- **File System (FS)** can organize data expected to be retained after a program terminates by providing procedures to store, retrieve and update data, as well as manage the available space on the devices which contain it.
- **File** is a block of arbitrary information, or resource for storing information, which is available to a computer program and is usually based on some kind of durable storage.
- **Volatile FS**는 DRAM에 존재하며, 전원이 꺼지면 모든 데이터가 사라진다.
- **Non-volatile/General/Non-journaling FS**는 파일이 업데이트 될 때, 즉시 물리적으로 업데이트한다.
- **Non-volatile/General/Journaling FS**는 파일 파일의 업데이트 내용을 로그로 남겨뒀다가, 나중에 한 번에 업데이트한다.
- **Dedicated FS**는 장치에 사용되는 FS이다.
- **Special-purposed FS**에는 NFS, Virtual FS, Proc FS등이 있다.
- **Root File System** generally is small and contains very critical files, including libraries and utilities, etc. This file system mounted in "/" directory.
- **Command related i-node**

`ls –i`	i-node number를 보여줌.
`ls –il`	i-node 정보를 자세히 보여줌.
`ln [-s]`	Hard link [Soft link]

- **Hard Links** is a reference to the physical data on a file system.

  All named files are hard links. / More than one name can be associated with the same physical data. / Hard links can only refer to data that exists on the same file system.

  When a file has more than one link, you can remove any one link and still be able to access the file through the remaining link. It is good way to backup files without copy.

- **Soft or Symbolic Links** is an indirect pointer to a file.

  It can point to a directory, a file on a different file system or a nonexistent file (broken link).

  The i-node number is different from the pointed to file. / The link counter of the new symbolic link file is "1". / Symbolic link file does not affect the link counter of the pointed to file. / The type field of symbolic file contains the letter "l". / The symbolic link file and the pointed to file have different status information such as file size, last modification time etc.

  The symbolic link to a directory has a file type of "l" (the first letter of the permission field). / The permission on the link are set to "rwx" for all. / chmod on the link applies to the actual directory (or file), the permissions on the link stay the same. / Can point to nonexistent directory.

- **Direct/Indirect Blocks**

  파일의 크기가 작으면 Direct, 크기가 커지면 Indirect 방식으로 데이터를 접근한다. 커질수록 간접 접근 수가 늘어난다 (Single, Double, Triple……).

- **Configuring a Linux Kernel**

`make menuconfig`	키보드 또는 마우스를 사용
`make xconfig`	GUI를 이용
`make config`	터미널에서 텍스트를 이용

- **Building the Linux Kernel Binary**

1. Run `make config`
2. Run `make dep` dependency 체크
3. Run `make clean`
4. Run `make bzImage` 이미지 압축
5. Run `make modules` 모듈 컴파일
6. Run `make modules install` 모듈 설치
7. Copy image file to `/boot/vmlinuz`
8. Config lilo/grub about the new kernel

lilo/grub은 부트매니저임.

- **Kernel Porting** make an existing kernel source to run on a target system.
- **Board Support Package (BSP)** includes Bootloader, Drivers, Network Protocols.
- **I/O Protection** prevent users from performing illegal I/Os. 잘못된 I/O를 방지.
- **Memory Protection** prevent users from modifying kernel code and data structures. 접근할 수 없는 메모리의 접근을 방지.
- **CPU Protection** prevent a user from using the CPU for too long. CPU 독점 방지.
- **Privileged Mode**

  Special instructions for Mapping, TLB, Device registers, I/O channels, etc.

  Setting processor features mode bits.

  Device accesses in privileged mode.

  이러한 기능은 하나의 주체(OS)에서 관리 되어야함.
- **Protection Rings**

  Level 0 for OS kernel
  Level 1, 2 for OS services (system call)
  Level 3 for Applications
- **System Call Mechanism**

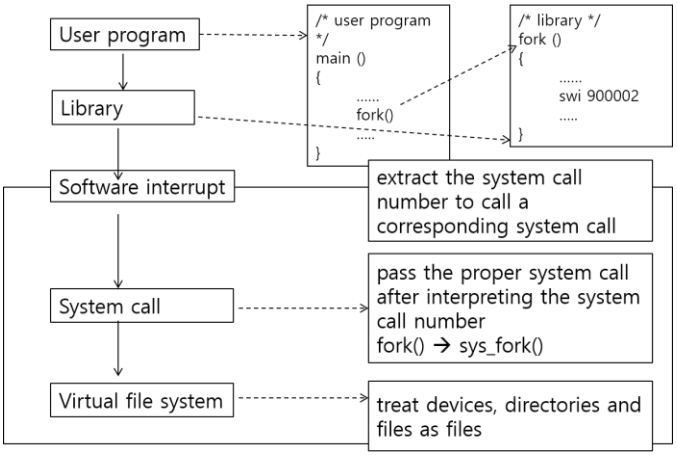
  1. User code can be arbitrary.
  2. User code cannot modify kernel memory.
  3. Makes a system call with parameters.
  4. The call mechanism switches code to kernel mode.
  5. Execute system call.
  6. Return with results.
- **System Calls** is interface between a process and the Operating System kernel.

  Process management / Memory management / File management / Device management / Communication
- **Trap Handler**

  System call이 발생되면 System Service Dispatcher이 System Service Table에서 요청한 System call을 찾아서 System Service를 실행한다.

  Exception이 발생하면 Exception Dispatcher가 발생된 Exception의 handler를 찾아서 실행함.

  Exception은 CPU 내부에서 발생한다
  Interrupt는 외부 장치에서 발생한다.
  Trap은 SW Interrupt이다.
- **Sequences to Handle `fork` System Call**



`swi` 명령어는 SW interrupt이다. 명령어가 처리되면 User mode에서 Privileged mode로 바뀐다.
900002는 System Call의 번호이다.

- **System Call Directories**

  related to process: `<linux>/kernel`
  related to file: `<linux>/fs`
  related to CPU: `<linux>/arch`
  related to ARM: `<linux>/arch/arm/kernel`

  The header files of a system call function must be in `<linux>/arch/arm/include/asm` directory because it is a kernel function.
- **Allocate a System Call Number** in `<linux>/arch/arm/include/asm/unistd.h`
- **Register a System Call** in `<linux>/arch/arm/kernel/calls.S`

- **Kernel Build and Fusing a Target System**

  1. Modify `<linux>/arch/arm/include/Makefile`
  2. Make `<linux>/Makefile`
  3. Copy a new kernel image through USB, after recompiling it.
  4. Reboot it after fusing it into a target system.

- **System Call Implementation**

```
01. #include <linux/kernel.h>
02.
03. asmlinkage void sys_hello(){
04.     printk("Hello\n");
05. }
```

<linux>/kernel/hello.c

Add a new system call with a new number.

```
...
#define __NR_...   (__NR_SYSCALL_BASE+375)
#define __NR_hello (__NR_SYSCALL_BASE+376)
...
```

<linux>/arch/arm/include/asm/unistd.h

Add a new system call

```
...
CALL(sys_...)
CALL(sys_hello)
...
```

<linux>/arch/arm/kernel/calls.S

Edit makefile (obj-y 목록은 무조건 built-in)

```
...
obj-y := ... hello.o
...
```

<linux>/arch/arm/kernel/Makefile

Test

```
01. #include <linux/unistd.h>
02. #include <error.h>
03.
04. int main(){
05.     syscall(__NR_hello);
06.     return 0;
07. }
```

- **copy_to_user and copy_from_user**

  Functions to safely copy data to/from user space. Check Validity of pointer arguments for you. Return zero if successful, number of bytes that fail if there is a problem. Immune to page faults, pre-emption, null pointers, other errors, etc.

- **printk**

  May be called from almost anywhere in kernel. Write to system log and output survives crashes almost all of the time.

  System logs in `/var/log/messages`, it needs root privileges to read. Log-levels are 0(high) to 7(low).

- **Kernel APIs: Memory Allocation**

```
#include <linux/slab.h>
void *kmalloc(size_t size, int priority);
void kfree(const void *obj);
```

- **Kernel APIs: The Ptr to a Current Running Proc**

```
#include <linux/sched.h>
struct task_struct *current;
// current->pid
//      The process ID of the current
//      running process
// current->comm
//      The command name of the current
//      running process
//      현재 프로세스를 실행시킨 명령어
```

- **Signals**

  Some signals can be handled (caught) by the process. Handling takes the form of registering a signal handler. Signal handlers are functions that get called when the signal occurs.

- Some signals cannot be handled by the process, such as `SIGKILL`, `SIGSTOP`.

```
// Examine and change a signal action
int sigaction(
        int signum,
        const struct sigaction *act,
        struct sigaction *oldact);
// install a new signal handler (depre)
sighandler_t signal(
        int signum,
        sighandler_t hanlder);
// Wait for a signal
int pause(void);
// Send a signal
int kill(pid_t pid, int sig);
// arranges for a SIGALRM signal
// to be delivered
unsigned int alarm(unsigned int seconds);
```

- **Signal Program**

```
01. #include <stdio.h>
02. #include <unistd.h> // sleep(1)
03. #include <signal.h>
04.
05. void ex_program(int sig);
06.
07. int main(void){
08.     // Ctrl+C to ex_program function.
09.     (void)signal(SIGINT, ex_program);
10.     while(1){
11.             printf("sleeping..");
12.             sleep(1);
13.     }
14.     return 0;
15. }
16.
17. void ex_program(int sig){
18.     printf("wake up call\n");
19.     printf("Catched signal: %d", sig);
20.     // Ctrl+C to Default handler
21.     (void)signal(SIGINT, SIG_DFL);
22. }
```

- **Kernel Module** is Pluggable module to the OS that adds functionality.

  동적으로 추가 가능하고, kernel의 내용을 읽고 쓸 수 있다. Kernel 모듈에 사용되는 전역 변수나 함수는 Kernel Symbol Table에 등록되고, /proc/kallsyms를 통해서 제공된다.

  EXPORT_NOSYMBOLS: private

  EXPORT_SYMBOL(), EXPORT_SYMBOL_GPL(): public

- **Module Programming 2.4**

```
01. #include <linux/kernel.h>
02. #include <linux/module.h>
03. #include <linux/init.h>
04.
05. MOCULE_LICENSE("GPL");
06.
07. int module_start(){
08.     ...
09.     return 0;
10. }
11.
12. void module_end(){
13.     ...
14. }
15.
16. module_init(module_start);
17. module_exit(module_end);
```

- **Module Programming 2.6**

```
01. #include <linux/module.h>
02. #include <linux/kernel.h>
03.
04. int init_module(void){
05.     printk(KERN_INFO "Hello world");
06.     return 0;
07. }
08.
09. void cleanup_module(void){
10.     printk(KERN_INFO "Goodbye world");
11. }
```

- **Module Commands**

  To insert (Linux kernel에 모듈을 동적으로 추가)

```
insmod <modname>
modprobe <modname> <args>
```

  To remove

```
rmmod <modname>
modprobe -r <modname>
```

  To list (현재 kernel에 추가되어 실행되고 있는)

```
lsmod
modprobe -l
```

- **The /proc** is pseudo file system. Real time, resides in the virtual memory. It doesn't exist any particular media.

  Tracks the processes running on the machine and the state of the system. Highly dynamic. The size of the proc directory is 0 and the last time of modification is the last bootup time.

  The contents of the /proc FS can be read by anyone who has the requisite permission. Certain parts of the /proc FS can be read only by the owner of the process and of course root.

  The contents of the /proc are used by many utilities which grab the data from the particular /proc directory and display it such as top, ps, lspci, etc.

- **/proc/buddyinfo** contains the number of free areas of each order for the kernel buddy system.

- **/proc/slabinfo** is information about memory usage on the slab level.

- **/proc/cmdline** is kernel command line.
- **/proc/cpuinfo** is information about the processors.
- **/proc/devices** is list of device drivers configured into the currently running kernel.
- **/proc/dma** shows which DMA channels are being used at the moment.
- **/proc/fb** is frame buffer devices.
- **/proc/filesystem** is FS configured/supported into/by the kernel.
- **/proc/iomem** shows the current map of the system's memory for its various devices.
- **/proc/ioports** provides a list of currently registered port regions used for input or output communication with a device.
- **/proc/kcore** represents the physical memory of the system and is stored in the core file format.

  Unlikke most /proc files, kcore does display a size. This value is given in bytes and is equal to the size of physical memory (RAM) used plus 4KB.

  Its contents are designed to be examined by a debugger, such as gdb the GNU debugger. Only the root user has the rights to view this file.
- **/proc/kmsg** used to hold messages generated by the kernel. These messages are then picked up by other programs, such as klogd.
- **/proc/loadavg** provides a look at load average.

  The first three columns measure CPU utilization of the last 1, 5, and 10 minute periods.

  The fourth column shows the number of currently running processes and the total number of processes.

  The last column displays the last process ID used.
- **/proc/locks** displays the files currently locked by kernel.
- **/proc/mdstat** contains the current information for multiple-disk, RAID configuration.
- **/proc/meminfo** is information about the current utilization of RAM on the system.
- **/proc/misc** lists miscellaneous drivers registered on the miscellaneous major device, which is number 10.
- **/proc/modules** displays a list of all modules that have been loaded by the system.
- **/proc/mounts** provides a quick list of all mounts in use by the system.
- **/proc/mtrr** refers to the current Memory Type Range Registers (MTRRs) in use with the system.
- **/proc/partitions** very detailed information on the various partitions currently available to the system.
- **/proc/stat** keeps track of a variety of different statistics about the system since it was last restarted.
- **/proc/swap** measures swap space and its utilization.
- **/proc/uptime** contains information about how long the system has on since its last restart.
- **/proc/version** tells the versions of the Linux kernel.
- **/proc/{The numerical named directories}** is the process ID which currently running.

  It contains detail information about corresponding process. You have full access only to the processes that you have started.
- **/proc/{numerical}/cmdline** contains the whole command line used to invoke the process. The contents of this file are the command line arguments with all the parameters. 이 프로세스를 실행 시킨 명령어.
- **/proc/{numerical}/cwd** is symbolic link to the current working directory.
- **/proc/{numerical}/environ** contains all the process-specific environment variables.

- **/proc/{numerical}/exe** symbolic link of the executable.
- **/proc/{numerical}/maps** is parts of the process' address space mapped to a file.
- **/proc/{numerical}/fd** contains the list of file descriptors as opened by the process.
- **/proc/{numerical}/root** is symbolic link pointing to the directory which is the root FS for the process.
- **/proc/{numerical}/status** is information about the process.
- **/proc/self/** is link to the currently running process.
- **/proc/bus/** contains information specific to the various buses available on the system such as ISA, PCI and USB buses.
- **/proc/driver/** specific drivers in use by kernel.
- **/proc/fs/** is information about FS.
- **/proc/ide/** is information about IDE devices.
- **/proc/irq/** used to set IRQ to CPU affinity.
- **/proc/net/** contains networking parameters and statistics such as arp, device, route.
- **/proc/scsi/** is information about SCSI devices.
- **/proc/sys/** allows you to make configuration changes to a running kernel.

  Changing a value within a /proc/sys file is done by the 'echo' command.

  Any configuration changes made thus will disappear when the system is restarted.
- **/proc/sys/dev** provides parameters for particular devices on the system.
- **/proc/sys/fs/, /proc/sys/net/, /proc/sys/vm/**
- **/proc/sys/kernel/**acct, crtl-alt-del, domainname, hostname, threads-max, panic

- **Module Programming – Use /proc FS Functions**

```
struct proc_dir_entry* creat_proc_entry(
        const char     *name,
        mode_t         mode,
        struct proc_dir_entry *parent
);
```

<linux/proc_fs.h>

This function creates a regular file with the name `name`, the mode `mode` in the directory `parent`. To create a file in the root of proc FS, use `NULL` as `parent` parameter.

Proc FS에 새로운 파일 엔트리를 생성하는 함수.

- **Module Programming – `proc_fs.h`**

```
01. #include <linux/kernel.h>
02. #include <linux/module.h>
03. #include <linux/init.h>
04. #include <linux/proc_fs.h>
05. #define PROC_FILENAME "helloProc"
06.
07. static int hello_read_proc(
08.     char   *buf,  char   **start,
09.     off_t  offset, int    count,
10.     int    *eof,  void   *data);
11.
12. static int module_start(void){
13.     struct proc_dir_entry *entry;
14.     if((entry == create_proc_entry(
15.             PROC_FILENAME,
16.             S_IRUGO, NULL)) == NULL)
17.             return -EACCES;
18.     entry->read_proc = hello_read_proc;
19.     return 0;
20. }
21.
22. static void module_end(void){
23.     printk("Good bye.\n");
24.     remove_proc_entry(
25.             PROC_FILENAME, NULL);
26. }
27.
28. static int hello_read_proc(
29.     char   *buf,  char   **start,
30.     off_t  offset, int    count,
31.     int    *eof,  void   *data){
32.     int len;
33.     len = sprintf(buf, "/proc FS");
34.     return len;
35. }
36.
37. MOCULE_LICENSE("GPL");
38. EXPORT_NO_SYMBOLS;
39. module_init(module_start);
40. module_exit(module_end);
```

- **Symbol Table**

  변수나 함수가 심볼로 등록되면, 다른 모듈에서 해당 변수나 함수를 전역 변수나 전역 함수인 것처럼 사용될 수 있음.

  하지만 해당 심볼을 정의하는 모듈이 Live 상태가 아니라면, 해당 심볼을 사용하는 모듈은 insmod가 안된다. 참조 카운트가 0이어야 rmmode 가능.

- **Module Programming – sched.h**

```
01. #include <linux/kernel.h>
02. #include <linux/module.h>
03. #include <linux/init.h>
04. #include <linux/proc_fs.h>
05. #include <linux/sched.h>
06. #define PROC_FILENAME “gettaskinfo”
07.
08. static int getTaskInfo_read_proc(
09.     char   *buf,  char   **start,
10.     off_t  offset,int    count,
11.     int    *eof,  void   *data);
12.
13. static int module_start(void){
14.     struct proc_dir_entry *entry;
15.
16.     printk(“Hello\n”);
17.
18.     if((entry == create_proc_entry(
19.             PROC_FILENAME,
20.             S_IRUGO, NULL)) == NULL)
21.             return -EACCES;
22.     entry->read_proc
23.             = getTaskInfo_read_proc;
24.     return 0;
25. }
26.
41. void module_end(void){
42.     printk(“Bye\n”);
43.
44.     remove_proc_entry(
45.             PROC_FILENAME, NULL);
27. }
28.
29. static int getTaskInfo_read_proc(
30.     char   *buf,  char   **start,
31.     off_t  offset,int    count,
32.     int    *eof,  void   *data){
33.     static char *s strState[] = {
34.             “Running”, “Sleep”,
35.             “Disk Sleep”, “Stop”,
36.             “Tracing Stop”,
37.             “Zombie”, “Dead”};
38.     int state, stateID, count = 0;
39.
40.     for(    stateID = 0,
41.             state = current->state;
42.             state > 0;
43.             state >>=1, ++stateID);
44.
45.     count += sprintf(buf + count,
46.      “pid: %d\n”, current->pid);
47.
48.     count += sprintf(buf + count,
49.      “command: %s\n”, current->comm);
50.
51.     count += sprintf(buf + count,
52.      “state: %s\n”,
53.     current->strState[stateID]);
54.
55.     return count;
56. }
57.
58. MOCULE_LICENSE(“GPL”);
59. EXPORT_NO_SYMBOLS;
60. module_init(module_start);
61. module_exit(module_end);
```

```
$ insmod getTaskInfo.ko
Hello
$ cat /proc/getTaskInfo
pid: 1064
command: cat
state: Running
$ rmmod getTaskInfo
Bye
```

- **Characters Devices** can be accessed as a stream of bytes such as mouse, keyboard, display, etc.
- **Block Devices** can be accessed as a block. A device that can host a filesystem such as disk.
- **Network Interfaces** is in charge of sending and receiving data packets, driven by the network subsystem of the kernel.
- **Device Driver** is software to control devices through kernel. It is a kernel module. Not all kernel module is a device driver, but vice versa isn't.
- **Major and Minor Numbers**

  The major number identifies the driver associated with the device. It is common for a driver to control several devices; the minor number provides a way for the driver to differentiate among them.

  Both major and minor number are restricted between 0 and 255 in the version 2.4 kernel. The number 0 and 255 is reserved.

  mknod is used to create a device fie, super user privileges are required for this operation.

  ```
  $ mknod /dev/hello c 254 0
  ```

- **Register or Unregister a Char Device Driver**

```
int register_chardev(
       unsigned int   major,
       const char     *name
       struct file_operations *fops);
```

```
int unregister_chardev(
       unsigned int   major,
       const char     *name);
```

<linux/fs.h>

Character Device를 Linux Kernel에 등록/해제 시켜 주는 함수. name은 /proc/devices 에 표시됨.

```
$ cat /proc/devices
```

major 인자를 0으로 하면 사용되지 않는 number 가 동적으로 사용된다.

- **File Operations**

An open device is identified internally be a file structure, and the kernel uses the file_operations structure to access the driver's functions.

You can implement only the most important device methods, and uses the tagged format to declare its file_operations structure.

```
struct file_operations fops{
       .owner: This module
       .read, .write,
       .ioctl, .open, .release};
```

- **File Operations: Open Method**

The open method is provided for a driver to do any initialization in preparation for later operations.

1. Increment the usage count.
2. Check for device-specific errors.
3. Initialize the device, if it is being opened for the first itme.
4. Identify the minor number and update the f_op pointer, if necessary.
5. Allocate and fill any data structure to be put in filp->private_data.

- **File Operations: Release Method**

The role of the release method is the revers of open.

1. Deallocate anything that open allocated in filp->private_data.
2. Shut down the device on last close.
3. Decrement the usage count.

- **File Operations: Read and Write Method**

The read and write methods perform a similar task, that is, copying data from and to application code.

```
ssize_t (*read)(
       struct file    *filp,
       char           *buff,
       size_t         count,
       loff_t         *offp);
```

```
ssize_t (*write)(
       struct file    *filp,
       char           *buff,
       size_t         count,
       loff_t         *offp);
```

<linux/include/media/media-devnode.h>

filp is the file pointer and count is the size of the requested data transfer.

The buff points to the user buffer holding the data to be written or the empty buffer where the newly read data should be placed.

offp is a pointer to a "long offset type" object that indicates the file position the user is accessing.

- **Cross-space Data Transfer**

```
unsigned long copy_to_user(
       void           *to,
       const void     *from,
       unsigned long  count);
```

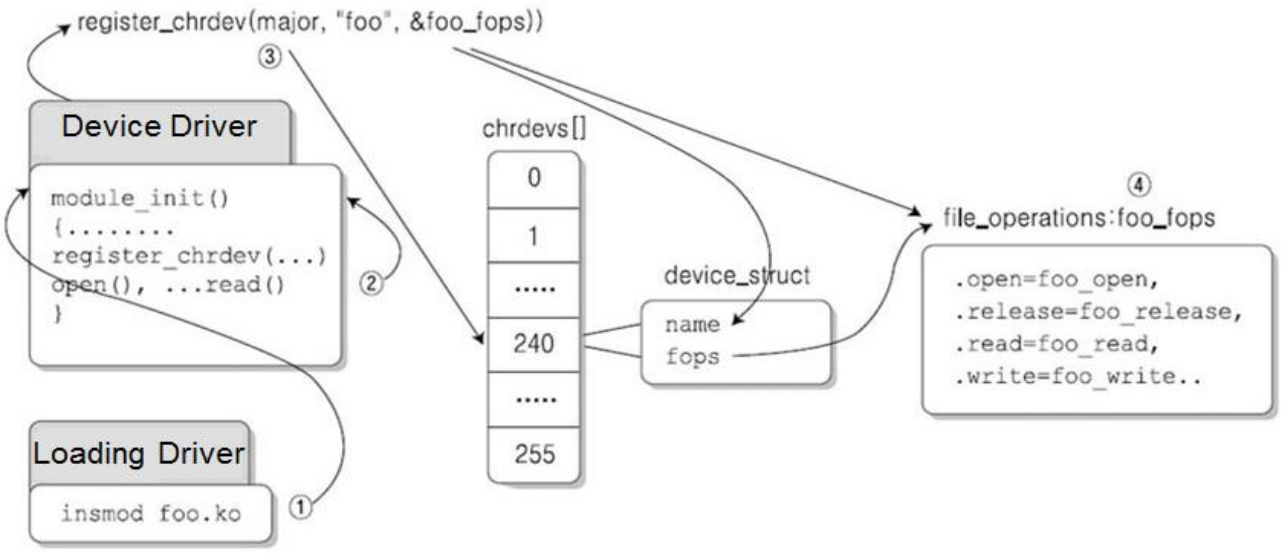
```
unsigned long copy_from_user(
       void           *to,
       const void     *from,
       unsigned long  count);
```

<linux/uaccess.h>

to is the address that copy data to.
from is the address that copy data from.
count is the data size.

read: transfer data from kernel to user.
write: transfer data from user to kernel.

- **The Sequences to Register a Character Device Driver**



- **Block Devices**

  Every block device must provide an interface to the buffer cache (as well as normal file operations). The data in the buffer cache can be reordered.

- **Network Device Drivers**

  Any network transaction is made through an interface/device. They are not accessed as files but by name (e.g. eth0).

  Interface can be hardware (e.g. Eth0) or pure software I/O (e.g. lo(=loopback)).

  Can be seen the registered network device drivers at `/proc/net/dev`.

  ```
  $ cat /proc/net/dev
  ```

- **FPGA LED Driver**

```
ssize_t iom_led_write(...){
    unsigned char value;
    const char *tmp = gdata;
    if (copy_from_user(&value, tmp, 1))
        return -EFAULT;
    outb(value,
        (unsigned int)iom_fpga_led_addr);
    return length;
}
```

```
ssize_t iom_led_read(...){
    unsigned char value;
    char *tmp = gdata;
    value = inb(
        (unsigned int)iom_fpga_led_addr);
    if (copy_to_user(tmp, &value, 1))
        return -EFAULT;
    return length;
}
```

```
int __init iom_led_init(void){
    int result;
    result = register_chrdev(
        IOM_LED_MAJOR, IOM_LED_NAME,
        &iom_led_fops);

    if(result < 0) {
        printk(KERN_WARNING
            "Can't get any major\n");
        return result;
    }

    // ioremap(phy addr, size)
    // mapping physical memory
    // to virtual memory.
    iom_fpga_led_addr =
        ioremap(IOM_LED_ADDRESS, 0x1);
    iom_demo_addr =
        ioremap(IOM_DEMO_ADDRESS, 0x1);

    outb(UON,(unsigned int)iom_demo_addr);
    printk(
        "init module, %s major
        number : %d\n",
        IOM_LED_NAME, IOM_LED_MAJOR);
    return 0;
}
```

```
void __exit iom_led_exit(void) {
    iounmap(iom_fpga_led_addr);
    iounmap(iom_demo_addr);
    unregister_chrdev(
        IOM_LED_MAJOR,
        IOM_LED_NAME);
}
```

- **FPGA FND (7-Segment) Driver**

```
int __init iom_fpga_find_init(void){
    ...
    // there are 4 fnd.
    iom_fpga_fnd_addr = ioremap(
        IOM_FND_ADDRESS, 0x4);
    iom_demo_addr = ioremap(
        IOM_FND_ADDRESS, 0x1);
    ...
}
```

```
ssize_t iom_fpga_fnd_write(...){
    int  i;
    unsigned char value[4];
    const char *tmp = gdata;
    if(copy_from_user(&value, temp, 4))
        return –EFAULT;
    for(i = 0; I < length; i++)
        outb(value[i], (unsigned int)
            iom_fpga_fnd_addr + i);
    return length;
}
```

- **FPGA DOT MATRIX Driver**

```
int __init iom_fpga_find_init(void){
    ...
    // there are 10 dot array.
    iom_fpga_fnd_addr = ioremap(
        IOM_FND_ADDRESS, 0x10);
    iom_fpga_fnd_addr = ioremap(
        IOM_FND_ADDRESS, 0x1);
    ...
}
```

```
ssize_t iom_fpga_fnd_write(...){
    int  i;
    unsigned char value[10];
    const char *tmp = gdata;
    if(copy_from_user(&value, temp, 4))
        return –EFAULT;
    for(i = 0; I < length; i++)
        outb(value[i], (unsigned int)
            iom_fpga_dor_addr + i);
    return length;
}
```

- **FPGA TEXT LCD Driver**

```
int __init iom_fpga_find_init(void){
    ...
    // there are 32 characters.
    iom_fpga_fnd_addr = ioremap(
        IOM_FOGA_LCD_ADDRESS, 0x32);
    iom_fpga_fnd_addr = ioremap(
        IOM_FND_ADDRESS, 0x1);
    ...
}
```

```
ssize_t iom_fpga_text_lcd_write(...){
    int  i;
    unsigned char value[32];
    const char *tmp = gdata;
    if(copy_from_user(&value, temp, 32))
        return –EFAULT;
    value[length] = 0;
    for(i = 0; I < length; i++)
        outb(value[i], (unsigned int)
            iom_fpga_text_lcd_addr + i);
    return length;
}
```

- **FPGA Dip Switch Driver**

```
int __init iom_fpga_switch_init(void){
    ...
    // there are 8 switches
    iom_fpga_fnd_addr = ioremap(
        IOM_FND_ADDRESS, 0x1);
    iom_fpga_fnd_addr = ioremap(
        IOM_FND_ADDRESS, 0x1);
    ...
}
```

```
ssize_t iom_fpga_dip_switch_read(...){
    unsigned char dip_sw_value;
    dip_sw_value = inb((unsigned int)
        iom_fpga_dip_switch_addr);
    if(copy_to_user(gdata, &dip_sw_value,
        1)) return –EFAULT;
    return length;
}
```

```
...
dev= open(“/dev/fpga_dip_switch”, O_RDWR);
if(dev < 0){
    printf(“Device Open Error”);
    close(dev);
    return -1;
}

while(!quit){
    usleep(40’0000); //us, 40ms = 0.4sec
    read(dev, &dip_sw_buff, 1);
    printf(“Read dip switch: 0x%02X \n”,
        dip_sw_buff);
}

close(dev);
```

- **FPGA Push Switch Driver**

```
int __init iom_fpga_switch_init(void){
    ...
    // there are 9 buttons
    iom_fpga_fnd_addr = ioremap(
       IOM_FPGA_PUSH_SWITCH_ADDRESS, 0x9);
    iom_fpga_fnd_addr = ioremap(
       IOM_FPGA_PUSH_SWITCH_ADDRESS, 0x1);
    ...
}
```

```
ssize_t iom_fpga_push_switch_read(...){
    int i;
    unsigned char push_sw_value[MAX_BTN];
    for(i = 0; i < length; i++)
        push_sw_value[i]  =  inb((unsigned
int)iom_fpga_push_switch_addr + i);
    if(copy_to_user(gdata, push_sw_value,
        length)) return -EFAULT;
    return length;
}
```

- **FPGA Buzzer Driver**

```
int __init iom_fpga_switch_init(void){
    ...
    iom_fpga_fnd_addr = ioremap(
       IOM_FPGA_PUSH_SWITCH_ADDRESS, 0x1);
    iom_fpga_fnd_addr = ioremap(
       IOM_FPGA_PUSH_SWITCH_ADDRESS, 0x1);
    ...
}
```

```
ssize_t iom_buzzer_write(...){
    unsigned char value;
    const char *tmp = gdata;
    if(copy_from_user(&value, tmp, 1))
        return -EFAULT;
    outb(value, (unsigned int)
        iom_fpga_buzzer_addr);
    return length;
}
```

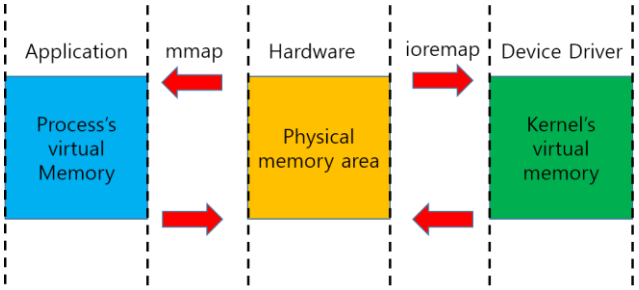
- **Device Size**

device	size
LED	1
FND	4
Dot Matrix	10
Text LCD	32
Dip Switch	1
Push Switch	9
Buzzer	1
Step Motor	4

- **Frame Buffer** is a hardware displaying graphics on linux system. Typically, LCD controller has this on embedded system. Frame buffer device can be accessed with `/dev/fb0`. It can be accessed via system call or `mmap`.

Select all list of frame buffer device nodes.

```
$ ls -al /dev/fb*
```



- **Cheat Note**

sched.h

```
create_proc_entry(filename, S_IRUGO, NULL)
remove_proc_entry(filename, NULL)
```

write

```
copy_from_user(&kernel, user, size)
outb(value, (unsigned int)addr)
```

read

```
value = inb(addr)
copy_to_user(user, &kernel, size)
```

init

```
register_chrdev(major, name, &fops)
vir_addr = ioremap(phy_addr, size)
```

exit

```
iounmap(vir_addr)
unregister_chrdev(major, name)
```

application

device = open(dev_name, O_RDWR|O_NDELAY)

ioctl(device, value)

write(device, write_buffer, size)

read(device, read_buffer, size)

- **Command to Restart NFS Server**

  ```
  service nfs-kernel-server restart
  ```

- **Mount NFS**

  Host: 165.225.167.200/mnt/nfs_host

  Target: 165.225.167.201/mnt/nfs_target

  ```
  mount –t nfs 165.225.167.200:/mnt/nfs_host
   /mnt/nfs_target
  ```

- **Change Mod**

  ```
  chmod 660 ./*
  chmod –R 660 ./*
  ```

- **Module Programming**

  기능을 커널에 동적으로 추가할 수 있는 커널 모듈을 만들어 프로그래밍 하는 것.

- **NFS**

  다른 컴퓨터에 있는 파일 시스템을 local 파일 시스템처럼 접근가능하게 해주는 프로토콜

- **Major number and Minor number**

  Major: 장치의 type을 구분하기 위한 번호

  Minor: 같은 type의 여러 장치들을 구분하기 위한 번호.

- **Build Linux Kernel**

  1. Modify `<linux>/arch/arm/kernel/Makefiel`
  2. Make `<linux>/Makefile`
  3. Copy a new kernel image through USB, after recompiling it.
  4. Reboot it after fusing it into a target system.

- **more and less**

  둘 다 화면에 내용을 표시해주는 유틸리티지만, more는 아래로만 스크롤 되는 오래된 유틸리티 이고, less는 위/아래 모두 스크롤 되는 유틸리티이다.

  less는 거대하기때문에 소형 임베디드 시스템에서는 less보다 more가 더 자주 사용된다.

-

- **Implement a System Call into Linux Kernel**

  1. Implement a system call into hello.c

     Edit `<linux>/arch/arm/kernel/hello.c`

     ```
     01. #include <linux/kernel.h>
     02.
     03. asmlinkage void sys_hello(){
     04.     printk(“Hello\n”);
     05. }
     ```

  2. Add new system call number into

     `<linux>/arch/include/asm/unistd.h`

     ```
     ...
     #define __NR_setns (__NR_SYSCALL_BASE+375)
     #deinfe __NR_hello (__NR_SYSCALL_BASE+376)
     ```

  3. Register a new system call into

     `<linux>/arch/arm/kernel/calls.S`

     ```
     /* 375 */    CALL(sys_setns)
                  CALL(sys_hello)
     ```

  4. Edit Makeifle.

     Modify `<inux>/arch/arm/kernel/Makefile`

     ```
     obj=y := elf.o entry-armv.o ...
              ...
              traps.o hello.o

     obj=&(...) += compact.o
     ```

  5. Test-hello.c is a program to test a newly added system call.

     ```
     01. #include <linux/unistd.h>
     02. #iunclude <errono.h>
     03.
     04. int main(){
     05.     syscall(__NR_hello);
     06.     return 0;
     07. }
     ```

-