REPORT

과목명	시스템프로그래밍
교수님	김성우
학 과	응용소프트웨어공학과
학 번	20193139
이 름	박수빈
날 짜	2023.11.29

東義大學校
DONG-EUI UNIVERSITY

1. 실습에 필요한 준비 사항

1. 파일 처리 함수에 대하여 조사하고 이해한다.

- 저수준 함수들

▣ UNIX 파일 접근 프리미티브(기본함수)

◈ 프로그램 안에서 파일들을 다루기 위해 UNIX 가 제공하는 기본적인 프리미티브들

◈ UNIX 커널에 의해 제공되는 I/O 장치에 직접 접근을 제공하는 시스템 호출들의 작은 집합

이름	의 미
open	읽거나 쓰기 위해 파일을 열거나, 또는 빈 파일을 생성
creat	빈 파일을 생성한다
close	열려진 파일을 닫는다
read	파이로부터 정보를 추출한다
write	파일에 정보를 기록한다
lseek	파일 안의 지정된 바이트로 이동한다.
unlink	파일을 제거한다.
remove	파일을 제거하는 다른 방법
fcntl	한 파일에 연관된 속성을 제어한다.

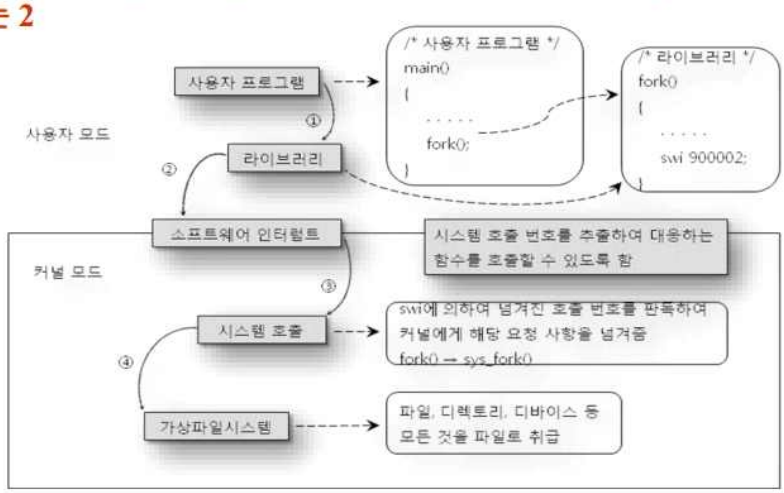
- 시스템 호출 동작과정

▣ 시스템 호출 처리 과정

◈ 리눅스의 시스템 호출 함수 중 하나인 fork()의 처리 과정

➢ 소프트웨어 인터럽트(swi 900002)에 의해 사용자 모드에서 커널 모드로 전환되어 커널 자원 접근 가능

➢ 시스템 호출 테이블의 시작점이 900000이고 fork() 시스템 호출 번호는 2



- open 함수

▣ 기능

◈ 기존의 파일을 읽거나 쓰기 전에 항상 파일 개방

▣ 사용법

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

int open(const char *pathname, int flags, [mode_t mode]);
```

- ➢ **pathname : 개방될 파일의 경로**
- ➢ **Flags : 접근 방식 지정**
  - **O_RDONLY** 읽기 전용으로 개방
  - **O_WRONLY** 쓰기 전용으로 개방
  - **O_RDWR** 읽기 및 쓰기 용으로 개방
  - 아래 상수는 위의 상수와 **OR** 해서 사용
  - **O_CREAT** 파일이 없으면 생성, 아래 **mode** 필요함
  - **O_APPEND** 파일 쓰기 시 파일 끝에 추가
  - **O_TRUNC** 파일이 이미 존재하고 쓰기 권한으로 열리면 크기를 0
- ➢ **Mode : 보안과 연관, 생략 가능**

- close 함수

▣ 기능

◈ **Open** 의 역, 개방 중인 파일을 닫음

▣ 사용법

```
#include <unistd.h>

int close(int filedes);
```

- ➢ **filedes : 닫혀질 파일 기술자**

```
filedes = open("file", O_RDONLY);
.
.

close(filedes);
```

- creat 함수

▣ 기능

◈ 파일을 생성하는 대안적 방법

▣ 사용법

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

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

➢ **pathname** : 개방될 파일의 경로

➢ **Mode** : 필요한 접근 허가 제시, 생략 가능

```
filedes = creat("/tmp/newfile", 0644);
filedes = open("/tmp/newfile", O_WRONLY | O_CREAT | O_TRUNC, 0644);
```

- read 함수

▣ 기능

◈ 파일로부터 임의의 문자들 또는 바이트들을 호출 프로그램의 제어 하에 있는 버퍼로 복사

▣ 사용법

```
#include <unistd.h>

ssize_t read(int filedes, void *buffer, size_t n);
```

➢ **filedes** : 이전의 **open** 또는 **creat** 로부터 얻은 파일 기술자

➢ **buffer** : 자료가 복사되어질 문자 배열의 포인터

➢ **n** : 파일로부터 읽혀질 바이트의 수

- write 함수

▣ 기능

◈ 문자 배열인 프로그램 버퍼로부터 외부 파일로 자료를 복사

▣ 사용법

```
#include <unistd.h>

ssize_t write(int filedes, const void *buffer, size_t n);
```

➢ **filedes** : 이전의 **open** 또는 **creat** 로부터 얻은 파일 기술자

➢ **buffer** : 자료가 복사되어질 문자 배열의 포인터

➢ **n** : 파일로 쓰여질 바이트의 수

- lseek 함수

▣ 기능

◈ 읽기쓰기 포인터의 위치 (다음에 읽거나 쓸 바이트 위치) 변경

◈ 파일에 대한 임의 접근 가능

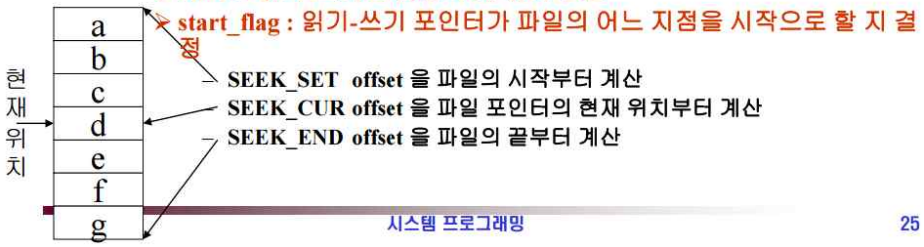
▣ 사용법

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

off_t lseek(int filedes, off_t offset, int start_flag);
```

➢ **filedes :** 개방되어 있는 파일의 파일 기술자

➢ **offset :** 읽기-쓰기 포인터의 새 위치 결정

➢ **start_flag :** 읽기-쓰기 포인터가 파일의 어느 지점을 시작으로 할 지 결정



2. 입출력 장치 파일에 대하여 조사하고 요약하여 보고서에 정리하여 본다.

▣ **UNIX 시스템은 수행중인 한 프로그램에 대해 자동적으로 세 개의 파일을 개방**

◈ 표준 입력 (0), 표준 출력 (1), 표준 오류(2)

◈ **# prog_1 | prog_2 : prog_1 의 표준 출력을 prog_2 의 표준 입력으로 사용**

◈ 표준 입력과 출력 파일 기술자는 유연하고 일관된 프로그램 구성 수단

▣ 표준 입력

◈ 디폴트로 키보드로부터 자료를 입력받음

◈ **# prog_name < infile**

▣ 표준 출력

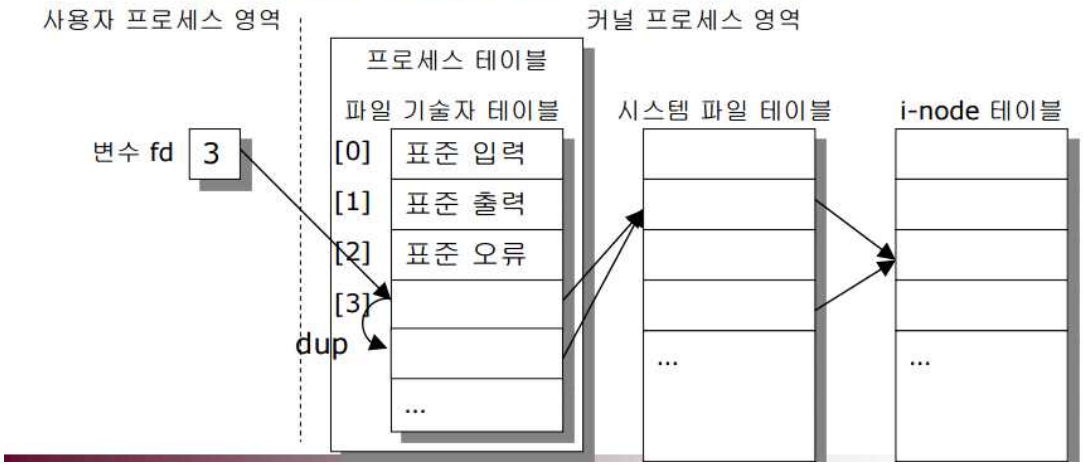
◈ 디폴트로 단말기 화면으로 자료를 출력

◈ **# prog_name > outfile**

▣ 시스템의 파일 관리

◈ 파일 기술자 테이블, 시스템 파일 테이블, i-node 테이블을 통해 각 프로세스가 사용하는 파일 관리

- ➢ 시스템 파일 테이블 – 시스템의 모든 파일에 대한 정보(상태, 현재 오프셋 등) 수록
- ➢ i-node 테이블 – 실제 저장된 파일의 정보 수록

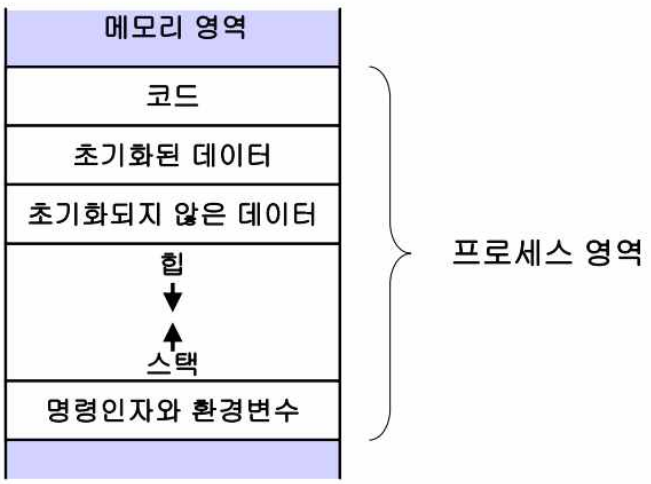


3. 교재에서 프로세스 및 시그널에 대한 내용을 읽고 이해한다.

▣ 프로그램 – 디스크 상에 저장되어 있는 실행 가능한 파일

▣ 프로세스 – 수행중인 프로그램

◈ 쉘 은 하나의 명령을 수행하기 위해 어떤 프로그램을 시작할 때마다 새로운 프로세스를 생성



▣ 프로세스 – 수행중인 프로그램

◈ 쉘 은 하나의 명령을 수행하기 위해 어떤 프로그램을 시작할 때마다 새로운 프로세스를 생성

- ➢ **cat file1 file2** # 프로세스 생성
- ➢ **ls | wc –l** # 2 개의 프로세스 생성

◈ **UNIX** 프로세스 환경은 디렉토리 트리처럼 계층적인 구조

- ➢ 가장 최초의 프로세스는 **init** 이며 모든 시스템과 사용자 프로세스의 조상

◈ 프로세스 시스템 호출들

이름	의 미
fork	호출 프로세스와 똑같은 새로운 프로세스를 생성
exec	한 프로세스의 기억공간을 새로운 프로그램으로 대체
wait	프로세스 동기화 제공. 연관된 다른 프로세스가 끝날 때까지 기다린다
exit	프로세스를 종료

▣ fork 시스템 호출

◈ 기능

- ➢ 기본 프로세스 생성 함수
- ➢ 성공적으로 수행되면 호출 프로세스(부모 프로세스) 와 똑같은 새로운 프로세스(자식 프로세스) 생성

◈ 사용법

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

pid_t fork(void);
```

◈ 반환 값(return value)

- ➢ 정상 실행
  - 실행(부모) 프로세스는 : 생성(자식) 프로세스의 프로세스 ID 반환
  - 생성(자식) 프로세스는 : 0을 반환
- ➢ 이상 실행 : 음수 값을 반환

◈ 프로세스 식별번호(IDentifier)

▣ exit 시스템 호출

◆ 기능

➢ 프로세스 종료

◆ 사용법

```
#include <stdlib.h>

void exit (int status);
```

➢ **status :** 종료 상태 를 나타냄

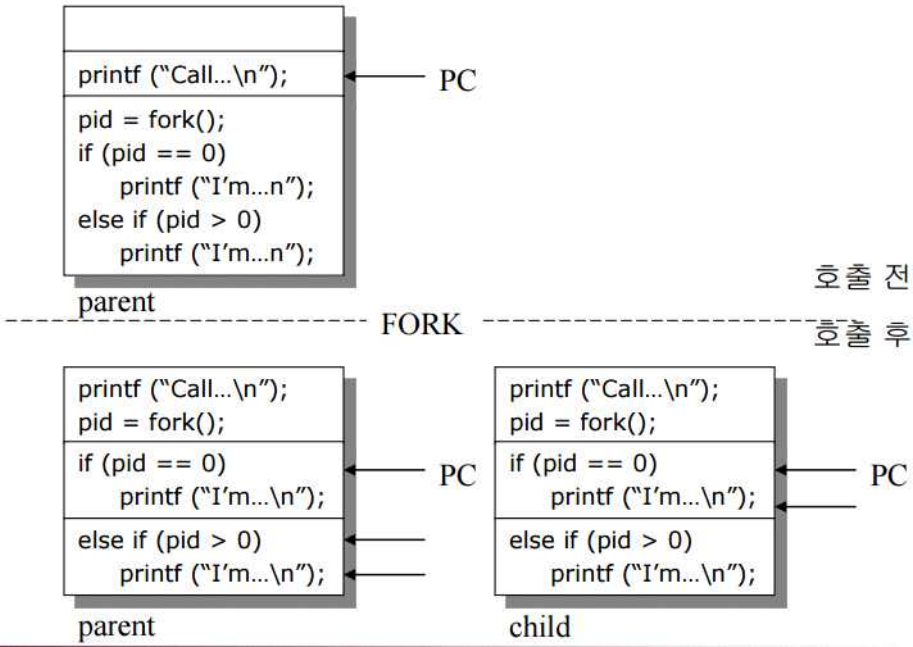
➢ 반환값: 정상적 종료: **0**, 그렇지 않을 경우: **0** 이 아닌 값

◆ 모든 개방된 파일 기술자들을 닫는다.

▣ 프로세스 생성 함수 사용 예

◆ 프로세스 생성 예제 프로그램

➢ **fork** 호출 후 두 프로세스는 바로 다음 문장부터 수행 계속



▣ atexit 함수

◈ 기능

➢ 실행 프로세스 종료 시 호출되는 루틴 설정

◈ 사용법

```
#include <stdlib.h>
void atexit (void (*func) (void));
```

➢ 프로그래머가 종료 루틴을 정의

▣ wait 시스템 호출

◈ 기능

➢ 자식 프로세스들 중 하나가 수행을 마치기를 기다린다

◈ 사용법

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

pid_t wait(int *status);
```

➢ **status : wait**가 복귀될 때 유용한 상태 정보 (퇴장 상태)

➢ 때로 부모 프로세스가 **fork** 로 자식을 만든 직후에 호출

```
cpid = fork();
if (cpid == 0) {          /* 자식 : 무언가 일을 수행한다 ... */
} else {                  /* 부모 : 자식을 기다린다 ... */
        cpid = wait (&status);
}
```

▣ 시그널

◈ 다른 프로세스에게 이벤트 발생을 알리는 소프트웨어 인터럽트

◈ 시그널 발생 예

➢ 프로그램 실행 시 **Ctrl-C**(인터럽트 키) 를 눌러 강제 종료시킬 때

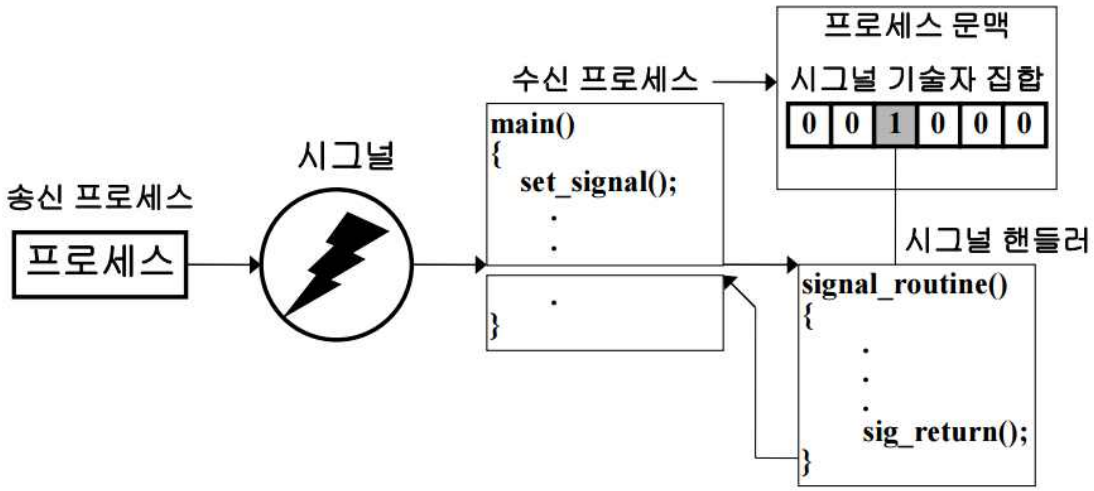
➢ 백그라운드 작업을 종료시킬 때, **kill** 명령 사용

▣ 시그널 동작

◈ 1. 기본적으로 설정한 동작 수행

◈ 2. 시그널을 무시하고 프로그램 수행 계속

◈ 3. 미리 정해진 일을 수행



▣ sigaction 시스템 호출

◈ 기능

➢ 프로세스가 특정 시그널에 대해 다음 세 가지 행동 중 하나를 지정

- 프로세스는 종료하고 코어 덤프
- 시그널을 무시
- 시그널 핸들러에 지정된 함수를 수행

◈ 사용법

```
#include <signal.h>
int sigaction(int signo, const struct sigaction *act,
                         struct sigaction *oact);
```

➢ **signo: signal number**

➢ **act : sigaction 구조체**

➢ **oact: 이전 설정값**

▣ 데몬(daemon) 프로세스

◈ 백그라운드로 돌면서 여러 작업을 하는 프로그램

◈ 수행 과정

➢ 프로세스를 백그라운드로 만든다 (fork 호출 및 부모 프로세스 종료)

➢ 제어 터미널이 없는 세션과 그룹 리더로 설정 (setsid() 호출)

– 제어 터미널이 있으면 터미널 문자 등으로 죽을 수 있음

➢ 현재 작업 디렉토리를 루트 디렉토리("/")로 설정

➢ umask(0)로 변경하여 파일 접근 권한 미리 설정

➢ 열고 있는 모든 파일들을 닫는다

– 표준입력, 표준출력, 표준 오류는 /dev/null 로 설정

➢ 오류 메시지를 남기기 위해 로그 파일 설정

◈ daemon 함수 활용

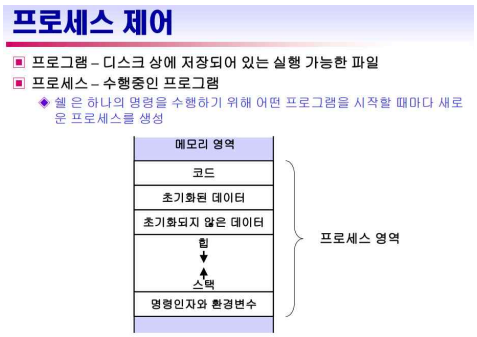
```
#include <unistd.h>

int daemon(int nochdir, int noclose);
```

➢ NOCHDIR가 0이면, 현재 디렉토리를 /로 바꿈

➢ NOCLOSE가 0이면, 표준입력, 표준출력, 표준 오류 파일을 닫음

4. 리눅스의 프로세스 모델과 시그널/쓰레드와의 관계에 대하여 조사하고 요약하여 보고서에 정리하여 본다.

- 리눅스 프로세스 모델



- 시그널/쓰레드와의 관계

1. 시그널 처리는 기본적으로 프로세스 단위로 처리를 합니다.

2. 시그널은 모든 쓰레드에게 시그널을 보냅니다. 이런 현상을 막기 위해 "시그널 마스크"를 이용하여 제어가 가능합니다.

5. 교재에서 프로세스 간 통신에 대한 내용을 읽고 이해한다.

프로세스간 통신

▣ 프로세스간 통신 기법
- ◈ 시그널, 파일 잠금, 파이프, 메시지 큐, 세마포어, 공유 메모리, 소켓 등

▣ 파일을 이용한 레코드 잠금
- ◈ 레코드 잠금(record locking)
  - ➢ 프로세스가 특정 파일의 일부 레코드에 대하여 잠금 기능 설정
  - ➢ 다른 프로세스로 하여금 이 파일에 접근하지 못하도록 함
- ◈ 종류
  - ➢ 읽기 잠금 – 다른 프로세스들이 해당 영역에 쓰기 잠금 불가
  - ➢ 쓰기 잠금 – 다른 프로세스들이 해당 영역에 읽기와 쓰기 잠금 모두 불가

파일을 이용한 레코드 잠금

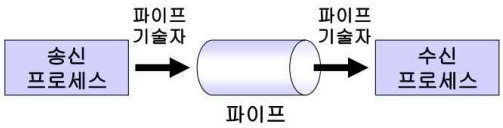
▣ **fcntl** 시스템 호출
- ◈ 기능
  - ➢ 파일 제어 : **fd** 가 가리키는 파일을 **cmd** 명령에 따라 제어한다.
- ◈ 사용법

```
#include <fcntl.h>

int fcntl(int fd, int cmd, struct flock *lock);
```

  - ➢ **cmd : fcntl** 이 어떻게 동작할 것인지 결정
    - – **F_GETLK** : 레코드 잠금 정보 획득, 정보는 셋째 인자 **lock** 에 저장
    - – **F_SETLK** : 파일에 레코드 잠금 적용, 불가하면 즉시 –1 반환
    - – **F_SETLKW** : 파일에 레코드 잠금 적용, 불가하면 잠금 해제를 기다림
  - ➢ 성공적인 호출에 대하여, 반환값은 동작에 달려 있다

▣ 레코드 잠금의 단점
- ◈ 파일을 이용 – 추가적인 자원 낭비 및 비효율적
- ◈ 경쟁 문제
- ◈ 외부에서 파일에 접근할 때의 보안 문제

▣ 파이프
- ◈ **UNIX** 의 고유한 프로세스간 통신 방식
- ◈ 한 프로세스를 다른 프로세스와 연결시켜 주는 단방향 채널



▣ **pipe** 시스템 호출
- ◈ 기능
  - ➢ 파이프 생성
  - ➢ 파이프를 가리키는 파일 기술자 쌍을 생성하고, 이를 **filedes** 에 저장
- ◈ 사용법

```
#include <unistd.h>

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

  - ➢ **filedes[0]**은 읽기 위한 파이프
  - ➢ **filedes[1]**은 쓰기 위한 파이프
- ◈ 반환값
  - ➢ 정상적 종료: **0**, 그렇지 않을 경우: **0** 이 아닌 값
  - ➢ 오류 번호(**errno**)
    - – **EMFILE** : 사용자 프로세스가 너무 많은 파일 기술자를 사용하는 경우
    - – **ENFILE** : 시스템 파일 테이블이 꽉 찬 경우
    - – **EFAULT** : **filedes** 변수가 유효하지 못하는 경우

▣ select 시스템 호출

◈ 기능

- 다중 입출력 관리
- 지정된 파일 기술자 집합 중 읽기와 쓰기가 준비된 것이 있는지, 또는 오류가 있는지 등을 검사

◈ 사용법

```
#include <sys/time.h>

int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct
timeval *timeout);
void FD_CLR(int fd, fd_set *set);
void FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
```

- **select 첫째 인자 n은 검사할 파일 기술자 번호 최대값+1**
- **FD_SET : 파일 기술자를 집합에 추가**
- **FD_CLR : 파일 기술자를 집합에서 뺀다**
- **FD_ISSET : 파일 기술자가 집합의 일부분인지 아닌지를 검사**
- **FD_ZERO : 파일 기술자 집합을 초기화**

6. 리눅스에서 비동기 통신 기법에 대하여 조사하고 요약하여 보고서에 정리하여 본다.

POSIX 비동기 입출력

▣ 특징

◈ 프로그램이 입출력 연산을 수행하는 동안 블록되지 않고 비동기식으로 수행

◈ 라이브러리 수준에서 구현됨

◈ 주의: 리눅스 커널이 지원하는 별도의 비동기 입출력 기능이 있음

▣ 관련 함수

◈ 사용법

```
#include <aio.h>
#include <unistd.h>
#include <sys/types.h>

int aio_read(struct aiocb *__aiocbp); /* 비동기 읽기 연산 시작 */
int aio_write(struct aiocb *__aiocbp); /* 비동기 쓰기 연산 시작 */
int aio_error(const struct aiocb *__aiocbp); /* aio_read/aio_write 연산 완료 대기 및 상태 반환 */
ssize_t aio_return(struct aiocb *__aiocbp); /* aio_read/aio_write를 통해 전송된 바이트 수 반환 */
int aio_cancel(int __fildes, struct aiocb *__aiocbp); /* 처리 중인 비동기 IO 취소 */
void aio_suspend(const struct aiocb *__list[], int __nent, /* 지정 요청 완료 까지 프로세스 중지 */
                 const struct timespec *__restrict __timeout);
int lio_listio(int __mode, const struct aiocb *__list[__restrict_arr], /* 다중 비동기 연산 요청 */
               int __nent, struct sigevent *__restrict __sig);
```

▣ AIO 제어블록

◈ AIO를 제어하는 자료구조

```
struct aiocb
{
   int aio_fildes;              /* 파일 기술자. */
   int aio_lio_opcode;          /* 수행할 연산(lio_listio와 연관). */
   int aio_reqprio;             /* 요청 우선순위 오프셋. */
   volatile void *aio_buf;      /* 버퍼. */
   size_t aio_nbytes;           /* 전송 바이트 수. */
   struct sigevent aio_sigevent; /* 시그널 번호 및 값. */

   /* Internal fields */
   ...

};
```

2. 실습 사항

1. 자신의 github 저장소에 lab3 프로젝트를 생성하고 과제 프로그램을 업로드하였습니다.
git url: https://github.com/toughC/lab3
2~10. github 저장소에 제출하였습니다.

3. 검토

1학년 때 c언어를 배우면서 큰 어려움이 없었는데, 쉘 스크립트 짜면서 문자열을 사용하는 방법을 다시 생각하고 이해하면서 코드 작성할 때 어려움이 있었습니다. 유튜브 영상이나 서칭하면서 에러난 부분을 찾고 이해하고 다시 작성하면서 제 부족함을 찾을 수 있는 기회가 되었습니다.