

모던 리눅스 교과서 1주차

최우성

차 례

차 례	2
제 1 장 리눅스 소개	3
1.1 모던 환경이란 무엇인가?	3
1.2 리눅스 역사	4
1.3 리눅스 배포판	4
1.4 리소스 가시성	5
제 2 장 리눅스 커널	6
2.1 리눅스 아키텍처	6
2.2 CPU 아키텍처	7
2.2.1 x86-64 아키텍처	7
2.2.2 ARM 아키텍처	7
2.2.3 RISC-V 아키텍처	8
2.3 커널 구성요소	8
2.3.1 프로세스 관리	8
2.3.2 메모리 관리	10
2.3.3 네트워킹	12
2.3.4 파일시스템	12
2.3.5 디바이스 드라이버	12
2.3.6 시스템 콜	13
2.4 커널 확장	16
제 3 장 셸과 스크립팅	19
3.1 기본 개요	19

리눅스 소개

1.1 모던 환경이란 무엇인가?

작성 시점인 2024년 기준 리눅스는 다양한 환경에서 사용된다.

- 모바일 디바이스
안드로이드 OS는 리눅스의 변형이다.
- 클라우드 컴퓨팅
리눅스는 CSP에서 제공하는 서버의 OS로 많이 사용된다.
- IoT
리눅스는 사물 인터넷 OS로도 사용된다.
- 프로세스 아키텍처의 다양성
전통의 x86-64 이외에도 ARM, RSIC-V 같은 ISA도 지원된다.
- 컨테이너 인프라
Kubernetes, Docker로 대표되는 컨테이너 인프라에서 기본적으로 사용된다.

1.2 리눅스 역사

90년대

1991년 리누스 토르발스에 의해 시작되었으며 수많은 기여자의 도움으로 리눅스 1.0.0은 3년이 되기도 전에 릴리즈되었다.

이 시점에서 Unix/GUN 소프트웨어를 실행할 수 있는 운영체제라는 최초 목표는 달성되었다. 또한, 이 기간에 최초의 상용 리눅스인 Red Hat 리눅스가 등장했다.

2000 - 2010

여러 기업에서 리눅스를 본격적으로 도입하며 폭넓적으로 성장하는 시기이다.

이 시기부터 Linux는 Unix, Windows Server 같은 경쟁 서버보다 점유율 우위를 점하기 시작한다. 또한, 여러 배포판(distro)가 출시되며 경쟁했다.

2010 이후

단순 서버용 이상으로 확장되어 IoT, 모바일 디바이스에서도 사용한다.

또한, 컨테이너 인프라가 본격적으로 사용되기 시작했다.

그리고, 상용 서버 배포판은 사실상 Red Hat, Debian 기반으로 수렴했다.

보다 상세한 한글 자료는 [리눅스 29돌 사건으로 보는 그 역사](#)를 참고하라.

1.3 리눅스 배포판

리눅스 커널은 단순 시스템 콜과 디바이스 드라이버의 집합을 의미한다.

리눅스 배포판은 커널과 관련 구성요소의 전체 번들을 의미한다. 관련 구성요소에는 패키지 관리자, 파일시스템 레이아웃, init system, shell이 포함된다.

1.4 리소스 가시성

리눅스는 리소스의 전역 보기(global view)를 지원한다.

여기서 전역보기는 무엇이고 전역보기의 반대는 무엇이며 리소스는 무엇인가?

리소스란?

리소스는 소프트웨어 실행을 지원하는 데 사용할 수 있는 모든 것으로 간주될 수 있다.
예를 들면 다음과 같은 것이 리소스로 간주될 수 있다.

- 하드웨어
- 파일시스템
- HDD/SSD
- process
- device
- routing table(Network)
- 자격증명(credential)

전역 리소스

예를 들어 HDD 용량 확인, 특정 파일 읽기, ps -ef로 pid 확인하기 같은 것은 여러 계정에서 동일하게 볼 수 있다.

리눅스에서 할 수 있는 대부분의 것이 전역에서 되므로 리눅스의 모든 것은 전역일 것이라 기대할 수 있지만 실제로 그렇지 않다.

로컬 리소스

모든 리소스가 전역인 것은 아니다. 예를 들면 namespace를 통해 리소스의 로컬보기를 지원 할 수 있다. 또한, 특정 프로세스가 메모리 리소스를 고갈시키는 것을 방지하기 위해 메모리 소비를 제한하는 것도 메모리가 전역 리소스로 사용하지 않을 수 있다든 것을 보여준다. 이처럼 리소스의 사용을 제한하는 것은 프로세스의 격리의 일종이며 리눅스에서는 **cgroup**이라는 커널 기능을 사용해 이러한 종류의 격리를 제공한다.

리눅스 커널

2.1 리눅스 아키텍처

- **하드웨어 계층**

CPU와 메인 메모리, 디스크, NIC, I/O 디바이스 모두를 총칭한다.

- **커널 계층**

하드웨어 계층과 사용자 영역 계층의 사이에 위치한다.

이번 chapter에서 다루는 계층이다.

- **사용자 영역 계층**

shell 및 ps, ssh 같은 유ти리티, GUI를 비롯해 대부분의 앱이 실행되는 계층이다.

다른 계층간 인터페이스는 리눅스 운영체제 패키지의 일부이다. 그 중 커널과 사용자 영역 계층 인터페이스를 **시스템 콜(system call)**이라 부른다.

하드웨어와 커널 사이의 인터페이스는 시스템 콜과 달리 단일 인터페이스가 아니라 일반적으로 하드웨어별 그룹화된 개별 인터페이스 모음으로 구성된다.

- CPU 인터페이스
- main memory 인터페이스
- 네트워크 인터페이스와 드라이버
- 파일시스템과 블록 디바이스 드라이버 인터페이스
- 캐릭터 디바이스, 하드웨어 인터럽트, 키보드, 터미널, 기타 I/O 등의 입력 디바이스를 위한 디바이스 드라이버

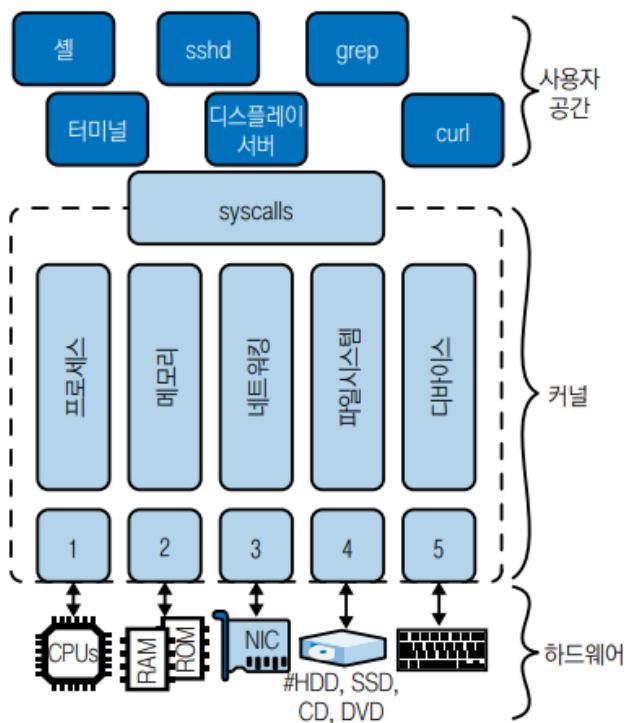


그림 2.1: 리눅스 아키텍처 개요

일반적으로 **커널 모드**는 추상화를 제한함으로써 빠르게 실행함을 의미하는 반면, **사용자 모드**는 상대적으로 느리지만 더 안전하고 편리한 추상화를 의미한다. 대부분의 경우 커널 모드를 신경쓰지 않아도 사용에 지장은 없지만, 커널과 상호 작용하는 방법(시스템 콜)을 아는 것은 중요하다.

2.2 CPU 아키텍처

2.2.1 x86-64 아키텍처

x86과 amd64를 합쳐서 x86-64라고 부른다. x86은 인텔 32-bit ISA이며, amd64는 64-bit ISA이다. 대부분의 경우 사용되는 CPU이며 오래전부터 사용되어왔고, CISC(Complex Instruction Set Computer) 아키텍처 에너지 효율은 높지 않다.

2.2.2 ARM 아키텍처

RISC(Reduced Instruction Set Computer) 아키텍처이며 적은 명령어 집합을 사용한다. RISC는 CISC에 비해 전력 소모가 적다는 장점을 가지고 있으며, 저전력 환경(임베디드, 휴대용 디바이스)에서 널리 사용된다.

2.2.3 RISC-V 아키텍처

ARM과 달리 개방형 RISC 표준으로 아직 널리 사용되지는 않는다. 다만 ARM과 달리 라이센스 비용이 없어 주목받고 있다.

2.3 커널 구성요소

커널 코드에서 제공하는 주요 기능은 다음과 같다.

- **프로세스 관리:** 실행 파일을 기반으로 프로세스 시작
- **메모리 관리:** 프로세스에 메모리 할당 및 파일을 메모리에 매핑
- **네트워킹:** 네트워크 인터페이스 관리 및 네트워크 스택 제공
- **파일시스템:** 파일 관리를 제공하고, 파일 생성과 삭제 지원
- **캐릭터 디바이스와 디바이스 드라이버 관리**

2.3.1 프로세스 관리

커널에는 프로세스 관리와 관련된 부분이 여러개 있다. 그중 일부는 인터럽트 같은 CPU 아키텍처 관련 사항을 처리하고, 다른 부분은 프로그램 실행과 스케줄링에 중점을 둔다.

일반적으로 프로세스는 실행 가능한 프로그램(또는 바이너리)을 기반으로 하며, 사용자가 대면하는 유닛(unit)이다. 반면에 스레드는 프로세스 컨텍스트상 실행 유닛을 말한다. 멀티스레드라는 용어가 존재하는 것처럼 프로세스에는 여러 실행 유닛이 병렬로 실행되며 이는 잠재적으로 다른 CPU에서 실행될 수 있다.

아래는 실제 리눅스에서 프로세스를 관리하는 단위이다.

- **세션**

하나 이상의 프로세스 그룹을 포함하고 선택적으로 tty가 연결된 상위 수준의 사용자 대면 유닛. 커널은 textbf 세션 ID(SID)라는 번호를 통해 세션을 식별한다.

- **프로세스 그룹**

하나 이상의 프로세스가 포함되어 있으며, 한 세션에는 foreground 프로세스 그룹이 둘 이상일 수 없다. 커널은 **프로세스 그룹 ID(PGID)**라는 숫자를 통해 프로세스 그룹을 식별한다.

- **프로세스**

여러 리소스(주소 공간, 하나 이상의 스레드, 소켓 등)를 그룹으로 추상화한 것이며, 커널은 /proc/self를 통해 현재 프로세스를 사용자에게 노출한다. 커널은 **프로세스 ID(PID)**라는 숫자를 통해 프로세스를 식별한다.

- **스레드**

커널에 의해 프로세스로 구현된 유닛을 말한다. 즉 스레드를 나타내는 전용 데이터 구조는 없다. 오히려 스레드는 특정 리소스(ex - 메모리, signal handler)를 다른 프로세스와 공유하는 프로세스다. 커널은 스레드 ID(TID)와 스레드 그룹(TGIO)를 통해 스레드를 식별하며, 공유된 TGID 같은 멀티스레드 프로세스를 의미한다.

- **태스크**

커널에는 sched.h에 정의된 **task_struct**라는 데이터 구조가 있으며, 이는 프로세스와 스레드 구현의 기반을 형성한다. 이 데이터 구조는 스케줄링 관련 정보, 식별자(ex - PID, TGID), signal handler, 성능이나 보안과 관련된 기타 정보를 수집한다. 즉, 앞서 언급한 모든 유닛은 태스크에서 파생되거나 고정(anchor)된다. 하지만 캐스크는 커널 외부에 그대로 노출되는 일이 없다.

실제로 그러한지 실습해보자.

아래는 ps -j로 확인할 수 있는 정보이다.

02:53:57			~ → ps -j		
PID	PGID	SID	TTY	TIME	CMD
493	493	493	pts/0	00:00:00	bash
878	878	493	pts/0	00:00:00	ps

그림 2.2: 실습

1. bash 셸 프로세스의 PID, PGID, SID는 모두 -이다.
ls -la /proc/493/task/493/로 태스트 수준의 정보를 수집할 수 있다.
2. ps 프로세스의 PID/PGID는 878이고 SID는 셸과 동일하다.

리눅스의 작업 데이터 구조가 스케줄링과 관련된 정보를 준비된 상태로 가지고 있음을 앞에서 언급했던 적이 있다. 이는 항상 프로세스는 아래 그림처럼 특정한 상태(state)에 있음을 의미한다.

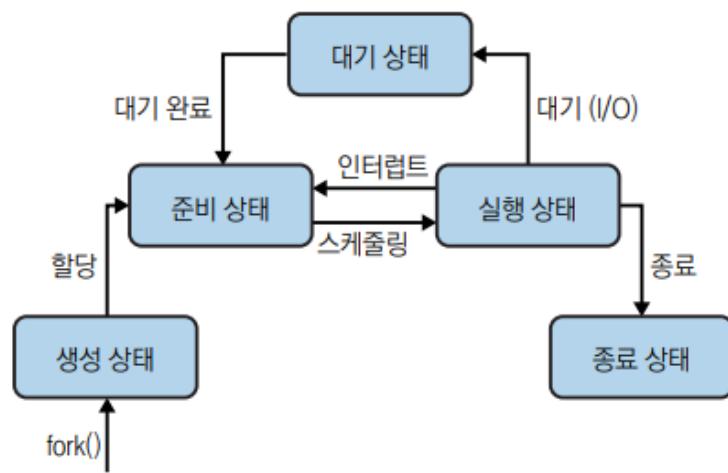


그림 2.3: 리눅스 프로세스 상태

이벤트가 일어날 때마다 상태 전환이 일어난다. 예를 들어 실행 중인 프로세스가 일부 I/O 작업을 진행했으나 이를 실행할 수 없을 때 대기 상태로 전환될 수 있다.

보다 상세한 리눅스 프로세스의 상태는 다음 글 [리눅스 프로세스 상태](#)를 참고하라

2.3.2 메모리 관리

가상 메모리는 시스템이 물리적으로 가지고 있는 것 보다 더 많은 메모리를 갖고 있는 것처럼 보이게 한다. 사실 모든 프로세스는 가상 메모리를 얻는다. 물리 메모리와 가상 메모리는 모두 **페이지**(page)라고 부르는 고정길이 묶음으로 나뉜다.

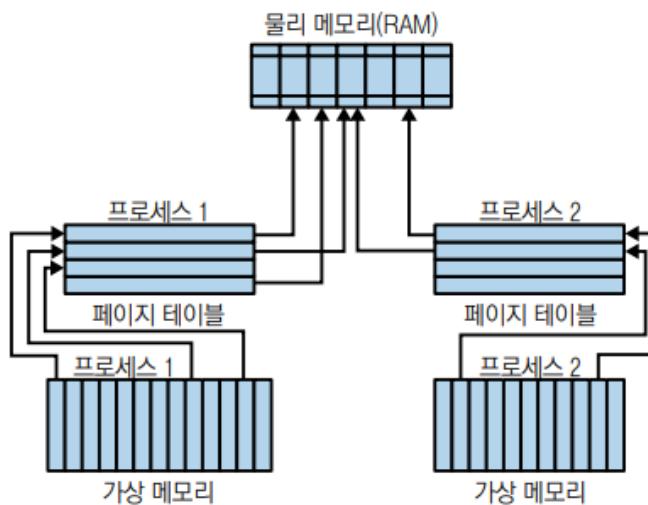


그림 2.4: 가상 메모리 관리 개요

이전 페이지의 그림은 고유한 페이지 테이블이 있는 두 프로세스의 가상 주소 공간을 보여준다. 이 페이지 테이블은 프로세스의 가상 페이지를 주 메모리의 물리적 페이지에 맵핑(mapping)한다.

각 프로세스 수중의 페이지 테이블을 통해 여러 가상 페이지가 동일한 물리적 페이지를 가리킬 수 있다. 즉, 기존 공간을 최적으로 사용하면서 각 프로세스에 그들의 페이지가 실제로 RAM에 존재한다는 환상을 효과적으로 일으키는 방법으로 이는 실제 가지고 있는 물리 메모리 용량 이상으로 메모리를 사용할 수 있게 하여 어떤 의미로는 메모리 관리의 핵심이라 할 수 있다.

CPU가 프로세스의 가상 페이지에 접근할 때마다 원칙적으로 CPU는 프로세스가 사용하는 가상 주소를 이에 해당하는 실제 주소로 변환해야 한다. 이 절차의 속도를 높히기 위해 요즘 CPU 아키텍처는 TLB(translation lookaside buffer)라는 조회용 on-chip 을 지원한다. TLB는 작은 cache로 mapping된 주소가 누락된 경우 CPU가 프로세스 페이지 테이블을 통해 페이지의 실제 주소를 계산하고 TLB를 업데이트한다.

전통적으로 리눅스의 기본 페이지 크기는 4KB였으나 커널 버전 2.6.3 부터는 그 이상의 대형 페이지를 지원한다. 예를 들어 64-bit 리눅스에서는 프로세스당 총 128TB의 가상 주소 공간을 사용할 수 있으며, 약 64TB의 물리 메모리를 사용할 수 있다.

아래는 /proc/meminfo 인터페이스를 사용해 메모리 관련 정보를 파악하는 예시이다.

```
05:41:11 ~ → grep MemTotal /proc/meminfo
MemTotal:      16275324 kB
05:41:19 ~ → grep VmallocTotal /proc/meminfo
VmallocTotal:  34359738367 kB
05:41:34 ~ → grep Huge /proc/meminfo
AnonHugePages: 3475456 kB
ShmemHugePages: 0 kB
FileHugePages:  0 kB
HugePages_Total: 0
HugePages_Free: 0
HugePages_Rsvd: 0
HugePages_Surp: 0
Hugepagesize:    2048 kB
Hugetlb:        0 kB
```

그림 2.5: /proc/meminfo 실습

1. 물리적 메모리에 대한 세부 정보 나열한다. 약 16GB이다.
2. 가상 메모리에 대한 세부 정보를 나열한다. 약 34TB이다.
3. 대형 페이지 정보를 나열한다. 여기서 페이지 크기는 2MB이다.

2.3.3 네트워킹

리눅스의 네트워크 스택은 계층화된 아키텍처를 따른다.

- 소켓

추상화 커뮤니케이션을 위해 필요

- TCP(Transmission Control Protocol) 및 UDP(User Datagram Protocol)
각각 연결형 통신과 비연결형 통신

- 인터넷 프로토콜(IP)

기기의 주소 지정을 위해 필요

이와 같은 세 가지 작업은 커널이 처리하는 모든 것이다. HTTP나 SSH 같은 애플리케이션 계층 프로토콜은 주로 사용자 영역에서 구현된다.

2.3.4 파일시스템

리눅스는 파일시스템을 사용해 HDD, SSD, 플래시 메모리 같은 저장 디바이스의 파일과 디렉터리를 구성한다. ext4, btrfs, NTFS 같은 다양한 용형의 파일시스템이 있으며 동일한 파일 시스템의 인스턴스도 여러 개 사용할 수 있다.

가상 파일시스템(Virtual File System, VFS)은 원래 여러 파일시스템 유형과 인스턴스를 지원하기 위해 도입되었다. VFS의 최상위 계층은 열기, 닫기, 읽기, 쓰기 기능 등 공통 API 추상화를 제공하며, VFS의 최하위 계층은 주어진 파일시스템에 대한 **플러그인**이라고 불리는 파일시스템 추상화다.

2.3.5 디바이스 드라이버

드라이버는 커널에서 실행되는 코드이다. 그 역할을 키보드, 마우스, HDD 같은 실제 하드웨어 디바이스나 /dev/pts/ 아래의 의사 터미널 같은 의사 디바이스(pseudo-device)를 관리하는 것이다. 의사 디바이스는 가상의 디바이스지만 실제 디바이스처럼 취급될 수 있다.

드라이버는 커널에 정적으로 빌드될 수도 있고, 필요할 때 동적으로 로드될 수 있도록 커널 모듈로 빌드될 수 있다.

2.3.6 시스템 콜

터미널에 `touch test.txt`를 입력하거나 앱 중 하나가 원격 시스템에서 파일 컨텐츠를 다운로드하길 원한다면 결국에는 '파일 생성'이나 '어떤 주소에서 모든 바이트 읽기'와 같은 상위 수준의 명령을 일련의 구체적인 아키텍처 종속 단계로 전환하도록 리눅스에 요청하게 된다. 즉, 커널이 노출하는 서비스 인터페이스와 해당 사용자 영역의 엔티티 호출은 시스템 호출의 모음이라 하며, 시스템 콜이라 부른다.

리눅스에 수백개의 시스템 콜이 존재하며, 일반적으로 이러한 시스템 콜을 직접 호출하는 대신 C 표준 라이브러리라는 것을 통해 호출한다. 표준 라이브러리는 wrapper 기능을 제공하여 glibc나 musl 같은 다양한 구현체에서 사용할 수 있다.

wrapper 라이브러리는 시스템 콜 실행의 반복적인 저수준 처리를 다룬다는 중요한 작업을 수행한다. 시스템 콜은 소프트웨어 인터럽트로 구현되기 때문에 예외 처리기로 제어권을 넘기는 예외를 발생시킨다. 시스템 콜이 호출될 때마다 처리해야 할 여러 단계가 존재하며 아래의 예시를 보면서 짚어보자.

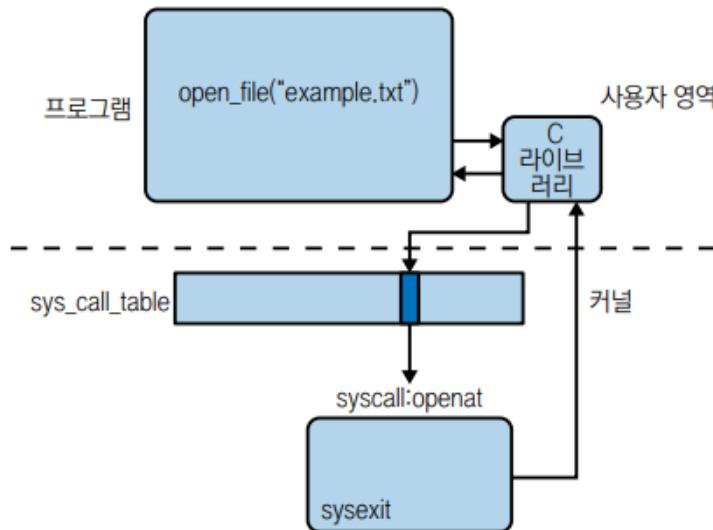


그림 2.6: 리눅스의 시스템 콜 실행 단계

1. syscall.h와 아키텍처 종속 파일(커널은 시스템 콜 테이블이라 부르는 파일 사용)에 정의되어 메모리에 있는 함수 포인터 배열(sys_call_table이라는 변수에 저장됨)을 통해 시스템 콜과 해당 처리기를 추적한다.
2. 시스템 콜 멀티플렉서처럼 동작하는 `system_call()` 함수를 실행하면 먼저 하드웨어 컨텍스트를 스택에 저장한 다음 검사를 수행하고(ex - 추적이 되는지 여부), 그 이후에 `sys_call_table`의 각 시스템 콜 번호의 index가 가리키는 함수로 점프한다

3. sysexit로 시스템 콜이 완료되면 wrapper 라이브러리는 하드웨어 컨텍스트를 복원하고, 프로그램 실행은 사용자 영역에서 다시 시작된다.

위와 같은 단계에서 주목해야 할 것은 시간이 많이 소요되는 작업인 커널 모드와 사용자 영역 모드간의 전환이다.

시스템 콜이 실제로 어떻게 사용되는지 보는 구체적 예시를 보자. **strace**를 사용해 내부 동작을 확인해보자.

그림 2.7: ls 내부 동작

1. strace에 ls가 사용하는 시스템 콜을 기억하도록 요청한다. 실제로는 상당히 길기 때문에 생략된 예시이다.
 2. execve 시스템 콜은 /usr/bin/ls를 실행해 쉘 프로세스를 교체한다.
 3. brk 시스템 콜은 메모리를 할당하는 오래된 방식이다. malloc을 사용하는 것이 더 안전하고 이식성이 좋다. malloc은 시스템 콜이 아니며, mallocopt를 사용하여 접근한 메모리 양에 따라 brk 시스템 콜 혹은 mmap 시스템 콜을 사용해야 하는지 여부를 결정하는 함수이다.
 4. access 시스템 콜은 프로세스가 특정 파일에 접근할 수 있는지 확인한다.
 5. openat 시스템 콜은 O_RDONLY—O_CLOEXEC 플래그(마지막 인수)를 사용해 디렉터리 파일 디스크립터(여기서 첫 번째 인수인 AT_FDCWD, 현재 디렉터리를 나타냄)와 관련된 /etc/ld.so.cache 파일을 연다.
 6. read 시스템 콜은 파일 디스크립터(첫 번째 인수, 3)로 부터 832바이트(마지막 인수)를 버퍼(두 번째 인수)로 읽는다.

```

$ strace -c \
          curl -s https://mhausenblas.info > /dev/null
% time      seconds   usecs/call     calls    errors syscall
----- -----
 26.75  0.031965        148       215           mmap
 17.52  0.020935        136       153           3 read
 10.15  0.012124        175        69           rt_sigaction
  8.00  0.009561        147        65           1 openat
  7.61  0.009098        126        72           close
...
  0.00  0.000000          0         1           prlimit64
-----
100.00  0.119476        141      843        11 total

```

그림 2.8: curl 수행 시간 분석

strace는 사용자 영역과 커널 사이에 이벤트 라이브 스트림을 가로채는 방식으로 어떤 시스템 콜이 어떤 순서로 어떤 인수를 사용해 호출되었는지 정확하게 파악하기 위해 유용하다. strace는 또한 성능 진단에도 유용하다. curl 명령이 어디서 대부분의 시간을 소비하는지 살펴보자.

1. -c 옵션을 사용하여 사용된 시스템 콜의 개요 통계를 작성한다.
2. curl의 출력물 모두 버린다.

아래의 표는 커널 구성요소와 시스템 전체에 걸쳐 널리 사용되는 시스템 콜 목록이다. [매뉴얼 페이지](#)를 참고하여 보다 상세한 시스템 콜 정보를 확인할 수 있다.

카테고리	시스템 콜 예시
프로세스 관리	clone, fork, execve, wait, exit, getpid, setuid, setns, getrusage, capset, ptrace
메모리 관리	brk, mmap, munmap, mremap, mlock, mincore
네트워킹	socket, setsockopt, getsockopt, bind, listen, accept, connect, shutdown, recvfrom, recvmsg, sendto, sethostname, bpf
파일시스템	기타
시간	time, clock_settime, timer_create, alarm, nanosleep
시그널	kill, pause, signalfd, eventfd
전역	uname, sysinfo, syslog, acct, _sysctl, iopl, reboot

표 2.1: 시스템 콜 예시

2.4 커널 확장

대체로 일상적인 작업에 필요하지 않지만, 커널 고급 주제이다. 주로 커널을 확장하는 방법에 중점을 둔다.

쉬운 내용부터 시작하자. 현재 사용중인 커널을 다음의 명령을 사용해 간단히 확인할 수 있다.

```
06:16:39 ~ ➔ uname -srn
Linux 5.15.133.1-microsoft-standard-WSL2 x86_64
```

그림 2.9: uname 예제

- 이 uname 출력값을 통해 5.15 커널을 사용하고 있음을 알 수 있다.

이제 커널 버전을 알았으니, 커널 소스 코드에 기능을 추가해서 빌드하지 않고도 커널 외부로 확장하는 방법에 대한 문제를 해결할 수 있다. 이렇게 확장하려면 모듈을 사용할 수 있다.

모듈

모듈은 요청 시 커널에 로드할 수 있는 프로그램이다. 즉, 커널을 다시 컴파일하거나 시스템을 재부팅할 필요가 없다. 최근 리눅스는 대부분의 하드웨어를 자동으로 감지해 해당 모듈을 자동으로 로드한다. 하지만, 수동으로 로드하고 싶은 경우도 있다. 예를 들어 제조사의 모듈 대신 서드파티 모듈을 사용하려는 경우이다.

다음 명령을 실행하면 실행 가능한 모듈을 나열할 수 있다.

```
$ find /lib/modules/$(uname -r) -type f -name '*.ko*'
/lib/modules/5.11.0-25-generic/kernel/ubuntu/ubuntu-host/ubuntu-host.ko
/lib/modules/5.11.0-25-generic/kernel/fs/nls/nls_iso8859-1.ko
/lib/modules/5.11.0-25-generic/kernel/fs/ceph/ceph.ko
/lib/modules/5.11.0-25-generic/kernel/fs/nfsd/nfsd.ko
...
/lib/modules/5.11.0-25-generic/kernel/net/ipv6/esp6.ko
/lib/modules/5.11.0-25-generic/kernel/net/ipv6/ip6_vti.ko
/lib/modules/5.11.0-25-generic/kernel/net/sctp/sctp_diag.ko
/lib/modules/5.11.0-25-generic/kernel/net/sctp/sctp.ko
/lib/modules/5.11.0-25-generic/kernel/net/netrom/netrom.ko
```

그 다음 실제로 로드한 모듈을 확인해보자.

```
$ lsmod
Module           Size  Used by
...
linear          20480  0
crc32_pclmul    16384  1
crc32_clmulni_intel 16384  0
ghash_clmulni_intel 16384  0
virtio_net      57344  0
net_failover    20480  1 virtio_net
ahci            40960  0
aesni_intel    372736  0
crypto_simd     16384  1 aesni_intel
cryptd          24576  2 crypto_simd,ghash_clmulni_intel
glue_helper     16384  1 aesni_intel
```

이 정보는 /proc/modules를 통해 볼 수 있다는 것을 주목하라. 이는 의사 파일시스템 인터페이스를 통해 정보를 노출하는 커널 덕분이다. 다음은 커널을 확장하는 현대적 대안인 eBPF이다.

커널을 확장하는 현대적인 방법: eBPF

요즘 커널 기능 확장으로 eBPF(아무 의미 없음)가 인기를 끌고 있다. 원래 BPF(Berkeley Packet Filter)로 불렸다.

기술적으로 eBPF(아무 의미 없음)는 리눅스 커널의 기능이며, 이를 활용하기 위해 커널 3.1.5 이상 버전이 필요하다. 시스템 콜을 사용해 커널 기능을 안전하고 효율적으로 확장한다. eBPF는 맞춤형 RISC 64-bit ISA를 사용한 커널 내 가상 머신으로 구현되었다.

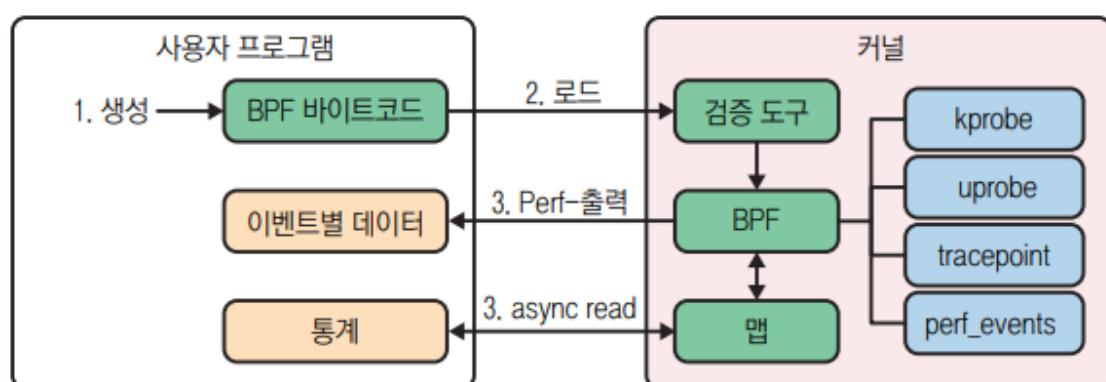


그림 2.10: 리눅스 커널의 eBPF 개요

eBPF는 이미 다양한 곳에서 사용되고 있다.

- **쿠버네티스에서 포드 네트워킹(pod networking)을 활성화하기 위한 CNI 플러그인**

대표적인 예로, 실리움과 프로젝트 칼리코에서 사용된다. 또한 서비스 확장성을 위해 서도 사용된다.

- **관측가능성용**

iovisor/bpftrace 같은 리눅스 커널 추적과 허블(Hubble)을 사용한 클러스터 설정을 위해 사용된다.

- **보안 제어 역할**

컨테이너 런타임 스캔을 수행할 때 사용할 수 있다.

- **네트워크 로드밸런싱용**

로드밸런서로도 사용할 수 있다.

3

셸과 스크립팅

3.1 기본 개요

터미널

터미널은 텍스트로 된 사용자 인터페이스를 제공하는 프로그램이다. 터미널은 키보드에서 문자를 읽어 화면에 표시하는 기능을 지원한다. 초창기(Unix 시절)에는 하나의 메인프레임에 접속하기 위한 모니터와 키보드가 통합된 장치였으나 현재는 앱일 뿐이다. 그런 흔적이 tty(teletype)로 /dev 하위에 남아있다.

셸

셸(shell)은 스트림을 통해 입력, 출력을 처리하고, 변수를 지원하며, 사용 가능한 내장(built-in) 명령이 몇가지 있으며, 명령 실행 상태 및 상태를 처리하고, 일반적으로 대화식 사용과 스크립트 사용을 모두 지원한다.

공식적으로 셸은 sh로 정의하며, POSIX 셸이라는 용어를 접하게 되는데, 이는 스크립트와 이식성 맥락에서 중요하다. 초기에는 창안자의 이름을 딴 본 셸(Bourne Shell, sh)이 많이 쓰였지만, 최근에는 bash 셸(Bourne again shell)을 주로 사용한다.

어떤 셸이 사용되고 있는지 궁금하면 file -h /bin/sh 명령을 사용해 확인해보고, 만약 이 명령이 실패하면 echo \$0 또는 echo \$SHELL을 시도해보자.

이어서 스트립과 변수라는 두 가지 기본 기능에서 출발해 셸의 기본 기능을 알아보자.

스트립

셸은 입력과 출력을 위한 세 가지 기본 파일 디스크립터(File Descriptor, FD)를 모든 프로세스에 제공한다.

- stdin (FD 0)
- stdout (FD 1)
- stderr (FD 2)

이 FD들은 그림 3.1에 나와 있는 것처럼 기본적으로 화면과 키보드에 각각 연결되어 있다. 즉 특별하게 지정하지 않는 한, 셸에 입력하는 명령은 키보드에서 입력(stdin)을 가져오고 출력(stdout)을 화면에 전달한다. 다음 셸의 상호작용에서 위의 기본 동작을 확인할 수 있다.

```
$ cat
This is some input I type on the keyboard and read on the screen^C
```

cat을 사용한 방금의 예에서 기본값이 동작하는 것을 확인할 수 있다. 그리고 Ctrl+c를 사용해 명령을 종료했다.

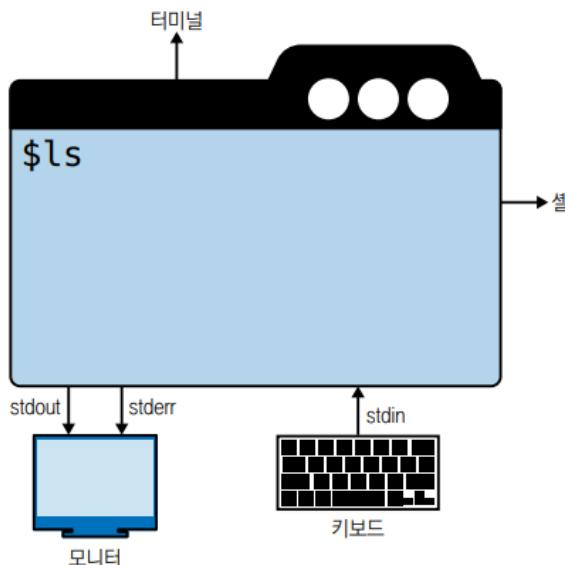


그림 3.1: 셸 I/O의 기본 스트림

셸이 제공하는 기본값을 사용하지 않으려면 스트림을 재지정(redirect)할 수 있다.

`$FD>`와 `<$FD`를 사용해 프로세스의 출력 스트림을 재지정할 수 있다. 여기서 `$FD`는 파일 디스크립터다. 예를 들어 `2>`는 `stderr` 스트림을 재지정한다는 의미다. `1>`는 `>`는

stdout이 기본값이므로 동일한 뜻이다. stdout과 stderr를 모두 재지정하려면 &>를 사용하고 스트림을 제거하려면 /dev/null을 사용하면 된다. 구체적인 예를 들어 이 사항이 어떻게 동작하는지 살펴보자. 아래는 curl을 통해 HTML 콘텐츠를 다운로드 하는 예제이다.

```
$ curl https://example.com &> /dev/null ❶

$ curl https://example.com > /tmp/content.txt 2> /tmp/curl-status ❷
$ head -3 /tmp/content.txt
<!doctype html>
<html>
<head>
$ cat /tmp/curl-status
% Total    % Received % Xferd  Average Speed   Time     Time     Time  Current
                   Dload  Upload   Total   Spent    Left  Speed
100 1256  100 1256    0      0  3187      0 --:--:-- --:--:-- --:--:--  3195
$ cat > /tmp/interactive-input.txt ❸

$ tr < /tmp/curl-status [A-Z] [a-z] ❹
% total    % received % xferd  average speed   time     time     time  current
                   dload  upload   total   spent    left  speed
100 1256  100 1256    0      0  3187      0 --:--:-- --:--:-- --:--:--  3195
```

1. stdout과 stderr을 모두 /dev/null로 재지정해 모든 출력값을 버린다.
2. 출력값과 상태값을 다른 파일로 재지정한다.
3. 대화식으로 값을 입력하고 파일에 저장한다. Ctrl+d를 사용해 캡처를 중지하고 콘텐츠를 저장한다.
4. stdin에서 값을 읽는 tr 명령을 사용해 모든 단어를 소문자로 만든다.

셸은 일반적으로 다음과 같은 여러 특수 문자를 이해한다.

- **앰피샌드(&)**
명령 마지막에 배피되면 백그라운드에서 명령을 실행한다.
- **백슬래시(\)**
긴 명령의 가독성을 높이기 위해 다음 행에서 명령을 계속할 때 사용한다.
- **파이프(|)**
한 프로세스의 stdout 값을 다음 프로세스의 stdin과 연결해 데이터를 파일에 임시로 저장하지 않고 바로 전달할 수 있다.

파이프는 매우 많은 사항을 내포하고 있다. 파이프와 관련된 사항은 다음 [유닉스 파이프](#)를 참고하고 유닉스와 마이크로서비스 사이 유사성을 다룬 [2018년 유닉스 철학 재조명](#) 기사도 참고하면 좋다.

변수

값을 하드코딩하고 싶지 않거나 불가능할 때는 언제든 변수를 사용해 값을 저장하고 변경할 수 있다. 사용 예는 다음과 같다.

- 리눅스가 노출하는 구성 항목을 처리하려는 경우 (ex - 셸이 \$PATH 변수에 저장된 실행 파일을 찾는 위치). 이는 변수를 읽기/쓰기할 수 있는 일종의 인터페이스다.
- 스크립트에서 사용자에게 값을 대화 형식으로 질문하려는 경우
- 긴 값을 한 번 정의해 입력을 줄이려는 경우(ex - HTTP API의 URL). 이 사용 사례는 변수를 선언한 후 값을 변경하지 않기 때문에 대략적으로 프로그램 언어의 const 값에 해당한다.

변수는 다음과 같이 두 가지 종류로 나뉜다.

- **환경변수**

셸 전체의 설정. env 명령어로 목록을 나열한다.

- **셸 변수**

현재 실행 상황에서 유효하다. bash에서 set 명령어로 목록을 나열할 수 있다. 하위 프로세스는 셸 변수를 상속하지 않는다.

배시에서 export 명령어를 사용해 환경 변수를 만들 수 있다. 변수의 값에 접근하고 싶을 때는 앞에 \$를 붙이고, 변수를 제거하고 싶을 때는 unset을 이용한다.

실제로 어떻게 보이는지 살펴보자.

```

$ set MY_VAR=42 ❶
$ set | grep MY_VAR ❷
_=MY_VAR=42

$ export MY_GLOBAL_VAR="fun with vars" ❸

$ set | grep 'MY_*' ❹
MY_GLOBAL_VAR='fun with vars'
_=MY_VAR=42

$ env | grep 'MY_*' ❺
MY_GLOBAL_VAR=fun with vars

$ bash ❻
$ echo $MY_GLOBAL_VAR ❻
fun with vars

$ set | grep 'MY_*' ❽
MY_GLOBAL_VAR='fun with vars'

$ exit ❾
$ unset $MY_VAR
$ set | grep 'MY_*' ❿
MY_GLOBAL_VAR='fun with vars'

```

1. MY_VAR라는 셀 변수를 생성하고 값을 42로 지정한다.
2. 셀 변수를 나열하고 MY_VAR를 필터링 한다. 환경변수로 내보내지 않았음을 나타내는_=에 유의하자.
3. MY_GLOBAL_VAR라는 새 환경변수를 만든다.
4. 셀 변수를 나열하고 MY_로 시작하는 모든 변수를 필터링해본다. 예상대로 이전 단계에서 만든 두 변수가 모두 표시된다.
5. 환경변수를 나열한다. 예상대로 MY_GLOBAL_VAR이 표시된다.
6. 새 셀 세션, 즉 MY_VAR을 상속하지 않는 현재 셀 세션의 자식 프로세스를 만든다.
7. 환경변수 MY_GLOBAL_VAR에 접근한다.
8. 현재 자식 프로세스에 있기 때문에 셀 변수를 나열하면 MY_GLOBAL_VAR만 나온다.
9. 자식 프로세스를 종료한 후 MY_VAR 셀 변수를 제거하고 셀 변수를 나열한다. 예상대로 MY_VAR이 사라졌다.

변수	유형	의미
EDITOR	POSIX	파일 편집 시 기본적으로 사용되는 프로그램의 경로
HOME	배시 셸	현재 사용자의 홈 디렉터리 경로
HOSTNAME	POSIX	현재 호스트의 이름
IFS	POSIX	필드를 구분할 문자 목록, 셸이 단어를 확장해서 분할할 때 사용함
PATH	POSIX	셸이 실행 가능한 프로그램을 찾는 디렉터리 목록을 포함
PS1	환경	사용 중인 주 프롬프트 문자열
PWD	환경	작업 디렉터리의 전체 경로
OLDPWD	배시 셸	마지막 cd 명령을 실행하기 직전의 디렉터리 경로
RANDOM	배시 셸	0에서 32767 사이의 임의의 정수
SHELL	환경	현재 사용되는 셸
TERM	환경	사용하는 터미널 에뮬레이터
UID	환경	현재 사용자의 고유 ID(정수)
USER	환경	현재 사용자 이름
-	배시 셸	foreground에서 실행된 이전 명령의 마지막 인수
?	배시 셸	종료 상태
\$	배시 셸	현재 프로세스의 ID(정수)
0	배시 셸	현재 프로세스의 이름

표 3.1: 일반적인 셸 변수와 환경변수

표 3.1에 일반적인 셸과 환경 변수를 정리했다. 이 환경변수들은 거의 모든 곳에서 찾을 수 있으며, 잘 이해하고 사용하는 것이 중요하다. 모든 변수에 대해 echo \$XXX 명령을 사용하면 해당 값을 볼 수 있는데 여기서 XXX는 변수 이름이다.

종료상태

셸은 종료 상태(exit status)라고 하는 것을 사용해 명령 실행 완료를 명령 호출자에게 알린다. 일반적으로 리눅스 명령은 종료될 때 상태를 반환한다. 이는 정상적인 종료 (원활한 경로 또는 happy path라고 부름)일 수도 비정상 종료일 수도 있다. 종료 상태값 0은 명령이 오류 없이 성공적으로 실행되었음을 의미하는 반면 1에서 255 사이의 값은 실패를 나타낸다. 종료 상태를 확인하려면 echo \$?를 사용한다.

일부 셸에서는 마지막 상태값만 사용할 수 있으므로 파이프 사용 시 종료 상태 처리에 주의해야 한다. \$PIPESTATUS를 사용하면 이런 제약사항을 해결할 수 있다.

내장 명령어

셸에는 여러 내장 명령어(built-in command)가 있다. 몇 가지 유용한 예로는 yes, echo, cat, read가 있다. help 명령을 사용하면 내장 명령어 목록을 나열할 수 있다. 그러나 그 외 모든 것은 보통 /usr/bin(사용자 명령의 경우)나 /usr/sbin(관리 명령의 경우)에 있는 셸 외부 프로그램이라는 점을 기억하자. 그러면 실행 파일을 어디에서 찾을지 어떻게 알 수 있을까? 다음과 같은 방법이 있다.

```

10:44:27 ~ → which ls
/usr/bin/ls
11:49:27 ~ → type ls
ls is aliased to `__util_alias_init_ls'
11:49:30 ~ →
11:49:53 ~ → █

```

그림 3.2: 실행 파일 위치 확인 예시

작업 제어

작업 제어는 대부분의 셸이 지원하는 기능이다. 기본적으로 명령을 입력하면 그 명령은 일반적으로 화면과 키보드를 제어하며, ‘foreground에서 실행된다’고 한다. 프로세스를 백그라운드에서 시작하려면 명령 마지막에 &를 넣고, foreground 프로세스를 백그라운드로 보내려면 Ctrl+z를 누르면 된다.

```

$ watch -n 5 "ls" & ❶
$ jobs ❷
Job      Group      CPU      State      Command
1        3021      0%      stopped    watch -n 5 "ls" &
$ fg ❸
Every 5.0s: ls                                         Sat Aug 28 11:34:32 2021
Dockerfile
app.yaml
example.json
main.go
script.sh
test

```

1. 명령 끝에 &를 넣으면 백그라운드에서 명령이 실행된다.
2. 모든 작업(job)의 목록을 출력한다.
3. fg 명령을 사용하면 프로세스를 foreground로 가져올 수 있다. watch 명령을 종료하려면 Ctrl+c를 사용한다.

셸을 닫은 후에도 백그라운드 프로세스를 계속 실행하려면 nohup 명령을 앞에 추가하면 된다. 또한, 이미 실행 중이지만 앞에 nohup이 붙지 않은 프로세스의 경우에는 이미 실행 이후라도 disown을 사용하면 동일한 효과를 얻을 수 있다. 마지막으로 실행 중인 프로세스를 제거하려면 다양한 수준의 강제성과 함께 kill 명령을 사용할 수 있다.

모던 리눅스 명령어

자주 사용되는 명령 중 일부는 대체될 수 있는 모던 명령어가 존재한다. drop-in 교체도 존재하고, 일부는 기능을 확장한 것도 존재한다. 다만, 어떤 도구를 선택할지 고민이 된다면 사용하는 리눅스 배포판에서 이미 검증된 도구를 사용하는 것이 가장 좋은 방법이다.

책과 별개로 개인적인 의견을 남긴다. 아래 제공되는 모던 리눅스 명령어는 배포판에 default로 설치되는 경우가 아닌 경우가 많다. 따라서 설치 후 사용해야 하는 경우가 많고, 이는 일반적인 스크립트 작성에 활용하기 어렵다는 것이다. 즉, 일반적으로 활용은 어렵다.

결론적으로, 실제 활용하기엔 제약이 존재하므로 책의 모든 부분을 작성할 계획은 없으며 간략한 소개에 그칠 것이다.

exa로 디렉터리 내용 나열하기

디렉터리에 무엇이 들어 있는지 알고 싶으면 ls 혹은 그 변형 중 하나를 매개변수와 함께 사용한다. 이에 대응하는 모던 리눅스 명령어가 있는데 exa이다.

bat로 파일 내용 보기

cat 명령어의 대안으로 bat 명령어 사용을 고려해볼 수 있다. bat 명령은 구문을 강조하여 표시할 수 있으며 자체 pager 기능이 존재한다.

	File: main.go
1	package main
2	
3	import (
4	"fmt"
5	"net/http"
6)
7	
8	func main() {
9	http.HandleFunc("/", HelloServer)
10	http.ListenAndServe(":8080", nil)
11	}
12	
13	func HelloServer(w http.ResponseWriter, r *http.Request) {
14	fmt.Fprintf(w , "Hello, %s!", r .URL.Path[1:])
15	}
	File: app.yaml
1	apiVersion: apps/v1
2	kind: Deployment
3	metadata:
4	name: something
5	+ namespace: xample
6	spec:
7	selector:
8	matchLabels:
9	app: sample
10	replicas: 2
11	template:
12	metadata:
13	labels:
14	app: sample
15	spec:
16	containers:
17	- name: example
18	image: public.ecr.aws/mhausenblas/example:stable

그림 3.3: bat으로 랜더링한 Go 파일과 YAML 파일

일반 작업

행 탐색과 조작

셀 프롬프트에 명령을 입력할 때는 행을 탐색하거나 행을 조작하는 등 다양한 작업을 한다. 아래의 표는 일반적으로 사용되는 셀의 단축키 목록이다.

동작	명령어	비고
행의 시작으로 커서 이동	Ctrl+a	-
행의 마지막으로 커서 이동	Ctrl+e	-
커서를 한 문자 앞으로 이동	Ctrl+f	-
커서를 한 문자 뒤로 이동	Ctrl+b	-
커서를 한 단어 앞으로 이동	Alt+f	왼쪽 Alt에서만 작동
커서를 한 단어 뒤로 이동	Alt+b	-
현재 문자 삭제	Ctrl+d	-
커서 왼쪽 문자 삭제	Ctrl+h	-
커서 왼쪽 단어 삭제	Ctrl+w	-
커서 오른쪽의 모든 항목 삭제	Ctrl+k	-
커서 왼쪽의 모든 항목 삭제	Ctrl+u	-
화면 지우기	Ctrl+l	clear 명령과 동일
명령어 취소	Ctrl+r	SIGINT 시그널 전송
실행 취소	Ctrl+_	bash만 해당
기록 검색	Ctrl+r	일부 셸만 해당
검색 취소	Ctrl+g	기록 검색에 대응함

표 3.2: 단축키 목록

모든 단축기가 모든 셸에서 지원되는 것은 아니며 셸마다 다르게 구현될 수 있다. 이런 단축기는 Emacs 편집 키 입력을 기반으로 한다. 만약 vi 키 입력을 기반으로 하고 싶다면 .bashrc 파일에 **set -o vi**를 사용하면 된다.

파일 내용 관리

vi 같은 편집기를 사용하지 않고도 텍스트를 편집할 수 있다.

```
10:33:42 ~ ➔ echo "First line" > /tmp/something
10:33:55 ~ ➔ cat /tmp/something
First line
10:34:03 ~ ➔ echo "Second line" >> /tmp/something && \
> cat /tmp/something
First line
Second line
10:34:36 ~ ➔ sed 's/line/LINE/' /tmp/something
First LINE
Second LINE
10:34:55 ~ ➔ cat << 'EOF' > /tmp/another
> First line
> Second line
> Third line
> EOF
10:35:25 ~ ➔ diff -y /tmp/something /tmp/another
First line
Second line
First line
Second line
> Third line
```

1. echo 출력을 재지정해 파일을 생성한다.
 2. 파일 내용을 확인한다.
 3. 파일에 한 행을 추가한 후 내용을 확인한다.
 4. sed를 사용해 파일 내용을 바꾸고 stdout으로 출력한다.
 5. redirection을 사용해 파일을 생성한다.
 6. 생성한 파일의 차이점을 보여준다.

긴 파일 보기

셀의 한 화면에 표시할 수 없을 만큼 행 수가 많은 파일의 경우 less 또는 bat과 같은 pager를 사용할 수 있다. 페이지 나누기(paging) 기능을 활용하면 프로그램은 출력을 분할해서 화면에 나타낼 수 있는 분량에 맞게 페이지를 나눠 표시하고, 각 페이지를 탐색할 수 있는 명령어(다음 페이지 보기, 이전 페이지 보기 등)를 제공한다.

긴 파일을 처리하는 또 다른 방법으로 **head**와 **tail** 명령어가 있다. 예를 들어 파일의 시작 부분을 출력하고 싶다면 다음과 같이 실행한다.

```
$ for i in {1..100} ; do echo $i >> /tmp/longfile ; done ❶
$ head -5 /tmp/longfile ❷
1
2
3
4
5
```

1. 긴 파일을 생성한다(100줄)
2. 긴 파일의 처음 다섯 행을 출력한다.

지속적으로 추가되는 파일의 실시간 업데이트를 받으려면 다음을 사용하면 된다.

```
$ sudo tail -f /var/log/Xorg.0.log ❶
[ 36065.898] (II) event14 - ALPS01:00 0911:5288 Mouse: device is a pointer
[ 36065.900] (II) event15 - ALPS01:00 0911:5288 Touchpad: device is a touchpad
[ 36065.901] (II) event4 - Intel HID events: is tagged by udev as: Keyboard
[ 36065.901] (II) event4 - Intel HID events: device is a keyboard
...
...
```

1. tail을 이용해 로그 파일의 마지막을 출력한다. 여기서 -f 옵션은 따르다(follow)라는 의미로, 결과를 지속적으로 확인하거나 자동 업데이트함을 의미한다.

날짜와 시간 처리

date 명령은 고유한 파일 이름을 생성할 때 유용하다. 유닉스 타임스탬프 등 여러 형식으로 날짜를 생성하고 다양한 날짜와 시간 형식 간에 변환할 수도 있다.

```
10:19:14 ~ → date +%s  
1707743961  
10:19:21 ~ → date -d @1707743961 '+%m/%d/%Y:%H:%M:%S'  
02/12/2024:22:19:21
```

1. 유닉스 타임스탬프를 생성한다.
2. 유닉스 타입스탬프를 사람이 읽을 수 있는 날짜로 변환한다.