네트워크 프로그래밍

16. IPC – PIPE & FIFO

= = named pipe

프로세스간 통신의 기본 개념

프로세스간 통신의 기본이해

프로세스끼리 직접 데이터교환은 불가능하다
so, 규약과 비슷하게 os에서 제공해 주는기법 = =ipc

프로세스간 통신

- 두 프로세스 사이에서의 데이터 전달
- 두 프로세스 사이에서의 데이터 전달이 가능 하려면, 두 프로세스가 함께 공유하는 메모리가 존재해야 한다.

프로세스간 통신의 어려움

- 모든 프로세스는 자신만의 메모리 공간을 독립적으로 구성한다.
- 즉, A 프로세스는 B 프로세스의 메모리 공간에 접근이 불가능하고, B 프로세스는 A 프로세스의 메모리 공간 접근이 불가능하다.
- 따라서 운영체제가 별도의 메모리 공간을 마련해 줘야 프로세스간 통신이 가능하다.

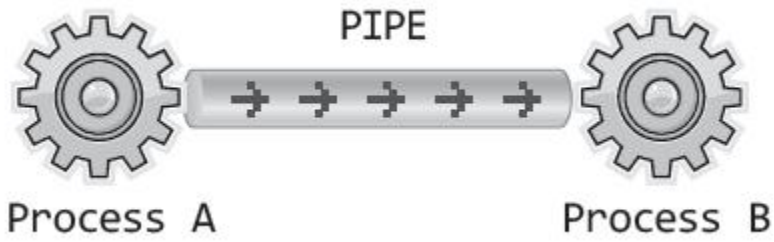
파이프 기반의 프로세스간 통신

```
#include <unistd.h>

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

➜ 성공 시 0, 실패 시 -1 반환

- filedes[0] 파이프로부터 데이터를 수신하는데 사용되는 파일 디스크립터가 저장된다. 즉, filedes[0]는 파이프의 출구가 된다.
- filedes[1] 파이프로 데이터를 전송하는데 사용되는 파일 디스크립터가 저장된다. 즉, filedes[1]은 파이프의 입구가 된다.



위의 함수가 호출되면, 운영체제는 서로 다른 프로세스가 함께 접근할 수 있는 메모리 공간을 만들고, 이 공간의 접근에 사용되는 파일 디스크립터를 반환한다.

파이프 생성의 예

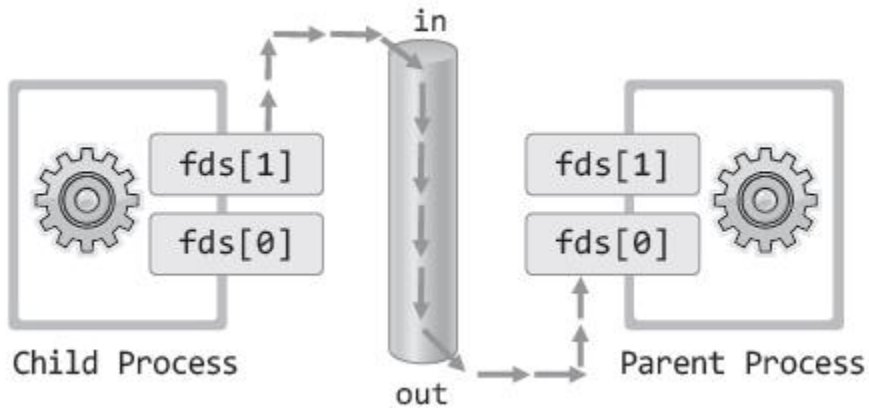
2가 1보다 먼저 호출될 경우
read 할 내용이 없기때문에
1.블락모드의 경우 블락
2.논블락 모드의 경우 논블락

파일의 10을 읽어야 할 경우
10을 읽기전까지 리턴X
but, 파이프의 경우
10을 읽어야 해도 먼저온 것이 5가 있다면
5를 바로 리턴

pipe1.c

```
int main(int argc, char *argv[])
{
    int fds[2];
    char str[]="Who are you?";
    char buf[BUF_SIZE];
    pid_t pid;

    pipe(fds);
    pid=fork();
    if(pid==0)
    {
        write(fds[1], str, sizeof(str));
    }
    else
    {
        read(fds[0], buf, BUF_SIZE);
        puts(buf);
    }
    return 0;
}
```



핵심은 파이프의 생성과 디스크립터의 복사에 있다!

실행결과

```
root@my_linux:/tcpip# gcc pipe1.c -o pipe1
root@my_linux:/tcpip# ./pipe1
Who are you?
```

프로세스간 양방향 통신: 잘못된 방식

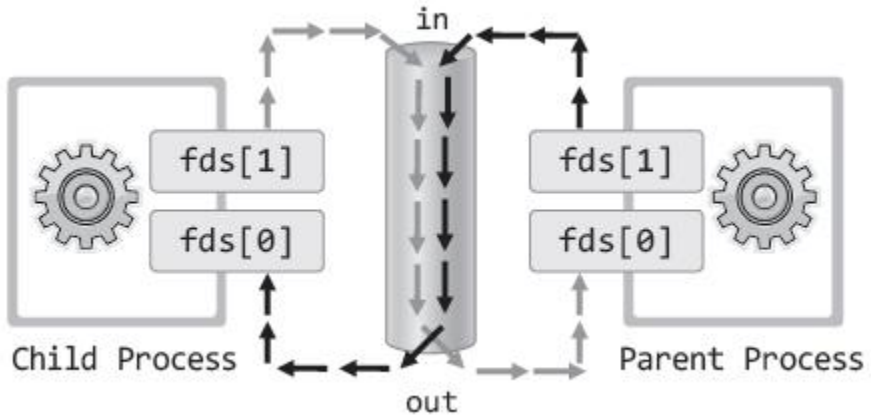
타이밍이 약간이라도 어긋나 경우
빨대의 음료수를 양쪽에서 먹듯이
제대로 실행되지 않는다.

만약 쌍방향 통신이 필요하다면
>>파이프를 2개 만들면됨
pipe3.c

pipe2.c

```
int main(int argc, char *argv[])
{
    int fds[2];
    char str1[]="Who are you?";
    char str2[]="Thank you for your message";
    char buf[BUF_SIZE];
    pid_t pid;

    pipe(fds);
    pid=fork();
    if(pid==0)
    {
        write(fds[1], str1, sizeof(str1));
        sleep(2);
        read(fds[0], buf, BUF_SIZE);
        printf("Child proc output: %s \n", buf);
    }
    else
    {
        read(fds[0], buf, BUF_SIZE);
        printf("Parent proc output: %s \n", buf);
        write(fds[1], str2, sizeof(str2));
        sleep(3);
    }
    return 0;
}
```



하나의 파이를 이용해서 양방향 통신을 하는 경우, 데이터를 쓰고 읽는 타이밍이 매우 중요해진다. 그런데 이를 컨트롤 하는 것은 사실상 불가능하기 때문에 이는 적절한 방법이 될 수 없다. 왼쪽의 예제에서 sleep 함수의 호출문을 주석처리 해 버리면 문제가 있음을 쉽게 확인할 수 있다.

프로세스간 양방향 통신: 적절한 방식

pipe3.c

```
int main(int argc, char *argv[])
{
    int fds1[2], fds2[2];
    char str1[]="Who are you?";
    char str2[]="Thank you for your message";
    char buf[BUF_SIZE];
    pid_t pid;

    pipe(fds1), pipe(fds2);
    pid=fork();
    if(pid==0)
    {
        write(fds1[1], str1, sizeof(str1));
        read(fds2[0], buf, BUF_SIZE);
        printf("Child proc output: %s \n", buf);
    }
    else
    {
         read(fds1[0], buf, BUF_SIZE);
         printf("Parent proc output: %s \n", buf);
         write(fds2[1], str2, sizeof(str2));
         sleep(3);
    }
    return 0;
}
```

양방향 통신을 위해서는 이렇듯 두 개의 파이프를 생성해야 한다. 그래야 입출력의 타이밍에 따라서 데이터의 흐름이 영향을 받지 않는다.

실행결과

```
root@my_linux:/tcpip# gcc pipe3.c -o pipe3
root@my_linux:/tcpip# ./pipe3
Parent proc output: Who are you?
Child proc output: Thank you for your message
```

프로세스간 통신의 적용

메시지를 저장하는 형태의 에코 서버

pipe선언 및 등록은
무조건 fork이전에 이루어져야한다
(그래야만 파이프로 프로세스간 통신가능)

echo_storeserv.c의 핵심코드

```
pipe(fds);
pid=fork();
if(pid==0)
{
    FILE * fp=fopen("echomsg.txt", "wt");
    char msgbuf[BUF_SIZE];
    int i, len;

    for(i=0; i<10; i++)
    {
        len=read(fds[0], msgbuf, BUF_SIZE);
        fwrite((void*)msgbuf, 1, len, fp);
    }
    fclose(fp);
    return 0;
}
```

서버에서 처음으로 생성하는 자식 프로세스

파이프를 생성하고 자식 프로세스를 생성해서,
자식 프로세스가 파이프로부터 데이터를 읽어서
저장하도록 구현되어 있다.

실제 파이프 사용시,
파이프를 여러개 쓰지 않으면
누가 쓰고 누가 읽고를 보장 할 수 없기때문에
unnamed pipe는 추천되는 방식이 아니다

```
clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_adr, &adr_sz);
. . . .
pid=fork();
if(pid==0)
{
    close(serv_sock);
    while((str_len=read(clnt_sock, buf, BUF_SIZE))!=0)
    {
        write(clnt_sock, buf, str_len);
        write(fds[1], buf, str_len);
    }

    close(clnt_sock);
    puts("client disconnected...");
    return 0;
}
else
    close(clnt_sock);
```

서버에서 연결 허용 시마다 생성하는 자식 프로세스

accept 함수 호출 후 fork 함수호출을 통해서 파이프의 디스크립터를 복사하고, 이를 이용해서 이전에 만들어진 자식 프로세스에게 데이터를 전송한다.

Named PIPE

Named Pipe

- (Unnamed) 파이프의 단점
  - 부모와 자식 프로세스 사이의 통신에만 사용 가능
  - 영구히 존재할 수 없음

- 명명된 파이프 (FIFO)
  - 임의의 두 프로세스 연결에 사용 가능
  - 파이프와 같이 프로세스간 통신 및 동기화 기능 지원
  - 파일 시스템에서 특수 파일 형태로 영구적으로 존재
  - 파일 이름을 부여하여 생성되며, 접근 허가 설정을 통해 모든 프로세스가 접근 가능

Named Pipe의 생성

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

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

- path
  - FIFO 경로명
- mode
  - FIFO 접근권한 모드
- 성공 시 0, 실패 시 -1 반환

```
if (mkfifo("/tmp/myfifo", S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)) == -1)
    perror("Failed to create myfifo");
```

Named Pipe의 제거

```
#include <unistd.h>

int unlink (const char *path);
```

- path
  - FIFO 경로명
- 성공 시 0, 실패 시 -1 반환

```
if (unlink("/tmp/myfifo") == -1)
    perror("Failed to remove myfifo");
```

open으로 FIFO 열기

- open(const char *path, O_RDONLY);
  - 어떤 프로세스가 쓰기 상태로 같은 FIFO를 열 때까지 반환하지 않음
- open(const char *path, O_RDONLY | O_NONBLOCK);
  - (성공에 의한) 즉시 반환 〉 즉, 상대가 열던 안열던 열게된다는 것
- open(const char *path, O_WRONLY);
  - 어떤 프로세스가 읽기 상태로 같은 FIFO를 열 때까지 반환하지 않음
- open(const char *path, O_WRONLY | O_NONBLOCK);
  - 즉시 반환
    - 어떤 프로세스가 읽기 상태로 열린 FIFO를 가지지 않으면 -1을 반환 (ENXIO)
    - 어떤 프로세스가 읽기 상태로 열린 FIFO를 가진다면 파일 기술자 반환

FIFO 읽기와 쓰기
(I/O on pipes and FIFOs has exactly the same semantics.)

- FIFO(& pipe) 특성
  - The communication channel provided by a pipe is a *byte stream*.
    - There is no concept of message boundaries.
  - A pipe has a limited capacity.
    - Since Linux 2.6.11, the pipe capacity is 65536 bytes.

지금부터의 특성은 ,pipe건 fifo건 똑같이 적용

오늘 5/30이후
다음시간 파일 읽기쓰기부터 해주세요 라고할것

FIFO 읽기와 쓰기
(I/O on pipes and FIFOs has exactly the same semantics.)

- 읽기
  - Blocking 모드
    - If a process attempts to read from an empty pipe, then read() will block until data is available.
  - Non-blocking 모드
    - read() will return -1 and set errno to [EAGAIN]
  - When the last writer for a FIFO closes the FIFO, an end of file is generated for the reader of the FIFO.
    - read() will return 0.

fifo에 대해서 마지막으로 닫는녀석이
eof를 날리게 됨
〉〉소켓과 특성이 같음

FIFO 읽기와 쓰기
(I/O on pipes and FIFOs has exactly the same semantics.)

- 쓰기

SIGPIPE발생시 기본동작이 종료니까
이를 핸들러에서 바꿔주는게 필요하다(앞 내용기억)

  - If we write to a pipe whose read end has been closed, the signal SIGPIPE is generated. If we either ignore the signal or catch it and return from the signal handler, write returns -1 with errno set to EPIPE.

상대가 없을때 쓸 경우, write는 -1을 반환하고, errno을 EPIPE로 만든다

  - PIPE_BUF specifies the maximum amount of data that can be written atomically to a FIFO.

원자적으로 써진다

    - On Linux, **PIPE_BUF** is 4096 bytes.

SIGPIPE

이벤트: 종료된 소켓에 쓰기를 시도할 때 발생

기본동작: 프로그램 종료

FIFO 읽기와 쓰기
(I/O on pipes and FIFOs has exactly the same semantics.)

- 쓰기

원자적으로 써지기 때문에 다른녀석들과 같이 쓴다고 해도, 내것이 여러개로 쪼개져서 들어갈 일은 없다

  - O_NONBLOCK disabled, n <= PIPE_BUF
    - All n bytes are written atomically; write() may block if there is not room for n bytes to be written immediately

만약 파이프에 n byte공간이 없다면 block을 걸어놓는다
〉〉만약 남는부분에 그냥 써버리면 원자적으로 수행한다는 조건을 위배 할 수 있기때문에!

  - O_NONBLOCK enabled, n <= PIPE_BUF
    - If there is room to write n bytes to the pipe, then write() succeeds immediately, writing all n bytes; otherwise write() fails, with errno set to EAGAIN.

어차피 buf보다 크니까 원자적인것을 보장 할 수 없으니,
자리있으면 쓰고 자리가 1도없으면 block

여러명이 쓴 내용이 섞여서 구별할 수 없는 상태

  - O_NONBLOCK disabled, n > PIPE_BUF
    - The write is nonatomic: the data given to write() may be interleaved with write()s by other process; the write() blocks until n bytes have been written.
  - O_NONBLOCK enabled, n > PIPE_BUF
    - If the pipe is full, then write() fails, with errno set to EAGAIN. Otherwise, from 1 to n bytes may be written (i.e., a "partial write" may occur; the caller should check the return value from write() to see how many bytes were actually written), and these bytes may be interleaved with writes by other processes.

위와 같지만, 자리가 1도 없으면 return한다

FIFO 읽기와 쓰기
(I/O on pipes and FIFOs has exactly the same semantics.)

- 쓰기
  - A write of PIPE_BUF bytes or less will not be interleaved with the writes from other processes to the same pipe (or FIFO).
  - But if multiple processes are writing to a pipe (or FIFO), and if we write more than PIPE_BUF bytes, the data might be interleaved with the data from the other writers.