(FILE *stream);
        stream: 닫을 파이프의 파일 포인터
반환값
        성공: 종료 상태
        실패: -1
헤더파일
        <stdio.h>
```

Pipe - popen( ), pclose( )

- 예제

```c
#include <stdio.h>

main(int argc, char *argv[])
{
    FILE *read_fp;
    char buffer[256];

    if ((read_fp = popen(argv[1], "r")) == NULL) {
        perror("popen failed");
        exit(1);
    }

    while (fgets(buffer, sizeof(buffer), read_fp)) {
        fputs(buffer, stdout);
    }

    pclose(read_fp);
    exit(0);
}
```

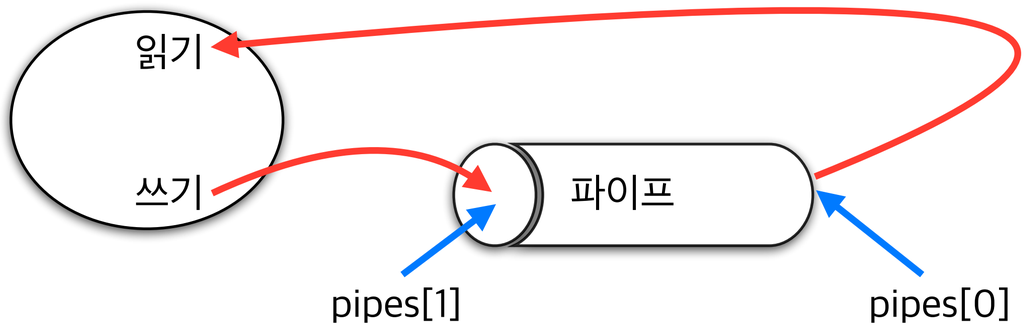
- 실행 예

```
s_guest@A1409-OSLAB-01:~/lab8$ gcc 25_7.c
s_guest@A1409-OSLAB-01:~/lab8$ ./a.out "uname -a"
Linux A1409-OSLAB-01 4.18.0-15-generic #16~18.04.1-Ubuntu SMP Thu F
s_guest@A1409-OSLAB-01:~/lab8$
```

Pipe - pipe( )

- 다음과 같이 pipe( ) 를 실행하면 pipes[0] 과 pipes[1] 의 두 개의 식별자와 함께 파이프가 생성된다.

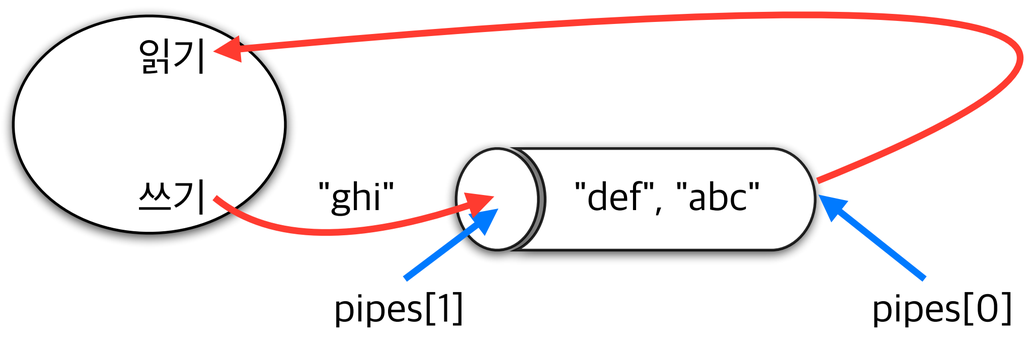
```
int pipes[2];
pipe(pipes);
```



Pipe - pipe( )

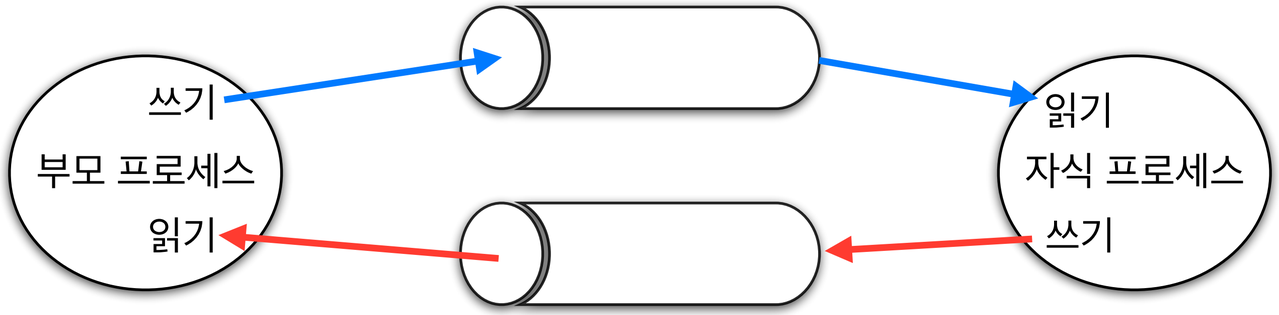
- 예제

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

#define SIZE 4

main()
{
    char *arg[3] = {"abc", "def", "ghi"};
    char buffer[SIZE];
    int pipes[2], i;

    if (pipe(pipes) == -1) {
        perror("pipe failed");
        exit(1);
    }

    for (i = 0; i < 3; i++) {
        write(pipes[1], arg[i], SIZE);
    }

    for (i = 0; i < 3; i++) {
        read(pipes[0], buffer, SIZE);
        printf("%s\n", buffer);
    }

    exit(0);
}
```



Pipe - pipe( )

- 앞선 예제는 단일 프로세스에서만 동작하는 예제이다. 따라서, 두 프로세스 간에 데이터를 주고 받으려면 다음과 같이 해야한다.



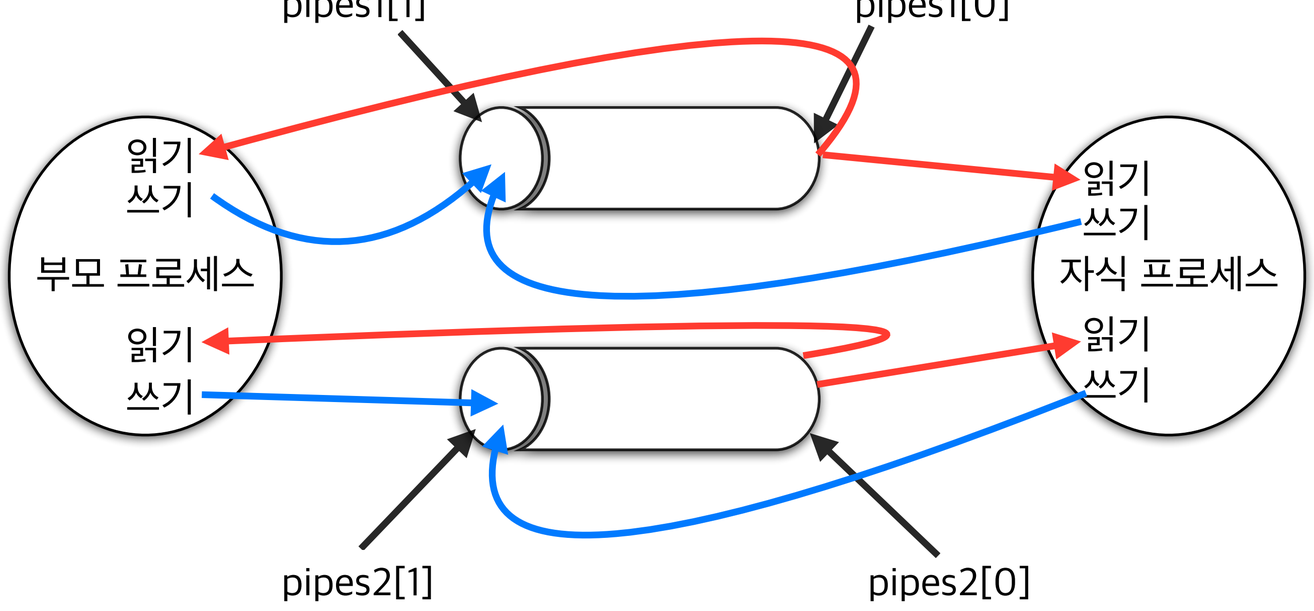
Pipe - pipe( )

- 예제

```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>

#define SIZE 256

main()
{
    int pipes1[2], pipes2[2], length;
    char read_buffer[SIZE], write_buffer[SIZE];

    if (pipe(pipes1) == -1 || pipe(pipes2) == -1) {
        perror("pipe failed");
        exit(1);
    }

    if (fork()) {
        close(pipes1[1]);
        close(pipes2[0]);

        if ((length = read(pipes1[0], read_buffer, SIZE)) == -1) {
            perror("read failed");
            exit(1);
        }
        write(STDOUT_FILENO, "receive message: ", strlen("receive message: "));
        write(STDOUT_FILENO, read_buffer, length);

        sprintf(write_buffer, "Hi Client!\n");
        write(pipes2[1], write_buffer, strlen(write_buffer));

        wait(NULL);

    } else {
        close(pipes1[0]);
        close(pipes2[1]);

        sprintf(write_buffer, "Hi Server!\n");
        write(pipes1[1], write_buffer, strlen(write_buffer));

        if ((length = read(pipes2[0], read_buffer, SIZE)) == -1) {
            perror("read failed");
            exit(1);
        }
        write(STDOUT_FILENO, "receive message: ", strlen("receive message: "));
        write(STDOUT_FILENO, read_buffer, length);
    }
    exit(0);
}
```

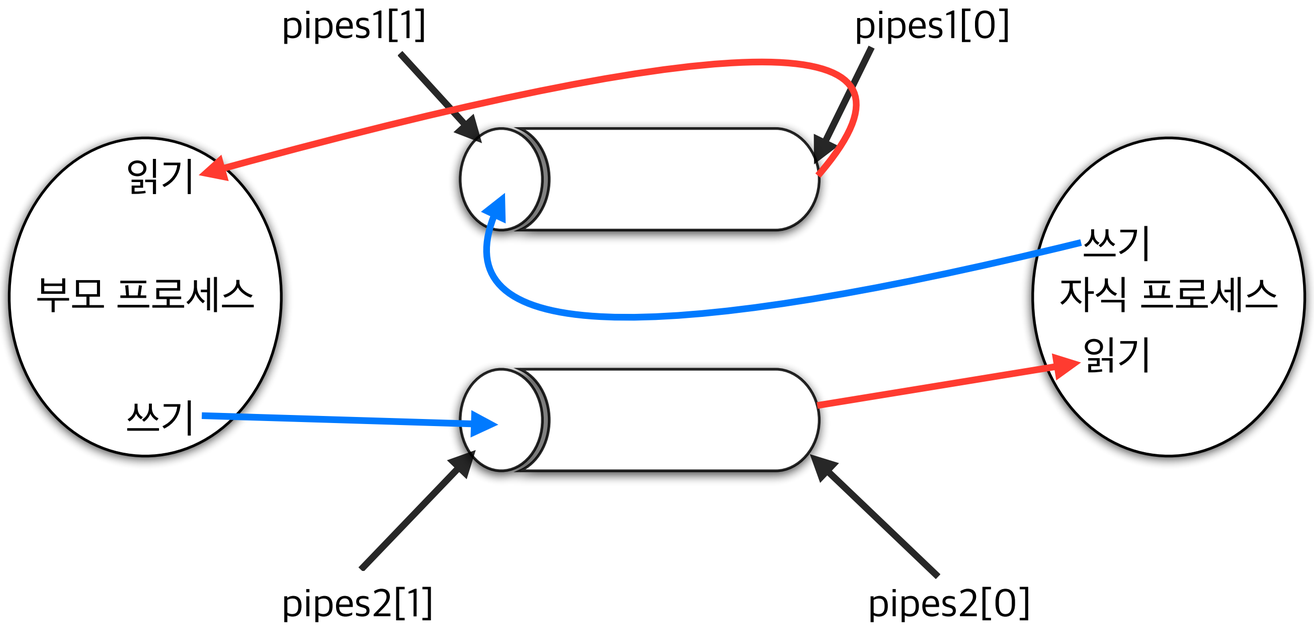
Pipe - pipe( )

- 예제 프로그램의 동작 과정은 다음과 같다.
- 두 개의 파이프를 생성한 상태에서 자식 프로세스를 생성한다.



Pipe - pipe( )

- 부모 프로세스는 pipes1[1] 과 pipes2[0] 을 제거하고 자식 프로세스는 pipes1[0] 과 pipes2[1] 을 제거한다.

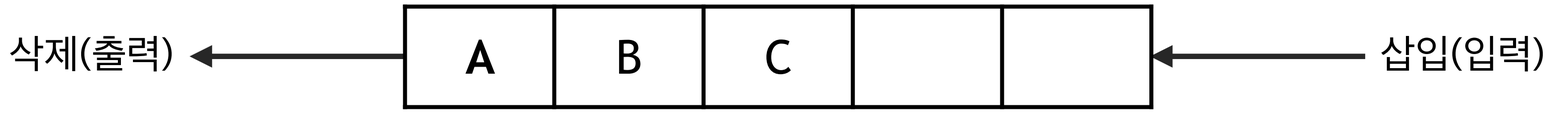


Outline

- Substance
  - **File Communication and PIPE**
  - **Message Queue**

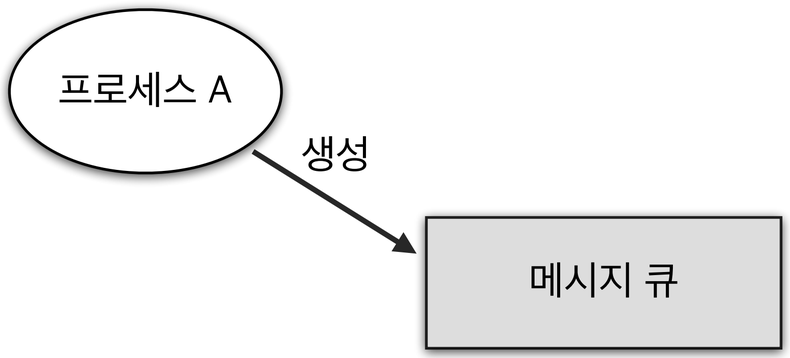
Message Queue

- 큐(queue)는 한 쪽 끝에서 데이터가 삽입(입력)되고 다른 끝에서는 데이터가 삭제(출력)되는 데이터 구조이며, 메시지(message)는 프로세스들 간에 주고 받는 문자열로 이루어진 데이터이다.



- 프로세스들은 이런 메시지 큐를 이용해 통신을 할 수 있다. 그 기본 동작을 예시를 통해 살펴보면 다음과 같다.

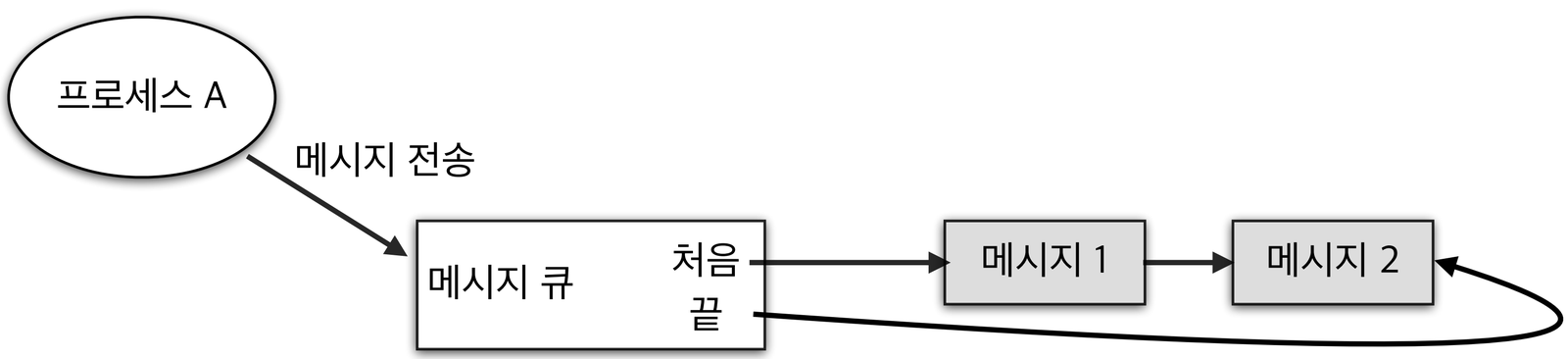
  (1) 메시지 큐 생성: 임의의 프로세스 A가 메시지 큐를 생성한다.



Message Queue

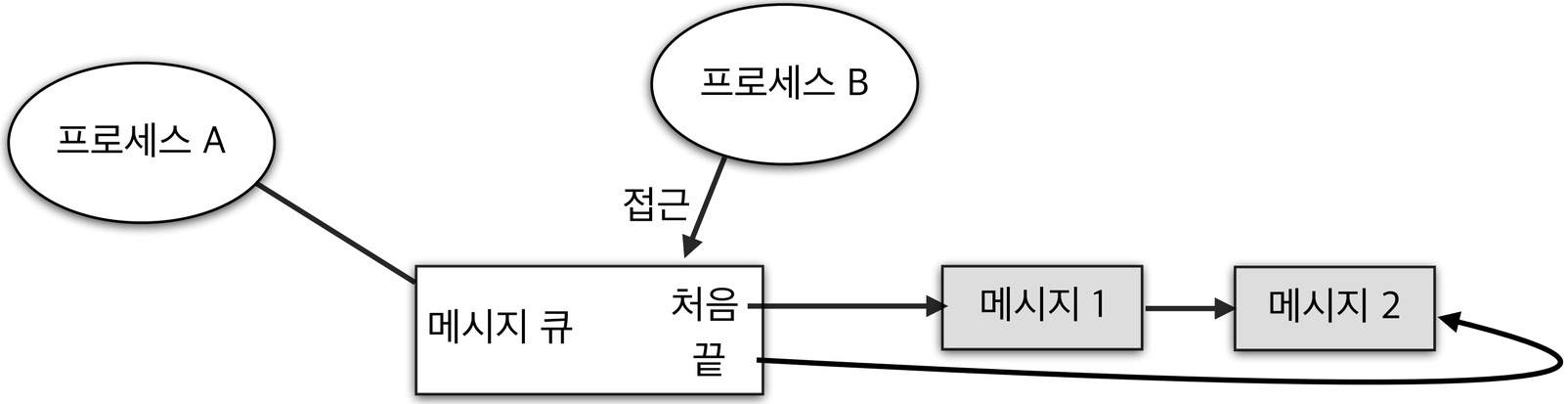
(2) 메시지 큐에 메시지를 전송: 프로세스 A가 메시지 큐에 메시지1 을 전송하면 메시지 큐는 메시지를 연결 리스트에 저장한다.

(3) 프로세스 A가 메시지2 를 전송하면 메시지 큐는 이 메시지도 연결 리스트에 저장한다.

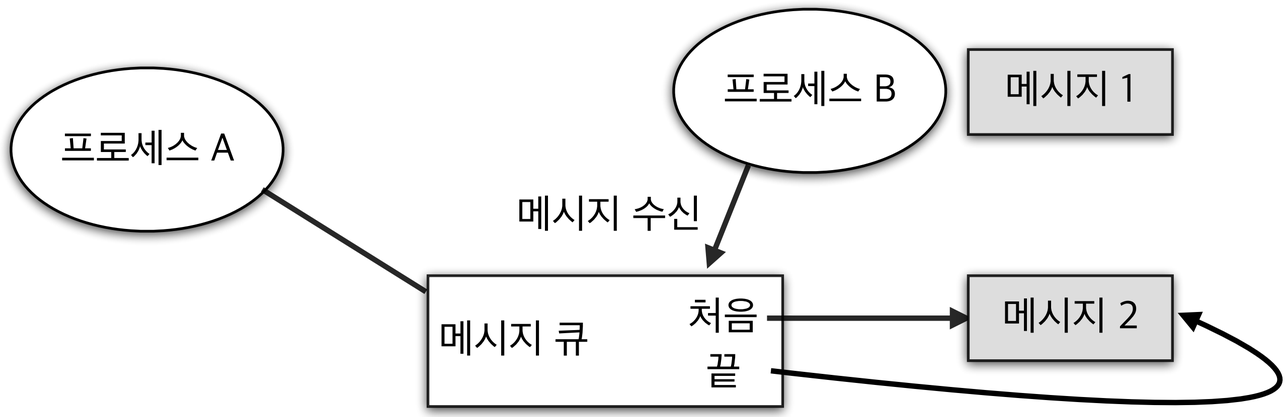


Message Queue

(4) 프로세스 B가 메시지 큐에 접근한다.



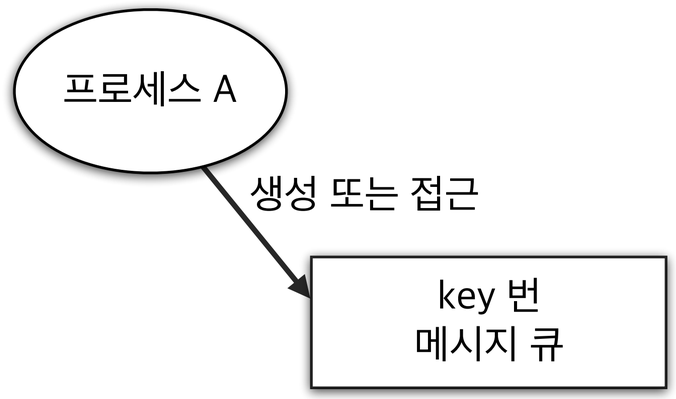
(5) 프로세스 B가 메시지 큐에 있는 메시지를 수신할 때는 기본적으로 첫 번째 메시지를 수신하고 큐에 있는 메시지를 삭제한다.



Message Queue

- 메시지 큐를 이용하기 위해서는 메시지 큐를 생성하거나 기존에 생성된 메시지 큐에 접근해야 한다.
- 이 때는 msgget( ) 함수를 이용한다. msgget( ) 함수가 key 라는 키를 갖는 메시지 큐를 생성하거나 기존 메시지 큐에 접근하여 메시지 큐 식별자인 정수값을 반환하면, 이 식별자를 다른 메시지 큐 함수가 이용하는 것이다.

msgget 함수
기능
　　메시지 큐를 생성한다.
기본형
　　int msgget(key_t key, int msgflg);
　　key: 시스템에서 식별하는 메시지 큐 번호
　　msgflg: 동작 옵션
반환값
　　성공: 메시지 큐 식별자
　　실패: -1
헤더파일
　　<sys/types.h>
　　<sys/ipc.h>
　　<sys/msg.h>



Message Queue

- 첫 번째 인수인 key 는 시스템에서 식별하는 메시지 큐 번호로 프로세스가 이 key 로 메시지 큐를 공유해 사용한다.

- 예를 들어, 1234번 메시지 큐를 생성하고 식별자를 반환하는데, 만약 시스템에 1234번 메시지 큐가 이미 존재할 경우 기존의 메시지 큐에 접근해 식별자를 반환한다.

```
msqid = msgget((key_t)1234, ... );
```

- 이 key 값은 시스템 전체 영역에서 사용하므로 경우에 따라서는 다른 프로세스에서 생성해서 사용하는 key와 충돌할 수 있다. 따라서, 이런 충돌을 피하기 위해 특수한 IPC_PRIVATE 라는 키워드를 사용해 값을 설정하면 유일한 메시지 큐를 생성한다.

```
msqid = msgget(IPC_PRIVATE, ... );
```

Message Queue

- 두 번째 인수인 msgflg 에 의해 msgget( ) 이 수행할 정확한 동작을 정할 수 있다. 관련 값은 다음과 같고 독립 혹은 OR 연산을 통해 함께 사용할 수 있다.

msgflg	의미
IPC_CREAT	key에 해당하는 메시지 큐가 없으면 큐를 새로 생성하는데, 이 때 접근 권한도 함께 부여해야 한다. 그러나, 메시지 큐가 이미 존재하면 이 옵션은 무시한다.
IPC_EXCL	메시지 큐가 있으면 실패하라 라는 의미로 -1을 반환한다. 이 값이 설정되지 않으면 기존 메시지 큐에 접근해 식별자를 반환한다.

- 다음은 IPC_CREAT를 사용하는 예로 1234 키를 갖는 메시지 큐가 없으면 생성하고 식별자를 반환하는데, 생성된 메시지 큐의 접근 권한은 0666이 된다. 그러나 1234 키를 갖는 메시지 큐가 있다면 접근하여 식별자를 반환한다.

```
msqid = msgget((key_t)1234, IPC_CREAT | 0666);
```

Message Queue

- 다음은 IPC_EXCL 을 사용한 경우로 1234 키를 갖는 메시지 큐가 없으면 생성하지만, 이미 있다면 실패로 -1을 반환한다.

```
msqid = msgget((key_t)1234, IPC_CREAT | IPC_EXCL | 0666);
```

- msgflg 에 다음과 같이 값을 주지 않을 수도 있는데, 이는 기존 메시지 큐를 이용하겠다는 의미이며, 메시지 큐가 없으면 실패로 -1 을 반환한다.

```
msqid = msgget((key_t)1234, 0);
```

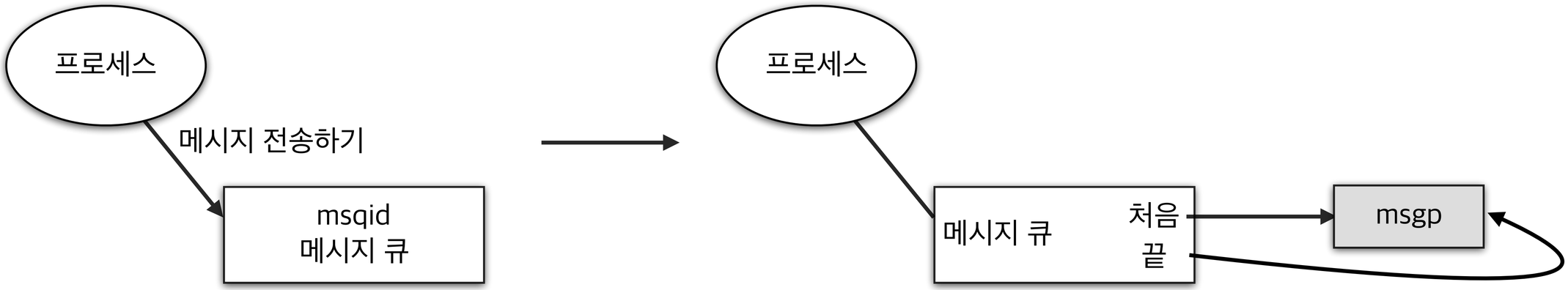
Message Queue

- 메시지 큐에 대한 식별자를 받아 이용할 수 있게 되면 메시지를 보내야 한다. 이 때, 사용하는 함수는 msgsnd( ) 이다.

msgsnd 함수
기능
　　메시지 큐에 메시지를 전송한다.
기본형
　　int msgsnd(int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg);
　　msgid: 메시지 큐 식별자
　　msgp: 전송할 메시지
　　msgsz: 메시지 크기
　　msgflg: 동작 옵션
반환값
　　성공: 0
　　실패: -1
헤더파일
　　<sys/types.h>
　　<sys/ipc.h>
　　<sys/msg.h>

Message Queue

- msgsnd( ) 함수는 메시지 큐에 크기가 msgsz 인 msgp 메시지를 전송하고 다음과 같이 전송에 성공하면 msgqid 메시지 큐에 메시지가 연결된다.



- 첫 번째 인수인 msqid 는 메시지를 전송하기 위한 메시지 큐의 식별자로 msgget( ) 함수가 반환하는 값이다.

```
msqid = msgget( ... );
msgsnd(msqid, ... );
```

Message Queue

- 두 번째 인수인 msgp 는 메시지를 저장하는 변수로 기본적인 데이터형은 다음과 같다.

```
#ifdef __USE_GNU
/* Template for struct to be used as argument for `msgsnd' and `msgrcv'.  */
struct msgbuf
  {
    __syscall_slong_t mtype;    /* type of received/sent message */
    char mtext[1];      /* text of the message */
  };
#endif
```

- 첫 번째 멤버인 mtype 에 메시지의 형식을 저장하고, 두 번째 멤버인 mtext 에 데이터를 저장한다. 위 형식은 전반적인 구성만을 나타낼 뿐 실제 프로그래밍에서는 다음과 같이 사용자가 임의로 만들어서 쓸 수 있다.

- 단, 첫 번째 멤버는 반드시 메시지 형식이어야 한다.

```
struct my_msg {
    long type;
    char data[100];
};
```

Message Queue

```
struct my_msg {
    long type;
    char data[100];
};
```

- 만약 위와 같은 메시지를 사용한다면, msgsz (메시지 사이즈)는 다음과 같은 계산으로 구할 수 있다.

```
msgsz = sizeof(struct my_msg) - sizeof(long);
```

- 세 번째 인수인 msgflg 는 다음과 같고 0으로 설정할 수도 있다.

msgflg	의미
IPC_NOWAIT	메시지 큐가 가득 차 있어 메시지를 더 이상 저장할 수 없을 경우에 실패로 -1을 반환한다. 이 값을 설정하지 않으면 메시지 큐가 메시지를 저장할 수 없을 때까지 기다린다.

- 예시)

```
struct msgbuf {
    long mtype;
    char mtext[100];
} msg_data = {1, "Hello?"};
msgsnd(msqid, &msg_data, strlen(msg_data.mtext), 0);
```

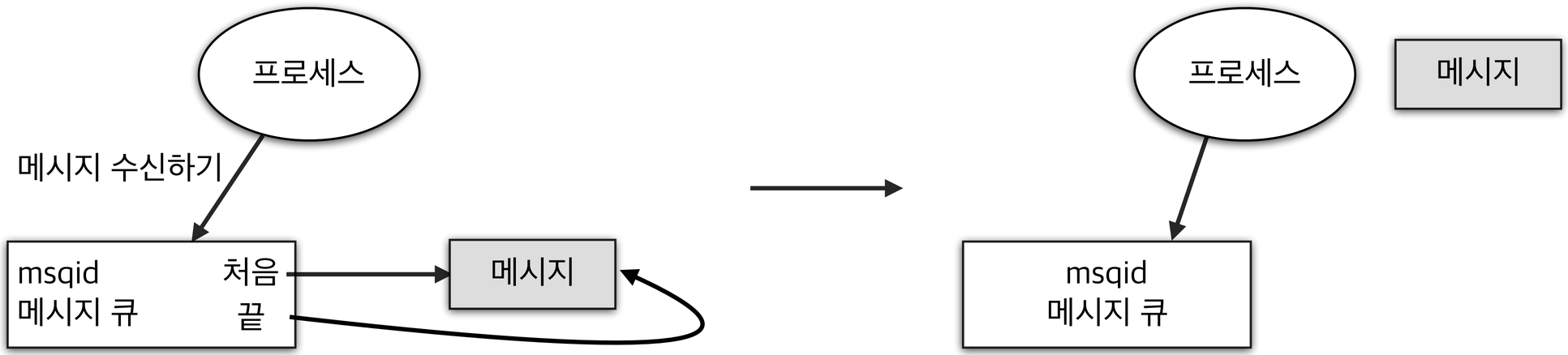
Message Queue

- 메시지 큐에 있는 메시지를 수신하는 함수는 msgrcv( ) 함수이다.

msgrcv 함수
기능
　　메시지 큐에 있는 메시지를 수신한다.
기본형
　　ssize_t msgrcv(int msqid, struct msgbuf *msgp, size_t msgsz, long msgtyp, int msgflg);
　　msqid: 메시지 큐 식별자
　　msgp: 메시지를 수신할 곳
　　msgsz: 수신 가능한 메시지 크기
　　msgtyp: 수신할 메시지 선택 조건
　　msgflg: 동작 옵션
반환값
　　성공: 수신한 메시지 크기
　　실패: -1
헤더파일
　　<sys/types.h>
　　<sys/ipc.h>
　　<sys/msg.h>

Message Queue

- msqid 메시지 큐로부터 메시지를 수신해 msgp 로 저장하고 수신된 메시지는 메시지 큐에서 제거된다.



- 함수에서는 다음과 같이 사용된다. 첫 번째 인수인 msqid 는 메시지를 수신하기 원하는 메시지 큐의 식별자로, msgget 에 의해 얻어진다.

```
msqid = msgget( ... );
msgrcv(msqid, ... );
```

Message Queue

- 두 번째 인수인 msgp 는 수신한 메시지를 저장할 인수로 msgsnd 의 msgp 와 같다.

```
struct msgbuf {
    long mtype;
    char mtext[100];
} msgp;
```

- 세 번째 인수인 msgsz 는 수신 가능한 메시지의 크기로 msgsnd 의 msgsz 처럼 메시지 형식을 저장하는 멤버의 크기는 포함하지 않는다. 그러므로 위의 예제에서는 100이 된다.

- 네 번째 인수인 msgtyp 는 여러 메시지 중에서 어떤 메시지를 수신 할지를 결정하는 인수이다. 의미는 다음과 같다.

msgtyp	의미
0	메시지 큐에서 첫 번째 메시지를 수신한다.
양수	이 값과 메시지의 메시지 형식(mtype)이 일치하는 메시지를 수신한다.
음수	이 값의 절대값과 같거나 작은 메시지 형식(mtype)을 갖는 메시지 중에서 가장 작은 메시지 형식을 갖는 메시지를 수신한다. 예를 들어, 세 개의 메시지 형식이 각각 5, 1, 9라 하자. msgtype 가 -10이라면 -10의 절대값인 10보다 작거나 같은 메시지 형식을 갖는 메시지는 모두 해당된다. 이 중 가장 작은 1을 메시지 형식으로 갖고 있는 메시지를 수신한다.

Message Queue

- 다섯 번째 인수인 msgflg 는 다음과 같은 값 단독 혹은 OR 연산을 사용하여 사용한다. 0도 가능하다.

msgflg	의미
IPC_NOWAIT	메시지 큐에 메시지가 없으면 실패로 -1을 반환한다. 이 값을 설정하지 않으면 메시지가 메시지 큐에 도착할 때까지 기다린다.
MSG_NOERROR	메시지 크기가 msgsz 보다 클 때 초과하는 부분을 자른다. 이 값을 설정하지 않으면 실패로 -1을 반환한다.

- 다음은 msgrcv 의 사용 예로 msqid 메시지 큐로부터 메시지를 읽어와서 msg_data 에 저장한다.

```
struct msgbuf {
    long mtype;
    char mtext[10];
} msg_data;
msgrcv(msqid, &msg_data, 10, 0, 0);
```

Message Queue

- msgctl( ) 함수를 이용하면 메시지 큐의 정보를 얻거나, 변경하거나, 제거할 수 있다.

msgctl 함수
기능
　　메시지 큐를 제어한다.
기본형
　　int msgctl(int msqid, int cmd, struct msqid_ds *buf);
　　msqid: 메시지 큐 식별자
　　cmd: 제어 종류
　　buf: 메시지 큐 정보에 대한 포인터
반환값
　　성공: 0
　　실패: -1
헤더파일
　　<sys/types.h>
　　<sys/ipc.h>
　　<sys/msg.h>

Message Queue

- 어떤 일을 할지를 정하는 cmd 인수에 들어갈 수 있는 값과 의미는 다음과 같다.

cmd	의미
IPC_STAT	메시지 큐의 정보를 얻어 buf에 저장한다. msgctl(msqid, IPC_STAT, &buf);
IPC_SET	메시지 큐의 정보를 buf에 저장한 값으로 변경한다. 단, msg_perm과 msg_qbytes 멤버에 대한 정보만을 변경할 수 있다. msgctl(msqid, IPC_SET, &buf);
IPC_RMID	메시지 큐를 제거한다. 이 때, 세 번째 인수인 buf는 0으로 지정한다. msgctl(msqid, IPC_RMID, 0);