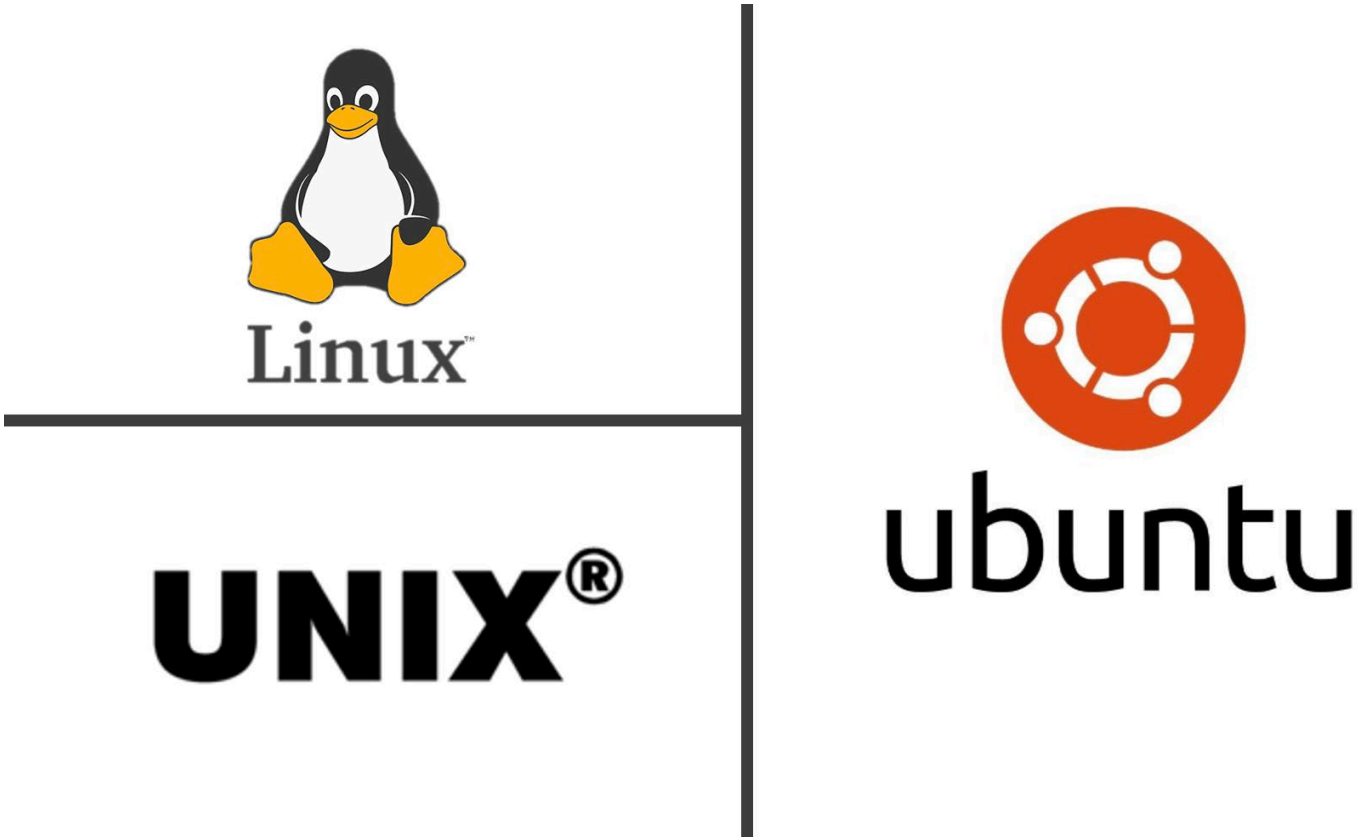


Chapter 09 메모리 매핑

목차

학습목표

- 통신 프로그램의 개념을 이해한다.
- 메모리 매핑을 이용한 IPC 기법을 이해한다.
- 메모리 매핑 함수를 사용해 프로그램을 작성할 수 있다.

01. 개요

■ 프로세스 간 통신

▪ 프로세스 간 통신(IPC)

- 동일한 시스템 안에서 수행 중인 프로세스끼리 데이터를 주고받는 것
- IPC에는 파이프 같은 특수 파일을 이용하거나, 메모리 매핑이나 공유 메모리 같은 메모리 영역을 이용하는 방법이 있음
- 또한 메시지 큐, 공유 메모리, 세마포어 등 유닉스 시스템 V에서 제공하는 IPC 방법이 있음
- 넓은 의미에서 프로세스 간 통신에는 종료 상태와 시그널 같은 정숫값을 주고받는 것도 포함
- 시그널도 프로세스 사이에서 데이터를 주고받으므로 프로세스 간 통신 방법 의 하나로 간주할 수 있음

■ 네트워크를 이용한 통신

▪ 네트워크를 이용한 통신

- 메일이나 파일, 클라이언트/서버 형태의 응용 프로그램 등 서로 다른 시스템 간의 네트워크를 이용한 통신이 증가
- 시스템에서 네트워크를 이용한 통신은 TCP/IP 프로토콜을 기본으로 하며, 소켓 라이브러리를 이용
- 이외에도 TLI가 있지만 소켓 라이브러리에 밀려 거의 사용하지 않음

01. 개요

■ 네트워크를 이용한 통신

▪ 메모리 매핑

표 9-1 메모리 매핑 함수

기능	함수
메모리 매핑	void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
메모리 매핑 해제	int munmap(void *addr, size_t length);
메모리 보호 모드 변경	int mprotect(void *addr, size_t len, int prot);
파일 크기 조정	int truncate(const char *path, off_t length); int ftruncate(int fd, off_t length);
메모리 매핑 변경	void *mremap(void *old_address, size_t old_size, size_t new_size, int flags);
매핑된 메모리 동기화	int msync(void *addr, size_t length, int flags);

02. 메모리 매핑과 해제

■ 메모리 매핑 : mmap(2)

```
#include <sys/mman.h>                                    [함수 원형]

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

- addr : 매핑할 메모리의 주소
- length : 메모리 공간의 크기
- prot : 보호 모드
- flags : 매핑된 데이터의 처리 방법 지정 상수
- fd : 파일 기술자
- offset : 파일 오프셋

- mmap() 함수의 특징
  - fd가 가리키는 파일에서 offset으로 지정한 오프셋부터 length 크기만큼 데이터를 읽어 addr이 가리키는 메모리 공간에 매핑
  - prot에는 읽기, 쓰기 등 보호 모드를 지정하고 flags에는 읽어온 데이터를 처리하기 위한 정보를 지정
  - 첫 번째 인자인 addr에는 매핑하려는 주소를 직접 지정하거나 그렇지 않을 경우 NULL을 사용
  - 두 번째 인자인 length에는 매핑할 메모리 공간의 크기를 지정

02. 메모리 매핑과 해제

■ 메모리 매핑 : mmap(2)

- mmap() 함수의 특징
  - 세 번째 인자인 prot에는 매핑한 데이터를 읽기만 할지, 쓰기나 실행도 허용할지 등 보호 모드를 지정
  - prot 인자에 사용할 수 있는 상수
    - **PROT_READ** : 매핑된 파일을 읽기만 함
    - **PROT_WRITE** : 매핑된 파일에 쓰기를 허용
    - **PROT_EXEC** : 매핑된 파일을 실행할 수 있음
    - **PROT_NONE** : 매핑된 파일에 접근할 수 없음
  - 네 번째 인자인 flags에는 매핑된 데이터를 처리하기 위한 다른 정보를 지정
  - flags에 지정할 수 있는 값
    - **MAP_SHARED**
      다른 프로세스와 데이터 변경 내용을 공유, 이 플래그가 설정되어 있으면 쓰기 동작은 매핑된 메모리의 내용을 변경
    - **MAP_SHARED_VALIDATE**
      MAP_SHARED와 같으나 전달받은 플래그를 커널이 모두 확인하고 모르는 플래그가 있을 경우 오류로 처리
    - **MAP_PRIVATE**
      데이터의 변경 내용을 공유하지 않음, 이 플래그가 설정되어 있으면 최초의 쓰기 동작에서 매핑된 메모리의 사본을 생성하고 매핑 주소는 사본을 가리킴
      MAP_SHARED와 MAP_PRIVATE 플래그 중 하나는 반드시 지정해야 하며, 두 플래그를 같이 지정 할 수는 없음
    - **MAP_ANONYMOUS**
    - fd를 무시하고 할당된 메모리 영역을 0으로 초기화, offset은 0이어야 하며 일부 시스템에서는 fd 대신 –1을 지정해야 할 수도 있음

02. 메모리 매핑과 해제

■ 메모리 매핑 : mmap(2)

- mmap() 함수의 특징
  - flags에 지정할 수 있는 값
    - **MAP_ANON**
      이 플래그는 MAP_ANONYMOUS와 동일한 플래그로 다른 시스템과의 호환성을 위해 제공
    - **MAP_FIXED**
      매핑할 주소를 정확히 지정
      MAP_FIXED 플래그를 지정하고 mmap() 함수가 성공하면 해당 메모리 영역의 내용은 매핑된 내용으로 변경
      해당 메모리 영역에 매핑된 파일이 있다면 이 파일의 매핑은 해제되고 새 파일이 매핑
      지정된 주소를 사용할 수 없으면 mmap() 함수는 실패
      이 플래그는 시스템의 종류나 커널 버전, C 라이브러리 버전에 따라 다를 수 있으므로 매우 신중하게 사용해야 함
    - **MAP_NORESERVE**
      MAP_PRIVATE를 지정하면 시스템은 매핑에 할당된 메모리 공간만큼 스왑 영역을 할당
      이 스왑 영역은 매핑된 데이터의 사본을 저장하는 데 사용
      MAP_ NORESERVE를 지정하면 스왑 영역을 할당하지 않음
      스왑 영역을 할당하지 않은 상태에서 매핑된 데이터를 수정할 경우 물리적 메모리에 빈 공간이 없으면 세그먼트 오류가 발생할 수 있음

  - 다섯 번째 인자인 fd는 매핑할 파일의 파일 기술자이고, 여섯 번째 인자인 offset은 매핑할 파일의 파일 오프셋
  - mmap() 함수는 페이지 단위로 메모리 매핑을 실행, 매핑된 영역의 마지막 페이지에 남는 부분은 0으로 채움
  - 매핑된 메모리 영역을 벗어난 공간에 접근하려고 하면 SIGBUS 혹은 SIGSEGV 시그널이 발생
  - mmap() 함수는 수행에 성공하면 매핑된 메모리의 시작 주소를 리턴
  - 매핑된 영역의 크기는 [시작 주소 + length]가 되며 실패하면 상수 MAP_FAILED를 리턴

02. 메모리 매핑과 해제

■ 메모리 매핑 : mmap(2)

- mmap() 함수와 기존 방식의 비교
  - 파일 입출력 함수

```
fd = open(...);
lseek(fd, offset, whence);
read(fd, buf, len);
```

  - mmap() 함수를 사용하여 파일의 오프셋을 이동시키고 read() 함수를 호출해 데이터를 buf로 읽음

```
fd = open(...);
address = mmap((caddr_t) 0, len, (PROT_READ | PROT_WRITE), MAP_PRIVATE, fd, offset);
```

  - 열린 파일의 내용을 mmap() 함수를 사용해 메모리에 매핑
  - 이후의 작업은 address가 가리키는 메모리 영역의 데이터를 대상으로 수행
  - 매번 read() 함수로 데이터를 읽어올 필요가 없음

02. 메모리 매핑과 해제

■ [예제 9-1] mmap() 함수로 메모리 매핑하기

```
01 #include <sys/mman.h>
02 #include <sys/stat.h>
03 #include <fcntl.h>
04 #include <unistd.h>
05 #include <stdlib.h>
06 #include <stdio.h>
07
08 int main(int argc, char *argv[]) {
09     int fd;
10     caddr_t addr;
11     struct stat statbuf;
12
13     if (argc != 2) {
14         fprintf(stderr, "Usage : %s filename₩n", argv[0]);
15         exit(1);
16     }
17
18     if (stat(argv[1], &statbuf) == -1) {
19         perror("stat");
20         exit(1);
21     }
22
23     if ((fd = open(argv[1], O_RDWR)) == -1) {
24         perror("open");
25         exit(1);
26     }
27
28     addr = mmap(NULL, statbuf.st_size, PROT_READ|PROT_WRITE,
29                 MAP_SHARED, fd, (off_t)0);
30     if (addr == MAP_FAILED) {
31         perror("mmap");
32         exit(1);
33     }
34     close(fd);
35
36     printf("%s", addr);
37 }
```

실행

```
$ cat mmap.dat
Hanbit
Academy
$ ch9_1.out
Usage : a.out filename
$ ch9_1.out mmap.dat
Hanbit
Academy
```

- **13행** 명령행 인자로 파일명을 제대로 지정했는지 확인
- **18행** 파일 크기를 알아내기 위해 stat() 함수로 파일 정보를 검색
- **23행** 명령행 인자로 지정한 파일을 실행
- **28행** 열린 파일의 내용을 mmap() 함수로 메모리 매핑
  매핑된 메모리 영역의 읽기와 쓰가 가능하도록 PROT_READ|PROT_WRITE로 지정하고 MAP_SHARED 플래그를 지정
- **34행** 파일 내용을 메모리에 매핑했으므로 파일을 닫음
  열린 파일을 닫아도 매핑된 메모리 주소인 addr을 이용해 데이터를 읽거나 쓸 수 있음
- **36행** addr이 가리키는 주소에 저장된 내용을 모두 출력
- **실행 결과** 먼저 명령행 인자로 지정한 파일인 mmap.dat의 내용을 살펴본 후 실행 결과를 보면 파일을 닫은 후에도 파일 내용이 모두 출력되었음을 확인

02. 메모리 매핑과 해제

■ 메모리 매핑 해제 : munmap(2)

```
#include <sys/mman.h>                                    [함수 원형]

int munmap(void *addr, size_t length);
```

- addr : 매핑된 메모리의 시작 주소
- length : 메모리 영역의 크기

- munmap() 함수의 특징
  - addr이 가리키는 영역에 length 크기만큼 할당해 매핑한 메모리를 해제
  - addr이 가리키는 메모리 주소가 mmap() 함수에서 지정한 위치가 아니면 어떻게 동작할지 정의되어 있지 않음
  - munmap() 함수로 매핑을 해제한 메모리 영역 에 접근하면 SIGSEGV 시그널이 발생
  - munmap() 함수는 수행에 성공하면 0을, 실패하면 –1을 리턴

02. 메모리 매핑과 해제

■ [예제 9-2] munmap() 함수로 메모리 매핑 해제하기

```
01 #include <sys/mman.h>
02 #include <sys/stat.h>
03 #include <fcntl.h>
04 #include <unistd.h>
05 #include <stdlib.h>
06 #include <stdio.h>
07
08 int main(int argc, char *argv[]) {
09     int fd;
10     caddr_t addr;
11     struct stat statbuf;
12
13     if (argc != 2) {
14         fprintf(stderr, "Usage : %s filename₩n", argv[0]);
15         exit(1);
16     }
17
18     if (stat(argv[1], &statbuf) == -1) {
19         perror("stat");
20         exit(1);
21     }
22
23     if ((fd = open(argv[1], O_RDWR)) == -1) {
24         perror("open");
25         exit(1);
26     }
27
28     addr = mmap(NULL, statbuf.st_size, PROT_READ|PROT_WRITE,
29                 MAP_SHARED, fd, (off_t)0);
30     if (addr == MAP_FAILED) {
31         perror("mmap");
32         exit(1);
33     }
34     close(fd);
35
36     printf("%s", addr);
37
38     if (munmap(addr, statbuf.st_size) == -1) {
39         perror("munmap");
40         exit(1);
41     }
42
43     printf("%s", addr);
44 }
```

- **8~41행** 매핑된 메모리 영역을 munmap() 함수로 해제
- **43행** 이미 해제된 메모리 영역에 접근을 시도
- **실행 결과** 36행에서 출력한 메모리 내용을 43행에서 다시 출력하려고 하면 SIGSEGV 시그널이 발생해 코어 덤프를 생성하고 프로세스가 종료됨을 알 수 있음

실행

```
$ ch9_2.out mmap.dat
Hanbit
Academy
세그멘테이션 오류 (코어 덤프됨)
```

02. 메모리 매핑과 해제

■ 보호 모드 변경 : mprotect(2)

```
#include <sys/mman.h>                                    [함수 원형]

int mprotect(void *addr, size_t len, int prot);
```

- addr : 매핑된 메모리의 시작 주소
- len : 메모리 영역의 크기
- prot : 보호 모드

- mprotect() 함수의 특징
  - 매핑된 메모리의 보호 모드는 mmap() 함수로 메모리 매핑을 수행할 때 초깃값을 설정
  - addr로 지정한 주소에 len 크기만큼 매핑된 메모리의 보호 모드를 prot에 지정한 값으로 변경
  - prot 인자에는 앞서 mmap() 함수에서 설명한 PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE을 사용
  - mprotect() 함수는 수행에 성공하면 0을, 실패하면 –1을 리턴

03. 파일 확장과 메모리 매핑

■ 경로명을 사용한 파일 크기 확장 : truncate(2)

```
#include <unistd.h>                                    [함수 원형]
#include <sys/types.h>

int truncate(const char *path, off_t length);
```

- path : 크기를 변경할 파일의 경로
- length : 변경하려는 크기

- truncate() 함수의 특징
  - truncate() 함수를 사용하면 path에 지정한 파일의 크기를 length로 지정한 크기로 변경 할 수 있음
  - 파일의 원래 크기가 length보다 크면 length 길이를 초과하는 부분은 버리고, length보다 작으면 파일의 크기를 증가시키고 해당 부분을 널 바이트('₩0')로 채움
  - truncate() 함수를 사용하려면 경로로 지정한 파일에 대한 쓰기 권한이 있어야 함
  - 이 함수는 파일의 오프셋을 변경하지는 않지만 파일의 상태 정보에서 st_ctime과 st_mtime을 수정
  - truncate() 함수는 수행에 성공하면 0을, 실패하면 -1을 리턴

03. 파일 확장과 메모리 매핑

■ 파일 기술자를 사용한 파일 크기 확장 : ftruncate(2)

```
#include <unistd.h>                                    [함수 원형]
#include <sys/types.h>

int ftruncate(int fd, off_t length);
```

- fd : 크기를 변경할 파일의 파일 기술자
- length : 변경하려는 크기

- ftruncate() 함수의 특징
  - ftruncate() 함수는 파일의 경로명 대신 파일 기술자를 받음, 즉, fd에 지정한 파일의 크기 를 length로 지정한 크기로 변경
  - 이때 파일 기술자는 크기를 변경할 파일을 열고 리턴 은 값이어야 함
  - 파일의 원래 크기가 length보다 크면 length 길이를 초과하는 부분은 버림
  - lengt보다 작으면 파일의 크기를 증가시키고 해당 부분을 널 바이트('₩0')로 채움
  - ftruncate() 함수는 일반 파일과 공유 메모리에만 사용할 수 있음
  - 이 함수는 파일의 오프셋을 변경하지 않지만 파일의 상태 정보에서 st_ctime과 st_mtime을 수정
  - ftruncate() 함수로 디렉터리에 접근하거나 쓰기 권한이 없는 파일에 접근하면 오류가 발생
  - ftruncate() 함수는 수행에 성공하면 0을, 실패하면 –1을 리턴

03. 파일 확장과 메모리 매핑

■ [예제 9-3] ftruncate() 함수로 파일 크기 확장하기

```
01 #include <sys/types.h>
02 #include <sys/mman.h>
03 #include <fcntl.h>
04 #include <stdio.h>
05 #include <unistd.h>
06
07 main() {
08     int fd, pagesize, length;
09     caddr_t addr;
10
11     pagesize = sysconf(_SC_PAGESIZE);
12     length = 1 * pagesize;
13
14     if((fd = open("m.dat", O_RDWR | O_CREAT | O_TRUNC, 0666)) == -1) {
15         perror("open");
16         exit(1);
17     }
18
19     if(ftruncate(fd, (off_t) length) == -1) {
20         perror("ftruncate");
21         exit(1);
22     }
23
24     addr = mmap(NULL, length, PROT_READ|PROT_WRITE, MAP_SHARED, fd, (off_t)0);
25     if(addr == MAP_FAILED) {
26         perror("mmap");
27         exit(1);
28     }
29
30     close(fd);
31
32     strcpy(addr, "ftruncate Test₩n");
33 }
```

실행

```
$ ls m.dat
ls: 'm.dat'에 접근할 수 없습니다: 그런 파일이나 디렉터리가 없습니다
$ ch9_3.out
$ cat m.dat
ftruncate Test
```

- **11행** sysconf() 함수를 사용해 시스템에 설정된 페이지 크기를 구함, 이는 mmap() 함수에서 페이지 단위로 메모리를 매핑하기 때문
- **14행** m.dat 파일을 읽기, 쓰기 가능 상태로 생성
- **19행** 새로 생성해 비어 있는 파일은 mmap() 함수로 매핑할 수 없으므로 ftruncate() 함수를 사용해 파일 크기를 페이지 크기로 증가시킴
- **24행** mmap() 함수로 m.dat 파일을 메모리에 매핑
- **32행** 매핑된 메모리에 문자열 "ftruncate Test"를 출력
- **실행 결과** m.dat 파일이 생성되고 문자열을 출력해 저장되었음을 확인할 수 있음

03. 파일 확장과 메모리 매핑

■ 메모리 주소를 다시 매핑하기 : mremap(2)

```
#define _GNU_SOURCE                                          [함수 원형]
#include <sys/types.h>

void *mremap(void *old_address, size_t old_size, size_t new_size, int flags);
```

- old_address : 크기를 변경할 메모리 주소
- old_size : 현재 메모리 크기
- new_size : 바꾸려는 메모리 크기
- flags : 0 또는 MREMAP_MAYMOVE

- mremap() 함수의 특징
  - mremap( ) 함수는 리눅스에서만 제공하는 것으로, 매핑된 메모리의 크기와 위치를 변경 할 수 있음
  - old_address에 현재 매핑된 메모리 주소를 지정하고 old_size에는 현재 메모리 크기를 지정
  - new_size에는 다시 매핑할 메모리 크기를 지정
  - 이 값은 old_size보다 크거나 작을 수 있음
  - flasg에는 0을 지정하거나 MREMAP_MAYMOVE를 지정
  - MREMAP_MAYMOVE를 지정하면 크기를 변경할 때 매핑의 위치를 이동해도 된다는 의미

03. 파일 확장과 메모리 매핑

■ 매핑된 메모리 동기화 : msync(2)

```
#include <sys/mman.h>                                    [함수 원형]

int msync(void *addr, size_t length, int flags);
```

- addr : 매핑된 메모리의 시작 주소
- length : 메모리 영역의 크기
- flags : 동기화 동작

- msync() 함수의 특징
  - msync() 함수는 인자로 지정한 addr로 시작하는 메모리 영역에서 [addr + length]만큼의 내용을 백업 저장 장치로 기록
  - 이 함수를 사용하지 않으면 munmap() 함수가 호출되기 전에 변경된 메모리 내용이 저장되는지 보장할 수 없음
  - flags는 msync() 함수의 동작을 지시하는 값으로, MS_ASYNC와 MS_SYNC 중 하나를 지정
    - **MS_ASYNC** : 비동기 쓰기 작업을 수행, msync() 함수는 즉시 리턴하고, 함수가 리턴한 후 적절한 시점에 쓰기 작업을 수행
    - **MS_SYNC** : 쓰기 작업을 완료할 때까지 msync() 함수가 리턴하지 않음, 메모리의 크기가 클 경우 비교적 시간이 걸릴 수 있음
    - **MS_INVALIDATE** : 메모리에 있는 기존 내용을 무효화할 것인지 확인
  - msync() 함수는 수행에 성공하면 0을, 실패하면 -1을 리턴

03. 파일 확장과 메모리 매핑

■ [예제 9-5] 데이터 교환하기

```
01 #include <sys/types.h>
02 #include <sys/mman.h>
03 #include <sys/stat.h>
04 #include <fcntl.h>
05 #include <unistd.h>
06 #include <stdlib.h>
07 #include <stdio.h>
08
09 int main(int argc, char *argv[]) {
10     int fd;
11     pid_t pid;
12     caddr_t addr;
13     struct stat statbuf;
14
15     if (argc != 2) {
16         fprintf(stderr, "Usage : %s filename₩n", argv[0]);
17         exit(1);
18     }
19
20     if (stat(argv[1], &statbuf) == -1) {
21         perror("stat");
22         exit(1);
23     }
24
25     if ((fd = open(argv[1], O_RDWR)) == -1) {
26         perror("open");
27         exit(1);
28     }
29
30     addr = mmap(NULL, statbuf.st_size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, (off_t)0);
31     if (addr == MAP_FAILED) {
32         perror("mmap");
33         exit(1);
34     }
35     close(fd);
36
37     switch (pid = fork()) {
38         case -1 : /* fork failed */
39             perror("fork");
40             exit(1);
41             break;
42         case 0 : /* child process */
43             printf("1. Child Process : addr=%s", addr);
44             sleep(1);
45             addr[0] = 'x';
46             printf("2. Child Process : addr=%s", addr);
47             sleep(2);
48             printf("3. Child Process : addr=%s", addr);
49             break;
50         default : /* parent process */
51             printf("1. Parent process : addr=%s", addr);
52             sleep(2);
53             printf("2. Parent process : addr=%s", addr);
54             addr[1] = 'y';
55             printf("3. Parent process : addr=%s", addr);
56             break;
57     }
58 }
```

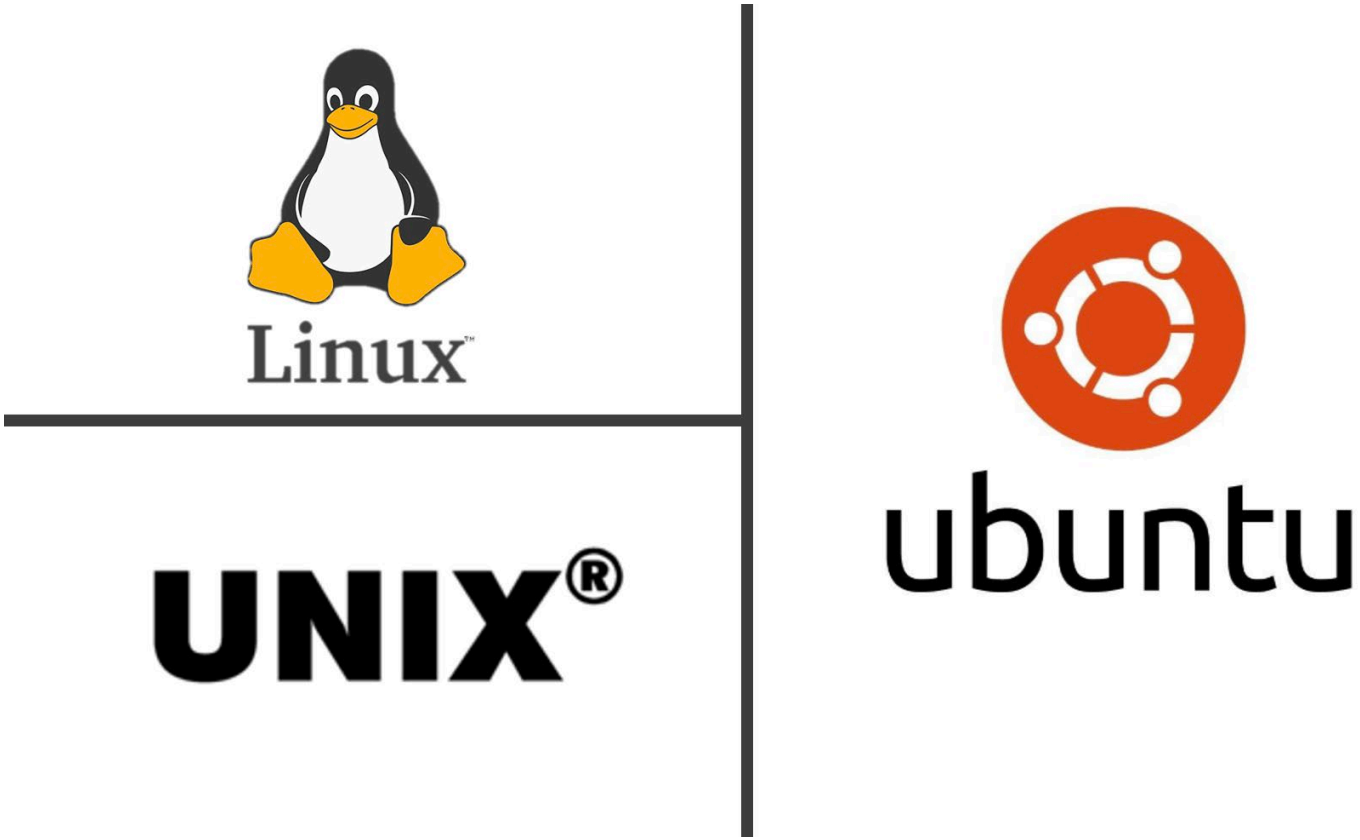
실행

```
$ cat mmap.dat
Hanbit Academy
$ ch9_5.out mmap.dat
1. Parent process : addr=Hanbit Academy
1. Child Process : addr=Hanbit Academy
2. Child Process : addr=xanbit Academy
2. Parent process : addr=xanbit Academy
3. Parent process : addr=xynbit Academy
3. Child Process : addr=xynbit Academy
$ cat mmap.dat
xynbit Academy
```

03. 파일 확장과 메모리 매핑

■ [예제 9-5] 데이터 교환하기

- **20행** : stat() 함수로 파일의 속성 정보를 검색해 파일 크기를 알아냄
- **25행** : 메모리에 매핑할 데이터 파일을 실행
- **30행** : 파일을 메모리에 매핑
- **37행** : fork() 함수를 실행해 자식 프로세스를 생성
  자식 프로세스는 부모 프로세스를 상속하므로 매핑된 메모리의 시작 주소인 addr을 공유
- **45행** : 자식 프로세스가 addr[0]의 내용을 'x'로 변경
  53행에 해당하는 출력 내용을 보면 자식 프로세스가 변경한 값을 부모 프로세스에서 읽을 수 있음
- **54행** : 부모 프로세스가 addr[1]의 내용을 'y'로 변경
  48행에 해당하는 출력 내용을 보면 부모 프로세스가 변경한 값을 자식 프로세스에서 읽을 수 있음
- **실행 결과** : 변경 내용을 부모 프로세스와 자식 프로세스가 각각 읽고 있음을 알 수 있음
  이 예제에서는 부모 프로세스와 자식 프로세스가 간단하게 출력만 수행하기 때문에 sleep() 함수를 사용해 실행 시간을 조정
  실제로는 wait() 함수를 사용하거나 시그널을 사용해 읽고 쓰는 순서를 조정할 수 있음



Chapter 10 파이프

목차

학습목표

- 파이프를 이용한 IPC 기법을 이해한다.
- 이름 없는 파이프를 이용해 통신 프로그램을 작성할 수 있다.
- 이름 있는 파이프를 이용해 통신 프로그램을 작성할 수 있다.

01. 개요

■ 파이프

▪ 파이프의 개요

- 두 프로세스 사이에서 한 방향으로 통신할 수 있도록 지원하는 것
- 셸에서 | 기호가 파이프를 의미
- 셸에서 파이프 기능은 한 명령의 표준 출력을 다음 명령에서 표준 입력으로 받아 수행하는 것을 의미

```
$ cat test.c | more
```

- 앞에 있는 명령인 cat test.c의 표준 출력을 다음 명령인 more의 표준 입력으로 사용
- 위 예를 실행하면 test.c를 화면 단위로 출력
- 파이프는 이름 없는 파이프(익명 파이프)와 이름 있는 파이프로 구분

01. 개요

■ 파이프

▪ 이름 없는 파이프: pipe

- 특별한 수식어 없이 그냥 파이프라고 하면 일반적으로 이름 없는 파이프(익명 파이프)를 의미
- 이름 없는 파이프는 부모-자식 프로세스 간에 통신을 할 수 있게 함
- 부모 프로세스에서 fork() 함수를 통해 자식 프로세스를 생성하고, 부모 프로세스와 자식 프로세스 간에 통신하는 것
- 따라서 '부모 프로세스 → 자식 프로세스' 또는 '자식 프로세스 → 부모 프로세스' 중 한 방향을 선택해야 함
- 파이프를 이용해 양방향 통신을 원할 경우 파이프를 2개 생성해야 함

표 10-1 이름 없는 파이프 생성 함수

기능	함수
간단한 파이프 생성	`FILE *popen(const char *command, const char *type);`
	`int pclose(FILE *stream);`
복잡한 파이프 생성	`int pipe(int pipefd[2]);`

01. 개요

■ 파이프

▪ 이름 있는 파이프: FIFO

- 부모-자식 프로세스 관계가 아닌 독립적인 프로세스들이 파이프를 이용하려면 파일처럼 이름이 있어야 함
- 이름 있는 파이프는 특수 파일의 한 종류로, FIFO라고도 함
- 모든 프로세스가 이름 있는 파이프를 이용해 통신할 수 있음

표 10-2 이름 있는 파이프 생성 명령과 함수

기능	명령과 함수
FIFO 생성 명령	mknod 파일명 p
	mkfifo [-m mode] … NAME …
FIFO 생성 함수	int mknod(const char *pathname, mode_t mode, dev_t dev);
	int mkfifo(const char *pathname, mode_t mode);

02. 이름 없는 파이프

■ 파이프 만들기 : popen(3)

```
#include <stdio.h>                                    [함수 원형]

FILE *popen(const char *command, const char *type);
```

- command : 셸 명령
- type : "r" 또는 "w"

- popen( ) 함수의 특징
  - 다른 프로세스와 통신하기 위해 파이프를 생성
  - 첫 번째 인자인 command에는 셸 명령을, 두 번째 인자인 type에는 "r"이나 "w"를 지정
  - "r"은 파이프를 읽기 전용으로, "w"는 쓰기 전용으로 실행
  - 내부적으로 fork() 함수를 실행해 자식 프로세스를 만들고, command에서 지정한 명령을 exec() 함수로 실행해 자식 프로세스가 수행하도록 함
  - popen() 함수는 자식 프로세스와 파이프를 만들고 mode의 값에 따라 표준 입출력을 연결
  - 리턴값은 파일 포인터로, 파일 입출력 함수에서 이 파일 포인터를 사용하면 파이프를 읽거나 쓸 수 있음
  - popen() 함수는 파이프 생성에 실패하면 널 포인터를 리턴

02. 이름 없는 파이프

■ 파이프 닫기 : pclose

```
#include <stdio.h>                                    [함수 원형]

int pclose(FILE *stream);
```

- pclose( ) 함수의 특징
  - pclose() 함수는 파일 입출력 함수처럼 인자로 지정한 파이프를 닫음
  - 관련된 waitpid() 함수를 수행하며 자식 프로세스들이 종료하기를 기다렸다가 리턴
  - pclose() 함수의 리턴 값은 자식 프로세스의 종료 상태
  - 이 함수는 파이프를 닫는 데 실패하면 –1을 리턴

02. 이름 없는 파이프

■ [예제 10-1] popen() 함수로 쓰기 전용 파이프 생성하기

```
01 #include <stdlib.h>
02 #include <stdio.h>
03
04 int main( ) {
05      FILE *fp;
06      int a;
07
08      fp = popen("wc -l", "w");
09      if (fp == NULL) {
10          fprintf(stderr, "popen failed₩n");
11          exit(1);
12      }
13
14      for (a = 0; a < 100; a++)
15          fprintf(fp, "test line₩n");
16
17      pclose(fp);
18 }
```

- **08행** "w" 모드를 사용해 쓰기 전용 파이프를 생성하고 자식 프로세스는 wc -l 명령을 수행하도록 함
  wc -l은 입력되는 데이터의 행 수를 출력하는 명령
- **14~15행** 부모 프로세스에서는 반복문을 사용해 문자열을 파이프로 출력
  앞서 언급했듯 이 자식 프로세스는 파이프로 입력되는 문자열을 읽어서 wc -l 명령을 수행

02. 이름 없는 파이프

■ [예제 10-2] popen() 함수로 읽기 전용 파이프 생성하기

```
01 #include <stdlib.h>
02 #include <stdio.h>
03
04 int main( ) {
05     FILE *fp;
06     char buf[256];
07
08     fp = popen("date", "r");
09     if (fp == NULL) {
10         fprintf(stderr, "popen failed₩n");
11         exit(1);
12     }
13
14     if (fgets(buf, sizeof(buf), fp) == NULL) {
15         fprintf(stderr, "No data from pipe!₩n");
16         exit(1);
17     }
18
19     printf("line : %s₩n", buf);
20     pclose(fp);
21 }
```

실행

```
$ ch10_2.out
line : 2021. 04. 18. (일) 21:02:46 KST
```

- **08행** 자식 프로세스에서는 date 명령을 수행
- **14행, 19행** 부모 프로세스에서는 자식 프로세스가 기록한 데이터를 14행에서 읽고 저장해 19행에서 출력
- **실행 결과** 현재 날짜와 시각이 출력됨, 자식 프로세스가 실행한 date 명령의 결과를 부모 프로세스가 읽어서 출력

02. 이름 없는 파이프

■ 파이프 만들기 : pipe(2)

```
#include <unistd.h>                                    [함수 원형]

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

- pipefd[2] : 파이프로 사용할 파일 기술자(2개)

- pipe( ) 함수의 특징
  - pipe() 함수는 인자로 크기가 2인 정수형 배열을 받음
  - 이 배열에 파일 기술자 2개를 저장
  - pipefd[0]은 읽기 전용으로 열고 pipefd[1]은 쓰기 전용으로 실행
  - pipe() 함수는 파이프를 생성하는 데 성공하면 0을, 실패하면 –1을 리턴

02. 이름 없는 파이프

■ pipe() 함수로 통신하는 과정

① pipe() 함수를 호출해 파이프에 사용할 파일 기술자를 얻음

- 파이프도 파일의 일종이므로 파일(파이프)을 읽고 쓸 수 있는 파일 기술자가 필요한데, 이를 pipe() 함수가 생성

그림 10-1 pipe() 함수를 이용한 파이프 생성

02. 이름 없는 파이프

■ pipe() 함수로 통신하는 과정

② fork() 함수를 수행해 자식 프로세스를 생성

- 이때 pipe() 함수에서 생성한 파일 기술자도 자식 프로세스로 복사
- 같은 파일 기술자를 부모 프로세스와 자식 프로세스가 모두 가지고 있

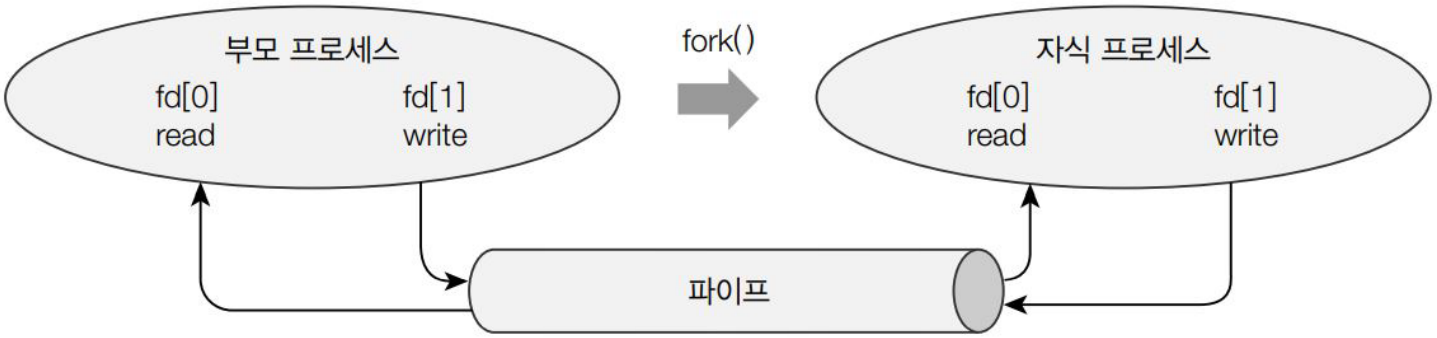


그림 10-2 자식 프로세스로 파일 기술자 복사

02. 이름 없는 파이프

■ pipe() 함수로 통신하는 과정

③ 파이프는 단방향 통신이므로 통신 방향을 결정

- 부모 프로세스에서 자식 프로세스로 통신할 수 있도록 파일 기술자들이 정리
- 부모 프로세스가 fd[1]에 쓴 내용을 자식 프로세스가 fd[0]에서 읽음
- 만약 파이프의 쓰기 부분이 닫혀 있다면 파이프에서 읽으려고 할 때 0이나 EOF가 리턴
- 파이프의 읽기 부분이 닫혀 있다면 파이프에 쓰려고 할 때 SIGPIPE 시그널이 발생

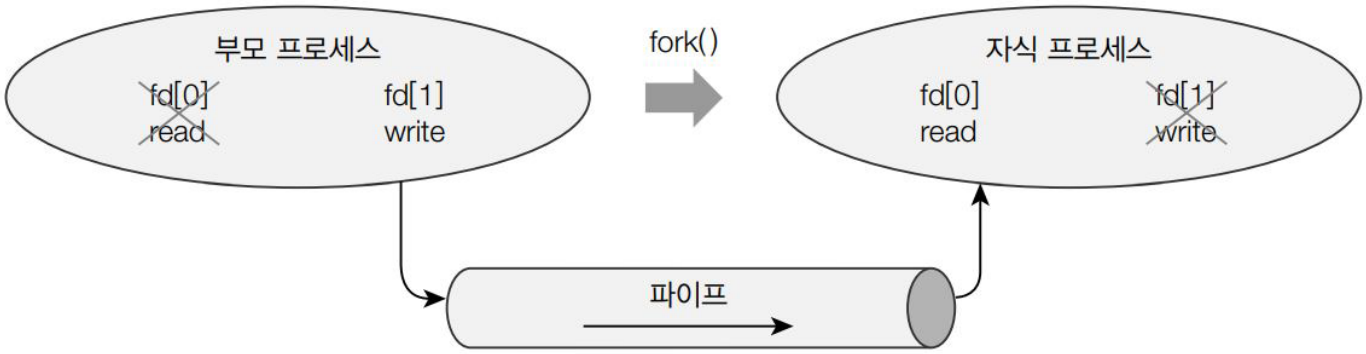


그림 10-3 부모 → 자식 방향으로 통신

02. 이름 없는 파이프

■ [예제 10-3] pipe() 함수로 통신하기

```
01 #include <sys/wait.h>
02 #include <unistd.h>
03 #include <stdlib.h>
04 #include <stdio.h>
05
06 int main( ) {
07     int fd[2];
08     pid_t pid;
09     char buf[257];
10     int len, status;
11
12     if (pipe(fd) == -1) {
13         perror("pipe");
14         exit(1);
15     }
16
17     switch (pid = fork( )) {
18         case -1 :
19             perror("fork");
20             exit(1);
21             break;
22         case 0 : /* child */
23             close(fd[1]);
24             write(1, "Child Process:", 15);
25             len = read(fd[0], buf, 256);
26             write(1, buf, len);
27             close(fd[0]);
28             break;
29         default :
30             close(fd[0]);
31             write(fd[1], "Test Message₩n", 14);
32             close(fd[1]);
33             waitpid(pid, &status, 0);
34             break;
35     }
36 }
```

실행

```
$ ch10_3.out
Child Process:Test Message
```

- **12행** pipe() 함수를 사용해 파이프를 생성, pipe() 함수의 인자로는 파일 기술자를 저장 할 배열을 지정
- **17행** fork() 함수를 사용해 자식 프로세스를 생성
- **22~28행** 자식 프로세스의 동작 부분임, 23행에서 fd[1]을 닫음, 이는 자식 프로세스에서 파이프를 읽기용으로 사용하겠다는 의미
  25행의 파이프 입력 부분인 fd[0]에서 문자열을 읽어들이고, 이를 26행에서 write() 함수를 사용해 화면에 출력
- **29~34행** 부모 프로세스의 동작 부분, 30행에서 fd[0]을 닫음, 이는 파이프를 출력용로 사용하겠다는 의미
  31행에서 fd[1]로 문자열을 출력, 33행에서는 자식 프로세스가 종료하기를 기다림
- **실행 결과** 부모 프로세스가 출력한 문자열을 자식 프로세스가 받아 출력하고 있음을 알 수 있음

02. 이름 없는 파이프

■ [예제 10-4] pipe() 함수로 명령 실행하기

```
01 #include <sys/wait.h>
02 #include <unistd.h>
03 #include <stdlib.h>
04 #include <stdio.h>
05
06 int main( ) {
07     int fd[2];
08     pid_t pid;
09
10     if (pipe(fd) == -1) {
11         perror("pipe");
12         exit(1);
13     }
14
15     switch (pid = fork( )) {
16         case -1 :
17             perror("fork");
18             exit(1);
19             break;
20         case 0 : /* child */
21             close(fd[1]);
22             if (fd[0] != 0) {
23                 dup2(fd[0], 0);
24                 close(fd[0]);
25             }
26             execlp("grep", "grep", "ssh", (char *)NULL);
27             exit(1);
28             break;
29         default :
30             close(fd[0]);
31             if (fd[1] != 1) {
32                 dup2(fd[1], 1);
33                 close(fd[1]);
34             }
35             execlp("ps", "ps", "-ef", (char *)NULL);
36             wait(NULL);
37             break;
38     }
39 }
```

실행

```
$ ch10_4.out
root        907      1  0  4월14 ?       00:00:00 sshd: /usr/sbin/sshd -D
                                         [listener] 0 of 10-100 startups
root      41326    907  0 20:02 ?        00:00:00 sshd: jw [priv]
jw        41498  41326  0 20:02 ?        00:00:00 sshd: jw@pts/0
jw        41748  41747  0 21:32 pts/0    00:00:00 grep ssh
```

02. 이름 없는 파이프

■ [예제 10-4] pipe() 함수로 명령 실행하기

- **10행** 파이프를 생성
- **15행** fork() 함수를 사용해 자식 프로세스를 생성, ps 명령의 결과는 기본으로 표준 출력으로 출력되고, grep 명령은 표준 입력을 통해 입력받음
  따라서 부모 프로세스와 자식 프로 세스 간의 통신이 표준 입출력 대신 파이프를 통해 이루어지도록 만들어야 함
- **21행** 자식 프로세스가 할 일은 부모 프로세스가 파이프로 출력하는 ps -ef 명령의 결과를 받아 grep ssh 명령을 수행하는 것
  따라서 파이프의 출력 부분이 필요 없으므로 fd[1]을 닫음
- **22~25행** fd[0]의 값이 0이 아니면, 즉 표준 입력이 아니면 fd[0]의 값을 표준 입력으로 복사 한 후 fd[0]을 닫음
  이제 자식 프로세스에서는 표준 입력을 fd[0]이 가리키는 파이프에서 읽음
- **26행** 자식 프로세스가 grep 명령을 exec() 함수로 호출, 이렇게 하면 grep 명령은 표준 입력을 통해 데이터를 읽어들이려 함
  이미 23행에서 표준 입력으로 파이프의 입력 파일 기 술자를 복사했으므로 결과적으로 파이프를 통해 데이터를 읽어들임
- **30행** 부모 프로세스의 동작을 살펴보면 자식 프로세스와 크게 다를 것이 없음을 알 수 있음
  우선 파이프의 입력 부분이 필요 없으므로 닫음
- **31~34행** 파이프의 출력 부분을 표준 출력으로 복사, 따라서 부모 프로세스에서 표준 출 력으로 무언가를 출력하면 파이프를 통해 출력
- **35행** exec() 함수를 사용해 ps -ef 명령을 실행, ps -ef 명령은 기본으로 표준 출력으로 출력하므로 결과가 파이프로 출력
  이 출력 결과를 자식 프로세스가 읽어들임
- **실행 결과** ps –ef | grep ssh 명령이 실행, 즉, 부모 프로세스에서 자식 프로세스로 ps –ef 명령의 실행 결과가 전달되었고 자식 프로세스는
  여기서 grep ssh 명령을 실행

02. 이름 없는 파이프

■ [예제 10-5] 양방향 통신하기

```
01 #include <sys/wait.h>
02 #include <unistd.h>
03 #include <stdlib.h>
04 #include <stdio.h>
05 #include <string.h>
06
07 int main( ) {
08     int fd1[2], fd2[2];
09     pid_t pid;
10     char buf[257];
11     int len, status;
12
13     if (pipe(fd1) == -1) {
14         perror("pipe");
15         exit(1);
16     }
17
18     if (pipe(fd2) == -1) {
19         perror("pipe");
20         exit(1);
21     }
22
23     switch (pid = fork( )) {
24         case -1 :
25             perror("fork");
26             exit(1);
27             break;
28         case 0 : /* child */
29             close(fd1[1]);
30             close(fd2[0]);
31             len = read(fd1[0], buf, 256);
32             write(1, "Child Process:", 15);
33             write(1, buf, len);
34
35             strcpy(buf, "Good₩n");
36             write(fd2[1], buf, strlen(buf));
37             break;
38         default :
39             close(fd1[0]);
40             close(fd2[1]);
41             write(fd1[1], "Hello₩n", 6);
42
43             len = read(fd2[0], buf, 256);
44             write(1, "Parent Process:", 15);
45             write(1, buf, len);
46             waitpid(pid, &status, 0);
47             break;
48     }
49 }
```

02. 이름 없는 파이프

■ [예제 10-5] 양방향 통신하기

- **08행** 파이프 2개를 생성할 것이므로 파일 기술자 배열을 2개 선언
- **13행** 파이프를 하나 생성, 생성된 파일 기술자 fd1은 부모 프로세스에서 자식 프로세스로 데이터를 보낼 때 사용
- **18행** 또 다른 파이프를 하나 생성, 생성된 파일 기술자 fd2은 자식 프로세스에서 부모 프로세스로 데이터를 보낼 때 사용
- **26행** 자식 프로세스는 부모 프로세스에서 오는 데이터를 읽는 데 사용할 fd1[0]을 남겨 두고 fd1[1]을 닫음
  또한 부모 프로세스로 데이터를 보내는 데 사용할 fd2[1]을 남겨두고 fd2[0]을 닫음
- **39~40행** 부모 프로세스는 자식 프로세스와 반대임, 자식 프로세스에 데이터를 보내는 데 사용할 fd1[1]을 남겨두고, 자식 프로세스에서 오는 데이터를 읽는 데 사용할 fd2[0]을 남겨둠

실행

```
$ ch10_5.out
Child Process:Hello
Parent Process:Good
```

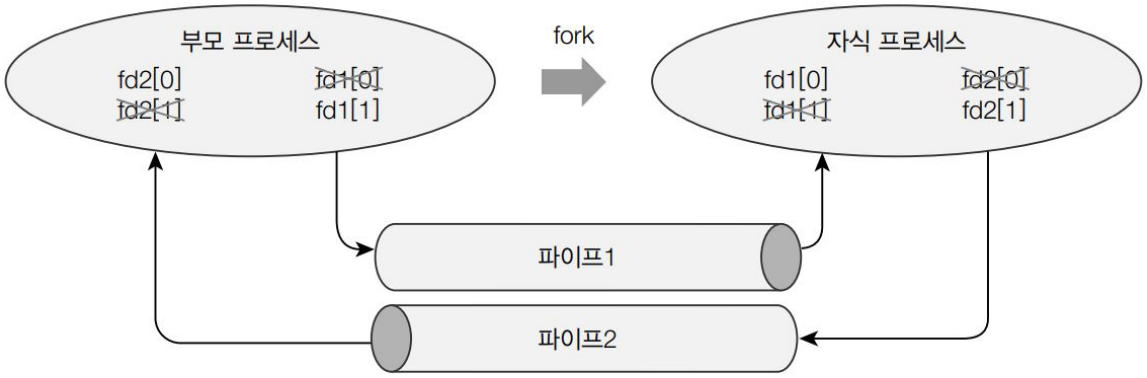


그림 10-4 양방향 통신 개념도

02. 이름 없는 파이프

■ [예제 10-5] 양방향 통신하기

- **31행, 41행** 41행에서 부모 프로세스가 쓴 데이터(Hello)를 31행에서 자식 프로세스가 읽어옴
- **32~33행** 31행에서 읽은 데이터를 화면으로 출력
- **35~36행**, 43행 부모 프로세스에 대한 응답으로 36행에서 자식 프로세스가 쓴 데이터(Good)를 43행에서 부모 프로세스가 읽음
- **44~45행** 43행에서 읽은 데이터를 화면으로 출력
- **실행 결과** 부모 프로세스와 자식 프로세스가 데이터를 주고받았음을 알 수 있음
  파일 기술자를 여닫는 과정이 다소 복잡하지만 이와 같이 파이프를 2개 생성하면 양방향 통신이 가능

03. 이름 있는 파이프

■ FIFO 특수 파일 생성 : mknod 명령

- mknod 명령의 특징
  - FIFO 파일뿐만 아니라 특수 파일도 생성하는 명령

```
mknod 파일명 p
```

  - -m 옵션은 새로 생성되는 FIFO 파일의 접근 권한을 지정
  - 이 옵션을 생략하면 umask 값에 따라 기본 권한을 설정

  - 예) mkfifo 명령으로 BIT_FIFO라는 FIFO 파일을 생성할 경우

```
$ mkfifo -m 0644 BIT_FIFO
$ ls -l BIT_FIFO
prw-r--r-- 1 jw jw 0 4월 19 20:51 BIT_FIFO
```

03. 이름 있는 파이프

■ 특수 파일 생성 : mknod()

[함수 원형]

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

int mknod(const char *pathname, mode_t mode, dev_t dev);
```

- pathname : 특수 파일을 생성할 경로
- mode : 특수 파일의 종류와 접근 권한 지정
- dev : 블록/문자 장치 설정값

- mknod( ) 함수의 특징
  - 첫 번째 인자인 pathname으로 지정한 경로에 특수 파일을 생성
  - 두 번째 인자인 mode에는 특수 파일의 종류와 접근 권한을 지정
  - mode에 지정하는 특수 파일의 종류는 다음 중 하나
    - **S_IFIFO** : FIFO 특수 파일
    - **S_IFBLK** : 블록 장치 특수 파일
    - **S_IFSOCK** : 유닉스 도메인 소켓 파일
    - **S_IFCHR** : 문자 장치 특수 파일
    - **S_IFREG** : 일반 파일
  - 세번째 인자인 dev는 생성하려는 특수 파일이 블록 장치 특수 파일이나 문자 장치 특수 파일일 때만 의미가 있음
  - mknod() 함수는 수행에 성공하면 0을, 실패하면 –1을 리턴

03. 이름 있는 파이프

■ FIFO 파일 생성 : mkfifo(3)

[함수 원형]

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

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

- pathname : FIFO 파일을 생성할 경로
- mode : 접근 권한 지정

- mkfifo( ) 함수의 특징
  - mkfifo() 함수는 pathname으로 지정한 경로에 접근 권한을 지정해 FIFO 파일을 생성

03. 이름 있는 파이프

■ [예제 10-6] 함수로 FIFO 파일 생성하기

```
01 #include <sys/types.h>
02 #include <sys/stat.h>
03 #include <stdlib.h>
04 #include <stdio.h>
05
06 int main( ) {
07     if (mknod("HAN-FIFO", S_IFIFO | 0644, 0) == -1) {
08         perror("mknod");
09         exit(1);
10     }
11
12     if (mkfifo("BIT-FIFO", 0644) == -1) {
13         perror("mkfifo");
14         exit(1);
15     }
16 }
```

실행

```
$ ch10_6.out
$ ls -l *FIFO
prw-r--r-- 1 jw jw 0  4월 19 21:02 BIT-FIFO
prw-r--r-- 1 jw jw 0  4월 19 21:02 HAN-FIFO
```

- **07행** mknod() 함수로 파일의 종류를 S_IFIFO로 지정
- **07행, 12행** 접근 권한은 두 함수 모두 0644로 지정
- **실행 결과** FIFO 파일이 생성

03. 이름 있는 파이프

■ [예제 10-7] (1) 서버 프로그램

```
01 #include <sys/stat.h>
02 #include <fcntl.h>
03 #include <unistd.h>
04 #include <stdlib.h>
05 #include <stdio.h>
06 #include <string.h>
07
08 int main() {
09     int pd, n;
10     char msg[] = "Hello, FIFO";
11
12     printf("Server =====₩n");
13
14     if (mkfifo("./HAN-FIFO", 0666) == -1) {
15         perror("mkfifo");
16         exit(1);
17     }
18
19     if ((pd = open("./HAN-FIFO", O_WRONLY)) == -1) {
20         perror("open");
21         exit(1);
22     }
23
24     printf("To Client : %s₩n", msg);
25
26     n = write(pd, msg, strlen(msg)+1);
27     if (n == -1) {
28         perror("write");
29         exit(1);
30     }
31     close(pd);
32 }
```

실행

```
$ ch10_7s.out
Server =====
To Client : Hello, FIFO
```

- **14행** mkfifo() 함수로 HAN-FIFO라는 FIFO 파일을 생성
- **19행** 서버에서 클라이언트로 데이터를 전송할 것이므로 HAN-FIFO 파일을 쓰기 전용으로 실행
- **26행** 클라이언트로 "Hello, FIFO"라는 메시지를 전송

03. 이름 있는 파이프

■ [예제 10-7] (2) 클라이언트 프로그램

```
01 #include <fcntl.h>
02 #include <unistd.h>
03 #include <stdlib.h>
04 #include <stdio.h>
05
06 int main() {
07     int pd, n;
08     char inmsg[80];
09
10     if ((pd = open("./HAN-FIFO", O_RDONLY)) == -1) {
11         perror("open");
12         exit(1);
13     }
14
15     printf("Client =====₩n");
16     write(1, "From Server :", 13);
17
18     while ((n=read(pd, inmsg, 80)) > 0)
19         write(1, inmsg, n);
20
21     if (n == -1) {
22         perror("read");
23         exit(1);
24     }
25
26     write(1, "₩n", 1);
27     close(pd);
28 }
```

```
$ ch10_7c.out
Client =====
From Server : Hello, FIFO
```

- **10행** HAN-FIFO 파일을 읽기 전용으로 실행, 클라이언트 프로그램은 이미, HAN-FIFO 파일이 있다고 가정하고 파일을 열고 있으므로 서버 프로그램을 먼저 실행해야 함
- **18~19행** 반복문을 수행하며 FIFO 파일에서 데이터를 읽어와 출력
- **실행 결과** 서버에서 보낸 메시지를 클라이언트가 정확히 받아 출력하고 있음
  서버 프로세스를 먼저 실행하면 클라이언트 프로세스가 실행할 때까지 기다리고 있다가 실행

시스템 프로그래밍

리눅스&유닉스

감사합니다.