시스템 프로그래밍

리눅스&유닉스

시스템 프로그래밍

리눅스&유닉스

Ch. 01 리눅스/유닉스 시스템 프로그래밍의 이해

목차

학습목표

- 리눅스/유닉스 시스템과 관련된 표준을 이해한다.
- 리눅스/유닉스 시스템 프로그래밍이 무엇인지 이해한다.
- 시스템 호출과 라이브러리 함수의 차이를 이해한다.
- 리눅스/유닉스 시스템의 기본 명령을 사용할 수 있다.
- C 컴파일러와 make 도구를 사용할 수 있다.

01. 리눅스/유닉스 시스템 프로그래밍이란

■ 리눅스와 유닉스

- 리눅스와 유닉스
  - 서버용 운영체제로 주로 사용
  - 최근엔 원조격이라고 할 수 있는 유닉스를 리눅스가 서버 운영체제 시장에서 밀어냄
  - 금융권에서는 유닉스 시스템을 리눅스 시스템으로 대체하는 U2L이 확산

- 시스템 호출
  - 시스템이 제공하 는 서비스를 프로그램에서 이용할 수 있도록 지원하는 프로그래밍 인터페이스를 의미
  - 리눅스/유닉스에서 동작하는 프로그램을 작성하려면 간단한 프로그램을 제외하고 대부분 시스템 호출을 이용

02. 리눅스/유닉스 시스템 표준

■ 유닉스

- 유닉스의 개발
  - 1969년에 미국의 통신 회사인 AT&T 산하의 벨 연구소에서 켄 톰슨과 데니스 리치가 개발한 운영체제
  - 처음에는 기존 운영체제처럼 어셈블리어로 개발
  - 데니스 리치가 개발한 C 언어를 사용해 1973년에 다시 만들면서 고급 언어로 작성된 최초의 운영체제

- 유닉스의 기능
  - 유닉스는 초기에 소스 코드가 공개되어 대학교나 기업에서 쉽게 이용할 수 있었고 이에 따라 다양한 기능이 추가
  - AT&T의 상용 유닉스(시스템 V)와 버클리 대학교의 BSD 계열로 나뉘어 각각 발전
  - BSD 버전은 버클리 대학교 학생들이 많은 기능을 추가했는데 그 중 TCP/IP 기반의 네트워크 기능을 추가

02. 리눅스/유닉스 시스템 표준

■ 유닉스

- 유닉스의 발전
  - 1991년에 등장한 리눅스는 오픈 소스로 공개되어 지속적으로 발전
  - 리눅스는 같은 커널을 기반으로 하는 다양한 형태의 배포판을 사용

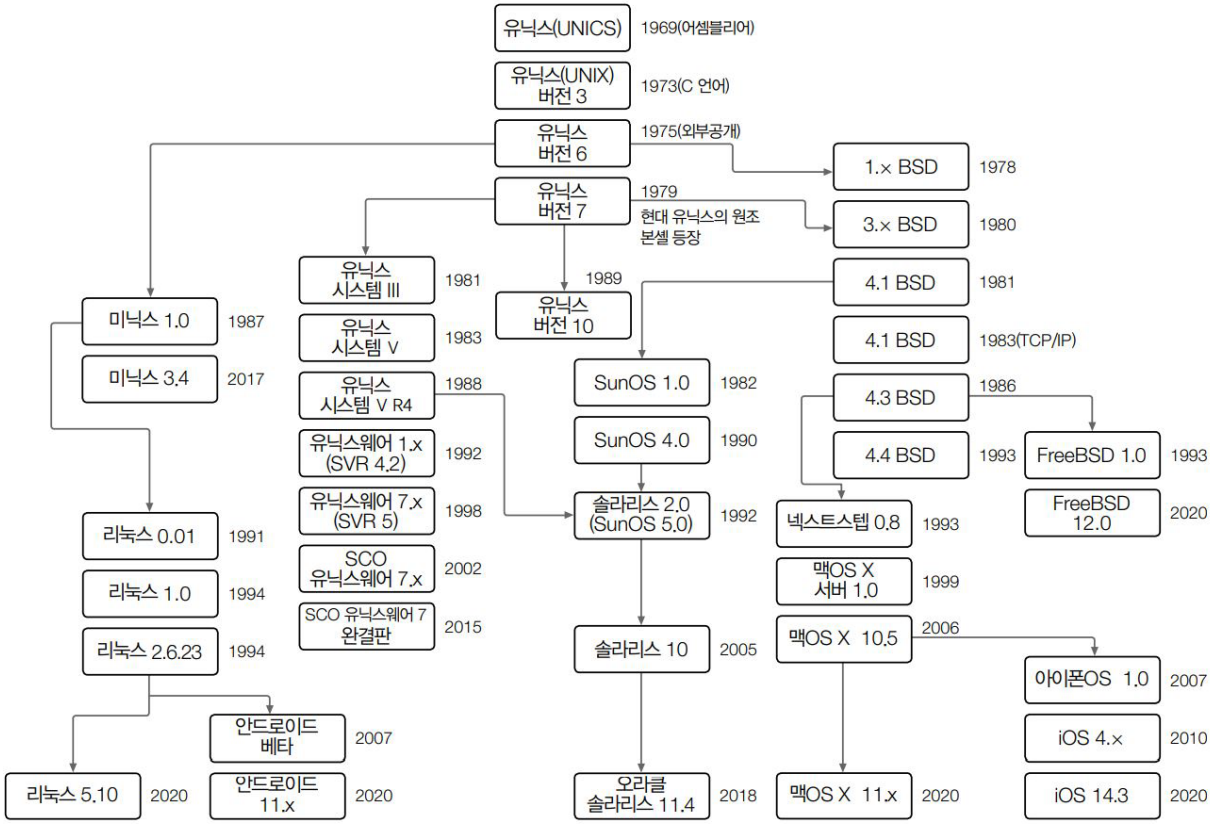

그림 1-1 유닉스와 리눅스의 발전 과정

02. 리눅스/유닉스 시스템 표준

■ 유닉스의 주요 표준

- ANSIC 표준
  - 미국 표준 협회로, 국제적으로 영향력 있는 표준을 정함
  - ANSI에서 표준화한 C 언어 명세가 ANSI C 표준으로, C 언어 문법과 라이브러리, 헤더 파일 등을 정의

- POSIX
  - 유닉스에 기반을 두고 있는 표준 운영체제 인터페이스
  - 서로 다른 유닉스 시스템 사이에서 상호 이식이 가능한 응용 프로그램을 개발하기 위해 정해진 표준
  - IEEE에서 정의한 규격으로, 유닉스 시스템의 공통 응용 프로그래밍 인터페이스를 정리

- X/Open 가이드
  - 1984년에 유럽의 유닉스 시스템 제조업체를 중심으로 설립된 단체로, 개방 시스템에 관한 표준 정의와 보급을 목적
  - 다양하게 파생되고 있는 유닉스 시스템 에서 응용 프로그램의 이식성을 높이는 것이 초기 목표
  - 1988년에 발표한 XPG3에서는 POSIX 표준을 통합했고, XPG의 최종 버전인 XPG4는 1992년에 발표

02. 리눅스/유닉스 시스템 표준

■ 유닉스의 주요 표준

- 단일 유닉스 명세 (SUS)
  - 운영체제가 유닉스라는 이름을 사용하기 위해 지켜야 하는 표준의 총칭
  - IEEE, ISO(JTC 1 SC22), 오픈 그룹의 표준화 작업결과물에 바탕을 두고 있으며 오스틴 그룹이 개발 및 유지·관리를 담당
  - 1980년대 중반부터 시작된 유닉스의 시스템 인터페이스를 표준화하기 위한 프로젝트에서 출발

- 시스템 V 인터페이스 정의 (SVID)
  - AT&T 유닉스 시스템 V의 인터페이스를 정의
  - 프로그램과 장치에서 이용할 수 있는 시스템 호출과 C 라이브러리에 관한 표준을 포함
  - POSIX나 X/Open 작업은 부분적으로 SVID에 기반
  - 1995년에 발표된 SVID 버전 4는 XPG4 및 POSIX 1003.1-1990과 호환성을 유지
  - SVID는 POSIX와 단일 유닉스 명세에 포함되면서 중요도가 떨어짐

03. 시스템 프로그래밍

■ 시스템 호출과 라이브러리 함수

- 시스템 호출

```
리턴값 = 시스템 호출명(인자, ...);
```

  - 시스템 호출명은 함수명처럼 사용할 이름이 정의

- 라이브러리 함수
  - 라이브러리: 미리 컴파일된 함수를 묶어서 제공하는 특수한 형태의 파일
  - 라이브러리 함수: 라이브러리에 포함된 함수를 의미
  - 리눅스 시스템에서 라이브러리는 보통 /usr/lib에 위치

  - 정적 라이브러리는 프로그램을 컴파일할 때 같이 적재되어 실행 파일을 구성
  - 공유 라이브러리는 실행 파일에 포함되지 않아 메모리를 효율적으로 사용하기 위해 사용

03. 시스템 프로그래밍

■ 시스템 호출과 라이브러리 함수

- 시스템 호출과 라이브러리 함수 비교

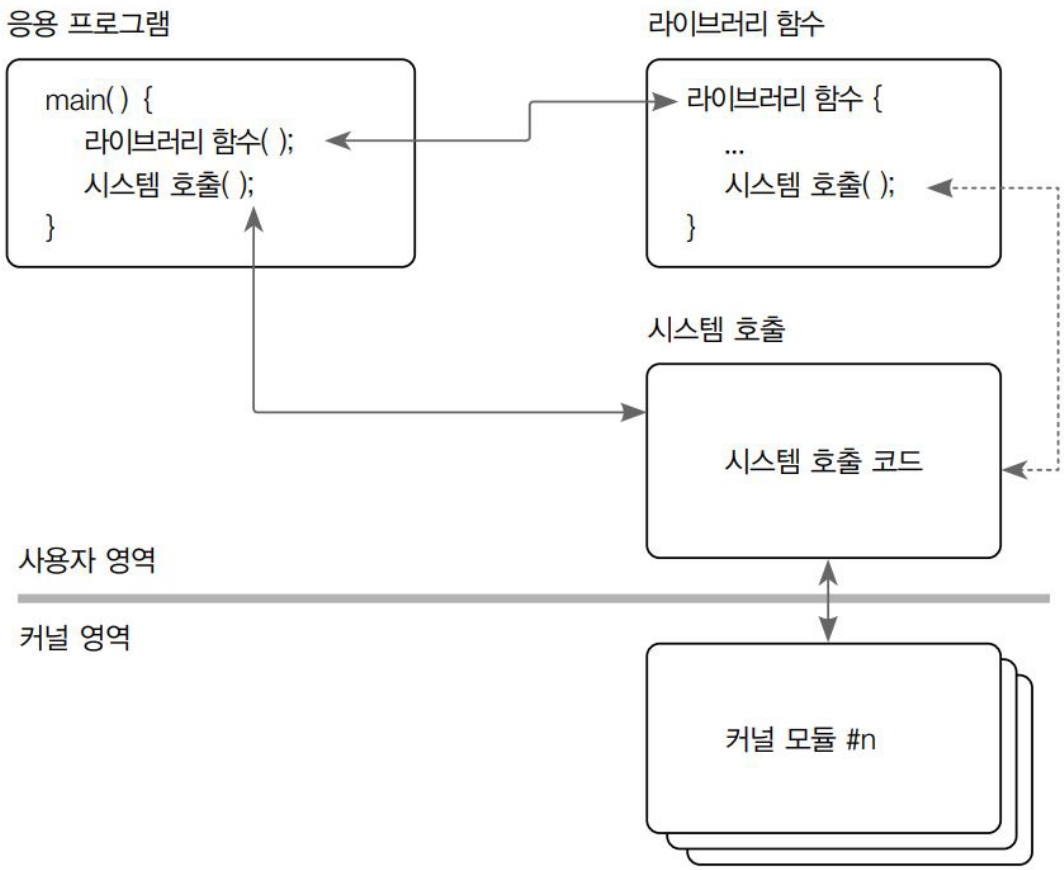

그림 1-2 시스템 호출과 라이브러리 함수의 비교

03. 시스템 프로그래밍

■ 시스템 호출과 라이브러리 함수

▪ man 페이지의 섹션 번호

- man 페이지
  - 명령이나 함수 등 시스템의 다양한 서비스에 대한 매뉴얼
  - 매뉴얼은 항목의 종류에 따라 섹션이 구분되어있음
    - 리눅스에서 사용하는 일반적인 명령에 대한 설명: 섹션 1
    - 시스템 호출: 섹션 2
    - 라이브러리 함수: 섹션 3
  - man 명령으로 검색하면 섹션 번호가 가장 낮은 것이 기본으로 출력됨
  - man 명령의 결과를 출력하는 형식은 리눅스와 유닉스에서 차이가 있음

03. 시스템 프로그래밍

■ 시스템 호출과 라이브러리 함수

▪ man 옵션

-a: 찾고자 하는 명령어의 검색된 메뉴얼 페이지를 모두 출력한다.
-h: 사용법을 출력
-f: whatis 명령과 동일하다.(명령어에 대한 기능을 간략하게 나타낸다.)
-k: aprpose 명령과 동일하다.(지정한 키워드를 포함하고 있는 명령어이다.)
-w: 찾고자 하는 문자의 메뉴얼 페이지가 있는 위치를 출력한다.

옵션	긴버전	설명
-k		해당 키워드로 발견되는 모든 매뉴얼의 내용을 검색해서 보여줌
-f		해당 키워드에 대해 완벽히 일치되는 매뉴얼 페이지에 대한 정보를 보여줌
-w	--path	man 명령 실행 시에 호출되는 '매뉴얼 페이지' 파일의 위치를 보여줌
-s, -S	--sections=섹션번호	특정 섹션을 지정할 때 사용

man 매뉴얼 섹션
1: 일반 명령어 메뉴얼 영역
2: 시스템 호출에 관한 메뉴얼 영역
3: C 표준 라이브러리 함수 관련 메뉴얼 영역
4: 장치 드라이버 특수파일에 대한 정보가 들어 있는 메뉴얼 영역
5: 특정 파일에 대한 정보가 들어있는 영역
6: 게임과 화면보호기 에 대한 정보가 들어가 있는 영역
7: 리눅스 파일 표준, 프로토콜, 시그널 목록 정보가 들어가있는 영역
8: 시스템 관리 명령어와 데몬 정보가 들어있는 영역
9: 커널 관리 정보가 들어있는 영역

Section 이름	설명
man1	실행가능한 프로그램 혹은 쉘 명령어
man2	시스템 호출
man3	라이브러리 호출
man4	Special File (장치, 장치 드라이버, Socket, /dev 디렉토리에 있는 형식과 관련된 규약 등)
man5	파일 포맷과 컨벤션 (예를 들어 /etc/passwd의 데이터 구성은 어떻게 되어 있다.. 등)
man6	Games
man7	Miscellanea (리눅스 시스템 파일 관련 표준, 규칙, 프로토콜, 시그널 목록 등)
man8	시스템 관리자를 위한 명령어
man9	리눅스 커널 루틴

03. 시스템 프로그래밍

■ 시스템 호출과 라이브러리 함수

▪ man 페이지의 섹션 번호

- 리눅스(우분투 리눅스)와 유닉스(솔라리스)에서 open() 함수를 검색하면 open(2)'로 표시 (섹션 2)
- 리눅스에서는 상단에 'System Calls'라고 명시

[리눅스(우분투 리눅스)]

```
OPEN(2)          Linux Programmer's Manual          OPEN(2)

NAME
       open, openat, creat - open and possibly create a file

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

[유닉스(솔라리스)]

```
OPEN(2)          Linux Programmer's Manual          OPEN(2)

NAME
       open, openat - open a file

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

03. 시스템 프로그래밍

■ 시스템 호출과 라이브러리 함수

▪ man 페이지의 섹션 번호

- 리눅스(우분투 리눅스)와 유닉스(솔라리스)에서서 fopen() 함수를 검색
- 리눅스에서는 'fopen(3)'으로 표시되어 섹션 3에 속한 함수인 것만 알리지만, 유닉스에서는 'fopen(3C)'로 표시

[리눅스(우분투 리눅스)]

```
OPEN(2)          Linux Programmer's Manual          OPEN(3)

NAME
       fopen, fdopen, freopen - stream open functions

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

[유닉스(솔라리스)]

```
OPEN(2)          Linux Programmer's Manual          OPEN(3C)

NAME
       fopen - open a stream
       fopen_s - open a stream with additional safety checks

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

03. 시스템 프로그래밍

■ 시스템 호출과 라이브러리 함수

▪ man 페이지의 섹션 번호

- 명령과 함수의 이름이 같은 경우: 'man uname'을 입력하면 명령(섹션 1)에 대한 설명만 볼 수 있음
- 섹션 2의 설명을 보려면 다음과 같이 해당 섹션을 지정해야 함

```
$ man uname

UNAME(1)          User Commands          UNAME(1)

NAME
       uname - print system information

SYNOPSIS
       uname [OPTION] ...
```

```
$ man -s 2 uname

UNAME(2)          Linux Programmer's Manual          UNAME(2)

NAME
       uname - get name and information about current kernel

SYNOPSIS
       #include <sys/utsname.h>
```

03. 시스템 프로그래밍

■ [예제 1-1] 시스템 호출의 오류 처리하기

```
01 #include <stdio.h>
02 #include <unistd.h>
03 #include <errno.h>
04
05 extern int errno;
06
07 int main() {
08     if(access("test.txt", F_OK) == -1) {
09         printf("errno=%d₩n", errno);
10     }
11 }
```

실행

```
$ ./ch1_1.out
errno=2
```

- **08~10행** access( ) 함수의 리턴값 검사, –1이면 오류 발생이므로 전역 변수 errno 값 검사
- **실행 결과** errno 변수 값은 2. 해당 시스템 호출에서 발생한 오류가 무엇인지 알려줌

■ 시스템 호출과 라이브러리 함수

▪ man 페이지의 섹션 번호

- errno에 저장된 값 2가 의미하는 바를 해석하려면 헤더 파일을 참조
- 리눅스는 asm-generic/errno-base.h 파일에 정의되어 있고 유닉스는 sys/errno.h 파일에 정의

```
$ vi /usr/include/asm-generic/errno-base.h
/* SPDX-License-Identifier : GPL-2.0 WITH Linux-syscall-note */
#ifndef _ASM_GENERIC_ERRNO_BASE_H
#define _ASM_GENERIC_ERRNO_BASE_H

#define EPERM           1               /* Operation not permitted */
#define ENOENT          2               /* No such file or directory */
(생략)
```

03. 시스템 프로그래밍

■ 시스템 호출과 라이브러리 함수

▪ man 페이지의 섹션 번호

- access() 함수에서 발생하는 오류 코드로는 EACCES, ELOOP, ENAMETOOLONG, ENODIR, EROFS 등이 있음
- man access 명령으로 access() 함수에서 발생하는 오류 코드와 해당 설명을 확인

```
$ man access
ACCESS(2)              Linux Programmer's Manual              ACCESS(2)

NAME
       access, faccessat - check user's permissions for a file
 (생략)
 ERRORS
       access() and faccessat() shall fail if:
       EACCES The requested access would be denied to the file, or search permission is denied
                    for one of the directories in the path prefix of pathname. (See also path_
                    resolution(7).)

       ELOOP Too many symbolic links were encountered in resolving pathname.

       ENAMETOOLONG
                    pathname is too long.
       ENOENT A component of pathname does not exist or is a dangling symbolic link.
       (생략)
```

03. 시스템 프로그래밍

■ [예제 1-2] 라이브러리 함수의 오류 처리하기

```
01 #include <stdlib.h>
02 #include <stdio.h>
03 #include <errno.h>
04
05 extern int errno;
06
07 int main() {
08     FILE *fp;
09
10     if((fp=fopen("test.txt", "r")) == NULL) {
11         printf("errno=%d₩n", errno);
12         exit(1);
13     }
14     fclose(fp);
15 }
```

실행

```
$ ./ch1_2.out
errno=2
```

- **10행** 라이브러리 함수인 fopen()을 사용해 test.txt 파일을 실행
  여기서는 파일이 존재하지 않으므로 오류가 발생해 NULL을 리턴
- **실행 결과** errno에 저장된 값이 2임을 알 수 있음
  fopen() 함수에서 발생할 수 있는 오류 코드는 man 페이지에서 찾아볼 수 있음

04. 시스템 도구

■ 기본 명령

■ 로그인/로그아웃 명령

명령	기능	주요 옵션	예제
telnet	리눅스 시스템에 접속	–	telnet ***.co.kr
ssh			ssh ***.co.kr
exit	리눅스 시스템 접속 해제	–	exit
logout			logout

04. 시스템 도구

■ 기본 명령

■ 파일/디렉터리 명령

명령	기능	주요 옵션	예제
pwd	현재 디렉터리 경로 출력	–	pwd
ls	디렉터리 내용 출력	–a : 숨김 파일 출력	ls –a /tmp
		–l : 파일 상세 정보 출력	ls –l
cd	디렉터리 이동	–	cd /tmp
			cd ~han01
cp	파일 복사	–	cp a.txt b.txt
	디렉터리 복사	–r : 디렉터리 복사	cp –r dir1 dir2
mv	파일명/디렉터리명 변경	–	mv a.txt b.txt
			mv dir1 dir2
	파일/디렉터리 이동	–	mv a.txt dir1
rm	파일 삭제	–	rm a.txt
	디렉터리 삭제	–r : 디렉터리 삭제	rm –r dir1
mkdir	디렉터리 생성	–	mkdir dir1
rmdir	빈 디렉터리 삭제	–	rmdir dir1
cat	파일 내용 출력	–	cat a.txt
more	화면 크기 단위로 파일 내용 출력	–	more a.txt
chmod	파일/디렉터리 접근 권한 변경	–	chmod 755 a.exe
			chmod go+x a.exe
grep	패턴 검색	–	grep abcd a.txt

04. 시스템 도구

■ 기본 명령

■ 프로세스 명령

명령	기능	주요 옵션	예제
ps	현재 실행 중인 프로세스의 정보 출력	–ef : 모든 프로세스의 상세 정보 출력	ps ps -ef ps -ef | grep ftp
kill	프로세스 강제 종료	–9 : 강제 종료	kill 5000 kill -9 5001

■ 기타 명령

명령	기능	주요 옵션	예제
su	사용자 계정 변경	– : 변경할 사용자의 환경 초기화 파일 실행	su su - su - han02
tar	파일/디렉터리 묶기	cvf : tar 파일 생성	tar cvf a.tar *
		tvf : tar 파일 내용 보기	tar tvf a.tar
		xvf : tar 파일 풀기	tar xvf a.tar
whereis	파일 위치 검색	–	whereis ls
which		–	which telnet

04. 시스템 도구

■ 기본 명령

- vi 편집기 내부 명령
  - vi는 기본 문서 편집기, vi로 문서를 편집하려면 다음과 같이 파일명을 지정해 문서를 실행

```
$ vi test.c
```

기능	명령	기능	명령
입력 모드 전환	i, a, o, O	명령 모드 전환	Esc
커서 이동	j, k, h, l 또는 방향키	행 이동	#G(50G, 143G 등) 또는 :행 번호
한 글자 수정	r	여러 글자 수정	#s(5s, 7s 등)
단어 수정	cw	명령 취소	u, U
검색해 수정	:%s/aaa/bbb/g	복사	#yy(5yy, 10yy 등)
붙이기	p	커서 이후 삭제	D(Shift + D)
글자 삭제	x, #x(3x, 5x 등)	행 삭제(잘라내기)	dd, #dd(3dd, 4dd 등)
저장하고 종료	:wq! 또는 ZZ	저장하지 않고 종료	:q!
행 붙이기	J(Shift + J)	화면 다시 표시	Ctrl + l (소문자 L)
행 번호 보이기	:set nu	행 번호 없애기	:set nonu

04. 시스템 도구

■ 컴파일 환경

- 컴파일이란
  - 텍스트로 작성한 프로그램을 시스템이 이해할 수 있는 기계어로 변환하는 과정
  - '컴파일을 한다' = 보통 컴파일 과정과 라이브러리 링크 과정을 하나로 묶어서 수행하는 것

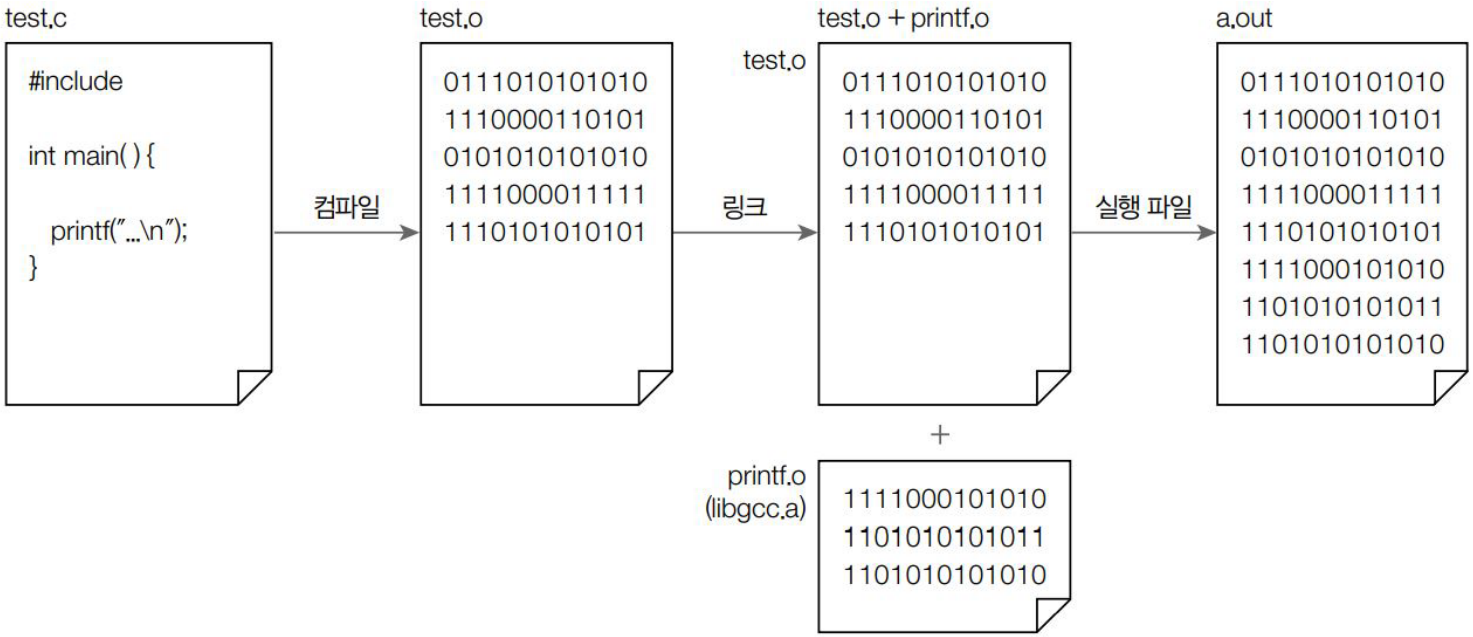

그림 1-3 C 프로그램의 컴파일 과정 예

04. 시스템 도구

■ GNU C 컴파일러 : gcc

▪ gcc

- 기능 = C 프로그램을 컴파일해 실행 파일을 생성
- 형식 = gcc [옵션][파일명]
- 옵션 -c : 오브젝트 파일(.o)만 생성
  -o 실행 파일명 = 지정한 이름으로 실행 파일을 생성, 기본 실행 파일명은 a.out
- 사용 예: $ gcc test.c
  $ gcc -c test.c
  $ gcc -o test test.c

04. 시스템 도구

■ GNU C 컴파일러 : gcc

▪ [예제 1-2] 컴파일 과정

- 파일명을 별도로 지정하지 않았으므로 a.out이라는 이름으로 실행 파일이 생성

```
$ gcc ch1_2.c
$ ls
a.out ch1_2.c
```

- 실행 파일명을 ch1_2.out이라고 하려면 다음과 같이 -o 옵션을 사용

```
$ gcc –o ch1_2.out ch1_2.c
$ ls
a.out ch1_2.out ch1_2.c
```

- 실행 파일명을 입력하면 프로그램이 실행
- 현재 디렉터리(.)가 경로에 설정되어 있지 않 다면 현재 디렉터리를 지정해 실행
- 이후에는 현재 디렉터리가 경로에 있다고 가정하여 ./를 표시하지 않음

$ ch1_2.out (현재 디렉터리가 경로에 있을 경우)
(또는)
$ ./ch1_2.out (현재 디렉터리가 경로에 없을 경우)

04. 시스템 도구

■ GNU C 컴파일러 : gcc

▪ Makefile과 make

- Makefile: 컴파일 명령, 소스 파일을 컴파일하는 방법, 링크할 파일, 실행 파일명 등을 설정 하는 파일
- make 명령: Makefile을 읽고 이 파일에서 지정한대로 컴파일을 실행해 실행 파일을 생성

```
$ sudo apt install make
```

04. 시스템 도구

■ [예제 1-3] make 명령 사용하기 (1)

```
01 #include <stdio.h>
02
03 extern int addnum(int a, int b);
04
05 int main() {
06      int sum;
07
08      sum = addnum(1, 5);
09      printf("Sum 1~5 = %d₩n", sum);
10 }
```

- **03행** 외부 파일에 정의된 addnum() 함수를 사용함을 선언
- **08행** addnum() 함수를 호출한

04. 시스템 도구

■ [예제 1-3] make 명령 사용하기 (2)

```
01 int addnum(int a, int b) {
02      int sum = 0;
03
04      for (; a <= b; a++)
05          sum += a;
06      return sum;
07 }
```

- addnum() 함수는 사용자 정의 함수user defined function로, 인자로 받은 범위의 합계를 계산해 리턴

04. 시스템 도구

■ [예제 1-3] make 명령 사용하기 (3)

```
01 # Makefile
02
03 CC=gcc
04 CFLAGS=
05 OBJS=ch1_3_main.o ch1_3_addnum.o
06 LIBS=
07 all : add.out
08
09 add.out : $(OBJS)
10      $(CC) $(CFLAGS) -o add.out $(OBJS) $(LIBS)
11
12 ch1_3_main.o : ch1_3_main.c
13      $(CC) $(CFLAGS) -c ch1_3_main.c
14 ch1_3_addnum.o : ch1_3_addnum.c
15      $(CC) $(CFLAGS) -c ch1_3_addnum.c
16
17 clean:
18      rm -f $(OBJS) add.out *.o core
```

- **03행** 컴파일 명령을 gcc로 지정
- **04행** 컴파일 옵션이 필요한 경우에 지정
- **05행** 생성할 오브젝트 파일명을 지정
- **06행** 기본 gcc 라이브러리 외에 다른 라이브러리가 필요한 경우 지정
- **07행** 생성할 실행 파일명을 add.out으로 지정
- **09~10행** 실행 파일인 add.out을 어떻게 생성할 것인지 지정
- **12~15행** 각 오브젝트 파일을 어떻게 생성할 것인지 지정
- **17~18행** make clean을 수행할 때 실행할 명령을 지정

04. 시스템 도구

■ [예제 1-3] make 파일 실행 결과

```
$ ls
Makefile ch1_3_addnum.c ch1_3_main.c
$ make
gcc -c ch1_3_main.c
gcc -c ch1_3_addnum.c
gcc -o add.out ch1_3_main.o ch1_3_addnum.o
$ ls
Makefile add.out ch1_3_addnum.c ch1_3_addnum.o ch1_3_main.c ch1_3_main.o
$ add.out
Sum 1~5 = 15
```

- 실행 파일과 오브젝트 파일을 모두 삭제하려면 make clean을 수행

```
$ ls
Makefile add.out ch1_3_addnum.c ch1_3_addnum.o ch1_3_main.c ch1_3_main.o
$ make clean
rm -f ch1_3_main.o ch1_3_addnum.o add.out *.o core
$ ls
Makefile ch1_3_addnum.c ch1_3_main.c
```

04. 시스템 도구

■ 오류 메시지 출력 : perror(3)

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

void perror(const char *s);
```

- s : 출력할 문자열

- perror()함수의 특징
  - 실행 파일과 오브젝트 파일을 모두 삭제하려면 make clean을 수행
  - Perror( ) 함수는 errno에 저장된 값을 읽어 이에 해당하는 메시지를 표준 오류(파일 기술자 2번)로 출력
  - Perror(3) 함수의 인자로는 일반적으로 프로그램 이름을 지정하는 것이 좋음

04. 시스템 도구

■ [예제 1-4] perror() 함수로 오류 메시지 출력하기

```
01 #include <stdio.h>
02 #include <unistd.h>
03 #include <errno.h>
04 #include <stdlib.h>
05
06 int main() {
07     if(access("test.txt", R_OK) == -1) {
08         perror("test.txt");
09         exit(1);
10     }
11 }
```

실행

```
$ ch1_4.out
test.txt: No such file or directory
```

- **07~08행** access() 함수에서 오류가 발생하면 perror() 함수를 호출한다. 이때 perror() 함수의 인자로 "test.txt"를 지정
- **09행** perror() 함수는 오류 메시지 출력만 하므로 오류의 결과로 프로그램을 종료해야 한다면 exit() 함수를 호출해야 함
- **실행 결과** 인자로 전달한 문자열 "test.txt"와 콜론이 출력되고 한 칸 띄어서 메시지가 출력

04. 시스템 도구

■ 오류 메시지 출력 : strerror(3)

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

char *strerror(int errnum);
```

- errnum : errno에 저장된 값

- strerror() 함수의 특징
  - strerror() 함수는 ANSI C에서 추가로 정의한 함수
  - 함수의 인자로 errno에 저장된 값을 받아 오류 메시지를 리턴
  - 리턴된 오류 메시지를 사용자가 적절하게 가공할 수 있다는 장점

04. 시스템 도구

■ [예제 1-5] strerror() 함수로 오류 메시지 출력하기

```
01 #include <stdio.h>
02 #include <unistd.h>
03 #include <errno.h>
04 #include <stdlib.h>
05 #include <string.h>
06
07 extern int errno;
08
09 int main() {
10       char *err;
11
12       if(access("test.txt", R_OK) == -1) {
13            err = strerror(errno);
14            printf("오류: %s(test.txt)₩n", err);
15            exit(1);
16       }
17 }
```

실행

```
$ ch1_5.out
오류: No such file or directory(test.txt)
```

- **3행** strerror() 함수는 인자로 받은 errno 변수에 저장된 오류 번호에 따라 오류 메시지가 저장된 문자열을 가리키는 포인터를 리턴
- **14행** 13행에서 리턴한 문자열을 적절한 형태로 가공해 오류 메시지를 출력
- **실행 결과** 14행에서 지정한 대로 출력

04. 시스템 도구

■ 메모리 할당 : malloc(3)

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

void *malloc(size_t size);
```

- size : 할당받을 메모리 크기

- malloc() 함수의 특징
  - 인자로 지정한 크기의 메모리를 할당하는 데 성공하면 메모리의 시작 주소를 리턴
  - 만약 메모리 할당에 실패하면 NULL 포인터를 리턴
  - 인자로 지정하는 메모리 크기는 바이트 단위
  - 할당된 메모리에는 어떤 형태의 데이터도 저장할 수 있음
  - malloc() 함수는 할당된 메모리를 초기화하지 않는다는 데 주의

  - 요소가 10개이고 각 요소의 크기가 20바이트인 배열을 저장할 수 있는 메모리를 할당 하는 예

```
char *ptr
ptr = calloc(10, 20);
```

04. 시스템 도구

■ 메모리 할당 : calloc(3)

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

void *calloc(size_t nmemb, size_t size);
```

- nmemb : 배열 요소의 개수
- size : 할당받을 메모리 크기

- calloc() 함수의 특징
  - calloc() 함수는 nmemb×size바이트 크기의 배열을 저장할 메모리를 할당
  - calloc() 함수는 할당된 메모리를 0으로 초기화

  - 요소가 10개이고 각 요소의 크기가 20바이트인 배열을 저장할 수 있는 메모리를 할당 하는 예

```
char *ptr
ptr = calloc(10, 20);
```

04. 시스템 도구

■ 메모리 추가 할당 : ralloc(3)

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

void *realloc(void *ptr, size_t size);
```

- ptr : 할당받은 메모리를 가리키는 포인터
- size : 할당받을 메모리 크기

- ralloc() 함수의 특징
  - realloc() 함수는 이미 할당받은 메모리에 추가로 메모리를 할당할 때 사용
  - 이전에 할당받은 메모리와 추가할 메모리를 합한 크기의 메모리를 새롭게 할당하고 주소를 리턴
  - 이때 이전 메모리의 내용을 새로 할당된 메모리로 복사

  - malloc() 함수로 할당받은 메모리에 추가로 100바이트를 할당하는 예

```
char *ptr, *new;
ptr = malloc(sizeof(char) * 100);
new = realloc(ptr, 100);
```

04. 시스템 도구

■ 메모리 해제 : free(3)

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

void free(void *ptr);
```

- ptr : 해제할 메모리 주소

- free() 함수의 특징
  - free() 함수는 사용을 마친 메모리를 해제하고 반납
  - . free() 함수가 성공하면 ptr이 가리키던 메모리는 더 이상 의미가 없음

04. 시스템 도구

■ 명령행 인자

- 명령행
  - 리눅스 시스템에서 사용자가 명령을 입력하는 행
  - 프롬프트가 나타나고 커서가 사용자 입력을 기다리고 있는 행

- 명령행 인자 (CLA)
  - 사용자가 명령행에서 명령을 실행할 때 해당 명령(실행 파일명)과 함께 지정하는 인자
  - 명령행 인자는 명령의 옵션, 옵션의 인자, 명령의 인자로 구성

- 명령행 인자의 전달
  - 보통 main() 함수는 다음과 같이 정의

```
int main() {...}
(또는)
int main(void) {...}
```

  - main() 함수에서 명령행 인자를 전달받으려면 다음과 같이 정의

```
int main(int argc, char *argv[]) {...}
```

04. 시스템 도구

■ [예제 1-6] 명령행 인자 출력하기

```
01 #include <stdio.h>
02
03 int main(int argc, char *argv[]) {
04        int n;
05
06        printf("argc = %d₩n", argc);
07        for (n = 0; n < argc; n++)
08              printf("argv[%d] = %s₩n", n, argv[n]);
09 }
```

실행

```
$ ch1_6.out -h 2000
argc = 3
argv[0] = ch1_6.out
argv[1] = -h
argv[2] = 2000
```

- **03행** 명령행 인자를 받기 위해 main() 함수에 argc와 argv를 선언한다.
- **06행** 인자의 개수를 저장한 argc 값을 출력한다.
- **07~08행** 각 인자를 담은 argv의 내용을 출력한다.
- **실행 결과** 명령행에서 실행 파일명인 ch1_6.out 외에 -h와 2000을 인자로 입력
  따라서 main() 함수에 전달된 총 개수를 나타내는 argc 값은 3
  argv[0]에 실행 파일명이 저장되고, 차례로 인자가 저장됨을 알 수 있음
  argv로 전달되는 값은 문자열이므로 printf() 함수로 출력하려면 형식 지정자 %s를 사용해야 함

04. 시스템 도구

■ 옵션 처리 : getopt(3)

- 명령행 인자로 전달된 옵션을 편리하 게 처리하도록 getopt() 함수가 제공
- 리눅스에서는 표준에 따라 관련 헤더 파일이 다를수 있음
- 리눅스는 POSIX를 따라 unistd.h를 선언

SVID3, XPG3

```
#include <stdlib.h>

int getopt(int argc, char * const argv[], const char *optstring);
extern char *optarg;
extern int optind, opterr, optopt;
```

POSIX.2, XPG4, SUS, SUSv2, SUSv3

```
#include <unistd.h>

int getopt(int argc, char * const argv[], const char *optstring);
extern char *optarg;
extern int optind, opterr, optopt;
```

04. 시스템 도구

■ 리눅스 명령 기본 규칙

▪ POSIX와 솔라리스의 규칙

- POSIX에서 정의한 명령에 대한 기본 규칙은 14개
- 솔라리스의 경우 POSIX의 규칙 1~13과 자체적으로 확장한 규칙 8개(14~21)로 총 21개 항목

• getopt() 함수 관련 항목 (POSIX)

- [규칙 3] 옵션의 이름은 한 글자여야 한다.
- [규칙 4] 모든 옵션의 앞에는 하이픈(-)이 있어야 한다.
- [규칙 5] 인자가 없는 옵션은 하나의 - 다음에 묶여서 올 수 있다
- [규칙 6] 옵션의 첫 번째 인자는 공백이나 탭으로 띄고 입력해야 한다
- [규칙 7] 인자가 있어야 하는 옵션에서 인자를 생략할 수 없다.
- [규칙 9] 명령행에서 모든 옵션은 명령의 인자보다 앞에 와야 한다.
- [규칙 10] 옵션의 끝을 나타내기 위해 --을 사용할 수 있다

04. 시스템 도구

■ 리눅스 명령 기본 규칙

▪ POSIX와 솔라리스의 규칙

- getopt() 함수 관련 항목 (솔라리스)

명령어 –a --긴 옵션1 –c 옵션 인자 –f 옵션 인자 --긴 옵션2=옵션 인자
--긴 옵션3 옵션 인자 파일명

- [규칙 15] 긴 옵션은 -- 다음에 와야 한다. 옵션명으로는 문자, 숫자, -만 사용할 수 있으며, -으로 연결한 1~3개 단어를 사용할 수도 있다.
- [규칙 16] '--이름=인자' 형태는 긴 옵션 사용에서 옵션의 인자를 상세하게 지정할 때 사용해야 한다 (예에서 긴 옵션2의 경우). '--이름 인자' 형태도 가능하다 (예에서 긴 옵션3의 경우)
- [규칙 17] 모든 명령은 긴 옵션 --version(-V도 지원)과 --help(-?도 지원)를 표준으로 지원해야 한다.
- [규칙 18] 모든 짧은 옵션에 대응하는 긴 옵션이 있어야 하고, 긴 옵션에도 대응하는 짧은 옵션이 있어야 한다.

04. 시스템 도구

■ [예제 1-7] getopt() 함수로 옵션 처리하기

```c
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    int n;
    extern char *optarg;
    extern int optind;

    printf("Current Optind : %d₩n", optind);
    while ((n = getopt(argc, argv, "abc:")) != -1) {
        switch (n) {
            case 'a':
                printf("Option : a₩n");
                break;
            case 'b':
                printf("Option : b₩n");
                break;
            case 'c':
                printf("Option : c, Argument=%s₩n", optarg);
                break;
        }
        printf("Next Optind : %d₩n", optind);
    }
}
```

실행

```
$ ch1_7.out
Current Optind : 1
$ ch1_7.out -a
Current Optind : 1
Option : a
Next Optind : 2
$ ch1_7.out -c
Current Optind : 1
ch1_7.out : option requires an argument -- 'c'
Next Optind : 2

$ ch1_7.out -c name
Current Optind : 1
Option : c, Argument=name
Next Optind : 3
$ ch1_7.out -x
Current Optind : 1
ch1_7.out : invalid option -- 'x'
Next Optind : 2
```

04. 시스템 도구

■ [예제 1-7] getopt() 함수로 옵션 처리하기

```
01 #include <stdio.h>
02 #include <unistd.h>
03
04 int main(int argc, char *argv[]) {
05       int n;
06       extern char *optarg;
07       extern int optind;
08
09       printf("Current Optind : %d₩n", optind);
10       while ((n = getopt(argc, argv, "abc:")) != -1) {
11          switch (n) {
12               case 'a':
13                    printf("Option : a₩n");
14                    break;
15               case 'b':
16                    printf("Option : b₩n");
17                    break;
18               case 'c':
19                    printf("Option : c, Argument=%s₩n", optarg);
20                    break;
21          }
22          printf("Next Optind : %d₩n", optind);
23       }
24 }
```

- **09행** 외부 변수 optind에 저장된 값을 출력
- **10행** getopt() 함수로 인자가 있는지 확인하고 옵션을 읽어옴
- **11~21행** switch 문을 이용해 옵션별로 출력
- **19행** -c 옵션의 인자가 저장된 외부 변수 optarg의 값을 출력
- **실행 결과** 실행 시 옵션을 지정하지 않으면 특별한 작업을 하지 않음
- -a와 같은 정상적인 옵션을 지정하면 이를 인식
- 옵션 인자가 필요한 -c 옵션에 인자를 지정하지 않으면
  인자가 필요하다는 오류 메시지가 출력되며 인자를 지정하면
  해당 인자가 인식
- -x와 같이 잘못된 옵션을 지정하면 잘못된 옵션이라는 오류 메시지가 출력

시스템 프로그래밍

리눅스&유닉스

감사합니다.