

# 시스템 프로그래밍

(임베디드 시스템 프로그래머를 위한)

Jin Seek Choi

jinseek@hanyang.ac.kr

Mobile Intelligence Routing Lab (Mir)

<http://mir.hanyang.ac.kr/~jinseek/>

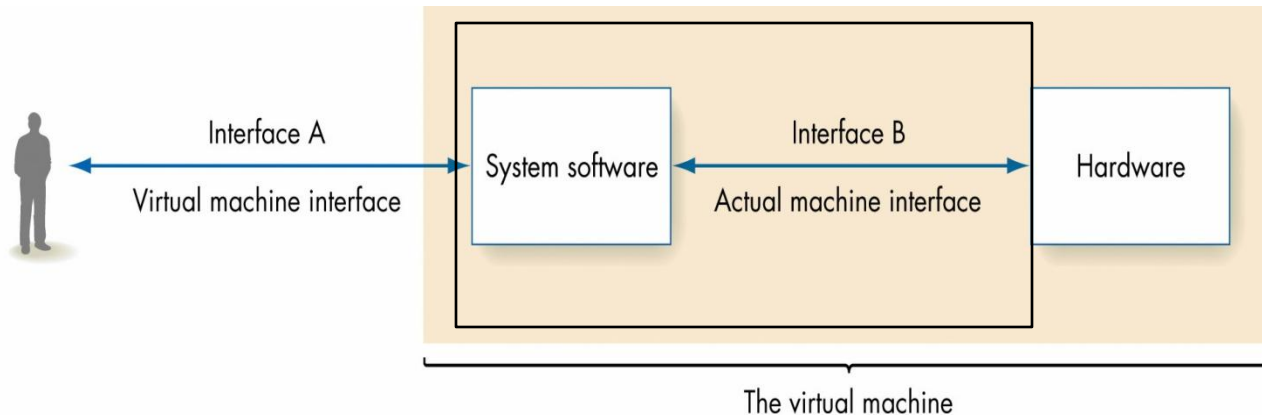
# 차례

---

- 시스템 프로그래밍 개념 및 리눅스 커널
- 파일 시스템 관리
  - 리눅스 파일 시스템 (디렉토리, 경로, 소유권,..)
  - 파일 기술자: 읽기, 쓰기, ..
  - 파일 관리 (creat, chmod, link, ..)
- Programmatic interface
  - System call function
- 프로세스 관리
  - Creation, scheduling, control, termination
  - Fork, exec, wait, getpid, ...
- 디바이스 드라이버
  - 문자 디바이스 드라이버
  - LED 디바이스 드라이버
- 프로세스 간 통신
  - 시그널과 시그널 처리
  - 파이프 라인
  - Inter-process communication, machine communication

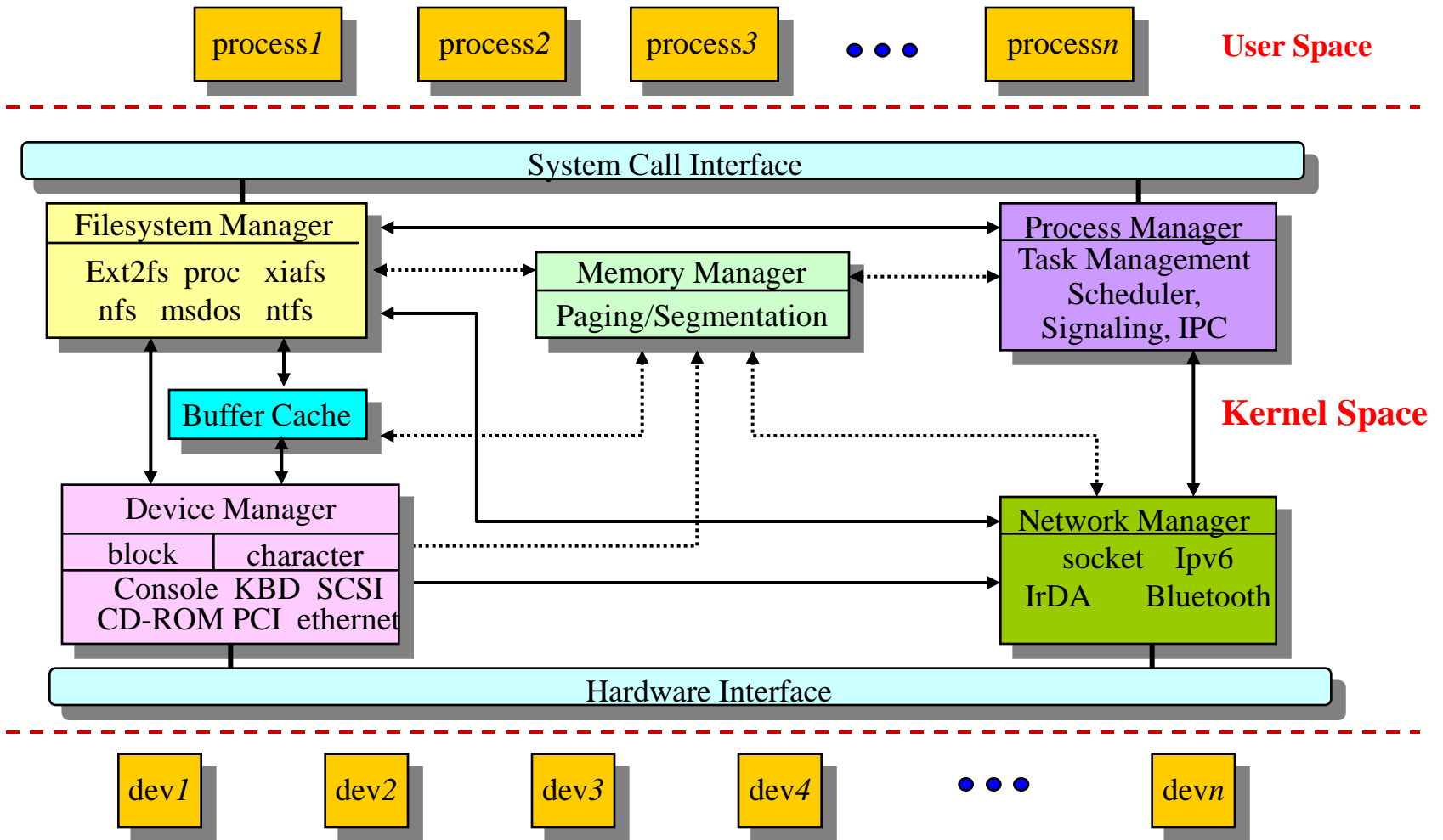
# 시스템 프로그래밍 개념

- 시스템 프로그래밍이란 컴퓨터 시스템을 다루는 프로그래밍
  - 시스템영역에서 이루어지는 데이터의 저장, 다중작업, 데이터통신, 최적화하는 일
  - 리눅스 커널 및 **핵심 라이브러리 및 드라이버** (입출력(I/O), 프로세스, 메모리, 디바이스, 신호 (Signals), 시간(Time), 네트워크, 파일시스템) 함수들 및 이들을 호출하는 프로그래밍)



운영체제는 저수준의 프로그래밍 인터페이스를 제공하는 물리적인 기계위에서 고수준의 프로그래밍 인터페이스 및 사용자 인터페이스를 제공하는 가상 기계(virtual machine) 다양한 하드웨어 위에서 추상화된 인터페이스를 제공해줌으로써, 개발자가 복잡한 주변장치들을 제어하는 어려움을 없애줌

# 리눅스 운영체제 구조



(Source : Linux Kernel Internals)

# 리눅스 시스템 프로그래밍? (1)

---

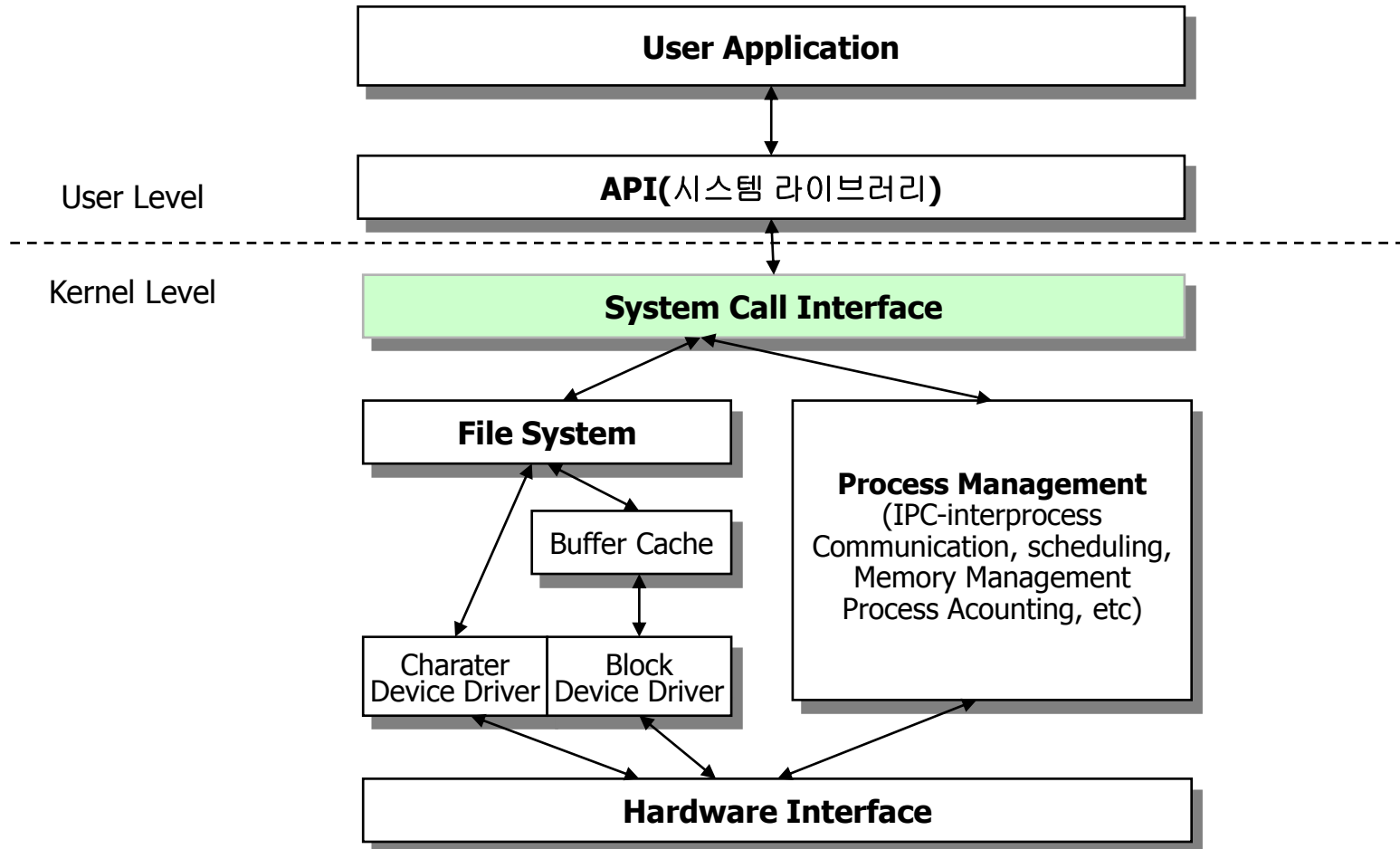
- “리눅스 시스템 프로그래밍”이란?
  - 리눅스 운영체제에서 제공하는 **시스템 호출(System Call)**을 사용해 프로그램을 작성하는 것
- 시스템 호출(System Call)
  - 리눅스 운영체제 서비스
    - 입출력장치 및 파일 시스템 접근
    - 프로세스 생성 및 관리
    - 사용자 및 시스템 정보 제공
    - 네트워크 연결 및 데이터 전송 등
  - 리눅스 시스템이 제공하는 서비스를 이용해 프로그램을 작성할 수 있도록 제공되는 프로그래밍 인터페이스
    - Windows's Win32 API
  - 기본적인 형태는 C 언어의 함수 형태로 제공

반환값 = 시스템호출명(인자, ...);

예) `int fd = open("/tmp/test.txt", O_RW);`

# 리눅스 시스템 프로그래밍? (2)

- 시스템 호출(System Call)



# 리눅스 시스템 프로그래밍? (3)

---

- 라이브러리 함수(Library Functions)
  - 라이브러리 : 미리 컴파일된 함수들을 묶어서 제공하는 특수한 형태의 파일
    - 예: C 표준 라이브러리
  - 자주 사용하는 기능을 독립적으로 분리하여 구현
    - ➔ 프로그램의 개발과 디버깅을 쉽게 하고 컴파일을 빠르게 수행
    - ➔ 프로그램 생산성 향상
  - /lib, /usr/lib에 위치, lib\*.a 또는 lib\*.so 형태로 제공
    - 정적 라이브러리(Static Library) : lib\*.a
      - 프로그램을 컴파일할 때에 실행파일 내부에 적재되어 실행파일 일부를 구성
      - 동일 라이브러리 코드가 중복 적재 가능
    - 공유 라이브러리(Shared Library) : lib\*.so
      - 프로그램 실행 시에 필요한 라이브러리를 적재
      - 여러 프로그램이 라이브러리 코드를 공유 ➔ 메모리의 효율적인 사용

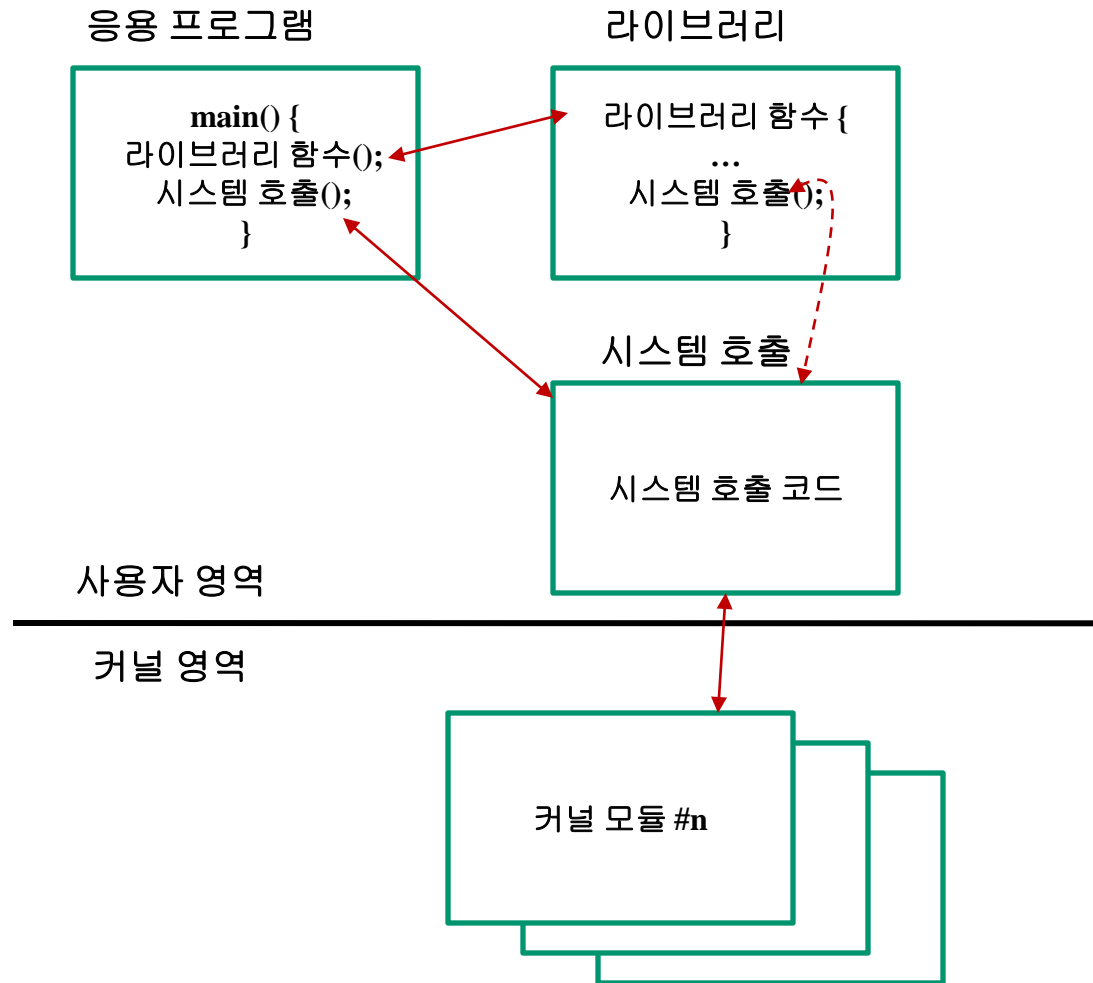
# 시스템 호출과 라이브러리 함수의 비교 (1)

- 시스템 호출

- 커널의 해당 서비스 모듈을 직접 호출하여 작업하고 결과를 리턴

- 라이브러리 함수

- 일반적으로 커널 모듈을 직접 호출 안함
- 일반적으로 시스템 호출에 대한 wrapper function





# 시스템 호출과 라이브러리 함수의 비교 (2)

- 시스템 호출 : man 페이지가 섹션 2에 속함

```
System Callsopen(2)  
  
NAME  
    open, openat - open a file  
SYNOPSIS  
    #include <sys/types.h>
```

- 라이브러리 함수 : man 페이지가 섹션 3에 속함

```
Standard C Library Functionsfopen(3C)  
  
NAME  
    fopen - open a stream  
SYNOPSIS  
    #include <stdio.h>
```

# 시스템 호출과 라이브러리 함수의 비교 (3)

- 시스템 호출의 오류 처리방법
  - 성공하면 0을 반환, 실패하면 -1을 반환
  - 전역변수 `errno`에 오류 코드 저장 :`man` 페이지에서 코드값 확인 가능

## [예제 1-1] 시스템 호출 오류 처리하기

ex1\_1.c

```
01 #include <unistd.h>
02 #include <stdio.h>
03
04 extern int errno;
05
06 int main(void) {
07     if (access("unix.txt", F_OK) == -1) {
08         printf("errno=%d\n", errno);
09     }
10
11     return 0;
12 }
```

```
# ex1_1.out
errno=2
```

```
# vi /usr/include/sys/errno.h
.....
/*
 * Error codes
 */
#define EPERM    1      /* Not super-user */
#define ENOENT   2      /* No such file or directory */
.....
```

# 시스템 호출과 라이브러리 함수의 비교 (4)

- 라이브러리 함수의 오류 처리방법
  - 오류가 발생하면 NULL을 반환, 함수의 리턴값이 int 형이면 -1 반환
  - errno 변수에 오류 코드 저장

[예제 1-2] 라이브러리 함수 오류 처리하기

ex1\_2.c

```
01 #include <stdlib.h>
02 #include <stdio.h>
03
04 extern int errno;
05
06 int main(void) {
07     FILE *fp;
08
09     if ((fp = fopen("unix.txt", "r")) == NULL) {
10         printf("errno=%d\n", errno);
11         exit(1);
12     }
13     fclose(fp);
14
15     return 0;
16 }
```

# ex1\_2.out  
errno=2

man fopen에서 확인

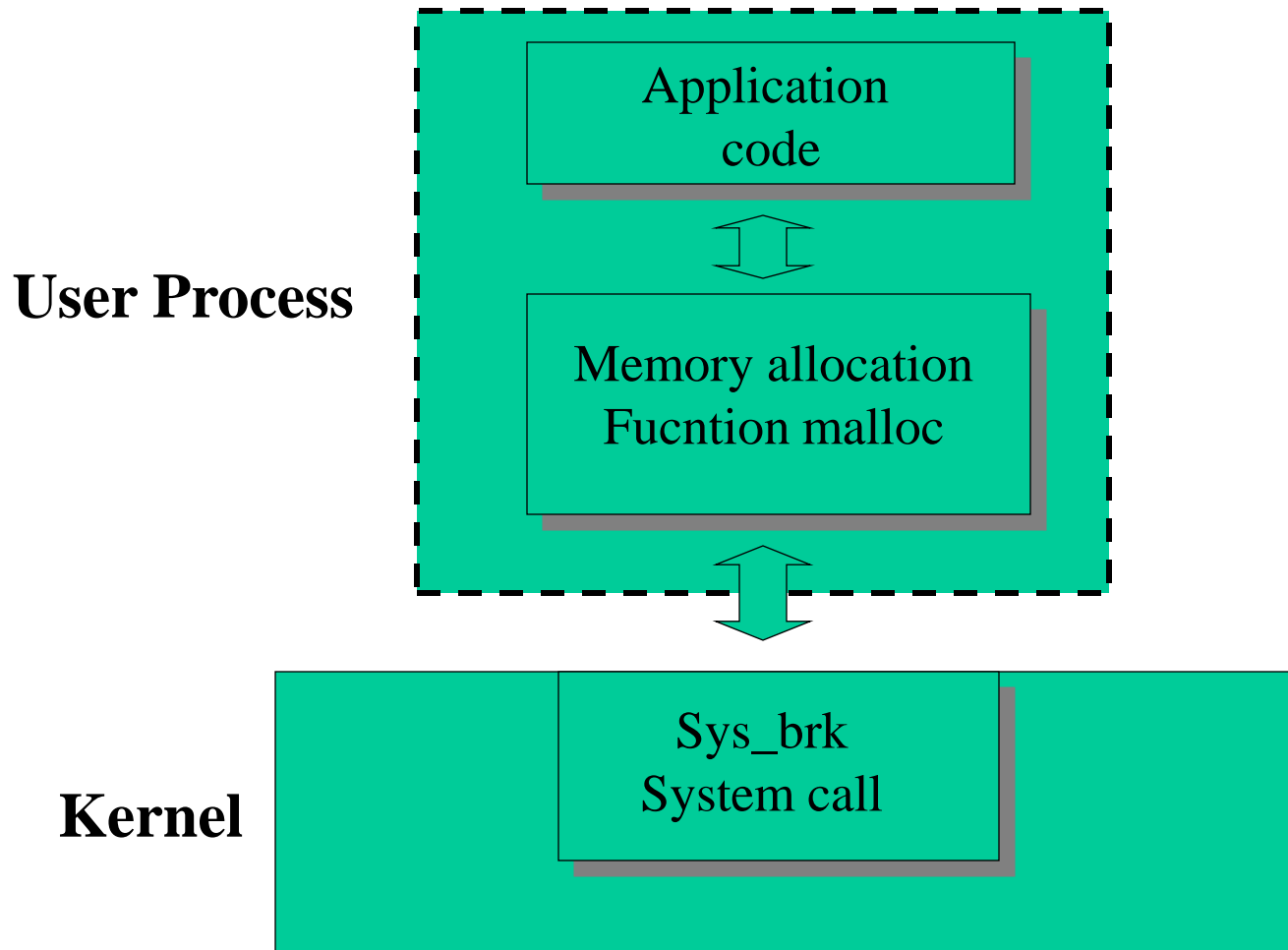
# 시스템 호출(System Call) 인터페이스

---

- 커널이 사용자 프로그램에게 제공하는 프로그래밍 인터페이스
  - fork(2) – create new process
  - open(2) – open a file
  - semop(2) - semaphore operations
- 표준 C 언어 라이브러리는 커널이 제공해주는 시스템 호출을 사용할 수 있게 하는 래퍼루틴(wrapper routine)을 포함
  - malloc(3) – memory allocation

# malloc 함수와 sbrk() 시스템 호출

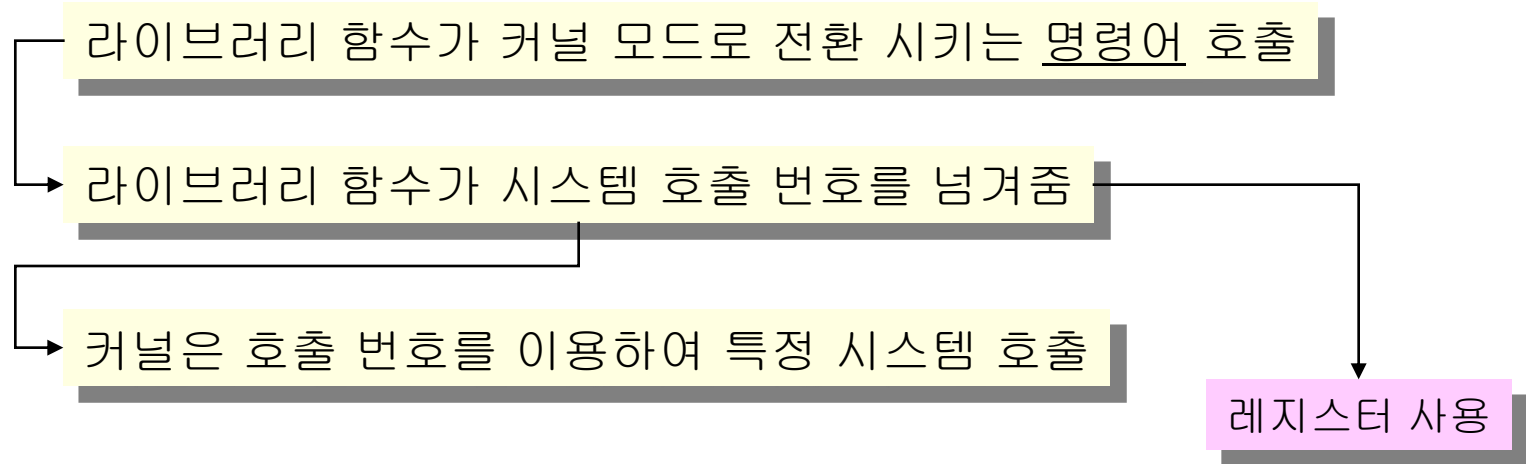
---



# System Call Step(1)

---

- Step1: User Program



# System Call Step(2)

---

- Step 2 : kernel

시스템 호출 번호에 해당하는 시스템 호출 테이블의 항목을 찾는다

시스템 호출에 대한 인수들의 개수를 결정한다.

인수들을 사용자 주소 공간에서 복사한다.

중도 실패 복귀 (abortive return)을 위해 **현행 문맥을 저장**

커널내의 시스템 호출 코드를 호출

# 리눅스 커널 시스템 호출

번호	함수명	소스코드 위치
1	sys_exit	<a href="#">kernel/exit.c</a>
2	sys_fork	<a href="#">arch/i386/kernel/process.c</a>
3	sys_read	<a href="#">fs/read_write.c</a>
4	sys_write	<a href="#">fs/read_write.c</a>
5	sys_open	<a href="#">fs/open.c</a>
6	sys_close	<a href="#">fs/open.c</a>
7	sys_waitpid	<a href="#">kernel/exit.c</a>
8	sys_creat	<a href="#">fs/open.c</a>
9	sys_link	<a href="#">fs/namei.c</a>
10	sys_unlink	<a href="#">fs/namei.c</a>
11	sys_execve	<a href="#">arch/i386/kernel/process.c</a>
12	sys_chdir	<a href="#">fs/open.c</a>
...	...	...
184	sys_capget	<a href="#">kernel/capability.c</a>
185	sys_capset	<a href="#">kernel/capability.c</a>
186	sys_sigaltstack	<a href="#">arch/i386/kernel/signal.c</a>
187	sys_sendfile	<a href="#">mm/filemap.c</a>
190	sys_vfork	<a href="#">arch/i386/kernel/process.c</a>

%> arch/i386/kernel/entry.S



# System Call Overview

---

- Character of System Call

- 커널 루틴을 통하여 결과를 얻음
- 사용자 모드에서 커널 모드로 전환됨
- 0x80 인터럽트를 이용 (trap gate 사용)

- Type of System Call

- 입출력 조작
  - 입출력 디바이스에 대한 access
- 프로세스 제어
  - 프로세스가 자기 자신의 실행을 제어할 수 있도록 함
- 프로세스간 통신
  - 다른 프로세스로 정보를 송신할 수 있도록 하기 위함
- 타이밍 서비스
- 상태 정보의 제공

# System Calls in Linux

---

- Relating to processes

System call	Description
<code>pid = fork( )</code>	Create a child process identical to the parent
<code>pid = waitpid(pid, &amp;statloc, opts)</code>	Wait for a child to terminate
<code>s = execve(name, argv, envp)</code>	Replace a process' core image
<code>exit(status)</code>	Terminate process execution and return status
<code>s = sigaction(sig, &amp;act, &amp;oldact)</code>	Define action to take on signals
<code>s = sigreturn(&amp;context)</code>	Return from a signal
<code>s = sigprocmask(how, &amp;set, &amp;old)</code>	Examine or change the signal mask
<code>s = sigpending(set)</code>	Get the set of blocked signals
<code>s = sigsuspend(sigmask)</code>	Replace the signal mask and suspend the process
<code>s = kill(pid, sig)</code>	Send a signal to a process
<code>residual = alarm(seconds)</code>	Set the alarm clock
<code>s = pause( )</code>	Suspend the caller until the next signal

# System Calls (File) in Linux

---

- Linux System calls relating to files

System call	Description
<code>fd = creat(name, mode)</code>	One way to create a new file
<code>fd = open(file, how, ...)</code>	Open a file for reading, writing, or both
<code>s = close(fd)</code>	Close an open file
<code>n = read(fd, buffer, nbytes)</code>	Read data from a file into a buffer
<code>n = write(fd, buffer, nbytes)</code>	Write data from a buffer into a file
<code>position = lseek(fd, offset, whence)</code>	Move the file pointer
<code>s = stat(name, &amp;buf)</code>	Get a file's status information
<code>s = fstat(fd, &amp;buf)</code>	Get a file's status information
<code>s = pipe(&amp;fd[0])</code>	Create a pipe
<code>s = fcntl(fd, cmd, ...)</code>	File locking and other operations

# System Calls (File) in Linux

---

- The fields returned by the *stat* System calls

Device the file is on
I-node number (which file on the device)
File mode (includes protection information)
Number of links to the file
Identity of the file's owner
Group the file belongs to
File size (in bytes)
Creation time
Time of last access
Time of last modification

# System Calls (File) in Linux

---

- Linux System calls relating to directories

System call	Description
s = mkdir(path, mode)	Create a new directory
s = rmdir(path)	Remove a directory
s = link(oldpath, newpath)	Create a link to an existing file
s = unlink(path)	Unlink a file
s = chdir(path)	Change the working directory
dir = opendir(path)	Open a directory for reading
s = closedir(dir)	Close a directory
dirent = readdir(dir)	Read one directory entry
rewinddir(dir)	Rewind a directory so it can be reread

# Process 관리

A **process** is a program in execution. A process must have system resources, such as memory and the underlying CPU.

The kernel support illusion of concurrent execution of multiple processes by scheduling system resources among the set of processes that are ready to execute

# 프로세스

---

- 프로그램은 디스크에 저장되어 있는 기계어 명령어 코드의 집합
- 프로세스는 수행상태에 있는 프로그램을 의미
- 리눅스는 다중 태스크 환경 지원
  - 각 태스크 또는 수행 쓰레드를 프로세스라 함
  - 각 프로세스는 자신이 수행할 문맥(context)를 가짐
- 리눅스는 단일 쓰레드 프로세스 모델
  - 솔라리스 및 Window NT는 다중 쓰레드 프로세스 모델
- 리눅스는 공평성(fairness)를 보장하기 위한 스케줄링 정책을 제공
- 리눅스에서 태스크는 task\_struct 구조체로 표현됨

# 프로세스 정보 획득

---

- SYNOPSIS

#include <unistd.h>

pid\_t getpid(void); returns: process ID of calling process

pid\_t getppid(void); returns: parent process ID of calling process

pid\_t getuid(void); returns: real user ID of calling process

pid\_t geteuid(void); returns: effective user ID of calling process

pid\_t getgid(void); returns: real group ID of calling process

pid\_t getegid(void); returns: effective group ID of calling process



# 프로세스 생성

---

- 전통적인 유닉스 시스템에서는 부모 프로세스의 자원 복제를 통해서 새로운 프로세스를 생성함
  - 부모 프로세스의 모든 자원을 복제해서 자식 프로세스가 가지고 있어야 하므로 비효율성과 지연을 초래
  - 대부분의 자식 프로세스는 부모 프로세스의 일부 자원만 수정
    - 자식 프로세스 생성후 곧바로 Execve를 호출함으로 부모 프로세스의 자원이 필요치 않은 경우가 대부분
- 해결 방안
  - COW(Copy On Write) 방식
    - 실제로 물리 페이지를 접근할때 부모 프로세스의 자원을 자식 프로세스의 영역으로 복제
  - Lightweight process
    - 부모와 자식간의 주소공간을 공유하고 실행 문맥만 따로 저장함
    - 대부분의 프로세스 데이터 스트럭처를 공휴(파일, 주소공간)
  - Vfork
    - 부모 프로세스의 메모리 주소 공간을 자식 프로세스와 공유

# fork

---

- SYNOPSIS

`#include <unistd.h>`

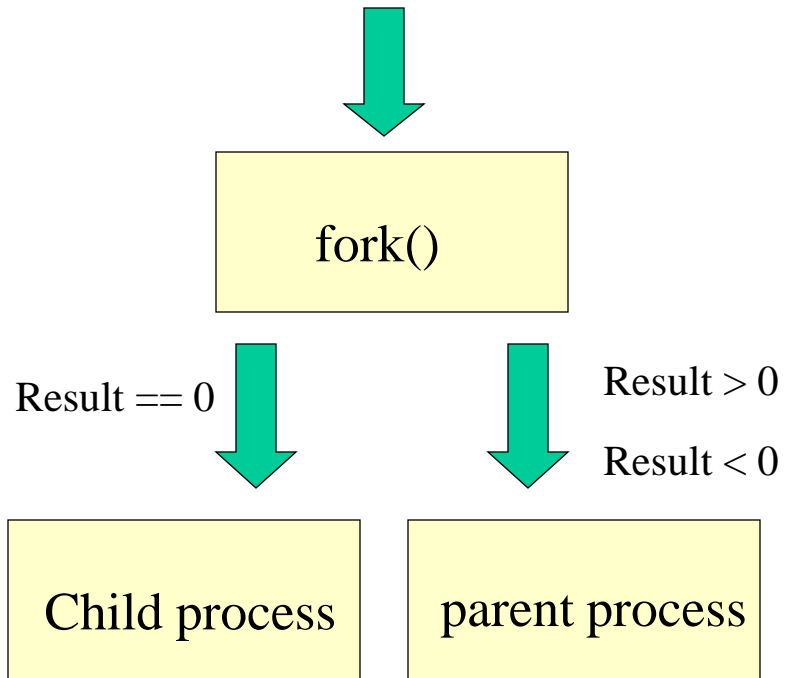
`pid_t fork(void);`

- DESCRIPTION

- 새로운 자식 프로세스 생성
- 파일 락(File lock)이나 전달된 시그널( pending signal)등은 상속되지 않음
- 리눅스에서 COW(copy-on-write page)를 통해서 구현
  - fork를 수행함으로써 발생하는 시스템 오버헤드는 부모 프로세스의 페이지 테이블을 복사 및 자식 프로세스의 고유한 태스크 스트럭처를 생성하는 하는 비용 및 시간이다.

- RETURN VALUE

- 성공적했을때, 부모 프로세스에게 자식 프로세스의 PID를 리턴
- 자식 프로세스는 0을 리턴
- 실패했을때, -1을 리턴하며, 자식 프로세스가 생성되지 않음



## 프로그램 소스 코드

```
if ( (pid = fork()) < 0)
    err_sys("fork error");

else if (pid == 0) {    /* child */
    자식 프로세스의 작업 수행
}

else /* parent */
    부모 프로세스의 작업 수행
```

# 리눅스 커널 내부의 Fork 수행 과정

---

1. 자식 프로세스의 프로세스 디스크립터와 커널 모드 스택을 저장할 공간(8KB)을 확보 : `alloc_task_struct()`
2. 확보된 공간에 부모 프로세스 디스크립터의 내용을 복사
3. 새롭게 생성될 프로세스를 저장할 엔트리가 있는지 확인함 : `find_empty_process()` - 리눅스에서는 `NR_TASKS`에 최대 태스크의 개수가 제한되어 있음
4. 새로운 프로세스에게 PID 할당 : `get_pid()`
5. 부모로부터 상속받은 프로세스 디스크립터의 내용중에서 상속받지 못한 내용을 채움 : 예를 들어서 프로세스 관계등
6. 자식 프로세스의 상태를 `TASK_RUNNING` 상태로 전환
7. 자식 프로세스의 PID를 부모 프로세스에게 돌려줌

# vfork

---

- SYNOPSIS

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

```
pid_t vfork(void);
```

- DESCRIPTION

- 새로운 프로세스를 생성한 후에 exec를 호출해서 프로그램을 수행시키고자 하는 목적으로 제공
- vfork는 fork 함수와 같은 호출 경로 및 리턴값을 가짐
- 부모 프로세스의 주소 공간을 자식 프로세스에게 복사하지 않고, 주소만 참조하는 방식 사용
- 자식 프로세스는 exec를 호출하기 전까지 부모 프로세스의 주소공간에서 수행

# Exec 관련 함수

---

- SYNOPSIS

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

```
int execl( const char *path, const char *arg, ...);
```

```
int execlp( const char *file, const char *arg, ...);
```

```
int execlp( const char *path, const char *arg , ..., char * const  
    envp[]);
```

```
int execv( const char *path, char *const argv[]);
```

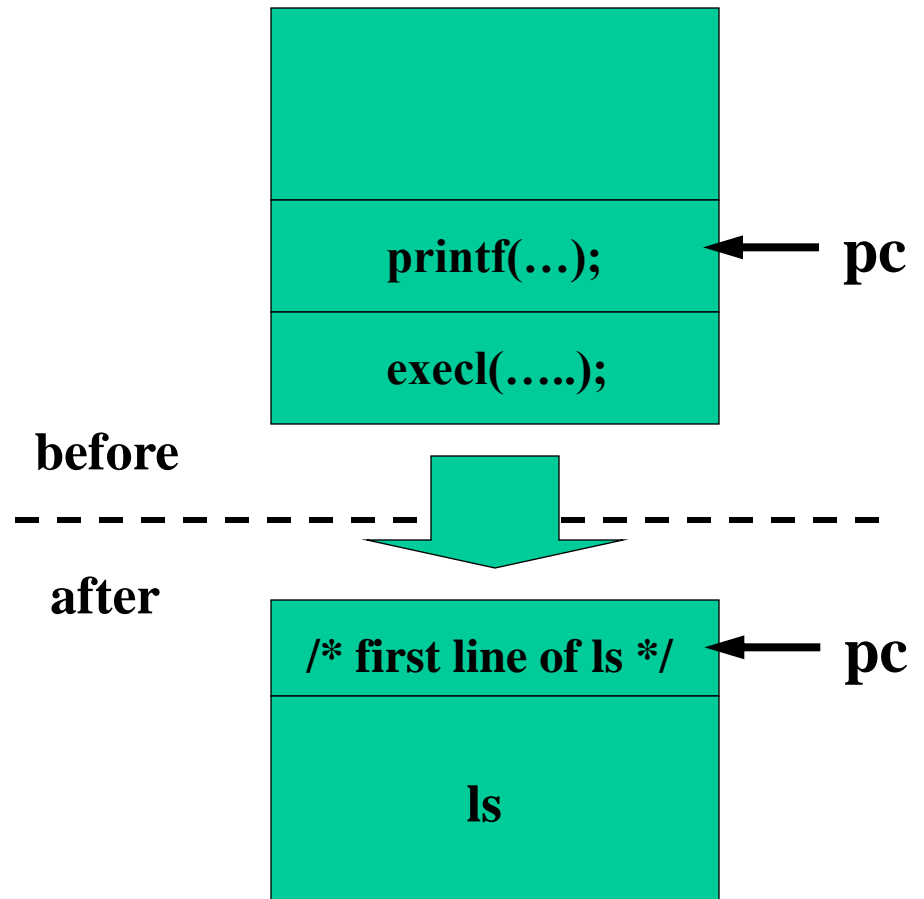
```
int execvp( const char *file, char *const argv[]);
```

- DESCRIPTION

- 새로운 프로그램 수행을 위한 함수
- 현재 프로세스를 새로운 프로세스로 대체

# execl 예제

```
main()
{
    printf("executing ls\n");
    execl("/bin/ls", "ls", "-l", (char*)0);
    perror("execl failed to run ls");
    exit(1);
}
```

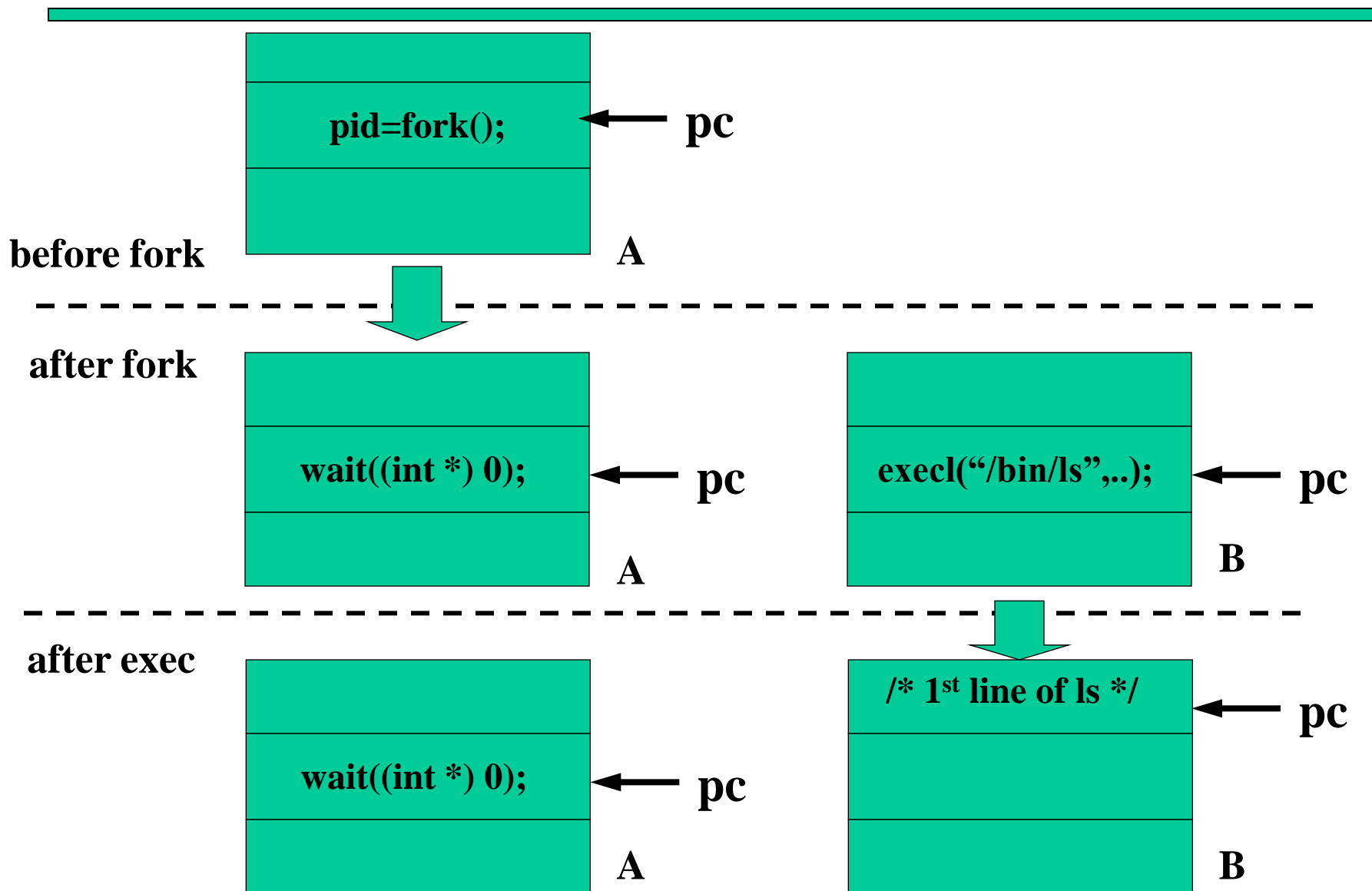


# fork 호출 후 exec 사용

---

```
main()
{
    pid_t pid;
    switch(pid = fork()){
        case -1: perror("fork failed"); exit(1);break;
        case 0 : execl("/bin/ls", "ls", "-l", (char*)0); /* child */
                perror("fork failed"); exit(1);break;
        default : wait((int *) 0); /* parent */
                printf("ls completed\n"); exit(0);
    }
}
```





# 프로세스 종료

---

- 정상적인 종료(normal termination)
  - main 함수로부터 return 수행
  - exit 함수를 호출
  - \_exit 함수를 호출 : exit 함수는 \_exit 함수를 호출
- 비정상적인 종료(abnormal termination)
  - abort 호출
  - 프로세스가 시그널을 받을때(자기 자신, 다른 프로세스, 커널)
    - 참조할 수 없는 주소 영역을 참조하려고 시도하는 경우
    - 0 으로 어떤값을 나누려고 시도하는 경우
- 관련 함수
  - exit, atexit

# exit

---

- SYNOPSIS

`#include <stdlib.h>`

`void exit(int status);`

- DESCRIPTION

- 현재 수행중인 프로세스를 종료
- 열린 파일 디스크립터를 닫음
- 부모 프로세스가 **wait**를 호출하고 블럭킹 상태에 있으면, 부모 프로세스에게 이벤트를 발생시켜 진행시킴
- 프로그래머가 정의한 **exit** 핸들링 루틴을 실행
- **clean-up** 액션에 정의된 나머지 작업 수행

# atexit

---

- SYNOPSIS

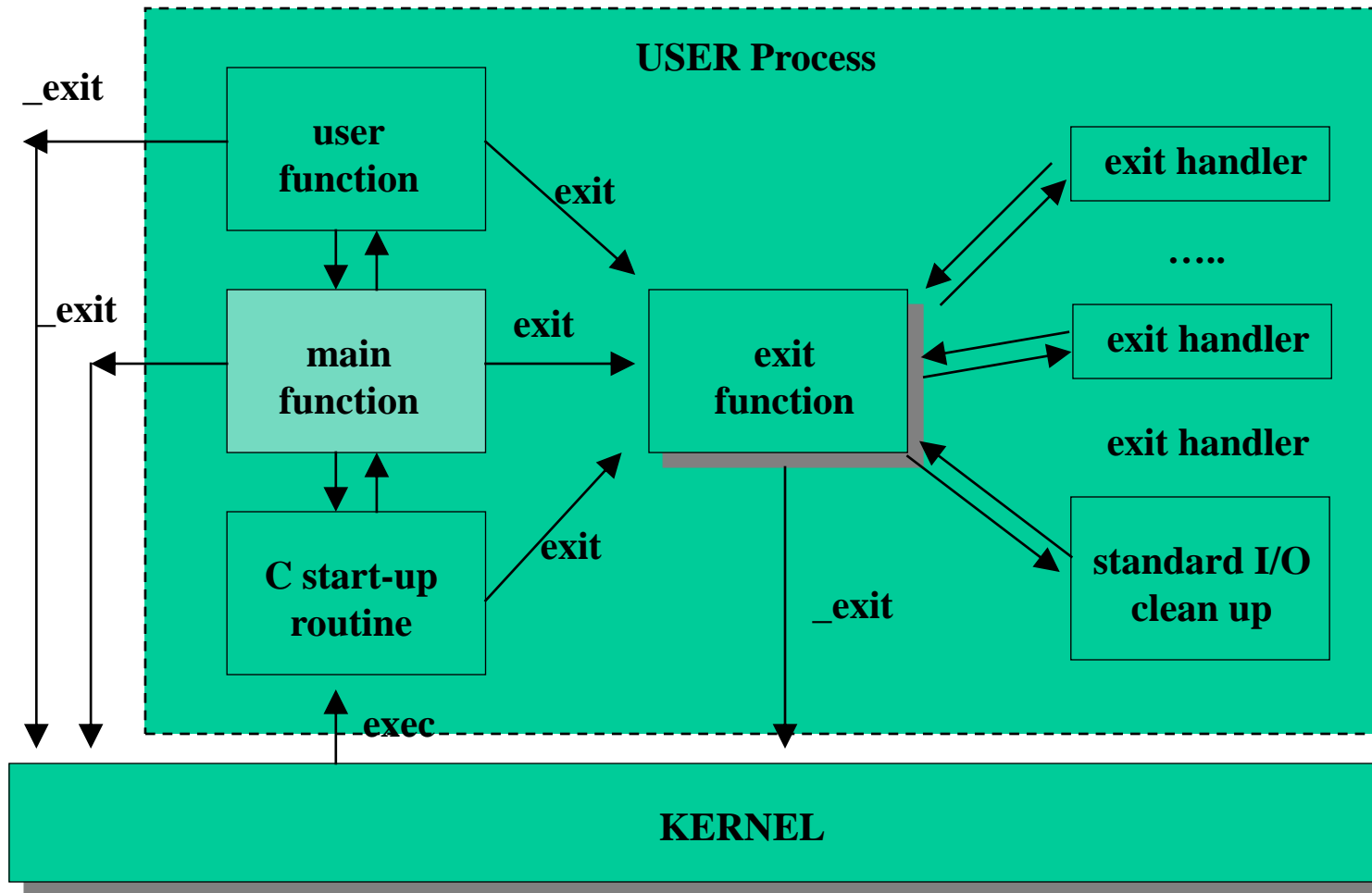
```
#include <stdlib.h>
```

```
int atexit(void (*function) (void));
```

- DESCRIPTION

- 프로그램이 종료될때 불러지는 함수 등록
- ANSI C에서는 32개까지 함수를 등록할 수 있으며, **exit** 함수에 의해서 각 함수가 자동적으로 호출되어짐

## How a C program is started and how it terminated



# wait

---

- SYNOPSIS

```
#include <sys/types.h>
```

```
#include <sys/wait.h>
```

```
pid_t wait(int* status);
```

```
pid_t waitpid(pid_t pid, int * status, int options);
```

- DESCRIPTION

- 프로세스간 동기화(synchronizing process) 제공
- 자식 프로세스가 수행중인 동안 일시적으로 프로세스의 수행을 중지
- 자식 프로세스가 종료하는 경우, 수행 재개
- 특정한 자식 프로세스의 종료를 기다리는 경우 waitpid 사용

- RETURN VALUE

- 성공 : 수행이 종료된 자식 프로세스의 PID 리턴
- 실패 : -1

# pstree

- 프로세스 계층도를 보여주는 프로그램



```
xterm
matrix:/homes/yuko> pstree
init--atd
|-automount
|-crond
|-fsgs
|-gpm
|-httpd---10*[httpd]
|-inetd--in.telnetd---login---tcsh
|   `--in.telnetd---login---tcsh---pstree
|-kflushd
|-klogd
|-kpiod
|-kswapd
|-lpd
|-mdrecoveryd
|-6*[mingetty]
|-nfsd---lockd---rpciod
|-7*[nfsd]
|-nmbd---nmbd
|-nscd---nscd---5*[nscd]
|-portmap
|-rpc.mountd
|-rpc.rquotad
|-rpc.statd
|-safe_mysqld---mysqld---mysqld---mysqld
|-smbd---7*[smbd]
|-syslogd
|-ucc-bin
|-update
|-xfs
|   `--xterm---tcsh---2*[xterm---tcsh]
matrix:/homes/yuko> █
```

# top

- 프로세스 정보를 보여주는 프로그램

```
xterm
4:39pm up 7 days, 14:19, 2 users, load average: 0.00, 0.00, 0.00
82 processes: 81 sleeping, 1 running, 0 zombie, 0 stopped
CPU states: 2.0% user, 2.1% system, 0.0% nice, 95.7% idle
Mem: 257724K av, 254624K used, 3100K free, 54608K shrd, 80784K buff
Swap: 2048248K av, 18172K used, 2030076K free 139808K cached
```

PID	USER	PRI	NI	SIZE	RSS	SHARE	STAT	LIB	%CPU	%MEM	TIME	COMMAND
594	nobody	17	-5	18692	2068	256	S<	0	1.8	0.8	16:51	ucc-bin
576	root	17	0	1360	1360	1092	S	0	1.6	0.5	11:13	fsgs
8048	yuko	11	0	1016	1016	816	R	0	0.5	0.3	0:00	top
1	root	0	0	484	484	412	S	0	0.0	0.1	0:01	init
2	root	0	0	0	0	0	SW	0	0.0	0.0	0:00	kflushd
3	root	0	0	0	0	0	SW	0	0.0	0.0	0:00	kpiod
4	root	0	0	0	0	0	SW	0	0.0	0.0	0:03	kswapd
5	root	-20	-20	0	0	0	SW<	0	0.0	0.0	0:00	mdrecoveryd
248	bin	0	0	416	408	332	S	0	0.0	0.1	0:00	portmap
299	root	0	0	804	796	592	S	0	0.0	0.3	0:00	nsd
301	root	0	0	804	796	592	S	0	0.0	0.3	0:02	nsd
302	root	0	0	804	796	592	S	0	0.0	0.3	0:00	nsd
303	root	0	0	804	796	592	S	0	0.0	0.3	0:00	nsd
304	root	0	0	804	796	592	S	0	0.0	0.3	0:00	nsd
305	root	0	0	804	796	592	S	0	0.0	0.3	0:00	nsd
306	root	0	0	804	796	592	S	0	0.0	0.3	0:00	nsd
318	root	0	0	504	500	404	S	0	0.0	0.1	0:00	syslogd
329	root	0	0	676	672	304	S	0	0.0	0.2	0:00	klogd
343	daemon	0	0	300	292	228	S	0	0.0	0.1	0:00	atd
357	root	0	0	556	556	464	S	0	0.0	0.2	0:00	crond
375	root	0	0	448	444	372	S	0	0.0	0.1	0:00	inetd
389	root	0	0	464	456	384	S	0	0.0	0.1	0:00	lpd
406	root	0	0	424	420	344	S	0	0.0	0.1	0:00	rpc.statd
417	root	0	0	188	144	116	S	0	0.0	0.0	0:00	rpc.rquotad
428	root	0	0	220	152	128	S	0	0.0	0.0	0:00	rpc.mountd
440	root	0	0	0	0	0	SW	0	0.0	0.0	0:01	nfsd



# 프로세스 관리

---

- 새로운 프로세스를 생성
  - Task
- 스케줄링
  - 어떤 프로세스가 CPU를 사용해야 하는지 결정
  - SCHED\_OTHER, SCHED\_RR, SCHED\_FIFO
- 타이머 관리
- 프로세스간 시그널 전송
- 프로세스가 종료될 때 프로세스가 사용했던 자원의 반납을 관리

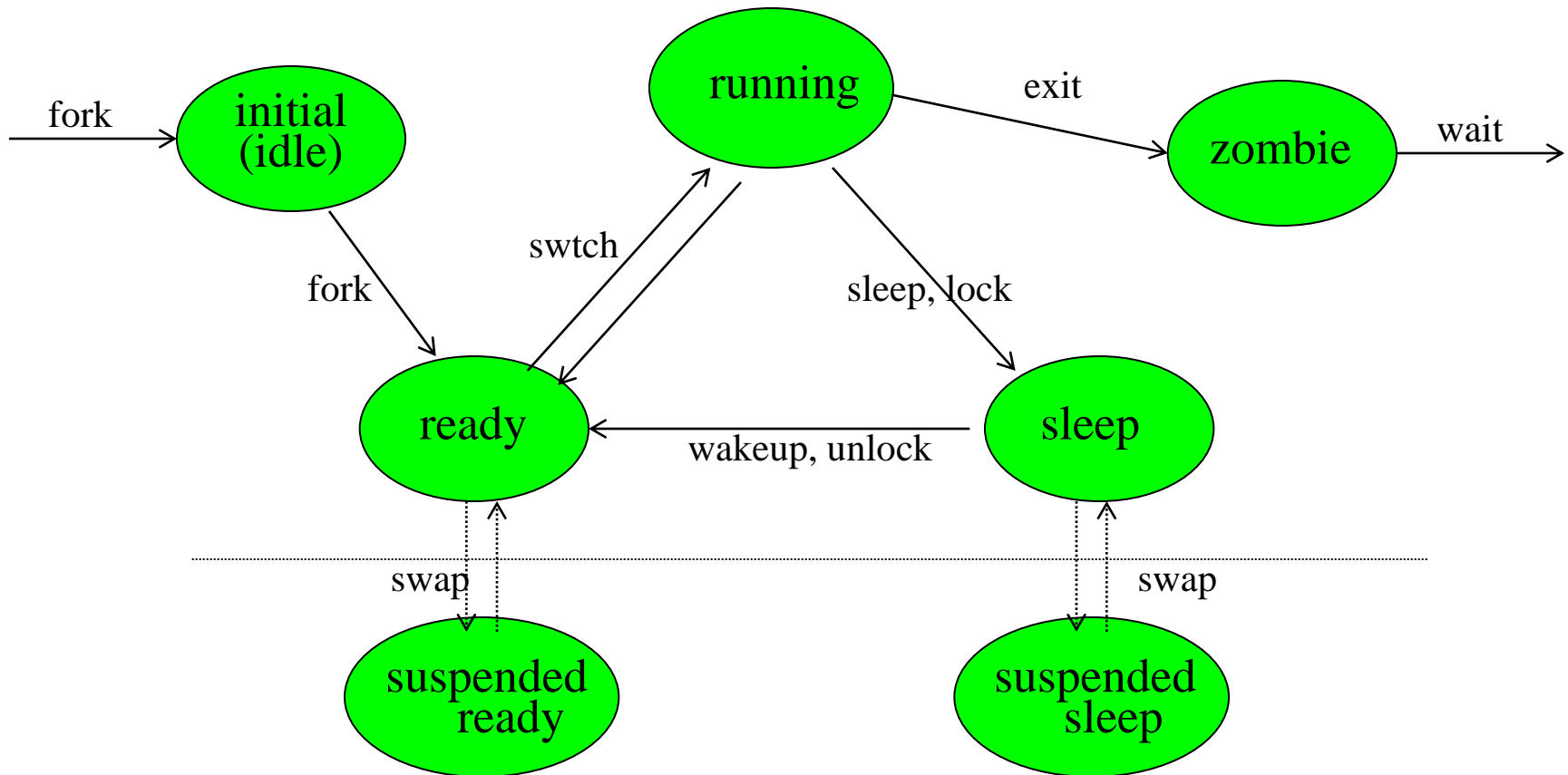
# 태스크란?

---

- 태스크 정의
  - 수행 중인 프로그램 (an instance of a running program)
  - 프로그램의 수행 환경 (an execution environment of a program)
  - 스케줄링 단위 (scheduling entity)
  - 제어 흐름과 주소 공간의 집합 (a control flow and address space)
  - multitasking을 지원하는 객체
- 커널의 태스크 관리
  - 생성과 소멸 (fork, exit)
  - 수행 (execve)
  - 문맥 교환 (context switch)
  - 수면과 깨움 (sleep, wakeup)
  - 사용자 수준/커널 수준 실행 (user level/kernel level running)

# 태스크 상태와 전이 (1/3)

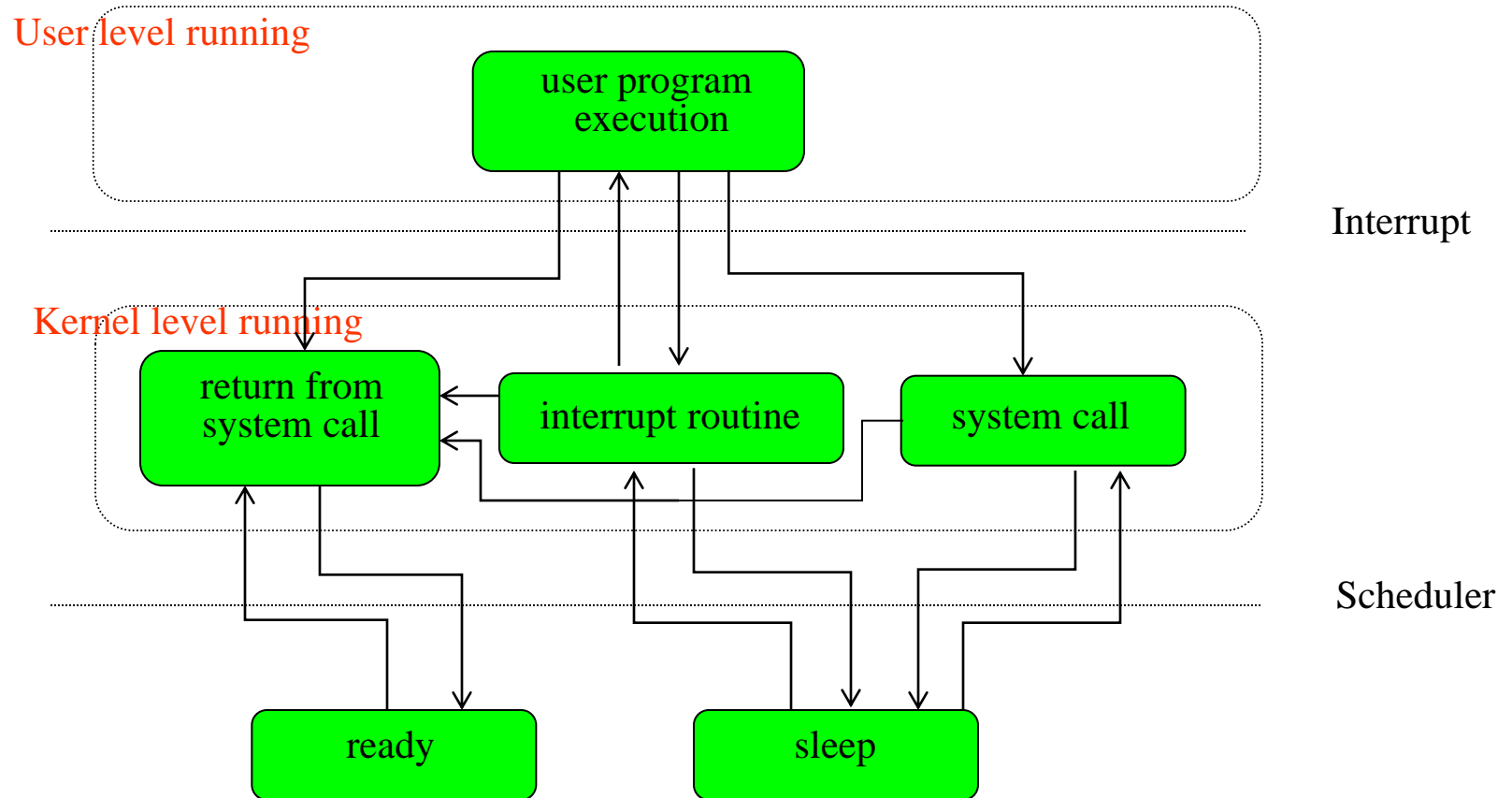
- 태스크 상태(State)와 전이(Transition)
  - 준비(ready) 상태, 실행(running) 상태, 수면(sleep) 상태



(Source : UNIX Internals)

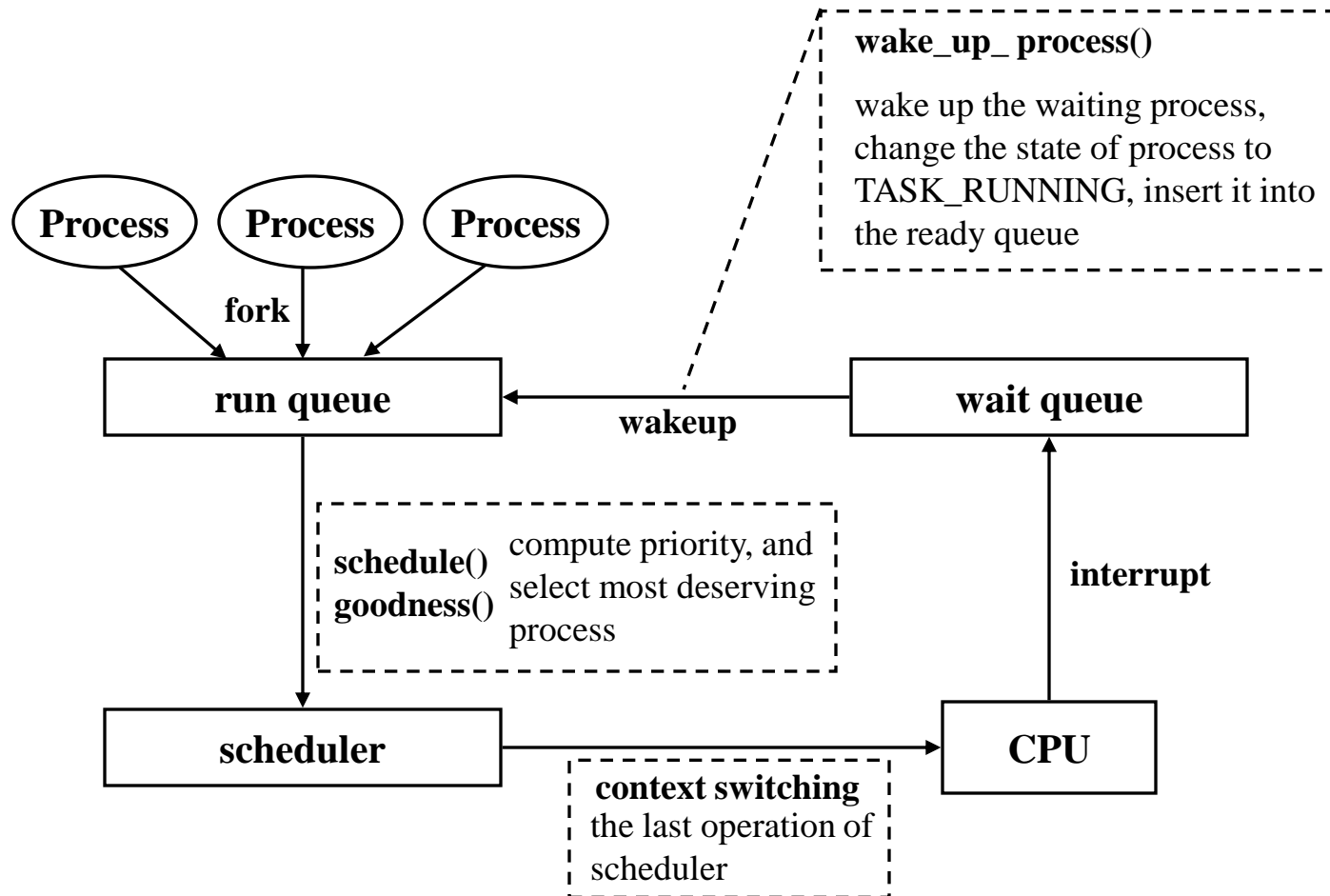
# 태스크 상태와 전이 (2/3)

- 실행 상태 구분 : 사용자 수준 실행/커널 수준 실행



# 태스크 상태와 전이 (3/3)

리눅스 프로세스 스케줄링 및 관련 함수



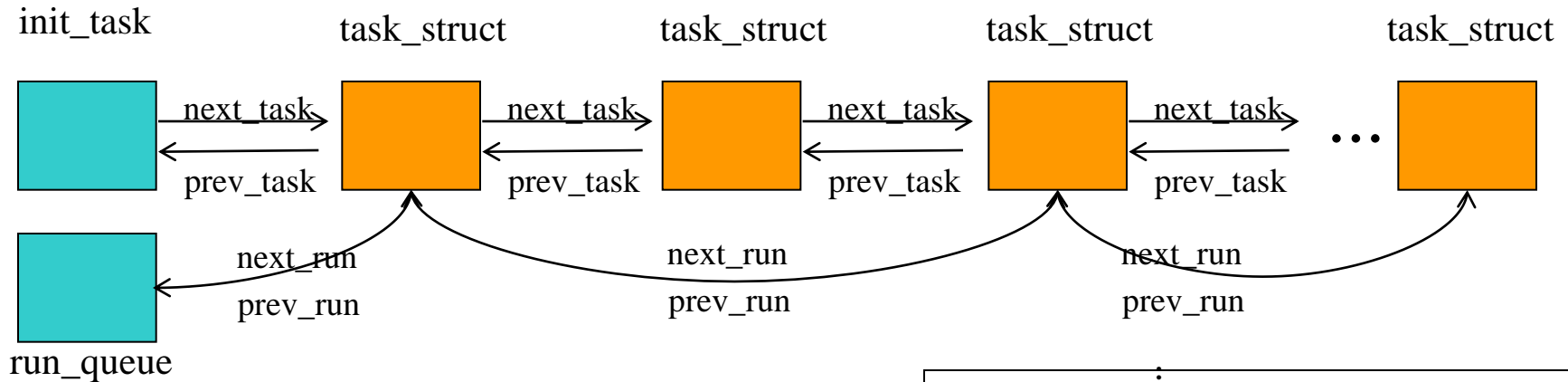
# 시스템 프로그래밍 구조- Task(1/11)

- **task\_struct 자료 구조** /\* include/linux/sched.h \*/
  - task identification : pid, pgrg, session, uid, euid, suid, fsuid
  - state : TASK\_RUNNING, TASK\_ZOMBIE, TASK\_INTERRUPTIBLE, TASK\_UNINTERRUPTIBLE, TASK\_STOPPED
  - task relationship : p\_pptr, p\_cptr, next\_task, next\_run
  - scheduling information : policy, priority, counter, rt\_priority, need\_resched
  - memory information : mm\_struct
  - signal information : signal\_struct, sigpending, signal, blocked
  - file information : files\_struct, fs\_struct
  - thread information : tss
  - time information : start\_time, times, timer\_list
  - executable format : personality, exec\_domain
  - resource limits : rlim
  - miscellaneous : flag, comm, maj\_flt, min\_flt, exit\_code 등

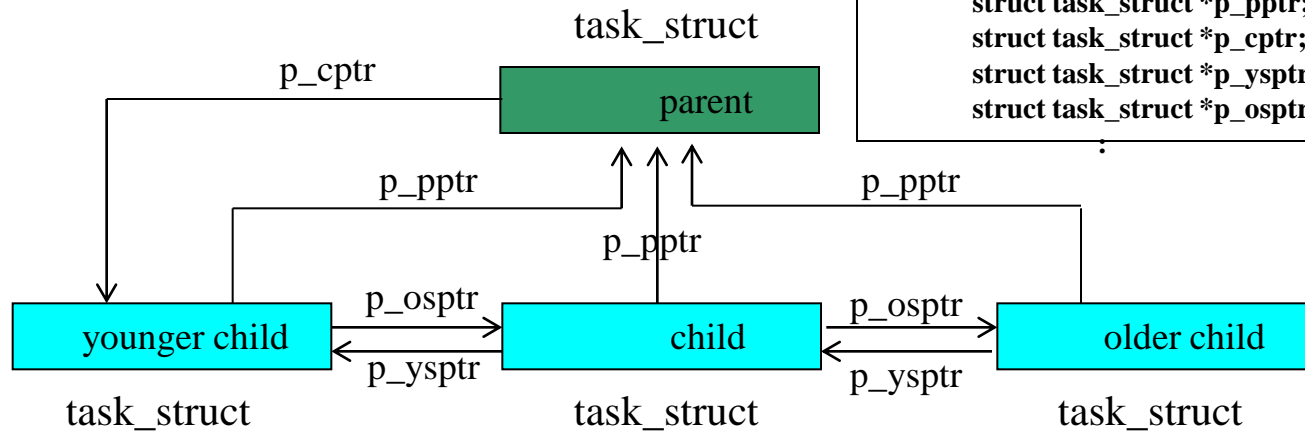
시스템 문맥에서  
가장 핵심적인 자료  
구조

# 시스템 프로그래밍 구조-Task(2/11)

- 태스크 연결 구조



- 태스크 가족 관계

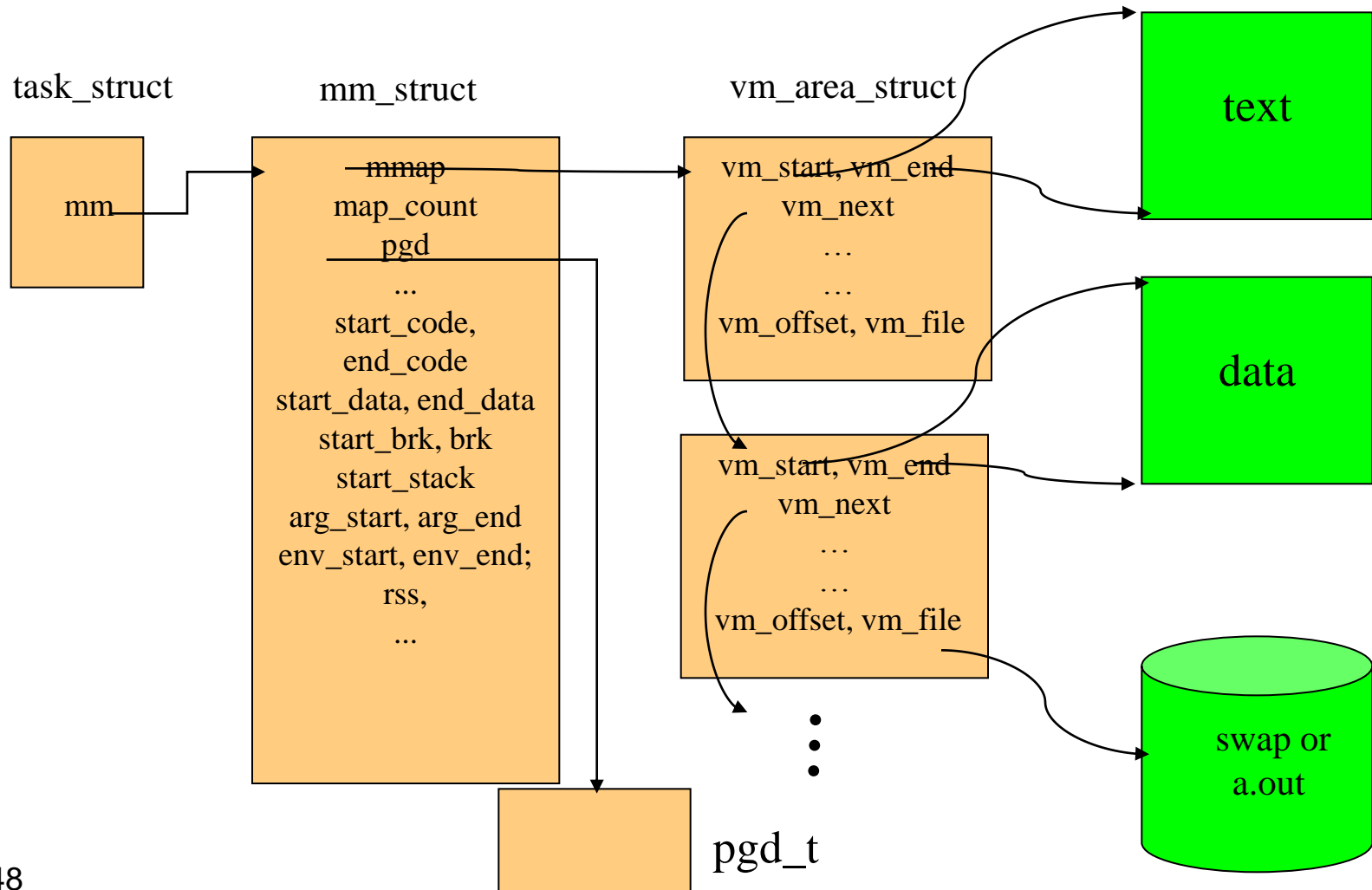


```
struct task_struct *next_task;  
struct task_struct *prev_task;  
  
struct task_struct *p_opptr; /* original parent */  
struct task_struct *p_pptr; /* parent */  
struct task_struct *p_cptr; /* youngest child */  
struct task_struct *p_ysptr; /* younger sibling */  
struct task_struct *p_osptr; /* older sibling */
```

# 시스템 프로그래밍 구조- Task(3/11)

## ■ 메모리 관리 자료 구조

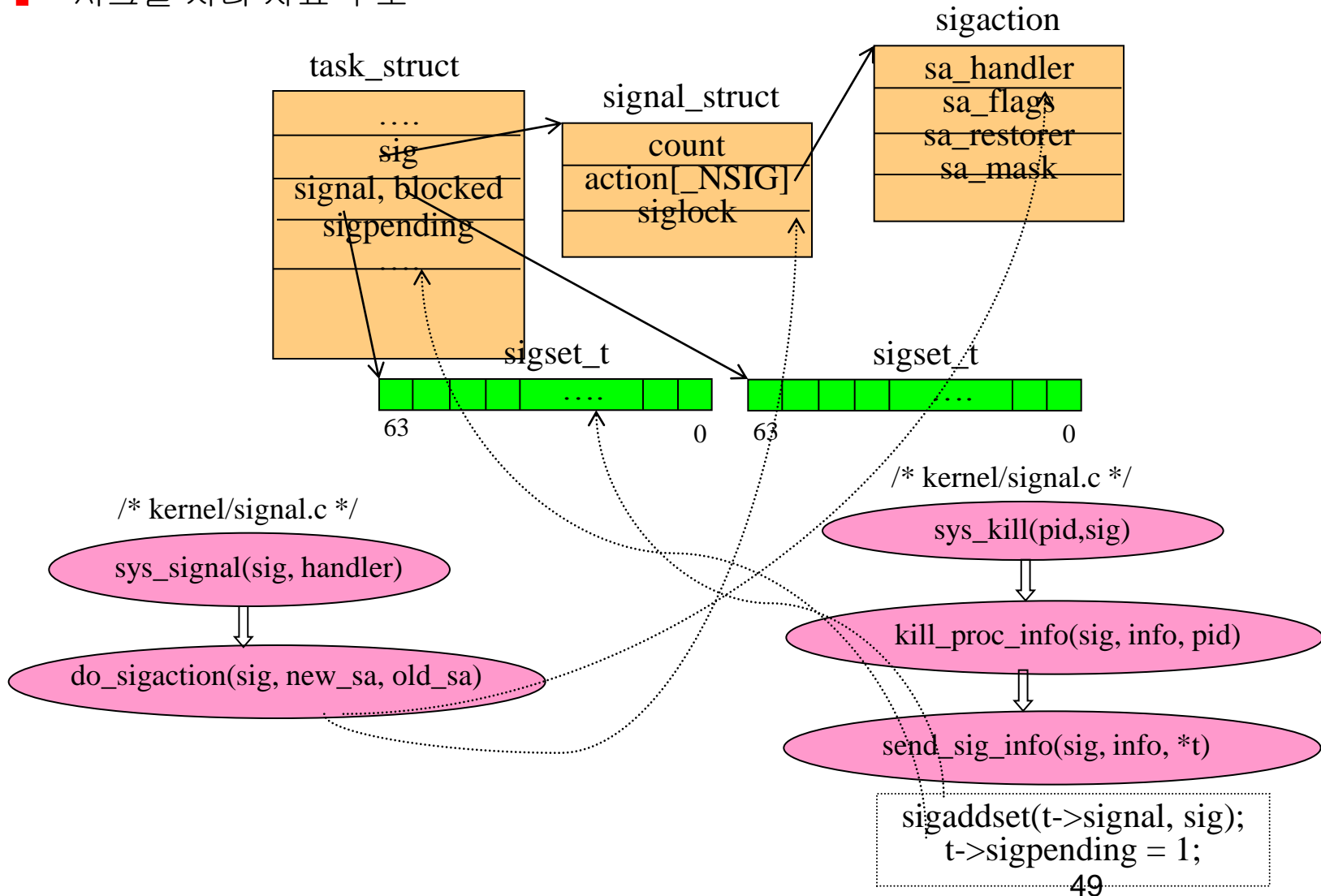
✓ include/linux/sched.h, include/linux/mm.h, include/asm-i386/page.h





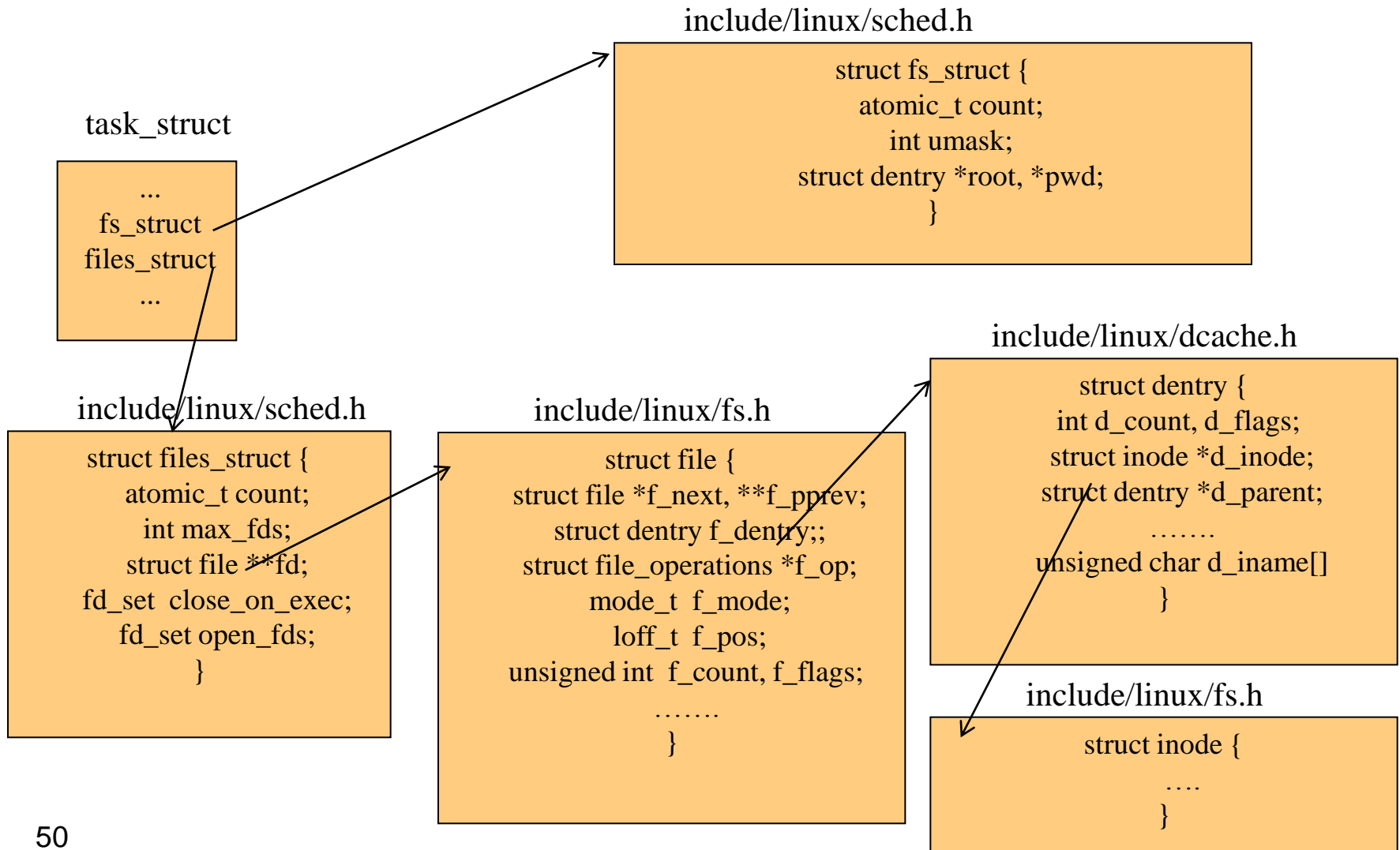
# 시스템 프로그래밍 구조-Task(4/11)

## ■ 시그널 처리 자료 구조



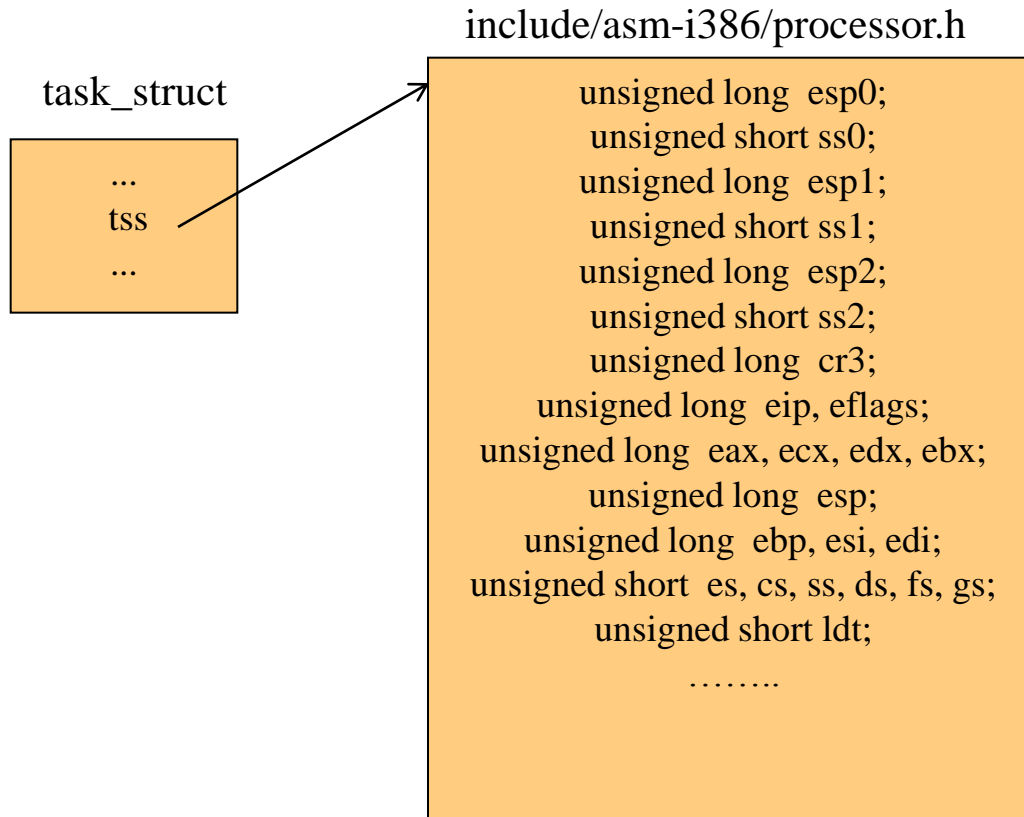
# 시스템 프로그래밍 구조-Task(5/11)

- 파일 관리 자료 구조 : fd, inode



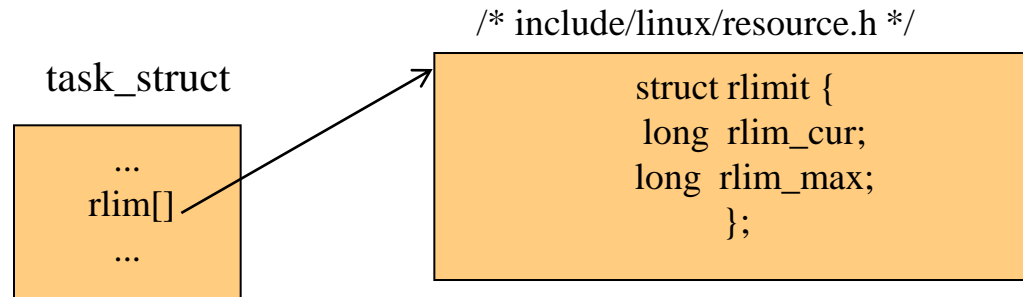
# 시스템 프로그래밍 구조- Task(6/11)

- 쓰레드 자료 구조 : CPU 추상화



# 시스템 프로그래밍 구조- Task(7/11)

## ■ 태스크의 사용 가능 자원 제한



/\* include/asm-i386/resource.h \*/

```
RLIM_NLIMIT      10  
RLIMIT_CPU       0  
RLIMIT_FSIZE     1  
RLIMIT_DATA      2  
RLIMIT_STACK     3  
RLIMIT_CORE      4  
RLIMIT_RSS       5  
RLIMIT_NPROC     6  
RLIMIT_NOFILE    7  
RLIMIT_MEMLOCK   8  
RLIMIT_AS        9
```

# 시스템 프로그래밍 구조- Task(8/11)

---

- LINUX Task 스케줄링

- clock tick은 10msec마다 발생
- 각 태스크에게 할당되는 시간 할당량(time quantum)은 10 clock ticks
- 실시간 태스크 지원
- task\_struct에서 스케줄링 관련 변수 (`/* include/linux/sched.h */`)
  - policy
    - 태스크 유형 (task type)
    - SCHED\_FIFO : 실시간 태스크, 비선점형
    - SCHED\_RR : 실시간 태스크, 선점형
    - SCHED\_OTHER : 일반 태스크, 선점형
  - priority
    - 태스크 우선순위
    - 태스크가 생성될 때 기본 값인 20 (DEF\_PRIORITY)으로 설정
    - `sys_nice()`나 `sys_setpriority()` 시스템 호출로 변경 가능

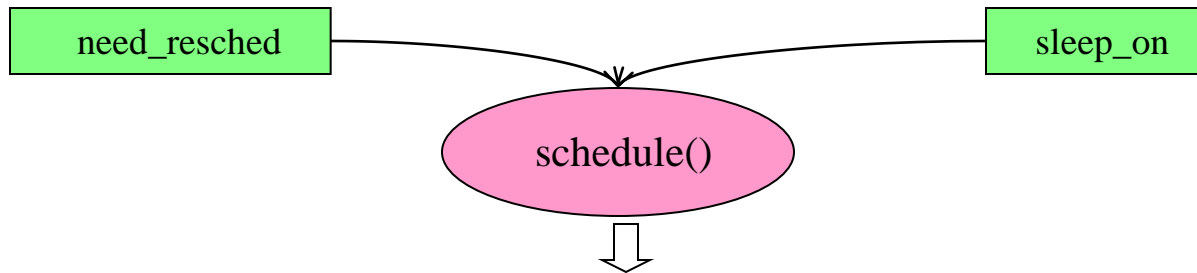
# 시스템 프로그래밍 구조- Task(9/11)

---

- counter
  - 태스크의 처리기 사용량
  - 태스크가 생성될 때 `p_priority` 값으로 설정
  - 태스크가 수행중일 때, `clock tick`이 발생하면 1씩 감소
  - 모든 태스크의 `p_counter` 값이 0이 되면, 모든 태스크의 `p_counter` 값을 `p_priority` 값으로 재 설정
  - 비 실시간 태스크의 경우, 스케줄러는 `p_priority + p_counter` 값이 가장 큰 태스크를 선택하여 수행시킨다.
- need\_resched
  - 스케줄링이 수행될 필요가 있으면 1로 설정
  - 커널 수준 실행 상태에서 사용자 수준 실행 상태로 전이될 때 (시스템 호출이나 인터럽트 처리를 마칠 때), 이 변수를 조사하여 1이면 스케줄러 호출
- rt\_priority
  - 실시간 태스크의 우선 순위, 보통 1000 이상의 값을 갖는다.
  - `sched_setscheduler(pid, policy, sched_param)` 시스템 호출로 설정
  - 실시간 태스크의 경우, 스케줄러는 이 값이 가장 큰 태스크를 선택하여 수행시킨다.

# 시스템 프로그래밍 구조-Task(10/11)

- 스케줄링 함수 : `schedule()` function /\* kernel/sched.c \*/



- schedule real time task first (rt\_priority)  
- select a task which has highest values of  
counter + priority (using goodness function)  
give advantage to the task which run this\_cpu  
give slight advantage to the task which has mm object  
- if (counter == 0) for all task  
    counter = priority  
- context switch : `switch_to (current, next)`  
/\* arch/i386/kernel/process.c \*/

# 시스템 프로그래밍 구조-Task(11/11)

- Linux 스케줄링 예
- 가정 : 시스템에 비 실시간 태스크 3개 존재, 태스크는 자발적으로 sleep하지 않음

millisecond	T1		T2		T3	
	priority	counter	priority	counter	priority	counter
0	20	20	20	20	20	20
10	20	10	20	20	20	20
20	20	10	20	10	20	20
30	20	10	20	10	20	10
40	20	0	20	10	20	10
	20	0	20	0	20	10
	20	20	20	20	20	20



# 태스크 정리

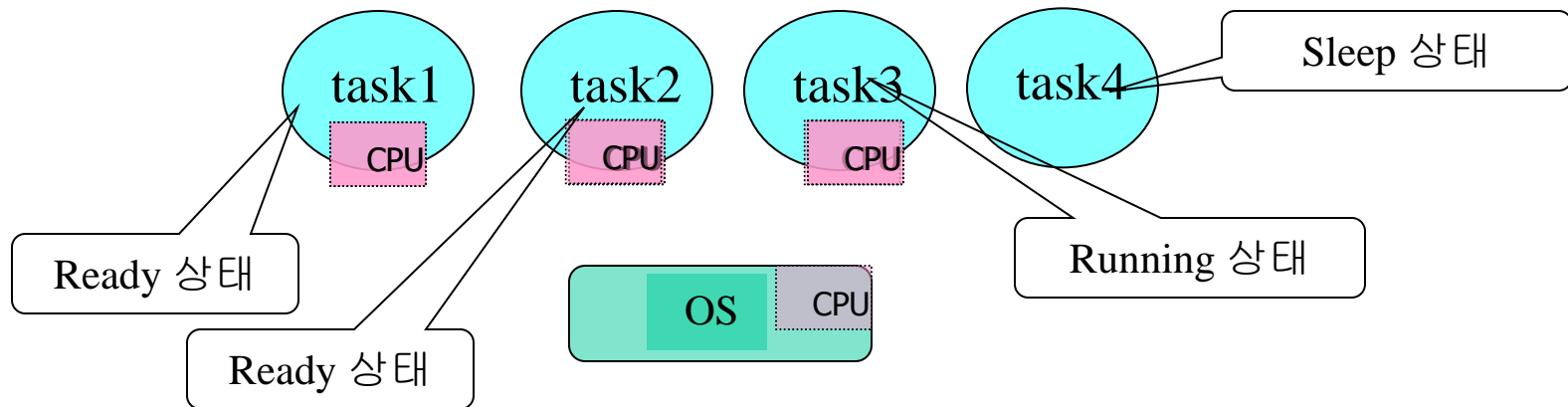
- 태스크란?

수행중인 프로그램 (생명을 갖게 됨)

- 다음 용어의 차이점은?

- ✓ 다중 태스킹 (Multitasking) 시스템
- ✓ 다중 프로그래밍 (Multiprogramming) 시스템
- ✓ 다중 프로세서 (Multiprocessor) 시스템

- Multitasking 시스템은 시분할 (timesharing) 기법 사용



- OS는 각 태스크마다 상태 등의 정보를 유지 (eg. 작업관리자)

# 시스템 프로그래밍 구조- 스케줄러

- 스케줄러

- 멀티 테스킹을 지원하며 실시간 스케줄링 정책을 지원하기 위해서 계속 진화
- 리눅스 스케줄러의 실행 단위 - 커널 쓰레드/ 사용자 프로세스 / 사용자 쓰레드
- 실시간 스케줄링 정책
  - 2.6 커널에서는 O(1) 스케줄러라는 새로운 스케줄러가 도입, POSIX 타이머와 같은 실시간 기능들이 커널에 포함
  - clock tick은 10msec마다 발생
  - 각 태스크에게 할당되는 시간 할당량(time quantum)은 10 clock ticks
  -

```
HostPC> top // 평균 시스템 부하를 보여줌, 시스템 가동 시간과 부하의 숫자
// PID USER PRI NI VIRT(가상 메모리 사용량) SHR(분할 페이지) S(프로세서 상태)
```

# 태스크 스케줄링 (1/4)

---

- 스케줄링 정의
  - 자원을 특정 객체가 사용할 수 있도록 할당하는 것.
  - 태스크 스케줄링에서는 처리기가 자원이 되고 태스크가 객체가 된다.
- 태스크 스케줄링 기준
  - 공정성(fairness) : 기아 상태(starvation)가 되는 태스크가 없어야 된다.
  - 효율성(efficiency) : 태스크 선택 과정이 빠르게 수행되어야 한다.
  - 응답 시간(response time) vs 처리율(throughput)
- 태스크 유형 (type of task)
  - Interactive
  - Batch (Computation-Bound)
  - Real-time

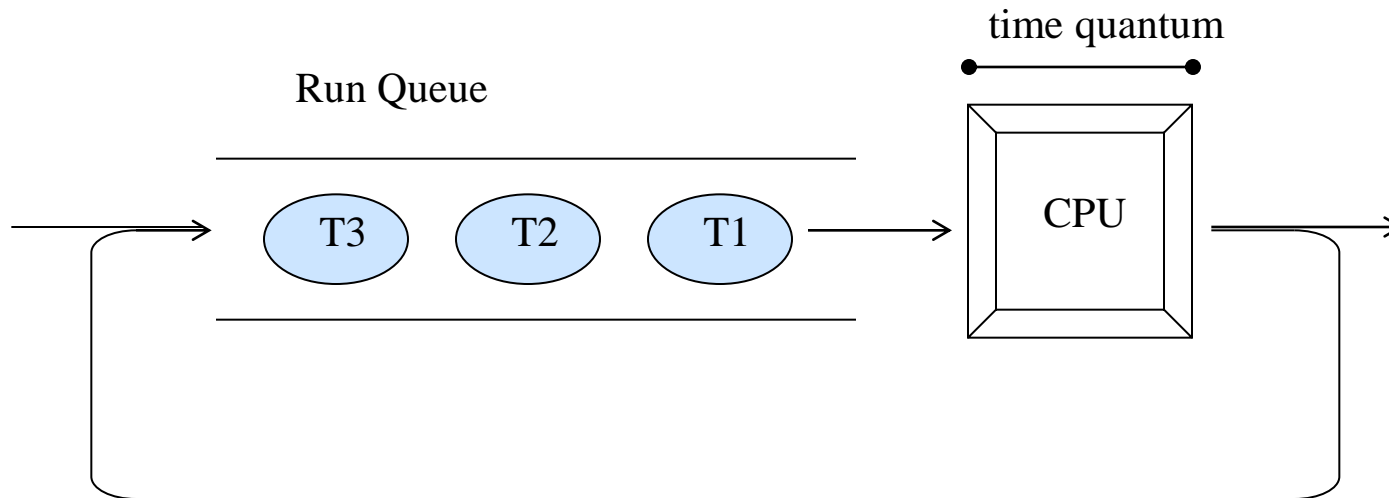
# 태스크 스케줄링 (2/4)

---

- 스케줄링 유형 (types of scheduling)
  - 선점형 스케줄링 (Preemptive scheduling)
  - 비선점형 스케줄링 (Non preemptive scheduling)
- 스케줄링 알고리즘의 예
  - FCFS (First Come First Service)
  - 라운드-로빈(Round-Robin)
  - SJF (Short Job First)
  - 다단계 피드백 큐(Multilevel Feedback Queue)
  - EDF (Earliest Deadline First)
  - RM (Rate Monotonic)
  - Fair Queuing
  - Gang Scheduling
  - Scheduling for Clustering System
  - Process Migration

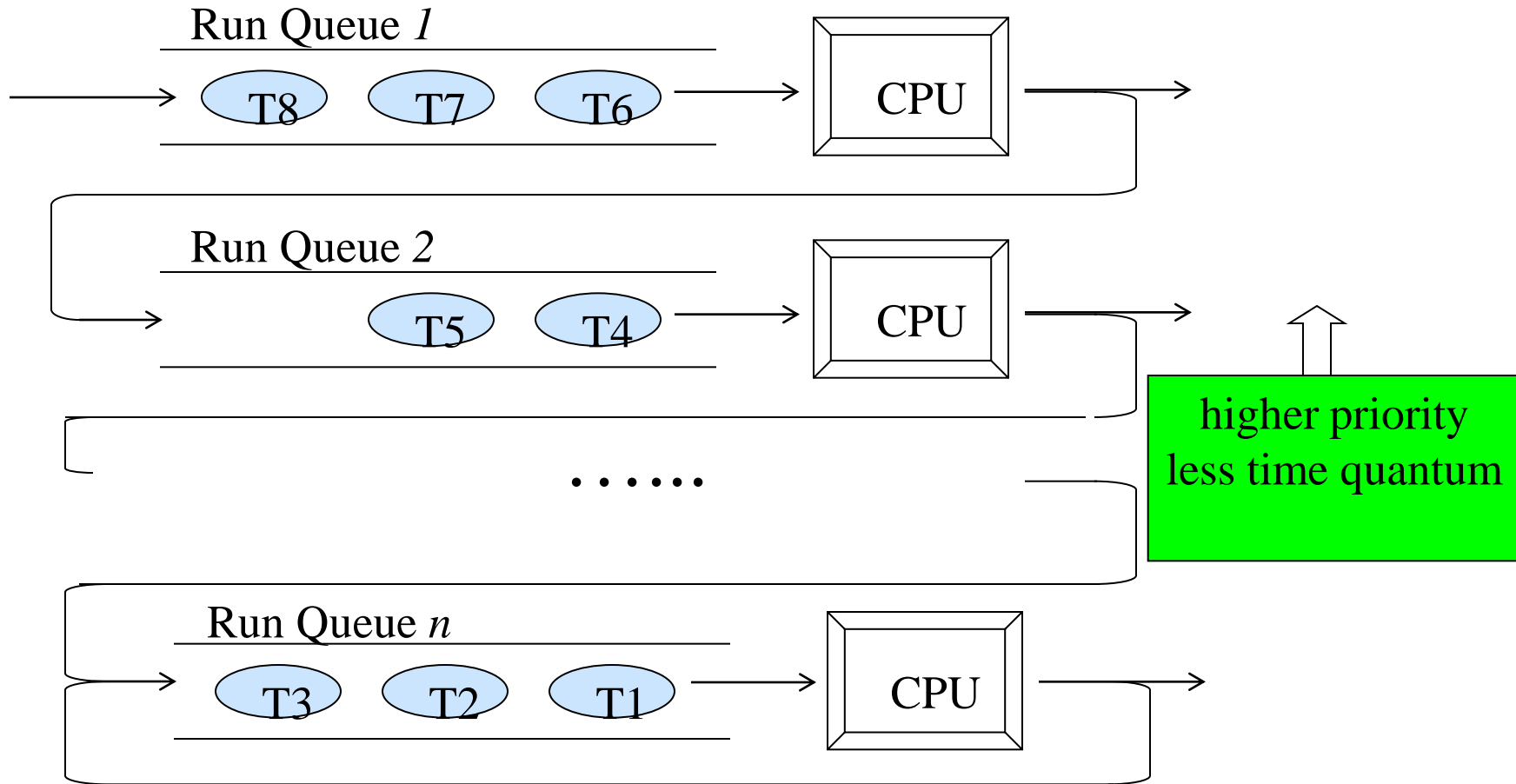
# 태스크 스케줄링 (3/4)

- UNIX 스케줄링
  - Round Robin



# 태스크 스케줄링 (4/4)

## - Multilevel Feedback Queue



- interactive task의 효과적인 처리 가능

# 스케줄링 관련 시스템 호출

---

int sched\_setscheduler(pid\_t pid, int policy, struct sched\_param \*p);

int sched\_getscheduler(pid\_t pid);

- pid의 프로세스의 스케줄링 정책 설정 및 현재 설정 상태 검색
- policy 인자는 SCHED\_OTHER, SCHED\_FIFO, SCHED\_RR

int sched\_get\_priority\_max(int policy);

int sched\_get\_priority\_min(int policy);

- policy가 갖을 수 있는 우선 순위 값의 범위 제공
- SCHED\_OTHER는 항상 0
- SCHED\_FIFO와 SCHED\_RR은 1에서 99 사이

int sched\_setparam(pid\_t pid, const struct sched\_param \*p);

int sched\_getparam(pid\_t pid, struct sched\_param \*p);

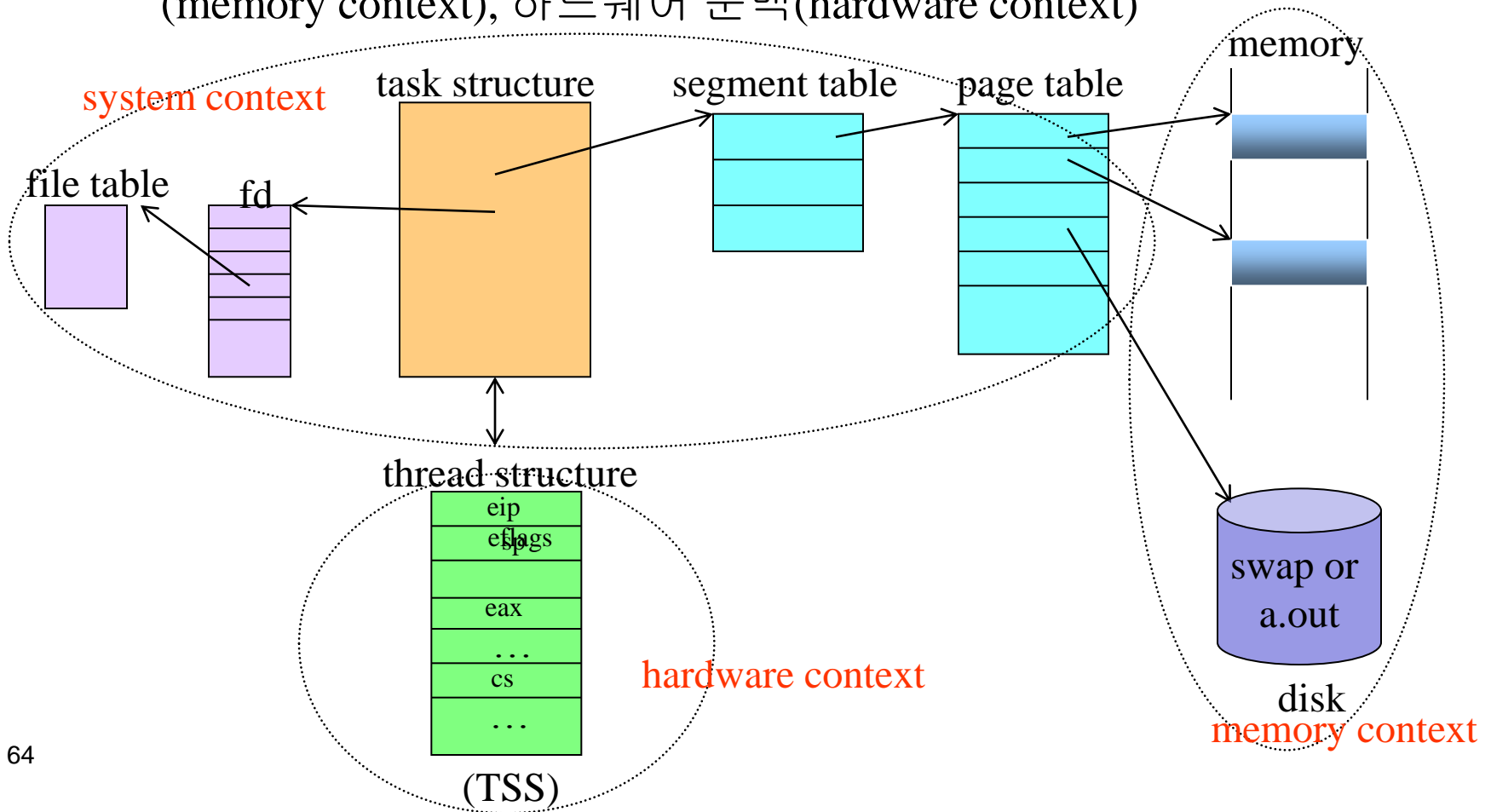
- pid 프로세스의 스케줄링 정책 정보 설정 및 검색
- 우선 순위 조작

int sched\_yield(void);

- 이 함수를 호출한 프로세스를 중단함

# 문맥 (Context)

- 문맥 (Context) :
  - 커널이 관리하는 태스크의 자원과 수행 환경 집합
  - 3 부분으로 구성 : 시스템 문맥 (system context), 메모리 문맥 (memory context), 하드웨어 문맥 (hardware context)





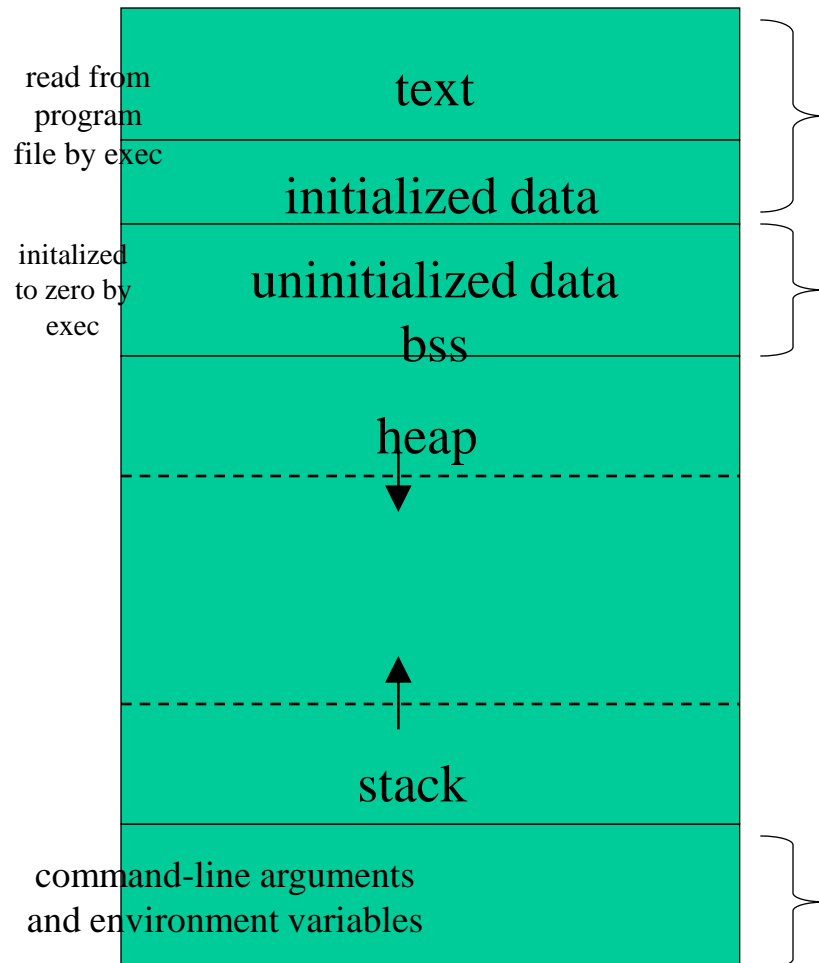
# 시스템 문맥

---

- 태스크를 관리하는 정보 집합: task struct
  - 태스크 정보: pid, uid, euid, suid, ...
  - 태스크 상태: 실행 상태, **READY** 상태, 수면 상태, ...
  - 태스크의 가족 관계: p\_pptr, p\_cptr, next\_task, next\_run
  - 스케줄링 정보: policy, priority, counter, rt\_priority, need\_resched
  - 태스크의 메모리 정보: 세그먼트, 페이지
  - 태스크가 접근한 파일 정보: file descriptor
  - 시그널 정보
  - 쓰레드 정보
  - 그 외: 수행 시간 정보, 수행 파일 이름, 등등 (kernel dependent)

# 메모리 관리

# Memory layout of a C program



- **Text segment**
  - 프로세서에 의해서 실행되는 기계어 코드가 위치
  - 공유 가능한 세그먼트
  - 읽기 전용으로 되어있음
- **Initialized data segment**
  - 초기화된 데이터 위치
  - `int maxcount = 99;`
- **uninitialized data segment**
  - 초기화되지 않은 데이터 위치
  - **bss** (**b**lock **s**tarted by **s**ymbol)
  - `long sum[1000];`
- **Heap**
  - 동적인 메모리 할당 장소
- **Stack**
  - 자동 변수 및 임시 변수 저장
  - 함수 호출시 복귀 주소 및 caller의 환경저장

# 시스템 프로그래밍 구조-

## 메모리관리(1/2)

- 드라이버, 파일시스템, 네트워크스택과 같은 커널 서브 시스템을 위한 동적 메모리를 제공
- 사용자 응용 프로그램을 위해 가상 메모리 지원
- 리눅스에서는 OS 와 응용프로그램은 각각 별도로 컴파일한다.
- 반면에 RTOS에서는 OS와 응용프로그램을 묶어서 컴파일한다
- 리눅스에서는 메모리를 페이지 단위로 나누어서 관리한다. 페이지의 대표적인 크기는 4KB 임

```
HostPC> du -h //디스크 용량을 블록이 아닌 Mbyte 나 Gbyte 단위로 보여줌
```

```
HostPC> df -h // 디스크의 남은 공간을 보여줌
```

```
HostPC> mount -t iso9660 /dev/cdrom /mnt/cdrom // CDRROM을 리눅스에서 사용
```

# 시스템 프로그래밍 구조-

## 메모리관리(2/2)

- 메모리 관리자 작업들

- 가상 메모리 제공 : 실제 물리적 메모리 보다 많은 용량의 가상 메모리를 제공
- 메모리 매핑 : 이미지와 데이터 파일을 프로세스의 주소공간에 매핑하기 위해서 사용된다. 메모리 매핑에서 파일의 내용은 프로세스의 가상 주소 공간에 직접 연결
- 공유 메모리 : 가상 메모리를 공유하는 기능을 제공
- 스왑 메모리 (swap memory) : 페이지 단위로 현재 실행되지 않는 프로세스 영역을 하드 디스크에 저장하는 작업을 의미.

```
HostPC> vmstat [option] [주기] [횟수]
procs -----memory----- ---swap--- ----io---- --system-- -----cpu-----

HostPC> vmstat -s // 현재까지의 모든 시스템 이벤트
HostPC> vmstat -3 // 3초 주기로 보여줌

HostPC> iostat [option] // CPU 및 디스크 IO 통계정보 표시
// -c: user mode, system mode, -t: ch 당 터미널에서 사용된 char 수

HostPC> free -s 5 -t //메모리 총 사용량을 보여주고 5초 단위로 보여줌
```

# 리눅스 파일 시스템

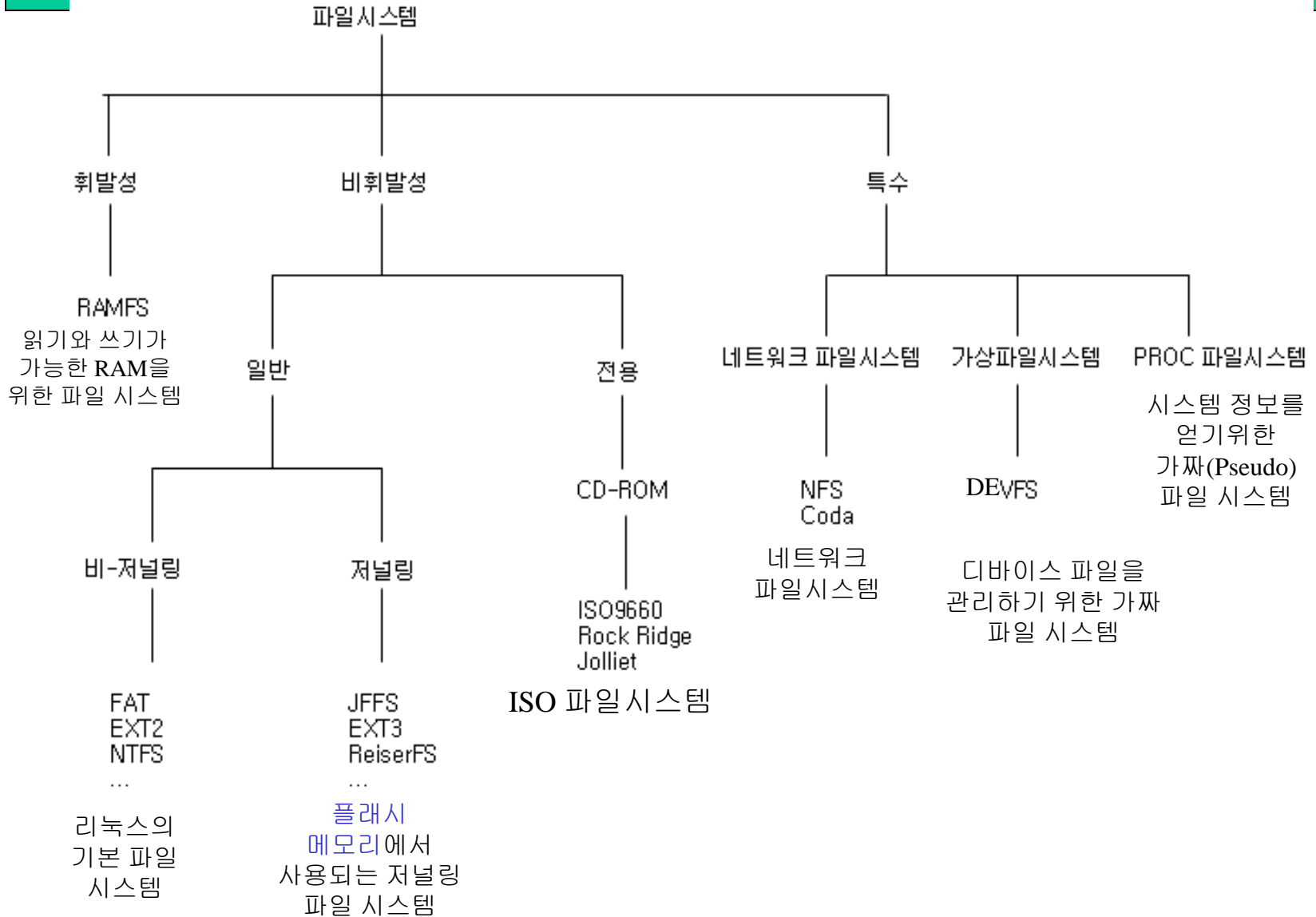
- File System 개념

- 개념: 운영체제가 저장 매체에 파일을 저장하는 방식
  - 저장매체 파티션: 저장 매체의 공간을 구획지어 독립된 가상 공간화
    - `Fdisk /dev/hda` //어떤 디스크의 파티션을 나눌지 생각 후 실행
  - 파일 시스템 구성
    - 파일 정보를 기록한 자료 구조의 생성 (inode 구조)
      - » Ex2, ex3, minix, CDRom, isofs, hpfs, NFS, sysv, ...
    - Mount: 파일 시스템 구조 내의 파일을 사용할 수 있도록 논리적인 파티션을 시스템에 연결함

- 임베디드 File System 개념

- Flash File System: 하드디스크와 유사
  - 가비지 컬렉션: 삭제 후 사용 가능
  - 닳기 균등화: 블록의 삭제회수에 의한 수명
  - 플래쉬 파일시스템: 파일시스템을 플래쉬메모리에 올리기 위한 중계
  - MTD(Memory Technology Device): 플래쉬메모리에 무관하게 동작하도록 지원

# File System 종류



# 시스템 프로그래밍 구조-

## 파일 시스템(3/3)

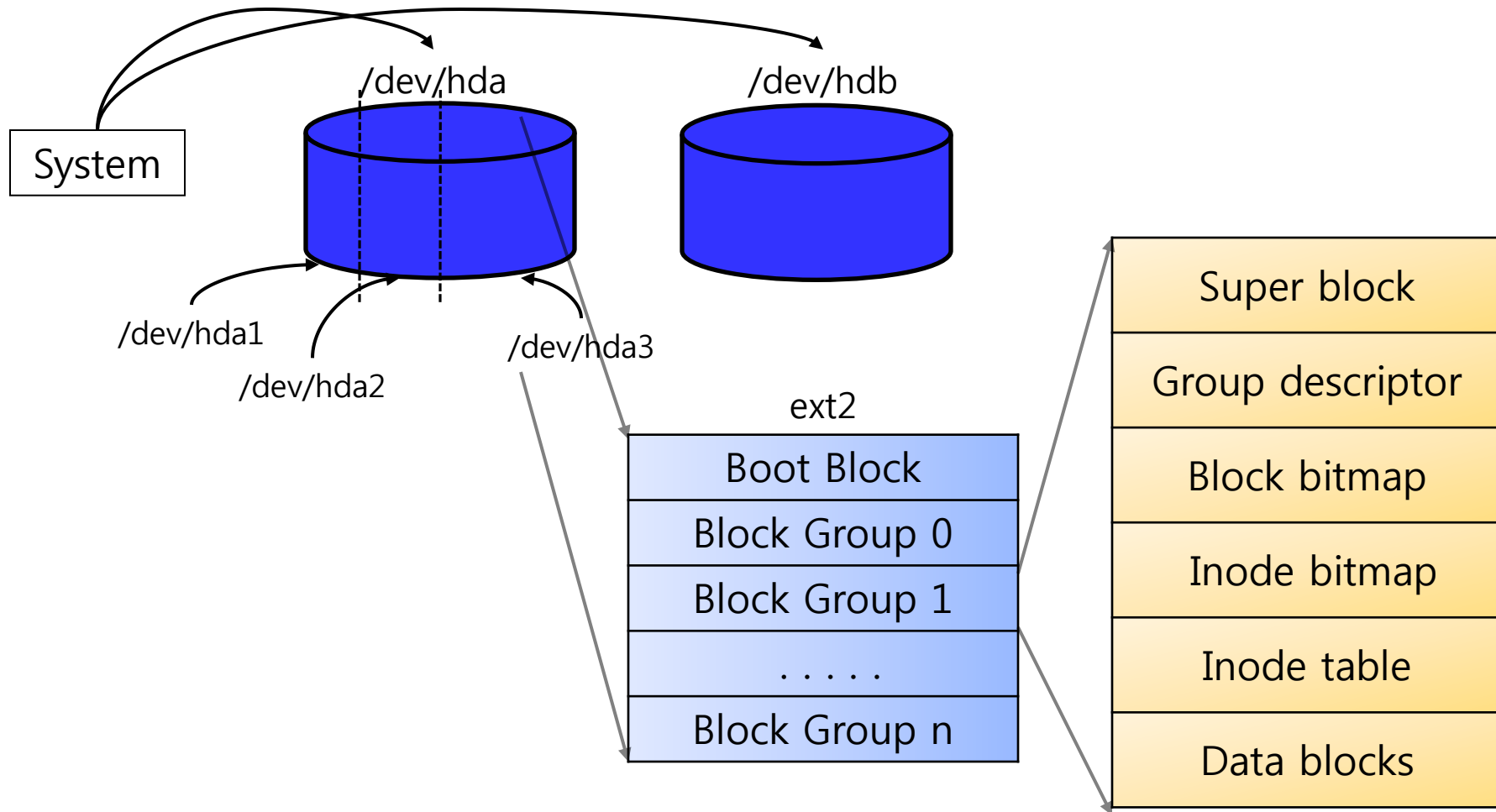
---

- EXT2 : 리눅스의 기본 파일 시스템
  - CRAMFS : 압축된 읽기 전용의 파일 시스템
  - ROMFS : 읽기 전용 파일 시스템
  - RAMFS : 읽기와 쓰기가 가능한 RAM을 위한 파일 시스템
  - JFFS2 : 플래시 메모리에서 사용되는 저널링 파일 시스템
  - PROCFS : 시스템 정보를 얻기위한 가짜(Pseudo) 파일 시스템
  - DEVFS : 디바이스 파일을 관리하기 위한 가짜 파일 시스템
- 
- 각 파일시스템은 성능, 공간활용, 또는 데이터 복구에 최적화 됨
  - 타겟 시스템은 요구 조건에 맞게 파일 시스템을 선택하여 사용함



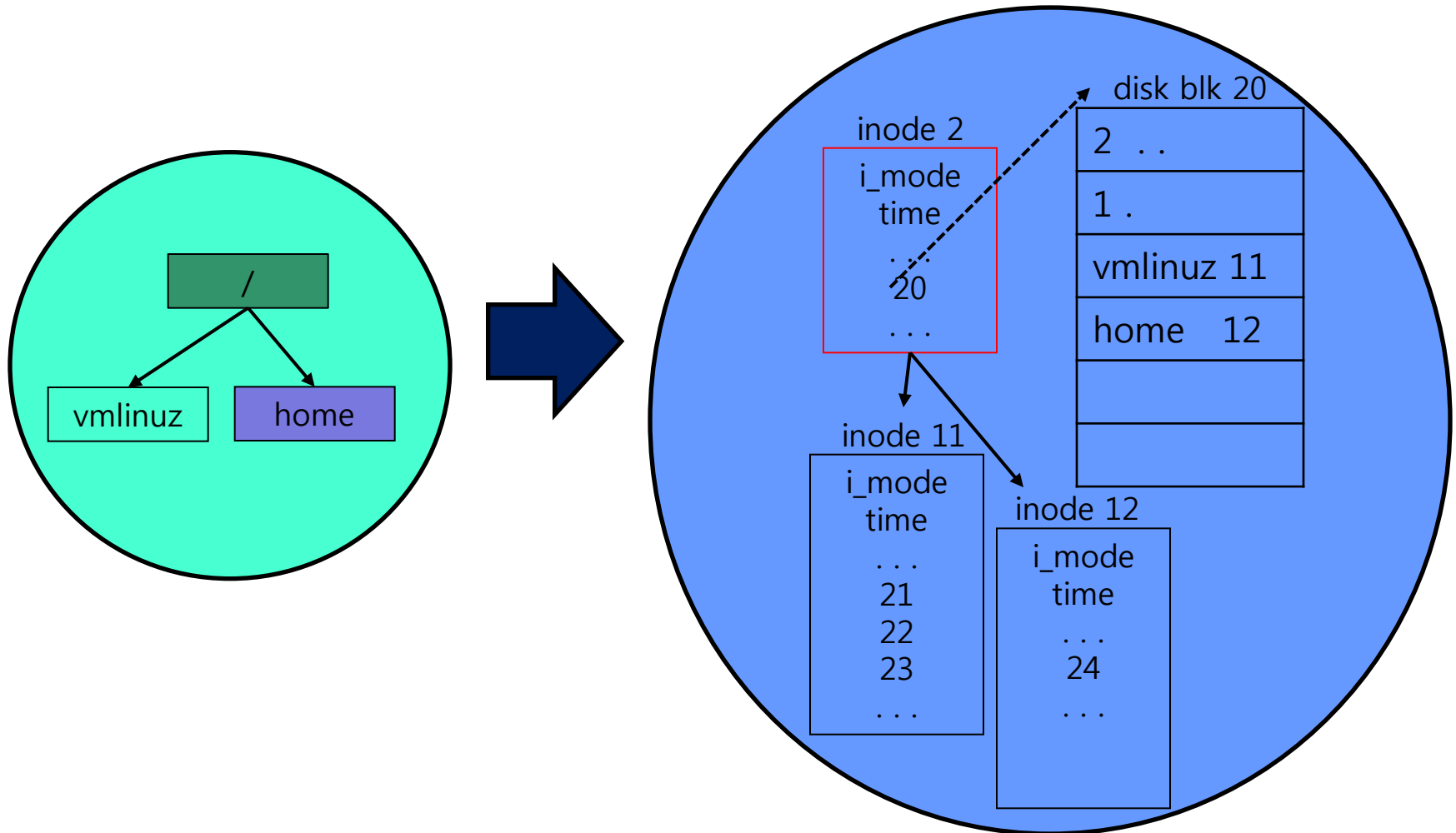
# ext2 파일 시스템 예

- 파티션과 파일 시스템



# ext2 파일 시스템

- ext2 의 inode 와 파일의 개념적 구조



# ext3 파일 시스템

---

- ext3 파일 시스템 특징
  - *~/include/linux/ext3\_fs.h*
  - 대부분의 자료 구조가 ext2 와 같도록 설계
  - hash 기반 HTree 구조 도입
    - 빠른 디렉토리 탐색을 위함 -> O(1) 시간에 접근 가능
    - 기존 ext2는 디렉토리를 linked list로 관리
  - 강력한 journaling 기능 지원
    - 별도의 공간을 두어 불시에 전원이 나가는 경우와 같은 결함을 허용
    - 지원하는 저널링 모드
      - Journal – fs에 데이터 기록전 저널에 기록, 추후에 fs에 저장
      - Ordered – meta 데이터만 저널에 기록, 사용자 데이터는 fs에 기록
      - Writeback – order와 유사, 쓰기 순서는 보장 안됨
  - Online-resizing
    - 파일 시스템이 관리하는 파티션의 용량을 동적으로 늘리거나 줄일 수 있음

# ext4 파일 시스템

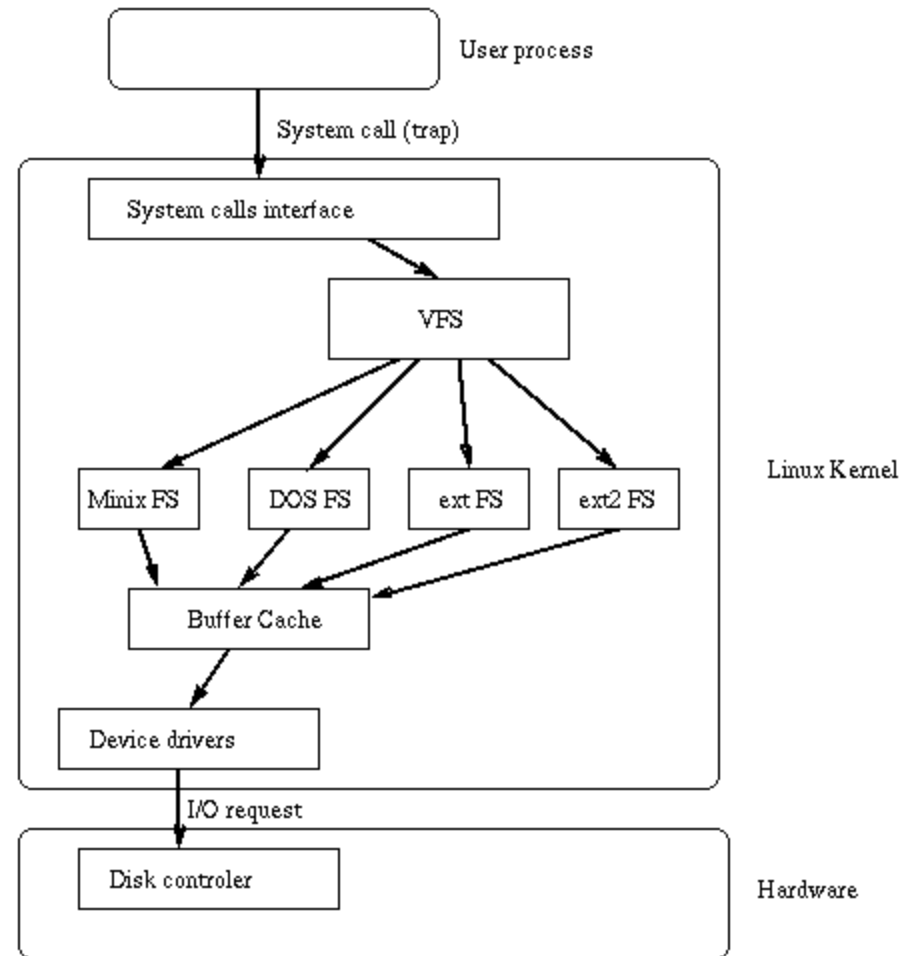
---

- ext4 파일 시스템 특징
  - *~/include/linux/ext4\_fs.h*
  - 선 할당 (preallocation), 지연 할당 (delayed-allocation)
    - 선 할당
      - 파일 시스템의 일관성, 속도 향상 위한 기술
      - 파일 생성 시, 미리 일정 개수의 블록할당 -> 물리적으로 연속인 블록 할당하여 성능 향상 시키려는 기법
    - 지연 할당
      - 단편화 방지
      - 자유 블록 카운트만 갱신, 실제 할당은 연기
  - Extend 기반 데이터 블록 유지
    - 대용량 파일의 메타 데이터 줄일 수 있음
  - 저널링 checksum, 대용량 파일 시스템 및 파일 지원, 온라인 단편화 제거 기능

# Summary - File System

- A typical file system
  - Open a file with authentication
  - Read/write data in files
  - Close a file
- 용도에 따라 다른 파일시스템을 사용하는 것이 효율적
  - FAT(File Allocation Table): MS-DOS와 윈도우즈 95
  - NTFS(NT File System): 윈도우즈 XP
  - **EXT2, EXT3: 리눅스**
  - ISO 파일시스템: CD-ROM
  - NFS: 네트워크
  - EXT3, **JFFS2: 저널링 파일시스템**
- 임베디드 시스템
  - 제한된 자원을 사용: 효율성 추구
  - 문제가 발생하면 자동으로 복구

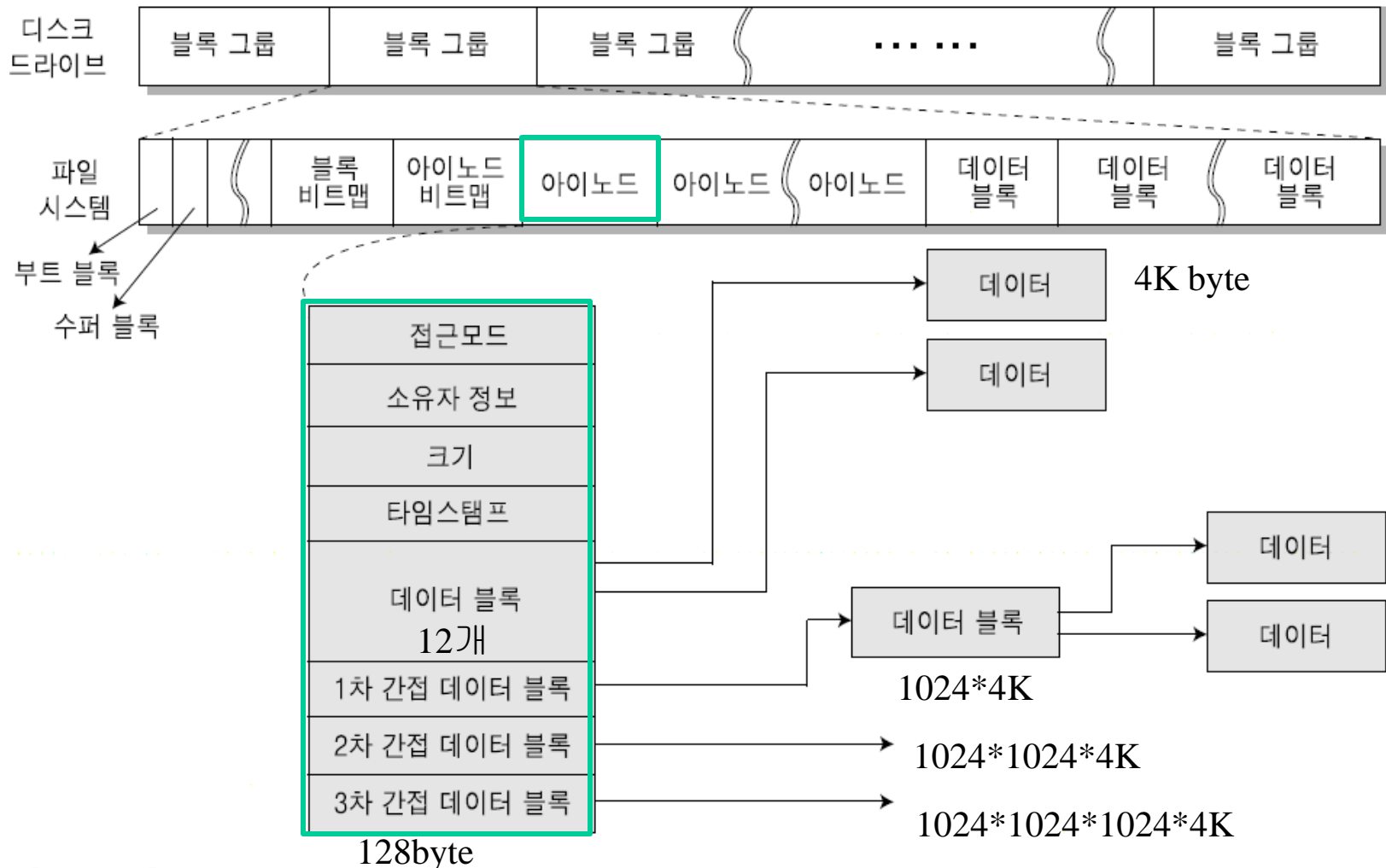
파일: 저장 매체에 보관된 데이터 집합의 추상적 개념



The diagram illustrates a hierarchical file system structure. At the top is the root directory, represented by a cylinder labeled **root fs**. Below the root is a large box labeled **filesystem1**. Inside this box, the root directory branches into several subdirectories: **/usr**, **/bin**, **/lib**, **/home**, **/etc** (highlighted in red), **/mnt**, and **/dev**. The **/usr** directory is linked to a cylinder labeled **/usr**. The **/bin** and **/lib** directories are linked to a box labeled **filesystem2**, which contains subdirectories **bin** and **lib**. The **/home** directory is linked to a box labeled **filesystem3**, which contains subdirectories **lib**, **src**, and **bin**. The **/etc** directory is linked to a box labeled **filesystem4**, which contains subdirectories **xxx** and **yyy**. The **/mnt** directory is linked to a box labeled **filesystem4**, which contains subdirectories **xxx** and **yyy**. The **/dev** directory is linked to a cylinder labeled **CD-ROM**. The **/lib** directory is labeled "가장 필수적인 공유 lib 실행파일" (Most essential shared library executable files). The **/etc** directory is labeled "Host specific conf. files, no program" (Host specific configuration files, no program). The **/mnt** directory is labeled "장치 관련 파일" (Device related files). The **/usr** directory is labeled "mount point". The **/dev** directory is labeled "mount point". The **/lib** directory is labeled "mount point". The **/etc** directory is labeled "mount point". The **/mnt** directory is labeled "mount point". The **/dev** directory is labeled "mount point".

## Mount : file system과 저장 장치와 연결

## 리눅스 파일시스템의 데이터 구조체



[그림 7-4] 파일시스템과 아이노드의 관계

---

- 아이노드 (inode)

- 리눅스 파일시스템의 가장 기초가 되는 데이터 구조체
- 파일에 관한 필수 정보를 포함
- index node의 약어
- 각 파일마다 하나씩 존재
- 파일에 대한 제어 정보와 데이터 블록 포인터를 포함
  - 파일 소유자 이름
  - 파일 접근 허가권
  - 파일 크기
  - 파일의 마지막 접근 시간
  - 파일의 생성 시간
  - 디스크의 블록 위치 등



# Inode 구조

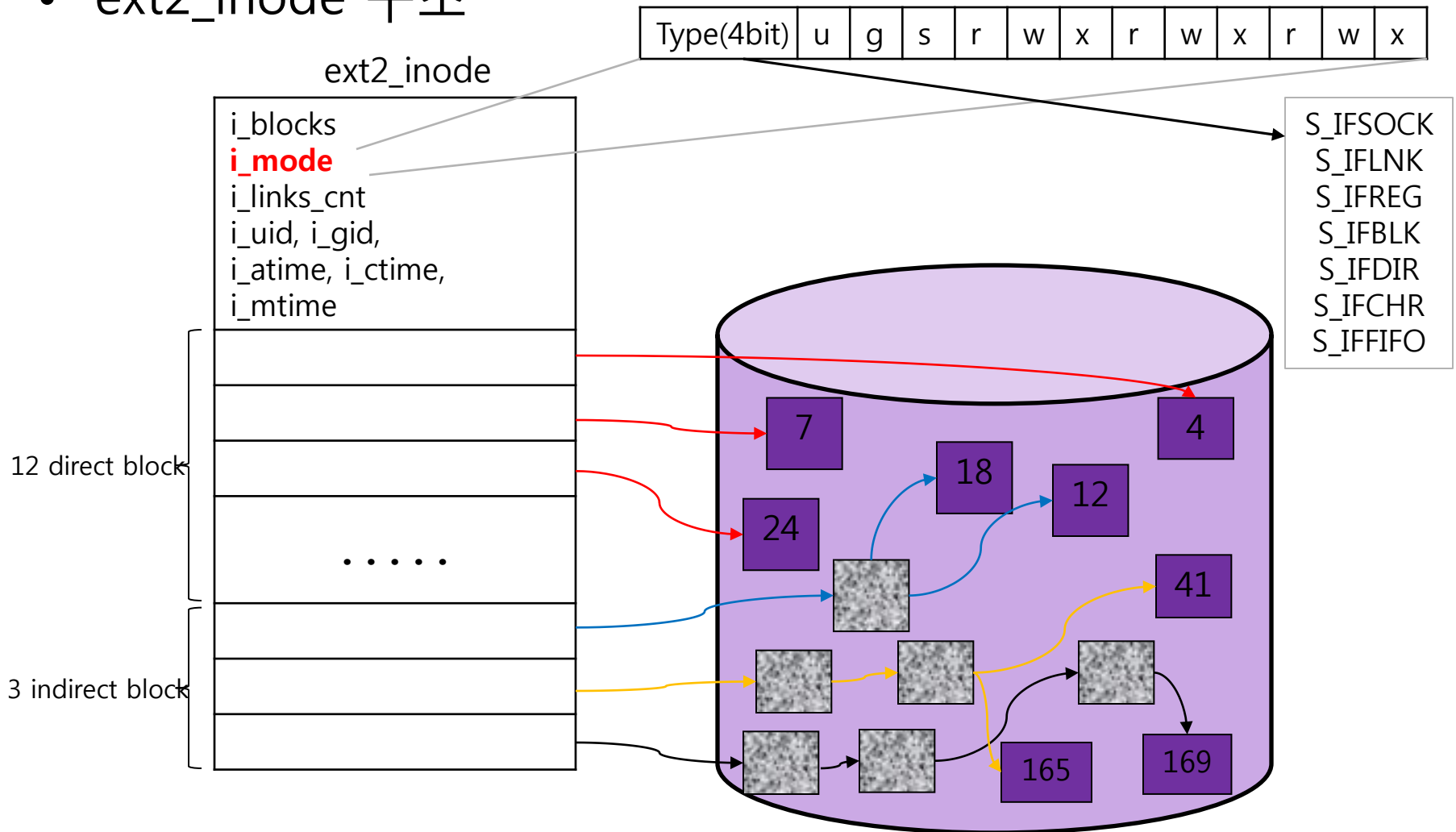
- Inode

- 리눅스 기본 fs인 ext2, ext3 가 채택하고 있는 구조
- *~/include/linux/ext2\_fs.h*

```
struct ext2_inode {
    __le16    i_mode; /* file mode */
    __le16    i_uid;  /* low 16 bits of owner Uid */
    __le32    i_size; /* Size in Bytes */
    __le32    i_atime, i_ctime; /* Access, Creation Time */
    __le32    i_mtime, i_dtime; /* Modification, Deletion Time */
    __le16    i_gid;  /* low 16bits of group Id */
    __le16    i_links_count; /* Links count */
    __le32    i_blocks, i_flags; /* Block count, File flags */
    .         .         .
    .         .         .
}
```

# Inode 구조

- ext2\_inode 구조



# Inode 구조

---

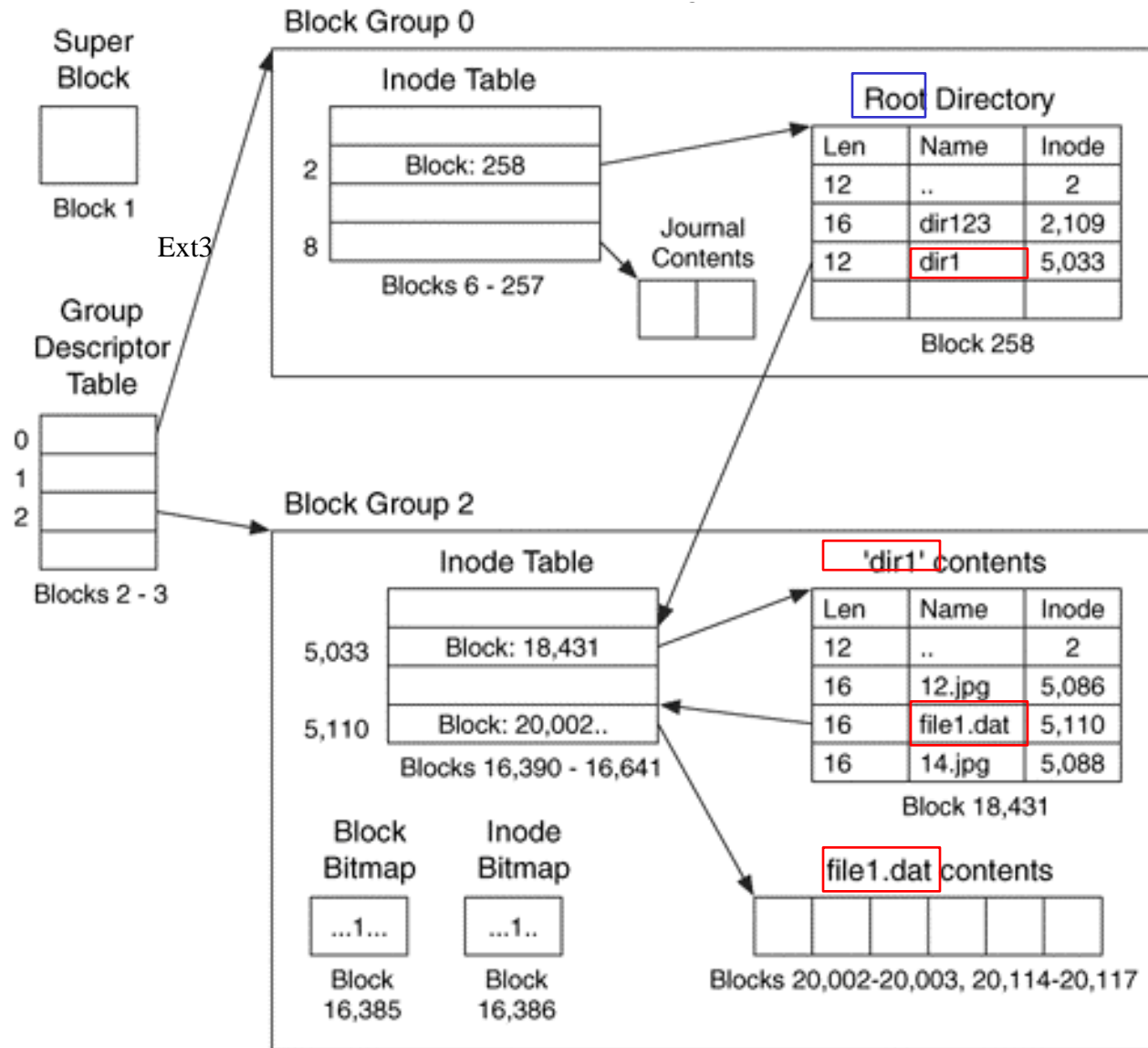
- ext2 파일시스템의 디렉터리 엔트리
  - *~/include/linux/ext2\_fs.h*

```
struct ext2_dir_entry {  
    __le32    inode; /* Inode number */  
    __le16    rec_len; /* Directory entry length */  
    __le16    name_len; /* Name length */  
    char      name [255]; /* File name */  
}
```

- msdos 파일시스템과의 차이
  - 이름과 inode 번호 같은 간단한 정보만을 유지
  - 실제 데이터 블록 인덱싱은 inode 자료구조에 유지

# Example

- Suppose we create a *file1.dat* under */dir/* in



# 시스템 프로그래밍 구조- 파일 시스템

## 요약(1/3)

- 리눅스에서는 여러 파일 시스템이 VFS (Virtual File system) 라는 layer에서 관리된다
- 리눅스 기기에서는 최소한 파일 시스템이 필요하다. RTOS에서는 그렇지 않다.
- 리눅스에서 파일 시스템이 필요한 이유
  - 응용프로그램이 OS와 별도로 컴파일되므로 별도 저장공간이 필요하다
  - 모든 저수준 기기들도 파일처럼 접근하여 사용가능
- 리눅스에서는 디스크 기반 파일 시스템 이외에 플래시 또는 ROM 기반 파일 시스템을 지원한다. 네트워크 파일 시스템 (NFS)도 지원한다
- 파일시스템은 OS가 파일을 시스템 디스크상에 구성하는 방식이다. 디스크 파티션상에 파일이 연속적이고 일정한 규칙을 가지고 저장된다.

# 시스템 프로그래밍 구조- 파일 시스템

## 요약(2/3)

---

- 리눅스에서 파일 시스템 저장 규칙
  - 슈퍼블록 : 파일 시스템의 전체적인 정보(크기, 마운트 회수, 블록 그룹 넘버, 블록 크기, 첫 번째 아이노드 등)를 포함하고 있다.
  - 아이노드(inode) : 리눅스 파일 시스템 데이터 구조체, 파일에 관한 필수 정보(데이터 블록의 위치 정보, 파일 형태, 파일 소유자, 아이노드, 파일 이름 제외)를 포함하며 **각 파일 이름에 부여되는** 고유 번호이다. 아이노드 테이블에서 아이노드 번호를 가지고 검색하여 알려준다.
  - 데이터 블록 : 파일에서 데이터를 저장하기 위해서 사용되는 공간이다. 이러한 데이터 블록은 아이노드에 저장되며, 하나의 아이노드는 보통 다수의 데이터 블록을 가지고 있다.
  - 디렉토리 블록 : 파일 이름과 아이노드 번호를 저장한다.
  - 간접 블록 : 아이노드에 데이터 블록의 위치 정보를 저장할 공간이 부족한 경우, 위치 정보를 저장하기 위한 공간을 동적 할당하게 된다. 이 때 할당된 공간을 간접 블록이라고 한다.

# 파일 시스템 관련 실습

---

- 루프 파일 시스템 내에 새로운 루프 파일 시스템 만들기
- `$ dd if=/dev/zero of=/mnt/point1/file.img bs=1k count=1000`  
1000+0 records in  
1000+0 records out //파일 이미지 생성
- `$ losetup /dev/loop0 /mnt/point1/file.img //loop0 저장 장치와 연결`
- `$ mke2fs -c /dev/loop0 1000 //주어진 크기의 새로운 ext2 파일시스템 생성`  
mke2fs 1.35 (28-Feb-2004)  
max\_blocks 1024000, rsv\_groups = 125, rsv\_gdb = 3  
Filesystem label= ...
- `$ mkdir /mnt/point2`
- `$ mount -t ext2 /dev/loop0 /mnt/point2 //파일시스템을 저장 장치와 연결`
- `$ ls /mnt/point2`  
lost+found
- `$ ls /mnt/point1`  
file.img lost+found

# jffs2 파일 시스템 실습

## - RAM 디스크 생성 실습

```
~$ dd if=/dev/zero of=./ramdisk.fs count=4096 bs=1024
```

// 1024byte의 한블록을 4096번 반복하여 4Mbyte의 램디스크 생성

```
~$ mkfs -t ext2 ramdisk.fs //dd 명령어로 생성된 램디스크에 파일 시스템 생성
```

## - jffs2:NOR형 플래쉬메모리를 사용하기 위한 파일 시스템

- 디렉토리에 필요한 파일 복사
- Jffs2 file system,으로 img 만듦

```
~$ cat > ./jffs/test.txt // 메시지 작성 및 확인
```

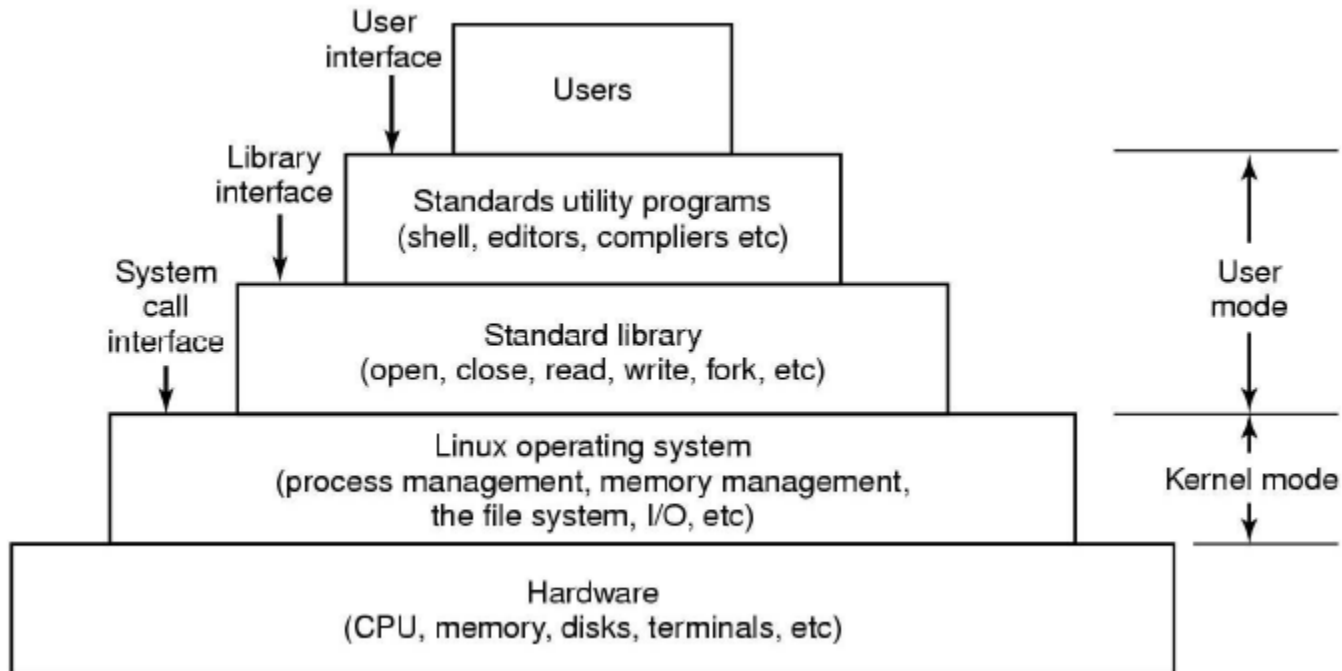
```
~$ mkfs.jffs2 -r jffs/ -o jffs2.img -e 0x40000 -p
```

```
//
```



# Interfaces to Linux

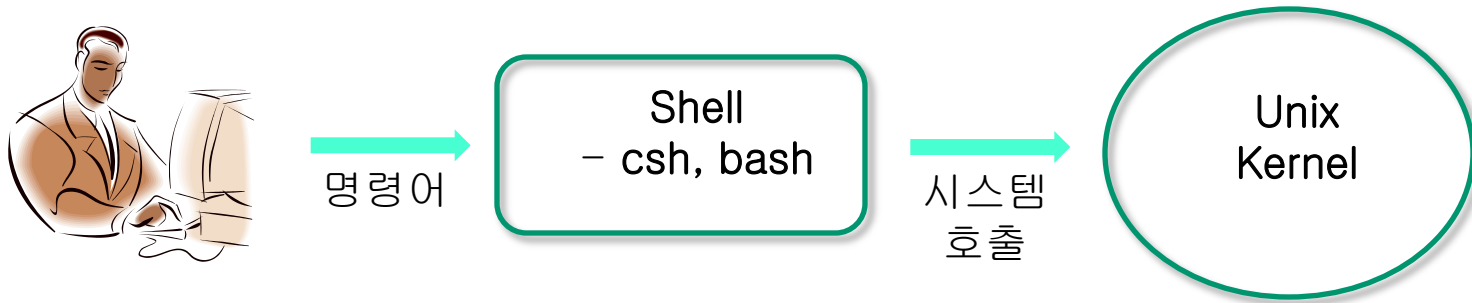
- Layers in a Linux system



# 리눅스 기본 명령어

# 리눅스 기본 명령 (1)

- 셸(Shell)
  - 명령어 해석기(Command Interpreter)
    - 리눅스 운영체제와 사용자 사이의 인터페이스를 담당



- 스크립트(Shell script) 처리 가능
  - 다양한 셸 연산(Shell Operations) 지원
  - 리다이렉션(redirection), 파이프라인(pipeline) 등
- 환경변수(Environment Variables) 지원
  - 프로그램 실행 환경 설정, 프로그램간의 간단한 통신 지원
  - PATH, HOME, CC, CFLAGS 등

# 리눅스 기본 명령 (2)

- 로그인/로그아웃

명령	기능	주요 옵션	예제
telnet	리눅스시스템에 접속	-	telnet hanb.co.kr
logout	리눅스시스템에서 접속해제	-	logout
exit		-	exit

- 프로세스 관련 명령

명령	기능	주요 옵션	예제
ps	현재 실행 중인 프로세스의 정보를 출력	-ef : 모든 프로세스에 대한 상세 정보 출력	ps ps -ef ps -ef   grep ftp
kill	프로세스 강제 종료	-9 : 강제 종료	kill 5000 kill -9 5001

# 리눅스 기본 명령 (3)

## • 파일/디렉토리 조작 명령

명령	기능	주요 옵션	예제
pwd	현재 디렉토리 경로 출력	-	pwd
ls	디렉토리 내용 출력	-a : 숨김파일출력 -l : 파일 상세정보 출력	ls -a /tmp ls -l
cd	현재 디렉토리 변경	-	cd /tmp cd ~han01
cp	파일/디렉토리 복사	-r : 디렉토리 복사	cp a.txt b.txt cp -r dir1 dir2
mv	파일/디렉토리 이름변경과 이동	-	mv a.txt b.txt mv a.txt dir1 mv dir1 dir2
rm	파일/디렉토리 삭제	-r : 디렉토리 삭제	rm a.txt rm -r dir1
mkdir	디렉토리 생성	-	mkdir dir1
rmdir	빈 디렉토리 삭제	-	mkdir dir2
cat	파일 내용 출력	-	cat a.txt
more	파일 내용을 쪽단위로 출력	-	more a.txt
chmod	파일 접근권한 변경	-	chmod 755 a.exe chmod go+x a.exe
grep	패턴 검색	-	grep abcd a.txt

# 리눅스 기본 명령 (4)

- 기타 명령

명령	기능	주요 옵션	예제
su	사용자 계정 변경	- : 변경할 사용자의 환경 초기화 파일 실행	su su - su - han02
tar	파일/디렉토리 묶기	-cvf : tar파일생성 -tvf : tar파일내용보기 -xvf : tar파일풀기	tar cvf a.tar * tar tvf a.tar tar xvf a.tar
whereis	파일 위치 검색	-	whereis ls
which		-	which telnet

# 리눅스 기본 명령 (5)

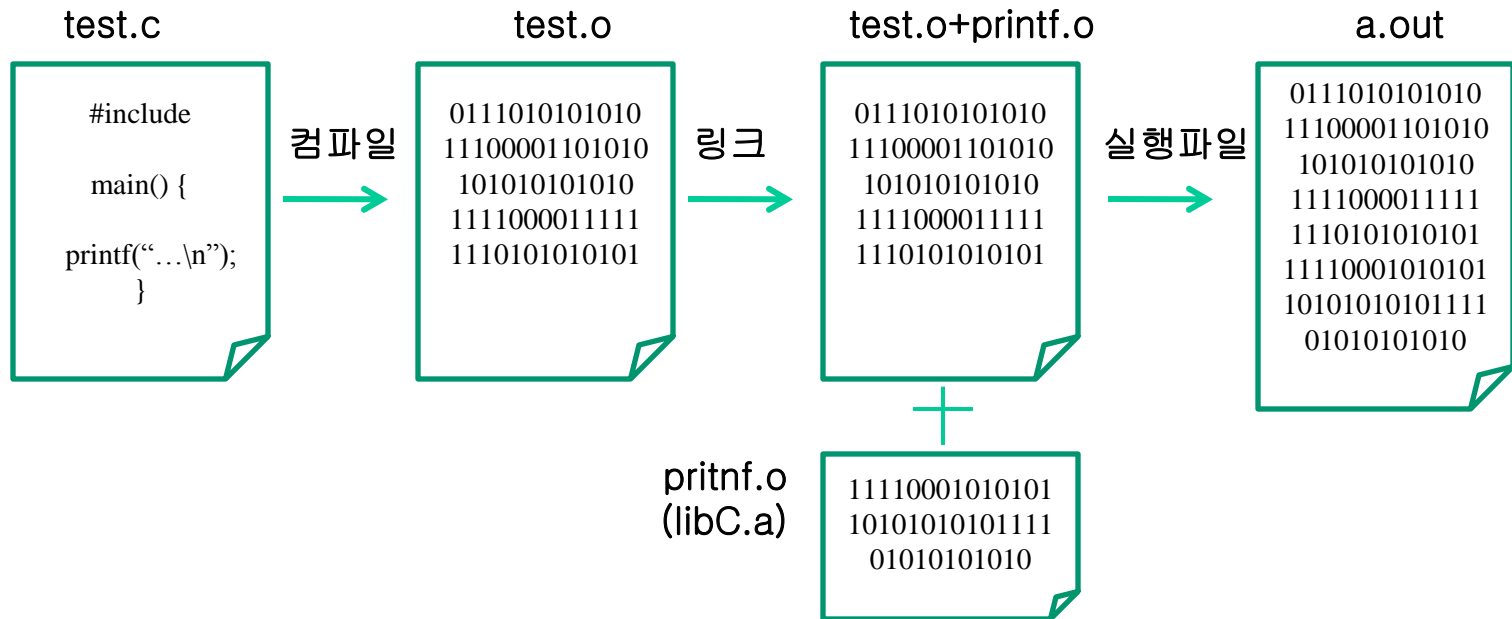
## • vi 편집기 내부 명령

기능	명령	기능	명령
입력모드전환	i,a,o,O	명령모드전환	<Esc>
커서이동	j,k,h,l 또는 방향키	행이동	#G (50G, 143G 등) 또는 :행번호
한글자수정	r	여러글자수정	#s (5s, 7s 등)
단어수정	cw	명령취소	u, U
검색하여수정	:%s/aaa/bbb/g	복사	#yy (5yy, 10yy 등)
붙이기	p	커서이후삭제	D(shift-d)
글자삭제	x, #x(3x,5x 등)	행삭제 (잘라내기)	dd, #dd(3dd, 4dd 등)
저장하고종료	:wq! 또는 ZZ	저장않고종료	:q!
행 붙이기	J(shift-j)	화면다시표시	ctrl+l
행번호보이기	:set nu	행번호없애기	:set nonu

# C언어 컴파일 환경 (1)

- 컴파일러(Compiler)

- 텍스트로 작성한 프로그램을 시스템이 이해할 수 있는 기계어로 변환
- 보통 컴파일 과정과 라이브러리 링크 과정을 묶어서 수행





# C언어 컴파일 환경 (2)

- GNU C 컴파일러 : gcc
  - 대부분 GNU C 컴파일러 사용([www.sunfreeware.com](http://www.sunfreeware.com))
  - /usr/bin 또는 /usr/local/bin 디렉토리에 설치됨

```
# vi ~/.profile
.....
PATH=$PATH:/usr/bin
export PATH
```



```
# . ~/.profile
```

바뀐 .profile 적용

- C컴파일러 사용

```
# gcc test.c
# ls
a.out test.c
```

기본 실행파일명은  
a.out

```
# gcc -o test test.c
# ls
test    test.c
```

실행파일명 지정은  
-o 옵션

# C언어 컴파일 환경 (3)

- make / Makefile

- 소스 파일이 여러 개를 묶어서 실행파일을 생성하는 도구
- make 명령은 Makefile의 내용에 따라 컴파일과정을 수행

```
# vi ~/.profile
.....
PATH=$PATH:/usr/bin
export PATH
```

[예제 1-3]

ex1\_3\_main.c

```
01 #include <stdio.h>
02 extern int addnum(int a, int b);
03
04 int main(void) {
05     int sum;
06
07     sum = addnum(1, 5);
08     printf("Sum 1~5 = %d\n",
sum);
09
10     return 0;
11 }
```

[예제 1 ex1\_3\_addnum.c

```
01 int addnum(int a, int b) {
02     int sum = 0;
03
04     for (; a <= b; a++)
05         sum += a;
06     return sum;
07 }
```

# C언어 컴파일 환경 (4)

## [예제 1-3] make 명령 사용하기

## Makefile

```
01 # Makefile
02
03 CC=gcc
04 CFLAGS=
05 OBJS=ex1_3_main.o ex1_3_addnum.o
06 LIBS=
07 all: add
08
09 add: $(OBJS)
10     $(CC) $(CFLAGS) -o add $(OBJS) $(LIBS)
11
12 ex1_3_main.o: ex1_3_main.c
13     $(CC) $(CFLAGS) -c ex1_3_main.c
14 ex1_3_addnum.o: ex1_3_addnum.c
15     $(CC) $(CFLAGS) -c ex1_3_addnum.c
16
17 clean:
18     rm -f $(OBJS) add core
```

ex1\_3\_main.c와  
ex1\_3\_addnum.c를  
묶어서 add라는  
실행파일 생성

```
# make
gcc -c ex1_3_main.c
gcc -c ex1_3_addnum.c
gcc -o add ex1_3_main.o
ex1_3_addnum.o

# ls
Makefile  add*  ex1_3_addnum.c
ex1_3_addnum.o  ex1_3_main.c
ex1_3_main.o

# add
Sum 1~5 = 15
```

# 오류 처리 함수 (1)

- 오류 메시지 출력 : `perror(3)`

```
#include <stdio.h>
void perror(const char *s);
```

## [예제 1-4] perror 함수 사용하기

ex1\_4.c

```
01 #include <sys/errno.h>
02 #include <unistd.h>
03 #include <stdlib.h>
04 #include <stdio.h>
05
06 int main(void) {
07     if (access("unix.txt", R_OK) == -1) {
08         perror("unix.txt");
09         exit(1);
10     }
11
12     return 0;
13 }
```

오류에 따라  
메시지 자동 출력

```
# ex1_4.out
unix.txt: No such file or directory
```

# 오류 처리 함수 (2)

- 오류 메시지 출력 : `strerror(3)`

```
#include <string.h>
char *strerror(int errnum);
```

[예제 1-5] `strerror` 함수 사용하기

ex1\_5.c

```
01 #include <sys/errno.h>
02 #include <unistd.h>
03 #include <stdlib.h>
04 #include <stdio.h>
05 #include <string.h>
06
07 extern int errno;
08
09 int main(void) {
10     char *err;
11
12     if (access("unix.txt", R_OK) == 1) {
13         err = strerror(errno);
14         printf("오류:%s(unix.txt)\n", err);
15         exit(1);
16     }
17
18     return 0;
19 }
```

오류에 따라  
메시지를 리턴

```
# ex1_5.out
오류: No such file or directory(unix.txt)
```

# 동적 메모리 할당 (1)

- 메모리 할당 : malloc(3)

```
#include <stdlib.h>
void *malloc(size_t size);
```

- 인자로 지정한 크기의 메모리 할당

```
char *ptr
ptr = malloc(sizeof(char) * 100);
```

- 메모리 할당과 초기화 : calloc(3)

```
#include <stdlib.h>
void *calloc(size_t nelem, size_t elsize);
```

nelem - elsize 만큼의 메모리를 할당하고, 0으로 초기화

```
char *ptr
ptr = calloc(10, 20);
```

# 동적 메모리 할당 (2)

- 메모리 추가 할당: realloc(3)

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

- 이미 할당받은 메모리(ptr)에 size 크기의 메모리를 추가로 할당

```
char *ptr, *new;
ptr = malloc(sizeof(char) * 100);
new = realloc(ptr, 100);
```

- 메모리 해제 : free(3)

```
#include <stdlib.h>
void free(void *ptr);
```

- 사용을 마친 메모리 반납

# 명령행 인자 (1)

- 명령행 : 사용자가 명령을 입력하는 행
  - 명령행 인자 : 명령을 입력할 때 함께 지정한 인자(옵션, 옵션인자, 명령인자 등)
  - 명령행 인자의 전달 : main 함수로 자동 전달

```
int main(int argc, char *argv[])
```

[예제 1-6] 명령행 인자 출력하기

ex1\_6.c

```
01 #include <stdio.h>
02
03 int main(int argc, char *argv[]) {
04     int n;
05
06     printf("argc = %d\n", argc);
07     for (n = 0; n < argc; n++)
08         printf("argv[%d] = %s\n", n, argv[n]);
09
10     return 0;
11 }
```

```
# ex1_6.out -h 100
argc = 3
argv[0] = ex1_6.out
argv[1] = -h
argv[2] = 100
```



# 명령행 인자 (2)

- 옵션 처리 함수: getopt(3)

```
#include <stdio.h>
int getopt(int argc, char * const argv[], const char *optstring);
extern char *optarg;
extern int optind, opterr, optopt;
```

- argc, argv : main 함수에서 받은 것을 그대로 지정
- optstring : 사용할 수 있는 옵션 문자, 옵션에 인자가 있을 경우 문자 뒤에 ‘:’ 추가
- optarg : 옵션의 인자 저장
- optind : 다음에 처리할 argv의 주소
- optopt : 오류를 발생시킨 문자
- opterr : 오류 메시지를 출력하지 않으려면 0으로 지정

# 명령행 인자 (3)

---

- 리눅스 명령의 옵션 규칙
  - 옵션의 이름은 한 글자여야 하며, 모든 옵션의 앞에는 하이픈(-)이 있어야 한다.
  - 인자가 없는 옵션은 하나의 ' - ' 다음에 묶여서 올 수 있다(-xvf)
  - 옵션의 첫 번째 인자는 공백이나 탭으로 띄고 입력한다.
  - 인자가 있어야 하는 옵션에서 인자를 생략할 수 없다.
  - 명령행에서 모든 옵션은 명령의 인자보다 선행하여야 한다.
  - 옵션의 끝을 나타내기 위해 --를 사용할 수 있다.

# 명령행 인자 (3)

- 옵션 처리

[예제 1-7] getopt 함수 사용하기

ex1\_7.c

```
01 #include <stdio.h>
02
03 int main(int argc, char *argv[]) {
04     int n;
05     extern char *optarg;
06     extern int optind;
07
08     printf("Current Optind : %d\n", optind);
09     while ((n = getopt(argc, argv, "abc:")) != -1) {
10         switch (n) {
11             case 'a':
12                 printf("Option : a\n");
13                 break;
14             case 'b':
15                 printf("Option : b\n");
16                 break;
17             case 'c':
18                 printf("Option : c, Argument=%s\n",
19                     optarg);
20                 break;
21             }
22         printf("Next Optind : %d\n", optind);
23     }
24     return 0;
25 }
```

```
# ex1_7.out
Current Optind : 1
# ex1_7.out -a
Current Optind : 1
Option : a
Next Optind : 2
# ex1_7.out -c
Current Optind : 1
ex1_7.out: option requires an
argument -- c
Next Optind : 2
# ex1_7.out -c name
Current Optind : 1
Option : c, Argument=name
Next Optind : 3
# ex1_7.out -x
Current Optind : 1
ex1_7.out: illegal option -- x
Next Optind : 2
```

---

- **프로그래밍 표준**

- 리눅스 시스템 프로그래밍과 관련 표준으로는 ANSI C, IEEE의 POSIX, X/Open그룹의 XPG3, XPG4, SVID, SUS가 있다.

- **리눅스 시스템 프로그래밍**

- 시스템 호출 : 리눅스 시스템이 제공하는 다양한 서비스를 이용하여 프로그램을 구현할 수 있도록 제공되는 프로그래밍 인터페이스
- 리눅스 시스템 프로그래밍 : 이러한 시스템 호출을 사용하여 프로그램을 작성하는 것

- **시스템 호출과 라이브러리 함수**

- 시스템 호출은 기본적인 형식은 C언어의 함수 형태로 제공된다.
- 시스템 호출은 직접 커널의 해당 모듈을 호출하여 작업을 하고 결과를 리턴한다.
- 라이브러리 함수들은 커널 모듈을 직접 호출하지 않는다.
- 라이브러리 함수가 커널의 서비스를 이용해야 할 경우에는 시스템 호출을 사용한다.
- `man` 명령을 사용할 때 시스템 호출은 섹션 2에 있고, 라이브러리 함수들은 섹션 3에 배치된다. 따라서 '`man -s 2 시스템 호출명`'과 같이 사용한다.
- 시스템 호출은 오류 발생시 -1을 반환하고 라이브러리 함수는 NULL 값을 반환한다.

---

- 리눅스 시스템 프로그래밍 도구

- 기본 명령 : pwd, ls, cd, mkdir, cp, mv, rm, ps, kill, tar, vi
- 컴파일 : GNC C 컴파일러(gcc)
- 컴파일 도구 : make명령과 Makefile
- 오류처리함수 : perror, strerror
- 동적메모리 할당 함수 : malloc, calloc, realloc, free
- 명령행인자 처리 함수 : getopt

Programmatic interface  
System call function

# mount 파일 시스템(2/3)

---

- mount: 논리적인 파티션을 시스템에 연결함  
system 내 일련의 파일들을 사용할 수 있도록 인식시킴
- ~\$ mount -t [파티션타입] [장치명] [마운트위치]

```
~$ mount -t iso9660 /dev/cdrom /mnt/cdrom //마운트 연결
```

```
~$ umount /dev/cdrom // 마운트해제
```

```
~$ mount -t nfs -o nolock 192.168.10.98:/home/jwjw /mnt/nfs
```

```
~$ umount /mnt/nfs
```

# File system 관련 실습

- # df: 디스크가 남은 공간을 확인하여 보여준다.

```
# df -k | head -6 //마운트된 파일시스템 보여줌
```

Filesystem	1024-blocks	Free	Used	Iused	%Iused	Mounted on
/dev/hd4 229376	138436	40%	4730	13%		/
/dev/hd2 8028160	962692	89%	110034	33%		/usr
/dev/hd9var 1835008	366400	81%	25829	24%		/var
/dev/hd3 524288	523564	1%	98	1%		/tmp
/dev/hd1 32768	32416	2%	5	1%		/home


- # ls -li

```
# ls -li /usr // Inode 5120 is the actual entry on disk.
```

1409	X11R6	314258	i686-linux	43090	libexec	13394	sbin
1417	bin	1513	i686-pc-linux-gnu	5120	local	13408	share
8316	distfiles	1517	include	776	man	23779	src
43	Doc	1386	info	93892	portage	36737	ssl
70744	gentoo-x86	1585	lib	5132	portage.old	784	tmp




# Proc 관련 실습

 **ps** Get the status of one or more processes (시스템 상에 수행 중인 프로세스 목록을 보여 줌.) Eg: ps ax // Eg : ps -ef | grep <process>

# ps // PID-process ID 보여줌

PID	USER	VSZ	STAT	COMMAND
1	root	1380	S	init
2	root	0	SW	[kthreadd]
3	root	0	SW	[ksoftirqd/0]

 **top** The top program provides a dynamic real-time view of a running system. It can display system summary information as well as a list of tasks currently being managed by the Linux kernel

# top // PID-process ID 보여줌 , PPID-parent process ID ;

Mem: 18552K used, 43364K free, 0K shrd, 1716K buff, 5832K cached

CPU: 0% usr 0% sys 0% nic 98% idle 0% io 0% irq 0% sirq

Load average: 0.00 0.00 0.00 1/35 13664

PID	PPID	USER	STAT	VSZ	%MEM	%CPU	COMMAND
854	1	root	S	1472	2%	1%	hostapd -P /var/run/wifi-phy0.pid -B
13664	13627	root	R	1376	2%	0%	top
1214	1212	root	S	1976	3%	0%	lldpd -d -c -f -s -e -L 2:FR:6:Commer

# Proc 관련 실습2

 **Cat /proc/cpuinfo** (시스템 상에 show\_cpuinfo() 보여 줌.)

```
# cat /proc/cpuinfo // PID-process ID 보여줌
system type      : Atheros AR9132 rev 2
machine          : Buffalo WZR-HP-G300NH
processor        : 0
cpu model        : MIPS 24Kc V7.4
BogoMIPS         : 266.24
wait instruction  : yes
microsecond timers : yes
```

 **Proc 파일 시스템** //시스템 내의 정보를 제공하기 위한 파일시스템

```
# cat /proc/devices // devices 정보 보여줌
Character devices:
1 mem
4 ttyS
...
180 usb
189 usb_device

Block devices:
259 blkext
31 mtblock
```

# Memory 관련 실습

 **Cat /proc/self/maps** (시스템 virtual dynamic shared object보여 줌.)

```
# root@OpenWrt:~# cat /proc/self/maps
00400000-00470000 r-xp 00000000 1f:03 361      /rom/bin/busybox
00470000-00471000 rw-p 00070000 1f:03 361      /rom/bin/busybox
00471000-00472000 rwxp 00000000 00:00 0        [heap]
2aaa8000-2aaad000 r-xp 00000000 1f:03 206      /rom/lib/ld-uClibc-0.9.30.1.so
2aaad000-2aaae000 rw-p 00000000 00:00 0
2aabc000-2aabd000 r--p 00004000 1f:03 206      /rom/lib/ld-uClibc-0.9.30.1.so
2aabd000-2aabe000 rw-p 00005000 1f:03 206      /rom/lib/ld-uClibc-0.9.30.1.so
2aabe000-2aac1000 r-xp 00000000 1f:03 205      /rom/lib/libcrypt-0.9.30.1.so
2aac1000-2aad0000 ---p 00000000 00:00 0
....
```