R E P O R T

1. 리눅스 명령어, 쉘 및 프로그래밍

과목명	운영체제
담당교수	김성우 교수님
학과	컴퓨터소프트웨어공학과
학번	20212979
이름	임진호

동의대학교
DONG-EUI UNIVERSITY

목 차

1. 리눅스의 이해

운영체제의 개념

컴퓨터 구성: 컴퓨터는 크게 하드웨어(물리적 장치)와 소프트웨어(프로그램)로 나뉜다.

소프트웨어 분류: 소프트웨어는 시스템 소프트웨어(운영체제 등)와 응용 소프트웨어(워드, 게임 등)로 구분된다.

운영체제란?: 운영체제(OS)는 컴퓨터를 작동시키고, 자원을 관리하며, 사용자의 응용 프로그램이 효율적으로 실행될 수 있도록 환경을 제공하는 기본 소프트웨어다.

부트스트랩 프로그램: 컴퓨터가 켜질 때 가장 먼저 적재되는 프로그램으로, 운영체제가 메모리에 올라가게 한다.

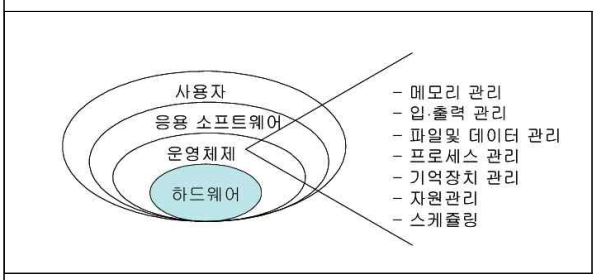
운영체제의 역할: 이후 컴퓨터 내의 모든 프로그램과 자원을 관리한다.

API와 서비스 요청: 응용 프로그램들은 정의된 응용프로그램 인터페이스(API)를 통해 운영체제에 서비스를 요청한다.

사용자 인터페이스: 사용자는 명령어(명령행 인터페이스)나 그래픽 환경을 통해 운영체제와 직접 대화할 수 있다.

운영체제의 구조

컴퓨터를 기동할 때 먼저 올려지는 프로그램이며, 핵심부는 주기억 영역에 상주



운영체제의 기능

1. 메모리 관리 (Memory Management)

주기억장치(RAM)의 사용을 제어하여 각 응용 프로그램이 사용할 메모리 영역을 효율적으로 배분한다.

→ 역할: 프로그램이 실행될 메모리 위치 결정, 메모리 할당 및 회수 관리

2. 입·출력 관리 (Input/Output Management)

컴퓨터 내부와 외부 장치 간의 데이터 흐름을 제어한다.

→ 역할: 프로그램과 디스크, 모니터, 프린터 등의 주변기기 간 데이터 교환 지원

3. 파일 및 데이터 관리 (File and Data Management)

파일의 생성, 삭제, 열기, 닫기 등의 기능을 수행하여 데이터를 체계적으로 보관하고 관리한다.

→ 역할: 파일 시스템 유지, 접근 제어 및 저장 공간 관리

4. 프로세스 관리 (Process Management)

실행 중인 프로그램(프로세스)의 생성, 제거, 실행 상태 등을 관리한다.

→ 역할: 프로세스 간 통신, CPU 할당, 프로세스 시작과 중지 제어

5. 기억장치 관리 (Storage Management)

보조기억장치 등 여러 프로그램이 공통으로 사용하는 저장 공간의 사용 상태를 관리한다.

→ 역할: 사용자 요청에 따라 저장 공간 할당 및 회수

6. 자원 관리 (Resource Management)

프린터, 드라이브, 모뎀 등 시스템 자원을 효율적으로 배분하고 충돌 없이 사용할 수 있도록 조정한다.

→ 역할: 자원 할당, 접근 제어, 시스템 성능 최적화

7. 스케줄링 (Scheduling)

동시에 여러 작업이 요청될 때, 작업의 실행 순서를 결정하여 시스템을 효율적으로 운용한다.

→ 역할: CPU 스케줄링, 우선순위 결정, 다중 작업 환경에서의 효율성 보장

운영체제가 가지는 목적

1. 처리능력 향상
2. 응답시간단축
3. 사용가능성향상
4. 신뢰도향상

구분

제어 프로그램과 처리 프로그램으로 구성,

제어프로그램은 감시프로그램, 작업관리 프로그램, 데이터관리 프로그램 3가지로 구성된다.

처리 프로그램은 언어번역 프로그램,서비스 프로그램으로 구성

Lunux

1980년대 리눅스는 유닉스를 PC 버전으로 개발하려 노력하였고, 1991년 헬싱키 대학생 ‘리누스 토발즈’에 의해 최초 리눅스 커널 개발되었다.

리눅스의 정의

리눅스(Linux)는 UNIX와 유사한 구조와 철학을 가진 오픈소스 운영체제로,

프로세스 스케줄링, 가상 메모리, 파일 관리, 장치 입·출력 등의 기본 서비스를 제공하는 운영체제 커널(kernel)이다.

현재는 임베디드 시스템, 개인용 PC, 서버, 고성능 워크스테이션 등 다양한 환경에서 사용되고 있으며, 인텔 호환 컴퓨터뿐만 아니라 맥킨토시, SUN, DEC, IBM 등의 시스템에서도 동작한다.

리눅스는

- 다중처리(Multiprocessing),
- 가상 메모리(Virtual Memory),
- 공유 라이브러리(SharedLibrary),
- 요구 시 메모리 적재(Demand Paging),
- TCP/IP 기반의 강력한 네트워킹 기능

등을 지원하여 UNIX에 상응하는 강력한 운영체제로 평가받고 있다.

리눅스의 철학 - 오픈소스 정신

-> 리눅스는 오픈소스(Open Source) 소프트웨어로, 누구나 소스코드를 자유롭게 열람·수정·배포할 수 있다.

-> 이 개념은 1980년대 초 리처드 스톨만(Richard Stallman)이 설립한 자유소프트웨어재단(FSF)의 GNU 프로젝트에서 시작되었다.

※ 리눅스는 GNU GPL(General Public License) 을 따르며

소스코드가 공개되어 있어 개발자 누구나 수정 및 개선 가능
단순한 '무료(Free)'가 아닌 **'자유(Freedom)'**의 의미를 강조
상업용 소프트웨어에 포함하거나 판매용으로 사용하는 것은 제한된다

GNU 소프트웨어 프로젝트

GNU 프로젝트는 1984년 시작되어, 누구나 자유롭게 사용할 수 있는 UNIX 호환 자유 소프트웨어를 만드는 것을 목표로 했다.

이 프로젝트의 핵심 철학은 "자유롭게 사용·수정·배포할 수 있는 소프트웨어"로, 리눅스는 이러한 GNU 철학을 기반으로 발전해 왔다.

즉, 가장 대표적인 오픈소스 OS이다.

오픈소스 라이센스

대표적으로 GPL, LGPL, MPL, BSD, Apache, MIT 라이선스가 있으며,
GPL처럼 소스 공개를 의무화한 강한 라이선스부터
BSD나 MIT처럼 공개 의무가 없는 자유로운 형태까지 다양하다.

약자	이름	특징
GPL	GeneralPublicLicense	소스코드를 배포하는 경우"GPL에의해 배포된다"고 명시하고, 소스코드 수정시에 반드시 공개 해야함
LGPL	LesserGPL	일부 라이브러리에 대해 소스코드 공개정도 완화
MPL	MozillaPublicLicense	소스코드 수정시에 공개해야 하지만, 다른코드와 결합 되는 경우 MPL코드 제외한 코드는 공개 의무 없음
BSD	BerkeleySoftwareDistribution	소스코드 공개 의무없음
AL	ApacheLicense	소스코드 공개 의무없음
MIT License	MITLicense	소스코드 공개 의무없음

리눅스 장점

멀티태스킹 및 멀티유저 지원	여러 사용자가 동시에 다양한 작업을 수행할 수 있음
강력한 네트워킹 기능	다양한 프로토콜(TCP/IP 등)을 지원해 서버 구축과 네트워크 관리에 유리함
유닉스 표준화 호환	POSIX 표준을 따르며 기존 유닉스 소프트웨어와 높은 호환성 유지
편리한 사용환경	텍스트 기반 셸과 GNOME·KDE 같은 그래픽 환경 모두 사용 가능
멀티플랫폼 지원	인텔, ARM 등 다양한 하드웨어에서 동작하며 멀티부팅 가능
다양한 파일시스템 지원	ext2/3, FAT, NTFS, NFS 등 여러 파일 시스템을 인식하고 사용 가능
다양한 프로세스,스레드 통신지원	파이프, 메시지 큐, 세마포어 등 여러 통신 방식 제공
효율적인 하드웨어 자원관리	가상 메모리와 페이징 기술로 자원 활용을 극대화

리눅스 단점

-> 공개 운영체제이기 때문에 문제점 발생시 보상받을 수 없음

-> 보안에 취약할 것이라는 선입관

M$ 윈도우와 리눅스의 차이점

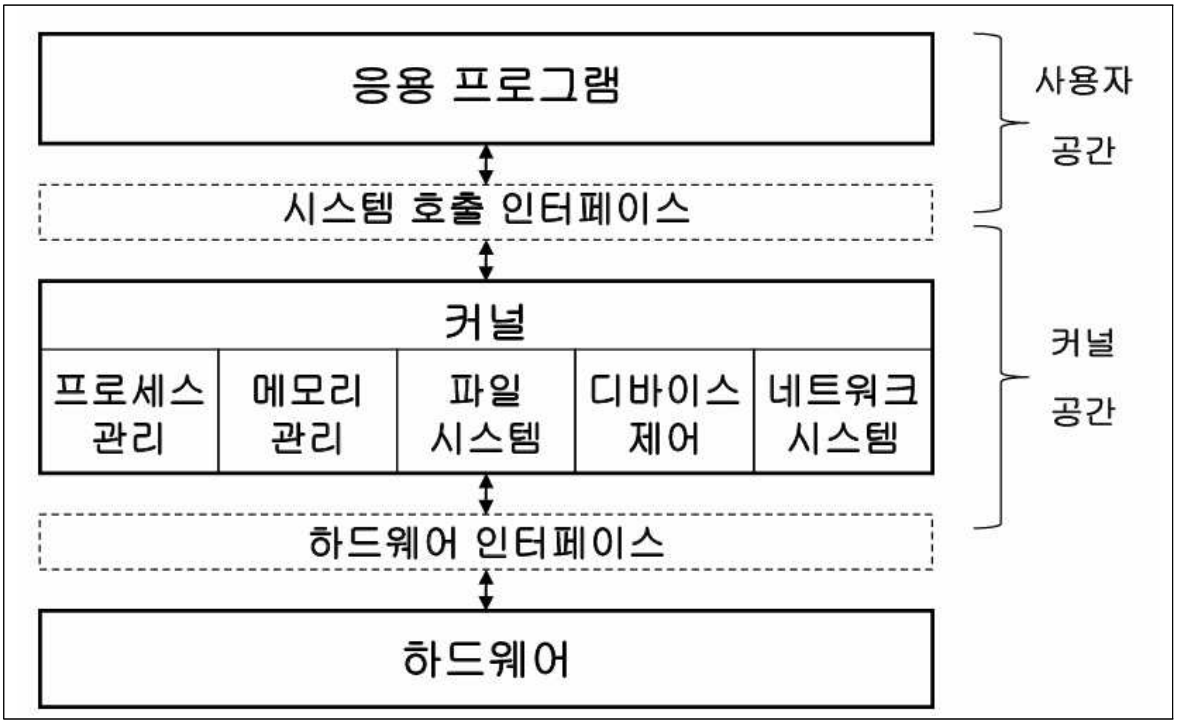
분야	특징
시스템 환경	리눅스는 저 사양 시스템에서도 효율적으로 작동한다.
멀티부팅	리눅스가 제공하는 GRUB 등은 매우 다양한 멀티 부팅을 제공한다.
PnP 기능	리눅스는 완벽한 PnP 기능이 제공되고 있지 않으며 별도의 하드웨어 지식을 필요로 한다.
한글지원	윈도우는 한를 IME를 운영체제에서 지원한다.
파티션 설정과 파일 시스템	윈도우는 NTFS,FAT 등 제한적이나 리눅스는 모든 다른 파일 시스템을 지원한다.
GUI	윈도우 GUI는 사용자들이 매우 쉽게 사용할 수 있으며, 리눅스는 X 윈도우를 바탕으로 보다 강력한 기능을 제공한다.
각종 드라이버지원	윈도우는 비디오, 네트워크, 사운드에 대한 지원율이 높고 리눅스는 지원되지 않는 드라이버도 있다.
응용프로그램	대부분의 상용 응용 프로그램은 윈도우 기반으로 작동되며 현재에는 리눅스용 응용프로그램이 많이 등장하고 있다. 또한 윈도우용 제품을 리눅스에서 실행시킬 수 있는 유틸리티도 있다.
네트워크 및 서버	리눅스 다양한 서버와 네트워킹 지원이 강력하다
개발도구	리눅스는 GNU의 강력한 컴파일러인 gcc를 무료로 사용할 수 있고 윈도우는 MSDN을 통해 개발자들을 지원하고 있다.
기술/서비스지원	리눅스는 윈도우에 비해 신속한 서비스가 미흡한 실정이다. 그러나 상용 배포판을 구한 사용자는 신속한 서비스가 가능하다.
가격	리눅스는 GPL에 의거 무료로 사용할 수 있다.

리눅스 내부 구조

리눅스 커널

● 응용 프로그램과 시스템 호출 인터페이스를 통해 서비스 제공

● 하드웨어 계층과 인터럽트 등을 통해 하드웨어 자원관리



※ 이 그림은 운영체제의 구조를 나타내며,
커널이 하드웨어와 응용 프로그램 사이에서 프로세스·메모리·파일·입출력 장치·네트워크 등을 관리하는 역할을 수행함을 보여준다.

리눅스의 미래

-> 서버시장 중심에서 데스크탑, 노트북,PDA 시장으로 빠르게 확산

-> 소규모 업체들은 전략적 제휴와 합병을 통해 시장 경쟁력을 높이고 있다

리눅스의 적용 분야	임베디드 및 실시간 시스템(Android), 데스크톱 PC, 서버 등 다양한 하드웨어 환경에서 사용 가능하다.
리눅스 보안 운영체제	오픈소스의 장점을 유지하면서도 보안성을 강화한 시스템으로 발전 중이다.
리눅스 표준화	다양한 배포판 간의 호환성을 높이고, 국제 표준을 따르는 방향으로 개선되고 있다.
클러스터링 및 슈퍼컴퓨터	여러 대의 시스템을 연결해 고성능 연산을 수행하는 클러스터 환경에서 리눅스가 주도적인 역할을 하고 있다.
클라우드 컴퓨팅	안정성과 확장성이 뛰어나 클라우드 서버 운영체제로 널리 사용되고 있으며, 미래 핵심 인프라 기술로 자리 잡고 있다.

2. 리눅스 사용 환경

리눅스 설치 (WSL2)

WSL 설치	wsl--install
WSL 업데이트	wsl--update

-> 파워쉘을 종료하고 재시작

관리자 권한으로 실행 후
WSL2 버전으로설정 -> $ wsl—set-default-version 2

Microsoft Store 에서 “Ubuntu” 검색하여설치
실행후 사용자계정 및 패스워드 생성 및 쉘 구동

파일편집기

nano 편집기	메모장과 비슷
vim 편집기	명령어가 복잡하지만 익숙해지면 편함
vscode	가장 인기있는 편집기

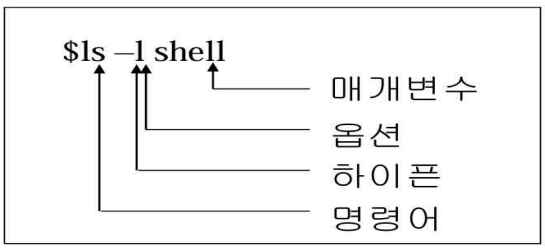
Gedit 편집기 - 별도의 그래픽 창
설치

```
$ sudo apt update
$ sudo apt install gedit
```

실행 --->> $ gedit
(sudo)는 관리자 권한으로 접근을 의미.

3. 리눅스 명령어

명령어 구조



1. 명령어 종류
a. 비대화식 명령어
-> 해당 명령어만 입력하여 원하는 결과 표시
b. 대화식 명령어
-> 명령어들이 실행하는 동안에 어떤 것을 질문하고 사용자가 대답하고 하는 일련의 과정이 끝나면 결과로 보여줌

코드	키	의미
init	^C	실행중인 프로그램의 중단
erase	^H	마지막 문자의 삭제
werase	^W	마지막 한 문자의 삭제
kill	^U	한 줄 전체의 삭제
quit	^₩	프로그램을 중단시키고 코어(core)파일에 저장
stop	^S	화면표시의 정지
start	^Q	화면표시의 재개
eof	^D	더 이상 자료 없음을 표시

->> 셸에서 사용하는 특수 문자들

기본 명령어들

ls (디렉토리파일보기)

일반형식	ls[옵션]
주요옵션	-a, --all : 디렉토리내의모든파일을출력 -l, --format=long : 파일종류, 사용권한, 크기등출력 -s, --size : 1K 단위로파일크기를표시 -t, --sort=time : 최근에만들어진파일부터출력 -c, --time : 최근에변경한파일부터출력 -R : 하위디렉토리까지출력ㅇ

```
리눅스@사용자 $ ls
 Desktop Mail bashrc book2 package
```

pwd (현재 작업중인 디렉토리 확인)

일반형식	pwd

```
리눅스@사용자 $  pwd
 /home/linux
```

cd 디렉토리 변경

일반형식	cd [directory]

```
리눅스@사용자 $cd /usr
리눅스@사용자 $pwd
 /usr
```

touch (파일만들기)

일반형식	touch [-acm] [-r ref_file | -t time] file ... touch [-acm] [date_time] file ...
주요옵션	-a, --all : 디렉토리내의모든파일을출력 -l, --format=long : 파일종류, 사용권한, 크기등출력 -s, --size : 1K 단위로파일크기를표시 -t, --sort=time : 최근에만들어진파일부터출력 -c, --time : 최근에변경한파일부터출력 -R : 하위디렉토리까지출력ㅇ

```
리눅스@사용자 $ touch out
리눅스@사용자 $ ls
-rw-rw-r— 1 리눅스사용자 0  9월 3 19:24 out
```

cat (출력 및 파일만들기)

일반형식	cat 파일명 : 파일 내용 표준 출력 cat > 파일명 : 화면에 출력하고 파일에 저장

```
리눅스@사용자 $ cat > catTest.txt
 Hello!
 Nice to meet you. ← Ctrl-D 입력
리눅스@사용자 $ ls
 Test.c Test2 catTest.txt  mvTest.c
```

cp (파일 복사)	
일반형식	cp [-fip] source dest cp [-fipr] source.. dest_dir
주요옵션	-f : 복사할파일이있을경우삭제하고복사 -i : 복사할파일이있을경우복사할것인지물어봄 -p : 원본파일의모든정보를보존한채복사 -r : 하위디렉토리에있는모든파일을복
리눅스@사용자 $ cp /bin/date Test 리눅스@사용자]$ cd Test 리눅스@사용자 $ ls Test.c date	

rm (파일의 삭제)	
일반형식	rm [-firv] source dest
주요옵션	울파일이있을경우강제로삭제 -i : 지울 파일이있을경우지울 것인지물어봄 -r : 하위 디렉토리에있는모든파일을삭제 -v : 지우는 파일정보를출력

mv (파일 이름 변경와 옮기기)	
일반형식	mv [-fi] source dest mv [-fi] source … dest_dir
주요옵션	-b : 대상파일이지워지기전에백업파일을만듦 -f : 대상파일의접근허가와관계없이무조건파일을이동 -i : 대상파일이기존파일이면, 덮어쓸것인지물어봄 -u : 대상파일보다원본파일이최근의것일때업그레이드 -v : 파일옮기는과정을자세하게보여준다

mkdir (디렉토리 생성)	
일반형식	mkdir [-m mode] [-p] dir …
주요옵션	-m : 새로운디렉토리의허가모드를지정한모드로설정 -p : 하위 디렉토리가존재하지않는경우함께생성

rmdir (디렉토리 삭제)	
일반형식	rmdir [-p] dir …
주요옵션	-p : 지정한하위디렉토리까지삭제

man (온라인 매뉴얼 출력)	
일반형식	man [section] name
주요옵션	section -아래구분에서매뉴얼을찾아보여준다
1절-Commands (명령어) 2절-System Calls (시스템호출) 3절-Subroutines (라이브러리함수) 4절-Special files (특수 파일) 5절-File formats and conventions (파일 형식) 6절-Games (게임)	

4. 정규표현식

구분	표현식	내용
기본 정규 표현식 (Basic Regular Expressions)	.	점의 개수 만큼 아무 문자나 대체 (replaces any character)
	^	문자열의 처음 시작 부분 매칭 (matches start of string)
	$	문자열의 끝 부분 매칭 (matches end of string)
	*	* 앞의 문자와 매칭 (matches up zero or more times the preceding character)
	₩	특수 문자와 매칭 (Represent special characters)
	( )	정규 표현식 그룹 (Groups regular expressions)
	?	정확히 한개의 문자와 매칭 (Matches up exactly one character)
간격 정규 표현식 (Interval Regular Expressions)	{n}	앞의 문자와 'n'번 매칭 (Matches the preceding character appearing 'n' times exactly)
	{n, m}	앞의 문자와 'n'번 매칭하되 'm'번 이하로 매칭 (Matches the preceding character appearing 'n' times but not more than m)
	{n, }	앞의 문자와 'n'번 이상 매칭 (Matches the preceding character only when it appears 'n' times or more)
확장 정규 표현식 (Extended Regular Expressions)	₩+	₩+ 앞의 문자가 한번 이상 출현한 문자열과 매칭 (Matches one or more occurrence of the previous character)
	₩?	₩? 앞의 문자가 1번 이하로 출현한 문자열과 매칭 Matches zero or one occurrence of the previous character

-> 정규표현식은 텍스트에서 패턴으로 검색·추출하기 위한 표현 방식이다.

.는 임의의 한 글자와 일치하고, ^는 문자열 시작, $는 문자열 끝을 의미한다. 메타문자 자체를 찾으려면 앞에 \(백슬래시)를 붙여 이스케이프한다(예: \.). 문자 집합/범위는 대괄호로 표기한다(예: [A-Za-z0-9], [0-9]-[0-9]).

또한 grep은 정규표현식으로 필터링할 때 자주 쓰이며, 파이프라인(|)과 함께 다른 명령의 출력에서 원하는 정보만 추출하는 데 활용한다.

예시: dmesg | grep '^usb', ps aux | grep '[n]ginx', grep -E '^[A-Z].*\.log$' files.txt

정규표현식 예시 10

[A-Za-z_] [A-Za-z0-9_]*	c언어,java에서의 변수명에 해당하는 정규표현식, 첫 번째 오는 문자가 영어알파벳만 가능하고 뒤에는 알파벳과 숫자만 가능하고 길이는 무한이란 의미이다.
\.log$	.log로 끝나는 줄 (마침표 이스케이프)
^[A-Z]{3}$	대문자 3글자만으로 된 줄
^010-[0-9]{4} -[0-9]{4}$	휴대폰 형식 010-1234-5678
^[0-9]{4}-[01] [0-9]-[0-3][0-9]$	날짜(YYYY-MM-DD) 대략 검사
^([0-9]{1,3}\.){3} [0-9]{1,3}$	IPv4 주소 단순 형식
^[A-Za-z0-9._%+-]+ @[A-Za-z0-9.-] +\.[A-Za-z]{2,}$	이메일 대략 검사
https?://	http 또는 https로 시작하는 URL
^[^#]	#로 시작하지 않는 줄(주석 제외)

^(dev|staging|prod)$	dev/staging/prod 중 하나만 일치

5. 파일 편집기

문서 편집기

Text 형식의문서를새로만들고수정하고하는일련의작
업을하는데쓰는유틸리티프로그램

대화형 유틸리티	
라인 에디터	스크린
-> ed, ex	-> vi, emacs, nano
모든 에디터는 현재 수정한 값들이 원본 파일에 그대로 저장되지 않고 임시공간인 버퍼(buffer)에 저장	

● 명령어 다음에 파일명을 입력

-> ed [파일명]

입력 모드 → 출력 모드(Ctrl-C)

출력 모드 → 입력 모드(a (append), I (insert), c (change))

w 명령어 - 현재 버퍼에 있는 모든 내용을 저장

q 명령어 - 편집 작업 종료

기본 명령어

명령	의미	명령	의미
a	문자열 추가(append)	c	지정 행의 문자열 치환(change)
i	지정 행에 문자열 삽입(insert)	d	지정 행 삭제(delete)
s	문자열 치환(substitute)	p	현재 내용 출력(print)
e	다른 파일 읽어들임(enter)	+	아래로 행 이동
r	다른 파일 내용 삽입	-	위로 행 이동
w	저장	/	아래로 문자열 검색
l	출력할 수 없는 문자 표시(list)	?string?	위로 문자열 검색
k	특정 행을 알파벳으로 지정	m	특정 행 이동(move)
=	지정 행 번호 출력	u	이전 명령 취소(undo)

스트림 편집기

1. 파일이나 표준 입력으로부터 버퍼로 읽어 각 행마다 편집 명령을 실행한 후 그 결과를 다시 표준 출력으로 표시하는 에디터
2. "sed" 에디터의 가장 큰 특징은 원본파일을 변경하지 않음
3. 반복작업이나 아주 큰 파일을 처리할 때 매우 유용

sed	
일반형식	sed [options] filename
주요옵션	-f : 스크립트파일을불러옴 -e : 여러개의스크립트를하나의명령행에지정 -n : 명령어지정행을제외하고표준출력하지않음

기본 명령어

명령	의미	명령	의미
s	문자열 치환	c	지정 행의 문자열 치환
a	맨 마지막에 문자열 추가	p	출력
i	지정 행에 문장 삽입	w	저장

Nano 편집기

초보자용 문서 편집기

-> vi와는 달리 Windows의 메모장과 비슷하게 쉽게 사용

-> 사용 명령 => $nano[파일이름]

명 령	내 용
CTRL+g	도움말 보기
CTRL+o	파일 저장
CTRL+x	Nano 빠져나오기
CTRL+a	현재 행의 처음으로 이동
CTRL+e	현재 행의 처음으로 이동
CTRL+v	이전 페이지로 이동(page-up)
CTRL+y	다음 페이지로 이동(page-down)
CTRL+w	문자열 찾기
CTRL+d	현재 커서 위치의 한 글자 삭제
CTRL+k	한 줄 삭제
CTRL+u	마지막으로 삭제된 줄 복구

Vi 편집기

Emacs와 함께 리눅스 환경에서 가장 많이 쓰이는 편집기

설치 => $ sudo apt install vim

vi 또는 vim (시작)	
일반형식	vi [filename …]
주요옵션	filename -파일명은여러개, 와일드카드(“,?) 가능

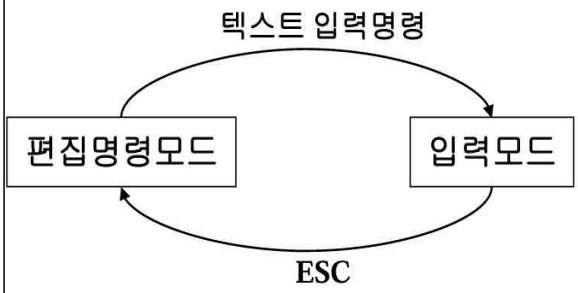
=> 읽기만 할 때는 vi 대신 view 사용

:q!	파일 내용을 저장하지 않고 종료
:wq! 또는 zz	파일 내용을 저장하고 종료

입력 모드와 편집 명령모드

1. 입력모드
-> 파일의 내용을 입력할 수 있는 모드
-> 입력 모드 전환 : I,a,o
-> 사용자가 키보드를 치면 그 내용이 화면에 나타남

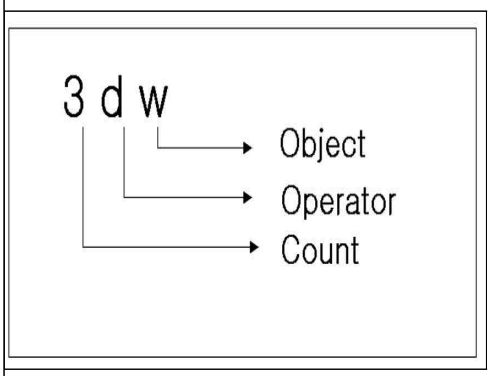
2. 편집명령모드
-> 편집명령을수행하는모드
-> 편집모드전환: ESC 키
-> 사용자의키보드입력이화면에나타나지않고 명령수행



명 령	내 용
a	커서 바로 뒤에서 입력을 시작
A	커서가 위치한 행의 마지막부터 입력을 시작
i	커서가 위치한 곳부터 입력을 시작
I	커서가 위치한 행의 맨 앞에서 입력을 시작
o	커서가 위치한 행의 아래 행부터 입력을 시작
O	커서가 위치한 행의 위부터 입력을 시작

vi 편집기의 두 가지 모드 전환 관계를 나타내며,
텍스트 입력 명령으로 입력 모드로 들어가고, ESC 키를 눌러 편집 명령 모드로 돌아가는 과정을 보여준다.

vi 편집기의 명령 구조



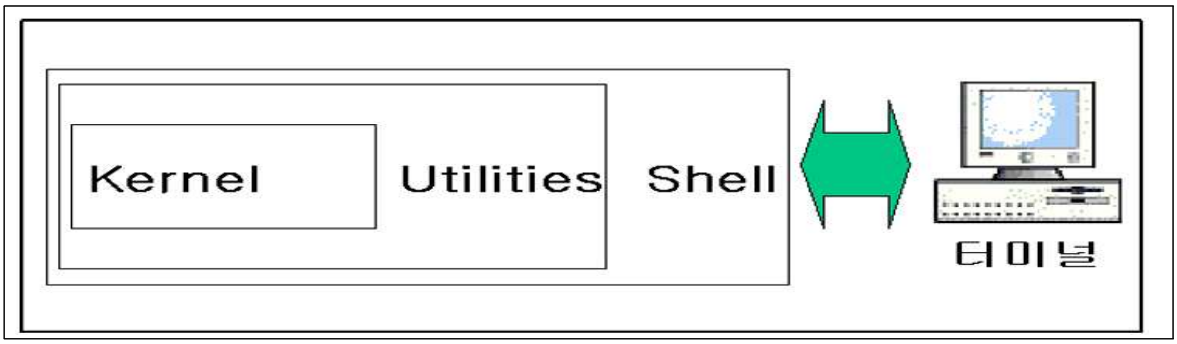
vi명령어 사용

-> ":"(콜론)을사용
-> 파일전체나행단위로파일을제어할때사용
-> 라인번호를이용하여여러라인을복사하거나이동할때유용
-> 정규표현식사용가능

6. 쉘 환경

쉘

명령어 해석기 - 사용자가 입력한 명령어를 해석해서 주는 프로그램으로
커널 주위를 둘러싸며 사용자와의 인터페이스 담당



사용자와 운영체제 사이의 인터페이스 역할이다.

Bourne Shell	가장오래된유닉스쉘
C Shell	C 언어와유사한 문법사용
Korn Shell	특히명령어기억, 별명(Alias)기능, 제어기능 등을가지고있음
Bash	리눅스의기본쉘로서많이사용

사용자 환경설정

환경변수

-> 사용자가 각자 원하는 형태로 자신의 컴퓨터 사용 환경을 설정해 두기 위한 변수들

환경변수	설명		
USER	로그인한 사용자 이름	EDITOR	기본 편집기
LOGNAME	프로세스와 관련된 로그인 사용자 이름	PS1	프롬프트
HOME	사용자의 홈 디렉토리	TERM	터미널 유형
LANG	LC_ALL 등이 지정되지 않았을 때 로케일 이름	HISTFILE	히스토리 파일
LC_ALL	우선적으로 지정되는 로케일 이름. LC_COLLATE, LC_CTYPE, LC_MESSAGES, LC_MONETARY, LC_NUMERIC, LC_TIME 등을 포함	HISTFILESIZE	히스토리 파일 크기
PATH	실행 파일을 탐색하는 경로/디렉토리 목록	HISTSIZE	히스토리에 저장된 갯수
PWD	현재 작업 디렉토리	TZ	타임 존 정보
SHELL	사용자의 로긴 쉘 파일의 절대경로	COLUMNS	터미널 열 수
DISPLAY	X 윈도우 디스플레이 장치	LINES	터미널 줄 수

-주요 공통 환경 변수들-

echo 명령 : 환경 변수 내용 확인

set 명령 : 환경 변수 설정 및 현재 환경변수의 목록 출력

사용자 환경 파일

사용자가 처음 로그인할때, 쉘을 수행할때, 로그아웃할 때 시스템이 자동으로 수행하는 환경파일

쉘 이름	환경파일 이름		
	로그인	쉘 수행	로그아웃
csh(tcsh)	.login	.cshrc	.logout
bash	.profile	.bashrc	.bash_logout

주요쉘의 사용자환경파일

Bash

강력한 쉘 프로그래밍 - 다양한 작업 제어기능과 연산 기능 제공

-> profile - 로그인시 수행 파일

-> bash_logout -로그아웃 시 수행 파일

-> bashrc -bash 쉘이수행할때마다실행

Bash 환경변수 설정

환경변수이름 = 값 : 환경변수 지정

export 명령 : 환경변수를 bash에 알림

별명 지정

형식 : alias 별명 = '원래명령어;

ex)

alias cls 'clear' => csh일 때

alias cls= 'clear' => bash일 때

bash 쉘 프롬프트 바꾸기

특수문자

특수문자	내용
\h	호스트 이름
\u	사용자 이름
\w	작업 디렉토리의 절대경로
\W	작업 디렉토리의 이름
\d	오늘 날짜
\t	현재 시간
\#	사건 번호
\!	히스토리 번호

사용 예시

-> exportPS1='[\h:\w]'

-> cd test

-> source .bashrc # 쉘 재실행

메타(Meta Character) 문자-쉘에서 특수하게 인식하는 문자들

쉘 문장 해석 시 메타문자가 존재하면 이 문자에 대한 특별한 기능 수행

문자	의 미
>	표준출력을 파일에 기록하는 출력 재지향(redirection)
>>	표준 출력을 파일 끝에 덧붙이는 출력 재지향
<	파일로부터 표준 입력을 읽는 입력 재지향
*	0개의 이상의 문자와 일치하는 파일 치환 대표 문자
?	단일 문자와 일치하는 파일 치환 대표 문자
[...]	대괄호 사이의 어떤 문자와도 일치하는 파일 치환 대표문자
|	어떤 프로세스의 출력을 다른 프로세스의 입력으로 보내는 파이프 기호
;	명령 실행 순서에 사용
||	이전의 명령이 실패하면 실행하는 조건부 실행
&&	이전의 명령이 성공하면 실행하는 조건부 실행
&	명령어를 백그라운드로 실행
#	# 문자에 뒤따르는 모든 문자들을 주석 처리
$	변수의 값을 표현
`cmd` 또는 $(cmd)	명령을 실행하고 값으로 대치
\	명령 해석을 지연하고 다음 줄로 계속 입력

메타 문자 기능을 없애는 방법 - 바로 앞에 \ 사입

ex)

& echo 메타문자

-> 메타문자

& echo 메타문자 > imsi.txt

& more imsi.txt

->메타문자

표준 입,출력 제어

기호	의미
>	표준출력을 파일로 기록
>>	표준출력을 파일의 끝에 추가
<	파일로부터 입력을 읽음

파이프 라인

한 명령의 출력을 다른 명령이나 쉘의 입력으로 연결하여 사용

여러 명령을 연결하여 복잡하거나 큰 작업을 빠르고 쉽게 구성 가능

|를 사용하여 연결

사용 예시
$ ls | grep ^f
->file.txt
->forsum.sh
->func.sh

와일드 문자

대표문자	의미
*	모든 문자열
?	한 문자와 일치
[..]	적어도 [ ] 안의 한 문자와 일치, “-” 를 이용하여 범위 지정 가능

조건부 실행
반환 값이 : “0”의 값을 반환하면 명령어성공, 1은 실패
&& : 이전 명령이 정상 종료인 경우에만 다음 명령 실행
|| : 이전 명령이 비정상 종료인 경우에만 다음 명령 실행
일반적인 and, or이다

7. 리눅스에서 많이 쓰이는 스크립트 언어

Bash
리눅스에서 기본적으로 제공되는 셸 언어
-> 명령어 실행: 시스템 명령어를 간편하게 실행할 수 있다.
-> 변수와 제어 구조: 변수 선언, 조건문(if, case), 반복문(for, while) 등을 지원
-> 스크립트 작성: .sh 파일로 저장하여 실행할 수 있고, 주로 시스템 관리 작업에 사용

Perl
텍스트 처리에 강력한 언어로, 시스템 관리와 웹 개발에 많이 사용
-> 정규 표현식: 강력한 정규 표현식 지원으로 텍스트 조작에 유리
-> CPAN: 다양한 모듈을 쉽게 사용할 수 있는 패키지 관리자
-> 스크립트 작성: .pl 파일로 저장하여 실행할 수 있다.

Python
강력하고 유연한 프로그래밍 언어로, 다양한 용도로 사용
-> 가독성: 간단하고 명확한 문법으로 초보자도 쉽게 배울 수 있다
-> 다양한 라이브러리: 시스템 관리, 데이터 처리, 웹 개발 등 다양한 라이브러리를 제공
-> 스크립트 작성: .py 파일로 저장하여 실행할 수 있다.

Ruby
간결하고 유연한 문법을 가진 프로그래밍 언어로, 주로 웹 개발에 사용
->객체 지향 프로그래밍: 모든 것이 객체로 처리
->Rails 프레임워크: Ruby on Rails와 같은 강력한 웹 프레임워크가 있다.
->스크립트 작성: .rb 파일로 저장하여 실행할 수 있다.

8 bash 쉘 스크립트 언어를 사용한 프로그래밍

Bash
강력한 쉘 프로그래밍 - 다양한 작업 제어기능과 연산 기능 제공 -> profile - 로그인시 수행 파일 -> bash_logout -로그아웃 시 수행 파일 -> bashrc -bash 쉘이수행할때마다실행

쉘 스크립트(Shell Script)는 여러 명령을 순서대로 모아놓은 명령어 집합 파일이다.
반복적인 작업을 자동화하고 시스템 관리, 파일 처리, 백업 등의 작업을 효율적으로 수행할 수 있다.

구분	설명	예시
변수선언	데이터를 저장하는 공간	count=10
출력명령어	변수나 문자열 출력	echo “value: $count”
조건문	특정 조건에 따라 분기 수행	if [$a -gt $b]; then echo “a>b”l fi
반복문	명령 반복 수행	for i in 1 2 3; do echo $i; done
함수	반복되는 코드의 재사용	function hello() { echo "Hi"; }
입력	사용자 입력 받기	read name

Bash 스크립트의 특징

인터프리터 언어	한 줄씩 해석되며 실행되므로 컴파일 과정이 필요 없음
자동화 용이	명령어 조합으로 반복 작업을 쉽게 자동화.
시스템 제어 가능	프로세스, 파일, 네트워크 등 운영체제 자원 제어 가능.
모듈화 지원	함수나 외부 스크립트 호출을 통해 구조적 프로그래밍 가능.

히스토리 기능 사용
사용자가 이전에 사용한 명령ef을 기록하고 나중에 다시 불러 사용
출력 예시]$history 1005 vi .ddd 1006 ls-al 1007 vi .exrc 1008 rm.exrc 1009 rm.exrc 1010 ls 1011 ls-al 1012 ls-al|more

간단한 명령어

!1010	히스토리 목록중 1010번에 해당하는 명령수행
!!	바로 직전 명령 수행
!h	최근 명령중 h로 시작하는 명령 수행

9. C프로그래밍 환경

C언어 특징

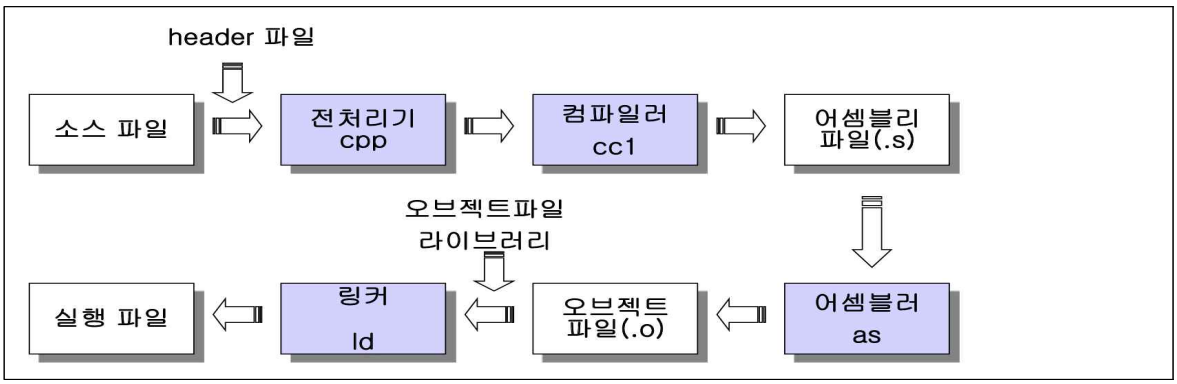
-> 타 기종간의 이식성과 유연성이 강력하다
-> 간결한 문법구조와 실행속도가 매우 빠르다.
-> 시스템프로그래밍 가능
-> 다양한 데이터 유형
-> 가독성이 뛰어나고 유지보수가 좋다
-> 모듈과 함수에 의한 프로그램 구조이다.

GCC

수 많은 컴퓨터 프로그래밍 언어들을 위한 라이브러리와 컴파일러들의 모음

파일	설명
GCC	GNU C 와C++ 컴파일러, cc1 과g++ 의기능을모두포함
CPP	C언어 전처리기
CCI	실제 C 컴파일러
AS	어셈블러
LD	링커

소스 프로그램부터 최종 실행 파일이 만들어지는 과정



1. 소스 파일 작성 (.c)
   사용자가 C언어로 프로그램을 작성한 코드.
2. 전처리기 (cpp)
   #include, #define 같은 전처리 지시문을 처리하고,
   모든 헤더파일을 포함한 새로운 코드로 변환.
   결과물: 전처리된 코드 (.i 파일)
3. 컴파일러 (cc1)
   전처리된 C 코드를 어셈블리어 코드(.s) 로 변환.
4. 어셈블러 (as)
   어셈블리 코드를 기계어 형태의 목적 파일(.o) 로 변환.
5. 링커 (ld)
   여러 개의 .o 파일과 라이브러리(libc 등) 를 연결해
   하나의 실행 파일(executable) 을 생성.
6. 실행 파일 생성 및 실행
   완성된 실행 파일은 OS의 로더(loader) 에 의해 메모리에 적재되어 실행된다.

GCC 명령어 옵션

옵션	기능
-v	실행 명령어들과 버전을 출력한다
-E	전처리만 실행한다; 컴파일하거나 어셈블하지 않는다
-S	컴파일만 실행한다; 어셈블하거나 링크하지 않는다
-c	컴파일 또는 어셈블한다; 링크하지 않는다
-g	운영체제 고유 형식으로 디버깅 정보를 만든다
-o <file>	출력을 file 에 둔다
-I<dir>	헤더 파일을 검색할 디렉토리를 추가한다
-L<dir>	-l 을 위해 검색할 디렉토리를 추가한다
-D<macro>	매크로를 미리 지정한다
-O<level>	최적화 수전을 지정한다. level 이 없으면 –O1 과 같다
-l<lib>	링크할 라이브러리 파일을 지정한다

전처리 예

전처리를 수행하면, C 구문에서"#"으로 시작하는 모든 코드가 해석되어 소스에 포함된다

```
gcc-E-o hello.i hello.c
[linux@seps linux]$ cat hello.i
# 1 "hello.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "hello.c"
# 1 "/usr/include/stdio.h" 1 3
```

.

프로그램 링킹

하나 또는 여러개의 목적(Object) 파일을 통합하여 하나의 실행 가능한 실행프로그램으로 만드는 작업

링커	링킹에사용하는도구프로그램
로더	실행프로그램을주기억장치에설치한후실행시킴

링킹의종류

-> 링킹(Linking)은 여러 개의 목적 파일(Object File)과 라이브러리를 결합하여 실행 파일을 생성하는 과정이다.

-> 링킹은 수행 시점에 따라 정적 링킹(Static Linking) 과 동적 링킹(Dynamic Linking) 으로 구분된다.

구분	장점	단점
정적링킹	실행파일 하나만으로 실행가능	실행 파일 크기 증가, 라이브러리 수정 시 재 컴파일 필요
동적링킹	실행 파일 크기가 작다, 메모리 절약이 된다	실행 파일 외에 라이브러리 필요

※ 참고 리눅스와 유닉스에서는 대부분의 프로그램이 **동적 링킹**방식으로 실행된다.

라이브러리 개념

라이브러리(Library)는 자주 사용하는 함수나 기능을 모아둔 프로그램 모듈의 집합이다

구분	사용시점	특징
정적 라이브러리(.a)	정적 링킹 시	실행 파일에 포함되어 독립 실행 가능
공유 라이브러리(.so)	동적 링킹 시	여러 프로그램이 하나의 라이브러리를 공유
동적 라이브러리(.so)	동적 로딩 시 (실행 중)	프로그램 실행 중 필요 시 로딩/제거 가능 (유연성 높음)

.예시

-> 웹 응용 프로그램에서 플러그인 모듈을 실행 중에 불러오는 기능은 동적 라이브러리 방식의 대표적 예다.

정적 라이브러리

ar 명령을 사용하여 오브젝트 파일들을 묶음

ar	
일반형식	ar [options] archive files…
주요옵션	d : 아카이브로부터 오브젝트 모듈들을 제거 r : 아카이브에 오브젝트 모듈들을 삽입. 이전에 존재하는 같은 모듈이 있으면 새로운 모듈로 대체. t : 아카이브 내용을 출력 x : 아카이브로부터 오브젝트 모듈을 추출 c : 아카이브파일을 생성 s : 아카이브에 오브젝트 파일 인덱스를 기록

공유 라이브러리

gcc 명령의 여러 옵션사용

- shared 는 공유 라이브러리를 사용한다는 명령

Make

많은 프로그램 모듈로 구성된 대규모 프로그램소스를 효율적으로 유지하고 일관성있게 관리하도록 도와주는 도구

모듈화된 프로그램

-여러개의 부분 프로그램 또는 모듈을 포함

장점 : 재사용 및 디스크 공간을 효율적으로 사용

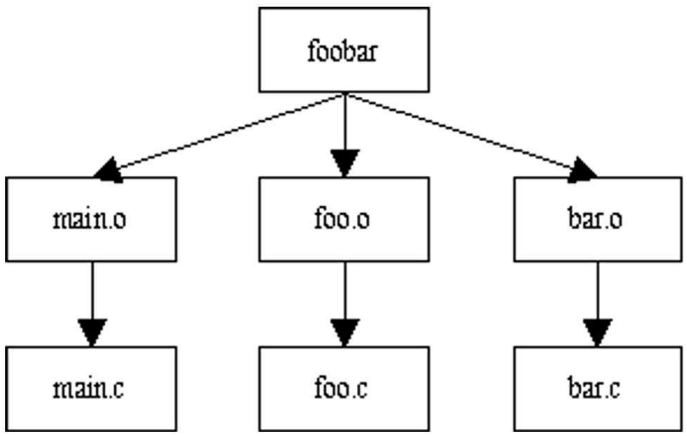
단점: 수정 시에 수작업으로 소스 프로그램 컴파일

하지만, 유지 보수에 어려움이 존재

makefile

-> Make 도구에서 가장 중요한 파일

-> 형식 : 대상(TARGET) … : 의존하는파일들(PREREQUISITES) … 명령(COMMAND)



최종실행파일“foobar” 는main.o, foo.o, bar.o 에의존하며, 마찬가지로 main.o, foo.o, bar.o 는 각각 main.c, foo.c, bar.c 에 의존

10 RCS 및 GNU autotools 과 Cmake

RCS (Revision Control System)
RCS는 텍스트 기반 프로그램의 수정 및 버전 관리를 위한 시스템입니다. 주로 단일 파일 또는 소규모 프로젝트의 버전 이력 관리에 사용되었으며, 간단한 사용법을 유지하는 것이 특징이다.

RCS의 기본 사용 방법
-> ci <filename> (Check in): → 새로운 파일에 대한 버전 관리를 시작하거나, 수정한 파일에 대해 새로운 버전을 생성할 때 사용됩니다. → 이 명령을 실행하면 원래의 소스 파일은 삭제되고, 버전 관리를 위한 파일(.c,v와 같은 형식)이 RCS라는 전용 디렉토리에 생성됩니다 → 사용자에게 버전 이력 관리를 위한 설명문(log message) 입력을 요청한다.
co <filename> (Check out): → 버전 관리 파일(.c,v)로부터 소스 파일을 작업 디렉토리로 가져오는 명령이다. → co -l <filename>: 파일을 가져옴과 동시에 잠금(lock) 상태로 만들어, 해당 파일의 수정을 예약하고 다른 사용자의 동시 수정을 방지한다.

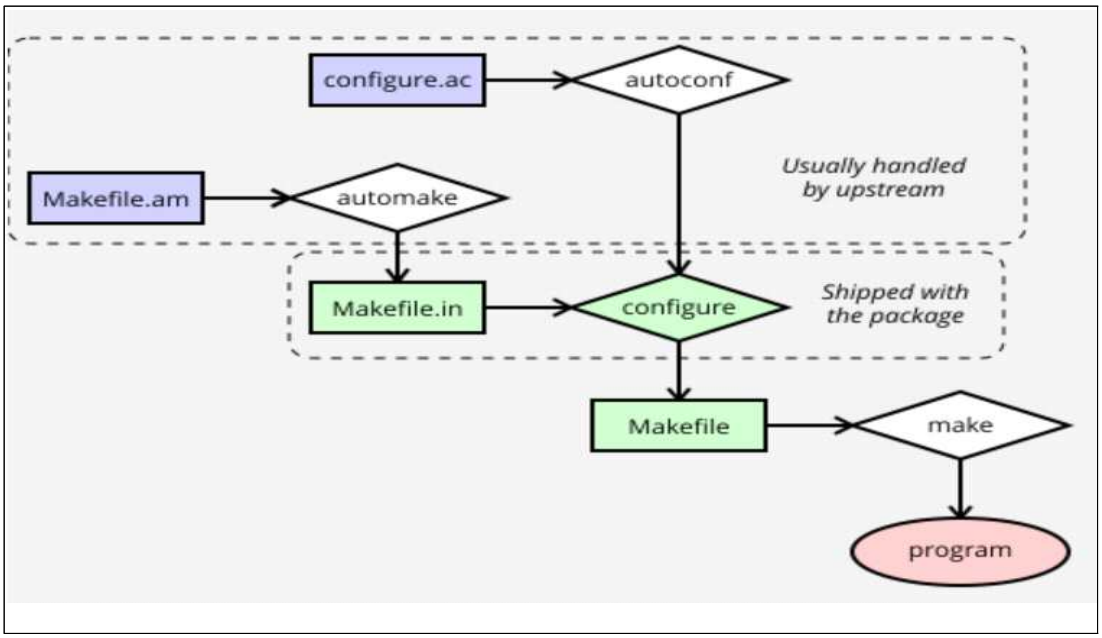
GNU Autotools (autoconf, automake, libtool)
GNU Autotools는 유닉스 계열 시스템에서 소프트웨어 패키지의 이식성(Portability)을 높이고, 해당 시스템에 맞는 빌드 환경을 자동으로 구성하기 위해 사용되는 도구 모음입니다. 주요 구성 요소로는 Autoconf, Automake, Libtool이 있다
Autoconf (configure.ac → configure): configure.ac 파일을 읽어 시스템 환경을 검사하는 스크립트 파일인 ./configure를 생성한다. Automake (Makefile.am → Makefile.in): Makefile.am 파일을 읽어 표준화된 Makefile.in 파일을 생성한다. Libtool: 정적/공유/동적 라이브러리 관리를 지원하여, 플랫폼 간 라이브러리 사용의 복잡성을 해소한다.

GNU Autotools의 빌드 환경 구성 방법

1. Configure 단계	다운로드한 패키지의 루트 디렉토리에서 ./configure 스크립트를 실행합니다. 이 스크립트는 Autoconf에 의해 생성된 것으로, 호스트 시스템의 환경(라이브러리 위치, 설치 경로 등)을 스캔하고 빌드에 필요한 변수를 설정하여 최종 Makefile을 생성합니다.
2. Make 단계	make 명령을 실행합니다. Automake에 의해 생성된 Makefile을 기반으로 소스 코드를 컴파일하고 기계어 코드로 변환하여 애플리케이션을 빌드합니다.
3. Install 단계	make install 명령을 실행합니다. 빌드된 파일들을 configure 단계에서 감지된 시스템의 적절한 위치로 복사하여 설치를 완료합니다.

장점	플랫폼 호환성: 다양한 Unix-like 시스템에서 동작 자동화: 소프트웨어의 구성 및 빌드를 자동화하여 사용자 편의성을 증대
단점	복잡성: 설정 파일이 복잡할 수 있어 초보자에게는 어려울 수 있다 속도: 초기 구성 과정이 느릴 수 있다.

GNU autotools의 내부 구성 요소



구성요소	역할	생성파일
autoconf	configure.ac 파일을 읽어 시스템 환경을 자동 감지하고, 설정 스크립트 생성	configure
automake	Makefile.am 파일을 이용해 표준화된 Makefile.in 파일을 생성	Makefile.in
libtool	여러 운영체제에서 공통적으로 사용할 수 있는 라이브러리 생성(정적/동적)을 지원	.a, .so 라이브러리

CMake

이식성을 높인 소스 코드 빌드 도구

-> GNU Autotools 하고 비슷하지만 유닉스/리눅스뿐만 아니라MS 윈도우, Mac OS등에서도 지원함

-> CMake는 하나의 CMakeLists.txt로 다양한 플랫폼의 빌드 파일(Unix Makefiles, Ninja, Visual Studio 등)을 생성해 주는 메타 빌드 도구다.

역할: CMake가 직접 빌드하지 않고, “빌드 시스템용 파일”을 만들어 주며 실제 빌드는 make·ninja·IDE가 수행한다.

핵심 단위: add_executable/add_library로 타겟을 만들고, target_link_libraries·target_include_directories 같은 target_* 명령으로 속성을 부여하는 현대식 타겟 중심 스타일을 쓴다.

vi CMakeList.txt → CMake 빌드 스크립트 작성

mkdir build && cd build → 소스와 빌드를 분리하기 위한 별도 디렉토리 생성 (out-of-source build)

```
jinho@localhost:~/make$ vi CMakeList.txt
jinho@localhost:~/make$ mkdir build;
jinho@localhost:~/make$ cd build
jinho@localhost:~/make/build$ cmake..
cmake..: command not found
jinho@localhost:~/make/build$ cmake ..
Command 'cmake' not found, but can be installed with:
sudo snap install cmake  # version 4.1.1, or
sudo apt  install cmake  # version 3.22.1-1ubuntu1.22.04.2
See 'snap info cmake' for additional versions.
jinho@localhost:~/make/build$ sudo apt update
```

결과 프롬프트에 CMake가 설치되어 있지 않아 업데이트 후 설치해준다.

```
jinho@localhost:~/make$ vi CMakeLists.txt
jinho@localhost:~/make$ cd ~/make/build
jinho@localhost:~/make/build$ cmake ..
-- Configuring done
-- Generating done
-- Build files have been written to: /home/jinho/make/build
jinho@localhost:~/make/build$ make
Consolidate compiler generated dependencies of target foobar
[ 25%] Building C object CMakeFiles/foobar.dir/main.c.o
[ 50%] Linking C executable foobar
[100%] Built target foobar
jinho@localhost:~/make/build$ ./foobar
Hello, world!
Goodbye, my love.
jinho@localhost:~/make/build$
```

make .. → 상위 디렉토리의 CMakeLists.txt를 읽어 Makefile 자동 생성

make → CMake가 생성한 Makefile을 바탕으로 main.c, foo.c, bar.c를 컴파일

[100%] Built target foobar → foobar 실행 파일 빌드 완료

./foobar 실행

의존성 관리와 외부 라이브러리 탐색까지 지원하기 때문에 효율적이다.

11. 디버깅 및 오류처리, Valgrind 정리

디버깅
구현된 소스 코드에 대한 검증이다. -> 디버깅 단계 1. 테스트 : 결함이나 버그가 존재하는지 확인 2. 안정화 : 버그를 반복 가능한 사태로 만듦 3. 구역화 : 코드의 위치를 탐색 4. 정정 : 코드 수정 5. 검증 : 제대로 동작하는지 확인
GDB를 이용한 디버깅

GDB(GNU Debugger)는 C 프로그램의 실행 중 오류를 추적하고 수정할 수 있도록 돕는 디버깅 도구이다.
디버깅 정보를 포함한 실행 파일을 만들기 위해서는 gcc 컴파일 시 -g 옵션을 사용해야 한다.
이 옵션은 디버깅 정보를 목적 코드에 삽입하여 GDB가 소스 코드를 추적할 수 있게 한다.
 	
-g1 : 최소 수준의 디버깅 정보 (기본 변수 정도만)
-g2 : 일반적인 디버깅 정보 (기본값)
-g3 : 매크로 정의 등 추가 정보 포함
예시
$ gcc -g dbg_first.c -o dbg_first
$ gdb dbg_first

1. files 관련 명령

cd	디버거의 작업 디렉토리를 변경
file	디버깅할 실행 파일 지정
list	특정 함수 또는 줄 번호의 코드 표시
pwd	현재 디렉토리 경로 확인
search	정규식을 이용해 코드 내 특정 문자열 검색

2. running 관련 명령

run	프로그램 실행 시작
continue	정지된 프로그램의 실행을 계속
kill	정지된 프로그램의 실행을 계속
step[N]	한 줄씩 실행하며 함수 내부로 진입 (N번 반복 가능)
next[N]	한 줄씩 실행하지만 함수 내부로는 들어가지 않음

3. breakpoints 관련 명령

break	특정 줄 번호나 함수 시작 지점에 정지점(breakpoint) 설정
watch	특정 변수 값이 변경될 때 실행 중단
clear	특정 줄의 정지점 해제
delete	정지점 번호를 지정하여 제거

4. data 관련 명령

display	프로그램이 정지될 때마다 특정 변수의 값을 자동출력
delete display	display 설정을 해제
print	변수의 현재 값을 일회성으로 출력
set variable	변수 값을 수동으로 변경
whatis	변수 또는 표현식의 자료형 출력

5. status / info 관련 명령

info	디버깅 중인 프로그램의 상태 정보를 표시
info args	함수의 인자 정보 표시
info locals	현재 스택 프레임 내 지역 변수 표시
info breakpoints	현재 설정된 정지점 정보 표시
info files	디버깅 대상 실행 파일 정보 표시
info types	정의된 데이터형 정보 표시

프로그램 디버깅
디버깅할 프로그램의 인자 지정
-> 디버깅할 프로그램이 입력 방법으로 프로그램인자를 사용하는 경우 이를 gdb에서 지원할 수 있어야 원활하게 프로그램을 디버깅 할 수 있다. gdb에서는 이를 위하여 run 명령어에서 직접 프로그램 인자를 입력하거나 set args 명령어를 통하여 프로그램 인자를 지정할 수 있다

단계별 실행	
각줄또는함수단위로동작상태를추적	
step	한 단계 실행. 함수 호출시 함수 내부 디버깅 가능
step n	n번 step 명령어 실행
Next	한 단계 실행. step 명령어와 달리 함수 호출도 한 단계로 처리
next n	n번 next명령어 실행

프로그램 상태 디버깅
프로그램이 실행되는 동안 변수 값, 메모리 상태, 실행 흐름 등 내부 상태를 관찰하고 분석하는 과정
-> 특정 변수의 값이 예상대로 변하는지 확인 (print, display 명령어 사용)
-> 프로그램이 어디서 멈췄는지 확인 (info, status 명령어 사용)

오류처리

오류번호 사용

fopen(), fclose(), socket(), listen() 등 많은 라이브러리 함수에서는 오류 발생시 함수의 반환값으로NULL 또는-1을 반환하고, 해당 오류 정보를 알려주기 위하여 오류번호 errno를 사용한다.

오류번호 errno

형식	#include <errno.h> extern int errno;
기능	가장 최근의 오류값을 나타냄
오류값	EBADF 잘못된 파일 기술자 EINPROGRESS 진행중인 오퍼레이션 EINVAL 잘못된 프로그램 인자 ENOENT 없는 파일 또는 디렉토리 ERANGE 결과값이 너무 큼 ETIMEDOUT 시간 초과된 오퍼레이션

strerror처리

발생 오류 번호인 errno에 대해 strerror() 함수를 사용하여 오류 원인 출력가능

	오류원인출력: strerror()
형식	#include<string.h> char *strerror(int errnum);
기능	오류원인을설명하는문자열을출력한다
반환값	인자errnum에 해당하는 문자열을반환하거나, 알지못하는오류의경우unknown error 메시지를 반환한다.

perror 처리

-> perror() 함수는 가장 최근의 오류원인을 문자열로 출력해줌

-> perror() 함수사용시 오류 발생함수를 사용하여 오류발생 지점을 명시하는 것이좋다

	시스템오류메시지출력: perror()
형식	#include<stdio.h> void perror(const char *s);
기능	문자열s가 NULL이 아니면 문자열s 다음에 ":"와 가장 최근의 오류원인을 설명하는 문자열을 함께 출력한다. 문자열s가 NULL이면 가장 최근의 오류원인을 설명하는 문자열만 출력한다.
반환값	없음

조건부 오류 처리

특정 수식을 이용한 오류조건을 검사하여 거짓이면, 오류메시지를 출력하고 프로그램 종료

	조건부종료: assert()
형식	#include<assert.h> void assert(int expression);
기능	만약수식expression이 거짓이면, 표준 출력stdout으로 오류 메시지를 출력하고 abort()를 호출해 프로그램을 종료한다. 만약 NDEBUG가 정의되어 있으면 실행되지않는다.

반환값	없음

메모리 손상 검사도구(valgrind)

-> 메모리 누수 뿐 아니라 다양한 메모리 손상에 대해 검사

-> 초기화 되지 않은 메모리 사용, 메모리 오버플로우, 스택 손상, 메모리 해제 후 해당 포인터에 대한 참조 등의 다양한 메모리 손상에 대해서도 검사해 준다.

-> 다양한 기능이 모듈화되어 있음

1. 메모리검사
2. 캐쉬 프로파일링
3. 쓰레드 오류 검출 등

일반형식	valgrind [옵션] 프로그램 [프로그램 옵션]
옵션	설명
-v	공유 객체, 경고 등과 같은 추가적인 정보를 나타낸다.
--tool=<도구이름>	Valgrind 내부도구를 실행. 디폴트는 memchec 이다.
--log-file=<로그파일>	저장할 로그파일 이름을 지정한다.
--xml=<yes|no>	yes 이면 출력 형식이 xml이다.
--leak-check<summary|yes |no|full>	메모리 누수에 대해 설정. 디폴트는 summary 이다.

예시 명령어 : valgrind —leak-check=full

-> ./a.out 실행 바이너리 ./a.out을 Valgrind Memcheck 도구로 감싸 실행하고, 동적 메모리 누수와 해제 오류를 상세하게(full) 분석하여 누수 위치(할당 스택 트레이스 포함)까지 보고한다.

사용환경 :

Linux/Unix에서 많이 쓰이고, Windows는 제한적 지원(WSL이나 cygwin 등으로는 사용 가능

<실습>

1. 리눅스 설치 및 테스트

관리자 권한으로 powerShell 실행

WSL 설치	wsl —install
WSL 업데이트	wsl --update

Microsoft Store 에서 “Ubuntu” 검색하여설치

-> Ubuntu 22.04.5

실행 후 사용자 계정 및 패스워드 생성 및 쉘 구동

```
PS C:₩WINDOWS₩system32> wsl --install
다운로드 중: Ubuntu
설치 중: Ubuntu
제공된 이름의 배포가 이미 있습니다. --name 사용하여 다른 이름을 선
택하십시오.
오류 코드: Wsl/InstallDistro/Service/RegisterDistro/ERROR_ALREADY_E
XISTS
PS C:₩WINDOWS₩system32> wsl --update
업데이트 확인 중입니다.
Linux용 Windows 하위 시스템 버전을 2.6.1(으)로 업데이트하는 중입니
다.
PS C:₩WINDOWS₩system32>
```

리눅스 테스트

lsb_release -a	배포판 및 버전 확인
uname -a	커널 버전 및 시스템 정보
pwd	현재 작업 디렉터리 확인
ls -al	파일 및 폴더 목록 보기

```
jinho@localhost:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 22.04.5 LTS
Release:        22.04
Codename:       jammy
jinho@localhost:~$ uname -a
Linux localhost 6.6.87.2-microsoft-standard-WSL2 #1 SMP PREEMPT_DYNAMIC Thu
 Jun  5 18:30:46 UTC 2025 x86_64 x86_64 x86_64 GNU/Linux
jinho@localhost:~$ pwd
/home/jinho
jinho@localhost:~$ ls -al
total 168
drwxr-x--- 16 jinho jinho  4096 Sep 30 14:58 .
drwxr-xr-x  3 root  root   4096 Apr 11  2025 ..
-rw-------  1 jinho jinho  4951 Oct 16 23:26 .bash_history
-rw-r--r--  1 jinho jinho   220 Apr 11  2025 .bash_logout
-rw-r--r--  1 jinho jinho  3771 Apr 11  2025 .bashrc
drwx------  3 jinho jinho  4096 Sep  9 10:39 .cache
drwx------  5 jinho jinho  4096 Sep 15 18:14 .config
drwxr-xr-x  3 jinho jinho  4096 Sep  9 10:39 .dotnet
-rw-r--r--  1 jinho jinho    48 Sep 30 14:57 .gitconfig
drwxr-xr-x  2 jinho jinho  4096 Apr 12  2025 .landscape
```

2. 리눅스 명령어를 사용해보고, 정규표현식을 사용하는 리눅스 유틸리티(grep,find)을 사용해본다.

ls (디렉토리파일보기)	
일반형식	ls[옵션]
주요옵션	-a, --all : 디렉토리내의모든파일을출력 -l, --format=long : 파일종류, 사용권한, 크기등출력 -s, --size : 1K 단위로파일크기를표시 -t, --sort=time : 최근에만들어진파일부터출력 -c, --time : 최근에변경한파일부터출력 -R : 하위디렉토리까지출력

ls

```
jinho@localhost:~$ ls
Test      catTest.txt  fib_fork.c        git_repos  temp     test.sh.save
Test.txt  ch4          fib_fork.c~       make       test
Test3     fib_fork     fibonacci_thread  name       test.c
Ttest     fib_fork.    file.txt          out        test.sh
```

ls -a

```
jinho@localhost:~$ ls -a
.               .lesshst                  Ttest            make
..              .local                    catTest.txt      name
.bash_history   .motd_shown               ch4              out
.bash_logout    .profile                  fib_fork         temp
.bashrc         .sudo_as_admin_successful fib_fork.        test
.cache          .viminfo                  fib_fork.c       test.c
.config         .vscode-server            fib_fork.c~      test.sh
.dotnet         Test                      fibonacci_thread test.sh.save
.gitconfig      Test.txt                  file.txt
```

ls -l

```
jinho@localhost:~$ ls -l
total 96
-rw-r--r-- 1 jinho jinho     0 Sep 15 17:50 Test
lrwxrwxrwx 1 jinho jinho    11 Sep 15 18:09 Test.txt -> catTest.txt
drwxr-xr-x 2 jinho jinho  4096 Sep 15 17:53 Test3
drwxr-xr-x 2 jinho jinho  4096 Sep 15 17:47 Ttest
-rw-r--r-- 1 jinho jinho    25 Sep 15 17:40 catTest.txt
drwxr-xr-x 2 jinho jinho  4096 Oct 14 21:56 ch4
-rwxr-xr-x 1 jinho jinho 16360 Apr 11  2025 fib_fork
-rw-r--r-- 1 jinho jinho   920 Apr 11  2025 fib_fork.
-rw-r--r-- 1 jinho jinho   919 Apr 11  2025 fib_fork.c
-rw-r--r-- 1 jinho jinho   912 Apr 11  2025 fib_fork.c~
drwxr-xr-x 3 jinho jinho  4096 Apr 12  2025 fibonacci_thread
-rw-r--r-- 1 jinho jinho   141 Sep 16 09:21 file.txt
drwxr-xr-x 3 jinho jinho  4096 Sep 30 10:05 git_repos
drwxr-xr-x 4 jinho jinho  4096 Sep 30 15:16 make
drwxr-xr-x 2 jinho jinho  4096 Sep 15 18:26 name
```

ls -s

```
jinho@localhost:~$ ls -s
total 96
 0 Test          16 fib_fork           4 git_repos    4 test.c
 0 Test.txt       4 fib_fork.          4 make         4 test.sh
 4 Test3          4 fib_fork.c         4 name         4 test.sh.save
 4 Ttest          4 fib_fork.c~        0 out
 4 catTest.txt    4 fibonacci_thread   4 temp
 4 ch4            4 file.txt          16 test
```

ls -t

```
jinho@localhost:~$ ls -t
ch4          test.sh.save  Test3  catTest.txt        fib_fork.   test.c
make         file.txt      temp   out                fib_fork
git_repos    name          Test   fibonacci_thread   fib_fork.c
test.sh      Test.txt      Ttest  fib_fork.c~        test
```

ls -R

```
jinho@localhost:~$ ls -R
.:
Test       catTest.txt  fib_fork.c        git_repos  temp      test.sh.save
Test.txt   ch4          fib_fork.c~       make       test
Test3      fib_fork     fibonacci_thread  name       test.c
Ttest      fib_fork.    file.txt          out        test.sh

./Test3:

./Ttest:

./ch4:
a.out    hello.i  main    sub.c        testlsearch.c  tofunc
hello    hello.o  main.c  sub.o        testopt        tofunc.c
hello.c  hello.s  main.o  testlsearch  testopt.c

./fibonacci_thread:
fibonacci_thread  fibonacci_thread.c  prodcons

./fibonacci_thread/prodcons:
prodcons  prodcons.c
```

cd 디렉토리 변경	
일반형식	cd [directory]

리눅스@사용자 $cd /usr

리눅스@사용자 $pwd # 현재 작업중인 디렉토리 확인

/usr

```
jinho@localhost:~$ cd /usr
jinho@localhost:/usr$ pwd
/usr
```

rm (파일의 삭제)	
일반형식	rm [-firv] source dest
주요옵션	울파일이있을경우강제로삭제 -i : 지울 파일이있을경우지울 것인지물어봄 -r : 하위 디렉토리에있는모든파일을삭제 -v : 지우는 파일정보를출력

```
jinho@localhost:~$ cd Test3
jinho@localhost:~/Test3$ ls
Test.txt
jinho@localhost:~/Test3$ rm Test.txt
jinho@localhost:~/Test3$ ls
jinho@localhost:~/Test3$
```

rm -i 파일이름

```
jinho@localhost:~/Test3$ rm -i Test.txt
rm: remove regular file 'Test.txt'? n
jinho@localhost:~/Test3$ ls
Test.txt
```

mkdir (디렉토리 생성)	
일반형식	mkdir [-m mode] [-p] dir …
주요옵션	-m : 새로운디렉토리의허가모드를지정한모드로설정 -p : 하위 디렉토리가존재하지않는경우함께생성

```
jinho@localhost:~/Test3$ mkdir deletfile
jinho@localhost:~/Test3$ ls
Test.txt  deletfile
jinho@localhost:~/Test3$
```

rmdir (디렉토리 삭제)	
일반형식	rmdir [-p] dir …
주요옵션	-p : 지정한 하위 디렉토리까지 삭제 -r : 하위 디렉토리 까지 삭제

rmdir - 폴더가 비어있지 않으면 삭제할 수 없다.

```
jinho@localhost:~/Test3$ ls
Test.txt  deletfile
jinho@localhost:~/Test3$ cd deletfile
jinho@localhost:~/Test3/deletfile$ rmdir twofile
jinho@localhost:~/Test3/deletfile$ cd ..
jinho@localhost:~/Test3$ rmdir deletfile
jinho@localhost:~/Test3$ ls
Test.txt
jinho@localhost:~/Test3$
```

cp (파일 복사)	
일반형식	cp [-fip] source dest cp [-fipr] source.. dest_dir
주요옵션	-f : 복사할파일이있을경우삭제하고복사 -i : 복사할파일이있을경우복사할것인지물어봄 -p : 원본파일의모든정보를보존한채복사 -r : 하위디렉토리에있는모든파일을복

cp (Test.txt파일을 Test3에 복사)

```
jinho@localhost:~$ ls
Test          ch4          fib_fork.c~          name      test.sh
Test.txt      deletfilea   fibonacci_thread     out       test.sh.save
Test3         fib_fork     file.txt             temp
Ttest         fib_fork.    git_repos            test
catTest.txt   fib_fork.c   make                 test.c
jinho@localhost:~$ cp Test.txt Test3
jinho@localhost:~$ cd Test3
jinho@localhost:~/Test3$ ls
Test.txt
jinho@localhost:~/Test3$
```

mv (파일 이름 변경와 옮기기)	
일반형식	mv [-fi] source dest mv [-fi] source … dest_dir
주요옵션	-b : 대상파일이지워지기전에백업파일을만듦 -f : 대상파일의접근허가와관계없이무조건파일을이동 -i : 대상파일이기존파일이면, 덮어쓸것인지물어봄 -u : 대상파일보다원본파일이최근의것일때업그레이드 -v : 파일옮기는과정을자세하게보여준다

mv 파일 이름 변경

```
jinho@localhost:~/Test3$ ls
Test.txt
jinho@localhost:~/Test3$ mv Test.txt change.txt
jinho@localhost:~/Test3$ ls
change.txt
```

mv 폴더 위치 변경

```
jinho@localhost:~/Test3$ mkdir testfile
jinho@localhost:~/Test3$ ls
change.txt  testfile
jinho@localhost:~/Test3$ mv change.txt testfile/
jinho@localhost:~/Test3$ ls
testfile
jinho@localhost:~/Test3$ cd testfile
jinho@localhost:~/Test3/testfile$ ls
change.txt
```

grep 문자열 검색 -> 기본 사용법 :grep '검색할패턴' 파일명

grep 'main' test.c -> test.c 파일에서 'main'이 포함된 줄 출력

```
jinho@localhost:~$ grep 'main' test.c
int main() {
```

grep 'TODO' *.c -> 여러 파일에서 찾기:

```
jinho@localhost:~$ grep 'int' *.c
fib_fork.c:void fibonacci(int n) {
fib_fork.c:    int a = 0, b = 1, next;
fib_fork.c:    printf("Fibonacci sequence up to %d terms:\n",
fib_fork.c:    for (int i = 0; i < n; i++) {
fib_fork.c:        printf("%d ", a);
fib_fork.c:    printf("\n");
fib_fork.c:int main(int argc, char *argv[]) {
fib_fork.c:        fprintf(stderr, "Usage: %s <positive_intege
argv[0]);
fib_fork.c:    int n = atoi(argv[1]);
fib_fork.c:        fprintf(stderr, "Error: negative number not
d.\n");
fib_fork.c:        printf("Child process finished.\n");
main.c:int main() {
main.c:    int x = 10, y = 2;
```

grep -E 'int|float' test.c ->'int' 또는 'float'가 포함된 줄 출력

```
jinho@localhost:~$ grep -E 'int|float' test.c
int main() {
    printf("Hello from Ubuntu WSL!\n");
```

find: 파일/디렉토리 검색 -> 기본 사용법 : find [검색경로][조건]

find . -name "*.c" -> 현재 폴더에서 모든 .c 파일 찾기

```
jinho@localhost:~$ find . -name "*.c"
\./fib_fork.c
./maindlopen.c
./ch4/testopt.c
./ch4/hello.c
./ch4/tofunc.c
./ch4/testlsearch.c
./ch4/main.c
./ch4/sub.c
./libsrc/arithmetic.c
./make/foo.c
./make/bar.c
```

find /home/jinho -name "TestLab.txt" -> 특정 이름의 파일 찾기

```
jinho@localhost:~$ find /home/jinho -name "TestLab.txt"
/home/jinho/gittest/TestLab.txt
```

find . -mtime -3 -> 3일 이내 수정된 파일 찾기

```
jinho@localhost:~$ find . -mtime -3
.
./.lesshst
./libarithmetic.so
./.addexpr.sh.swp
./deletfilea
./.viminfo
./maindlopen.c
./arithmetic.o
./libsrc
./libsrc/arithmetic.c
./libsrc/arithmetic.h
```

find . -size +1M -> 크기가 1MB 이상인 파일 찾기

```
jinho@localhost:~$ find . -size +1M
./.vscode-server/bin/6f17636121051a53c88d3e605c491d22af2ba755/node_modules/@xterm/xterm/lib/xterm.mjs.map
./.vscode-server/bin/6f17636121051a53c88d3e605c491d22af2ba755/node_modules/@vscode/tree-sitter-wasm/wasm/tree-sitter-typescript.wasm
./.vscode-server/bin/6f17636121051a53c88d3e605c491d22af2ba755/node_modules/@vscode/ripgrep/bin/rg
./.vscode-server/bin/6f17636121051a53c88d3e605c491d22af2ba755/node_modules/@vscode/vsce-sign/bin/vsce-sign
./.vscode-server/bin/6f17636121051a53c88d3e605c491d22af2ba755/extensions/css-language-features/server/dist/node/920.cssServerMain.js
```

3. vi 파일 편집 도구를 사용하여 파일을 작성, 수정하여 본다.

vi 편집기

(1) make 실습

여러 개의 소스 파일(main.c, foo.c, bar.c)을 자동으로 컴파일하고 실행 파일(foobar)을 생성한다.

Makefile, main.c, foo.c, bar.c 작성

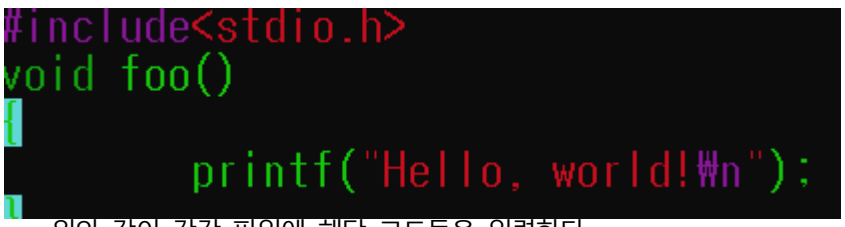
```
/* main.c */
void foo(), bar();
int main()
{
        foo();
        bar();
}
```

```
/* foo.c */
#include <stdio.h>
void foo()
{
    printf("Hello, world!₩n");
}
```

```
/* bar.c */
#include <stdio.h>
void bar()
{
    printf("Goodbye, my love.₩n");
}
```

```
[linux@seps make]$ make
gcc -c main.c
gcc -c foo.c
gcc -c bar.c
gcc -o foobar main.o foo.o bar.o
[linux@seps make]$
```

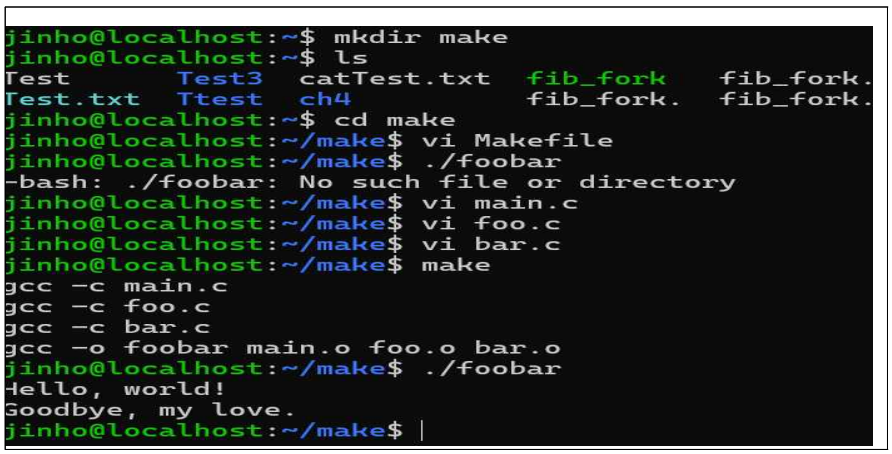
```
[linux@seps make]$ ./foobar
Hello, world!
Goodbye, my love.
[linux@seps make]$
```



-> 위와 같이 각각 파일에 해당 코드들을 입력한다.

make 명령어 실행 → foobar 실행 파일 생성

실행 결과 확인



실행을 하게 되면 다음과 같이 결과물이 정상적으로 출력되는 것을 알 수 있다.

make는 재사용 및 디스크 공간을 효율적으로 사용할 수 있지만 유지보수에 어려움이 있다.

(2)CMake 실습

vi CMakeList.txt → CMake 빌드 스크립트 작성

mkdir build && cd build → 소스와 빌드를 분리하기 위한 별도 디렉토리 생성 (out-of-source build)

```
jinho@localhost:~/make$ vi CMakeList.txt
jinho@localhost:~/make$ mkdir build;
jinho@localhost:~/make$ cd build
jinho@localhost:~/make/build$ cmake..
cmake..: command not found
jinho@localhost:~/make/build$ cmake ..
Command 'cmake' not found, but can be installed with:
sudo snap install cmake  # version 4.1.1, or
sudo apt  install cmake  # version 3.22.1-1ubuntu1.22.04.2
See 'snap info cmake' for additional versions.
jinho@localhost:~/make/build$ sudo apt update
```

결과 프롬프트에 CMake가 설치되어 있지 않아 업데이트 후 설치해준다.

```
jinho@localhost:~/make$ vi CMakeLists.txt
jinho@localhost:~/make$ cd ~/make/build
jinho@localhost:~/make/build$ cmake ..
-- Configuring done
-- Generating done
-- Build files have been written to: /home/jinho/make/build
jinho@localhost:~/make/build$ make
Consolidate compiler generated dependencies of target foobar
[ 25%] Building C object CMakeFiles/foobar.dir/main.c.o
[ 50%] Linking C executable foobar
[100%] Built target foobar
jinho@localhost:~/make/build$ ./foobar
Hello, world!
Goodbye, my love.
jinho@localhost:~/make/build$
```

make .. → 상위 디렉토리의 CMakeLists.txt를 읽어 Makefile 자동 생성

make → CMake가 생성한 Makefile을 바탕으로 main.c, foo.c, bar.c를 컴파일

[100%] Built target foobar → foobar 실행 파일 빌드 완료

./foobar 실행

CMakesms make와 달리 CMakeLists.txt를 바탕으로 자동으로 Makefile을 생성해준다.

즉, 플랫폼이 독립적이며,

의존성 관리와 외부 라이브러리 탐색까지 지원하기 때문에 효율적이다.

practice.txt를 편집하여 코드를 작성하고 cat으로 출력을 해본다

```
jinho@localhost:~/filemake$ vi practice.txt
jinho@localhost:~/filemake$ cat practice.txt
#include <stdio.h>

int main:
        printf("he      return0;
}
jinho@localhost:~/filemake$
```

그 후 vi편집기로 다시 파일을 수정해 보면서 vi 편집기에 익숙해져 본다.

```
jinho@localhost:~/filemake$ vi practice.txt
jinho@localhost:~/filemake$ cat practice.txt
#include <stdio.h>

int main(){
        printf("hello, vi!₩n");
        return0;
}
```

파일은 I를 눌러 insert하고 코드를 다 작성하면 esc로 나간 후 :wq!로 저장하고 vi를 종료한다.

4. 쉘 명령어에 익숙해지고, 별명,쉘 프롬프트,히스토리 기능의 변경이 포함된 .bashrc와 같은 쉘 설정 파일을 직접 수정해본다.

쉘 환경 확인

```
jinho@localhost:~$ echo $SHELL
/bin/bash
```

bash 버전 확인

```
jinho@localhost:~$ bash --version
GNU bash, version 5.1.16(1)-release (x86_64-pc-linux-gnu)
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

별명 만들기 (alias)

```
jinho@localhost:~$ alias ll='ls -alF'
```

-> ls -alF를 이제 ll로 사용가능 합니다.

alias한 ll 사용

```
jinho@localhost:~$ alias ll='ls -alF'
jinho@localhost:~$ ll
total 176
drwxr-x--- 18 jinho jinho  4096 Oct 17 02:11 ./
drwxr-xr-x  3 root  root   4096 Apr 11  2025 ../
-rw-------  1 jinho jinho  5209 Oct 17 00:21 .bash_history
-rw-r--r--  1 jinho jinho   220 Apr 11  2025 .bash_logout
-rw-r--r--  1 jinho jinho  3771 Apr 11  2025 .bashrc
```

쉘 프롬프트 변경

```
jinho@localhost:~& export PS1="\h\u@:\w& "
localhostjinho@:~&
```

-> 사용자명과 호스트명의 순서를 바꾸었다.

히스토리

!1 -> 1번째 명령문, !2 2번째 명령문

!1 -> 직전 명령문 !cat -> cat이 들어간 명령문 들의 과거 목록을 출력 후 실행.

```
localhostjinho@:~& !1
cd
localhostjinho@:~& !2
ls
Test      catTest.txt  fib_fork.         file.txt   name  test.c
Test.txt  ch4          fib_fork.c        filemake   out   test.sh
Test3     deletfilea   fib_fork.c~       git_repos  temp  test.sh.save
Ttest     fib_fork     fibonacci_thread  make       test
localhostjinho@:~& !!
ls
Test      catTest.txt  fib_fork.         file.txt   name  test.c
Test.txt  ch4          fib_fork.c        filemake   out   test.sh
Test3     deletfilea   fib_fork.c~       git_repos  temp  test.sh.save
Ttest     fib_fork     fibonacci_thread  make       test
localhostjinho@:~& !cat
cat practice.txt
```

.bashrc 쉘 설정 파일 수정

-> vi ~/.bashrc로 편집을 시작한다.

```
alias mkd='mkdir'
export PS1="₩u₩$ "
!mkd

-- INSERT --
```

다음과 같은 별명, 쉘 프롬프트, 히스토리 기능을 추가하고 :wq!로 저장하고 vi 편집기에서 나간다.

source ~/.bashrc로 현재 셸에서 bashrc파일을 바로 실행한다

```
u@h:w$source ~/.bashrc
!mkd: command not found
jinho$ vi ~/.bashrc
```

5. bash 쉘 스크립트를 이용하여 다음 조건에 맞는 bash 쉘 스크립트 작성

1. “ls” 명령을 사용하여 현재 디렉토리에서 파일 개수를 출력하는 쉘 프로그램을 작성 해본다.

```
jinho$ vi count_files.sh
```

vi 편집기를 이용하여 count_files.sh를 생성하며 편집을한다

```
#i/bin/bash
ls | wc -l
```

-> count_filse.sh 파일의 코드

```
jinho$ chmod +x count_files.sh
jinho$ ./count_files.sh
24
```

count_files에 실행 권한을 준 후 ./로 실행 해준다.

2. 반복문(while, for, until)을 사용하여 구구단을 표시하는 쉘 스크립트를 작성하시오

3개의 반복문중 상태가 아니라 횟수에 집중되는 문제이므로 while이나 until보다 for문이 더 활용하기 쉽다.

while문

```
dan=2
while [ $dan -le 9]
do
        echo "-------$dan 단-------"
        i=1
        while [ $i -le 9]
        do
                ehco "$dam X $i = $((dan*i))
                i=$((i+1))
        done
        dan=$((dan+1))
done
```

for 문

```
for dan in {2..9}
do
        echo "-------$dan단-------"
        for i in {1..9}
        do
                echo "$dan X $i = $((dan *i))"
        done
done
```

for문으로 실행

```
jinho$ ./gugu_f.sh
--------2단--------
2 X 1 = 2
2 X 2 = 4
2 X 3 = 6
2 X 4 = 8
2 X 5 = 10
2 X 6 = 12
2 X 7 = 14
2 X 8 = 16
2 X 9 = 18
--------3단--------
3 X 1 = 3
3 X 2 = 6
3 X 3 = 9
3 X 4 = 12
3 X 5 = 15
3 X 6 = 18
3 X 7 = 21
3 X 8 = 24
3 X 9 = 27
--------4단--------
4 X 1 = 4
4 X 2 = 8
4 X 3 = 12
4 X 4 = 16
4 X 5 = 20
4 X 6 = 24
4 X 7 = 28
4 X 8 = 32
4 X 9 = 36
--------5단--------
5 X 1 = 5
5 X 2 = 10
5 X 3 = 15
5 X 4 = 20
5 X 5 = 25
5 X 6 = 30
5 X 7 = 35
5 X 8 = 40
5 X 9 = 45
--------6단--------
6 X 1 = 6
6 X 2 = 12
6 X 3 = 18
6 X 4 = 24
6 X 5 = 30
6 X 6 = 36
6 X 7 = 42
6 X 8 = 48
6 X 9 = 54
--------7단--------
7 X 1 = 7
7 X 2 = 14
7 X 3 = 21
7 X 4 = 28
7 X 5 = 35
7 X 6 = 42
7 X 7 = 49
7 X 8 = 56
7 X 9 = 63
--------8단--------
8 X 1 = 8
8 X 2 = 16
8 X 3 = 24
8 X 4 = 32
8 X 5 = 40
8 X 6 = 48
8 X 7 = 56
8 X 8 = 64
8 X 9 = 72
--------9단--------
9 X 1 = 9
9 X 2 = 18
9 X 3 = 27
9 X 4 = 36
9 X 5 = 45
9 X 6 = 54
9 X 7 = 63
9 X 8 = 72
9 X 9 = 81
```

3. "let" 또는 "expr" 명령을 사용하여 피보나치 수열을 나타내는 쉘 스크립트를 작성하시오

let을 이용한 i<10 까지의 피보나치 수열

```
#!/bin/bash

count=10
numbera=0
numberb=1

echo "피보나치 수열 (let 버전):"
for ((i=0; i<$count; i++))
do
    echo "$numbera"
    let fn=numbera+numberb
    let numbera=numberb
    let numberb=fn
done
~
```

결과

```
피보나치 수열 (let 버전):
0
1
1
2
3
5
8
13
21
34
```

expr 버전

```
count=10
numbera=0
numberb=1

echo "피보나치 수열 (expr 버전):"
i=0
while [ $i -lt $count ]
do
    echo "$numbera"
    fn=$(expr $numbera + $numberb)
    numbera=$numberb
    numberb=$fn
    i=$(expr $i + 1)
done
```

-> expr는 $를 사용하고 let은 사용하지 않는다.

결과

```
피보나치 수열 (expr 버전):
0
1
1
2
3
5
8
13
21
34
```

4. 다음과 같은 쉘 스크립트를 작성하시오 (별찍기)

*****

****

***

**

*

해당코드

```
#!/bin/bash
for ((i=5; i>=1; i--))
do
    for ((j=1; j<=i; j++))
    do
        echo -n "*"
    done
    echo
done
```

결과

```
jinho$ ./staradd.sh
*****
****
***
**
*
```

6. 사칙연산에 대한 함수와 이 함수들을 이용하는 예제 프로그램을 각각 작성한 후 정적,공유,동적 라이브러리 방식을 각각 이용하여 예제 프로그램을 실행시킨 결과를 보이시오.

사칙연산 c 파일 만들기 vi arithmetic.c

```c
int add(int a, int b) { return a + b; }
int subtract(int a, int b) { return a - b; }
int multiply(int a, int b) { return a * b; }
int divide(int a, int b) { return b ? a / b : 0; }
```

사칙연산 h 파일 만들기 (헤더)

```c
int add(int a, int b);
int subtract(int a, int b);
int multiply(int a, int b);
int divide(int a, int b);
```

실행할 main 함수 만들기

```c
#include <stdio.h>
#include "arithmetic.h"

int main() {
    int x = 10, y = 2;
    printf("add: %d\n", add(x, y));
    printf("subtract: %d\n", subtract(x, y));
    printf("multiply: %d\n", multiply(x, y));
    printf("divide: %d\n", divide(x, y));
    return 0;
}
```

위에 만들었던 헤더파일을 include해주고 x와 y값을 지정해주고 모든 함수를 print해준다.

정적 라이브러리 만들기

a. 오브젝트 파일 생성 -> gcc -c arithmetic.c

b. 정적 라이브러리 생성 -> ar rcs libarithmetic.a arithmetic.o

c. 컴파일 -> gcc main.c -L. -larithmetic -o main_static

d. 실행 -> ./main_static

```
jinho$ gcc -c arithmetic.c
jinho$ ar rcs libarithmetic.a arithmetic.o
jinho$ gcc main.c -L. -larithmetic -o main_static
jinho$ ./main_static
add: 12
subtract: 8
multiply: 20
divide: 5
```

공유 라이브러리 만들기

생성 -> gcc -fPIC -c arithmetic.c

gcc -shared -o libarithmetic.so arithmetic.o

컴파일 -> gcc main.c -L. -larithmetic -o main_shared

실행(라이브러리 경로 지정) export LD_LIBRARY_PATH=.

./main_shared

```
gcc -fPIC -c arithmetic.c
gcc -shared -o libarithmetic.so arithmetic.o
gcc main.c -L. -larithmetic -o main_shared
```

```
jinho$ export LD_LIBRARY_PATH=.
jinho$ ./main_shared
add: 12
subtract: 8
multiply: 20
divide: 5
```

동적 라이브러리(런타임 로딩)

vi main_dlopen.c 로 편집

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

int main() {
    void *handle = dlopen("./libarithmetic.so", RTLD_LAZY);
    if (!handle) {
        printf("Error: %s\n", dlerror());
        return 1;
    }

    int (*add)(int, int) = dlsym(handle, "add");
    int (*subtract)(int, int) = dlsym(handle, "subtract");
    int (*multiply)(int, int) = dlsym(handle, "multiply");
    int (*divide)(int, int) = dlsym(handle, "divide");

    int x = 10, y = 2;
    printf("add: %d\n", add(x, y));
    printf("subtract: %d\n", subtract(x, y));
    printf("multiply: %d\n", multiply(x, y));
    printf("divide: %d\n", divide(x, y));

    dlclose(handle);
    return 0;
}
```

공유라이브러리 준비 후 컴파일, 실행

```
jinho$ vi main_dlopen.c
jinho$ gcc -fPIC -c arithmetic.c
jinho$ gcc -shared -o libarithmetic.so arithmetic.o
jinho$ gcc main_dlopen.c -ldl -o main_dynamic
jinho$ ./main_dynamic
add: 12
subtract: 8
multiply: 20
divide: 5
```

7. 6번 문제에 대하여, c 소스 파일들을 목적 파일로 만들고, 또한 이 파일들을 이용하여 라이브러리 또는 실행 파일로 만들어주는 Makefile을 작성하여 시켜보시오. 이때, 라이브러리 관련 파일들은 서브 디렉토리에 두도록 한다.

```
project/
├── libsrc/          # 라이브러리용 소스, 헤더
│   ├── arithmetic.c
│   └── arithmetic.h
├── main.c           # 예제 프로그램
└── Makefile         # 최상위 Makefile
```

--> 해당 폴더들이 다음과 같은 구조로 파일들을 옮긴다. (6번에서 사용했던 함수들 재사용)

```
# Makefile (project/Makefile)
CC = gcc
AR = ar
CFLAGS = -I./libsrc
LIBDIR = ./libsrc
LIBSRC = $(LIBDIR)/arithmetic.c
LIBOBJ = $(LIBDIR)/arithmetic.o
LIBSTATIC = $(LIBDIR)/libarithmetic.a
LIBSHARED = $(LIBDIR)/libarithmetic.so
MAIN = main.c
TARGET_STATIC = main_static
TARGET_SHARED = main_shared

all: static shared

# 1. 목적 파일 만들기
$(LIBOBJ): $(LIBSRC)
        $(CC) -c $(LIBSRC) -o $(LIBOBJ)

# 2. 정적 라이브러리 만들기
static: $(LIBOBJ) $(MAIN)
        $(AR) rcs $(LIBSTATIC) $(LIBOBJ)
        $(CC) $(MAIN) -L$(LIBDIR) -larithmetic $(CFLAGS) -o $(TARGET_
STATIC)
```

```
# 3. 공유 라이브러리 만들기
shared: $(LIBSRC) $(MAIN)
        $(CC) -fPIC -c $(LIBSRC) -o $(LIBOBJ)
        $(CC) -shared -o $(LIBSHARED) $(LIBOBJ)
        $(CC) $(MAIN) -L$(LIBDIR) -larithmetic $(CFLAGS) -o $(TARGET_
SHARED)

# 4. 실행 파일 삭제
clean:
        rm -f $(LIBDIR)/*.o $(LIBDIR)/*.a $(LIBDIR)/*.so main_static
main_shared
```

-> 다음과 같이 Makefile에 코드를 입력해준다

소스 코드를 컴파일하여 정적 라이브러리와 공유 라이브러리를 각각 생성하고, 이 라이브러리들을 사용하는 두 개의 다른 실행 파일을 만드는 과정을 자동화하는 스크립트이다.

make (makefile을 찾아 실행) 정적,공유 라이브러리, 실행파일등이 자동으로 생성된다.

```
jinho@localhost:~$ make
gcc -c ./libsrc/arithmetic.c -o ./libsrc/arithmetic.o
ar rcs ./libsrc/libarithmetic.a ./libsrc/arithmetic.o
gcc main.c -L./libsrc -larithmetic -I./libsrc -o main_static
gcc -fPIC -c ./libsrc/arithmetic.c -o ./libsrc/arithmetic.o
gcc -shared -o ./libsrc/libarithmetic.so ./libsrc/arithmetic.o
gcc main.c -L./libsrc -larithmetic -I./libsrc -o main_shared
```

정적 라이브러리 실행파일 실행

```
jinho@localhost:~$ ./main_shared
add: 12
subtract: 8
multiply: 20
divide: 5
```

export LD_LIBRARY_PATH=./libsrc 명령어로 공유 라이브러리 실행전 경로 지정

-< ./main_shared

```
jinho@localhost:~$ export LD_LIBRARY_PATH=./libsrc
jinho@localhost:~$ ./main_shared
add: 12
subtract: 8
multiply: 20
divide: 5
```

make clean -> 생성된 목적파일/라이브러리/실행파일 모두 삭제됨

```
jinho@localhost:~$ make clean
rm -f ./libsrc/*.o ./libsrc/*.a ./libsrc/*.so main_static main_shared
```

8. GIT 의 사용법을 정리하고 Github 사이트에 자신의 계정을 만들고 위아래의 모든 과제를 lab2 프로젝트에 올리시오.

1. Git 사용법 기본 정리

Git은 버전 관리를 위한 도구

변경 이력 추적, 협업, 백업, 분기(브랜치), 병합(merge) 등 가능

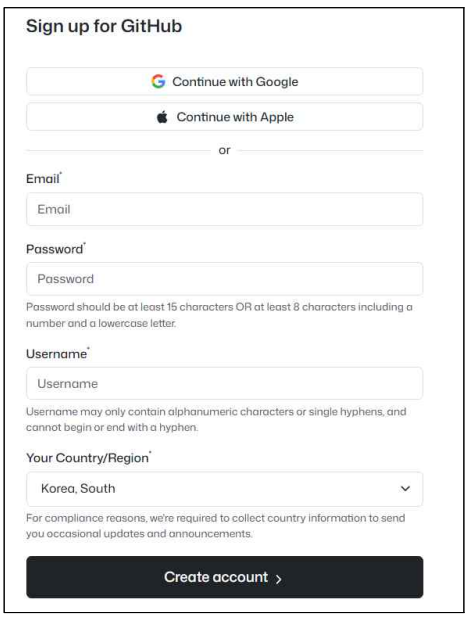
주요 명령어

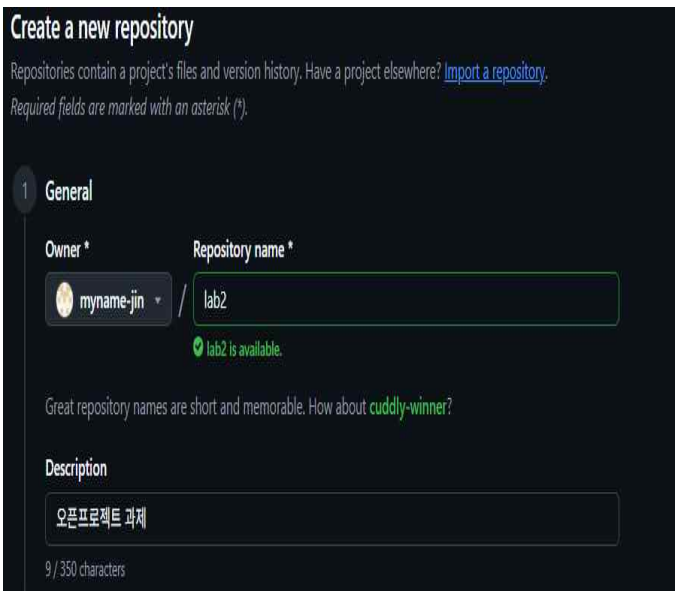
git init	로컬 저장소 초기화
git add .	변경된 모든 파일 추가(스테이지)
git commit -m"메시지"	변경사항 커밋(기록)
git remote add origin https://github.com/id	Github 연결(원격 저장소 추가)
git push -u origin master	로컬 커밋 푸시(Github에 업로드)
git pull origin master	변경사항 최신화(다른 PC 등에서)

git status	현재 상태 확인
git log	커밋 이력 확인
git branch[이름]	브랜치 생성
git checkout[이름]	브랜치 이동
git merge	병합

Github 계정 만들기

1. https://github.com 사이트에 들어간다
2. 계정 생성(Sign Up)
3. 인증 및 자동 등로방지 단계를 완료한다
4. Create new repository





Ubuntu 환경에서의 git

깃 업데이트	sudo apt update
깃 설치	sudo apt install git

깃 다운로드

```
jinho@localhost:~$ sudo apt update
sudo apt install git
[sudo] password for jinho:
Sorry, try again.
[sudo] password for jinho:
Get:1 http://security.ubuntu.com/ubuntu jammy-security InRelease [129
 kB]
Hit:2 http://archive.ubuntu.com/ubuntu jammy InRelease
Get:3 http://archive.ubuntu.com/ubuntu jammy-updates InRelease [128 k
B]
Get:4 http://security.ubuntu.com/ubuntu jammy-security/main amd64 Pac
kages [2736 kB]
Get:5 http://archive.ubuntu.com/ubuntu jammy-backports InRelease [127
 kB]
```

git 주소와 내 디렉토리를 연결

```
jinho@localhost:~/gittest$ git remote add origin https://github.com/m
yname-jin/gittest.git
```

git init (초기화)

```
jinho@localhost:~/gittest$ git init
hint: Using 'master' as the name for the initial branch. This default
 branch name
hint: is subject to change. To configure the initial branch name to u
se in all
hint: of your new repositories, which will suppress this warning, cal
l:
```

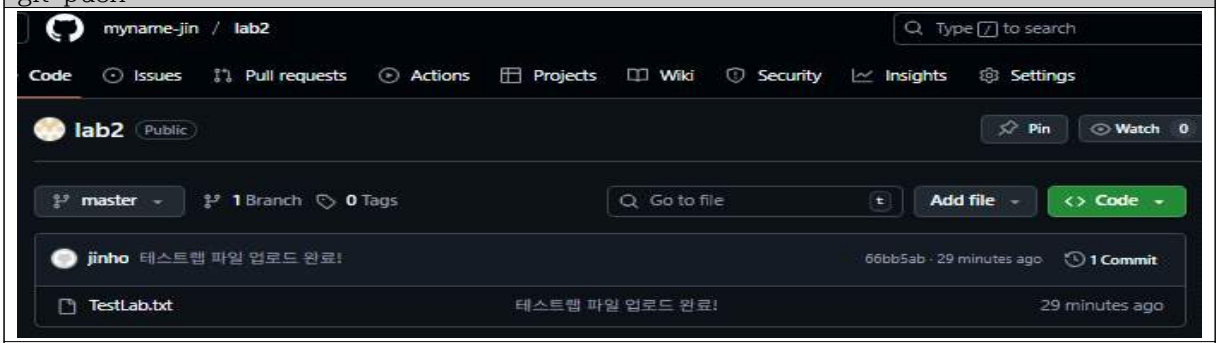
txt파일을 add

```
jinho@localhost:~/gittest$ git add TestLab.txt
jinho@localhost:~/gittest$ git add .
```

commit (파일을 저장소에 올린다)

```
jinho@localhost:~/gittest$ git commit -m "테스트랩 파일 업로드 완료!"
[master (root-commit) 66bb5ab] 테스트랩 파일 업로드 완료!
 1 file changed, 1 insertion(+)
 create mode 100755 TestLab.txt
```

git push



lab보고서를 작성중이기 때문에 임시 파일을 업로드 -> 추후에 작성 완료 후 올릴 예정

git url : https://github.com/myname-jin/lab2

9. GDB와 Vscode의 사용법을 정리하고, 위 2번 실습 예제에 대하여 GDB 및 Vscode 테스트 결과를 나타내시오.

GDB 사용법

-> GDB(GNU Debugger)는 C/C++ 프로그램 디버깅 툴

-> 주요 기능: 브레이크포인트 설정, 변수 값 확인, 단계별 실행

디버깅 정보 포함해서 컴파일	gcc -g main.c -o main
프로그램 디버깅 시작	gdb ./main
main 함수에서 실행 중단(브레이크포인트)	break main
프로그램 실행 시작	run
다음 한 줄 실행	next
함수 안으로 한 단계 진입	step
변수 값 출력	print변수명
실행 계속	continue
종료	quit

VSCode 사용법

-> VSCode(Visual Studio Code)는 코드 편집과 디버깅이 가능한 IDE

-> C/C++ Extension 설치 필요 (C/C++ by Microsoft)

-> F5(디버깅), Breakpoint, 변수값 즉시 보기 등 GUI 기반으로 편리한 디버깅

기본 사용 순서

1. C/C++ Extension 설치
2. launch.json 생성 (디버깅 환경 세팅: 자동 생성 됨)
3. Build & Debug

   코드 편집 → Ctrl+S (저장)

   F5: 디버깅 시작 (브레이크포인트 걸기, 변수 확인, 실행 흐름 제어)

   변수/스택/콘솔 창에서 실시간 값 추적

GDB로 실습2번 테스트

vi test_grep.c (테스트할 c파일을 생성)

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

int main() {
    printf("grep 명령 실행\n");
    system("grep main test.c");
    return 0;
}
```

gcc -g test_grep.c -o test_grep -> 디버그 정보 포함 컴파일

gdb ./test_grep

```
jinho@localhost:~/test3$ gdb ./test_grep
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04.2) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.or
pl.html>
This is free software: you are free to change and redistr
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
```

(gdb) break main -> main 함수에 브레이크포인트 설정

```
(gdb) break main
Breakpoint 1 at 0x1171: file test_grep.c, line 5.
```

(gdb) run -> 프로그램 실행

```
(gdb) run
Starting program: /home/jinho/Test3/test_grep
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-g
o.1".

Breakpoint 1, main () at test_grep.c:5
warning: Source file is more recent than executable.
5           printf("grep 명령 실행\n");
```

(gdb) next -> 한 줄씩 실행하며 system() 함수 호출 전후 관찰

```
(gdb) next
grep 명령 실행
6           system("grep main test.c");
```

(gdb) quit -> 프로그램 종료

```
(gdb) quit
A debugging session is active.

        Inferior 1 [process 1719] will be killed.

Quit anyway? (y or n) y
```

vscode 테스트

-> gdb와 같은 파일인 test_grep.c를 활용

test_grep.c U

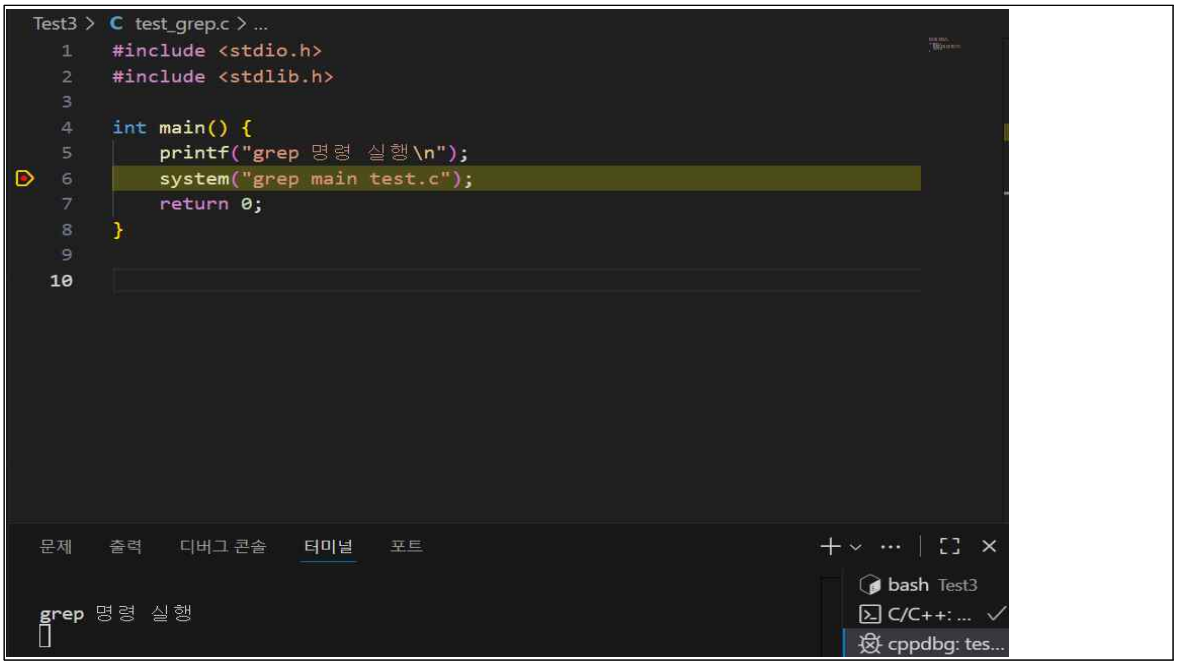
Test3 > test_grep.c > ...

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

int main() {
    printf("grep 명령 실행\n");
    system("grep main test.c");
    return 0;
}
```

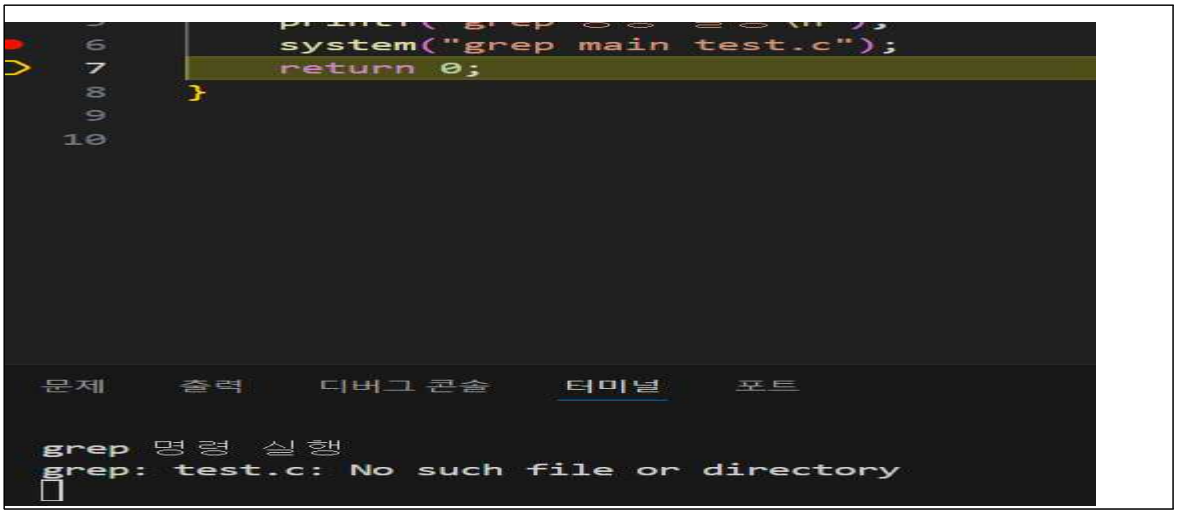
중단점 지정

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

int main() {
    printf("grep 명령 실행\n");
    system("grep main test.c");
    return 0;
}
```

-> F5 디버그 시작



-> 다음과 같이 중단점 윗줄인 명령실행 부분만 출력이되고 멈춘 상태이다.



->F10으로 한줄만 스크립트를 진행하여 system 행이 실행되었다.

```
1    './csu/../sysdeps/nptl/libc_start_call_main.h' 소스를 로드할 수 없습니
```

-> 마지막 줄 까지 디버그를 하게되면 다음과 같은 내용이 뜬다

->‘./csu/../sysdeps/nptl/libc_start_call_main.h' 소스를
로드할 수 없습니다. 'SourceRequest' not supported..

10. assert() 함수 구현 : 라이브러리 함수인 assert() 함수와 동일하게 동작하는 my_assert() 함수를 구현하고, 이를 이용하여 5장 예제 14에서 assert() 함수를 대체하여 프로그램을 실행해 보자

assert()함수와 동일하게 동작하는 my_assert()함수

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

void my_assert(int expression, const char *expr_str, const char *file, int line) {
    if (!expression) {
        fprintf(stderr, "Assertion failed: %s, file %s, line %d\n", expr_str, file, line);
        abort();
    }
}

// 사용 예시 (매크로로 편리하게)
#define MY_ASSERT(expr) my_assert((expr), #expr, __FILE__, __LINE__)
```

->매크로를 이용해 함수 호출 시 파일 이름과 줄 번호 등 디버깅 정보를 자동으로 전달

```c
int main() {
    int x = 0;
    MY_ASSERT(x != 0);  // 거짓이므로, assert 에러 메시지가 뜸
    return 0;
}
```

-> 테스트를 위한 메인 함수 생성

실행결과

```
jinho$ gcc -g my_assert.c -o my_assert
jinho$ ./my_assert
Assertion failed: x != 0, file my_assert.c, line 16
Aborted (core dumped)
```

-> Assertion failed... : my_assert가 출력한 에러 메시지(조건, 파일명, 줄번호)

-> Aborted (core dumped) : assert와 같이 abort()로 강제 종료됨을 표시

5장 14번 예제

```
11 main(int argc, char *argv[])
12 {
13    FILE *f;
14
15    if (argc < 2) {
16      printf("Usage: perror use file name\n");
17      exit(1);
18    }
19
20    if ( (f = fopen(argv[1], "r")) == NULL ) {
21      perror("fopen");    // perror(NULL)로 대체하여 실행해 보자.
22      exit(1);
23    }
24
25    printf("Open a file \"%s\".\n", argv[1]);
26
27    fclose(f);
28 }
```

-> 결과

```
[linux@seps ch5]$ perror_use nofilename
fopen: No such file or directory
```

assert함수

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


void my_assert(int expression, const char *expr_str, const char *file, int line) {
    if (!expression) {
        fprintf(stderr, "Assertion failed: %s, file %s, line %d\n", expr_str, file, line);
        abort();
    }
}

#define MY_ASSERT(expr) my_assert((expr), #expr, __FILE__, __LINE__)

int main(int argc, char *argv[]) {
    FILE *f;

    MY_ASSERT(argc >= 2);  // assert 대체

    if ((f = fopen(argv[1], "r")) == NULL) {
        perror("fopen");
        exit(1);
    }

    printf("Open a file \"%s\".\n", argv[1]);
    fclose(f);
}
```

-> 테스트를 위해 txt파일 생성 => echo "hello world" > test.txt

인자 없이 테스트

```
jinho$ ./use_myassert
Assertion failed: argc >= 2, file use_myassert.c, line 17
Aborted (core dumped)
```

-> 인자가 없으므로 assert 조건이 거짓, my_assert 함수가 assert처럼 오류 메시지와 프로그램 종료를 처리함.

존재하지 않는 파일 실행 - perror 에러 메시지 확인 케이스

```
jinho$ ./use_myassert nofile.txt
fopen: No such file or directory
```

-> 입력받은 파일명을 찾지 못하므로 fopen이 실패하여 perror에서 해당 오류 메시지가 출력됨.

존재하는 파일 실행 - 정상 동작 케이스

```
jinho$ ./use_myassert test.txt
Open a file "test.txt".
```

-> 파일 내용을 읽는게 아니라 파일명을 읽어주는 방식으로 코드를 구성하여 파일명을 출력

11. 위 6-7번 실습문제의 프로그램에 대하여 gprof 프로파일링과 Valgrind 메모리 누수 디버깅을 적용하여 보시오.

gprof 프로파일링 6번 적용

```
jinho$ gcc -pg main.c arithmetic.c -o arithmetic
jinho$ ls
arithmetic  arithmetic.c  arithmetic.h  gmon.out  main.c
```

-> 컴파일하여 gmon.out 파일을 만들어준다

```
jinho$ cat gprof_arithmetic.txt
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00        1     0.00     0.00  add
  0.00      0.00     0.00        1     0.00     0.00  divide
  0.00      0.00     0.00        1     0.00     0.00  multiply
  0.00      0.00     0.00        1     0.00     0.00  subtract
```

-> cat gprof_arithmetic.txt 명령어 실행

gprof 프로파일링 7번 적용

```
CC = gcc
AR = ar
CFLAGS = -I./libsrc -pg
LIBDIR = ./libsrc
LIBSRC = $(LIBDIR)/arithmetic.c
LIBOBJ = $(LIBDIR)/arithmetic.o
LIBSTATIC = $(LIBDIR)/libarithmetic.a
LIBSHARED = $(LIBDIR)/libarithmetic.so
```

-> 기존 makefile의 CFLAGS를 위의 내용처럼 바꾼다. -pg 추가(pg추가하면 gprof 적용된 것)

gprog 적용 make clean

```
jinho$ make clean
rm -f ./libsrc/*.o ./libsrc/*.a ./libsrc/*.so main_static main_shared
```

grof 적용 make static

```
jinho$ make static
gcc -c ./libsrc/arithmetic.c -o ./libsrc/arithmetic.o
ar rcs ./libsrc/libarithmetic.a ./libsrc/arithmetic.o
gcc main.c -L./libsrc -larithmetic -I./libsrc -pg -o main_static
```

grof 적용 ./main_static -> 정적실행

```
jinho$ ./main_static
add: 12
subtract: 8
multiply: 20
divide: 5
```

cat 명령어로 gprof 분석 결과 출력

```
cat gprof_main_static.txt
add: 12
subtract: 8
multiply: 20
divide: 5
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00        1     0.00     0.00  add
  0.00      0.00     0.00        1     0.00     0.00  divide
  0.00      0.00     0.00        1     0.00     0.00  multiply
  0.00      0.00     0.00        1     0.00     0.00  subtract
```

Valgrind를 이용한 실습 6번 메모리 누수 디버깅

-> valgrind --leak-check=full ./arithmetic

```
jinho$ valgrind --leak-check=full ./arithmetic
==978== Memcheck, a memory error detector
==978== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==978== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==978== Command: ./arithmetic
==978==
add: 12
subtract: 8
multiply: 20
divide: 5
==978==
==978== HEAP SUMMARY:
==978==     in use at exit: 0 bytes in 0 blocks
==978==   total heap usage: 2 allocs, 2 frees, 9,648 bytes allocated
==978==
==978== All heap blocks were freed -- no leaks are possible
==978==
==978== For lists of detected and suppressed errors, rerun with: -s
==978== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

=> 사진은 valgrind 명령어의 실행 결과로, 프로그램에 메모리 누수가 없음을 확인할 수 있다.

Valgrind를 이용한 실습 7번 메모리 누수 디버깅

```
jinho$ make clean
rm -f ./libsrc/*.o ./libsrc/*.a ./libsrc/*.so main_static main_shared
```

```
jinho$ make static
gcc -c ./libsrc/arithmetic.c -o ./libsrc/arithmetic.o
ar rcs ./libsrc/libarithmetic.a ./libsrc/arithmetic.o
gcc main.c -L./libsrc -larithmetic -I./libsrc -pg -o main_static
```

-> clean과 static해준다.

valgrind –leak-check=full ./main_static 명령어 실행

```
jinho$ valgrind --leak-check=full ./main_static
==1003== Memcheck, a memory error detector
==1003== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1003== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==1003== Command: ./main_static
==1003==
add: 12
subtract: 8
multiply: 20
divide: 5
==1003==
==1003== Process terminating with default action of signal 27 (SIGPROF)
==1003==    at 0x497BA8A: __open_nocancel (open64_nocancel.c:39)
==1003==    by 0x498A5DF: write_gmon (gmon.c:370)
==1003==    by 0x498AE4E: _mcleanup (gmon.c:444)
==1003==    by 0x48A7494: __run_exit_handlers (exit.c:113)
==1003==    by 0x48A760F: exit (exit.c:143)
==1003==    by 0x488BD96: (below main) (libc_start_call_main.h:74)
==1003==
==1003== HEAP SUMMARY:
==1003==     in use at exit: 9,576 bytes in 2 blocks
==1003==   total heap usage: 2 allocs, 0 frees, 9,576 bytes allocated
==1003==
==1003== LEAK SUMMARY:
==1003==    definitely lost: 0 bytes in 0 blocks
==1003==    indirectly lost: 0 bytes in 0 blocks
==1003==      possibly lost: 0 bytes in 0 blocks
==1003==    still reachable: 9,576 bytes in 2 blocks
==1003==         suppressed: 0 bytes in 0 blocks
==1003== Reachable blocks (those to which a pointer was found) are not shown.
```

```
==1003== Reachable blocks (those to which a pointer was found) are not shown.
==1003== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==1003==
==1003== For lists of detected and suppressed errors, rerun with: -s
==1003== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
Profiling timer expired
```

-> 프로그램이 종료 시점에 2개의 메모리 블록에 총 9,576 바이트가 남아있음

-> LEAR SUMMARY 부분의 definitely lost: 0 bytes" → 명확한 메모리 누수 없음!

==1003==의 마지막줄 ERRORR SUMMARY라인을 통해 런타임 메모리 오류 없음을 알 수 있다.

<검토>

이번 실습을 통해 리눅스 환경에서의 명령어 사용, 컴파일 과정, 디버깅 및 성능 분석 도구 활용에 대해 실제로 경험할 수 있었습니다.

1. 명령어와 컴파일 과정의 이해

처음에는 gcc 명령어로 소스코드를 직접 컴파일하며, -g(디버깅 정보), -pg(프로파일링 정보) 옵션의 역할을 명확히 알게 되었습니다. Makefile을 활용해 여러 파일과 라이브러리를 자동으로 빌드하는 과정에서, 경로 설정과 옵션 적용이 얼마나 중요한지 체감했습니다. 특히, 컴파일 옵션이 제대로 적용되지 않으면 gprof나 Valgrind 같은 도구가 정상적으로 동작하지 않는다는 점을 실습을 통해 직접 확인할 수 있었습니다.

2. 디버깅 및 성능 분석 도구 활용

gprof를 사용해 함수별 실행 시간과 호출 횟수를 분석하는 방법을 익혔고, 실제로 Flat profile 결과를 보고 프로그램의 구조와 성능을 파악할 수 있었습니다. Valgrind를 통해서는 메모리 누수 여부를 자동으로 검사할 수 있었는데, "definitely lost: 0 bytes"와 "ERROR SUMMARY: 0 errors"라는 결과를 통해 코드의 메모리 안전성을 확인할 수 있었습니다. 실습 중에 gprof와 Valgrind가 내부적으로 사용하는 메모리(예: still reachable)가 남아있을 수 있다는 점도 알게 되어, 도구의 결과를 해석하는 능력이 중요함을 느꼈습니다.

3. 실습을 통해 느낀 점

명령어 하나하나의 의미와, 컴파일·디버깅 옵션의 중요성을 직접 체험하면서, 실제 개발 환경에서 이런 도구들이 얼마나 필수적인지 깨달았습니다. 실습 과정에서 발생한 에러(예: 파일 경로 오류, 옵션 누락 등)를 해결하는 과정이 반복되면서, 문제 해결력과 리눅스 명령어 숙련도가 크게 향상되었습니다. 앞으로 더 복잡한 프로젝트에서도 Makefile, gprof, Valgrind 등 도구를 적극적으로 활용해 코드의 성능과 안정성을 높일 수 있을 것 같습니다.