

REPORT



파일처리, 프로세스 및 프로세스 간 통신

제 출 일 20 . 11 . 16

교 수 명 김성우

과 목 시스템 프로그래밍

학 과 컴퓨터소프트웨어 공학과

학 년 3학년

학 번 20163346

이 름 김창호

I. 내용 정리

1. 리눅스의 파일 처리 함수

1) 저수준 파일 입출력

- UNIX/LINUX에서 사용하는 기본 방법
- 정수 파일 디스크립터(descriptor)번호 사용
- 유닉스 커널의 시스템을 호출하여 파일 입출력 수행.
- 시스템 호출을 이용하므로 파일에 좀 더 빠르게 접근이 가능하고, 바이트 단위로 파일의 내용을 다루므로, 일반 파일뿐만 아니라 특수 파일도 읽고 쓸 수 있다.
- 하지만, 바이트 단위로만 입출력을 수행하므로 응용프로그램을 작성하려면 바이트를 적당한 형태의 데이터로 변환하는 함수를 추가하는 등 추가적인 기능을 구현해야 한다.

이름	의 미
open	읽거나 쓰기 위해 파일을 열거나, 또는 빈 파일을 생성
creat	빈 파일을 생성한다
close	열려진 파일을 닫는다
read	파일로부터 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.

그림 1. 저수준 파일 처리 함수 종류

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

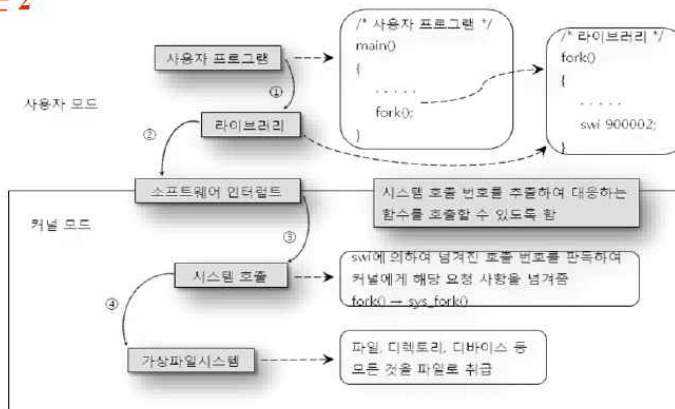


그림 2. 리눅스 시스템 호출함수 fork()의 처리 과정

표 1. 저수준 입출력 함수

<div> <div>(1) open 시스템 호출</div> <div> <div>■ 기능</div> <div>◆ 기존의 파일을 읽거나 쓰기 전에 항상 파일 개방</div> </div> <div> <div>■ 사용법</div> <div> <pre>#include <sys/types.h> #include <sys/stat.h> #include <fcntl.h> int open(const char *pathname, int flags, [mode_t mode]);</pre> <div> <div>➢ pathname : 개방될 파일의 경로</div> <div>➢ Flags : 접근 방식 지정</div> <div> <div>- O_RDONLY</div> <div>읽기 전용으로 개방</div> <div>- O_WRONLY</div> <div>쓰기 전용으로 개방</div> <div>- O_RDWR</div> <div>읽기 및 쓰기 용으로 개방</div> <div>- 아래 상수는 위의 상수와 OR 해서 사용</div> <div>- O_CREAT</div> <div>파일 없으면 생성, 아래 mode 필요함</div> <div>- O_APPEND</div> <div>파일 쓰기 시 파일 끝에 추가</div> <div>- O_TRUNC</div> <div>파일 이미 존재하고 쓰기 권한으로 열리면 크기를 0</div> </div> <div>➢ Mode : 보안과 연관, 생략 가능</div> </div> </div> </div> </div>	<div> <div>■ 파일 생성(O_CREAT)시 접근 권한(mode) 지정</div> <div> <table> <tr> <th>사용자 권한</th><th>그룹 권한</th><th>기타 사용자 권한</th></tr> <tr> <td>S_IRWXU 읽기, 쓰기, 실행 가능</td><td>S_IRWXG 읽기, 쓰기, 실행 가능</td><td>S_IRWXO 읽기, 쓰기, 실행 가능</td></tr> <tr> <td>S_IRUSR 읽기 가능</td><td>S_IRGRP 읽기 가능</td><td>S_IROTH 읽기 가능</td></tr> <tr> <td>S_IWUSR 쓰기 가능</td><td>S_IWGRP 쓰기 가능</td><td>S_IWOTH 쓰기 가능</td></tr> <tr> <td>S_IXUSR 실행 가능</td><td>S_IXGRP 실행 가능</td><td>S_IXOTH 실행 가능</td></tr> </table> <div> <pre>[linux@seps ch7]\$ ls -al 합계 68 drwxrwxr-x 2 linux linux 4096 2월 11 08:56 . drwxrwxr-x 12 linux linux 4096 2월 3 11:32 .. -rwxrwxr-x 1 linux linux 10502 2월 11 08:56 file_creat -rw-rw-r-- 1 linux linux 727 2월 11 08:56 file_creat.c -rw-r--r-- 1 linux linux 16 2월 11 08:56 t.txt</pre> </div> </div> </div>	사용자 권한	그룹 권한	기타 사용자 권한	S_IRWXU 읽기, 쓰기, 실행 가능	S_IRWXG 읽기, 쓰기, 실행 가능	S_IRWXO 읽기, 쓰기, 실행 가능	S_IRUSR 읽기 가능	S_IRGRP 읽기 가능	S_IROTH 읽기 가능	S_IWUSR 쓰기 가능	S_IWGRP 쓰기 가능	S_IWOTH 쓰기 가능	S_IXUSR 실행 가능	S_IXGRP 실행 가능	S_IXOTH 실행 가능
사용자 권한	그룹 권한	기타 사용자 권한														
S_IRWXU 읽기, 쓰기, 실행 가능	S_IRWXG 읽기, 쓰기, 실행 가능	S_IRWXO 읽기, 쓰기, 실행 가능														
S_IRUSR 읽기 가능	S_IRGRP 읽기 가능	S_IROTH 읽기 가능														
S_IWUSR 쓰기 가능	S_IWGRP 쓰기 가능	S_IWOTH 쓰기 가능														
S_IXUSR 실행 가능	S_IXGRP 실행 가능	S_IXOTH 실행 가능														
<div> <div>(2) close 시스템 호출</div> <div> <div>■ 기능</div> <div>◆ Open 의 역, 개방 중인 파일을 닫음</div> </div> <div> <div>■ 사용법</div> <div> <pre>#include <unistd.h> int close(int filedes);</pre> <div>➢ filedes : 닫혀질 파일 기술자</div> <pre>filedes = open("file", O_RDONLY); . . close(filedes);</pre> </div> </div> </div>	<div> <div>(3) create 시스템 호출</div> <div> <div>■ 기능</div> <div>◆ 파일을 생성하는 대안적 방법</div> </div> <div> <div>■ 사용법</div> <div> <pre>#include <sys/types.h> #include <sys/stat.h> #include <fcntl.h> int creat(const char *pathname, mode_t mode);</pre> <div> <div>➢ pathname : 개방될 파일의 경로</div> <div>➢ Mode : 필요한 접근 허가 제시, 생략 가능</div> </div> <pre>filedes = creat("/tmp/newfile", 0644); filedes = open("/tmp/newfile", O_WRONLY O_CREAT O_TRUNC, 0644);</pre> </div> </div> </div>															
<div> <div>(3) read 시스템 호출</div> <div> <div>■ 기능</div> <div>◆ 파일로부터 임의의 문자들 또는 바이트들을 호출 프로그램의 제어 하에 있는 버퍼로 복사</div> </div> <div> <div>■ 사용법</div> <div> <pre>#include <unistd.h> ssize_t read(int filedes, void *buffer, size_t n);</pre> <div> <div>➢ filedes : 이전의 open 또는 creat 로부터 얻은 파일 기술자</div> <div>➢ buffer : 자료가 복사되어질 문자 배열의 포인터</div> <div>➢ n : 파일로부터 읽혀질 바이트의 수</div> </div> </div> </div> </div>	<div> <div>(4) write 시스템 호출</div> <div> <div>■ 기능</div> <div>◆ 문자 배열인 프로그램 버퍼로부터 외부 파일로 자료를 복사</div> </div> <div> <div>■ 사용법</div> <div> <pre>#include <unistd.h> ssize_t write(int filedes, const void *buffer, size_t n);</pre> <div> <div>➢ filedes : 이전의 open 또는 creat 로부터 얻은 파일 기술자</div> <div>➢ buffer : 자료가 복사되어질 문자 배열의 포인터</div> <div>➢ n : 파일로 쓰여질 바이트의 수</div> </div> </div> </div> </div>															
<div> <div>(5) lseek 시스템 호출, 임의의 접근</div> <div> <div>■ offset 은 음수 가능</div> <div>◆ 시작점으로부터 거꾸로 이동 가능</div> </div> <div> <div>■ 기존 파일의 끝에 추가하여 쓰는 예제</div> <div> <pre>filedes = open (filename, O_RDWR); lseek (filedes, (off_t) 0, SEEK_END); write (filedes, outbuf, OBSIZE);</pre> <div>◆ 또 다른 방법 : O_APPEND 추가하여 open 호출</div> </div> <div> <div>■ 파일의 크기 알아내는 예제</div> <pre>off_t filesize; int filedes; . . filesize = lseek (filedes, (off_t) 0, SEEK_END);</pre> </div> </div> </div>	<div> <div>■ 기능</div> <div> <div>◆ 읽기쓰기 포인터의 위치 (다음에 읽거나 쓸 바이트 위치) 변경</div> <div>◆ 파일에 대한 임의의 접근 가능</div> </div> <div> <div>■ 사용법</div> <div> <pre>#include <sys/types.h> #include <unistd.h> off_t lseek(int filedes, off_t offset, int start_flag);</pre> <div> <div>➢ filedes : 개방되어 있는 파일의 파일 기술자</div> <div>➢ offset : 읽기-쓰기 포인터의 새 위치 결정</div> <div>➢ start_flag : 읽기-쓰기 포인터가 파일의 어느 지점을 시작으로 할 지 결정</div> <div>SEEK_SET offset 을 파일의 시작부터 계산</div> <div>SEEK_CUR offset 을 파일 포인터의 현재 위치부터 계산</div> <div>SEEK_END offset 을 파일의 끝부터 계산</div> </div> <div> <div>현재 위치</div> <div> <div>a</div> <div>b</div> <div>c</div> <div>d</div> <div>e</div> <div>f</div> <div>g</div> </div> <div>시스템 프로그래밍</div> </div> </div> </div></div>															

<p>(6) 파일 링크 관련 함수</p> <p>◆ 기능</p> <ul style="list-style-type: none"> ➢ 파일에 대한 링크와 관련된 동작 수행 <p>◆ 사용법</p> <pre>#include <unistd.h> int link(const char *oldpath, const char *newpath); int unlink(const char *pathname); int symlink(const char *oldpath, const char *newpath);</pre> <p>➢ link() : oldpath 파일에 대한 newpath 하드 링크를 생성</p> <p>➢ unlink() : 파일 시스템에서 pathname 파일 삭제. 파일이 심볼릭 링크이면 링크를 삭제</p> <p>➢ symlink() : oldpath 파일에 대한 newpath 심볼릭 링크를 생성</p>	<p>(7) fcntl 함수</p> <p>■ 기능</p> <p>◆ 개방된 파일에 대한 제어</p> <p>■ 사용법</p> <pre>#include <sys/types.h> #include <unistd.h> #include <fcntl.h> /* 주의: 마지막 인수의 타입은 생략부호 "..."에 의해 표시된 대로 가변적 */ int fcntl(int filedes, int cmd, ...);</pre> <p>➢ filedes : 제어할 파일 기술자</p> <p>➢ cmd : 제어에 대한 특정 기능</p> <ul style="list-style-type: none"> - F_GETFL: open에 의해 설정된 현재 파일의 상태 플래그 반환 - F_SETFL: 파일에 연관된 상태 플래그를 재지정 <p>➢ 세번째 인수부터는 인수 종에 따라 좌우</p>
<p>(8) remove/rename 함수</p> <p>■ 기능</p> <p>◆ 한 파일을 시스템으로부터 제거</p> <p>■ 사용법</p> <pre>#include <stdio.h> int remove(const char *pathname);</pre> <p>➢ pathname : 제거될 파일의 경로, 이 때 pathname 이 파일을 가리키면 unlink() 함수가 호출되고 디렉토리이면 rmdir() 함수가 호출.</p>	<p>■ 기능</p> <p>◆ 파일의 위치/이름 변경</p> <p>■ 사용법</p> <pre>#include <stdio.h> int rename(const char *oldpath, const char *newpath);</pre> <p>➢ oldpath : 바꾸거나 이동할 원래 파일 경로</p> <p>➢ newpath : 바꾸거나 이동한 후 만들어지는 파일 경로</p>

2) 고수준 파일 입출력

- 다른 말로 '표준 입출력 라이브러리' 라고 부름.
- 저수준 파일 입출력의 '바이트 단위'로만 입출력'의 단점을 커버함.
- C언어 표준 함수로 제공되며, 버퍼를 이용해 한꺼번에 읽기/쓰기를 수행함.
- 다양한 입출력 데이터 변환 기능도 이미 구현이 되어 있어서 데이터 형에 따라 편리하게 사용 가능.

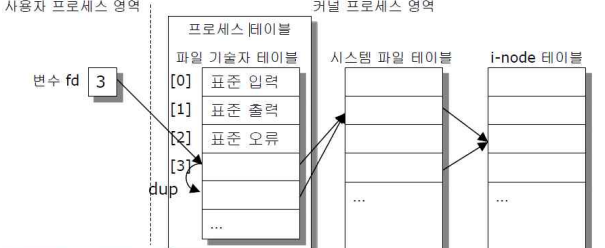
기능	함수 원형
파일 열기/닫기	FILE *fopen(const char *path, const char *mode); int fclose(FILE *stream);
파일 읽기/쓰기	int fgetc(FILE *stream); int fputc(int c, FILE *stream); char *fgets(char *s, int size, FILE *stream); int fputs(const char *s, FILE *stream); int fscanf(FILE *stream, const char *format, ...); int fprintf(FILE *stream, const char *format, ...);
파일 위치 재배치	int fseek(FILE *stream, long offset, int whence); long ftell(FILE *stream); void rewind(FILE *stream);

3) 저수준/고수준 파일 입출력 차이

	저수준 파일 입출력	고수준 파일 입출력
파일 지시자	int fd	FILE *fp;
특징	<ul style="list-style-type: none"> - 훨씬 빠르다 - 바이트 단위로 읽고 쓴다 - 특수 파일에 대한 접근이 가능하다. 	<ul style="list-style-type: none"> - 사용하기 쉽다. - 버퍼 단위로 읽고 쓴다. - 데이터의 입출력 동기화가 쉽다. - 여러 가지 형식을 지원한다.
주요 함수	open, close, read, write, dup, dup2, fcntl, lseek, fsync	fopen, fclose, fread, fwrite, fputs, fgets, fprintf, fscanf, freopen, fseek

2. 입출력 장치 파일

표 2. 표준 입출력, 파일 표현 함수

<p>(1) 표준 입력/ 표준 출력</p> <p>■ UNIX 시스템은 수행중인 한 프로그램에 대해 자동적으로 세 개의 파일을 개방</p> <ul style="list-style-type: none"> ◆ 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 <p>■ 표준 입력</p> <ul style="list-style-type: none"> ◆ 디플트로 키보드로부터 자료를 입력받음 ◆ # prog_name < infile <p>■ 표준 출력</p> <ul style="list-style-type: none"> ◆ 디플트로 단말기 화면으로 자료를 출력 ◆ # prog_name > outfile 	<p>(2) 표준 오류</p> <ul style="list-style-type: none"> ◆ 오류와 경고 메시지를 위한 특별한 출력 채널 ◆ # make > log.out 2> log.err <ul style="list-style-type: none"> ➢ Log.out : 표준 출력 결과, log.err : 표준 오류 결과 ◆ 시스템 호출 write 와 파일 기술자 2 를 사용 <pre>char msg[6] = "boob\n"; . . write (2, msg, 5);</pre>																
<p>(2) 파일의 표현 - 시스템 파일 관리</p> <ul style="list-style-type: none"> ◆ 파일 기술자 테이블, 시스템 파일 테이블, i-node 테이블을 통해 각 프로세스가 사용하는 파일 관리 <ul style="list-style-type: none"> ➢ 시스템 파일 테이블 - 시스템의 모든 파일에 대한 정보(상태, 현재 오픈셋 등) 수록 ➢ i-node 테이블 - 실제 저장된 파일의 정보 수록 	<p>(3) 파일 기술자 복사 함수</p> <p>■ 기능</p> <ul style="list-style-type: none"> ◆ 파일 기술자를 새로운 항목으로 복사 <ul style="list-style-type: none"> ➢ dup 함수는 소스 파일 기술자를 사용하지 않는 가장 작은 파일 기술자로 복사 ➢ dup2 함수는 소스 파일 기술자를 대상 파일 기술자로 복사 ◆ 복사되기 전후의 기술자들은 같은 시스템 파일 항목 공유 <ul style="list-style-type: none"> ➢ 락, 파일 위치 포인터, 플레그 공유 <p>■ 사용법</p> <pre>#include <unistd.h> int dup (int filedes); int dup2 (int filedes, int filedes2);</pre> <ul style="list-style-type: none"> ➢ filedes : 존재하는 소스 파일 기술자 항목 ➢ filedes2: 복사할 대상 파일 기술자 항목 ➢ 반환되는 값은 새로운 파일 기술자 번호, 복사가 실패하면 -1 반환 																
<p>(4) 파일 상태</p> <p>파일 상태</p> <p>■ LINUX stat 구조체</p> <ul style="list-style-type: none"> ◆ 파일 정보 저장 ◆ <bits/stat.h> 에 정의 <pre>struct stat { dev_t st_dev; /* device */ ino_t st_ino; /* inode */ mode_t st_mode; /* protection */ nlink_t st_nlink; /* number of hard links */ uid_t st_uid; /* user ID of owner */ gid_t st_gid; /* group ID of owner */ dev_t st_rdev; /* device type (if inode device) */ off_t st_size; /* total size, in bytes */ unsigned long st_blksize; /* blocksize for filesystem I/O */ unsigned long st_blocks; /* number of blocks allocated */ time_t st_atime; /* time of last access */ time_t st_mtime; /* time of last modification */ time_t st_ctime; /* time of last change */ };</pre> <p>■ 파일 유형을 테스트하는 매크로들</p> <ul style="list-style-type: none"> ◆ <sys/stat.h> 에 정의 <table border="1"> <thead> <tr> <th>매크로</th> <th>의미</th> </tr> </thead> <tbody> <tr> <td>S_ISREG()</td> <td>Regular file (일반적인 파일 형태)</td> </tr> <tr> <td>S_ISDIR()</td> <td>Directory file (다른 파일을 가지는 파일 형태)</td> </tr> <tr> <td>S_ISCHR()</td> <td>Character special file (문자 장치에 사용)</td> </tr> <tr> <td>S_ISBLK()</td> <td>Block special file (블록 장치에 사용)</td> </tr> <tr> <td>S_ISFIFO()</td> <td>FIFO pipe (프로세스간 IPC 에 사용)</td> </tr> <tr> <td>S_ISLNK()</td> <td>Symbolic link file (다른 파일에 대한 링크 파일 형태)</td> </tr> <tr> <td>S_ISSOCK()</td> <td>socket file (네트워크 통신에 사용)</td> </tr> </tbody> </table>	매크로	의미	S_ISREG()	Regular file (일반적인 파일 형태)	S_ISDIR()	Directory file (다른 파일을 가지는 파일 형태)	S_ISCHR()	Character special file (문자 장치에 사용)	S_ISBLK()	Block special file (블록 장치에 사용)	S_ISFIFO()	FIFO pipe (프로세스간 IPC 에 사용)	S_ISLNK()	Symbolic link file (다른 파일에 대한 링크 파일 형태)	S_ISSOCK()	socket file (네트워크 통신에 사용)	<p>■ 기능</p> <ul style="list-style-type: none"> ◆ 파일 유형 등의 상태 정보 처리 <p>■ 사용법</p> <pre>#include <sys/types.h> #include <sys/stat.h> #include <unistd.h> int stat(const char *filename, struct stat *buf); int fstat(int filedes, struct stat *buf); int lstat(const char *filename, struct stat *buf);</pre> <ul style="list-style-type: none"> ➢ stat() : filename 이 주어지면 해당 파일에 대한 정보 획득 ➢ fstat() : 파일 기술자를 인자로 파일 정보 획득 ➢ lstat() : stat() 와 비슷하나 해당 파일이 심볼릭 링크인 경우 링크의 정보 획득 ➢ buf : 파일 정보를 정의한 stat 구조체 포인터
매크로	의미																
S_ISREG()	Regular file (일반적인 파일 형태)																
S_ISDIR()	Directory file (다른 파일을 가지는 파일 형태)																
S_ISCHR()	Character special file (문자 장치에 사용)																
S_ISBLK()	Block special file (블록 장치에 사용)																
S_ISFIFO()	FIFO pipe (프로세스간 IPC 에 사용)																
S_ISLNK()	Symbolic link file (다른 파일에 대한 링크 파일 형태)																
S_ISSOCK()	socket file (네트워크 통신에 사용)																

3. 프로세스 및 시그널

1) 프로세스



프로세스 영역

- 쉽게 말해서, 수행중인 프로그램임.
- 쉘은 하나의 명령을 수행하기 위해 어떤 프로그램을 시작할 때 마다 새로운 프로세스를 생성함.
- UNIX 프로세스 환경은 디렉터리 tree처럼 계층적인 구조임
 - 가장 최초의 프로세스는 init 이며, 모든 시스템과 사용자 프로세스의 조상.

그림 3. 프로세스 영역

이름	의미
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

그림 4. 프로세스 시스템 호출들

표 3. 프로세스 시스템 호출

<p>(1) fork 시스템 호출</p> <p>◆기능</p> <ul style="list-style-type: none"> ➢ 기본 프로세스 생성 함수 ➢ 성공적으로 수행되면 호출 프로세스(부모 프로세스)와 똑같은 새로운 프로세스(자식 프로세스) 생성 <p>◆사용법</p> <pre>#include <sys/types.h> #include <unistd.h> pid_t fork(void);</pre> <p>◆반환 값(return value)</p> <ul style="list-style-type: none"> ➢ 정상 실행 <ul style="list-style-type: none"> - 실행(부모) 프로세스는 : 생성(자식) 프로세스의 프로세스 ID 반환 - 생성(자식) 프로세스는 : 0을 반환 ➢ 이상 실행 : 음수 값을 반환 <p>◆프로세스 식별번호(Identifier)</p>	<p>■ 프로세스 생성 함수 사용 예</p> <p>◆프로세스 생성 예제 프로그램</p> <p>➢ fork 호출 후 두 프로세스는 바로 다음 문장부터 수행 계속</p> <pre>printf("Call...\n"); pid = fork(); if (pid == 0) printf("I'm...\n"); else if (pid > 0) printf("I'm...\n");</pre> <p>호출 전</p> <p>parent</p> <p>FORK</p> <p>호출 후</p> <p>parent</p> <p>child</p> <p>PC</p> <p>PC</p>
<p>(2) exit 시스템 호출</p> <p>◆기능</p> <ul style="list-style-type: none"> ➢ 프로세스 종료 <p>◆사용법</p> <pre>#include <stdlib.h> void exit (int status);</pre> <ul style="list-style-type: none"> ➢ status : 종료 상태를 나타냄 ➢ 반환값: 정상적 종료: 0, 그렇지 않을 경우: 0이 아닌 값 <p>◆모든 개방된 파일 기술자들을 닫는다.</p>	<p>(3) atexit 함수</p> <p>◆기능</p> <ul style="list-style-type: none"> ➢ 실행 프로세스 종료 시 호출되는 루틴 설정 <p>◆사용법</p> <pre>#include <stdlib.h> void atexit (void (*func) (void));</pre> <p>➢ 프로그래머가 종료 루틴을 정의</p>

시그널의 종류

이름	의 미	이름	의 미
SIGHUP	hangup. 단말기 연결이 끊어졌을 때 그 단말기에 연결된 모든 프로세스에 보냄. 이것을 받으면 종료.	SIGPIPE	write on a pipe with no-one to read it. 종료한 파이프나 소켓에 쓸 때 보냄. (파이프는 또다른 프로세스간 통신 방식)
SIGINT	interrupt. 사용자가 인터럽트키를 칠 때 단말기와 연결된 모든 프로세스에 보냄. 수행중인 프로그램을 중지시키는 일반적인 방법	SIGALRM	alarm clock. 타이머가 만료되었을 때 프로세스에 보냄. alarm 함수로 이루어짐.
SIGQUIT	quit. SIGINT와 마찬가지로 사용자가 단말기에서 종료(quit)키를 칠 때 보냄. 종료키의 일반적인 값은 ASCII FS 또는 CTRL-\	SIGTERM	software termination. 프로세스를 종료시키기 위해 사용자에게 의해 사용.
SIGILL	illegal instruction. 비정상적인 명령 수행 시 보냄	SIGSTKFLT	stack fault. 스택 오류일 때 보냄.
SIGTRAP	trace trap. ptrace 시스템과 함께 gdb 또는 adb 등의 디버거에 의해 사용되는 특별한 시그널.	SIGCHLD	child status has changed. 자식 프로세스가 종료하거나 중단될 때 부모 프로세스에 보냄. 이 시그널을 받으면 무시.
SIGABRT	abort. 현재 프로세스가 abort 함수를 호출할 때 보냄. 비정상적인 종료(abnormal termination)가 됨. 이것을 받으면 코어 덤프하고 종료.	SIGCONT	continue. 이 시그널을 받으면 중단된 프로세스의 경우 계속 실행하고, 실행중일 때는 무시.
SIGFPE	floating-point exception. 오버플로우나 언더플로우 같은 부동 소숫점 오류가 발생했을 때 보냄	SIGSTOP	stop, unblockable. 프로세스를 중단시키기 위해 보냄.
SIGKILL	kill. 프로세스로부터 다른 프로세스를 종료시키기 위해 보냄.	SIGTSTP	keyboard stop. 사용자가 일시 중지 키 (Ctrl-Z)을 칠 때 보냄. SIGSTOP와 비슷
SIGUSR1	SIGTERM과 마찬가지로 이것들은 커널에 의해 사용되는 것이 아니라 사용자가 원하는 목적을 위하여 사용 가능.	SIGTTIN	백그라운드 프로세스가 단말기로부터 읽기를 시도할 때 보냄.
SIGSEGV	segmentation violation. 프로세스가 유효하지 않은 메모리 주소에 접근할 때 보냄. 이것을 받으면 비정상적 종료.	SIGTTOU	백그라운드 프로세스가 단말기로 쓰기를 시도할 때 보냄.
		SIGURG	프로세스에게 네트워크 연결에 긴급하거나 대역을 벗어난 자료가 수신되었음을 알림.
		SIGSYS	bad arguments to a system call. 이것은 잘못된 시스템 호출을 보냈을 때 보냄.

그림 5. 시그널 종류

시그널 집합

■ 시그널 집합 - 시그널을 원소로 하는 집합

■ 관련 시스템 호출

◆ 기능

- 시그널 집합 관련 기능 수행
- 각각의 시그널을 따로 처리하는 것보다 편리

◆ 사용법

```
#include <signal.h>
/* 초기화 */
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
/* 조작 */
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
```

- sigemptyset - 비어있는 시그널 집합 생성
- sigfillset - 모든 시그널을 포함하는 집합 생성
- sigaddset - 특정 시그널 번호를 집합에 추가
- sigdelset - 특정 시그널 번호를 집합에서 제거

그림 6. 시그널 집합

표 4. 시그널 처리 함수

(1) 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

◆ sigaction 구조체

```
struct sigaction {
    union {
        __sig_handler_t sa_handler; /* 함수, SIG_DFL 또는 SIG_IGN */
        void (*sa_sigaction)(int, siginfo_t *, void *); /* 시그널 핸들러 포인터 */
    } __sigaction_handler;
    sigset_t sa_mask; /* sa_handler에서 봉쇄할 시그널 */
    int sa_flags; /* 시그널 동작 변경자 */
};
```

- sa_handler: signal number
- Sa_sigaction: sigaction 구조체
- Sa_mask: 이전 설정값
- Sa_flags:
 - SA_NOCLDSTOP
 - SA_RESETHAND
 - SA_RESTART
 - SA_NODEFER

SA_SIGINFO

표 5. 시그널 전송 함수

<p>(2) kill 시스템 호출</p> <p>◆기능</p> <ul style="list-style-type: none"> ➢ 다른 프로세스에게 특정 시그널 전송 <p>◆사용법</p> <pre>#include <sys/types.h> #include <signal.h> int kill(pid_t pid, int sig);</pre> <p>➢ sig: 전송할 시그널</p> <p>➢ pid: 시그널 sig 를 받을 프로세스 ID</p> <ul style="list-style-type: none"> - 양수: 해당 프로세스 식별번호를 가진 프로세스 - 0: kill 을 호출한 프로세스와 그룹에 속하는 모든 프로세스에게 전송 - -1: 1번 프로세스를 제외한 모든 프로세스에게 전송, 프로세스 유효사용자 ID가 슈퍼유저가 아니면, 유효사용자 ID가 같은 모든 프로세스에게 전송 - 음수: 프로세스 그룹 식별번호가 pid 의 절대값과 같은 모든 프로세스에게 전송 	<p>(3) pause/raise 시스템 호출</p> <p>◆기능</p> <ul style="list-style-type: none"> ➢ 시그널이 도착할 때까지 프로세스 실행을 중단하고 대기 <p>◆사용법</p> <pre>#include <unistd.h> int pause(void);</pre> <p>◆기능</p> <ul style="list-style-type: none"> ➢ 현재 프로세스에게 시그널 전송 <p>◆사용법</p> <pre>#include <signal.h> int raise(int sig);</pre> <p>➢ sig: 전송할 시그널</p>
<p>(4) raise 시스템 호출</p> <p>◆기능</p> <ul style="list-style-type: none"> ➢ 현재 프로세스에게 시그널 전송 <p>◆사용법</p> <pre>#include <signal.h> int raise(int sig);</pre> <p>➢ sig: 전송할 시그널</p>	<p>(5) alarm 시스템 호출</p> <p>◆기능</p> <ul style="list-style-type: none"> ➢ 지정된 초 후에 현재 프로세스에게 SIGALRM 시그널을 전달 <p>◆사용법</p> <pre>#include <unistd.h> unsigned int alarm(unsigned int secs);</pre> <p>➢ secs: SIGALRM 시그널을 전송하기 전까지 기다리는 시간(초)</p> <p>◆실행한 후 즉시 복귀하여 정상적인 수행 계속</p> <p>◆alarm 취소</p> <ul style="list-style-type: none"> ➢ alarm(0) 호출

표 6. 시그널 차단/상태 복귀 함수

<p>(1) sigprocmask 시스템 호출</p> <p>◆기능</p> <ul style="list-style-type: none"> ➢ 특정 시그널을 차단하도록 허용 <p>◆사용법</p> <pre>#include <signal.h> int sigprocmask(int how, const sigset_t *set, sigset_t *oset);</pre> <p>➢ how: sigprocmask 가 어떻게 동작할 것인지를 결정</p> <ul style="list-style-type: none"> - SIG_BLOCK: 차단하는 시그널은 현재 설정된 것들과 set 에 포함된 것 - SIG_UNBLOCK: set 에 포함된 시그널들이 차단되지 않도록 설정 - SIG_SETMASK: set 에 포함된 시그널들이 차단되도록 설정 <p>➢ set: 선택할 시그널들의 집합</p> <p>➢ oset: 이전에 설정된 시그널들의 마스크 값</p>	<p>(2) sigsetjmp, siglongjmp/저장 및 복귀 시스템</p> <p>◆기능</p> <ul style="list-style-type: none"> ➢ sigsetjmp 함수는 현재 프로그램 상태를 저장 ➢ siglongjmp 함수는 저장된 위치로 복귀 <p>◆사용법</p> <pre>#include <setjmp.h> int sigsetjmp(sigjmp_buf env, int savesigs); void siglongjmp(sigjmp_buf env, int val);</pre> <p>➢ env: 프로그램 위치를 저장할 sigsetjmp_buf 형의 객체</p> <p>➢ val: 복귀한 뒤 sigsetjmp 에 의해 반환할 값</p> <p>◆sigsetjmp 함수는 현재의 스택과 시그널 마스크 저장</p> <p>➢ siglongjmp 가 호출되어 저장한 위치로 복귀할 때 이들을 복구</p> <p>◆sigsetjmp 함수의 반환값</p> <ul style="list-style-type: none"> ➢ 직접 반환되면 0 을 반환 ➢ Siglongjmp 에 의해 반환되면 0 이 아닌 값(val) 반환
--	--

3) 데몬 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

(1) 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

그림 7. damon 함수 사용

(2) Syslog 관련 함수

◆ 기본 형

```
#include <syslog.h>

void openlog(const char *ident, int option, int facility);
void syslog(int level, const char *format, ...);
void closelog(void);
```

- 옵션: LOG_CONS – 로그 메시지를 syslogd로 보낼 수 없으면 대신 콘솔 표시
- 기능(facility) 로그 레벨

기능	설 명	레벨	설 명
LOG_AUTH	보안	LOG_EMERG	응급, 시스템 불능
LOG_AUTHPRIV	보안, 제한된 허가권 파일	LOG_ALERT	긴급, 즉각 고쳐야 할 상태
LOG_CRON	CRON과 AT	LOG_CRIT	치명적인 상태
LOG_DAEMON	시스템 데몬	LOG_ERR	오류 상태
LOG_KERN	커널 생성 메시지	LOG_WARNING	경고 상태
LOG_LOCAL0~7	사용자 정의	LOG_NOTICE	일반, 중요한 상태
LOG_SYSLOG	Syslogd 데몬	LOG_INFO	정보 메시지
LOG_USER	사용자 응용	LOG_DEBUG	디버그 메시지

그림 8. Syslog 관련 함수

4. 리눅스의 프로세스 모델과 시그널/쓰레드와의 관계

1) 리눅스 프로세스의 시그널

□ 시그널을 수신한 프로세스의 반응

1. 시그널에 대해 기본적인 방법으로 대응한다. 대부분의 시그널에 대해서 프로세스는 종료하게 된다.
2. 시그널을 무시한다. 단, SIGKILL과 SIGSTOP은 무시될 수 없다.
3. 프로그래머가 지정한 함수(핸들러)를 호출한다.

□ 시그널 핸들러 (handler)

- 프로세스가 특정 시그널을 포착했을 때 수행해야 할 별도의 함수
 - 프로세스는 시그널을 포착하면 현재 작업을 일시 중단하고 시그널 핸들러를 실행
 - 시그널 핸들러의 실행이 끝나면 중단된 작업을 재개

□ 시그널 종류

- "/usr/include/asm/signal.h"

2) 프로세스 시그널과 쓰레드 시그널

- 프로세스는 각각 logical control flow를 가지고 있으며, 서로 공유를 하지 않는다.
- 쓰레드는 서로 code, data를 공유한다.
- 즉 프로세스는 서로 시그널을 보내도 각자 알아서 수행하지만, 쓰레드는 서로 데이터를 공유하기 때문에 어떤 프로세스에서 자기가 생성한 쓰레드 하나에게 시그널을 보낼 경우 그 프로세스가 생성한 모든 쓰레드에게 시그널이 전달된다.
- 따라서 프로세스는 시그널을 쓰레드로 전달할 때, 특정 쓰레드에게만 시그널을 받도록 조치를 하는데, 이것을 '**쓰레드 마스크**'라고 한다.
- 프로세스가 보낸 특정 시그널에 대해 '**쓰레드 마스크**'를 씌우는 것으로, 마스킹 된 시그널은 다른 쓰레드에게 공유되지 않으며 이 시그널을 받기 원하는 쓰레드는 시그널에 대한 마스크를 해제하고 시그널을 송신한다.

5. 프로세스 간 통신

1) 파일을 통한 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

■ 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

◆ Lock

- **struct flock** : 레코드 잠금에 대한 내용 기술

```
struct flock {
    short l_type; /* 잠금 유형 */
    short l_whence; /* 잠금 영역을 정하는 기준 */
    off_t l_start; /* 잠금 영역의 시작 위치 */
    off_t l_len; /* 잠금 영역의 바이트 단위 길이 */
    pid_t l_pid; /* 명령어 F_GETLK 일 때 잠금 설정하는 프로세스ID */
};
```

➢ l_type : 잠금 유형

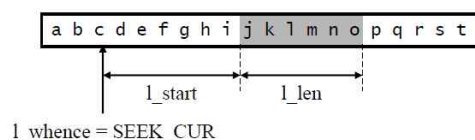
- **F_RDLCK** : 읽기 잠금 지정
- **F_WRLCK** : 쓰기 잠금 지정
- **F_UNLCK** : 잠금 해제

➢ l_whence, l_start, l_len

- 잠금 위치 지정

➢ l_pid

- 잠금 프로세스 ID



(3) 표준 파이프 입출력 함수

■ popen 과 pclose 함수

◆ 기능

- **popen** : pipe와 fork를 이용하여, 쉘 명령어의 입력 혹은 출력을 파이프와 연결한 다음 쉘 명령어를 실행
- **pclose** : 사용하고 난 파이프와 명령 프로세스를 닫음

◆ 사용법

```
#include <stdio.h>

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

- **command** : 수행할 명령어를 나타내는 문자열의 포인터
- **type** : pipe의 읽기('r') 또는 쓰기('w') 방향

◆ 반환값

- **popen** 은 성공적인 호출에 대하여 파이프와 연관된 파일 기술자 반환
- **pclose** 는 성공적인 호출에 대하여 0을 반환
- 오류시에는 -1 이 반환되고, **errno** 가 적절히 설정

(4) FIFO를 이용한 파이프, 명명파이프

■ 파이프의 단점

- ◆ 프로그램 내에서는 부모와 자식 프로세스 간 데이터 전달 가능하지만, 프로세스가 종료되면 파이프도 삭제
- ◆ 독립된 두 프로세스 사이에 파이프를 통해 데이터 전달 불가능

■ → FIFO 또는 명명 파이프(named pipe)

- ◆ 특수한 형태의 파일
- ◆ 입출력 처리 방식은 선입선출(first-in first-out) 방식
- ◆ 파일 **inode**를 가지므로 영구적이고 임의의 프로세스가 접근 가능

■ mkfifo 시스템 호출

◆ 기능

- **FIFO** 파일 생성

◆ 사용법

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

- **pathname** : 생성할 FIFO 파일의 경로 지정
- **mode** : 이 파일의 허가권을 설정

3) 고급 프로세스 간 통신

■ 고급 IPC 개요

◆ UNIX 와 LINUX 에서 지원하는 고급 IPC 자원

1. 메시지 큐를 통하여 프로세스들 간에 메시지 전달
2. 세마포어를 통하여 프로세스들 간의 동기화
3. 프로세스 간 메모리 영역 공유

◆ 자원 요청 시 동적으로 데이터 구조 생성

- 명시적으로 삭제하지 않으면 시스템이 종료될 때까지 메모리에 존재
- 독립적인 프로세스 간 통신에 사용 가능

■ 종류

◆ System V IPC (또는 XSI IPC)

- IPC 자원(메시지큐, 세마포어, 공유메모리)을 파일이 아닌 키, ID 등으로 관리

◆ POSIX IPC

- IPC 자원을 메모리 상의 가상 파일시스템을 통해 접근
- System V IPC보다 더 새롭고 직관적이고 사용하기 쉬움
- 실시간(-lrt) 또는 쓰레드(-lpthread) 라이브러리와 함께 링크하여 사용

■ System V IPC

◆ 관련 함수, 파일, 라이브러리

구분	메시지 큐	세마포어	공유메모리
헤더파일	sys/msg.h	sys/sem.h	sys/shm.h
생성, 개방 함수	msgget	semget	shmget
제어 함수	msgctl	semctl	shmctl
IPC 동작 함수	msgsnd msgrcv	semop	shmat shmdt

◆ IPC 자원의 구별

➢ Key (32 bit)

- 파일 경로와 비슷하며 프로그래머가 자유롭게 선택하거나 **ftok**로 생성

➢ IPC 설비 식별자(32 bit)

- 커널에 의해 자원에 부여되며, 시스템 내에서 고유한 값

➢ ftok 함수 - 프로젝트 ID로부터 Key 값으로 변환

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

key_t ftok(const char *pathname, int proj_id);
```

고급 IPC 자원을 위한 쉘에서 제공하는 명령

◆ ipcs - 전체 시스템에서 사용하고 있는 고급 IPC 자원 정보 출력

```
[linux@seps ipcs]$ ipcs

----- Shared Memory Segments -----
key      shmid  owner  perms  bytes  nattch status
0x00000049 1146880 linux  666    1024    0

----- Semaphore Arrays -----
key      semid  owner  perms  nsems  status
0x00000049 65536 linux  666    1

----- Message Queues -----
key      msqid  owner  perms  used-bytes  messages
0x00000049 0 linux  666    0 0
```

◆ ipcrm - 시스템으로부터 특정한 고급 IPC 자원을 제거

일반형식	ipcrm [msg sem shm] id
주요옵션	msg : 메시지 큐를 삭제하는 경우 sem : 세마포어를 삭제하는 경우 shm : 공유 메모리를 삭제하는 경우

(1) 고급 프로세스 간 통신 - Sys V 메시지 큐

- 프로세스간에 문자나 바이트 열로 이루어진 메시지를 전달하기 위한 일종의 버퍼와 같은 큐
- 메시지 큐와 파이프의 차이점
 - 파이프와 달리 메시지의 크기가 제한적임.
 - select 등을 사용할 수 없음
 - 우선순위 등에 따라 메시지 관리 기능



표 8. Sys V 메시지큐

<p>(1) msgget 시스템 호출</p> <p>◆ 기능</p> <ul style="list-style-type: none"> 메시지 큐 생성 <p>◆ 사용법</p> <pre>#include <sys/msg.h> int msgget(key_t key, int permflags);</pre> <ul style="list-style-type: none"> key: 생성할 메시지 큐에 대한 고유 번호 <ul style="list-style-type: none"> 이 값을 통해 프로세스들이 메시지 큐를 공유 가능 IPC_PRIVATE을 사용 - 다른 프로세스가 생성한 key 값과 중복되지 않는 유일한 메시지 큐 생성 permflags: 생성 시 메시지 큐 사용 허가권 지정 <ul style="list-style-type: none"> 메시지를 보내기 위해서는 쓰기 허가권 설정 메시지를 받기 위해서는 읽기 허가권 설정 <p>◆ 반환값</p> <ul style="list-style-type: none"> 성공적인 호출에 대하여 메시지 큐 식별자 반환 오류시에는 -1 이 반환되고, errno 가 적절히 설정 	<p>(2) msgsnd 시스템 호출</p> <p>◆ 기능</p> <ul style="list-style-type: none"> 메시지 큐로 메시지 전송 <p>◆ 사용법</p> <pre>#include <sys/msg.h> int msgsnd(int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg);</pre> <ul style="list-style-type: none"> msqid 가 가리키는 메시지 큐에 메시지를 추가 msgp: 전송할 버퍼의 주소 <pre>struct msgbuf { long mtype; /* 메시지 유형, 0 보다 커야 한다 */ char mtext[xxx]; /* 메시지 데이터 */ };</pre> <ul style="list-style-type: none"> msgflg: 0 이거나 IPC_NOWAIT <ul style="list-style-type: none"> IPC_NOWAIT 가 설정되어 있으면, 즉시 반환 그렇지 않으면, 메시지 큐가 메시지를 저장할 수 있을 때까지 대기 msgsz: 전송할 메시지의 최대 바이트 크기를 지정
<p>(3) msgrcv 시스템 호출</p> <p>◆ 기능</p> <ul style="list-style-type: none"> 메시지 큐로부터 메시지 수신 <p>◆ 사용법</p> <pre>#include <sys/msg.h> ssize_t msgrcv(int msqid, struct msgbuf *msgp, size_t msgsz, long msgtyp, int msgflg);</pre> <ul style="list-style-type: none"> msqid 가 가리키는 메시지 큐로부터 메시지를 읽음 msgp: 수신할 버퍼의 주소 msgsz: msgp 가 가리키는 구조체에 저장할 수 있는 최대 바이트 크기를 지정 msgtyp: 수신할 메시지의 유형 결정 <ol style="list-style-type: none"> 0: 메시지 큐의 제일 앞에 있는 메시지를 읽음 양수: 큐에서 같은 유형을 가진 첫 메시지를 읽음. 다만, msgflg 인자에 MSG_EXCEPT 가 지정되어 있으면, msgtyp과 같지 않은 유형의 첫 메시지를 읽음 음수: 큐에서 msgtyp 의 절대값보다 작거나 같은 유형의 첫 메시지를 읽음 msgflg: 0 이거나 IPC_NOWAIT, IPC_NOERROR 또는 MSG_EXCEPT <ul style="list-style-type: none"> IPC_NOWAIT 가 설정되어 있으면, 즉시 반환 그렇지 않으면, 메시지 큐가 메시지를 저장할 수 있을 때까지 대기 	<p>(4) msgctl 시스템 호출</p> <p>◆ 기능</p> <ul style="list-style-type: none"> 메시지 큐 제어 (메시지 큐의 정보를 얻거나, 변경하거나, 삭제) <p>◆ 사용법</p> <pre>#include <sys/msg.h> int msgctl(int msqid, int cmd, struct msqid_ds *buf);</pre> <ul style="list-style-type: none"> msqid가 가리키는 메시지 큐에 cmd 가 지정한 동작을 수행 cmd: 수행할 명령 <ul style="list-style-type: none"> IPC_STAT: msqid 와 관련된 메시지 큐의 정보를 buf 의 위치에 저장 IPC_SET: 메시지 큐의 정보를 buf 가 가리키는 값으로 변경, 변경 가능한 값은 msg_perm.uid, msg_perm.gid, msg_perm.mode, msg_qbytes IPC_RMID: 메시지 큐와 관련된 데이터 구조를 즉시 삭제 buf: 메시지 큐 데이터 구조의 주소 <pre>struct ipc_perm msg_perm; time_t msg_stime; /* 최근 msgsnd 시간 */ time_t msg_rtime; /* 최근 msgrcv 시간 */ time_t msg_ctime; /* 최근 변경 시간 */ unsigned short msg_qnum; /* 큐의 메시지 수 */ unsigned short msg_qbytes; /* 큐의 최대 바이트 수 */ pid_t msg_lspid; /* 최근 msgsnd 의 pid */ pid_t msg_lrpid; /* 최근 수신 pid */</pre>

(2) 고급 프로세스 간 통신 - Sys V 세마포어

- 프로세스 간 효율적인 동기화 방식
- 열쇠와 비슷하여, 어떤 프로세스가 세마포어를 획득하면 공유 자원에 접근하거나 동작을 수행하도록 허용하고, 그렇지 않으면 세마포어가 해제되기를 기다리며 대기함.
- 기능
 - 프로세스들 간의 공유 자원 접근 제공 (상호 배제 요구 조건 만족)
 - 파이프나 메시지 큐는 대규모 자료 전송에는 적합하지만, 자원낭비가 크므로 동기화 수단에는 적합하지 않다.

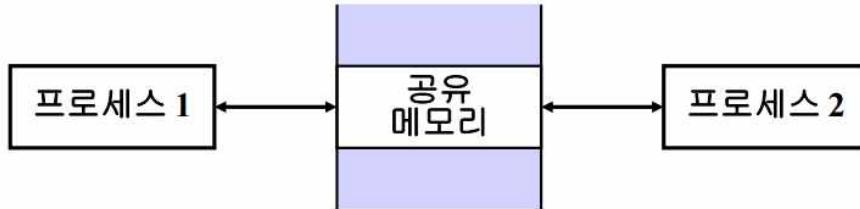


표 9. Sys V 세마포어

<p>(1) 세마포어의 구현 / 예시</p> <p>◆ 두 가지 연산을 가지는 어떤 정수값</p> <p>◆ 데이터 구조</p> <pre>unsigned short semval; /* 세마포어 값 */ unsigned short semzcnt; /* 0 이 되기를 기다리는 프로세스의 수 */ unsigned short semncnt; /* 증가하기를 기다리는 프로세스의 수 */ pid_t sempid; /* 최근 연산을 수행한 프로세스 ID */</pre> <p>◆ 세마포어 사용 예</p> <ul style="list-style-type: none"> ➢ 프로세스 간 동기화 ➢ 프로세스 간 공유 자원 접근 <p>p (S); 공유 자원을 사용하여 필요한 일을 수행한다;</p> <p>v (S);</p>	<p>(2) 세마포어의 연산</p> <hr/> <p>wait(S) 또는 P(S) : 세마포어의 값이 영이 아니면, 세마포어의 값을 일만큼 감소시킨다; 그렇지 않으면, 세마포어의 값이 영이 아닐 때까지 기다린다. 그런 다음 세마포어의 값을 일만큼 감소시킨다;</p> <p>signal(S) 또는 V(S) : 세마포어의 값을 일만큼 증가시킨다; 대기하고 있는 프로세스가 있으면, 대기 리스트로부터 한 프로세스를 깨운다;</p>
<p>(3) semget 시스템 호출</p> <p>◆ 기능</p> <ul style="list-style-type: none"> ➢ 세마포어 집합 생성 <p>◆ 사용법</p> <pre>#include <sys/sem.h> int semget(key_t key, int nsems, int permflags);</pre> <p>➢ key : 생성할 세마포어 집합에 대한 고유 번호</p> <ul style="list-style-type: none"> - 이 값을 통해 프로세스들이 세마포어 공유 가능 - IPC_PRIVATE을 사용 - 다른 프로세스에서 생성한 key 값과 중복되지 않는 유일한 세마포어 집합 생성 <p>➢ nsems : 세마포어 집합에서 생성할 세마포어 개수</p> <p>➢ permflags : 생성 시 세마포어 사용 허가권 지정</p> <p>◆ 반환값</p> <ul style="list-style-type: none"> ➢ 성공적인 호출에 대하여 세마포어 집합 식별자 반환 ➢ 오류시에는 -1 이 반환되고, errno 가 적절히 설정 	
<p>(4) semop 시스템 호출</p> <p>◆ 기능</p> <ul style="list-style-type: none"> ➢ 세마포어 연산 <p>◆ 사용법</p> <pre>#include <sys/sem.h> int semop(int semid, struct sembuf *sops, size_t nsops);</pre> <p>➢ semid 가 가리키는 세마포어 집합 중에서 sops 가 가리키는 구조체 배열에서 nsops 만큼 지정된 것들에 대하여 세마포어 연산을 수행</p> <p>➢ sops : sembuf 구조체의 주소</p> <pre>struct sembuf { unsigned short sem_num; /* 세마포어 번호 */ short sem_op; /* 세마포어 연산 */ short sem_flg; /* 연산 플래그 */ };</pre> <p>➢ sem_flg : IPC_NOWAIT 또는 SEM_UNDO</p> <ul style="list-style-type: none"> - IPC_NOWAIT 가 설정되어 있으면, 즉시 반환 - SEM_UNDO : 프로세스가 종료(exit) 할 때 이 연산이 수행되지 않음 	<p>sem_op : 수행하는 연산</p> <ul style="list-style-type: none"> ➢ 양수 : “세마포어 값을 증가시키는 연산” 수행. <ul style="list-style-type: none"> - 이 값을 semval 에 더함. 이 연산은 항상 진행되며, 세마포어 값이 증가하기를 기다리는 프로세스들을 깨움. ➢ 0 : “0을 기다리는 연산” 수행. <ul style="list-style-type: none"> - semval 이 0이면 그대로 진행 - 그렇지 않고 IPC_NOWAIT 플래그가 설정되어 있으면 즉시 오류값 반환 - 그렇지 않으면, semzcnt 가 하나 증가하고, semval 이 0이 될 때까지 대기 ➢ 음수 : “세마포어 값을 감소시키는 연산” 수행. <ul style="list-style-type: none"> - semval 값이 sem_op 의 절대값보다 크거나 같으면, 연산은 즉시 진행되어 semval 은 sem_op 의 절대값만큼 감소 - semval 값이 sem_op 의 절대값보다 작고 IPC_NOWAIT 가 설정되어 있으면, 즉시 반환되고 오류값은 EAGAIN - 그렇지 않으면 semncnt 값이 하나 증가하고, 세마포어 값이 커질 때까지 프로세스가 대기
<p>(5) semctl 시스템 호출</p> <p>◆ 기능</p> <ul style="list-style-type: none"> ➢ 세마포어 집합 제어 (세마포어의 정보를 얻거나, 변경하거나, 삭제) <p>◆ 사용법</p> <pre>#include <sys/sem.h> int semctl(int semid, int sem_num, int cmd, union semun arg);</pre> <p>➢ semid 가 가리키는 세마포어 집합 중 sem_num 이 지정하는 세마포어에 대하여 cmd 가 지정한 동작을 수행</p> <p>➢ arg : 다음 유니온으로 지정</p> <pre>union semun { int val; /* SETVAL 을 위한 값 */ struct semid_ds *buf; /* IPC_STAT, IPC_SET 을 위한 버퍼 */ unsigned short int *array; /* GETALL, SETALL 을 위한 배열 */ struct seminfo *__buf; /* IPC_INFO 를 위한 버퍼 */ };</pre> <p>➢ semid_ds 구조체 : 세마포어 집합의 정보를 가짐. 다음 요소 포함</p> <pre>struct ipc_perm sem_perm; /* 최근 연산 시간 */ time_t sem_otime; /* 최근 변경 시간 */ time_t sem_ctime; /* 최근 변경 시간 */ ushort sem_nsems; /* 세마포어의 수 */</pre>	<p>➢ cmd : 수행할 명령</p> <ul style="list-style-type: none"> - IPC_STAT : semid 와 관련된 세마포어 집합 정보를 arg.buf 의 위치에 저장 - IPC_SET : 세마포어 집합의 정보를 arg.buf 가 가리키는 값으로 변경, 변경가능한 값은 sem_perm.uid, sem_perm.gid, sem_perm.mode - IPC_RMID : 세마포어 집합과 관련된 데이터 구조를 즉시 삭제 - GETVAL : 세마포어 집합에서 sem_num 번째 세마포어 값 반환 - SETVAL : 세마포어 집합에서 sem_num 번째 세마포어 값을 arg.val 로 설정 - GETPID : 세마포어 집합에서 sem_num 번째 세마포어의 sempid 값 반환 - GETNCNT : 세마포어 집합에서 sem_num 번째 세마포어의 semncnt 값 반환 - GETZCNT : 세마포어 집합에서 sem_num 번째 세마포어의 semzcnt 값 반환 - GETALL : 세마포어 집합의 모든 세마포어의 값을 arg.array 에 저장 - SETALL : 세마포어 집합의 모든 세마포어의 값을 arg.array 으로 설정

(3) 고급 프로세스 간 통신 - Sys V 공유 메모리

- 둘 이상의 프로세스가 특정 메모리 영역을 공유하여 자료에 접근.
- 앞에서 말한 세 가지 고급 IPC 기법 중에서 가장 유용함.
- 프로세스들 간의 자료를 공유함.



- IPC 공유 메모리 영역에 대한 자료구조.
- 프로세스 접근을 위해 프로세스 주소 공간에 공유 메모리 영역 추가.
- 사용이 끝나면 주소 공간에서 공유 메모리를 제거함.

표 10. Sys V 공유 메모리

<p>(1) shmget 시스템 호출</p> <p>◆ 기능</p> <ul style="list-style-type: none"> ➢ 공유 메모리 생성 <p>◆ 사용법</p> <pre>#include <sys/shm.h> int shmget(key_t key, size_t size, int permflags);</pre> <p>➢ key : 생성할 공유 메모리 영역에 대한 고유 번호</p> <ul style="list-style-type: none"> - 이 값을 통해 프로세스들이 공유 메모리 영역을 공유 가능 - IPC_PRIVATE를 사용 - 다른 프로세스가 생성한 key 값과 중복되지 않는 유일한 공유 메모리 영역 생성 <p>➢ size : 생성하는 공유 메모리 영역의 최소 바이트 크기를 지정</p> <p>➢ permflags : 생성 시 공유 메모리 영역 사용 허가권 지정</p> <ul style="list-style-type: none"> - 공유 메모리 영역에 쓰기 위해서는 쓰기 허가권 설정 - 공유 메모리 영역을 읽기 위해서는 읽기 허가권 설정 <p>◆ 반환값</p> <ul style="list-style-type: none"> ➢ 성공적인 호출에 대하여 공유 메모리 영역 식별자 반환 ➢ 오류시에는 -1 이 반환되고, errno 가 적절히 설정 	<p>(2) shmat, shmdt 시스템 호출</p> <p>◆ 기능</p> <ul style="list-style-type: none"> ➢ 공유 메모리 연산 <p>◆ 사용법</p> <pre>#include <sys/shm.h> int *shmat(int shmid, const void *shmaddr, int shmflg); int shmdt(const void *shmaddr);</pre> <p>➢ shmat는 shmid가 가리키는 메모리 영역을 호출 프로세스 주소공간에 붙임</p> <p>➢ shmdt는 shmaddr의 메모리 영역을 호출 프로세스 주소공간으로부터 떼어냄</p> <p>➢ shmaddr : 프로세스 주소공간에 붙이거나 떼어낼 공유 메모리 영역의 주소</p> <ul style="list-style-type: none"> - NULL 일 때 시스템이 공유 메모리 영역이 부착될 적절한 주소를 선택 <p>➢ shmflg</p> <ul style="list-style-type: none"> - SHM_RND : shmaddr의 값에 가장 가까운 주소에 공유 메모리 영역이 부착 - SHM_RDONLY : 공유 메모리 영역은 해당 프로세스에 대해 읽기만을 허용 <p>◆ 반환값</p> <ul style="list-style-type: none"> ➢ shmat는 성공적인 호출에 대하여 부착된 공유 메모리 주소를 반환 ➢ shmdt는 성공적인 호출에 대하여 0을 반환 ➢ 오류시에는 -1 이 반환되고, errno 가 적절히 설정된다.
<p>(3) shmctl 시스템 호출</p> <p>◆ 기능</p> <ul style="list-style-type: none"> ➢ 공유 메모리 제어 (공유 메모리 영역의 정보를 얻거나, 변경하거나, 삭제) <p>◆ 사용법</p> <pre>#include <sys/shm.h> int shmctl(int mqid, int cmd, struct shmid_ds *buf);</pre> <p>➢ shmid가 가리키는 메모리 영역에 cmd가 지정한 동작을 수행</p> <p>➢ cmd : 수행할 명령</p> <ul style="list-style-type: none"> - IPC_STAT : shmid와 관련된 공유 메모리 영역의 정보를 buf의 위치에 저장 - IPC_SET : 공유 메모리 영역의 정보를 buf가 가리키는 값으로 변경, 변경 가능한 값은 shm_perm의 멤버들과 shm_ctime - IPC_RMID : 공유 메모리 영역과 관련된 데이터 구조를 즉시 삭제 <p>➢ buf : 공유 메모리 객체 데이터 구조의 주소</p> <pre>struct ipc_perm shm_perm; /* 연산 허가권 */ int shm_segsz; /* 영역의 크기 (바이트 수) */ time_t shm_atime; /* 최근 부착 시간 */ time_t shm_dtime; /* 최근 떼어낸 시간 */ time_t shm_ctime; /* 최근 변경 시간 */ unsigned short shm_cpid; /* 생성자의 pid */ unsigned short shm_lpid; /* 최근 연산자의 pid */ short shm_nattch; /* 현재 부착 횟수 */</pre>	

(4) 고급 프로세스 간 통신 – POSIX

◆ 관련 함수, 파일, 라이브러리

구분	메시지 큐	세마포어	공유메모리
헤더파일	mqqueue.h	semaphore.h	sys/mman.h
생성, 개방, 삭제 함수	mq_open mq_close mq_unlink	sem_open sem_close sem_unlink sem_init sem_destroy	shm_open shm_unlink
제어 함수	mq_getattr mq_setattr		ftruncate fstat
IPC 동작 함수	mq_send mq_receive mq_notify	sem_wait sem_trywait sem_post sem_getvalue	mmap munmap
접근 파일	/dev/mqueue	/dev/shm	/dev/shm
라이브러리	-lrt	-lpthread	-lrt

- POSIX는 서로 다른 UNIX OS의 공통 API를 정리하여 이식성이 높은 유닉스 응용 프로그램을 개발하기 위한 목적으로 IEEE가 책정한 애플리케이션 인터페이스 규격이다
- POSIX를 통하여 메시지 큐, 세마포어, 공유 메모리에 적용이 가능하다.
- 비동기 입출력이 가능하다.

6. 리눅스에서 비동기 통신 기법

- 프로그램이 입출력 연산을 수행하는 동안 블록되지 않고 비동기식으로 수행함.
- 라이브러리 수준에서 구현됨.
- 비동기 알림을 위해 쓰레드를 생성하거나 시그널을 사용함(비효율적임)
- 주의 : 리눅스 커널이 지원하는 별도의 비동기 입출력 기능이 있음

관련함수

◆ 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, 비동기 통지

■ 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 */
    ...
};
```

■ 비동기 통지

◆ 시그널 사용

◆ SIGEV_THREAD 방식

➢ 별도의 쓰레드 내의 `notification->sigev_notify_function` 호출

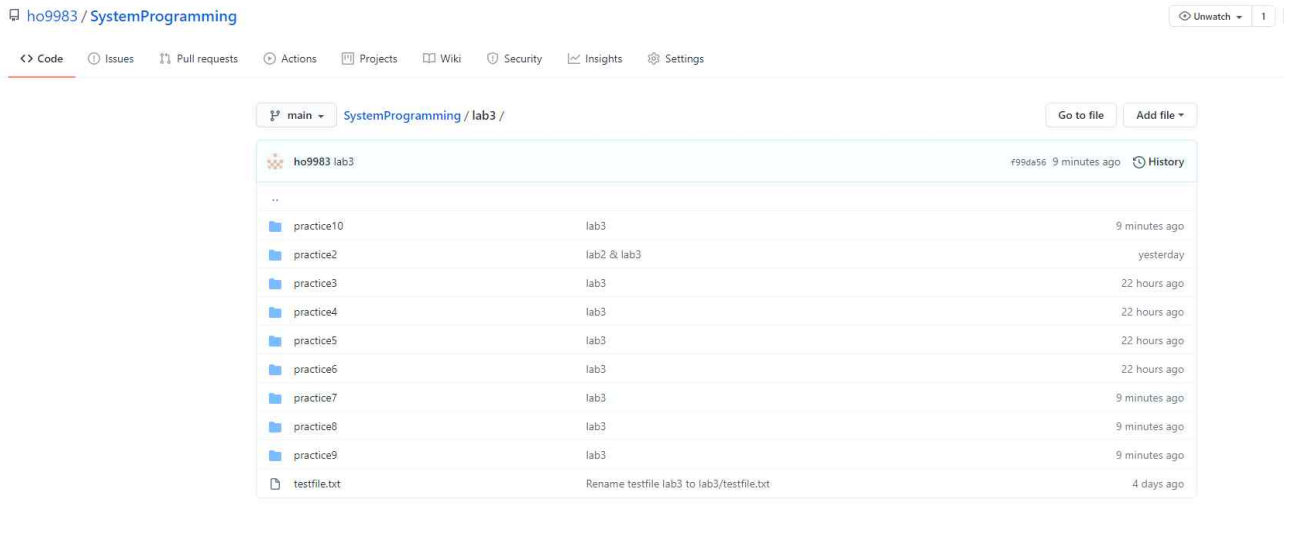
◆ SIGEV_SIGNAL 방식

➢ 인출력이 완료되면 `notification->sigev_signo`로 지정된 시그널 전송

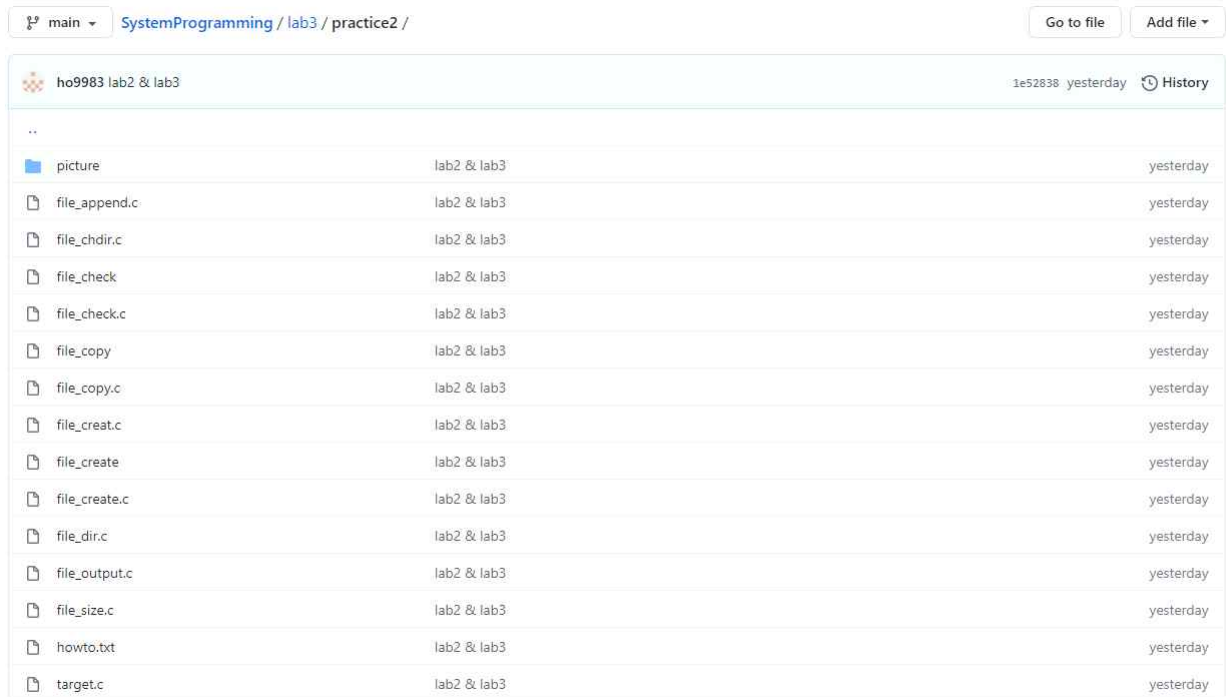
II. 실습

1. 자신의 github 저장소에 lab3 프로젝트를 생성하고 아래의 모든 과제 프로그램을 업로드 한다.

git url : <https://github.com/ho9983/SystemProgramming/tree/main/lab3>



2. 파일 및 디렉터리와 관련된 함수들을 사용하여 프로그램을 작성하고 실행하여보고, 익숙해지도록 사용해 본다.



- 일일이 모든 실습 과정에 대해 사진을 첨부하기엔 보고서가 지나치게 길어지기에 git에 있는 실습 파일들 목록으로 대체합니다.

3. 주어진 디렉터리 내에 존재하는 파일과 디렉터리를 나열하고, 디렉터리의 경우 재귀적으로 방문해서 그 디렉터리 내에 존재하는 파일과 디렉터리를 나열하는 프로그램을 작성하시오. 즉, "ls -R" 명령과 동일한 결과를 보이도록 하시오.

1) 소스코드

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <dirent.h>

void showDirStruct(const char *name);

int main(void) {
    showDirStruct(".");
    return 0;
}

void showDirStruct(const char *name){
    DIR *dir;
    struct dirent *fileInDir;
    char array[20][100];
    char *pathArray[20];
    char path[1024];
    int pnum=0, num=0;

    for(int a = 0; a<20; a++){
        pathArray[a] = NULL;
    }
    for(int b = 0; b<20; b++){
        for(int c = 0; c<100; c++){
            array[b][c] = 0;
        }
    }
    if (!(dir = opendir(name))) return;

    printf("%s: \n", name);
    while ((fileInDir = readdir(dir)) != NULL) {
        if (fileInDir->d_type == DT_DIR) {
            if (strcmp(fileInDir->d_name, ".") == 0 || strcmp(fileInDir->d_name, "..") == 0 || strcmp(fileInDir->d_name, ".git") == 0)
                continue;
            snprintf(path, sizeof(path), "%s/%s", name, fileInDir->d_name);
            for(int d = 0; d< strlen(path); d++){
                array[pnum][d] = path[d];
            }
            pathArray[pnum] = array[pnum];
            pnum = pnum+1;
            printf("%s ", pathArray[pnum-1]);
        }
        else { printf("%s ", fileInDir->d_name); }
    }
    printf("\n\n");
    while(pathArray[num] != NULL){
        if(pathArray[num] == NULL) break;
        showDirStruct(pathArray[num]);
        num = num+1;
    }
    closedir(dir);
}
```

← 메인 부분

함수 부분 →

2) 실행 화면

```
kch9983@ubuntu:~/System_programming/lab3$ ls
3practice3  practice2  practice3  practice4  testfile.txt
kch9983@ubuntu:~/System_programming/lab3$ ./3practice3
.:
testfile.txt  ./practice4  ./practice2  ./practice3  3practice3

./practice4:
practice4  practice4.c

./practice2:
target.c  file_output  ./practice2/picture  howto.txt  file_copy.c  output.txt  file_check  file_
check.c  file_chdir.c  file_size  file_dir.c  t.txt  file_append  file_create.c  file_output.c  f
ile_append.c  file_create  file_size.c  file_copy  file_dir  file_chdir

./practice2/picture:
file_check.png  file_size.png  file_chdir.png  file_create.png  file_append.png  file_copy.png  f
ile_output.png  file_dir.png

./practice3:
practice3.c  3practice3

kch9983@ubuntu:~/System_programming/lab3$ ls -R
.:
3practice3  practice2  practice3  practice4  testfile.txt

./practice2:
file_append  file_chdir.c  file_copy  file_create.c  file_output  file_size.c  picture
file_append.c  file_check  file_copy.c  file_dir  file_output.c  howto.txt  target.c
file_chdir  file_check.c  file_create  file_dir.c  file_size  output.txt  t.txt

./practice2/picture:
file_append.png  file_check.png  file_create.png  file_output.png
file_chdir.png  file_copy.png  file_dir.png  file_size.png

./practice3:
3practice3  practice3.c

./practice4:
practice4  practice4.c
kch9983@ubuntu:~/System_programming/lab3$
```


4. 몇 개의 문장을 타자하도록 하여 잘못 타이핑한 횟수와 평균 분당 타자수를 측정하는 타자 연습 프로그램을 구현하여 보시오.

1) 소스코드

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <termios.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>
#include <math.h>
#define PASSWORDSIZE 12

void typingFunc();

int main(void){
    typingFunc();
    return 0;
}
```

함수 부분, →
타자 오류 검사와
분당타수 부분이 같이 있다.

← 메인 부분

```
void typingFunc() {
    int fd;
    double speed=0, total=0;
    int nread, cnt=0, errcnt=0, typcnt=0;
    char ch, text[] = "The magic thing is that you can change it.";
    struct termios init_attr, new_attr;
    time_t start_time, end_time; // time.h

    fd = open(ttynamename(filename(stdin)), O_RDWR);
    tcgetattr(fd, &init_attr);

    new_attr = init_attr;
    new_attr.c_lflag &= ~ICANON;
    new_attr.c_lflag &= ~ECHO;

    new_attr.c_cc[VMIN] = 1;
    new_attr.c_cc[VTIME] = 0;
    if (tcsetattr(fd, TCSANOW, &new_attr) != 0) {
        fprintf(stderr, "Cannot set the terminal attribute\n");
    }
    printf("@@Type this sentence. It must be the same to the blank.@@ \n%s \n", text);
    start_time = time(NULL); // start timer
    while ((nread=read(fd, &ch, 1)) > 0 && ch != '\n') {
        if (ch == text[cnt++]) {
            write(fd, &ch, 1);
            typcnt++; //typing count
        }
        else {
            write(fd, "?", 1);
            errcnt++; //error count
        }
    }
    end_time = time(NULL); // end timer

    total = difftime(end_time, start_time); // The time between start_time and end_time

    speed = typcnt/(total/60); // Get the typing speed per minute
    printf("\nTyping error count is : %d\n", errcnt);
    printf("Speed average in 1 minute : %.1f\n", speed);
    tcsetattr(fd, TCSANOW, &init_attr);
    close(fd);
}
```

2) 실행 화면

```
kch9983@ubuntu:~/System_programming/lab3/practice4$ ls
pra4 pra4.c
kch9983@ubuntu:~/System_programming/lab3/practice4$ ./pra4
@@Type this sentence. It must be the same to the blank.@@
The magic thing is that you can change it.
The magic thing ?s that you can change i?.
Typing error count is : 2
Speed average in 1 minute : 200.0
kch9983@ubuntu:~/System_programming/lab3/practice4$
```

5. 프로세스와 관련된 함수들을 사용하여 프로그램을 작성하고 실행하여 보고, 익숙해지도록 사용해 본다.

1) 소스파일 목록들

main ▾ SystemProgramming / lab3 / practice5 /

ho9983 lab3	
..	
prac5Capture	lab3
execls	lab3
execls.c	lab3
exitprocess	lab3
exitprocess.c	lab3
forkprocess	lab3
forkprocess.c	lab3
waitprocess	lab3
waitprocess.c	lab3

2) 실행결과

(1) forkprocess.c

```
kch9983@ubuntu:~/System_programming/lab3/practice5$ ./forkprocess
Calling fork
I'm the parent process
I'm the child process
kch9983@ubuntu:~/System_programming/lab3/practice5$
```

(2) waitprocess.c

```
kch9983@ubuntu:~/System_programming/lab3/practice5$ gcc -o waitprocess waitprocess.c
kch9983@ubuntu:~/System_programming/lab3/practice5$ ./waitprocess
Exit status from 9724 was 5
kch9983@ubuntu:~/System_programming/lab3/practice5$
```

(3) exitprocess.c

```
kch9983@ubuntu:~/System_programming/lab3/practice5$ ./exitprocess
enter exit status : 0
kch9983@ubuntu:~/System_programming/lab3/practice5$ echo $?
0
kch9983@ubuntu:~/System_programming/lab3/practice5$ ./exitprocess
enter exit status : 2
kch9983@ubuntu:~/System_programming/lab3/practice5$ echo $?
2
kch9983@ubuntu:~/System_programming/lab3/practice5$
```

(4) execls.c

```
kch9983@ubuntu:~/System_programming/lab3/practice5$ gcc -o execls execls.c
kch9983@ubuntu:~/System_programming/lab3/practice5$ ./execls
Executing execl.
total 64
-rwxrwxr-x 1 kch9983 kch9983 8760 Nov 14 10:06 execls
-rw-rw-r-- 1 kch9983 kch9983 243 Nov 14 10:00 execls.c
-rwxrwxr-x 1 kch9983 kch9983 8720 Nov 14 10:03 exitprocess
-rw-rw-r-- 1 kch9983 kch9983 154 Nov 14 09:51 exitprocess.c
-rwxrwxr-x 1 kch9983 kch9983 8656 Nov 14 10:02 forkprocess
-rw-rw-r-- 1 kch9983 kch9983 387 Nov 14 10:02 forkprocess.c
-rwxrwxr-x 1 kch9983 kch9983 8864 Nov 14 10:05 waitprocess
-rw-rw-r-- 1 kch9983 kch9983 443 Nov 14 09:52 waitprocess.c
kch9983@ubuntu:~/System_programming/lab3/practice5$
```

6. `system` 함수는 쉘 명령이 실행되도록 하는데, 예를 들면, `system("ls -la")`을 호출하면, 현재 디렉터리의 파일들을 나열해 준다. 이와 같은 기능을 수행하는 함수를 직접 구현하여 보자. 또, 이 함수를 이용하는 예제 프로그램을 통해서 "a.out ls -la" 와 같이 명령이 잘 동작하도록 해 보자.

1) 소스코드

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

int main(int argc, char* argv[])
{
    if(argc == 1) {
        printf("Usage: %s <exec command> , [arg0, arg1, arg2, ...]\n", argv[0]);
        return 1;
    }
    pid_t pid = fork();
    if(pid < 0) {
        perror("fork error");
        return 1;
    }
    if(pid == 0) {
        wait(0);
    }
    else {
        execvp(argv[1], argv+1);
        return 0;
    }
}
```

2) 실행 화면

```
kch9983@ubuntu:~/System_programming/lab3/practice6$ gcc -o prac6 prac6.c
kch9983@ubuntu:~/System_programming/lab3/practice6$ ./prac6 ls -al
total 24
drwxrwxr-x 2 kch9983 kch9983 4096 Nov 14 10:14 .
drwxrwxr-x 7 kch9983 kch9983 4096 Nov 14 10:07 ..
-rwxrwxr-x 1 kch9983 kch9983 8808 Nov 14 10:14 prac6
-rw-rw-r-- 1 kch9983 kch9983 382 Nov 14 10:13 prac6.c
kch9983@ubuntu:~/System_programming/lab3/practice6$
```

7. 시그널과 관련된 함수들을 사용하여 프로그램을 작성하고 실행하여 보고, 익숙해지도록 사용해 본다.

main SystemProgramming / lab3 / practice7 /

ho9983 lab3	
..	
alarmsignal	lab3
alarmsignal.c	lab3
blocksignal	lab3
blocksignal.c	lab3
handlelgnoresignal	lab3
handlelgnoresignal.c	lab3
pionacci	lab3
pionacci.c	lab3
prac7handlesig.png	lab3
prac7handlesignal.png	lab3
sendsignal	lab3
sendsignal.c	lab3

- 실습 2번과 마찬가지로 소스 파일 목록으로 대체 합니다.

8. 프로세스 간 통신 함수들을 사용하여 프로그램을 작성하고 실행하여 보고, 익숙해지도록 사용해 본다.

main	SystemProgramming / lab3 / practice8 /	Go to file	Add file
ho9983 lab3	f99da56 26 minutes ago	History	
..			
filelock	lab3	26 minutes ago	
filelock.c	lab3	26 minutes ago	
iopipe	lab3	26 minutes ago	
iopipe.c	lab3	26 minutes ago	
readfifo	lab3	26 minutes ago	
readfifo.c	lab3	26 minutes ago	
readshm	pra8	23 hours ago	
readshm.c	pra8	23 hours ago	
recvpmq	pra8	23 hours ago	
recvpmq.c	pra8	23 hours ago	
sendmq	pra8	23 hours ago	
sendmq.c	pra8	23 hours ago	
testlock	lab3	26 minutes ago	
testsem	pra8	23 hours ago	
testsem.c	pra8	23 hours ago	
writefifo	lab3	26 minutes ago	
writefifo.c	lab3	26 minutes ago	
writeshm	pra8	23 hours ago	
writeshm.c	pra8	23 hours ago	

- 실습 2번과 마찬가지로 소스 파일 목록으로 대체 합니다.

9. 메시지 큐를 사용하여 텍스트 기반의 간단한 채팅 프로그램을 구현하시오.

1) 소스코드

(1) prac9_Client

```
int main() {
    mqd_t qd;
    struct mq_attr q_attr;
    char send_data[BUFSIZE]; //define BUFSIZE 32
    int status;
    pid_t pid;

    q_attr.mq_maxmsg = 10;
    q_attr.mq_msgsize = BUFSIZE;

    while(1){
        if ((pid = fork()) == 0){
            memset(send_data, 0, BUFSIZE);
            printf("Input > ");
            scanf("%s", send_data);
            // define QNAME "my_queue"
            if ((qd = mq_open(QNAME, O_CREAT | O_RDWR, 0600, &q_attr)) == -1) {
                perror ("mq_open failed");
                exit (1);
            }
            //define PRIORITY 1
            if (mq_send(qd, send_data, strlen(send_data), PRIORITY) == -1) {
                perror ("mq_send failed");
                exit (1);
            }
            if (mq_close(qd) == -1) {
                perror ("mq_close failed");
                exit (1);
            }
            exit(0);
        } else if (pid > 0){
            pid = wait(&status);
            sleep(1);
        } else {
            perror("fork failed");
            exit(1);
        }
    }
}
```


(2) prac9_Server

```
#define BUFSIZE 32
#define QNAME "/my_queue"
#define PRIORITY 1

char recv_data[BUFSIZE];

int main() {
    mqd_t qd;
    pid_t pid;
    int status;
    struct mq_attr q_attr, old_q_attr; int prio;

    char buf[BUFSIZE];
    q_attr.mq_maxmsg = 10;
    q_attr.mq_msgsize = BUFSIZE;

    while(1){
        if ((pid = fork()) == 0){

            if ((qd = mq_open(QNAME, O_RDWR | O_NONBLOCK, 0600, NULL)) == -1) {
                perror ("mq_open failed");
                exit (1);
            }
            q_attr.mq_flags = 0;
            if (mq_setattr(qd, &q_attr, NULL)) {
                exit (1);
            }

            if (mq_getattr(qd, &old_q_attr)) {
                perror ("mq_getattr failed");
                exit (1);
            }

            if (mq_receive(qd, recv_data, BUFSIZE, &prio) == -1) {
                perror ("mq_send failed");
                exit (1);
            }

            printf ("Client Say : %s \n", recv_data);

            if (mq_close(qd) == -1) {
                pid = wait(&status);
                perror ("mq_close failed");
                exit (1);
            }

            if (mq_unlink(QNAME) == -1) {
                perror ("mq_unlink failed");
                exit (1);
            }
            exit(0);
        }else if(pid > 0){
            pid = wait(&status);
            sleep(3);
        }else{
            perror("fork failed");
            exit(1);
        }
    }
}
```

2) 실행 화면

```
kch9983@ubuntu: ~/System_programming/lab3/practice9
kch9983@ubuntu:~/System_programming/lab3/practice9$ ./pra9_client
Input > hi
Input > Is there anybody!?
Input > OMG..
Input > BYEBYE
Input > ^C
kch9983@ubuntu:~/System_programming/lab3/practice9$
```

```
kch9983@ubuntu: ~/System_programming/lab3/practice9
kch9983@ubuntu:~/System_programming/lab3/practice9$ ./pra9_server
mq_open failed: No such file or directory
Client Say : hi
mq_open failed: No such file or directory
mq_open failed: No such file or directory
mq_open failed: No such file or directory
Client Say : Is there anybody!?
mq_open failed: No such file or directory
mq_open failed: No such file or directory
Client Say : OMG..
mq_open failed: No such file or directory
Client Say : BYEBYE
mq_open failed: No such file or directory
mq_open failed: No such file or directory
^C
kch9983@ubuntu:~/System_programming/lab3/practice9$
```

10. 공유 메모리를 사용하여 한 파일을 다른 파일로 복사하는 프로그램을 작성하시오. 단, 부모(읽는 프로세스)와 자식(쓰는 프로세스)프로세스가 공유 메모리 영역을 동시에 접근하는 일이 없도록 세마포어 같은 동기화 기법을 활용하시오.

- 해당 실습에 대해서는 시간이 없어서 하지 못했습니다.
- 추후에 시험기간에 공부차원에서 다시 업로드 할 것입니다.

11. 간단한 쉘 프로그램을 만들고 다음과 같이 동작하도록 수정하시오.

- 1) "exit"를 치면 프로그램을 끝내도록 프로그램을 수정하시오.
- 2) csh, bash 등에서처럼 쉘 명령의 마지막에 '&'을 입력하면 백 그라운드로 실행 되도록 프로그램을 수정하시오.
- 3) csh, bash 등에서처럼 인터럽트 키(SIGINT: Ctrl-C, SIGQUIT: Ctrl-Z) 가 동작하도록 프로그램을 수정하시오.
- 4) 파일 재지향(>, <) 및 파이프(|) 기능이 가능하도록 프로그램을 수정하시오.
- 5) ls, pwd, cd, mkdir, rmdir, ln, cp, rm, mv, cat 명령을 팀원이 공평하게 나누어 구현하시오.

- 해당 내용은 프로젝트 보고서를 통하여 제출 하였습니다.

Ⅲ. 검토 및 소감

리눅스에는 윈도우에서 굳이 컴파일러를 깔아야 하는 것 단점을 커버 할 수 있는 쉘 프로그래밍이 가능하다는 것에 대한 편리성을 다시 한 번 느꼈고, 리눅스 안에서의 파일처리 함수 예제코드와 프로세스, 프로세스 간 통신 예제코드를 통해 리눅스 프로그래밍에 대해 어느 정도 익숙해 졌다는 것을 느꼈습니다.

하지만 아직 프로세스와 스레드 간의 통신방식에 대해 위에 기술한 것이 맞는 것인지 궁금하기도 하고, 또 아직 제가 프로세스 통신 기법에 대해 심화과정이나 응용으로 넘어간다면 완전히 헤맨다는 것을 느꼈습니다.

해당 리포트를 통한 실습 이외에도 다시 강의 자료나, 리눅스 프로그래밍 관련 예제들을 다시 차근차근 풀어보면서 응용하는 실력을 키워야겠다는 생각이 들었습니다.