리눅스 내부 구조

리눅스의 역사

- 리눅스는 **UNIX** 표준에 기반한 현대적인, 무료 운영 체제
- **1991**년에 **Linux Torvalds** 에 의해 **UNIX** 호환의 커널 설계 목적으로 처음 개발
- 리눅스의 역사는 인터넷을 통한 전 세계의 수 많은 사용자에 의한 협동 작업의 결과
- 범용 **PC** 하드웨어 상에서 효율적이고 신뢰성 있게 동작하도록 설계되었지만, 다양한 하드웨어 플랫폼에서도 잘 동작
- 핵심 리눅스 운영 체제 커널은 고유의 코드를 가지지만 많은 무료 **UNIX** 소프트웨어를 실행시켜 무료 **UNIX**-호환 운영 체제로 발돋움

리눅스 커널(Linux Kernel)

- 버전 **0.01 (May 1991)**
  - 네트웍 기능이 없고
  - **80386**-호환 인텔 프로세서와 **PC** 하드웨어에서만 동작
  - 극히 제한된 장치-구동기 지원
  - **Minix** 파일 시스템만 지원.
- **Linux 1.0 (March 1994)** 이 포함하는 새로운 특성들:
  - **UNIX** 의 표준 **TCP/IP** 네트웍 프로토콜들 지원
  - **BSD**-호환 소켓 인터페이스 **(**네트웍 프로그래밍**)**
  - 이더넷 위에서 **IP** 를 실행시키기 위한 장치 구동기 지원
  - 강화된 파일 시스템
  - 고성능 디스크 접근을 위한 **SCSI** 제어기 지원
  - 추가 하드웨어 지원
- 버전 **1.2 (March 1995)** 는 최종 **PC-only Linux kernel.**

리눅스 커널 2.0

- **June 1996** 배포, **2.0** 은 두 가지 주요한 기능 추가:
  - 다양한 **CPU** 구조 지원- **64-bit native Alpha** 칩을 완전히 지원
  - 다중 프로세서 구조 지원
- 다른 새로운 기능들:
  - 개선된 메모리-관리 코드
  - 개선된 **TCP/IP** 성능
  - 적재 가능한 모듈 사이의 의존성을 처리하고 모듈의 자동 적재를 위해 내부 커널 스레드 지원
  - 표준화된 설정 인터페이스
- **Motorola 68000** 시리즈, **Sun Sparc, PPC, Arm** 등 지원

리눅스 커널 2.2

- **January 1999** 발표
- 추가 기능
  - **SMP**에서의 성능 향상
  - **MCA** 버스 지원
  - **PCI** 서브시스템 변경
  - 집 드라이브**(Zip drive,** 병렬 포트를 통해 연결되는 **IDE** 장치**) IrDA(**적외선 통신**)** 지원
  - 다양한 파일 시스템 추가 **- NTFS(**읽기전용**), FAT32, MS Joliet, HFS, ROMFS, autofs**
  - **Unix98** 방식의 **pty**같은 **Unix98** 표준 반영
  - **IP** 방화벽**(firewall)**은 **IP** 체인**(chains)**으로 대체
  - 네트워킹 성능 향상

리눅스 커널 2.4

- **January 2001** 발표
- 주요 기능
  - 엔터프라이즈급**(enterprise level)**의 커널
    - 커널의 여러 제한을 없애고 대용량 작업 가능
    - 동시 실행 프로세스가 많을 때의 스케줄링이 더욱 효율적으로 수행
    - 사용자/그룹 개수 **: 40**억명**(32**비트**),** 물리 메모리**: 64GB**
    - **16**개의 이더넷 카드와 **10**개의 **IDE** 컨트롤러 사용 가능
    - 파일 크기 무제한 및 프로세스 숫자 무한대
  - 하드웨어 지원 확대
    - 완전한 **PnP** 지원 **- ISAPnP** 포함**(ISA** 장치들이 **PCI** 처럼 **PnP** 기능 지원**)**
    - 자원 분배 문제 해결 - 디바이스 파일 시스템**(DevFS)** 도입
    - **CPU** 지원 확충 **- IA-64(Intel), SH(Super Hitachi) CPU** 포함
    - 문자 장치 지원 **- I2O, PCMCIA, USB, IEEE1394(firewire), UDMA** 지원 강화
    - 블록 장치 지원 **– LVM(Logical Volume Manager), IDE Floppy/tape/DVD** 지원
  - 기타 기능
    - 프로세스간 통신**(IPC)** - 공유메모리 방식이 **POSIX** 표준과 호환되도록 변경
    - 네트워킹 기능 강화
    - 커널 레벨 웹 데몬 지원 **- kHTTPd**

리눅스 커널 2.6

- **December 2003** 안정버전 배포
- 특징
  - 핵심 하드웨어 지원
    - 임베디드 기기를 위한 크기 축소 - **uCLinux(MMU** 지원없는 리눅스) 통합으로 **Hitachi H8/300, NEC v850, m68k** 계열 지원
    - 서버 기기 등을 위한 크기 확대 - **NUMA((Non-Uniform Memory Access)** 지원, **64** 비트 **CPU**들 지원
    - 하이퍼스레딩 **– Pentium 4** 이상에서 동작, 하나의 **CPU**로 둘 이상 동작
  - 시스템 성능 개선
    - 디바이스 지원 개수 증가 – 종래 **256**개에서 **4096**개로
    - 프로세스 **ID,** 사용자, 그룹 수 증가
      - 프로세스 **ID** - 종래 **32768** 개에서 **10**억개
      - 사용자, 그룹 수 - 종래 **65536**개에서 **40**억개
    - 응답성 향상
      - 커널의 선점형 동작 - 입출력 인터럽트(시스템 호출) 시 더 높은 우선 순위 프로세스가 실행을 준비하고 있으면 선점 가능
      - **O(1)** 스케줄러 사용 - 우선 순위에 따른 다중 준비 큐 지원
      - **FUTEX(fast userspace mutex)** 지원

리눅스 커널 2.6

- ◈ 모듈 서브 시스템과 통합 장치 모델 **(Unified Device Model)**
  - ➢ 모듈 서브 시스템의 안정성 향상
  - ➢ 디바이스 정보 지원
    - – 디바이스가 **/sys**에 마운트되어 디바이스 속성/개수/이름/ **IRQ/DMA**/전원공급 상태 등 확인 가능.
    - – 모듈 지원 하드웨어 정보를 모듈 밖에 공유하여 대체 디바이스로 강제 동작 가능.
  - ➢ 전원 관리 기능 지원 – **ACPI** 지원
  - ➢ 데스크탑 주변 장치 지원 **– PCI, USB, Firewire** 등 다양한 장치의 **hotplug** 지원
- ◈ 시스템 하드웨어 지원
  - ➢ **PnP BIOS** 기능, 다양한 전용 하드웨어 지원
  - ➢ 다양한 외부 디바이스 지원 **– USB 2.0** 등
  - ➢ 다양한 무선 디바이스 지원 **– IrDA, Bluetooth** 등
- ◈ 블록 디바이스 지원
  - ➢ 다양한 장치 버스 지원 **– ATA/IDE, SCSI, EDD BIOS** 지원
  - ➢ 다양한 파일 시스템 지원 **– ext2/ext3, Journaling File Systems(ReiserFS, JFS, XFS), LDM(NTFS, FAT32** 등 **Windows** 계열**)** 등 지원

리눅스 커널 2.6

- 입출력 지원
  - 휴먼 인터페이스 장치 지원 개선 – 향상된 모듈화, 비디오 장치, 마우스, 키보드 등 지원 확대
  - 사운드 시스템 변경 – **ALSA(Advanced Linux Sound Architecture)** 지원
- 소프트웨어 향상
  - 다양한 통신 프로토콜 지원 – **ipsec, ipv6** 지원
  - 네트웍 파일 시스템**(NFS)** 강화 – **NFSv4** 프로토콜 지원, **AFS** 지원, **CIFS** 클라이언트 기능 지원
- 기타 기능
  - 보안 기능 – 커널 기반 보안 모듈 지원, 모듈의 시스템 호출 불가능
  - 사용자 모드 리눅스**(User Mode Linux)** 지원 – 리눅스 상에서 사용자 모드로 리눅스 커널 실행 가능, 즉 복수의 리눅스 실행 가능
  - 새로운 커널 설정 환경 – **Kconfig** 도입(종래의 **CML1/2** 과 다름)

리눅스 시스템

- 리눅스는 버클리의 **BSD** 운영체제, **MIT** 의 **X** 윈도우 시스템, **FSF** 의 **GNU** 프로젝트들이 제공하는 많은 도구 사용
- 주 시스템 라이브러리는 **GNU** 프로젝트에 의해 시작되었고 리눅스 커뮤니티에 의해 개선됨.
- 리눅스 네트웍-관리 도구들은 **4.3BSD** 코드로부터 가져옴; 최근 **FreeBSD** 등에서는 거꾸로 리눅스로부터 코드 가져옴
- 리눅스 시스템은 인터넷 상의 개발자들의 느슨한 망에 의해 유지.

리눅스 배포판

- 표준, 미리 컴파일된 패키지 셋, 또는 배포판들은 기본 리눅스 시스템, 시스템 설치 및 관리 도구들, 공통 **UNIX** 도구들의 인스톨 패키지들 포함
- 처음 배포판들에서 패키지들은 모든 파일들을 적절한 위치에 푸는 수단; 현대의 배포판들은 고급 패키지 관리 포함
- 초기 배포판은 **SLS** 와 **Slackware** 등. ***Red Hat*** 과 ***Debian*** 이 대표적인 상용 및 비상용 배포판.
- **RPM** 패키지 파일 포맷은 다양한 리눅스 배포판에서 호환

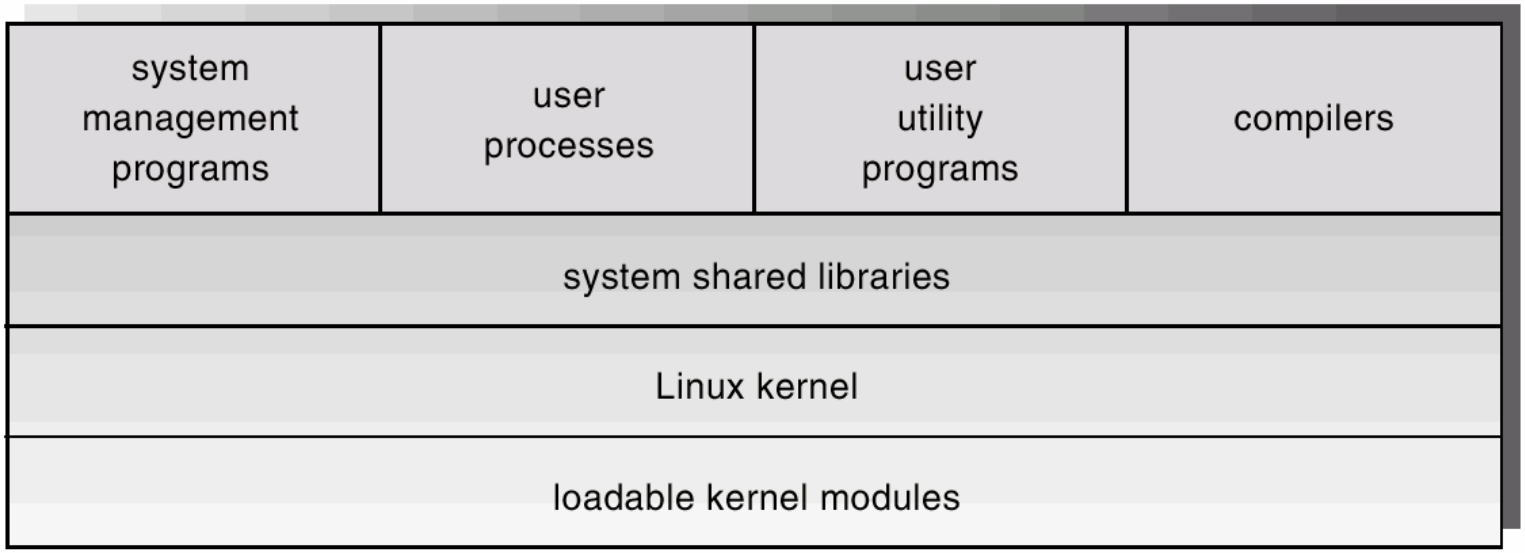
리눅스 라이센스

- 리눅스 커널은 **General Public License (GPL)** 하에서 배포
- **GPL** 은 **Free Software Foundation (FSF)** 가 정함
- 리눅스를 사용하는 누구든지 상용 제품을 만들 수 없다; **GPL** 하에 나온 소프트웨어는 바이너리 제품으로 배포될 수 없다.

설계 원칙

- 리눅스는 **UNIX**-호환 도구들의 풀셋을 가지는 멀티유저, 멀티태스킹 시스템.
- 파일 시스템은 전통적인 **UNIX** 형식을 따르고 표준 **UNIX** 네트웍 모델을 완벽하게 구현.
- 주 설계 목표는 속도, 효율, 표준화.
- 리눅스는 관련된 **POSIX** 문서를 따르도록 설계.
- 리눅스 프로그래밍 인터페이스는 **BSD** 보다 **SVR4 UNIX** 형식을 따름

리눅스 시스템 요소

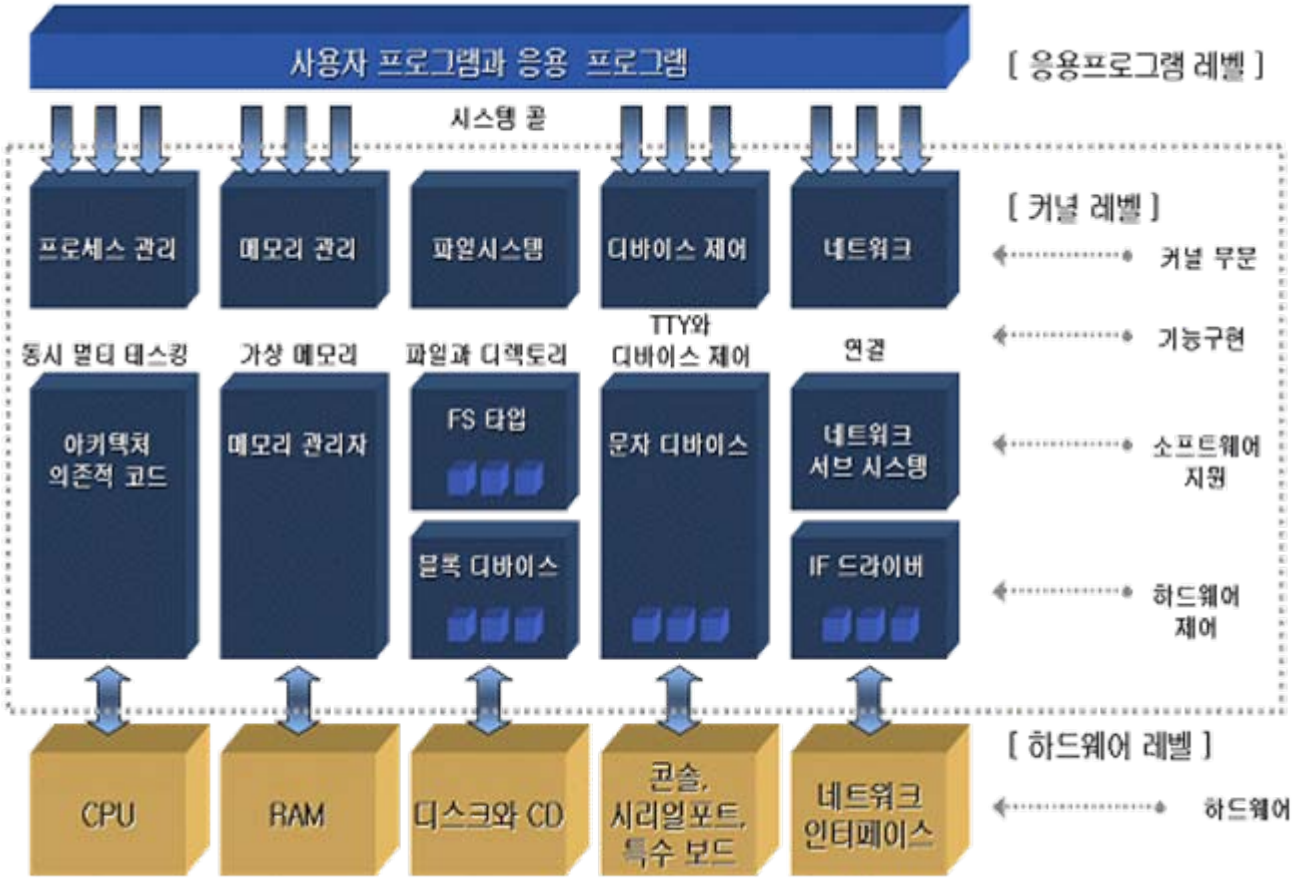

system management programs
user processes
user utility programs
compilers
system shared libraries
Linux kernel
loadable kernel modules

리눅스 시스템 요소

- 대부분의 **UNIX** 구현처럼, 리눅스는 세 가지 주요 코드 부분으로 구성
- 커널 **kernel** 은 운영 체제의 중요 추상화를 유지
  - 커널 코드는 컴퓨터의 모든 물리 자원에 대한 접근 가능하며 *커널 모드* ***kernel mode*** 에서 실행
  - 모든 커널 코드와 데이터 구조는 단일 주소 공간에 유지.
- 시스템 라이브러리 **system libraries**
  - 응용이 커널과 상호 작용하는 표준 함수 집합 정의
  - 많은 운영 체제 기능 구현
  - 기본 시스템 호출들의 많은 복잡한 버전 제공
- 시스템 유틸리티 **system utilities**
  - 개별적인 특정 관리 태스크(예, **daemon**) 수행

리눅스 커널

▣ 리눅스 커널 내부 구조

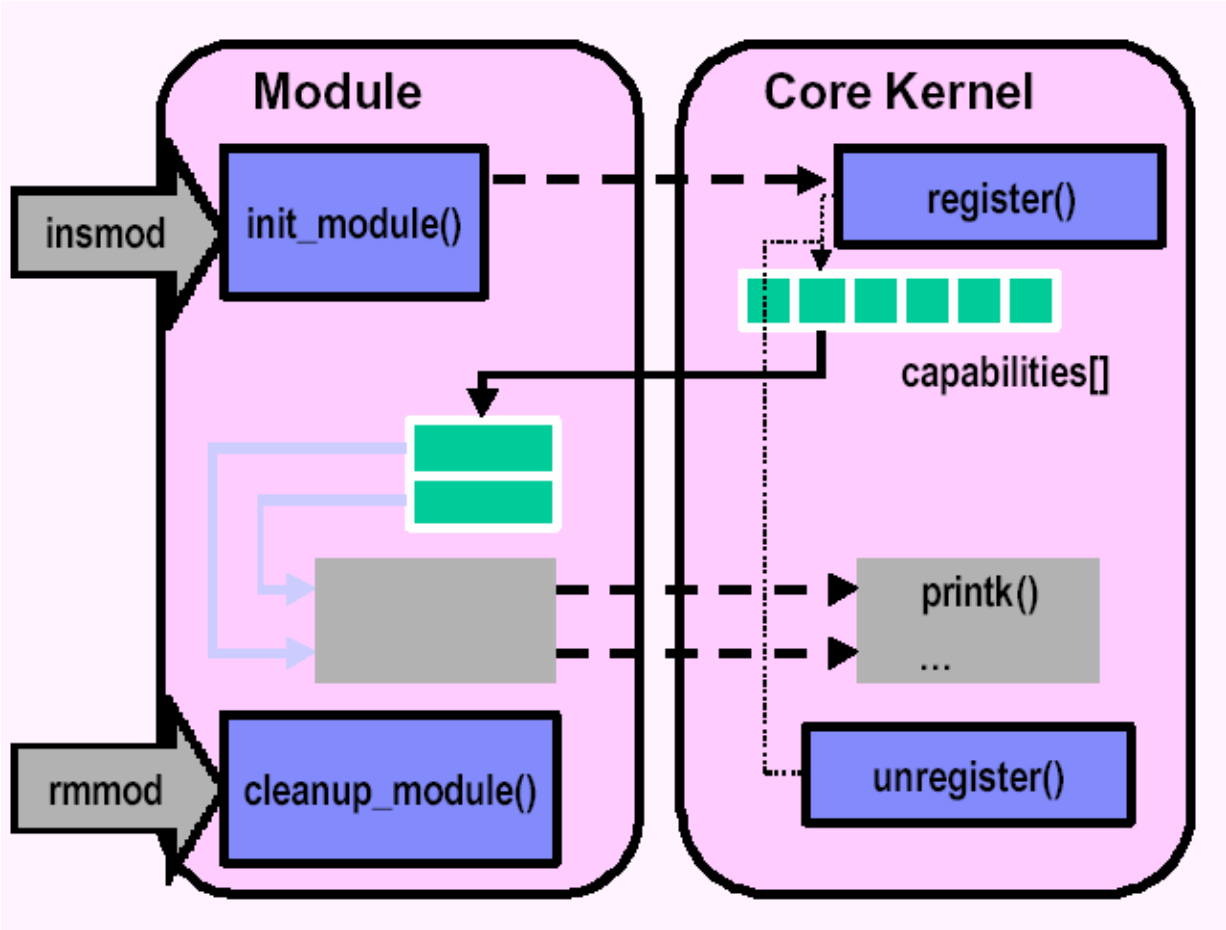


커널 모듈

- 커널의 다른 부분과 독립적으로 컴파일되고, 적재되고, 제거될 수 있는 커널 코드의 일부분
- 장치 구동기, 파일 시스템, 네트웍 프로토콜로 구현
- 모듈 인터페이스를 통해 제 3자가 **GPL** 하에서 배포될 수 없는 장치 구동기/파일 시스템을 작성, 배포 가능
- 리눅스 커널 모듈은 추가 장치 구동기 없이 표준 최소 커널과 함께 설정 허용
- 모듈 지원의 **3** 요소:
  - 모듈 관리
  - 드라이버 등록
  - 충돌 해결

커널 모듈

- 모듈의 동작 과정



모듈 관리

- 메모리로의 모듈 적재와 커널과의 결합 지원
- 모듈 적재의 두 부분:
  - 커널 메모리에서 모듈 코드 부분의 관리
  - 모듈에게 참조를 허용하는 심볼 처리
- 모듈 요청자 **(module requestor)**
  - 요청되었으나 적재되지 않은 모듈의 적재 관리
  - 동적으로 적재된 모듈이 사용되고 있는지, 모듈이 더 이상 사용되고 있지 않으면 제거할지 커널에게 문의

드라이버 등록

- 드라이버 등록은 새로운 드라이버가 사용 가능하다는 것을 전체 커널에게 알림.
- 커널은 모든 알려진 드라이버의 동적 테이블을 유지하고 아무 때나 이 테이블로 드라이버를 추가하거나 제거하는 루틴들 제공
- 등록 테이블:
  - **Device drivers**
  - **File systems**
  - **Network protocols**
  - **Binary format**

충돌 해결

- 하드웨어 자원을 예약하고 또 다른 드라이버의 사용으로부터 보호하도록 하는 매커니즘
- 충돌 해결 모듈의 목적:
  - 모듈들의 하드웨어 자원 접근에 대한 충돌 방지
  - 자동검출(*autoprobes*)의 기존의 디바이스 드라이버와의 인터페이스 방지
  - 같은 하드웨어에 접근하는 다수 드라이버들의 충돌 해결

프로세스 관리

- **UNIX** 프로세스 관리
  - 프로세스 생성과 새로운 프로그램 실행을 2개의 구별된 동작으로 분리
    - **fork** 시스템 호출 – 새로운 프로세스 생성
    - **execve** 시스템 호출 – 새로운 프로그램 실행
  - 프로세스는 운영 체제가 단일 프로그램의 실행 문맥 추적을 위해 유지해야 하는 모든 정보 포함
- **Linux** 프로세스 관리
  - **UNIX** 프로세스 관리 기능 상속
  - 프로세스 속성의 세 그룹: 프로세스의 **identity,** 환경, 문맥.

프로세스 인식

- **Process ID (PID)**
  - 프로세스에 대한 고유 인식자
  - 응용 프로그램이 다른 프로세스에게 신호를 주고, 수정하고, 기다리도록 하는 시스템 호출을 하도록 운영체제가 지정하는 데 사용.
- **Credentials.**
  - 모든 프로세스는 관련된 사용자 **ID** 와 하나 이상의 그룹 **ID** 를 가짐
  - 시스템 자원과 파일에 접근하기 위한 프로세스의 권한 결정
- **Personality.**
  - 전통적인 **UNIX** 시스템에는 없었으나, 리눅스에서는 각 프로세스가 어떤 시스템 호출의 의미를 약간 수정할 수 있는 관련 **presonality identifier** 를 가짐.
  - 주로 에뮬레이션 라이브러리가 시스템 호출이 **UNIX** 의 어떤 특징과 부합되는지의 요청에 대해 사용.

프로세스 환경

- 프로세스 환경
  - 부모로부터 상속
  - 2개의 널 문자로 종료되는 벡터로 구성
    - 인자 벡터(**argument vector**) – 실행 파일을 호출하는 데 사용되는 커맨드라인 인자 목록; 보통 프로그램 이름 자신으로 시작
    - 환경 벡터(**environment vector**)- 임의의 문자열 값을 가지는 환경 변수들에 해당하는 **"NAME=VALUE"** 쌍의 목록.
  - 프로세스간 환경 변수 전달과 자식 프로세스에 의한 변수 상속은 사용자-모드 시스템 소프트웨어에서 정보 전달의 유연한 수단.
  - 환경 변수 방식은 시스템 전체가 아닌 프로세스/사용자 기반 설정이므로 운영 체제의 전용화 제공.

프로세스 문맥

- 어떤 시점에서의 실행 프로그램의 (계속 변하는) 상태
  - **scheduling context –** 프로세스 문맥 중 가장 중요한 부분; 스케줄러가 프로세스를 중지하거나 재시작할 때 필요로 하는 정보.
  - 커널은 각 프로세스가 현재 소모하는 자원과 지금까지 소모한 전체 자원에 대한 계산된**(accounting)** 정보를 유지
  - 파일 테이블**(file table) –** 커널 파일 구조에 대한 포인터 배열; 파일 입출력 시스템 호출을 할 때, 프로세스는 이 테이블 인덱스로부터 파일 참조.
  - 파일 시스템 문맥**(file-system context) –** 파일 테이블은 기존의 개방된 파일 목록인 반면, 이것은 새로운 파일을 개방하는 요청에 적용. 새로운 파일 검색을 위해 사용되는 현재 루트와 디폴트 디렉토리들이 여기 저장됨.
  - **signal-handler table –** 특정한 시그널이 도착했을 때 호출되는 프로세스 주소 공간 내의 루틴 지정.
  - **virtual-memory context –** 프로세스의 개인 주소 공간의 전체 내용 묘사

프로세스와 스레드

- **Linux** 는 프로세스와 스레드에 대한 같은 내부 표현 사용
  - 스레드는 부모와 같은 주소 공간을 공유하는 그저 새로운 프로세스
- 새로운 스레드가 **clone** 시스템 호출로 생성될 때는 구별됨
  - **fork** 시스템 호출
    - 완전히 새로운 프로세스 문맥을 가진 프로세스 생성
  - **clone** 시스템 호출
    - 자신의 **ID** 를 가진 새로운 프로세스를 생성하지만, 부모의 데이터 구조를 공유
    - 두 스레드 사이에 공유되는 것에 대한 응용 프로그램의 세밀한**(fine-grained)** 제어 가능.

스케줄링

- 운영 체제 내에서 태스크들에게 **CPU** 시간을 할당하는 작업.
- 리눅스 스케줄링
  - 프로세스의 실행과 인터럽트는 물론, 다양한 커널 태스크의 실행도 포함.
  - 커널 태스크 실행
    - 실행 프로세스가 요청하는 태스크와 디바이스 드라이버 대신에 내부적으로 실행하는 태스크.

커널 동기화

- 커널 모드 실행 요청:
  - 실행 프로그램은 시스템 호출이나 페이지 오류 발생에 의한 운영 체제 서비스 요청.
  - 디바이스 드라이버가 하드웨어 인터럽트를 전달하여 커널에서 정의된 인터럽트 핸들러 실행 시작.
- 커널의 임계 영역이 다른 임계 영역에 의해 인터럽트되지 않고 실행되기 위해 커널 동기화 요구.

커널 동기화

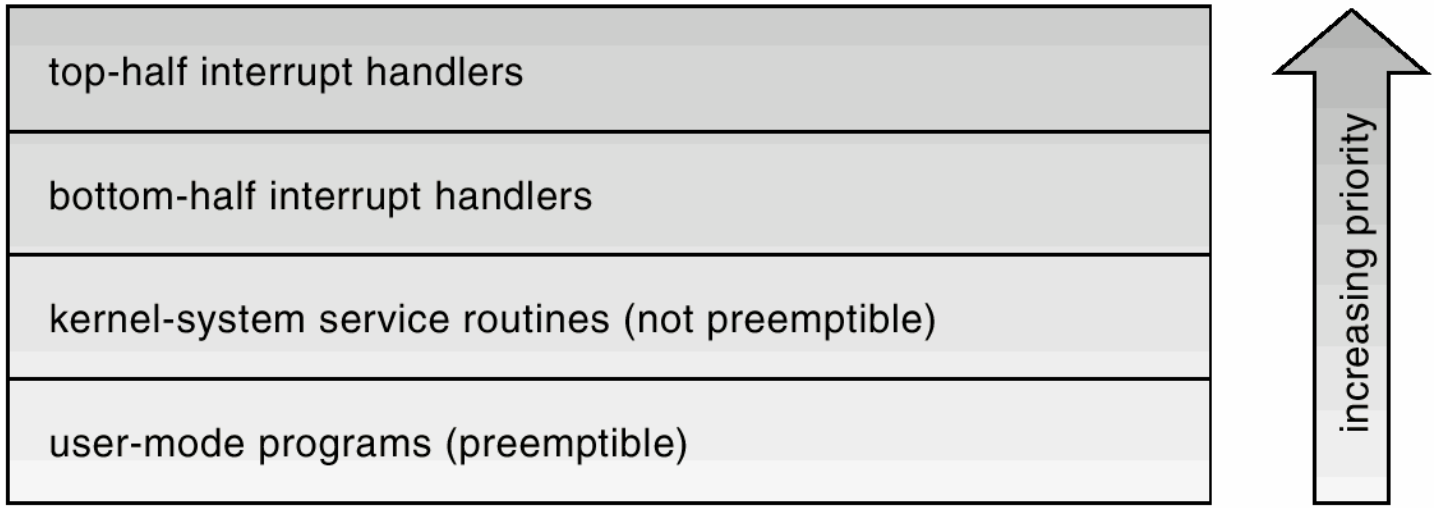
- 리눅스의 임계 영역(**critical section**) 보호 기법:
  1. 일반 커널 코드는 비선점(**nonpreemptible**)
     - 프로세스가 커널 시스템 서비스 루틴을 실행하는 동안 타이머 인터럽트가 수신되는 경우, 커널의 **need_resched** 플래그가 설정되고 스케줄러는 시스템 호출이 끝나자마자 스케줄러가 실행되고 사용자 모드로 전환.
  2. 인터럽트 서비스 루틴에서 발생하는 임계영역에 적용
     - 프로세서의 인터럽트 제어 하드웨어를 사용하여 임계 영역에서 인터럽트를 비활성화함으로써 커널은 공유 데이터 구조의 동시 접근의 위협 없이 진행 가능.

커널 동기화

- 성능 패널티를 피하는 동기화 구조 사용
  - 리눅스 커널은 긴 임계 영역동안 인터럽트를 비활성화하지 않고 실행하도록 허용.
- 인터럽트 서비스 루틴은 상반부(***top half***) 와 하반부(***bottom half***) 으로 구별.
  - 상반부(**top half**) – 일반 인터럽트 서비스 루틴; 재귀 인터럽트가 가능하지 않은 상태로 실행.
  - 하반부(**bottom half**) – 모든 인터럽트가 가능한 채로 실행; 자기 자신은 인터럽트되지 않음을 작은 스케줄러가 확인.
  - 보통의 포그라운드 커널 코드를 실행하는 동안 선택된 하반부 인터럽트를 비활성화하는 메커니즘 사용.

인터럽트 보호 단계

- 각 단계는 더 높은 단계에서 실행하는 코드에 의해 인터럽트 가능하지만, 같거나 낮은 단계에서 실행하는 코드에 의해 인터럽트되지 않음.
- 시간-공유 스케줄링 인터럽트가 발생하면 사용자 프로세스는 다른 프로세스에 의해 항상 선점 가능.



프로세스 스케줄링

- 리눅스는 2 가지 프로세스-스케줄링 알고리즘 사용:
  - 시간 공유(time-sharing) 알고리즘 - 다중 프로세스들 사이의 공평한 선점 스케줄링
  - 실시간(real-time) 알고리즘 – 절대적인 우선순위가 공평함보다 더 중요한 태스크들
- 프로세스의 스케줄링 클래스는 어떤 알고리즘을 적용할지를 결정.
- 시간 공유 프로세스
  - 리눅스는 우선순위, 신용(credit) 기반 알고리즘 사용.
  - 신용화(crediting) 규칙은 프로세스의 히스토리와 우선순위 반영

$$\text{credits} := \frac{\text{credits}}{2} + \text{priority}$$

  - 자동적으로 대화형 또는 I/O-바운드 프로세스를 우선순위화.

프로세스 스케줄링

- 리눅스는 **FIFO** 와 라운드 로빈**(round-robin)** 실시간 스케줄링 클래스 구현;
  - 각 프로세스는 스케줄링 클래스에 추가로 우선순위 가짐.
  - 스케줄러는 가장 높은 우선순위 프로세스를 실행; 똑같은 우선순위 프로세스들 중에서는 가장 오래 기다린 프로세스 실행
  - **FIFO** 프로세스는 종료하거나 정지**(block)** 될 때까지 실행 계속
  - 라운드-로빈 프로세스는 잠시 선점된 후 스케줄링 큐의 끝으로 이동하여, 같은 우선순위 라운드-로빈 프로세스들이 자동적으로 시간-공유됨.

대칭 멀티프로세싱

- **Linux 2.0** 커널에서 **SMP** 하드웨어 지원;
  - 독립된 프로세스 또는 스레드가 분산 프로세서 위에서 병렬로 실행 가능.
- **SMP** 제약 조건
  - 단일 커널 스핀락**(spinlock)** 을 사용하여 커널의 비선점 동기화를 보존
    - 오직 한 번에 하나의 프로세서만 커널-모드 코드 실행 가능.

메모리 관리

물리 메모리-관리

리눅스 물리 메모리 관리 시스템

- 페이지/페이지 그룹/작은 메모리 블록 할당 및 해제 처리
- 실행 프로세스의 주소 공간으로의 메모리 매핑된 가상 메모리 처리에 대한 추가 메커니즘 포함.

페이지 할당자(**page allocator**)

- 모든 물리 페이지 할당과 해제; 요청 시에 물리적으로 연속인 페이지 범위 할당 가능.

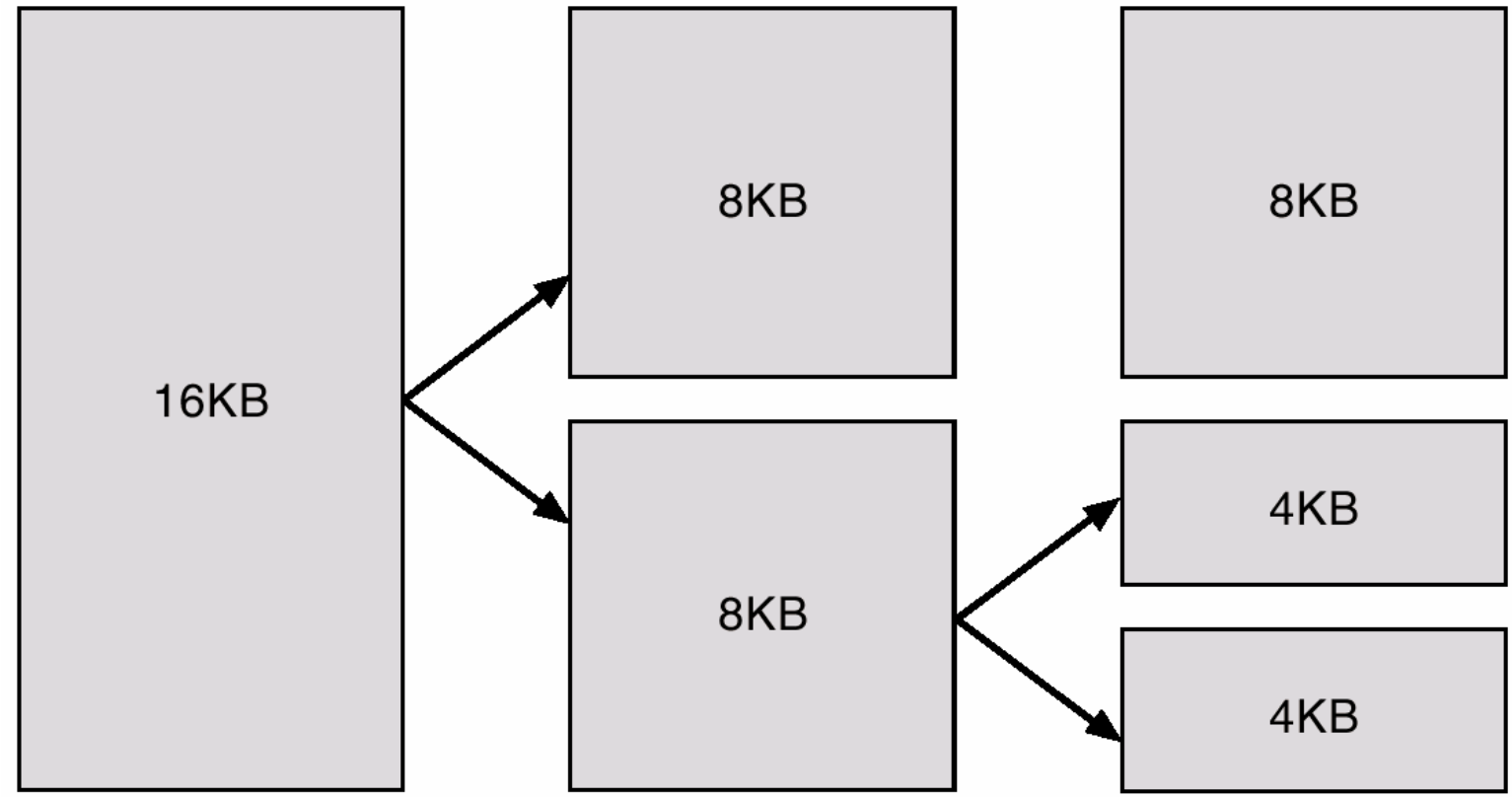
buddy-heap 알고리즘 사용

- 할당자는 사용 가능한 물리 페이지 추적을 위해 사용.
- 각 할당 가능한 메모리 영역은 옆 파트너와 쌍을 이룸.
- **2**개의 할당된 파트너 영역이 둘 다 해제되면, 결합하여 더 큰 영역 형성.
- 작은 메모리 요청에 대해 기존의 작은 빈 영역을 할당할 수 없으면, 더 큰 빈 영역이 두 부분으로 나누어짐.

리눅스 커널의 메모리 할당

- 정적(드라이버는 시스템 부팅 시 연속 메모리 영역 예약) 또는 동적(페이지 할당자를 통해)으로 발생.

Buddy Heap 으로 메모리 분리



가상 메모리

- 리눅스 가상 메모리 시스템
  - 각 프로세스가 볼 수 있는 주소 공간 유지
  - 요청 시 가상 메모리 페이지들을 생성, 이들 페이지들을 디스크로부터 적재 또는 디스크로 스와핑 아웃 관리.
- 가상 메모리 관리자
  - 프로세스 주소 공간의 2 가지 관점 유지:
    - 주소 공간의 레이아웃에 관한 명령을 묘사하는 논리적 관점.
      - 주소 공간은 겹치지 않는 페이지 단위 영역들의 집합으로 구성.
    - 프로세스의 하드웨어 페이지 테이블에 저장된 각 주소 공간의 물리적 관점.

가상 메모리

- 가상 메모리 영역의 특성
  - 저장소(**backing store**) – 페이지 영역들이 저장되는 곳; 영역들은 파일이나 형식 없이(***demand-zero* memory**) 저장
  - 쓰기에 대한 반응(**page sharing** 또는 **copy-on-write**).
- 커널은 새로운 가상 주소 공간 생성
  1. 프로세스가 **exec** 시스템 호출로 새로운 프로그램을 실행할 때
  2. **fork** 시스템 호출로 새로운 프로세스가 생성될 때
     - 부모 프로세스의 가상 주소 공간을 완전하게 복제.
     - 커널은 부모 프로세스의 **VMA descriptors**를 복사하고 자식을 위한 새로운 페이지 테이블 셋을 생성.
     - 부모의 페이지 테이블은 자식의 페이지 테이블로 직접 복사된다; 각 페이지의 참조 회수는 증가.
     - **fork** 한 후, 부모와 자식 프로세스는 주소 공간에서 같은 물리 메모리 페이지 공유.

가상 메모리

- **VM** 페이징 시스템
  - 메모리가 필요할 때 물리 메모리로부터 디스크로 메모리 페이지를 재배치.
  - **VM** 페이징 시스템의 두 부분:
    - **The pageout-policy algorithm –** 어느 페이지를 언제 디스크로 끄집어낼지 결정
    - 페이징 매커니즘 – 실제로 전송을 수행하고, 필요할 때 물리 메모리로 페이지 데이터를 다시 복원.

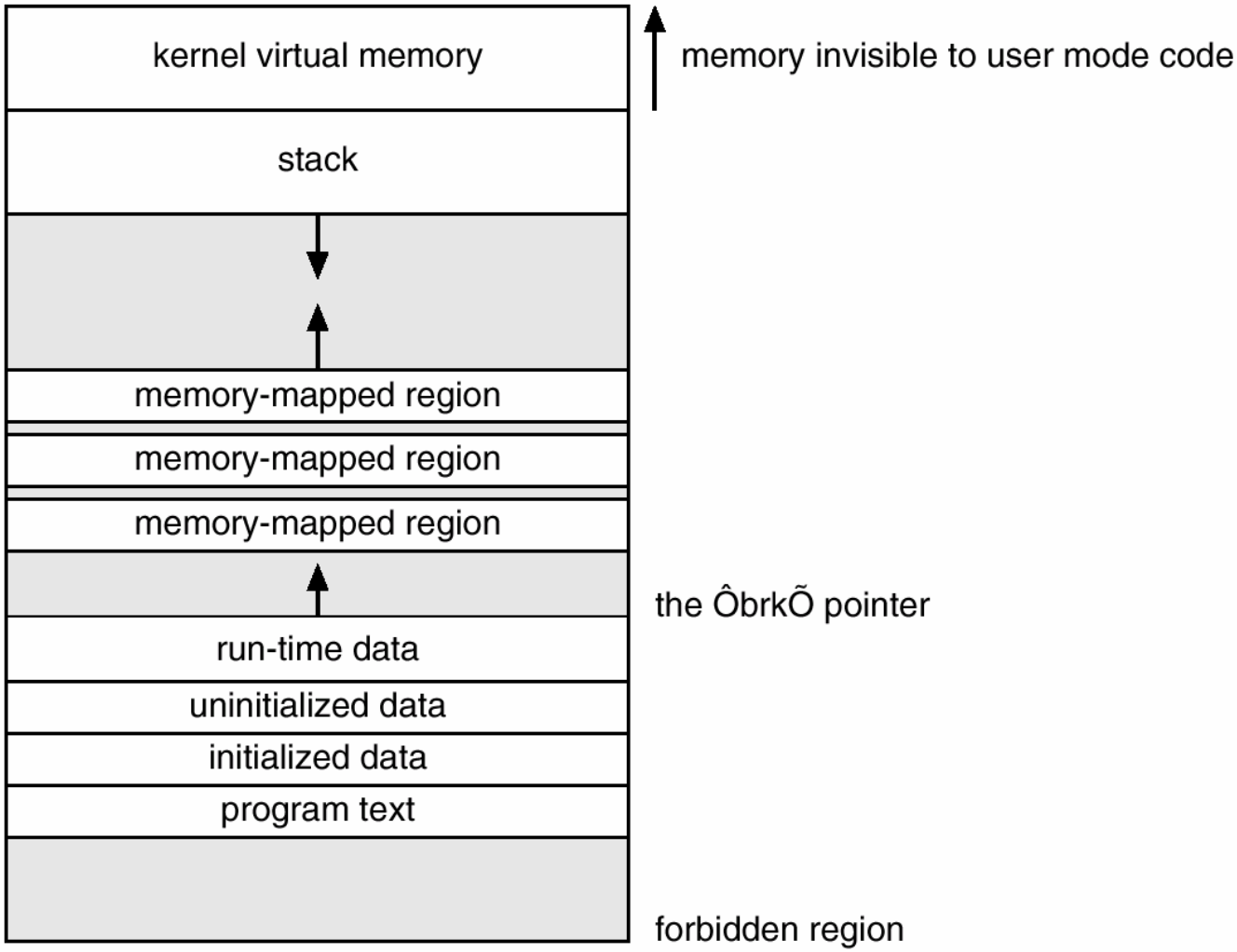
가상 메모리

- 리눅스 커널은 모든 프로세스의 가상 주소 공간 영역을 일정한, 구조-의존적인 영역으로 예약.
- 커널 가상-메모리 영역의 두 부분:
  - 정적 영역 - 시스템에서 모든 사용 가능한 물리 메모리 페이지에 대한 페이지 테이블 참조를 가지는 영역. 커널 코드를 실행할 때 물리 주소로부터 가상 주소로 쉽게 바꾼다.
  - 나머지 예약된 부분 – 이 페이지 테이블 항목들은 어떤 다른 메모리 영역을 가리키도록 수정될 수 있다.

사용자 프로그램 적재 및 실행

- 프로그램 적재를 위한 함수 테이블 관리
  - **exec** 시스템 호출이 일어날 때 주어진 파일을 적재할 수 있는 기회 제공.
- 다중 적재 루틴 등록
  - 리눅스는 **ELF** 와 **a.out** 이진 형식을 모두 지원.
- 처음, 이진-파일 페이지가 가상 메모리로 매핑된다; 프로그램이 물리 메모리에 없는 페이지에 접근할 때, 페이지 오류가 일어나고, 해당 페이지가 물리 메모리로 적재된다.
- **ELF**-형식 이진 파일
  - **ELF** 헤더와 몇몇 페이지-정렬 부분들로 구성
  - **ELF** 적재기는 헤더를 읽고 파일 부분들을 독립된 가상 메모리 영역으로 매핑한다.

ELF 프로그램에 대한 메모리 레이아웃



정적 및 동적 링킹

- 정적 링킹
  - 필요한 라이브러리 함수들이 직접 프로그램의 실행 가능한 이진 파일 내에 포함되는 경우
  - 단점 – 모든 생성 프로그램은 정확하게 같은 공용 시스템 라이브러리 함수들의 복사본을 포함.
- 동적 링킹
  - 시스템 라이브러리를 메모리로 한번만 적재
  - 물리 메모리와 디스크-공간 사용이 더욱 효율적.

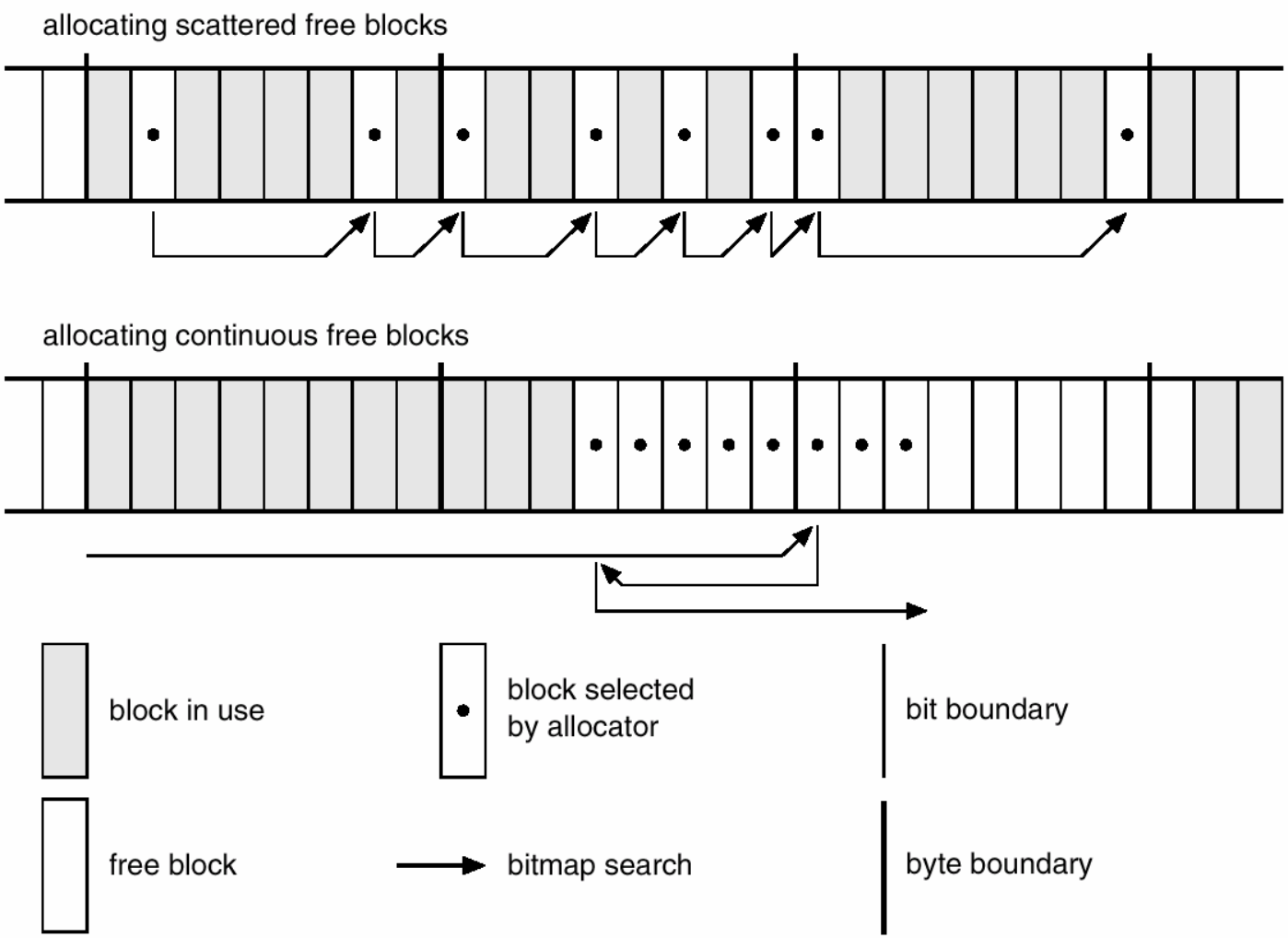
파일 시스템

- 리눅스 파일 시스템
  - **UNIX** 형태의 계층적인 트리 구조 채용.
- 가상 파일 시스템 ***virtual file system (VFS)***
  - 커널은 추상 계층 **VFS** 를 두어 상세한 구현을 숨기고 여러 가지 다른 파일 시스템을 관리
  - 객체 지향 원칙으로 설계
    - 파일 객체의 모습을 정의하는 정의 집합
      - *inode-object* 와 *file-object* 구조 : 개별 파일을 나타냄
      - *file system object* : 전체 파일 시스템을 나타냄
    - 이런 객체들을 처리하는 소프트웨어 계층.

리눅스 ext2 와 ext3 파일 시스템

- **ext2fs** 의 특정 파일에 속하는 데이터 블록 지정 방법
  - **BSD Fast File System (ffs)** 와 유사한 매커니즘 사용.
- **ext2fs** 와 **ffs** 와의 주요 차이점은 디스크 할당 정책
  - **ffs – 8Kb** 크기의 블록으로 디스크에 파일 할당, 각 블록은 작은 파일인 경우 **1Kb** 의 조각으로 나누어짐.
  - **ext2fs –** 조각은 사용하지 않으므로, 더 작은 단위로 할당 수행, 기본 블록 크기는 **1Kb**이며 **2Kb, 4Kb** 블록도 지원.
    - 논리적으로 이접한 파일 블록들을 디스크 상에 물리적으로 인접한 블록으로 두도록 설계된 할당 정책 사용. 그러므로, 몇몇 디스크 블록에 대한 **I/O** 요청을 단일 동작으로 수행 가능.
- **ext3** 파일 시스템
  - **ext2** 기반이지만 **Journaling File Systems**
    - 디렉토리와 파일 정보를 주기적인 검사 기록으로 보존하여 결함 포용
    - 시스템 결함으로 중단되어도 복구 가능하므로 데이터 무결성 보장

ext2fs 블록-할당 정책



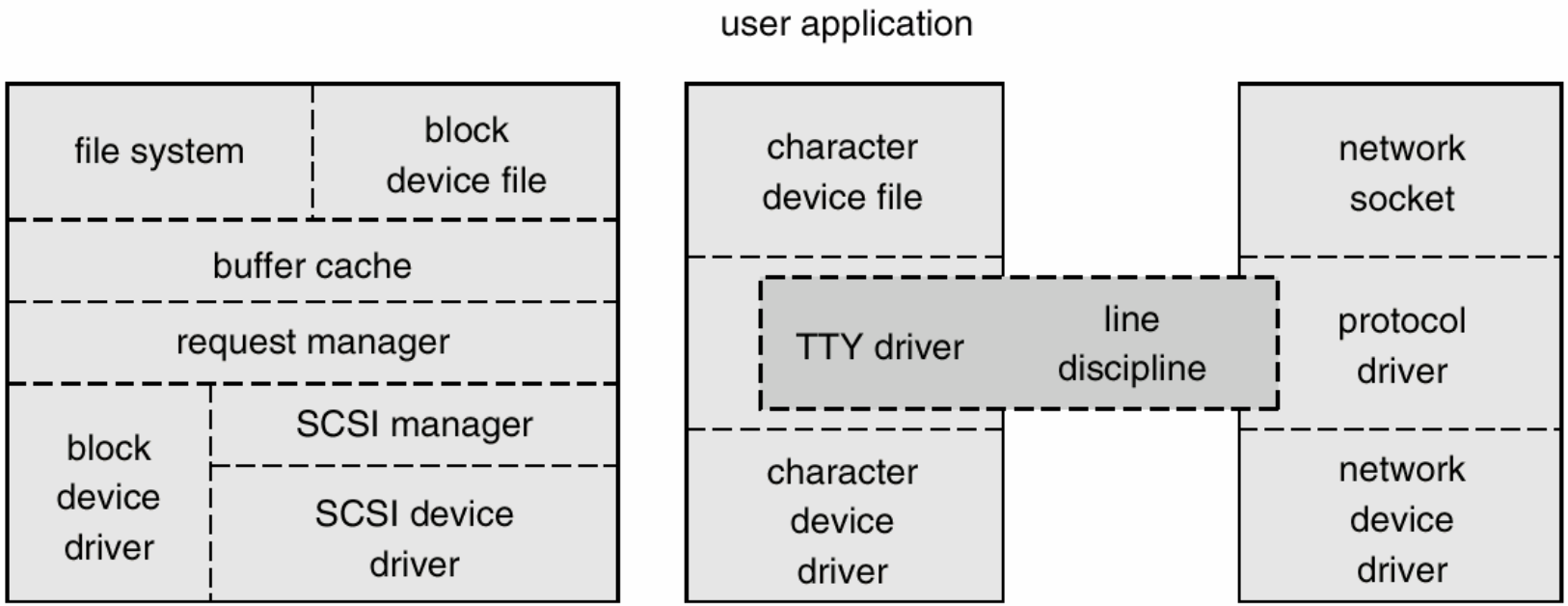
리눅스 proc 파일 시스템

- proc 파일 시스템
  - 데이터를 저장하는 것이 아니라, 사용자 파일 입출력 요청에 따라 내용이 계산됨.
- 디렉토리 구조와 내부 파일들 구현
  - 각 디렉토리와 내부 파일들의 고유하고 지속적인 **inode** 번호 정의해야 함.
  - 사용자가 특정 파일을 읽거나 디렉토리에 대한 검색을 수행할 때 어떤 동작이 수행되어야 하는지를 지정하기 위해 **inode** 번호 사용.
  - 파일로부터 데이터를 읽을 때, proc 은 적절한 정보를 모아, 텍스트 형식으로 바꾸고, 요청 프로세스의 **read** 버퍼에 넣는다.

입력과 출력

- 리눅스 디바이스-지향 파일 시스템은 두 가지 캐쉬를 통해 디스크 스토리지 접근
  - 데이터는 가상 메모리 시스템과 통합된 페이지 캐쉬**(page cache)**에 저장.
  - 메타데이터는 물리 디스크 블록으로 인덱싱된 독립적인 버퍼 캐쉬**(buffer cache)**에 저장.
- 리눅스 디바이스의 세 가지 클래스:
  - ***block devices*** **–** 완전히 독립된 고정 크기의 데이터 블록에 대한 임의 접근 허용
  - ***character devices*** **–** 대부분의 다른 장치 포함; 정규 파일의 기능을 지원할 필요는 없음.
  - ***network devices*** **–** 커널의 네트워킹 시스템을 통해 인터페이스.

장치-구동기 블록 구조


user application
file system
block device file
buffer cache
request manager
block device driver
SCSI manager
SCSI device driver
character device file
TTY driver
line discipline
character device driver
network socket
protocol driver
network device driver

블록 장치

- 시스템의 모든 디스크 장치에 대한 주요 인터페이스 제공
- ***block buffer* cache** 의 두 가지 주요 목적:
  - 활성화된 **I/O** 를 위한 버퍼 풀로 동작
  - 수행된 **I/O** 를 위한 캐쉬로 동작
- **The *request manager***
  - 블록 디바이스 드라이버로 버퍼 내용을 읽기 및 쓰기 관리.

문자 장치

- 문자 디바이스 드라이버
  - 고정된 데이터 블록에 대한 임의 접근을 제공하지 않는 디바이스 드라이버.
  - 드라이버의 다양한 파일 **I/O** 동작을 구현하는 함수들을 등록.
  - 커널은 문자 장치에 대한 파일 읽기 또는 쓰기 요청에 대해 거의 우선처리하지 않고 장치로 전달.
  - 예외 - 터미널 디바이스를 구현하는 특정한 문자 디바이스 드라이버는 커널이 표준 인터페이스를 관리함.

프로세스 간 통신

▣ 시그널

- ◈ **UNIX** 처럼 **Linux** 도 시그널을 통해 이벤트가 발생하였다는 사실을 프로세스에게 알림.
- ◈ 제한된 수의 시그널이 존재하며, 이들은 정보를 처리하지 않고, 단지 시그널이 발생하였다는 사실만 프로세스에게 전달함.
- ◈ 리눅스 커널은 커널 모드에서 수행하는 프로세스간에는 시그널을 사용하지 않고, 스케줄링 상태와 **wait.queue** 구조를 통해 통신.

프로세스 간 데이터 전달

- **Pipe 매커니즘**
  - 자식 프로세스가 부모에 대한 통신 채널을 상속받아, 파이프의 한 쪽 끝으로 데이터를 쓰거나 읽을 수 있음.
- **System V IPC**
  - 세마포어
  - 메시지 큐
  - 공유 메모리
    - 아주 빠른 통신 방식 제공
    - 프로세스가 공유 메모리 영역에 쓴 데이터는 다른 프로세스에 의해 즉시 자신의 주소 공간으로 매핑 가능.
    - 동기화하기 위해, 공유 메모리는 또 다른 프로세스간 통신 메커니즘과 함께 사용되어야 함.

공유 메모리 객체

- 공유 메모리 객체
  - 파일이 메모리-매핑된 메모리 영역에 대한 저장소로 동작할 수 있는 것처럼 공유 메모리 객체는 공유 메모리 영역에 대한 저장소로 동작.
  - 공유 메모리 매핑은 페이지 폴트에 대해 지속적인 공유-메모리 객체로부터의 페이지들로 매핑하도록 함.
  - 공유 메모리 객체는 가상 메모리로 매핑하는 프로세스가 존재하지 않아도 그 내용을 기억.

네트웍 구조

- 네트워킹은 리눅스의 핵심 기능 부분
  - **UNIX** 간 통신을 위한 표준 인터넷 프로토콜 지원.
  - 비 **UNIX** 운영 체제 고유의 프로토콜 구현.
    - 예) **PC** 네트워크 프로토콜인 **Appletalk** 와**IPX.**
- 리눅스 커널 내에서 **3** 개의 소프트웨어 계층으로 네트워킹 구현**:**
  - **The socket interface**
  - **Protocol drivers**
  - **Network device drivers**
- 리눅스 네트워킹 시스템에서 가장 중요한 프로토콜 집합은 인터넷 프로토콜 슈트**.**
  - 네트워크 상의 어디든지 다른 호스트 간 라우팅 구현.
  - 라우팅 프로토콜의 최상위에 **UDP, TCP and ICMP** 프로토콜 구축 **.**

보안

- **PAM (*pluggable authentication modules*)** 시스템
  - 리눅스에서 사용 가능.
  - 사용자를 인증할 필요가 있는 시스템 컴포넌트가 사용할 수 이는 공유 라이브러리에 기반.
- **UNIX** 및 리눅스 시스템의 접근 제어
  - 고유의 수치 인식자들 **(**uid **and** gid**)** 을 사용하여 수행.
  - 접근 모드(읽기, 쓰기, 실행)가 소유자, 그룹, 모든 이에게 허용되는지에 대한 보호 마스크 **(*protections mask*)**를 객체에 설정하여 수행.
- 표준 **UNIX** setuid 메커니즘 보완**:**
  - **POSIX** 사양에 있는 **saved *user-id*** 메커니즘 구현 **–** 프로세스는 유효 **uid** 를 반복해서 잃거나 재획득 허용.
  - 유효 **uid** 의 일부 권리를 보장하는 프로세스 특성 추가.
- 리눅스는 어떤 서버 프로세스에게 특권을 보장받지 않고 단일 파일에 대한 선택적인 접근을 허용하는 다른 메커니즘 제공.

참고자료

- **Silberschatz, Galvin, Gagne 저, *Operating Systems Concepts*, 6th edition, Wiley., 2002.**
  - 조유근, 고건, 김영찬 공역, ***Operating Systems Concepts*, 6판, 홍릉과학출판사, 2004.**
- **Bovet, Cesati 저, *Understanding the Linux kernel, 3rd edition*, O'Reilly., 2005.**
  - 심마로, 이호 역, 리눅스 커널의 이해, 개정판, 한빛미디어, **2003.**