시스템 프로그래밍 (System Programming)

0. 개요

- 운영체제 과목와 유사한 내용을 다루는 실습 위주의 과목
- 리눅스 OS를 통해 진행
 ㄴ (1) 인터넷 서비스 서버 환경이자, 현대 운영체제의 시초격인 UNIX 계열 OS를 익힌다.
 ㄴ (2) 고급 운영체제 기능을 직접 제어해보기 위함
- 선수과목: C언어, 자료구조, 컴퓨터구조, 운영체제에 대한 기초적인 이해

1. 리눅스의 역사와 철학

1) 리눅스(LINUX)
- 서버(+클라우드 컴퓨팅)에 많이 사용되는 운영체제
- plain하게 프로그래밍이 가능
 ㄴ ANSI C는 C언어 표준이 된다.
- 리누스 토발즈(Linus Torvals)가 개발
 ㄴ 대학에 있는 UNIX 컴퓨터를 집에서 사용하기 위해 모방하여 만듦
 ㄴ 다중 사용자, 다중 작업(시분할 시스템, 멀티태스킹)을 지원하는 유닉스 운영체제 지향

* 유닉스 계열 운영체제에서는 'VI' 에디터를 주로 사용한다. 최근에는 약간 개선된 VIM(VI iMproved)도 사용된다.

2) GNU 프로젝트 (Gnu is Not Unix)
- 리차드 스톨만: 초기 컴퓨터 개발 공동체의 상호협력적 문화로 돌아갈 것을 주장
  -> 1985년 GNU 선언문 발표
- 유닉스 운영체제를 대체할 수 있는 운영체제를 만들고 소스를 모두 오픈하겠다는 프로젝트
 ㄴ 70년대 유닉스 운영체제를 여러 회사에서 각자 개발, 소스를 공유하지 않음
- GNU 프로젝트 지원을 위해 자유 소프트웨어 재단(FSF) 설립, GNU 공개 라이선스(GPL) 규약 제공
 ㄴ GPL: 상업적 이용을 포함한 자유로운 수정과 배포가 가능하나, 사용 시 해당 프로젝트에 GNU 라이선스 적용 및 소스 코드 공개 의무.
- 운영체제 커널 'GNU Hurd' 개발 시도
 -> 라이브러리, 컴파일러, 에디터, 쉘 등 운영체제에 필요한 요소들도 개발됨
 -> 정작 GNU Hurd의 개발은 지연
 -> 리누스 토발즈가 리눅스 커널 소스를 오픈함
 -> 리차드 스톨만이 리누스 토발즈에게 GNU 프로젝트 합류 요청

* 스톨만은 리눅스를 'GNU/Linux' 라고 부르기를 희망했다고 한다.

2. 리눅스 설치하기

1) 클라우드 컴퓨팅을 통한 설치
- 클라우드 컴퓨팅: 서버 환경을 미리 구축해놓고, 간단한 설정으로 바로 사용할 수 있도록 만든 서비스
- 리눅스 설치 방법은 크게 가상 머신과 클라우드 컴퓨팅을 활용하는 방법으로 나뉜다.
- 가상 머신은 PC 환경에 따라 다를 수 있으니, 여기서는 동일한 환경에서 사용 가능한 후자 방식을 선택한다. (단, 초기 설정은 필요함)
- AWS 기준 설치 절차는 [서버 생성 -> 고정 아이피 할당 -> 서버 접속] 순으로 진행

2) AWS 서버 생성
(1) https://aws.amazon.com/ko/free/ 접속해서 무료 플랜으로 가입
(2) 로그인 후 우측 상단에서 지역을 서울로 설정
(3) EC2 생성
- EC2: 인스턴스라고도 부르며, AWS에서 서버를 지칭하는 용어를 의미한다
- 상단 검색창에 EC2 검색하여 클릭, 좌측 EC2 대시보드에서 인스턴스 클릭
- 인스턴스 시작 클릭
(4) 리눅스 배포판 선택
- 리눅스는 윈도우처럼 정형화된 프로그램이 없고, 에디터나 쉘, API, 라이브러리 등이 다양하게 존재한다.
- 모든 요소를 개별적으로 설치하기 불편하니, 특정 목적(일반사용자, 서버용, 클라우드용, 개발자용 등등)으로 사용하기 편하게 모아둔 것을 배포판(패키지)라고 한다.
ㄴ 유명한 배포판 리눅스로 우분투(Ubuntu), 레드 햇(Red Hat) 등이 있다. 일반 사용자용인 Ubuntu Desktop은 GUI 환경으로 되어 있다.
- 여기서는 'Ubuntu Server 18.04 LTS(HVM)'를 사용하기로 한다.
(5) 세부 설정
- 인스턴스 유형: 컴퓨터의 사양을 선택한다. 무료 사용 가능한 버전을 선택.
- 인스턴스 세부 정보: 별도로 변경 없이 다음으로 넘어간다.
- 스토리지 추가: 무료 제공 최대값인 30GB로 변경한다.
- 태그 추가: 컴퓨터가 여러 대일 때 이름을 붙이는 용도. 1대만 쓸 것이니 넘어간다.
- 보안 그룹 구성: 서버 접속 방법 선택. 기본값(SSH-TCP-22번 포트)으로 둔다.
- 인스턴스 시작 검토: 보안 경고는 무시하고 시작을 누른다.
(6) 키 페어 생성
- 서버 관리를 위한 인증 도구를 생성한다. 일종의 공인인증서.
- '새 키 페어 생성'을 선택하고 원하는 이름을 입력 후 키 페어 다운로드를 클릭
- 다운로드된 pem 파일은 소중하게 저장해 두고, '인스턴스 시작' 클릭
- 이후 홈 화면에서 EC2 - 인스턴스 클릭 시 자신의 서버가 동작하는 것을 확인할 수 있다.

3) Elastic IP를 통해 서버 접속
- Elastic IP: IP가 재할당되지 않도록 고정 아이피를 생성하여 할당하는 것

ㄴ 매번 아이피 주소가 달라져 접속이 어렵거나, 이로 인한 추가 비용 청구를 방지한다.
(1) EC2 좌측 메뉴에서 '탄력적 IP' 선택
(2) 탄력적 IP 주소 할당 -> 할당 버튼을 눌러 아이피를 할당받는다.
(3) 할당된 주소를 자신의 서버 인스턴스에 연결한다.
- 주소 할당 버튼 옆의 작업 버튼 -> 탄력적 IP 주소 연결 클릭
- 인스턴스는 위에서 만든 자신의 인스턴스를 선택한다. 다른 옵션은 건드릴 필요 없음.
- '할당된 IPv4 주소'를 확인한다. 이를 통해 서버에 접속할 수 있다.
(4-1) Mac 환경에서 접속하기
- 터미널에서 해당 파일 디렉토리로 들어간다.
- 'chmod 400 파일명' 입력 (해당 파일 접근을 소유자로 제한함)
- 'ssh -i 파일명 ubuntu@ip주소' 입력 -> yes 입력 -> 접속 완료
(4-2) 윈도우 환경에서 접속하기 (PuTTY)
- putty를 다운로드한다. 운영체제에 맞게(일반적으로 64bit x86) 선택한다.
- 이후 (putty가 아닌) PuTTYgen을 실행, Load에서 pem파일을 찾아 선택한다.
ㄴ 파라미터가 RSA, 비트 수는 2048임을 확인한다. 아니라면 바꿔준다.
- Save private key를 눌러 원하는 위치에 ppk 파일을 저장한다.
ㄴ pem 파일은 윈도우에서 작동하지 않으므로 ppk 파일로 변환해야하기 때문.
- PuTTY를 실행하고 Session의 Host Name에 'ubuntu@ip주소' 입력
- 좌측 메뉴에서 Connection - SSH - Auth 선택하고, 맨 아래 Private key file~에 Browse를 눌러 위에서 변환한 ppk 파일을 선택한다.
- Session으로 돌아와 이름을 붙여 Save하면 다음부터는 해당 세팅값을 Load해서 사용할 수 있다. (Port 22, Connection Type은 SSH인 것 확인)
- Open 버튼을 눌러 실행한다. (최초 실행 시 동의 화면이 뜨면 '예'를 눌러 넘어감)
(4-3) 윈도우 PowerShell로 접속하기
- PowerShell을 실행한다.
- pem 파일이 있는 경로(한글 디렉토리명 X)에서 'ssh -i 파일명 ubuntu@ip주소' 입력

4) 가상 머신을 통한 설치
- 클라우드를 통한 방식 외에, 가상 머신 소프트웨어를 통해 설치하는 방법은 아래와 같다.
(1) 우분투 설치
- Desktop을 설치해도 무관하나 여기선 Server 권장
(2) VMware 등 가상 머신 소프트웨어 설치
- 윈도우는 VMware Workstation Player, Mac은 VMware Fusion 등이 있다.
(3) 가상 머신의 사양을 설정하고 실행하여, 가상 머신에 우분투를 설치

3. 리눅스 기본 구성
- 유닉스 계열에 공통으로 적용된다.

1) 파일
- 모든 인터렉션은 파일을 읽고 쓰는 것처럼 이루어져 있다.

- 모든 자원에 대한 추상화 인터페이스로 파일 인터페이스를 활용한다.
 ㄴ 마우스, 키보드 등 모든 디바이스 관련 기술도 파일처럼 다루어진다.
 ㄴ 어떤 디바이스를 다루든 추상화된 input / output을 생각해보는 게 좋다.
- 전역 네임스페이스 제공
 ㄴ A드라이브, C드라이브 등으로 나누지 않고, 하나의 최상단 디렉토리가 존재한다.
 ㄴ 별도의 저장매체(외장하드 등)도 최상단 디렉토리 내의 주소로 접근한다.
- inode 고유값과 자료구조로 파일의 메타데이터가 구성되어, 주요 정보를 관리한다.

2) 프로세스
- 여러 프로그램이 각자의 일을 수행하면서 전체 시스템이 유기적으로 동작하는 모델
 ㄴ 아무 작업이 없어도 기본적으로 다양한 프로세스가 실행되고 있다.
- 리눅스의 실행 파일 포맷은 'ELF'
- 프로세스에서 시스템 콜 호출을 통해 리소스 처리 가능하도록 구성
 ㄴ 타이머, 시그널, IPC기법 등 시스템 콜 기반의 다양한 시스템 리소스 처리가 가능하다.
- 가상 메모리 지원
- 각 프로세스는 프로세스 ID(pid) 고유값으로 구분
- 최초 실행 시 init 프로세스가 구동되고, 그 다음부터는 fork() 라는 시스템 콜을 사용해서 신규 프로세스가 생성된다.

3) 권한
- 슈퍼 관리자인 root가 있음
- 사용자 권한 관리: 사용자를 그룹으로 묶어, 개별 사용자 및 그룹에 대해 권한 관리
- 리소스 권한 관리: 파일마다 소유자, 소유자가 속한 그룹, 모든 사용자의 3가지로 구분하여 읽기, 쓰기, 실행의 3가지 권한 관리
 ㄴ 접근 권한에 대한 정보는 inode의 자료구조에 저장된다.

4) 쉘
- 쉘: 사용자와 컴퓨터 간의 인터페이스. 사용자 명령을 해석하여 커널에 명령을 요청하며, 시스템콜을 사용해서 프로그래밍이 작성되어 있다.
- Bourne-Again Shell(bash): GNU 프로젝트의 일환으로 개발. 리눅스에서 디폴트로 사용.
- Korn Shell(ksh): 유닉스에서 가장 많이 사용됨.
- Bourne Shell(sh), C Shell(csh) 등도 사용한다.
- 쉘마다 명령어가 조금씩 다르며, 리눅스 명령어는 리눅스 기본 쉘인 bash의 명령어이다.

4. 리눅스 기초 명령어
- bash에서 사용되는 명령어에 해당한다.

1) 기본
- sudo (명령어): 다른 명령어 앞에 붙여 쓴다. 해당 명령을 관리자 권한으로 실행한다.
 ㄴ /etc/sudoers 파일에서 sudo 명령어 사용 권한을 설정할 수 있다.

- ls: 현재 디렉토리에 있는 요소들을 보여준다.
 ㄴ ls -al 옵션 사용 시 숨김 파일도 보여준다. 숨겨진 파일은 파일명이 . 로 시작한다.
- pwd: 현재 디렉토리를 보여준다.
- cd (디렉토리): 해당 디렉토리로 이동한다.
 ㄴ '.' 으로 시작하면 상위 디렉토리를 의미한다. './' 는 현재 위치한 디렉토리, '../' 은 현재 디렉토리가 위치한 상위 디렉토리를 의미한다.
- man (명령어): 해당 명령어의 메뉴얼을 보여준다. 옵션이 궁금할 때 사용한다.

2) 사용자 관련
- whoami: 현재 사용자 ID를 알려준다.
- passwd: 현재 사용자 ID의 암호를 변경한다.
- useradd (아이디): 새로운 사용자 ID를 만든다.
 ㄴ adduser (아이디): 위와 같으나, 사용자 기본 설정(홈 디렉토리 생성 등)을 자동으로 수행한다. 명령어 앞에 sudo를 붙여 관리자 권한으로 실행할 것.
- su (아이디): 해당 사용자로 전환한다. 현재 사용자의 환경설정을 사용한다.
 ㄴ su - (아이디): 위와 같으나, 변경되는 사용자의 환경설정을 기반으로 전환한다.

3) 파일 관련
- ls -al: 숨김 파일을 포함한 모든 파일의 상세정보를 출력한다.
 ㄴ 첫 문자는 파일이면 '-', 디렉토리면 'd'
 ㄴ 다음 9개 문자는 소유자 권한 3글자, 소유자 그룹 권한 3글자, 모든 사용자 권한 3글자
 ㄴ 3글자는 rwx 로 구성되며 각각 읽기, 쓰기, 실행 권한을 의미한다. 해당 자리가 '-' 이면 권한이 없거나 불가능한 것. (참고 링크)
- chmod (설정값) (파일명): 해당 파일의 권한을 변경한다. (change mod)
 ㄴ 기호 사용: 대상-동작-권한 종류 순으로 적는다. g+rx는 그룹 사용자에게 읽기와 실행 권한을 추가한다는 의미, o=rw 는 모든 사용자에게 권한을 읽기와 쓰기만 가능하게 설정.
  ㄴ 대상: u(사용자), g(그룹), o(모든 사용자), a(전부)
  ㄴ 동작: +(권한 추가), -(권한 해제), =(권한 설정)
  ㄴ 권한 종류: r(읽기), w(쓰기), x(실행)
  ㄴ u=rmx, g=rw, o=r 등 콤마(,)를 이용해 여러 권한 동시에 설정도 가능하다
 ㄴ 숫자 사용: 권한을 3자리 숫자로 입력한다. 각 자리수는 소유자/그룹/전체 권한을 의미.
  ㄴ 각 숫자는 대상에 대한 rwx 권한을 이진수로 나타낸 것이다.
  ㄴ 7은 이진수로 111이므로, rwx 모두 가능, 0은 모두 불가
  ㄴ 764는 소유자는 7(rwx), 그룹은 6(rw-), 기타 사용자는 4(r--) 권한 부여
 ㄴ chmod -R (설정값) (폴더명): 해당 디렉토리와 하위 디렉토리 모든 파일에 권한 설정
- chown (소유자:소유그룹) (파일명): 해당 파일의 소유자를 변경한다.
 ㄴ 위와 같이 -R 옵션으로 디렉토리에 대해서도 설정 가능
- cat (파일명): 해당 파일 내용을 터미널에 출력한다.
 ㄴ head 또는 tail (파일명): 해당 파일의 앞 또는 뒷부분 일부(기본 10줄)만 볼 수 있다.
 ㄴ more (파일명): 해당 파일을 터미널 한 페이지만큼만 출력하고, 엔터(1줄)나 스페이스(1페이지)로 더 볼 수 있다. q를 눌러 빠져나올 수 있다.

- rm (파일명): 파일 또는 폴더 삭제. 주로 -rf 옵션과 함께 사용한다.
 ㄴ -r 옵션: 하위 디렉토리를 포함한 모든 파일 삭제
 ㄴ -f 옵션: (경고나 확인 없이) 즉시 삭제한다. 리눅스는 휴지통이 없으니 주의.
- grep (패턴) (파일 또는 폴더명): 해당 파일(폴더)에서 패턴이 들어간 내용(파일)을 모두 찾아 출력한다.
 ㄴ -i(대소문자 무관), -v(해당 패턴 미포함만 검색), -n(번호 매겨서 출력), -l(파일명만 출력), -c(일치하는 라인 개수만 검색), -r(하위 디렉토리까지 검색) 등의 옵션이 있다.
 ㄴ 패턴을 "패턴1|패턴2|패턴3"과 같이 적어서 해당 패턴 중 하나 이상이 있는 경우를 모두 찾아낼 수 있다.

5. 리눅스 쉘 프로그래밍

1) 표준 입출력 (standard stream)
- 커맨드로 입력되는 프로세스는 3가지 스트림을 가진다.
 ㄴ 표준 입력 스트림, 표준 출력 스트림, 오류 출력 스트림
 ㄴ Standard Input/Output/Error Stream의 줄임말로, stdin, stdout, stderr라고 한다.
- 모든 스트림은 일반적인 plain text로 콘솔에 입력 및 출력하게 되어 있다.

2) 리다이렉션 (redirection)
- 표준 스트림의 흐름을 바꿔줄 수 있다.
 ㄴ 예를 들어, 명령어 표준 출력을 콘솔 화면 대신 파일에 쓸 때 사용
- 입력 또는 출력에 '<', '>' 기호를 사용해서 제어할 수 있다.
 ㄴ ex) ls -al > file.txt 입력 시, 출력될 내용이 콘솔 대신 file.txt 파일로 저장된다.
 ㄴ ex) head < file.txt 입력 시, file.txt 파일 내용 앞부분이 콘솔에 출력된다.
- 스트림은 앞에서부터 순서대로 동작한다.
 ㄴ ex) head < file.txt > file2.txt 입력 시, file.txt 내용을 head에 출력해야 하는데, 다시 이 출력될 내용을 file2.txt에 저장한다.
- 파일을 덮어쓰지 않고, 기존 파일에 추가하고 싶을 때는 '<<', '>>' 를 사용한다.

3) 파이프 (pipe)
- 한 프로세스의 출력 스트림을 다른 프로세스의 입력 스트림으로 사용할 때 쓴다.
- 명령어 사이에 '|'를 넣어 사용한다.
 ㄴ ex) ls | grep jiho 입력 시, ls로 현재 디렉토리 파일을 모두 출력한 후, 이 출력한 내용을 grep로 보내서 그 중에 jiho 키워드가 들어 있는 파일만 출력한다.

4) 포그라운드와 백그라운드
- 포그라운드 프로세스: 쉘에서 명령을 실행한 후, 해당 프로세스 종료 전까지 다른 입력이 불가능한 프로세스
- 백그라운드 프로세스: 사용자 입력과 상관없이 실행되는 프로세스
 ㄴ 쉘에서 프로세스 실행 시, 맨 뒤에 '&'를 붙이면 백그라운드 프로세스로 실행한다.

4-1) 프로세스 관련 명령어
- 기본적으로 포그라운드 프로세스는 작동중에 다른 명령어를 받지 않으므로, 포그라운드 프로세스는 인터럽트에 의해 제어된다.
- Ctrl + C: 포그라운드 프로세스를 강제 종료시킨다.
- Ctrl + Z: 프로세스를 실행 중지 상태로 변경하고 백그라운드 프로세스로 이동시킨다.
- jobs: 백그라운드 프로세스들의 목록과 상태를 볼 수 있다.
- bg (번호): 해당 번호의 백그라운드 프로세스를 실행시킨다. 번호는 jobs 명령어로 확인할 수 있으며, 번호를 입력하지 않으면 최근에 중단된 프로세스를 실행시킨다.
- ps (옵션): 프로세스 상태를 확인할 수 있다. (자세한 옵션 정리)
 ㄴ -a: 모든 사용자의 프로세스 출력 (보통 u,x 옵션과 함께 '-aux' 옵션이 많이 사용된다)
 ㄴ u: 프로세스 소유자에 대한 상세 정보 출력
 ㄴ -l: 프로세스 관련 상세 정보 출력 (-f보다 더 많은 정보를 보여준다)
 ㄴ x: 로그인 후 실행한 프로세스가 아닌 것들까지 출력 (시스템 부팅 시 자동 실행되는 데몬 프로세스들도 출력한다)
 ㄴ -e: 커널을 제외한 시스템상의 모든 프로세스 정보를 출력
 ㄴ -f: 풀 포맷. 프로세스 간 관계 정보까지 출력 (소유자 이름, 부모 프로세스 ID 등)
 ㄴ 상세 정보에서 VSZ는 가상 메모리 크기, RSS는 사용중인 실제 메모리 크기를 의미한다.
- kill (PID): 해당 프로세스 ID를 가진 프로세스를 종료한다. PID는 위의 ps로 확인 가능.
 ㄴ -9: 강제로 종료시킨다. kill에서 거의 항상 사용되는 옵션.

5) 파일 시스템
- 복잡한 기능/자료/데이터 등에서 핵심적 기능을 input/output을 가진 파일 형태로 '추상화(abstraction)'한 시스템
- 연결된 모든 디바이스도 파일 시스템으로 관리되므로, 전역 네임스페이스 안에 위치한다.
 ㄴ /dev 폴더에서 확인 가능. tty는 가상 터미널을 의미한다.
 ㄴ 터미널에 명령어를 입력하면 키보드 -> 가상 터미널 환경 -> 가상 파일 시스템 인터페이스 -> tty에 input이 들어가는 형태로 파일 입출력처럼 실행된다.
- 각 파일은 inode 기반 메타 데이터로 관리된다.
 ㄴ 권한, 소유자, 크기, 생성시간, 위치 등의 정보가 메타데이터로, inode 번호에 매치되어 있다.
- 파일 탐색은 각 디렉토리의 엔트리(dentry, directory entry)를 탐색하여 이루어진다.
 ㄴ dentry는 해당 디렉토리 내의 서브 디렉토리와 파일들의 정보(inode)를 가지고 있다.
 ㄴ ex) '/home/ubuntu/file.txt' 탐색 시 '/' dentry에서 'home'을 찾고, 'home'에서 'ubuntu'를 찾고, 'ubuntu'에서 'file.txt' 파일이름에 해당하는 inode를 얻는다.

5-1) 리눅스의 기본 디렉토리들
 - 루트 디렉토리는 '/' 이다.
 - 로그인 시 처음 위치하는 디렉토리는 '/home/사용자ID/' 이다.
 - 컴퓨터에 추가로 연결되는 저장장치는 /media/ 또는 /mnt/ 디렉토리의 하부에 위치하는 경우가 많다.

- /ect/ 폴더엔 다양한 설정 파일이 들어간다.
- /dev/ 폴더엔 마우스, 키보드, 터미널 등의 디바이스들이 파일 형태로 매핑되어 있다.
- /bin/ 과 /sbin/ 폴더엔 쉘 명령어와 실행 파일들이 들어 있다.

5-2) 특수 파일
- 블록 디바이스: HDD, CD, DVD 등 블록 또는 섹터 등 정해진 단위로 데이터 전송. IO 송수신 속도가 높다.
- 캐릭터 디바이스: 키보드, 마우스 등 바이트 단위로 데이터 전솔, IO 송수신 속도가 낮다.
- 파일 상세정보에서 앞자리가 d면 디렉토리를 의미하듯, 블록 디바이스는 b, 캐릭터 디바이스는 c로 시작한다.

6) 링크
- 동일한 파일을 하나 더 생성하는 복사 외에, 링크 기능을 지원한다.
- 하드 링크: 파일명은 다르지만 아이노드 번호가 동일한 파일을 생성한다. 즉 생성된 파일은 동일한 데이터를 가리킨다.
 ㄴ 해당 데이터를 가리키는 포인터가 하나 늘어난다.
 ㄴ 원본 파일을 삭제해도 데이터를 가리키는 하드 링크가 있으면 데이터가 삭제되지 않는다. 즉 원본 파일 삭제 시에도 원본의 inode 데이터만 삭제되고, 하드 링크로 파일을 열 수 있다.
- 소프트 링크: 기존 파일의 inode 구조체를 가리키는 파일을 생성한다. 생성된 파일은 실제 데이터를 가리키지 않고 원본 파일로 리다이렉션 되며, 아이노드 데이터도 원본과 다르다.
 ㄴ 윈도우의 '바로 가기' 생성과 동일하다.
 ㄴ 원본 파일이 삭제될 경우 소프트 링크된 파일은 사용할 수 없다.

6-1) 파일 복사 관련 명령어
- cp (파일/폴더명) (새 이름): 앞의 파일/폴더를 지정한 이름으로 현재 디렉토리에 복사한다.
 ㄴ -r: 폴더 복사 시 하위 디렉토리와 파일들까지 모두 복사한다.
 ㄴ -f: 확인 질문 없이 바로 복사한다.
- ln (파일명) (새 이름): 앞의 파일에 대해 지정한 이름의 하드 링크를 생성한다.
 ㄴ -s 옵션 사용 시 소프트 링크
- ls -i: 디렉토리 내 파일들의 아이노드 번호까지 출력한다.

6. 사용자 영역에서의 시스템 프로그래밍

1) 시스템 콜
- 리눅스는 커널 영역과 시스템 콜, 라이브러리(API)가 모두 C 언어로 만들어져 있다.
 ㄴ 유닉스 C 라이브러리: libc
 ㄴ 리눅스 C 라이브러리: GNU libc (glibc - 지립씨로 읽음)
- 시스템 프로그래밍은 커널 영역을 제어해야 하므로, 사용자 영역에서 접근하려면 반드시 시스템 콜을 사용해야 한다.

2) C 컴파일러
- 리눅스 C 컴파일러는 gcc(GNU C Compiler)를 사용한다.
 ㄴ 유닉스는 cc(C Compiler)
 ㄴ 설치 명령어: sudo apt-get install gcc
 ㄴ 버전 확인: gcc --version
 ㄴ 컴파일 방법: gcc (소스코드 파일명) -o (저장할 파일명)
  ㄴ ex) gcc test.c -o myfile
  ㄴ 저장할 파일명을 적지 않으면 a.out 으로 나온다.

3) ABI (Application Binary Interface)
- 응용 프로그램 바이너리 인터페이스
- 시스템마다 함수 실행 방식, 레지스터 활용, 시스템 콜, 라이브러리 링크 방식 등이 다르다. 실행 파일(바이너리)의 실행 방식을 결정하는 인터페이스인 ABI가 필요하다.
 ㄴ ex) 메모리의 0번 주소를 stack영역부터 시작할 것인가, code 영역부터 시작할 것인가?
- 툴체인(컴파일러 제작 프로그램)에서 제공하여 컴파일러에 정의되어 있다.
- ABI가 동일한 시스템에선 프로그램이 정상적으로 실행되지만, ABI가 다를 경우 해당 시스템에 맞게 다시 컴파일(+링크)해야 한다.

4) 표준화
- POSIX: 유닉스 프로그래밍 인터페이스 표준
 ㄴ IEEE에서 표준화 시도, 리차드 스톨만이 표준안 이름으로 POSIX 제안
- ANSI C: 다양한 C언어 변종이 존재하여 ANSI라는 단체에서 C 표준을 정립한 것
- 리눅스는 POSIX와 ANSI C를 지원한다.
- POSIX, C, C++ 및 소프트웨어의 표준은 끊임없이 업데이트되고 있다.
 ㄴ 프로그래밍 트렌드와 상관없이 시스템 레벨 단의 기술은 수십 년간 꾸준히 유지되고 있음

7. 프로세스 관리

1) 프로세스 ID (PID)
- 각 프로세스가 가지는 유니크한 값
 ㄴ 2바이트(16bit)로 구성되는 signed int 형태로 최대값은 32768
- 리눅스에서 프로세스 실행 시 기본적으로 스레드 하나가 포함된다.
 ㄴ 기본적으로 싱글스레드, 강제로 여러 스레드를 띄웠다면 멀티스레드 프로세스가 된다.
- 운영체제가 실행하는 최초 프로세스(init)의 pid는 1이며, 다른 프로세스는 자식 프로세스로서 생성되는 계층 구조를 가지고 있다. (자식 프로세스에서 다른 자식 프로세스 생성 가능)
 ㄴ 부모 프로세스의 pid 값을 ppid (parent pid) 라고 한다.
- 내부적으로 프로세스의 소유자(사용자)와 그룹을 정수로 된 UID/GID로 관리한다.
 ㄴ 보여줄 때는 UID와 사용자 이름의 매핑 정보를 기반으로 사용자 이름을 보여준다.

* getpid() / getppid()

- 이 프로세스의 pid 또는 ppid를 반환한다.
- C언어에서 함수 형태로 사용 가능. (<sys/types.h> <unistd.h> 헤더의 include 필요)

2) 프로세스 생성
- 메모리에 TEXT, DATA(+BSS), HEAP, STACK 공간을 생성하고, 프로세스 이미지를 해당 공간에 업로드한 후 실행을 시작한다.
- 프로세스는 기본적으로 다른 프로세스로부터 생성된다.
- fork(): 새로운 프로세스 공간을 만들고, fork를 호출한 부모 프로세스 공간을 그대로 복사
 ㄴ C에서 fork() 함수는 자식 프로세스일 경우 0, 부모 프로세스일 경우 생성된 자식 프로세스의 pid 값, 실패한 경우 -1을 반환한다. 리턴값을 비교해서 어느 프로세스인지 확인 가능.
 ㄴ 두 프로세스의 PC(Program Count)와 변수값은 동일
 ㄴ fork() 이후 두 프로세스는 다음 코드를 계속 실행한다. 두 프로세스가 다른 동작을 하게 하려면 반환값을 비교하여 분기를 나눠야 한다.
- exec 계열: 시스템 콜을 호출한 현재 프로세스 공간의 TEXT, DATA, BSS 영역을 새로운 프로세스의 이미지로 덮어씌운다. 별도의 프로세스 공간을 만들지 않는다.
 ㄴ execl, execv, execvp 등 여러 가지 시스템 콜을 가지고 있다.
 ㄴ 코드 영역이 덮어씌워지므로 exec 이후의 코드는 실행되지 않고 새 프로세스의 코드가 처음부터 실행된다. 즉 예외 처리는 exec 바로 뒤에 에러 메시지 출력을 넣는 형태가 된다.
 ㄴ execl(): 경로와 매개변수를 입력해서 해당 경로의 실행 파일을 새 프로세스로 실행한다.
 ㄴ execlp(): execl과 달리 현재 프로세스의 주소에서 탐색한다. 같은 디렉토리에 있다면 경로를 생략하고 파일명만 입력해서 실행 가능.
 ㄴ execle(): execlp에 환경 변수를 추가로 넘겨줄 때 사용한다.
 ㄴ execv(), execvp(), execve(): 위의 execl, execlp, execle에서 매개변수를 하나씩 적지 않고 문자열(char*) 형태로 만들어 넣는 버전.
- copy on write: fork할 때 4GB 공간을 모두 복사하면 시간이 오래 걸리니, 자식 프로세스 생성 시 부모 프로세스 페이지를 사용하는 기법. 부모 또는 자식 프로세스가 페이지 읽기가 아닌 쓰기를 요청할 때, 페이지를 복사하고 분리한다.
 ㄴ 프로세스 생성 시간을 줄이고, 새롭게 할당하는 페이지 수도 최소화한다.
 ㄴ 프로세스에 할당되는 메모리는 1GB의 커널 영역과 3GB의 사용자 영역으로 구성된다. 커널 영역도 프로세스마다 가상 메모리와 페이지 테이블이 할당되지만, 실제 메모리에선 여러 프로세스가 공유한다.
 ㄴ 수정 없이 동일하게 사용되는 부분은 가상 메모리의 할당도 최소화하기 위한 방식.

* 리눅스에서 쉘 프로그램을 만들 경우 fork()로 자식 프로세스를 만든 후, 자식 프로세스가 exec 계열 함수로 기능을 실행하는 것이 일반적이다.
- 부모 프로세스가 사라지는 것을 방지하고, 자식 프로세스 종료 후 결과에 따른 처리 가능
- 버퍼에 명령어 입력 받음 -> fork() -> 자식 프로세스에서 exec -> 부모 프로세스는 wait -> 자식 프로세스 동작이 끝나면 다시 다음 명령어 입력 받기 -> 반복

3) 프로세스 대기와 종료
- wait(): fork 이후 사용하면, fork로 생성한 자식 프로세스가 종료될 때까지 대기한다.

ㄴ 자식 프로세스 종료 시 종료 신호를 받으며, 그 후에 wait 다음 동작을 계속 실행한다.
ㄴ 부모-자식 프로세스 동기화, 부모 프로세스가 먼저 죽는 경우(고아 프로세스 또는 좀비 프로세스)를 방지하기 위한 시스템 콜
ㄴ 자식 프로세스는 실행 정보를 메모리에 남기고, 부모 프로세스는 이를 확인하고 종료하는 것이 일반적인 동작. 좀비 프로세스가 되면 불필요한 정보가 메모리에 남아있게 된다.
ㄴ 매개 변수로 int* 를 받아서, 해당 포인터에 자식 프로세스의 종료 상태값이 들어온다. 함수의 반환값은 자식 프로세스의 pid이다.
ㄴ WIFEXITED(int*)에 status를 넣어 자식 프로세스의 정상 종료 여부를 확인할 수 있다. 비정상 종료일 경우 WIFEXITED 함수의 반환값은 0이 된다.
ㄴ 사용 예시: wait(&value) -> WIFEXITED(value) == 0인지 아닌지 확인
- exit(): 프로세스를 즉시 종료한다. exit 함수 이후의 코드는 실행되지 않는다.
ㄴ main 함수에서 return시 내부적으로 _start() -> exit() 함수를 호출하게 된다. 즉 메인 함수의 return 0은 메인 함수 이후의 몇 가지 마무리 동작 외에는 exit(0)과 큰 차이가 없다.
ㄴ 매개변수로 종료 결과를 전달한다. 일반적으로 정상 종료면 0을 전달한다.
ㄴ 주요 동작: atexit()에 등록된 함수 실행, 모든 입출력 스트림 버퍼(stdin/out, stderr 등) 삭제, 프로세스가 오픈한 파일 모두 닫기, tmpfile()을 통해 생성한 임시 파일 삭제
ㄴ atexit(): 종료 시 실행할 함수를 atexit(Func)로 등록하고 정상 등록되면 0을 반환한다. exit가 실행될 때 등록된 함수를 등록한 순서의 역순으로 실행한다.

4) 프로세스 스케쥴링
- 리눅스는 우선순위 기반 스케쥴러를 사용한다.
- nice(): 우선순위를 변경할 수 있다. 매개변수로 우선순위 값(int)을 받는다.
ㄴ root가 소유한 프로세스는 우선순위를 높일 수 있고, 나머지는 원래보다 우선순위를 낮출 수만 있다.
ㄴ 스케쥴링 방식에 따라 우선순위가 적용될 수도, 안 될 수도 있다.
- getpriority(): 현재 프로세스 또는 이 프로세스와 관련된 우선순위 값을 반환한다.
- setpriority(): 현재 프로세스 또는 이 프로세스와 관련된 우선순위 값을 설정한다.
ㄴ 매개변수로 int, id_t의 두 가지를 받는다. set은 설정할 값(int)을 추가로 받는다.
ㄴ 첫 번째 매개변수에 어떤 우선순위를 대상으로 할지 입력한다. PRIO_PROCESS / PRIO_PGRP / PRIO_USER을 입력하면 각각 프로세스 / 프로세스 그룹 / 사용자를 의미한다.
ㄴ 두 번째 매개변수는 프로세스 ID(getpid()로 가져올 수 있음)를 쓰는데, 0을 쓰면 현재 프로세스 또는 현재 프로세스 그룹이 된다.

8. IPC 기법

1) 개요
- 프로세스는 다른 프로세스 공간의 특정 지점을 직접 가리킬 방법이 없다.
- 일반적으로 IPC 기법은 커널 공간이나 파일 등 공유 공간을 사용한다.

2) 파이프

- fork()로 복사한 자식 프로세스에 버퍼의 주소를 넘겨줘서 값을 읽게 한다.

```
// stdio.h, stdlib.h, unistd.h 헤더 필요
char* msg = "전달할 메시지";
char buffer[255];
int fd[2];
// 부모 프로세스일 경우
write(fd[1], msg, 255);
// 자식 프로세스일 경우
int nbytes = read(fd[0], buffer, 255);
```

3) 메시지 큐

- 메시지 큐를 만든다. id값을 통해 서로 다른 프로세스가 접근할 수 있다.
- msgget(key, msgflg): 정수로 된 key값을 넘겨 메시지 큐 id를 생성한다. 생성 실패 시 -1을 반환한다.
  ㄴ msgflg는 메시지 플래그. 새로운 키일 경우 'IPC_CREAT|접근권한'으로 식별자를 생성한다. 예를 들어 IPC_CREAT|0644 는 rw-r--r-- 접근권한을 의미한다.
- msgsnd(id, 버퍼, 버퍼 크기, 옵션): 메시지를 전송한다. 실패 시 -1을 반환한다.
- msgrcv(id, 버퍼, 메시지 크기, msgtyp, msgflg): 메시지를 받는다.
  ㄴ msgtyp는 0이면 첫 번째 메시지, 양수 타입일 경우 타입이 일치하는 첫 번째 메시지.
  ㄴ msgflg: 블록 모드면 0, 비블록 모드면 IPC_NOWAIT. 해당 위치에 데이터가 없을 때, 블록 모드면 데이터를 읽을 때까지 계속 기다리고, 비블록 모드면 다음 코드로 넘어간다. 일반적으로 0으로 설정.

```
// stdlib.h, stdio.h, string.h, sys/msg.h 헤더 필요
typedef struct msgbuf { long type; char text[50]; } Msgbuf;
Msgbuf msg;
key_t key = 1234;
msgid = msgget(key, IPC_CREAT|0644);
if (msgid < 0) exit(1);
// 메시지 전송할 때
msg.type = 1;
strcpy(msg.text, "Hello world!");
msgsnd(msgid, (void*)&msg, 50, IPC_NOWAIT)
// 메시지 받을 때
int len = msgrev(msgid, &msg, 50, 0, 0); // 받은 메시지 길이 반환
printf("Received: [%d] %s", len, msg.text);
```

- 유니크한 key를 생성하려면, ftok()를 이용한다. 파일 또는 폴더 경로와 정수를 입력하여, 둘을 조합한 키 값을 생성한다.
  ㄴ 입력한 파일이나 폴더를 삭제 후 재생성시 inode 값이 달라지므로 key도 달라진다.
- 참고: msgctl(id, 명령, 옵션) 으로 메시지 큐 컨트롤 가능.
  ㄴ msgctl(id, IPC_RMID, 0): 해당 메시지 큐를 삭제한다.

4) 공유 메모리

- 커널 공간에 공유 영역을 만들어 변수처럼 활용한다. 메시지 큐와 비슷하게 id값을 아는 누구나 접근 가능하지만, FIFO 방식 등의 제약 없이 노골적으로 사용하는 방식.
- shmget(key, size, shmflg): 공유 메모리 id를 반환한다.
  ㄴ shmflg 옵션 입력 방식은 msgget 참고.
- shmat(id, addr, shmflg): 공유 메모리를 생성하고 id에 연결한다. 연결된 메모리 공간의 시작 주소를 반환한다. 이후 반환된 주소를 공유 메모리의 변수처럼 사용하면 된다.
  ㄴ addr: 연결 주소. 보통 NULL을 입력하면 알아서 적절한 주소로 연결해준다.
  ㄴ shmflg: 권한. 0이면 읽기/쓰기 가능, SHM_RDONLY면 읽기만 가능.
- shmdt(addr): 공유 메모리를 해제한다.

```
// sys/types, sys/ipc, sys/shm, string, unistd, stdlib, stdio 헤더 필요
int shmid = shmget((key_t)1234, 10, IPC_CREAT|0644);
shmaddr = (char*)shmat(shmid, (char*)NULL, 0);
strcpy((char*)shmaddr, "Hello, world!"); // 쓰기
printf("%s", shmaddr);                   // 읽기
shmdt((char*)shmaddr);                   // 메모리 해제
// 위 동작들은 어느 프로세스에서든 실행 가능.
```

- 참고: shmctl(id, 명령, 옵션) 으로 공유 메모리 컨트롤 가능

5) 시그널

- 커널 또는 프로세스에서 다른 프로세스에 이벤트 발생을 알려주는 기법
  ㄴ 유닉스에서 30년 이상 사용된 전통적인 기법이다.
  ㄴ ex) Ctrl + C로 프로세스 종료시키기
- 커널 모드에서 사용자 모드 전환 시, 시그널 정보를 확인하여 처리한다.
  ㄴ 프로세스는 타이머 인터럽트 등에 의해 수시로 커널 모드로 전환된다.
- 주요 시그널
  ㄴ SIGKILL: 프로세스를 어떤 경우에든 종료시키는 강력한 시그널. 슈퍼관리자가 사용한다.
  ㄴ SIGALARM: 알람을 발생시킨다.
  ㄴ SIGSTP: 프로세스 중단 (Ctrl + Z)
  ㄴ SIGCONT: 중단된 프로세스 재실행
  ㄴ SIGINT: 인터럽트를 보내서 프로세스 종료 (Ctrl + C)
  ㄴ SIGSEGV: 프로세스가 다른 메모리영역 침범 시
- 특정 시그널 무시, 블록(블록 풀릴 때까지 시그널 처리 X), 재정의 등이 가능하다.
- kill(pid, 시그널 번호): pid에 해당하는 프로세스에 입력한 시그널을 보낸다.
  ㄴ 쉘에 kill -l을 입력하여 시그널 종류와 번호를 확인할 수 있다.
- signal(시그널 이름, 동작): 해당 시그널을 입력한 동작으로 재정의한다.
  ㄴ 동작에는 SIG_IGN(시그널 무시), SIG_DFL(디폴트 동작) 등이 있다.
  ㄴ signal(SIGINT, SIG_IGN): SIGINT를 무시한다.
  ㄴ signal(SIGINT, (void*)func): SIGINT 수신 시 func 함수를 호출한다.

9. 쉘 스크립트

1) 개념
- 쉘을 사용하여 프로그래밍이 가능하다. 쉘 명령어 기반 + 몇 가지 문법이 추가된 형태.
 ㄴ 쉘마다 문법이 조금씩 다르다. 여기서는 bash를 기준으로 한다.
- 서버 운영(DevOps) 시 자동화 작업에 쉘스크립트가 많이 사용된다.
 ㄴ ex) 오래된 로그 삭제
- 파일로 작성되며, 일반적으로 'file.sh' 형태의 sh 확장자를 가진다.
- ‘echo 출력할 내용’ 으로 내용을 출력할 수 있다.
 ㄴ 출력할 내용은 여러 개를 한 줄에 작성할 수 있다. 각 내용은 띄어쓰기로 구분한다.
 ㄴ ex) echo value1 value2 value3
- 쉘 스크립트 파일 첫 줄은 ‘#!/bin/bash’로 시작하며, 실행 권한을 가지고 있어야 한다.
```
ex) “Hello, world!”를 출력하는 명령어 작성
#!/bin/bash
echo "Hello, world!"
```
- #으로 시작하는 줄은 주석 처리된다.
- exit로 스크립트를 종료할 수 있다.
 ㄴ ‘exit 숫자’ 형태로 부모 프로세스에게 종료 코드를 전달할 수 있다.

2) 변수
- 변수: 선언은 ‘변수명=값’ 으로 선언하며, ‘변수명 = 값’같은 띄어쓰기는 허용되지 않는다.
 ㄴ 변수 사용은 ‘$변수명’ 으로 사용한다.
- 리스트 변수: ‘변수명=(값1 값2 값3)’ 형태로 선언한다. 각 값은 띄어쓰기로 구분한다.
 ㄴ 변수 사용은 ‘${변수명[인덱스]}’ 로 사용한다.
 ㄴ ‘${변수명[@]}’ 또는 ‘${변수명[*]}’로 모든 데이터를 출력할 수 있다.
 ㄴ ‘${#변수명[@]}’ 로 리스트에 몇 개의 값이 있는지 출력할 수 있다.
 ㄴ ex) 이름, 나이를 변수로 만들고 각각 출력하기
```
#!/bin/bash
info=("jiho" 27)
echo ${info[0]}
echo ${info[1]}
```
- 예약어: 사전에 정의된 지역 변수들이 존재한다.
 ㄴ $$: 현재 실행중인 쉘 스크립트 프로세스의 pid
 ㄴ $0: 쉘 스크립트 파일 이름
 ㄴ $1 ~ $9: 실행중인 쉘 스크립트 프로세스에 전달된 n번째 인수(=인자)
  ㄴ ls -al -z 로 ls.sh 스크립트 실행 시 $1은 ‘-al’이 된다.
 ㄴ $*: 모든 인수 리스트
  ㄴ 위의 예시에서 $*는 ‘-al -z’가 된다.
 ㄴ $#: 인수의 개수
 ㄴ $?: 최근 실행한 명령어의 종료 값. 정상 실행 및 종료되었다면 0이다.

ㄴ 1~125는 에러, 126은 실행되지 않음, 128~255는 시그널 발생을 의미한다.

* 다른 쉘 명령어의 반환값은 '$(명령어)' 형태로 쓸 수 있다.
 - 예를 들어, $(ls) 는 해당 스크립트가 위치한 디렉토리 내 파일을 배열로 반환한다.
  ㄴ ex) 디렉토리의 파일 리스트 출력하기

```
#!/bin/bash
filelist=( $(ls) )            # 괄호 빼고 filelist=$(ls) 도 가능
echo ${filelist[*]}
```

3) 연산자
- `expr 계산식` 으로 숫자 계산이 가능하다.
 ㄴ 앞뒤에는 ` 기호를 붙여야 한다. (작은따옴표가 아니라 키보드 좌측 상단의 ` 기호)
 ㄴ 계산식의 각 숫자, 변수, 기호 사이에는 띄어쓰기가 있어야 한다.
 ㄴ 연산자 * 와 괄호 () 앞에는 역슬래시를 넣어야 한다.
 ㄴ ex) num=`expr \( 3 \* 5 \) / 4 + 7`
  C 계열 언어에서 int num = (3*5)/4+7; 과 동일한 내용.
- 논리 연산자는 &&, ||, ! 가 사용되며 기능은 다른 언어와 같다.
 ㄴ &&, || 연산자는 각각 and와 or를 상징하는 -a, -o 로 대체할 수 있다.
 ㄴ 리터럴로 true와 false의 사용도 가능하다.

4) 조건문
- if [조건] then 명령문 fi 형태로 조건문을 사용한다. 조건 앞뒤에 공백은 필수.
 ㄴ C 계열 언어에서 if(조건) { 명령문 } 과 동일하다.
 ㄴ then과 fi 사이에만 있으면 들여쓰기와 무관하지만, 통상적으로 들여쓰기를 한다.
 ㄴ then ~ fi 사이에 조건을 추가할 경우 elif 또는 else의 사용도 가능하다.
 ㄴ 줄바꿈 생략 시 세미콜론을 추가한다. 'if [조건]; then 명령문; fi' 형태가 된다.
- 여러 조건이 있을 경우 [조건1] && [조건2] 형태로 연산 가능하다.
- 수치 비교는 ==, != 는 정상 작동하고, <, >는 [[조건]] 형태로 사용 시 동작한다.(<=, >= 는 동작하지 않음) 다만 아래와 같은 형태가 권장된다.
 ㄴ A -eq B: A == B (equal)
 ㄴ A -ne B: A != B (not equal)
 ㄴ A -lt B: A < B (less than)
 ㄴ A -le B: A <= B (less or equal)
 ㄴ A -gt B: A > B (greater than)
 ㄴ A -ge B: A >= B (greater or equal)
 ㄴ ex) 두 인자를 받아서, 첫 번째 인자값이 두 번째 인자보다 크면 출력하는 코드

```
#!/bin/bash
if [ $1 -gt $2 ]
then
        echo $2
fi
```

- 파일 상태 기반 조건: 입력한 주소의 파일(또는 폴더) 상태에 따라 참/거짓을 반환한다.
 ㄴ -e 파일명: 입력한 경로의 파일이 존재하면 참
 ㄴ -d 파일명: 파일이 디렉토리면 참
 ㄴ -h 파일명: 파일이 심볼릭 링크(=소프트 링크) 파일이면 참
 ㄴ -f 파일명: 일반 파일이면 참
 ㄴ -r 파일명: 읽기 가능하면 참
 ㄴ -w 파일명: 쓰기 가능하면 참
 ㄴ -x 파일명: 실행 가능하면 참
 ㄴ -s 파일명: 파일 크기가 0보다 크면 참
 ㄴ -u 파일명: 파일이 set-user-id가 설정되면 참
 ㄴ ex) 입력받은 파일이 있는지 없는지 검사하는 쉘 스크립트

```
#!/bin/bash
if [ -e $1 ]
then
        echo “파일이 존재함”
else
        echo “파일이 존재하지 않음”
fi
```

- 문자열 조건: 입력받은 내용이 있는지 없는지 검사할 때 주로 사용된다.
 ㄴ -z 문자열: 문자열 길이가 0이면 참. C#의 string.IsNullOrEmpty와 유사함.
 ㄴ -n 문자열: 문자열 길이가 0이 아니면 참. 위의 반대.

* 논리 연산 &&, ||, <, > 에서 에러가 나는 경우, 조건문에 [,] 대신 [[,]]를 사용하면 정상 작동하는 경우가 있다.

4-1) 조건문 활용
- 쉘 스크립트 해석하기: ‘ping -c 1 192.168.0.1 1> /dev/null’
 ㄴ ping: 서버가 접속해 있는 클라이언트 ip 주소에 응답 요청을 보내는 것. 일정 시간 응답하지 않으면 해당 클라이언트는 정상적으로 작동하지 않는 것으로 간주한다.
 ㄴ -c 1: 확인 요청을 1번만 하라는 옵션
 ㄴ 192.168.0.1: 요청을 보낼 ip주소
 ㄴ 1: 표준 출력. 0은 표준 입력, 1은 표준 출력, 2는 표준 에러를 의미한다.
 ㄴ >: 리다이렉션. 표준 출력 내용을 출력하는 대신 지정한 주소로 보낸다. 5-2) 참고.
 ㄴ /dev/null: null 디바이스의 경로. 표준 출력 내용을 버리라는 의미.
- 명령 실행 후 결과에 따른 처리를 하도록 스크립트를 작성하면 아래와 같다.

```
#!/bin/bash
ping -c 1 192.168.0.1 1> /dev/null
if [ $? == 0 ]   # $?는 최근 명령(ping)의 실행 결과. 0은 정상 종료(응답 받음)
then
        echo “핑 성공!”
else
```

```
        echo “핑 실패!”
    fi
```

5) 반복문

- 기본적으로 반복자를 지원하는 변수가 있을 때, 아래와 같이 사용한다.

```
    for 임시변수 in 변수
    do
        명령문
    done
```

ㄴ 임시변수 선언 시에는 변수명을 쓰면 되지만, 명령문에서 사용할 땐 $를 붙여야 한다.

ㄴ ex) 현재 디렉토리의 파일(및 폴더) 모두 출력하기

```
    #!/bin/bash
    for file in $(ls)
    do
        echo $file
    done
```

ㄴ 반복문도 한 줄에 쓸 수 있다. 위 예시를 한 줄로 표현하면 다음과 같다.

```
    #!/bin/bash
    for file in $(ls); do echo $file; done
```

- while문은 아래와 같이 사용한다.

```
    while [ 조건 ]
    do
        명령문
    done
```

ㄴ 위 for문의 예시를 while로 바꾸면 아래와 같다.

```
    #!/bin/bash
    filelist=()                 # $(ls)을 대입하면 전체를 하나의 원소로 받아온다.
                                # 따라서 $(ls)의 원소를 하나하나 넣어준다.
    for f in `ls`; do filelist+=($f); done
    count=${#filelist[@]}       # 반복 회수로 filelist의 원소 개수를 지정
    index=0
    while [ $index -lt $count ]     # index가 count보다 작으면 반복
    do
        echo ${filelist[$index]}
        index=`expr $index + 1`
    done
```

6-1) 스크립트 분석: 백업 스크립트

- 폴더 경로와 백업할 폴더 경로를 입력받아 파일을 백업하는 스크립트

```
    #!/bin/bash
    if [ -z $1 ] || [ -z $2 ]; then    # 매개변수가 빠졌다면 사용법을 출력
```

```
        echo usage: $0 sourcedir targetdir
    else
        from=$1
        to=$2
    #파일명은 날짜 출력 명령어 date를 사용해서 매번 다르게 해 준다.
        filename=backup.$(date +%y%m%d%H%M%S).tar.gz
        if ![ -d $to ]; then #이동할 위치에 디렉토리가 있으면 바로 복사
            tar -cvzf $to/$filename $from
        else            #없으면 디렉토리 생성 후 복사
            mkdir $to
            tar -cvzf $to/$filename $from
        fi
    fi
```

- tar: 압축 명령. 대상 파일을 압축 또는 해제하여 저장한다.
 ㄴ c 옵션: 파일을 묶음
 ㄴ x 옵션: 묶음을 해제
 ㄴ v 옵션: 묶음/해제 과정을 화면에 표시
 ㄴ z 옵션: gunzip을 사용
 ㄴ f 옵션: 파일 이름을 지정
 ㄴ 압축 시: tar -cvzf [압축된 파일 경로] [압축할 파일(폴더) 경로]
 ㄴ 해제 시: tar -xvzf [압축된 파일 경로]

6-2) 스크립트 분석: 로그 파일 정리 스크립트
- 2일 이상 지난 로그파일은 압축하고, 압축한 지 3일 지난 파일은 삭제하는 스크립트

```
    #!/bin/bash
    path=/var/log #이 경로엔 슈퍼관리자가 쓰는 파일이 많으니 명령은 sudo로 실행
    zipDay=1      #하루 지난 파일까지는 놔두고, 2
    deleteDay=2

    cd $path
    sudo find . -type f -name '*log.?' -mtime +$zipDay -exec bash -c "gzip {}" \;
    2>  # 'syslog.1' 과 같은 파일을 찾는다.
    sudo find . -type f -name '*.gz' -mtime +$deleteDay -exec bash -c "rm -f {}"
    \; 2>       # 확장자가 .gz인 파일을 찾는다.
```

- find: 파일 찾기 명령. 찾은 파일에 대해 다른 명령어를 수행하게 할 수도 있다.
 ㄴ ex) find 탐색대상 -type 타입 -name ‘검색어’ -exec bash -c "명령어1; 명령어2;“ \;
 ㄴ find . : 현재 디렉토리 탐색
 ㄴ -type f: 일반 파일
 ㄴ -name ‘검색어’: 검색어를 포함한 파일을 찾는다. 검색어는 정규표현식으로 입력한다.
 ㄴ -mtime +숫자: 생성 시각 기준 입력한 숫자를 초과한 파일을 탐색한다. 1 입력 시 2일 이상 경과한 파일만을 찾는다.

ㄴ -exec bash -c: bash를 통해 쉘 명령어를 실행한다.
ㄴ “gzip {}": 중괄호 안에 찾은 파일 경로를 입력한다. gzip은 압축만 하라는 명령어.

10. 스레드 제어

1) Pthread
- 리눅스의 스레드 표준 API. POSIX Thread를 뜻하며 피-스레드 라고 부른다.
- Pthread API: 유닉스 시스템 핵심 스레딩 라이브러리.
ㄴ 기능은 크게 스레드 관리(생성, 종료, join, detach 등)와 동기화(mutex 등)로 구분된다.
ㄴ 다른 스레딩 솔루션도 Pthread 기반으로 구현되어 있다.
ㄴ 저수준 API로 100여 개의 함수를 제공한다. 모든 함수는 pthread_ 로 시작한다.
ㄴ C 언어에서 <pthread.h> 헤더 파일에 정의되어 있다.
- Pthread 라이브러리: pthread는 기본 라이브러리인 glibc가 아니라, libpthread라는 별도의 라이브러리에 구현되어 있다. 따라서 컴파일 시 명시적으로 -pthread 옵션이 필요하다.
ㄴ ex) gcc -pthread test.c -o -test
ㄴ 지립씨(glibc)에 대한 설명은 6-1) 참고

2) 스레드 관리 명령어
- 스레드 생성: pthread_create(pthread_t, attr, func, arg)
ㄴ pthread_t: 생성된 스레드 식별자가 들어갈 자리. 참조형으로 넘긴다.
ㄴ attr: 스레드 설정 옵션이며 일반적으로 NULL로 둔다.
ㄴ func: 스레드에서 실행할 함수가 들어간다. (void* 타입)
ㄴ arg: func 함수의 인자가 들어간다. (void* 타입)
ㄴ 사용 예시)

```
pthread_t thread1;
void* func(void* ptr);
ret = pthread_create(&thread1, NULL, func, (void*)myValue1);
```

- 스레드 종료: pthread_exit(void* returnValue)
ㄴ 기본적으로 스레드는 함수 동작이 끝나면 종료되지만, 명시적 종료 함수가 있다.
ㄴ 매개변수는 리턴값을 넣으며, 정상 종료라면 보통 NULL을 넣는다.
- 스레드 조인: pthread_join(pthread_t, status)
ㄴ p_thread 식별자를 가진 스레드 종료까지 대기한다. status에 종료값이 담긴다.
ㄴ 해당 스레드는 메인 스레드에서 종료한다.
ㄴ 해당 스레드에서의 작업 결과를 기다려야 하거나, 종료값을 통해 메인 스레드에서 추가적인 동작을 진행할 때 사용한다.
- 스레드 디태치: pthread_detach(pthread_t)
ㄴ 스레드 종료 시 즉시 관련 리소스를 해제(free)한다.
ㄴ 메인 스레드는 해당 스레드 종료까지 기다리지 않고 다음 동작을 바로 실행한다.
- 스레드 생성 시 용도에 따라 join 또는 detach 중 하나는 명시적으로 해 주는 게 좋다.

3) 스레드 동기화 명령어
- 스레드가 임계 구역에 진입할 때는 mutex를 선언하여 lock/unlock 처리한다.
- 뮤텍스 선언: ‘pthread_mutex_t 변수명 = PTHREAD_MUTEX_INITIALIZER’
 ㄴ 해당 뮤텍스를 lock/unlock 함수에 인자로 넣어 임계 구역을 보호한다.
- 락 걸기: pthread_mutex_lock(뮤텍스) / pthread_mutex_unlock(뮤텍스)
 ㄴ ex) pthread_mutex_lock(&mutex_key);

11. 메모리와 파일 제어

1) mmap
- 메모리에 파일을 매핑하는 함수.
- 프로세스에서 파일에 접근하는 비용이 크기 때문에, 메모리에 매핑해두면 파일 처리를 빠르게 할 수 있다.
- 형태: void* mmap(void* start, size_t length, int prot, int flags, int fd, off_t offset)
 ㄴ 'start+offset' 부터 length 만큼의 물리 메모리 공간을 mapping할 것을 요청한다.
 ㄴ start: 보통 NULL(또는 0)을 사용. 매핑 시작 지점은 offset에서 입력한다.
 ㄴ length: 매핑할 메모리 공간의 크기
 ㄴ offset: 매핑을 원하는 물리 메모리 주소 지정.
 ㄴ prot: 보호 모드 설정 옵션. ‘PROT_READ | PROT_WRITE’가 많이 사용된다.
  ㄴ PROT_READ : 읽기 가능
  ㄴ PROT_WRITE : 쓰기 가능
  ㄴ PROT_EXEC : 실행 가능
  ㄴ PROT_NONE : 접근 불가
 ㄴ flags: 메모리 주소 공간 설정 옵션.
  ㄴ MAS_SHARED: 다른 프로세스와 공유 가능 (프로세스 밖에 공간 할당)
  ㄴ MAS_PRIVATE: 프로세스 내에서만 사용 가능 (프로세스 안에 공간 할당)
  ㄴ MAS_FIXED: 지정된 주소로 공간 지정
 ㄴ fd: 매핑할 file description 입력. open() 함수로 파일을 열어 반환받은 값을 입력한다.
- 동작 방식
 (1) mmap 실행 시, 가상 메모리 주소에 파일 주소를 매핑한다.
  ㄴ 해당 가상 메모리에 접근하지 전까지는 물리 메모리에 적재되지 않는다.
  ㄴ ‘운영체제’ 파일 ‘7-2-2) 요구 페이징’ 참고
 (2) 매핑한 가상 메모리에 처음 접근 시 페이지 폴트 인터럽트가 발생한다.
 (3) OS에서 파일을 복사하여 물리 메모리에 올린다.
 (4-1) read: 물리 메모리에 적재된 페이지를 읽는다.
 (4-2) write: 물리 메모리에서 데이터 수정 후, 페이지 상태 flag에서 dirty bit를 1로 수정
 (5) 파일을 닫을 때 물리 메모리 데이터가 파일에 업데이트된다.
  ㄴ 파일 닫기, 명시적 저장, 프로세스 종료 등 저장 시점은 여러 가지가 있다.
  ㄴ 수정할 때마다 파일을 저장하는 것에 비해 저장 횟수가 매우 적어진다.
- 장점: 읽기/쓰기 시 파일 접근을 메모리 포인터 조작으로 대체하여 성능을 개선한다.

- 단점: 매핑은 페이지 단위로 이루어지므로, 페이지 공간과 맞아떨어지지 않는 크기의 파일이라면 남는 공간은 0으로 채워서 할당한다. 한 페이지 정도의 낭비되는 공간이 생긴다.

2) mmap 활용
- msync(start, length, flags): 매핑한 메모리 데이터와 파일을 동기화한다.
 ㄴ 일반적으로 메모리의 내용을 파일에 명시적으로 저장한다.
 ㄴ start에는 mmap()에서 리턴받은 메모리 주소를 입력한다.
 ㄴ flags에는 옵션이 입력된다. 일반적으로 동기 또는 비동기 중 하나를 사용한다.
        ㄴ MS_ASYNC : 비동기 방식 (동기화 명령만 내리고 다음 코드 실행)
        ㄴ MS_SYNC : 동기 방식 (동기화 완료까지 대기)
        ㄴ MS_INVALIDATE : 현재 메모리 맵을 무효화하고 파일의 데이터로 갱신
- munmap(addr, length): 매핑한 메모리를 해제한다.
 ㄴ addr에는 mmap()에서 리턴받은 메모리 주소를 입력한다.
- 예제 코드

```
char* path = "UserStatus.txt"
struct stat info;
int fd = open(filepath, O_RDONLY, (mode_t)0600);
// fstat: 파일 정보를 구조체에 넣어주는 함수
fstat(fd, &info);
// st_size: 구조체의 크기를 반환한다.
// 쓰기로 할 경우 PROT_READ | PROT_WRITE 로 사용할 것
char* map = mmap(0, info.st_size, PROT_READ, MAP_SHARED, fd, 0);
// 매핑된 파일 사용
for (off_t i = 0; i < info.st_size; ++i) printf("User %c", map[i]);
// 매핑 해제
munmap(map, info.st_size);
// 파일 닫기
close(fd);
```

* stat 구조체는 파일의 inode 데이터를 담을 수 있도록 미리 정의된 구조체로, 가상 파일 시스템에서의 디바이스 정보, inode 번호, 메타데이터들이 들어갈 변수가 선언되어 있다.

3) inode 메타데이터 관리
- inode 메타데이터는 파일종류(권한), 소유자(그룹), 크기, 생성 및 수정 날짜, Direct blocks, Indirects 등으로 구성된다.
 ㄴ direct blocks(12개)는 데이터를 직접적으로 가리키며, Indirects는 다른 indirect 또는 direct block 주소를 가리킨다. 가변적인 데이터 크기에 대응할 수 있다.
- stat(path, buffer): path 경로의 파일을 buffer 구조체에 넣는다.
- fstat(file, buffer): stat와 같으나 경로 대신 file decrtiption을 넣는다.

4) 표준 입출력

- command로 실행되는 프로세스는 3가지 스트림을 가지고 있다.
  ㄴ 표준 입력, 표준 출력, 오류 출력 스트림 (stdin, stdout, stderr)
  ㄴ 모든 스트림은 일반적인 plain text로 콘솔에 출력된다.
- 프로세스에서 입력을 받거나 무언가를 출력하는 것은 가상 파일 시스템에서 stdin, stdout 등의 파일을 읽고 쓰는 것과 같다.
  ㄴ 입력 시 stdin에 입력한 내용이 작성되는 것