Operating System Structure

OS Services

- 프로그래머의 편의를 위해 제공되는 서비스
- 프로그래밍 작업이 더 쉽도록

유저에게 도움되는 기능

- UI
- 프로그램 실행
- I/O operations
- File-system manipulation
- communications
- 에러 탐지

시스템 작동 효율을 위한 기능

- 자원 할당
- 계정 관리
- Protection & Security

System Call

- OS가 제공하는 프로그래밍 인터페이스
- 보통 C/C++같은 High level language로 작성
    - ex) CPP의 read 함수
- 대부분 직접 시스템 콜을 호출하지 않고 API를 통해 접근
    - ex) Win32 API (window), POSIX API (unix), Java API (JVM)
- 왜 API를 사용하는가?
    - 프로그램 이식성
    - 시스템콜은 사용하기 어려움

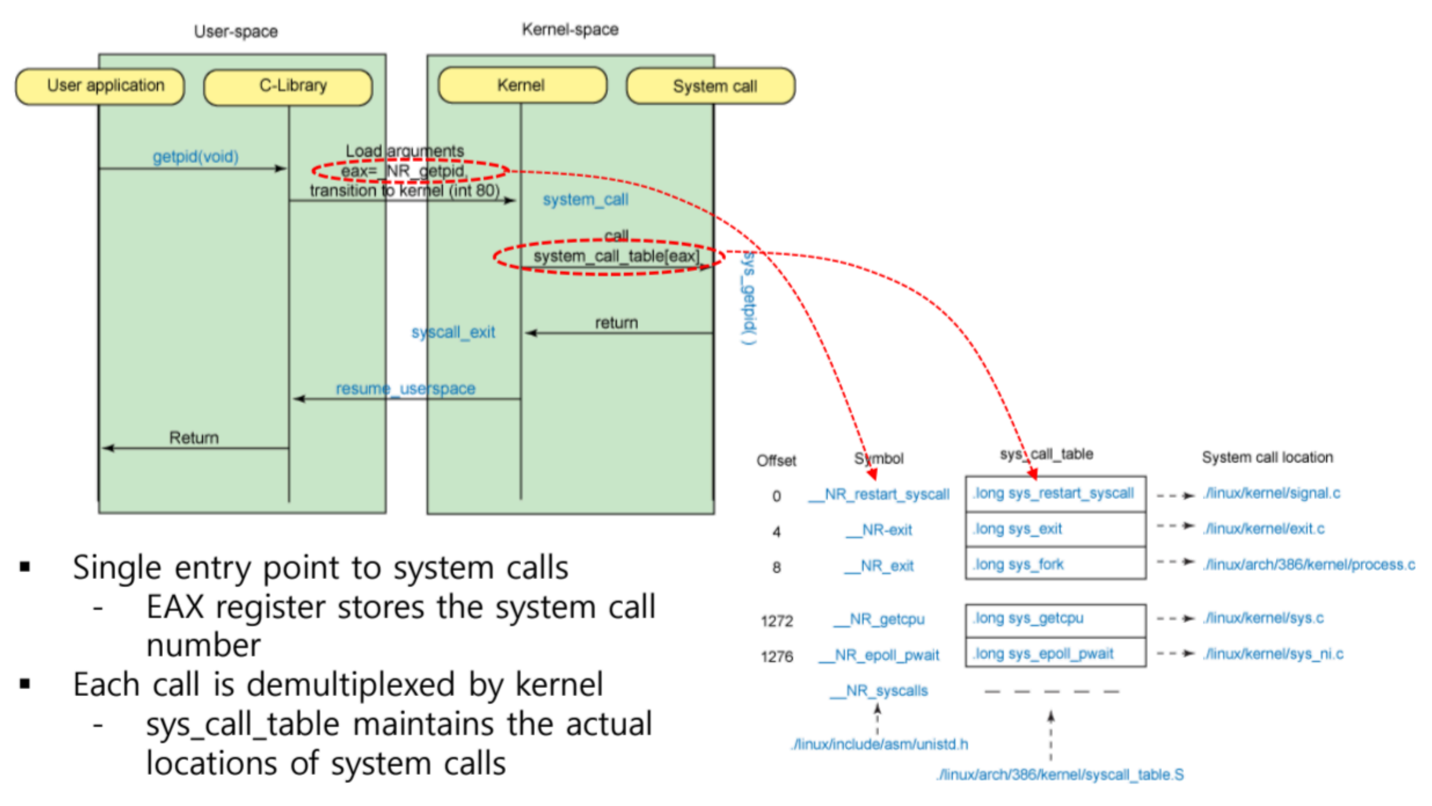
Implementation

- 시스템 콜 마다 번호가 있음
- 시스템 콜 인터페이스에 이 번호들로 인덱스된 테이블이 있음
- 테이블에는 각 호출 번호에 해당하는 시스템 호출 함수를 가르키는 포인터를 포함
- 시스템 콜 인터페이스가 OS 커널 안에서 시스템 콜을 호출하고 성공 / 실패를 나타내는 상태값과 반환값을 return
- 시스템 콜을 호출하는 함수는 시스템 콜의 원리를 몰라도 됨

Parameter Passing

- 시스템 콜에 사용되는 parameter들을 OS로 전달할 필요가 있음
- 세 가지 방법이 존재
    1. 가장 쉬운 방법 : parameter를 레지스터에 저장
        - parameter가 많으면 레지스터로는 부족할 수 있음
    2. parameter를 메모리에 블록 단위로 저장 후 블록의 시작 주소를 레지스터에 parameter로 전달
        - 실제로 리눅스와 Solaris에서 사용하는 방식
    3. parameter를 현재 프로세스의 스택에 넣고 OS가 pop하는 방식
- 블록과 스택 방법은 parameter의 수, 길이 제한이 없음
- 전달된 parameter는 OS가 시스템 콜을 실행시킬 때 사용

리눅스로 예시

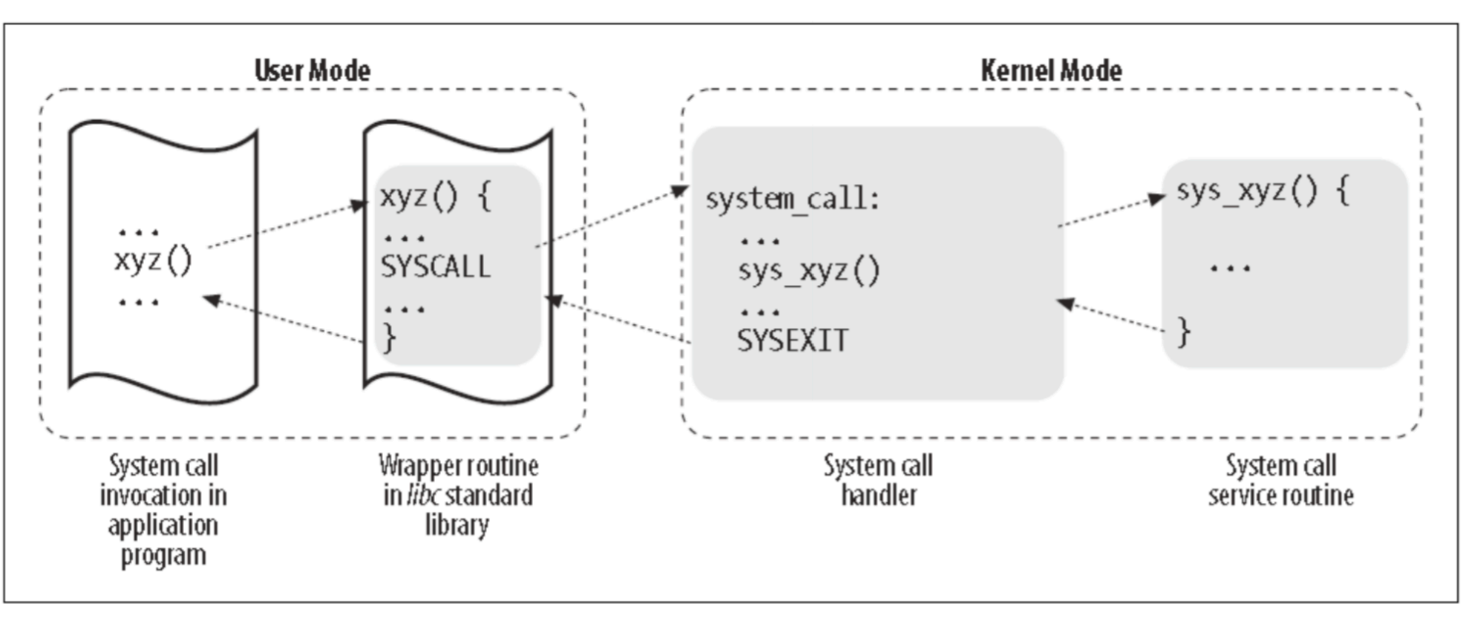


- Single entry point to system calls
    - EAX register stores the system call number
- Each call is demultiplexed by kernel
    - sys_call_table maintains the actual locations of system calls

- NR은 number의 줄임말

1. user가 C에서 getpid(); 명령어를 실행
2. C-Library에 작성된 코드대로 다음 과정을 거침
    1. eax 레지스터에 시스템 콜 번호를 저장
        - getpid의 시스템 콜 번호는 __ NR_getpid로 매크로 정의돼 있음
    2. 시스템 콜을 호출하는 인터럽트 발생
        - int 0x80 (128번) 은 instruction이며 여기서 int는 interrupt를 의미
        - 0x80은 인터럽트 벡터에서 시스템 콜 호출을 의미
        - 64비트에선 int 0x80보다 syscall이라는 instruction을 사용 (비슷한데 성능은 더 좋나봄)
    3. CPU가 커널 모드로 전환
    4. syscall 핸들러 실행
        - 번호에 맞는 시스템 콜을 찾음
    5. sys_getpid() 실행
        - 현재 프로세스 ID 반환
    6. 커널이 pid를 eax/rax에 저장한 후 유저모드로 복귀
    7. 유저는 반환값을 사용

시스템 콜 호출 라이브러리 관점

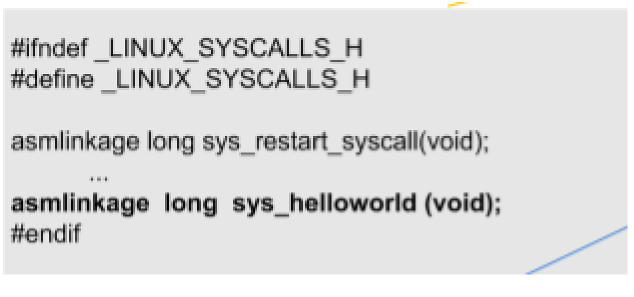


- app에서 시스템 콜이 들어간 함수 호출
- 이 함수는 library의 wrapper routine라 할 수 있음
    - 예로, malloc(), calloc(), free()는 모두 같은 시스템 콜을 사용함
    - 하지만 전부 다른 wrapper routine
- wrapper routine 속에서 시스템 콜 호출 인터럽트 발생 instruction 실행
    - syscall
    - system_call은 syscall을 수행(system call handler)하며 usermode에서 kernelmode로 스위칭 되면서 시작됨
- 커널 모드에서 system call handler가 번호에 맞는 시스템 콜을 찾음
- 시스템 콜을 찾으면 system call service routine 실행 후 결과값 반환

리눅스에서 시스템 콜 직접 만들기

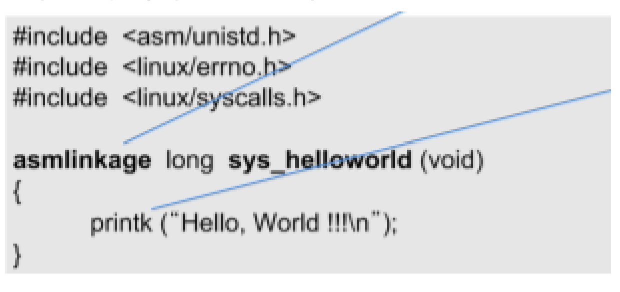
1. 시스템 콜 함수 정의

- 우선 /linux/include/linux/syscalls.h 헤더파일에 만들고자 하는 함수를 선언



```
#ifndef _LINUX_SYSCALLS_H
#define _LINUX_SYSCALLS_H

asmlinkage long sys_restart_syscall(void);
        ...
asmlinkage  long  sys_helloworld (void);
#endif
```

- asmlinkage는 함수의 모든 인자를 스택에 전달하도록 컴파일러에게 요청하는 매크로
- 함수를 정의하는 코드 작성

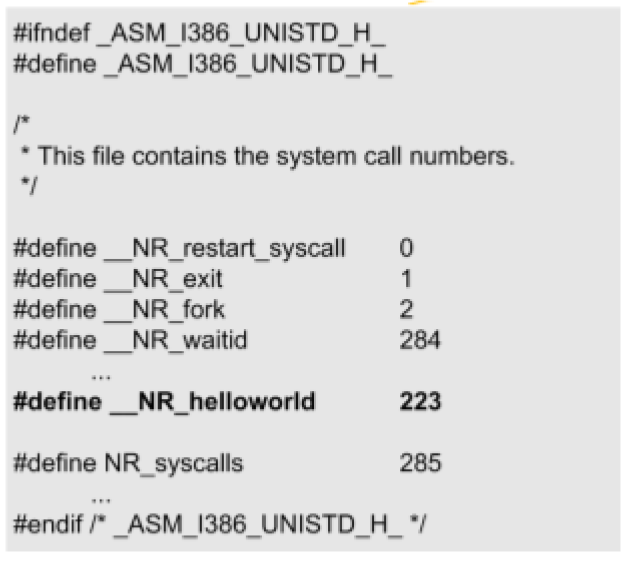


```
#include  <asm/unistd.h>
#include  <linux/errno.h>
#include  <linux/syscalls.h>

asmlinkage  long  sys_helloworld (void)
{
        printk (“Hello, World !!!\n”);
}
```

- 커널 레벨에서 작동하는 함수는 표준 C 라이브러리 사용 불가
- printf가 아닌 printk 사용

2. 시스템 콜 번호 할당

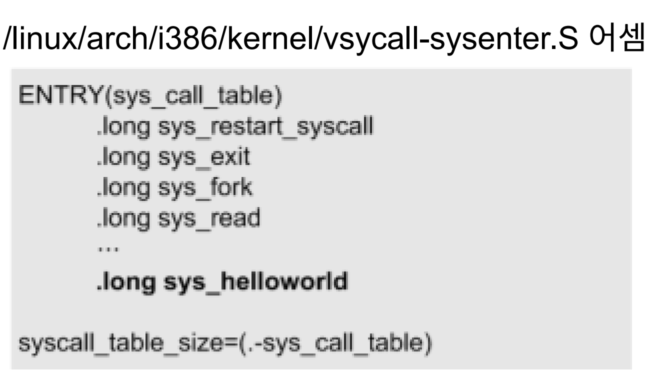
- 시스템 콜 번호는 매크로로 /linux/include/asm-i386/unistd.h 에 저장



```
#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_

/*
 * This file contains the system call numbers.
 */

#define __NR_restart_syscall      0
#define __NR_exit                 1
#define __NR_fork                 2
#define __NR_waitid               284
        ...
#define __NR_helloworld           223

#define NR_syscalls               285
        ...
#endif /* _ASM_I386_UNISTD_H_ */
```

3. 시스템 콜 함수 등록

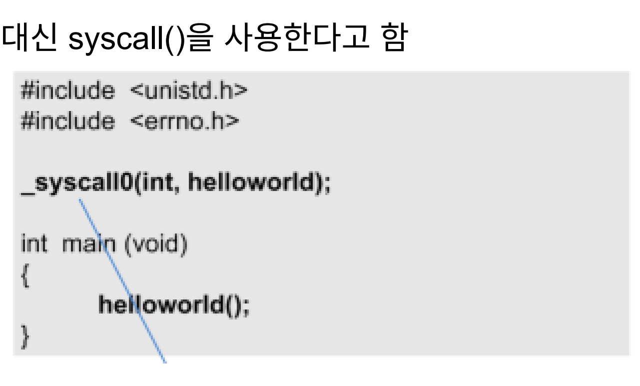
- /linux/arch/i386/kernel/vsycall-sysenter.S 어셈블리 파일에 시스템 콜 함수를 등록



```
ENTRY(sys_call_table)
        .long sys_restart_syscall
        .long sys_exit
        .long sys_fork
        .long sys_read
        ...
        .long sys_helloworld

syscall_table_size=(.-sys_call_table)
```

4. 사용 방법

- 표준 라이브러리에 등록되지 않은 시스템 콜을 호출할 때 필요한 어셈블리 코드가 없음
- _ syscallN() 매크로를 사용
- N은 인자의 개수
- 커널의 시스템 콜을 호출하는 래퍼 함수를 자동 생성하는 매크로
- 지금은 사라졌다고 함
- 대신 syscall()을 사용한다고 함

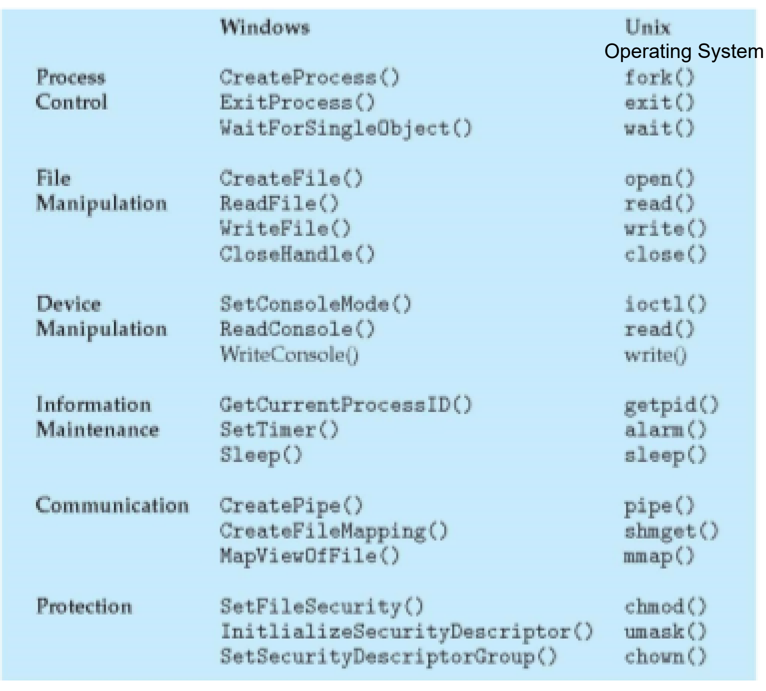


```
#include  <unistd.h>
#include  <errno.h>

_syscall0(int, helloworld);

int  main (void)
{
        helloworld();
}
```

시스템 콜 유형

- process control
    - 프로세스 생성 및 중단, 프로그램 적재 및 실행
    - 이벤트 대기, 에러 발생 시 메모리 덤핑 (로그 저장)
    - 메모리 할당 및 해제 (자원 관리도 프로세스의 역할)
    - single step execution을 사용하는 디버거
    - 프로세스간 공유 자원 접근 관리를 위한 락
- file manipulation (management)
    - 파일 생성, 제거, 열기, 닫기, 읽기, 쓰기
    - 파일 속성 불러오기, 수정
- device manipulation
    - 디바이스 request, release, read, write
    - 디바이스 속성 get
    - 디바이스 attach, detach in logically
- information maniputlation
    - 시간, 날짜, 시스템 데이터, 프로세스, 파일, 디바이스 속성 get / set
- communication
    - 프로세스 간 통신 연결, 삭제
    - message passing model로 메시지 송수신
        - process 간 통신 방법 (IPC) 중 하나
    - remote 디바이스 attach, detach
    - 상태 정보 전송
    - Shared-memory model이 메모리에 접근권을 생성, 및 획득
        - 이것도 IPC 중 하나
- protection
    - 자원 접근 제어, permisson get/set, 유저 접근 허락/거부

윈도우, 유닉스에 사용되는 시스템 콜 예시



	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

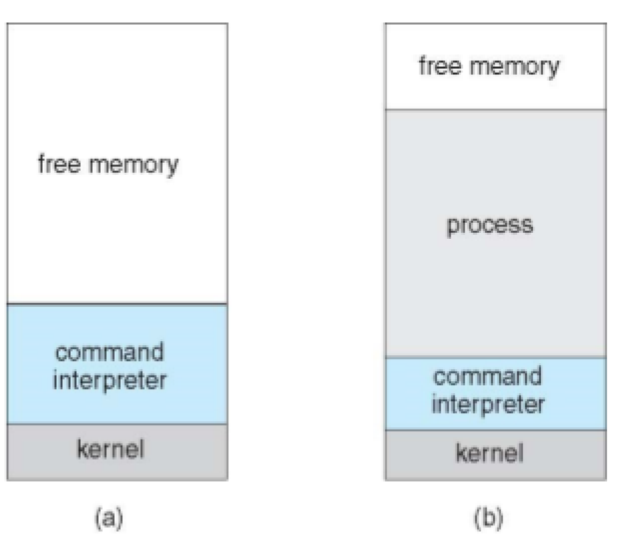
Operating System Structure

Standard C Library 예시

- printf()를 실행해보자
- write() 시스템 콜이 실행됨
- write() 시스템 콜은 파일 말고도 output stream에도 write 가능

멀티 태스킹에 사용되는 시스템 콜

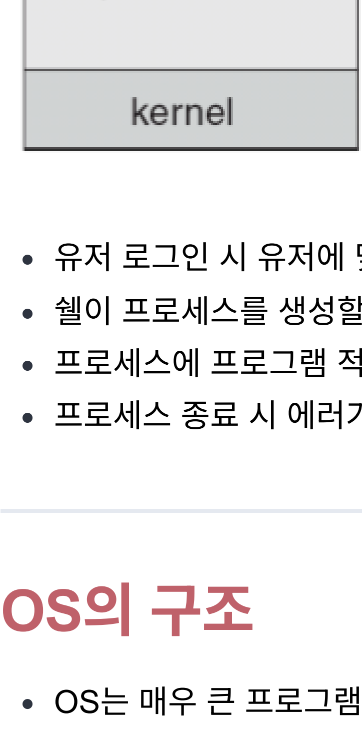
싱글 태스킹 예시 : MS-DOS



(a) At system startup (b) running a program

- 시스템 부팅 시 쉘 실행
- 쉘에서 프로그램을 간단하게 실행 가능
  - 프로세스는 만들어지지 않음
  - 하나의 메모리 공간만 차지
  - 프로그램을 메모리에 적재 시 커널을 제외하여 overwriting 수행
- 프로그램 종료 시 shell reloaded

멀티 태스킹 예시 : FreeBSD



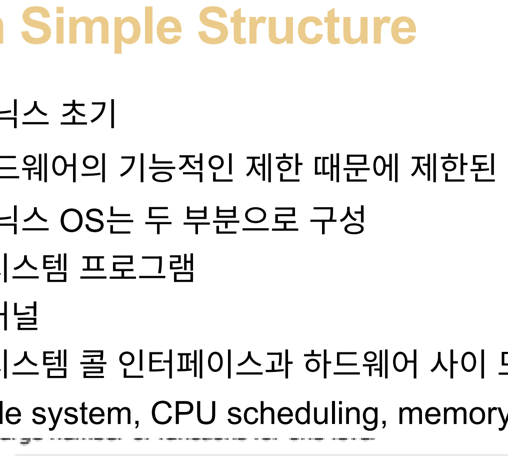
- 유저 로그인 시 유저에 맞는 쉘 실행
- 쉘이 프로세스를 생성할 때 fork() 시스템 콜 실행
- 프로세스에 프로그램 적재 시 exec() 시스템 콜 실행
- 프로세스 종료 시 에러가 있으면 0 보다 큰 정수 반환

OS의 구조

- OS는 매우 큰 프로그램
- OS의 진화 과정
  1. simple structure : MS-DOS
  2. more complex : UNIX
  3. Layered : an abstraction
  4. microkernel : mach

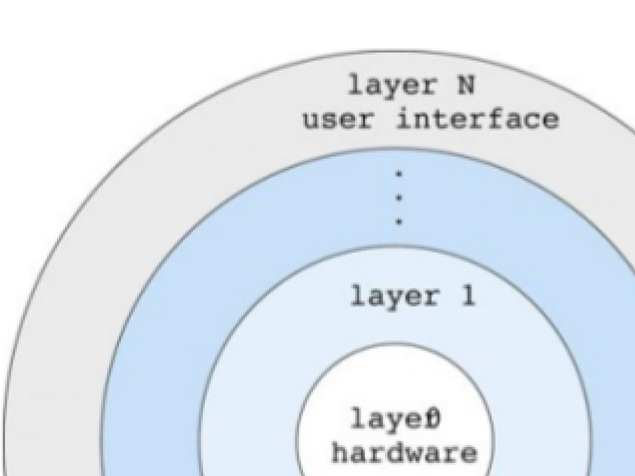
Monolithic structure

- 초창기 OS
- 모듈로 구분되어있지 않음
- 인터페이스와 기능의 레벨링이 제대로 정의되어있지 않음

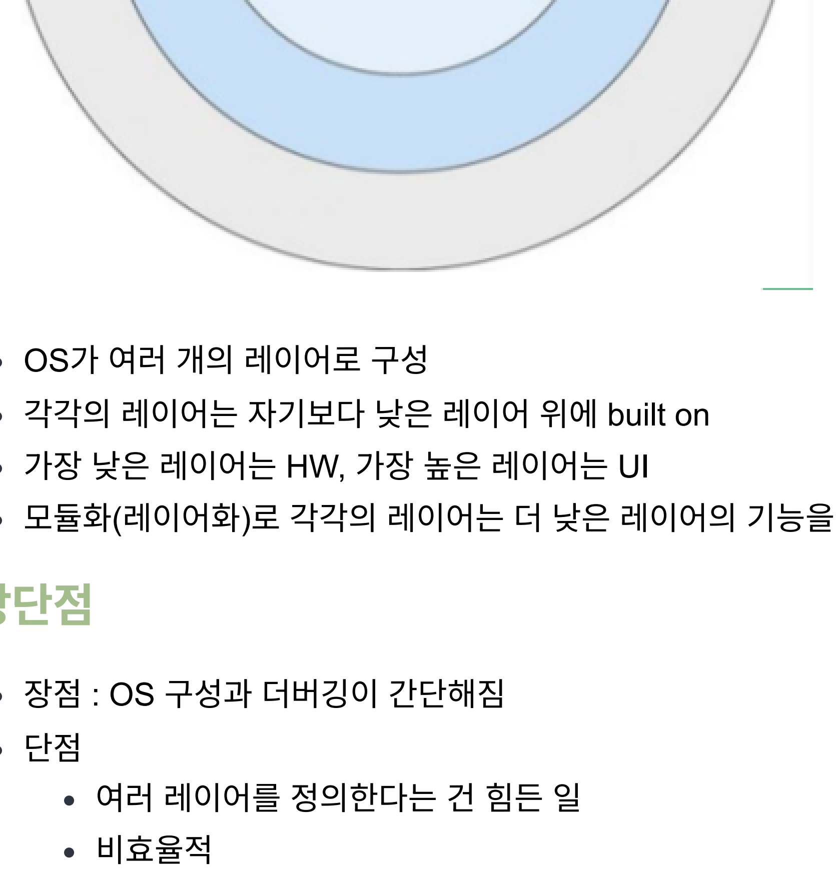


Non Simple Structure

- 유닉스 초기
- 하드웨어의 기능적인 제한 때문에 제한된 구조를 가졌었음
- 유닉스 OS는 두 부분으로 구성
  - 시스템 프로그램
  - 커널
  - 시스템 콜 인터페이스과 하드웨어 사이 모든 것들을 구현
  - file system, CPU scheduling, memory management, etc. 많은 함수들이 하나의 레벨에 존재



Layered Approach



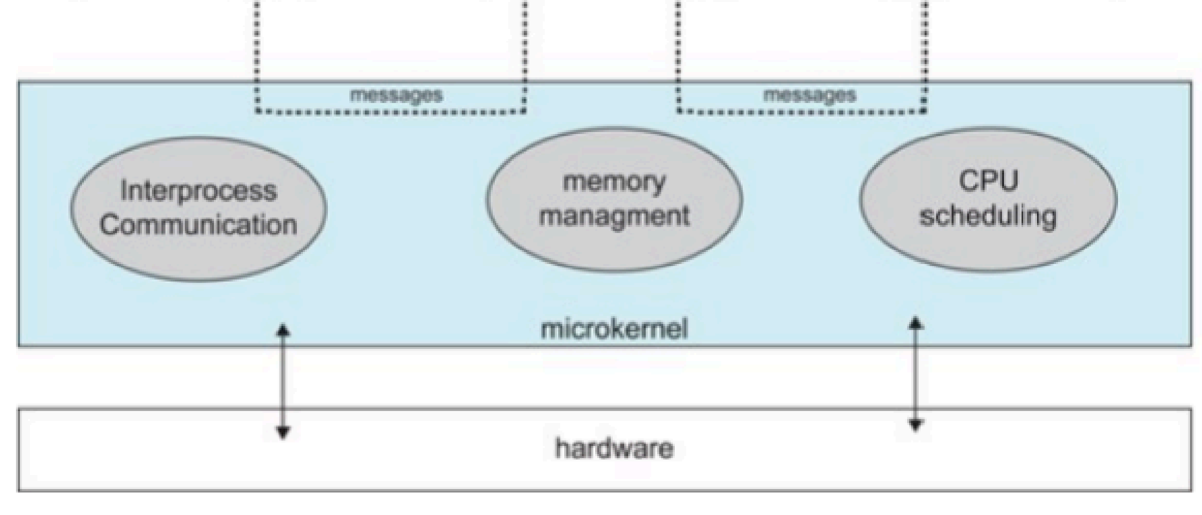
- OS가 여러 개의 레이어로 구성
- 각각의 레이어는 자기보다 낮은 레이어 위에 built on
- 가장 낮은 레이어는 HW, 가장 높은 레이어는 UI
- 모듈화(레이어화)로 각각의 레이어는 더 낮은 레이어의 기능을 사용

장단점

- 장점 : OS 구성과 더버깅이 간단해짐
- 단점
  - 여러 레이어를 정의한다는 건 힘든 일
  - 비효율적
    - 시스템 콜이 모든 레이어를 통과해야하며 각각의 레이어마다 오버헤드 발생

microkernels

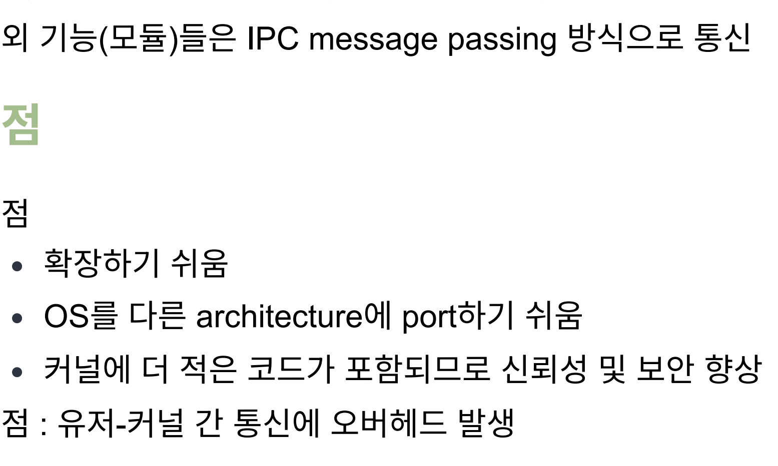
- monolithic kernel : 모든 OS 기능이 커널 내에서만 실행
- micro kernel : 최소한의 핵심 기능만 커널에, 나머지는 user space에서 실행



- A microkernel (μ-kernel) is the near-minimum amount of software that can provide the mechanisms needed to implement an operating system (OS).
- These mechanisms include low-level address space management, thread management, and inter-process communication (IPC).

Monolithic kernel based OS

Micro kernel based OS



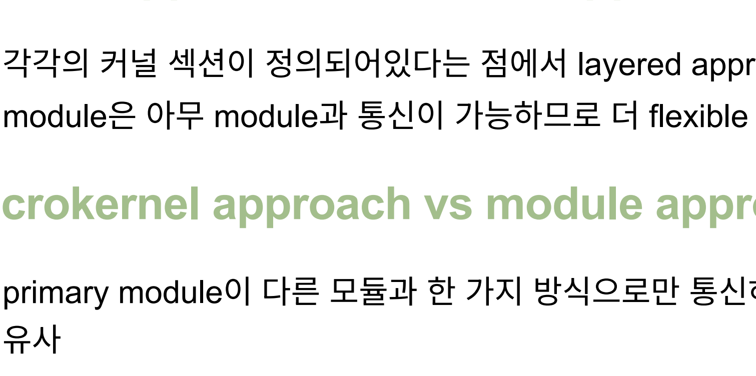
- IPC, 메모리-프로세스 관리, CPU 스케쥴링, 하드웨어 제어같은 핵심 기능만 커널에 존재
- 이외 기능(모듈)들은 IPC message passing 방식으로 통신

장단점

- 장점
  - 확장하기 쉬움
  - OS를 다른 architecture에 port하기 쉬움
  - 커널에 더 적은 코드가 포함되므로 신뢰성 및 보안 향상
- 단점 : 유저-커널 간 통신에 오버헤드 발생

모듈

- 현대 OS는 loadable kernel modules를 구현해놨음
- 각각의 core component는 분리되어있으며 object-oriented approach를 사용



- 커널은 부팅, 런타임 동안 core components들과 추가 services의 링크들을 갖게 됨
- 각각의 component들은 known interface를 통해 통신 가능

layered approach vs module approach

- 각각의 커널 섹션이 정의되어있다는 점에서 layered approach와 유사
- module은 아무 module과 통신이 가능하므로 더 flexible

microkernel approach vs module approach

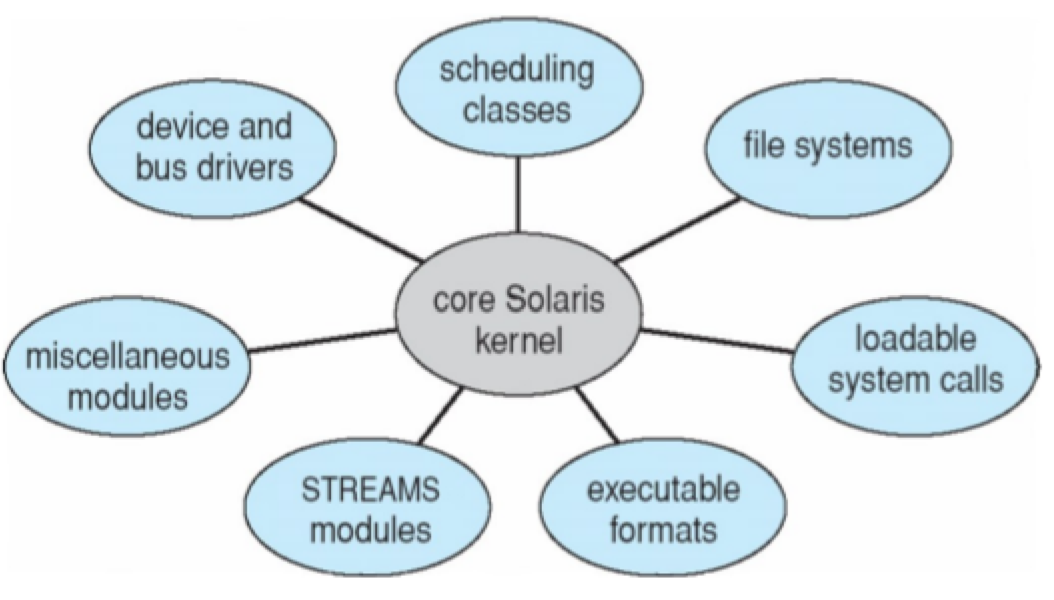
- primary module이 다른 모듈과 한 가지 방식으로만 통신하고 오직 하나의 core functions만 있다는 점에서 유사
- 모듈과 통신할 때 message passing을 하지 않아도 된다는 점에서 상위호환

Solaris가 사용한 Modular Approach

- solaris는 Sun Microsystems에서 개발한 Unix 기반 운영체제

- 다음과 같이 6개의 영역으로 나뉨

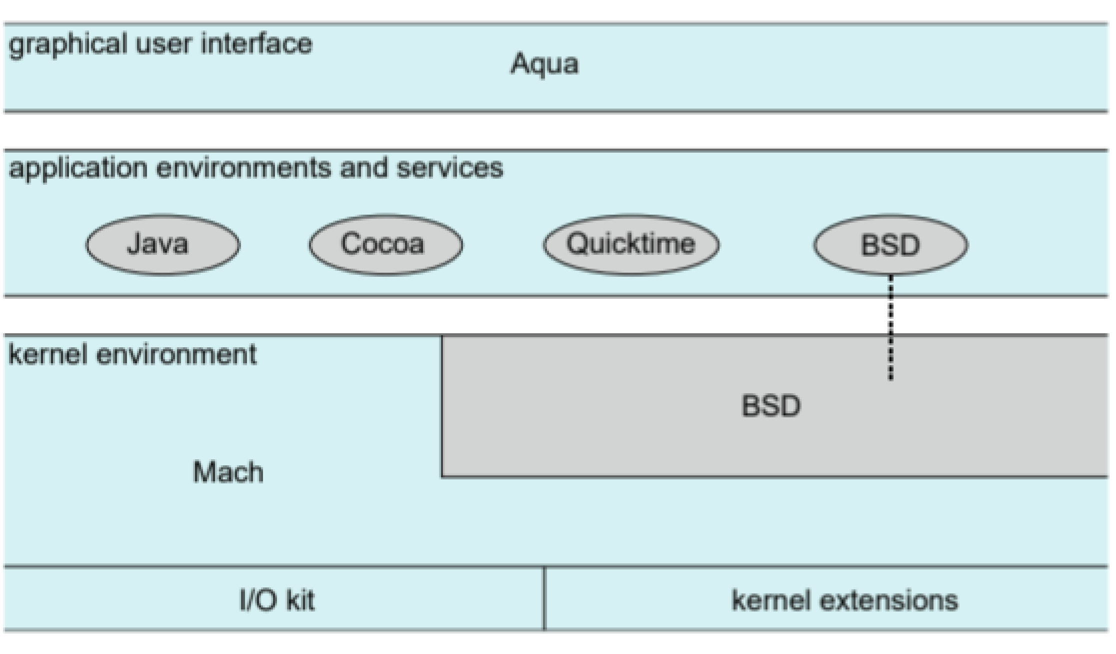
Operating System Structure



Hybrid Systems

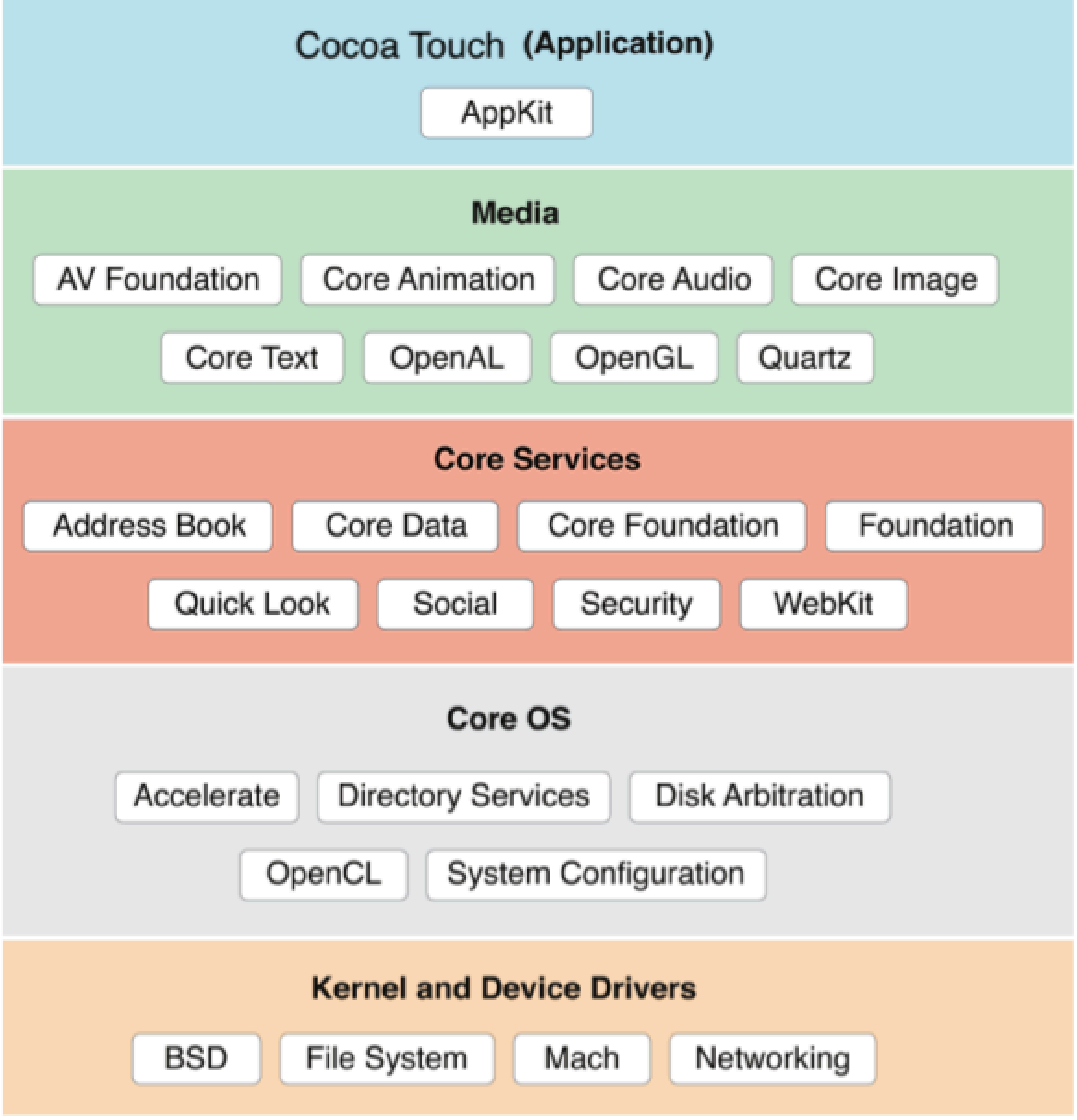
- 현대 OS는 지금까지 배운 모델 중 하나의 모델로만 구성되지 않음
- 성능, 보안, usability needs를 충족시키기 위해 여러 방법을 결합
- 리눅스와 솔라리스 커널은 커널 주소 공간에 존재 -> monolithic
  - +)기능의 동적 로딩을 위해 modular
- 윈도우는 대부분 monolithic, 일부 서브시스템을 위해 micro kernel 사용
- Apple Mac OS X는 hybrid, layered 설계를 따름
  - Aqua UI와 Cocoa 프로그래밍 환경

Max OS X 구조



IOS 구조

- 아이폰, 아이패드를 위한 applie mobile OS
- Mac OS X에 기능 추가
- 개발자가 app을 개발하기 쉽도록 Cocoa Touch 어플 제공
- 그래픽, 오디오, 비디오를 위한 media servies layer
- 클라우드 컴퓨팅, DB 같은 상위 레이어에 사용되는 서비스를 제공하는 Core services
- computational instructions에 동작하는 core operating system



Android

- Open Handset Alliance가 개발
  - 오픈소스
  - OHA의 대부분이 구글
  - 스마트폰 플랫폼을 open source로 만들고 제조사와 협력하여 발전시키는 것이 목표
- 리눅스 커널이 기반이지만 수정된 부분이 있음
  - process, memory, device-driver management 제공
  - power management를 추가
- Runtime enviorment에 기본 라이브러리 세트와 Dalvit VM이 포함
  - Runtime enviorment : 소프트웨어가 실행되는 동안 필요한 자원과 환경을 제공하는 시스템. 특히 프로그램이나 어플이 실행되는 동안 필요한 라이브러리, API, VM등을 일컬음.
  - Dalvit VM :
    - 안드로이드에서 자바 기반 앱을 실행하는 가상 머신
    - 기존 VM과 다르게 모바일 환경에 최적화됨
    - 더 적은 메모리와 CPU 자원을 사용
  - 안드로이드 API와 자바를 통해 앱 개발
- 라이브러리는 웹 브라우저, DB를 위한 frame work, multimedia, 잡다한 libc (C 라이브러리)로 구성됨

OS 디버깅

- OS는 에러 정보를 포함한 로그 파일을 생성
- 앱 failure 시 process의 메모리를 capture한 core dump file을 생성
- OS failure 시 kernel 메모리를 포함한 crash dump file을 생성

Performance Tuning

- bottlenecks을 제거함으로써 성능 향상
- OS는 시스템 작동 수치를 display 하거나 compute할 수단을 제공해야 함
- ex) 리눅스에 사용되는 TOP 프로그램, 윈도우의 task manager

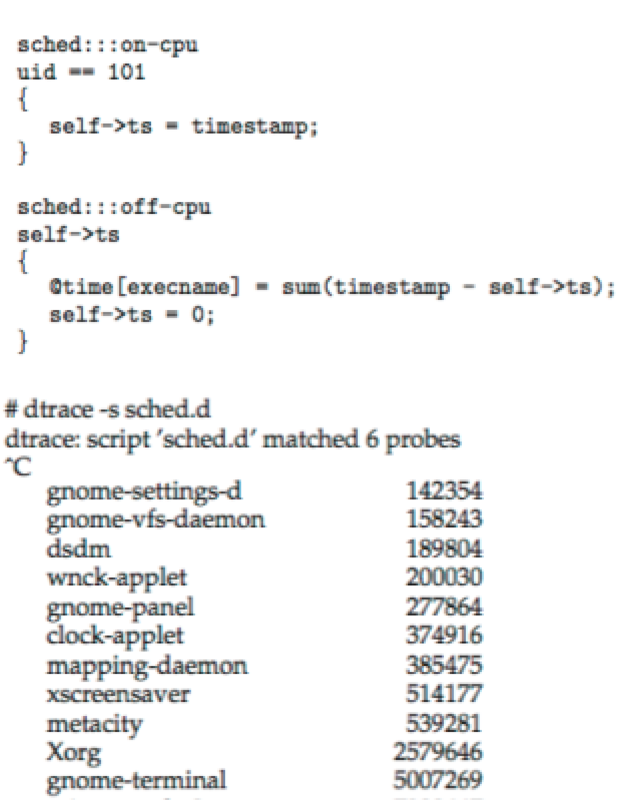
DTrace

- 시스템, 앱이 어떻게 동작하고 있는지 추적, 모니터링, 관찰하는 기술
- 특정 이벤트 발생 시 Probe가 fire되어 시스템 상태 캡처, 필요 정보 수집
- 특정 이벤트를 발생시킨 app/시스템에 정보 전달

```
# ./all.d 'pgrep xclock' XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
  0 -> XEventsQueued                          U
  0   -> _XEventsQueued                       U
  0     -> _X11TransBytesReadable             U
  0     <- _X11TransBytesReadable             U
  0     -> _X11TransSocketBytesReadable U
  0     <- _X11TransSocketBytesreadable U
  0     -> ioctl                              U
  0       -> ioctl                            K
  0         -> getf                           K
  0           -> set_active_fd                K
  0           <- set_active_fd                K
  0         <- getf                           K
  0         -> get_udatamodel                 K
  0         <- get_udatamodel                 K
...
  0         -> releasef                       K
  0           -> clear_active_fd              K
  0           <- clear_active_fd              K
  0           -> cv_broadcast                 K
  0           <- cv_broadcast                 K
  0         <- releasef                       K
  0       <- ioctl                            K
  0     <- ioctl                              U
  0   <- _XEventsQueued                       U
  0 <- XEventsQueued                          U
```

- XEventsQueued 시스템 콜의 예시
- libc 라이브러이에서 커널로, 커널에서 다시 돌아오는 모습
- U는 사용자 모드에서 실행, K는 커널 모드에서 실행됐음을 의미

Dtrace로 각 프로세스 당 사용한 CPU 시간 기록하기



```
sched:::on-cpu
uid == 101
{
   self->ts = timestamp;
}

sched:::off-cpu
self->ts
{
   @time[execname] = sum(timestamp - self->ts);
   self->ts = 0;
}
```

```
# dtrace -s sched.d
dtrace: script 'sched.d' matched 6 probes
^C
   gnome-settings-d            142354
   gnome-vfs-daemon            158243
   dsdm                        189804
   wnck-applet                 200030
   gnome-panel                 277864
   clock-applet                374916
   mapping-daemon              385475
   xscreensaver                514177
   metacity                    539281
   Xorg                       2579646
   gnome-terminal             5007269
   mixer applet2              7388447
   java                      10769137
```

Figure 2.21 Output of the D code.

OS Generation

- OS는 하드웨어 사양에 따라 조정이 필요
- OS는 어떤 기계든 돌아가야함
- SYSGEN 프로그램이 하드웨어 시스템의 특정 구성 정보 수집
  - 하드웨어 세부 정보를 확인한 후 그에 맞는 운영체제 버전이나 커널을 생성하기 위해 사용
  - system-tuned / system-specific compiled kernel를 빌드하는 데 사용
  - 특정 시스템에 맞게 최적화된 코드 생성

System Boot

- 파워가 들어오면 미리 정해둔 메모리 위치에서 시작
  - CPU의 PC가 boot block을 가르키게 됨
- ROM에 펌웨어가 저장되어 초기 부트 코드를 갖고 있음
  - firmware : 하드웨어 제어, 초기화. HW, SW 간의 중간 역할
- OS는 하드웨어를 시작할 수 있도록 설계되어야 함
  - ROM, EEPROM에 저장된 bootstrap loader가 커널을 찾고 메모리에 적재한 다음 실행시킴
  - 가끔 두 단계로 나뉨
    1. 특정 위치에 저장된 boot block을 ROM에 저장된 코드로 불러옴
    2. boot block이 bootstrap loader(GRUB)를 불러옴
- GRUB가 여러 디스크, 버전, 커널 옵션에 따라 커널을 선택할 수 있게 함
  - GRUB : grand unified bootstrap loader
- 커널이 로드되면 시스템이 작동되기 시작

정리

- 파워가 들어오면 CPU의 PC가 boot block을 가르키게 되며 이 코드를 실행
- boot block은 bootstrap loader (GRUB) 를 불러옴
- bootstrap loader는 커널을 선택할 수 있는 기회를 주고 이 커널 실행
- 커널이 실행되면서 OS 전부가 작동하기 시작