11장 파이프를 이용한 통신

1. 서론

- **동일 시스템에 있는 프로세스 사이의 통신 방법**
  - 프로세스는 파이프라는 **특수 파일**을 매개로 통신을 한다.
  - 이 특수 파일(파이프)에 대해 한 프로세스는 쓰기만 하고, 다른 프로세스는 읽기만 한다 (단방향 통신)
- **파이프 종류**
  - 임시 파이프
  - FIFO (named pipe)

1. 서론

- **임시 파이프**

```
$ ls | wc -w
    20
$ ls -l | wc -l
    20
```

  - 명령어 라인을 실행할 때 임시 파이프가 만들어져 프로세스 사이의 통신이 이루어진다.
  - 임시로 만들어진 것으로 명령어 라인의 실행이 끝나고 나면 파이프가 사라지므로 이름을 가지지 않는다.
  - ls 프로세스는 파이프에 대해 쓰기 작업만 수행하고, wc 프로세스는 읽기 작업만 수행한다.

1. 서론

- **FIFO (named pipe)**
  - mkfifo 명령을 사용하여 특수 파일인 named pipe를 생성한다.
  - named pipe는 ls 명령으로 확인할 수 있다.

```
$ mkfifo fifo
$ ls -l fifo
prw-r--r--    1 usp     student        0 Nov 19 03:22 fifo
```

```
$ tty
/dev/pts/8
$ cat > fifo
hello
hi
nice to meet you
```

```
$ tty
/dev/pts/9
$ cat < fifo
hello
hi
nice to meet you
```

1. 서론

- **FIFO(named pipe)로 채팅하기**
  - 두 개의 named pipe를 생성하여 하나는 A → B 프로세스로 다른 하나는 B → A 프로세스로 데이터를 전송한다.
  - 한 프로세스는 파이프에 쓰기만 수행하고 다른 프로세스는 파이프에서 읽기만 수행하도록 한다.

```
$ ls -l
prw-r--r--   1 usp     student       0 Nov 19 03:47 fifo1
prw-r--r--   1 usp     student       0 Nov 19 03:47 fifo2
```

```
$ cat < fifo2 &
$ cat > fifo1
hello
hi~
nice to meet you~
me too~
```

→
←
→
←

```
$ cat < fifo1 &
$ cat > fifo2
hello
hi~
nice to meet you~
me too~
```

2. pipe

- **프로세스 간 통신을 위한 임시 파이프를 생성한다.**

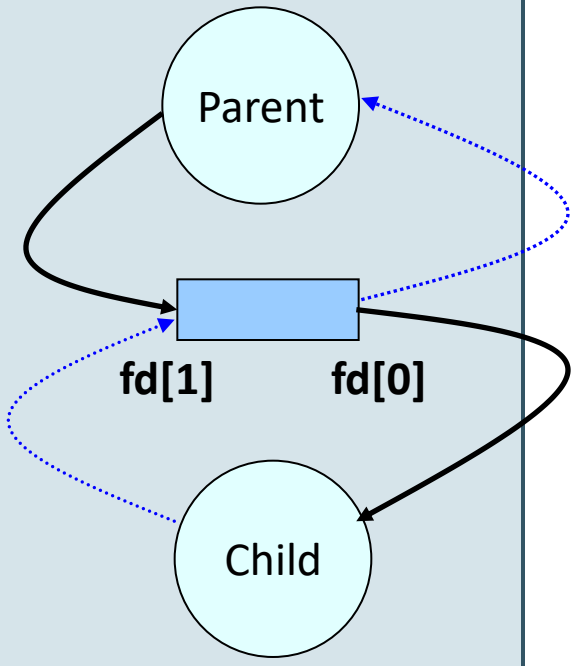
#include <unistd.h> int pipe(int *fd*[2]);	
fd	파일 기술자로 fd[0]은 읽기용이고 fd[1]은 쓰기용이다.
반환값	호출에 성공하면 0을 반환하고, 실패하면 -1을 반환한다.

- **pipe를 가리키는 한 쌍의 파일 기술자를 생성**
  - fd[0]은 읽기용
  - fd[1]은 쓰기용
  - 파일을 open하여 얻은 파일 기술자와 동일한 방법으로 사용한다.

2. pipe

```
#define SIZE    512
int main()
{
    char msg[SIZE];
    int fd[2];
    pid_t pid;

    pipe(fd);

    if ((pid = fork()) > 0) {
        close(fd[0]); strcpy(msg, "apple is red.\n");
        write(fd[1], msg, SIZE);
    }
    else {
        close(fd[1]);
        read(fd[0], msg, SIZE); printf("[child] %s\n", msg);
    }
}
```

```
$ ./ex11-04
[child] apple is red.
```



ex 11-04.c

2. pipe

- **파이프의 최대 크기는 정해져 있다.**
  - 파이프가 가득 차면 더 이상 파이프에 쓸 수 없기 때문에 write()는 블록된다.
  - 파이프가 비어 있으면 read()는 블록된다.

- **파이프의 읽는 쪽 프로세스가 종료되었는데**
  - 쓰는 쪽 프로세스가 write()를 호출하면 SIGPIPE 시그널을 받는다.

- **파이프의 쓰는 쪽 프로세스가 종료되면**
  - 읽는 쪽 프로세스는 파이프에 남아 있는 데이터를 읽을 수 있다.
  - 파이프가 비어 있으면 read()는 0을 반환한다.

3. 파이프를 이용한 통신 방법

- **고정된 크기의 메시지를 전달하는 방법**
  - 송신 측과 수신 측은 상호 약속된 크기의 메시지를 주고 받는다.
  - 실제 전송할 메시지가 약속된 메시지보다 작더라도 약속된 길이의 메시지를 전송해야 하므로 비효율적이다.
- **가변 크기의 메지지를 전달하는 방법**
  - 전달할 메시지만 파이프에 쓰기 때문에 크기가 제한된 파이프를 효율적으로 사용할 수 있다.
  - 메시지의 크기가 가변적이므로 송신 측이 보낸 메시지의 길이를 수신 측에 전달하지 않으면 정상적인 통신을 하기가 곤란하다.

3. 파이프를 이용한 통신 방법

고정된 크기의 메시지를 전달하는 방법

```
#define SIZE    512
int main()
{
    char msg[SIZE];
    int fd[2];
    pid_t pid;
    pipe(fd);
    if ((pid = fork()) > 0) {
        close(fd[0]); strcpy(msg, "apple is red.\n");
        write(fd[1], msg, SIZE);
    }
    else {
        close(fd[1]);
        read(fd[0], msg, SIZE); printf("[child] %s\n", msg);
    }
}
```

apple is red.\n\0?xr
p02r8qpqrst?!12bn
xp.2>34a?,3apq<e*
3)rpu-1@@g$%hjy
&1(pq4yy{+)*a.q.}1

```
$ ./a.out
[child] apple is red.
```

3. 파이프를 이용한 통신 방법

```
int main()
{
    char *msg1 = "apple is red";
    char *msg2 = "banana is yellow";
    int fd[2], len;
    pid_t pid;

    pipe(fd);

    if ((pid = fork()) > 0) {
        close(fd[0]);
        len = strlen(msg1) + 1;
        write(fd[1], &len, sizeof(int));
        write(fd[1], msg1, len);
        len = strlen(msg2) + 1;
        write(fd[1], &len, sizeof(int));
        write(fd[1], msg2, len);
    }
    else {
        ....
    }
}
```

가변 크기의 메지지를 전달하는 방법

12

apple is red

17

banana is yellow

```
char buffer[SIZE];
int nread, len;

close(fd[1]);

read(fd[0], &len, sizeof(int));
nread = read(fd[0], buffer, len);
printf("%d, %s\n", nread, buffer);

read(fd[0], &len, sizeof(int));
nread = read(fd[0], buffer, len);
printf("%d, %s\n", nread, buffer);
```

```
$ ./a.out
13, apple is red
17, banana is yellow
```

3. 파이프를 이용한 통신 방법

읽기 프로세스가 종료된 경우

```
#define SIZE    512

int main()
{
    char msg[SIZE];
    int fd[2];
    pid_t pid;

    signal(SIGPIPE, f);
    pipe(fd);

    if ((pid = fork()) > 0) {
        close(fd[0]); strcpy(msg, "apple is red.\n");
        write(fd[1], msg, SIZE); fprintf(stderr, "1st write\n"); sleep(1);
        write(fd[1], msg, SIZE); fprintf(stderr, "2nd write\n"); sleep(1);
        while (1);
    }
    else {
        close(fd[1]);
        read(fd[0], msg, SIZE); printf("[child] %s\n", msg);
    }
}
```

```
void f(int signo)
{
   fprintf(stderr, "SIGPIPE is received\n"); exit(-1);
}
```

```
$ ./a.out
1st write
[child] apple is red.
SIGPIPE is received
```

pipe1.c

3. 파이프를 이용한 통신 방법

```
#define SIZE    512
int main()
{
    char msg[SIZE];
    int fd[2];
    pid_t pid;

    signal(SIGPIPE, f);
    pipe(fd);

    if ((pid = fork()) > 0) {
        close(fd[0]); strcpy(msg, "apple is red.\n");
        write(fd[1], msg, SIZE); fprintf(stderr, "1st write\n");
    }
    else {
        close(fd[1]);
        while (1) {
            if (read(fd[0], msg, SIZE) == 0) {
                fprintf(stderr, "pipe is broken!!\n"); exit(-1);
            }
            printf("[read] %s\n", msg); sleep(1);
        }
    }
}
```

쓰기 프로세스가 종료된 경우

```
$ ./a.out
1st write
[read] apple is red.
pipe is broken!!
```

pipe2.c

3. 파이프를 이용한 통신 방법

쓰기 프로세스가 종료되지 않은 경우

```
#define SIZE    512
int main()
{
    char msg[SIZE];
    int fd[2];
    pid_t pid;

    signal(SIGPIPE, f);
    pipe(fd);

    if ((pid = fork()) > 0) {
        close(fd[0]); strcpy(msg, "apple is red.\n");
        write(fd[1], msg, SIZE); fprintf(stderr, "1st write\n"); while (1);
    }
    else {
        close(fd[1]);
        while (1) {
            if (read(fd[0], msg, SIZE) == 0) {
                fprintf(stderr, "pipe is broken!!\n"); exit(-1);
            }
            printf("[read] %s\n", msg); sleep(1);
        }
    }
}
```

```
$ ./a.out
1st write
[read] apple is red.
```

pipe3.c

3. 파이프를 이용한 통신 방법

- 파이프는 한쪽 방향으로만 통신이 가능
  - 파이프를 만들면 Read/Write 용 파일 기술자가 있으니 양방향 통신이 가능할 것 같은데…

```
int main()
{
      char msg[SIZE];
      int fd[2];
      pid_t pid;

      pipe(fd);

      if ((pid = fork()) > 0) {
            strcpy(msg, "apple is red.\n"); write(fd[1], msg, SIZE);
            read(fd[0], msg, SIZE); printf("msg from child: %s\n", msg);
      }
      else {
            read(fd[0], msg, SIZE); printf("msg from parent: %s\n", msg);
            strcpy(msg, "apple is yellow.\n"); write(fd[1], msg, SIZE);
      }
}
```

pipe4.c

```
msg from child: apple is red.
```

4. select

- **하나의 프로세스가 여러 프로세스와 메시지를 받는 경우**
  - **한 개의 파이프를 공유하지 않고 각 프로세스마다 별도의 파이프를 두면 편리하다.**
  - **하지만 여러 프로세스가 각 파이프에 메시지를 write하는 순서는 random하다.**
    - 따라서 메시지를 읽는 쪽에서는 파이프를 정해진 순서대로 읽는 것이 아니라 메시지가 도착한 순서대로 읽어야 한다.
  - **도착한 순서대로 메시지를 읽기 위해...**
    - select() 함수를 사용

4. select

- **select() 사용**
  - **파일 기술자 집합을 만든 뒤에 상태 변화가 있는지 감시하다가 상태 변화가 발생하면 해당 파일 기술자에 대한 처리(I/O)를 한다.**

    (예) ▶ 2개의 읽기용 파이프의 파일 기술자를 감시

    ▶ 감시 중 상태 변화가 발생한 파이프가 있으면 (즉, 메시지가 도착한 파이프가 있으면)

    ▶ 이 파이프에서 메시지를 읽음

4. select

- **지정한 파일 기술자 집합에서 상태 변화가 발생한 파일 기술자가 있는지 감시한다.**

int select(int *n*, fd_set **readfds*, fd_set **writefds*, fd_set **exceptfds*, struct timeval **timeout*);	
n	FD_SETSIZE 또는 readfds, wirtefds, exceptfds에 포함된 파일 기술자 중 가장 큰 값에 1을 더한 값
readfds	읽기용 파일 기술자 집합 – 읽을 내용이 생겼는지 감시
writefds	쓰기용 파일 기술자 집합 – 쓰기 가능한 상태가 되었는지 감시
exceptfds	파일 기술자 집합 – 예외 상황이 발생하였는지 감시
timeout	감시할 시간을 지정
반환값	readfds, writefds, exceptfds에서 변화가 발생한 파일 기술자 개수를 반환 감시할 시간이 경과한 경우에는 0을 반환 시그널을 catch한 경우에는 -1을 반환

4. select

- **readfds, writefds, exceptfds는 select()로부터 return할 때 상태 변화가 발생한 파일 기술자가 무엇인지를 나타내는 정보를 담아서 넘겨준다.**

- **인자 timeout이 NULL인 경우**
  - 주어진 파일 기술자에 상태 변화가 발생하거나 시그널을 catch할 때까지 무한정 대기한다.
  - 시그널을 catch한 경우에는 -1을 반환한다.

- **인자 timeout이 NULL이 아닌 경우**
  - 다음 페이지를 보자.

4. select

- **struct timeval**

```
struct timeval {
   long    tv_sec;        /* seconds */
   long    tv_usec;       /* microseconds */
};
```

- **timeout->tv_sec == 0 && timeout->tv_usec == 0인 경우**
  - select() 호출 시 상태 변화 여부와 상관없이 즉시 return한다. (즉 polling을 한다)
- **timeout->tv_sec ! = 0 || timeout->tv_usec != 0인 경우**
  - 지정된 시간 범위 내에서 상태 변화가 생길 때까지 기다린다.
  - 지정된 시간이 경과될 때까지 상태 변화가 생기지 않으면 0을 반환
  - 지정된 시간이 경과하지 않아도 시그널을 catch하면 -1을 반환

4. select

- **readfds, writefds, exceptfds를 NULL로 줄 수 있다.**
  - NULL로 주어진 경우 해당 파일 기술자에 대한 감시를 하지 않겠다는 것을 의미
- **readfds, writefds, exceptfds를 모두 다 NULL로 준 경우**
  - sleep()과 같은 용도로 사용될 수 있으며 sleep()보다 정밀한 시간 설정을 할 수 있음

```
struct timeval timeout;
timeout.tv_sec = 3;
timeout.tv_usec = 200000;
select(0, NULL, NULL, NULL, &timeout);
```

4. select

- **유용한 매크로**

매크로	기 능
FD_ZERO	파일 기술자 집합을 초기화 (공집합)
FD_SET	파일 기술자 집합에 지정한 파일 기술자를 추가
FD_CLR	파일 기술자 집합에서 지정한 파일 기술자를 제거
FD_ISSET	select()로부터 return한 후 사용하며, 파일 기술자 집합에서 특정 파일 기술자가 포함되어 있는지 확인

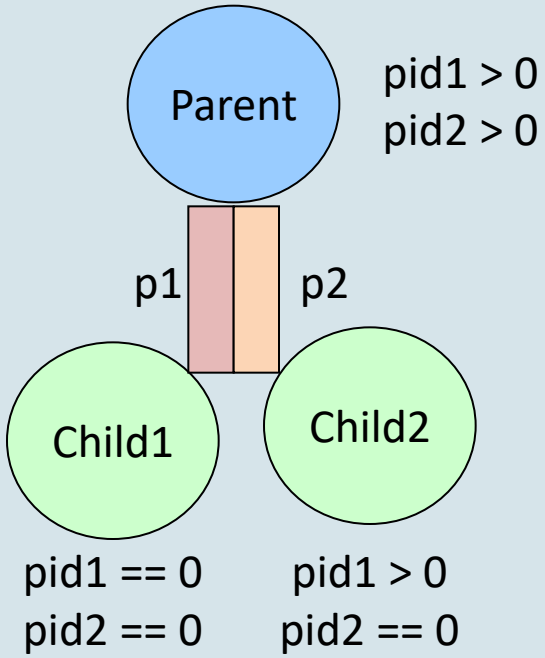
```
fd_set initset;
...
FD_ZERO(&initset);

FD_SET(pipe1[0], &initset);
FD_SET(pipe2[0], &initset);
FD_SET(pipe3[0], &initset);
...
```

```
if (select(..., &initset, ...) > 0) {
    if (FD_ISSET(pipe1[0], &initset)) ...;
    if (FD_ISSET(pipe2[0], &initset)) ...;
    if (FD_ISSET(pipe3[0], &initset)) ...;
}
```

4. select

ex 11-10.c

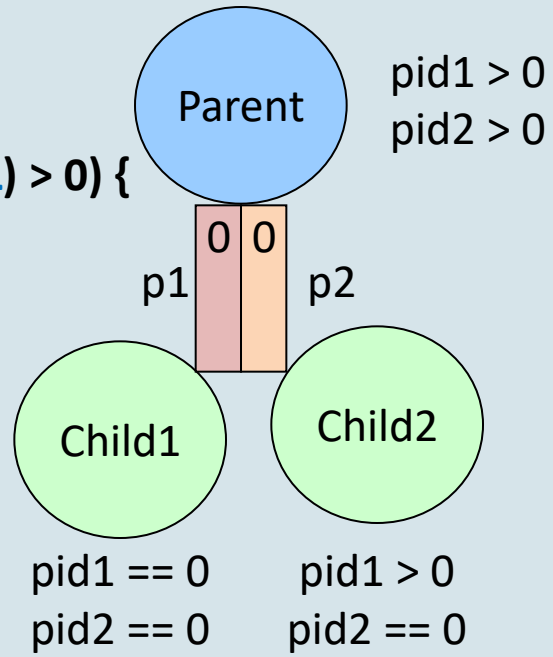
```
#define MSGSIZE 16

void onError(char *msg)
{
    fprintf(stderr, "%s", msg); exit(1);
}

int main()
{
    pid_t pid1 = 0, pid2 = 0;
    char msg[MSGSIZE];
    int p1[2], p2[2], i;
    fd_set initset, newset;

    pipe(p1); pipe(p2);

    if ((pid1 = fork()) == -1) onError("fail to call fork()\n");

    if (pid1 > 0) {
        if ((pid2 = fork()) == -1) onError("fail to call fork()\n");
    }
```



4. select

ex 11-10.c

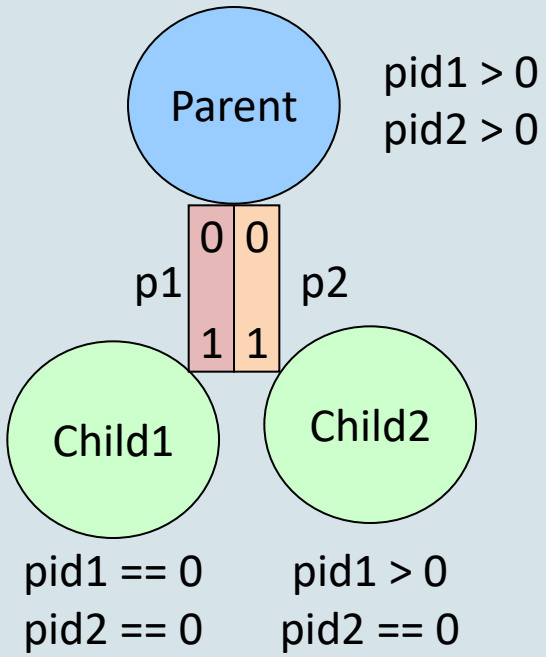
```
if (pid1 > 0 && pid2 > 0) {  /* parent process */
    close(p1[1]); close(p2[1]);

    FD_ZERO(&initset);
    FD_SET(p1[0], &initset);
    FD_SET(p2[0], &initset);

    newset = initset;

    while (select(p2[0] + 1, &newset, NULL, NULL, NULL) > 0) {
        if (FD_ISSET(p1[0], &newset)) {
            if (read(p1[0], msg, MSGSIZE) > 0)
                printf("[parent] %s\n", msg);
        }
        if (FD_ISSET(p2[0], &newset)) {
            if (read(p2[0], msg, MSGSIZE) > 0)
                printf("[parent] %s\n", msg);
        }
        newset = initset;
    }
}
```



4. select

ex 11-10.c

```
    else if (pid1 == 0 && pid2 == 0) {    /* child1 */
        close(p1[0]); close(p2[0]); close(p2[1]);

        for (i = 0; i < 3; i++) {
            sleep((i + 3) % 4);
            printf("child1: send message %d\n", i);
            write(p1[1], "i'm child1", MSGSIZE);
        }
        printf("child1: bye!\n"); exit(0);
    }
    else if (pid1 > 0 && pid2 == 0) {    /* child2 */
        close(p1[0]); close(p2[0]); close(p1[1]);
        for (i = 0; i < 3; i++) {
            sleep((i + 1) % 4);
            printf("child2: send message %d\n", i);
            write(p2[1], "I'm child2", MSGSIZE);
        }
        printf("child2: bye!\n"); exit(0);
    }
}
```



4. select

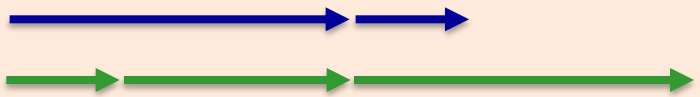
```
$ ./ex11-10
child2: send message 0
[parent] i'm child2
child1: send message 0
child1: send message 1
[parent] i'm child1
[parent] i'm child1
child2: send message 1
[parent] i'm child2
child1: send message 2
child1: bye!
[parent] i'm child1
child2: send message 2
child2: bye!
[parent] i'm child2
^C
```

실행 결과

두 개의 자식 프로세스가 무작위 순서로 보내는 메시지를 부모 프로세스가 select를 사용하여 도착하자마자 처리하고 있다.

child1
3초, 0초, 1초 간격

child2
1초, 2초, 3초 간격



5. 파이프와 exec 호출

- fork를 하기 전에 open()하여 얻은 파일 기술자는 fork() 후 부모 프로세스와 자식 프로세스가 공유한다.

```
int fd = open("a", O_RDONLY);
char buf[SIZE];

if ((pid = fork()) > 0) {
    read(fd, buf, SIZE);
}
else {
    read(fd, buf, SIZE);
}
```

```
int fd;
char buf[SIZE];

if ((pid = fork()) > 0) {
    fd = open("a", O_RDONLY);
    read(fd, buf, SIZE);
}
else {
    read(fd, buf, SIZE);
}
```

- pipe()로 생성한 파이프는 부모 프로세스 및 자식 프로세스와 같이 파일 기술자를 공유할 수 있는 프로세스 사이에서 통신을 하는 데 사용된다.

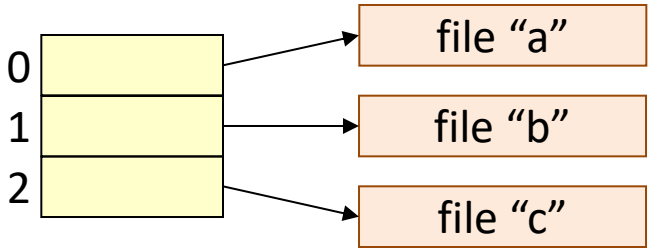
5. 파이프와 exec 호출

```
#define SIZE    512
int main()
{
    char msg[SIZE];
    int fd[2];
    pid_t pid;
    pipe(fd);
    if ((pid = fork()) > 0) {
        close(fd[0]); strcpy(msg, "apple is red.\n");
        write(fd[1], msg, SIZE); printf("[parent] %s\n", msg);
    }
    else {
        close(fd[1]);
        read(fd[0], msg, SIZE); printf("[child] %s\n", msg);
    }
}
```

fork를 하기 전에 open()하여 얻은 파일 기술자는 fork() 후 부모 프로세스와 자식 프로세스가 공유한다.

5. 파이프와 exec 호출

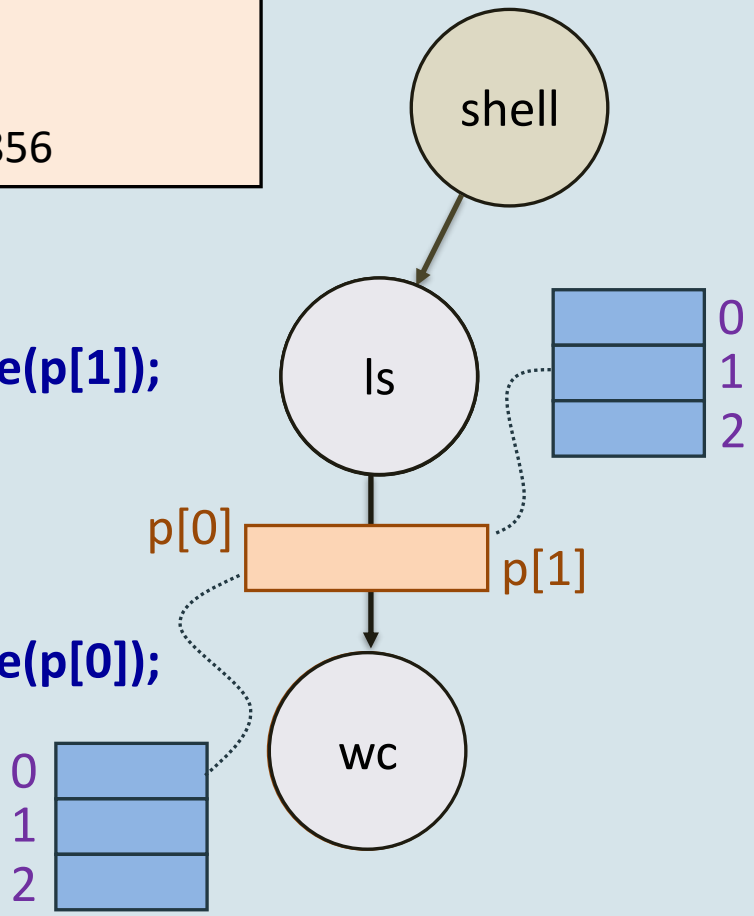
- **open된 파일은 exec()을 호출한 후에도 open되어 있다.**
  - exec()의 경우에는 변수를 계승할 수 없으므로 open된 파일에 대한 파일 기술자를 exec() 후에 실행되는 프로세스에 전달할 수 없다.
  - 하지만 이미 open된 파일은 exec() 후에도 open되어 있으므로 exec() 전 상황이 아래와 같다면 exec() 후에도 표준 입출력이 아닌 해당 파일에 대한 I/O를 수행한다.



5. 파이프와 exec 호출

```
int main()
{
        int p[2];
        pid_t pid;

        pipe(p);

        if ((pid = fork()) > 0) {
            close(p[0]); close(1); dup(p[1]); close(p[1]);
            execlp("ls", "ls", "-al", NULL);
            printf("fail to call exec\n");
        }
        else {
            close(p[1]); close(0); dup(p[0]); close(p[0]);
            execlp("wc", "wc", NULL);
            printf("fail to call exec\n");
        }
}
```

```
$ ./a.out
    46    407    2856

$ ls -al | wc
    46    407    2856
```



6. mkfifo

- **named 파이프를 생성한다.**

#include <sys/types.h> #include <sys/stat.h> int mkfifo(const char **pathname*, mode_t *mode*);	
pathname	특수 파일인 FIFO의 경로 이름 (일반 파일에 대한 경로를 지정하는 것과 같은 방법으로 지정한다.)
mode	pathname으로 지정한 특수 파일의 초기 접근 권한
반환값	호출에 성공하면 0을 반환하고, 실패하면 -1을 반환한다.

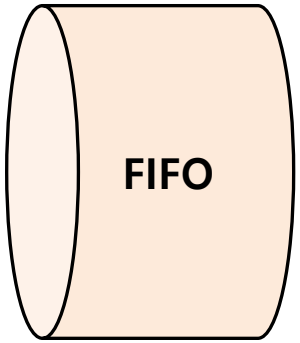
- **pipe()와 다른 점**
  - pipe()를 사용하여 만든 파이프는 프로세스가 종료하면 사라지는 임시 파이프이다.
  - mkfifo()로 만든 파이프는 프로세스 종료와 상관없이 항상 존재한다.

6. mkfifo

- **named 파이프(FIFO)를 사용한 통신**
  - 부모 자식 관계에 있지 않은 프로세스도 named 파이프를 통해 메시지를 주고 받을 수 있다.
- **named 파이프로 통신을 수행하려는 프로세스는**
  - named 파이프의 경로를 알고 있어야 한다.
  - named 파이프에 대한 적절한 권한이 있어야 한다.
  - 일반 파일을 개방하여 읽고 쓰듯이 named 파이프를 사용하면 된다.

1. 쓰기용으로 open
(FIFO가 읽기용으로 open되지 않았으면 open()은 블록됨)

3. FIFO에 쓰기
(FIFO가 가득찬 경우 쓰기 작업은 블록됨)



2. 읽기용으로 open
(FIFO가 쓰기용으로 open되지 않았으면 open()은 블록됨)

4. FIFO에서 읽기
(FIFO가 비어 있는 경우 읽기 작업은 블록됨)

6. mkfifo

메시지를 수신하는 측

```
#define MSGSIZE 64

int main()
{
      char msg[MSGSIZE];
      int fd, n, j;

      mkfifo("./fifo", 0600);
      if ((fd = open("./fifo", O_RDONLY)) < 0) {
          perror("fifo"); exit(-1);
      }

      fprintf(stderr, "fifo opened successfully…\n");

      for (j = 0; j < 3; j++) {
          if ((n = read(fd, msg, MSGSIZE)) < 0) {
              perror("read"); exit(1);
          }
          else if (n == 0) { fprintf(stderr, "broken pipe\n"); break; }

          printf("recv: %s\n", msg);
      }
      // unlink("./fifo");
}
```

fifo_rec.c

6. mkfifo

메시지를 송신하는 측

```
#define MSGSIZE 64

void f(int signo)
{
        fprintf(stderr, "SIGPIPE is received\n"); exit(-1);
}

int main()
{
        char msg[MSGSIZE];
        int fd;
        signal(SIGPIPE, f);
        if ((fd = open("./fifo", O_WRONLY)) < 0) {
            perror("fifo"); exit(-1);
        }
        fprintf(stderr, "fifo opened successfully...\n");
        while (1) {
              printf("input a message: "); scanf("%s", msg);

              if (write(fd, msg, MSGSIZE) == -1) {
                  perror("write"); exit(1);
              }
        }
}
```

fifo_snd.c

6. mkfifo

실행 결과

```
$ gcc –o fifo_rec fifo_rec.c
$ ./fifo_rec
fifo opened successfully…
recv: apple_is_red
recv: banana_is_yellow
recv: cherry_is_red
$ ls -al fifo
prw-------    1 usp      student        0 Jun 2 17:20 fifo
```

```
$ gcc –o fifo_snd fifo_snd.c
$ ./fifo_snd
fifo opened successfully…
input a message: apple_is_red
input a message: banana_is_yellow
input a message: cherry_is_red
input a message: berry_is_purple
SIGPIPE is received
```