CS 3502
Operating Systems

File and Directory

Kun Suo

Computer Science, Kennesaw State University

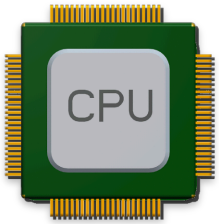
https://kevinsuo.github.io/

Outline

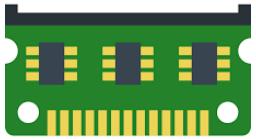
- File: abstraction of storage
  - Name, structure, type
  - Access, attribute, operation
- Memory-mapped files
- Directory: a file to organize files
  - Single-level, two-level, hierarchical
  - Path, operation

Revisit OS Abstractions

- Make one physical CPU look like multiple virtual CPUs
  - One or more virtual CPUs per process



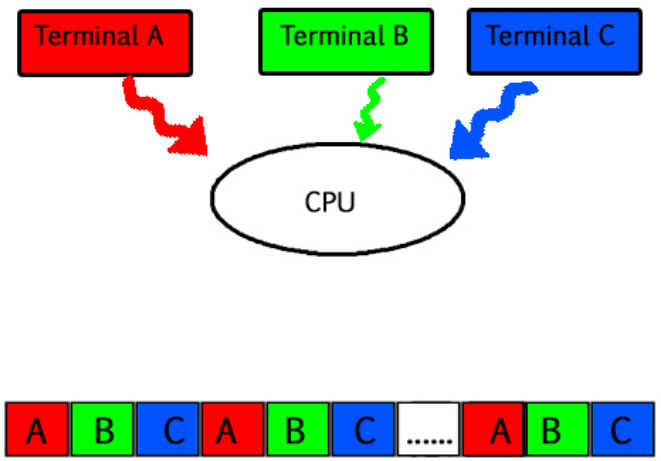
- Make physical memory (RAM) and look like multiple virtual memory spaces
  - One or more virtual memory spaces per process



- Make physical disk look like a file system
  - Physical disk = raw bytes.
  - File system = user's view of data on disk. It is an extended machine

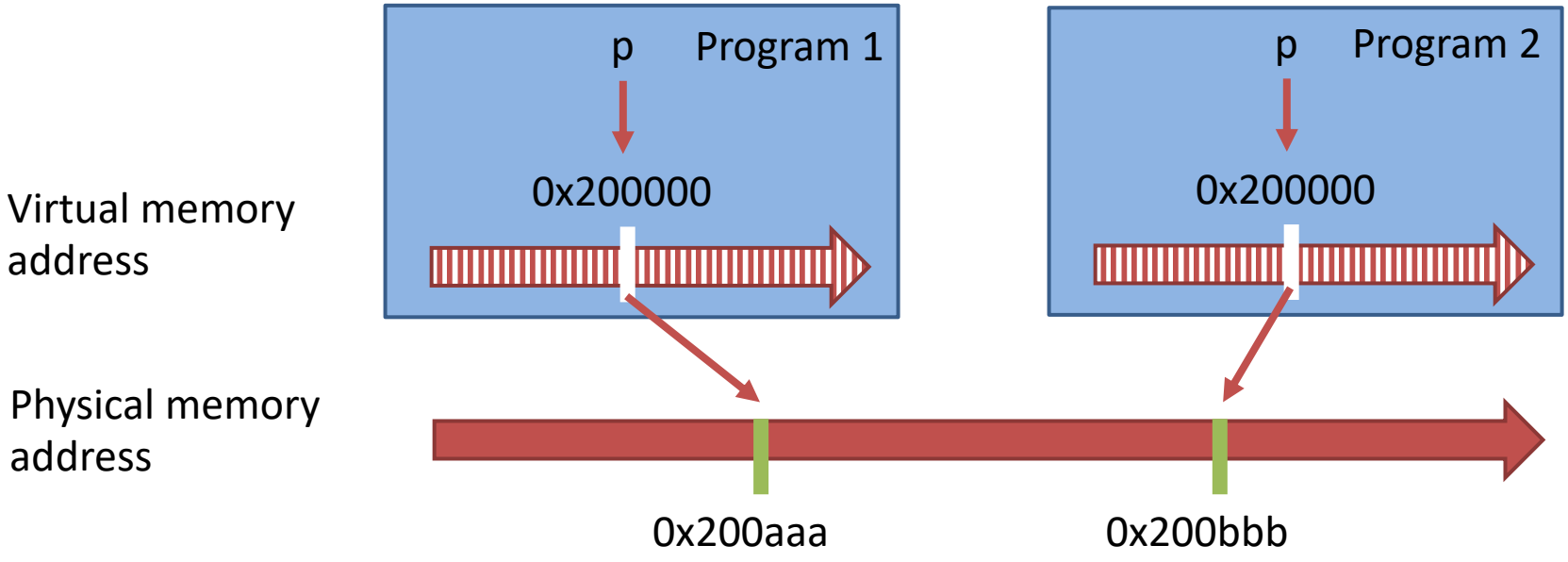
CPU Abstraction

- Process: virtualization of CPU
  - Run many instances of “cpu” program on a single core CPU. Even though we have only one processor, somehow all these programs seem to be running at the same time!
  - How? The operating system, with some help from the hardware, turns (or virtualizes) a single CPU (or small set of them) into a seemingly infinite number of CPUs and thus allowing many programs to seemingly run at once → virtualizing the CPU.



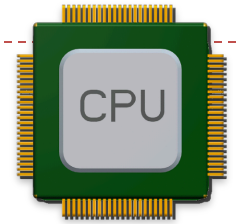
Memory Abstraction

- Address space: virtualization of memory
  - Each process accesses its own private virtual memory address space (sometimes just called its address space), which the OS somehow maps onto the physical memory of the machine.

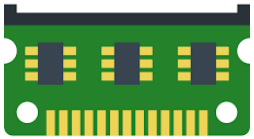


OS Abstractions

(1) Process: virtualization of CPU



(2) Address space: virtualization of memory

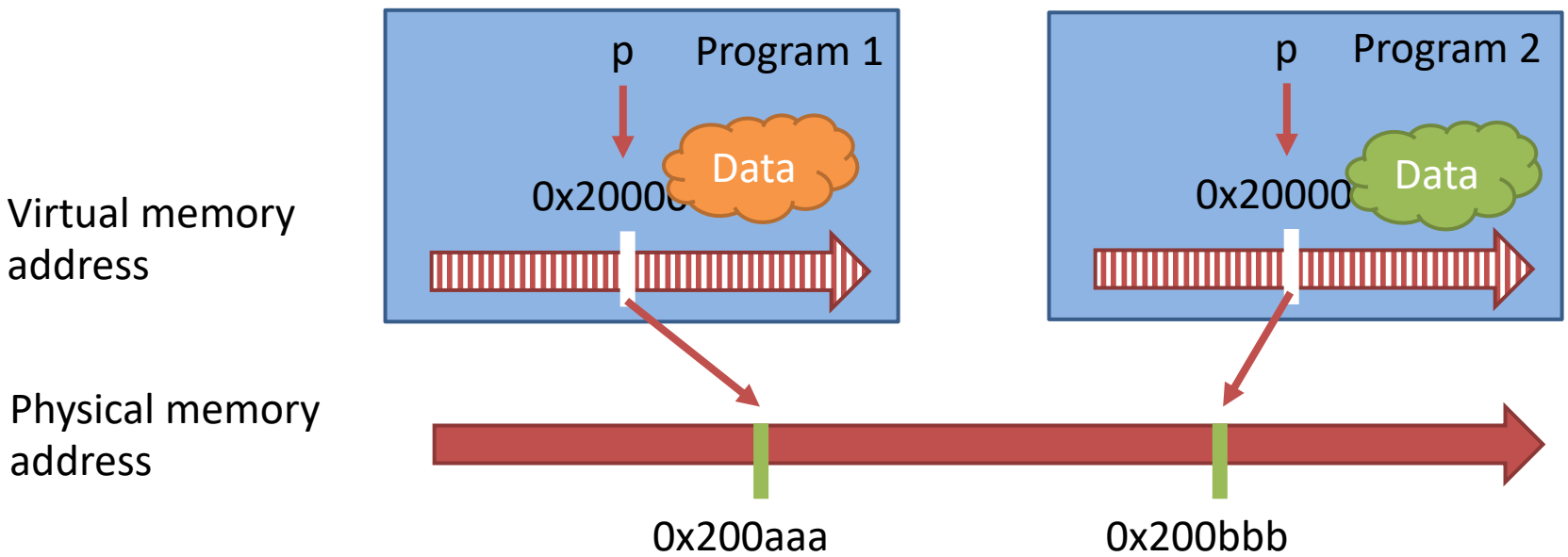


The above to allow a program to run as if it is in its own private, isolated world (CPU and memory)

(3) File: virtualization of persistent storage

Question

- Why do not we store information in the process address space?



Question

- Why do not we store information in the process address space? Bad idea because:

- Size is limited to size of virtual address space
  - May not be sufficient for airline reservations, banking, etc.
- The data is lost when the application terminates
  - Even when computer doesn't crash!
- Multiple process might want to access the same data
  - Imagine a telephone directory part of one process

Long-term Information Storage

- Three essential requirements for long-term information storage
  - Must store a large amount of data
  - Information stored must survive when processes stop using it
  - Multiple processes must be able to access the information concurrently

- A ***file*** is an abstraction of the long-term (persistent) data storage (From KB to GB) and can be access concurrently from different processes

- OS operates the files through the ***file system***

File Naming

- Two-part file names

Format

Extension	Meaning
file.bak	Backup file
file.c	C source program
file.gif	Compuserve Graphical Interchange Format image
file.hlp	Help file
file.html	World Wide Web HyperText Markup Language document
file.jpg	Still picture encoded with the JPEG standard
file.mp3	Music encoded in MPEG layer 3 audio format
file.mpg	Movie encoded with the MPEG standard
file.o	Object file (compiler output, not yet linked)
file.pdf	Portable Document Format file
file.ps	PostScript file
file.tex	Input for the TEX formatting program
file.txt	General text file
file.zip	Compressed archive

File Types

Is different from File Formats

- There exist 3 types of files in Linux
  - Regular files
  - Directory files
  - Special files
    - Block file(b)
    - Character device file(c)
    - Named pipe file or just a pipe file(p)
    - Symbolic link file(l)
    - Socket file(s)

File Types: Regular files

- Definition:
  - Readable file or Binary file or Image files or Compressed files etc.
- Linux: list regular files
  - ls -l | grep '^-'

fish /home/ksuo (ssh)

```
ksuo@ksuo-VirtualBox ~> ls -l | grep '^-'
-rw-rw-r--  1 ksuo ksuo        82 Nov  2 19:40 -
-rw-rw-r--  1 ksuo ksuo       625 Sep 25 14:07 1.txt
-rw-rw-r--  1 ksuo ksuo       443 Sep 25 14:07 2.txt
-rw-rw-r--  1 ksuo ksuo      2153 Sep 25 14:09 3.txt
-rw-rw-r--  1 ksuo ksuo        82 Nov  2 19:41 d
-rw-rw-r--  1 ksuo ksuo 164107798 May  6 01:11 linux-5.1.tar.gz
-rw-rw-r--  1 ksuo ksuo      3254 Oct 15 14:45 project-2-2.c
-rwxrwxr-x  1 ksuo ksuo     13144 Oct 15 14:45 project-2-2.o
-rw-rw-r--  1 ksuo ksuo      3254 Oct 15 14:37 project-2-2-old.c
-rwxrwxr-x  1 ksuo ksuo     13144 Oct 15 14:45 project-2-2-old.o
-rw-rw-r--  1 ksuo ksuo      2917 Sep 24 16:32 semaphore.c
-rwxrwxr-x  1 ksuo ksuo     12992 Sep 24 16:34 semaphore.o
-rwxrwxr-x  1 ksuo ksuo      8304 Sep  8 16:36 test
-rwxrwxr-x  1 ksuo ksuo      8304 Sep  8 15:56 test2
-rw-rw-r--  1 ksuo ksuo       227 Sep  8 15:56 test2.c
-rwxrwxr-x  1 ksuo ksuo      8304 Sep  8 17:18 test3
-rw-rw-r--  1 ksuo ksuo       252 Sep  8 17:18 test3.c
-rw-rw-r--  1 ksuo ksuo       227 Sep  5 15:51 test.c
```

File Types: Directory files

- Definition:
  - These type of files contains regular files/folders/special files stored on a physical device
- Linux: list directory files
  - ls -l | grep '^d'

```
fish  /home/ksuo (ssh)
ksuo@ksuo-VirtualBox ~> ls -l | grep '^d'
drwxr-xr-x  2 ksuo ksuo      4096 Sep  5 14:32 Desktop
drwxr-xr-x  2 ksuo ksuo      4096 Sep  5 14:32 Documents
drwxr-xr-x  2 ksuo ksuo      4096 Sep  5 14:32 Downloads
drwxrwxr-x  2 ksuo ksuo      4096 Sep 26 15:35 hw1
drwxrwxr-x  2 ksuo ksuo      4096 Sep 26 14:14 hw2
drwxrwxr-x  2 ksuo ksuo      4096 Oct 22 12:13 hw3
drwxrwxr-x 24 ksuo ksuo      4096 May  5 20:42 linux-5.1
drwxrwxr-x 25 ksuo ksuo      4096 Oct 22 10:54 linux-5.1-modified
drwxr-xr-x  2 ksuo ksuo      4096 Sep  5 14:32 Music
drwxr-xr-x  2 ksuo ksuo      4096 Sep  5 14:32 Pictures
drwxr-xr-x  2 ksuo ksuo      4096 Sep  5 14:32 Public
drwxr-xr-x  2 ksuo ksuo      4096 Sep  5 14:32 Templates
drwxr-xr-x  2 ksuo ksuo      4096 Sep  5 14:32 Videos
```

File Types: Block files

- Definition:
  - These files are hardware files most of them are present in /dev
- Linux: list block files
  - ls -l | grep '^b'

```
fish  /dev (ssh)                                                    ⌥⌘1
ksuo@ksuo-VirtualBox /dev> ls -l | grep '^b'
brw-rw----  1 root disk      7,   0 Nov  2 19:35 loop0
brw-rw----  1 root disk      7,   1 Nov  2 19:35 loop1
brw-rw----  1 root disk      7,  10 Nov  2 19:35 loop10
brw-rw----  1 root disk      7,  11 Nov  2 19:35 loop11
brw-rw----  1 root disk      7,  12 Nov  2 19:35 loop12
brw-rw----  1 root disk      7,  13 Nov  2 19:35 loop13
brw-rw----  1 root disk      7,  14 Nov  2 19:35 loop14
brw-rw----  1 root disk      7,  15 Nov  2 19:35 loop15
brw-rw----  1 root disk      7,  16 Nov  2 19:35 loop16
brw-rw----  1 root disk      7,   2 Nov  2 19:35 loop2
brw-rw----  1 root disk      7,   3 Nov  2 19:35 loop3
brw-rw----  1 root disk      7,   4 Nov  2 19:35 loop4
brw-rw----  1 root disk      7,   5 Nov  2 19:35 loop5
brw-rw----  1 root disk      7,   6 Nov  2 19:35 loop6
brw-rw----  1 root disk      7,   7 Nov  2 19:35 loop7
```

File Types: Character device files

- Definition:
  - It provides a serial stream of input or output. Your terminals are classic example for this type of files.
- Linux: list character device files (e.g., /dev)
  - ls -l | grep '^c'

```
fish  /dev (ssh)                                                    ⌥⌘1
ksuo@ksuo-VirtualBox /dev> ls -l | grep '^c'
crw-r--r--  1 root root     10, 235 Nov  2 19:35 autofs
crw-------  1 root root     10, 234 Nov  2 19:35 btrfs-control
crw-------  1 root root      5,   1 Nov  2 19:35 console
crw-------  1 root root     10,  58 Nov  2 19:35 cpu_dma_latency
crw-------  1 root root     10, 203 Nov  2 19:35 cuse
crw-------  1 root root     10,  61 Nov  2 19:35 ecryptfs
crw-rw----  1 root video    29,   0 Nov  2 19:35 fb0
crw-rw-rw-  1 root root      1,   7 Nov  2 19:35 full
crw-rw-rw-  1 root root     10, 229 Nov  2 19:35 fuse
crw-------  1 root root    242,   0 Nov  2 19:35 hidraw0
crw-------  1 root root     10, 228 Nov  2 19:35 hpet
crw-------  1 root root     10, 183 Nov  2 19:35 hwrng
crw-------  1 root root     89,   0 Nov  2 19:35 i2c-0
crw-r--r--  1 root root      1,  11 Nov  2 19:35 kmsg
crw-rw----  1 root disk     10, 237 Nov  2 19:35 loop-control
```

File Types: Pipe files

- Definition:
  - Pipe works as FIFO ("First In, First Out") and mostly is used for IPC
- Linux: list pipe files
  - ls -l | grep '^p'

```
fish  /home/ksuo (ssh)
ksuo@ksuo-VirtualBox ~> ls -l | grep '^p'

prw-r----- 1 root root 0 2010-02-15 09:35 /dev/.initramfs/usplash_outfifo

prw-r----- 1 root root 0 2010-02-15 09:35 /dev/.initramfs/usplash_fifo

prw------- 1 syslog syslog 0 2010-02-15 15:38 /var/run/rsyslog/kmsg
```

File Types: Symbolic link files

- Definition:
  - These are linked files to other files, such as directory or regular File
- Linux: list link files
  - ls -l | grep '^l'

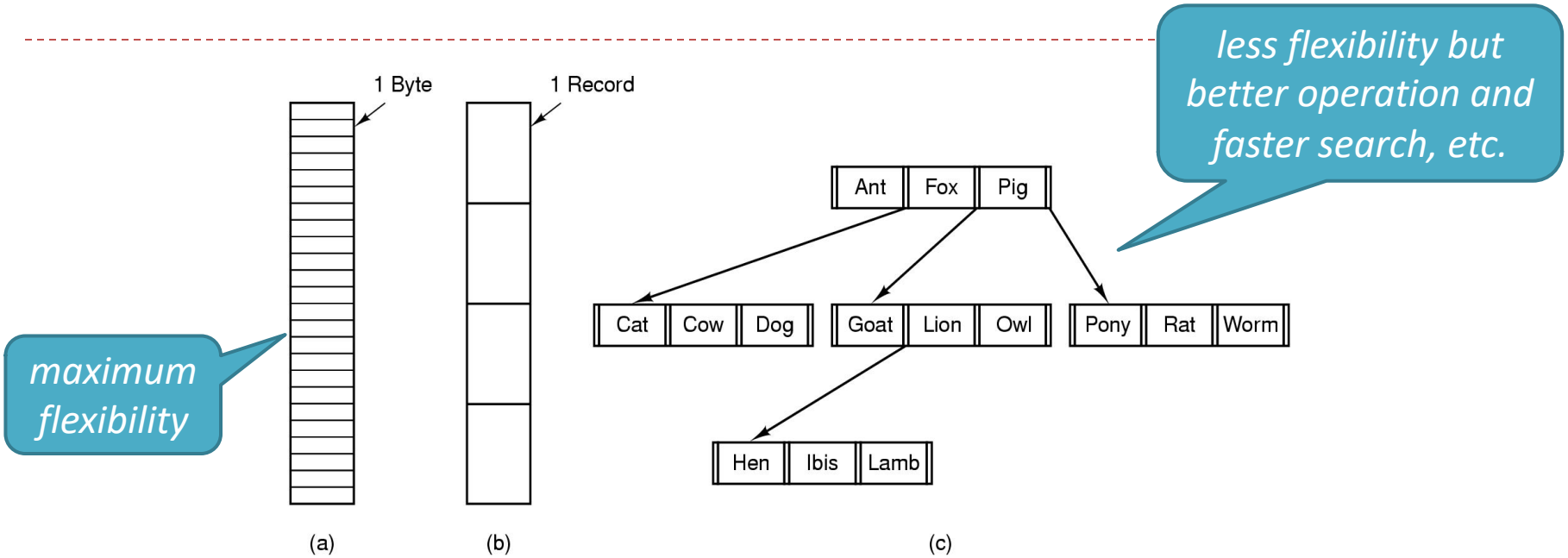
```
fish  /dev (ssh) — 89×12                                                    ⌥⌘1
ksuo@ksuo-VirtualBox /dev> ls -l | grep '^l'
lrwxrwxrwx  1 root root          3 Nov  2 19:35 cdrom -> sr0
lrwxrwxrwx  1 root root         11 Nov  2 19:35 core -> /proc/kcore
lrwxrwxrwx  1 root root          3 Nov  2 19:35 dvd -> sr0
lrwxrwxrwx  1 root root         13 Nov  2 19:35 fd -> /proc/self/fd
lrwxrwxrwx  1 root root         25 Nov  2 19:35 initctl -> /run/systemd/initctl/fifo
lrwxrwxrwx  1 root root         28 Nov  2 19:35 log -> /run/systemd/journal/dev-log
lrwxrwxrwx  1 root root          4 Nov  2 19:35 rtc -> rtc0
lrwxrwxrwx  1 root root         15 Nov  2 19:35 stderr -> /proc/self/fd/2
lrwxrwxrwx  1 root root         15 Nov  2 19:35 stdin -> /proc/self/fd/0
lrwxrwxrwx  1 root root         15 Nov  2 19:35 stdout -> /proc/self/fd/1
```

File Types: Socket files

- Definition:
  - A socket file is used to pass information between applications for communication purpose on two machines
- Linux: list socket files
  - ls -l | grep '^s'

```
fish  /home/ksuo (ssh)                                                    ⌥⌘1
ksuo@ksuo-VirtualBox ~> ls -l | grep '^s'

srwxrwxrwx 1 root root 0 2010-02-15 10:07 /var/run/cups/cups.sock

srwxrwxrwx 1 root root 0 2010-02-15 09:35 /var/run/samba/winbindd_privileged/pipe

srwxrwxrwx 1 mysql mysql 0 2010-02-15 09:35 /var/run/mysqld/mysqld.sock
```

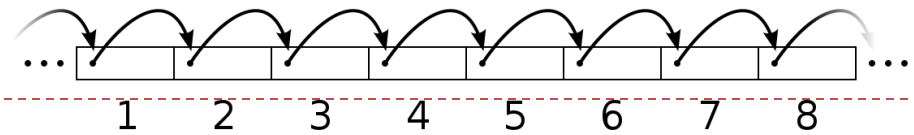
File Internal Structures



- Three kinds of file structures
  a) Byte Sequence: unstructured (Unix and WinOS view)
  b) Record sequence: r/w in records, relates to sector sizes (early machines' view)
  c) Complex structures, e.g. tree (data center server view)

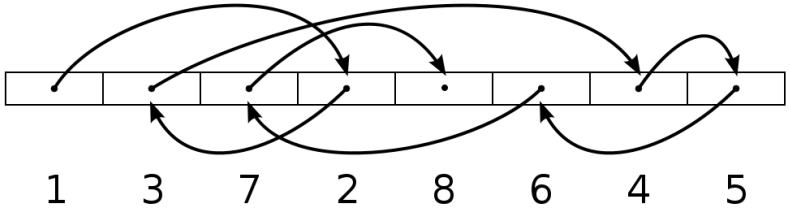
File Access

Sequential access



- Sequential access
  - read all bytes/records from the beginning
  - cannot jump around, could rewind or back up
  - convenient when medium was mag tape

Random access



- Random access
  - bytes/records read in any order
  - essential for database systems

File Attributes

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

File Attributes

- In Linux, use *stat* to check file attributes

```
fish  /home/ksuo (ssh)
ksuo@ksuo-VirtualBox ~> stat 1.txt
  File: 1.txt
  Size: 625             Blocks: 8          IO Block: 4096   regular file
Device: 801h/2049d      Inode: 2544958     Links: 1
Access: (0664/-rw-rw-r--)  Uid: ( 1000/    ksuo)   Gid: ( 1000/    ksuo)
Access: 2019-09-25 14:07:30.111665485 -0400
Modify: 2019-09-25 14:07:28.023687794 -0400
Change: 2019-09-25 14:07:28.023687794 -0400
 Birth: -
ksuo@ksuo-VirtualBox ~> stat Desktop/
  File: Desktop/
  Size: 4096            Blocks: 8          IO Block: 4096   directory
Device: 801h/2049d      Inode: 2490431     Links: 2
Access: (0755/drwxr-xr-x)  Uid: ( 1000/    ksuo)   Gid: ( 1000/    ksuo)
Access: 2019-11-02 19:32:10.811999887 -0400
Modify: 2019-09-05 14:32:54.064000000 -0400
Change: 2019-09-05 14:32:54.064000000 -0400
 Birth: -
```

File Operations

1. Create
2. Delete
3. Open
4. Close
5. Read
6. Write
7. Append
8. Seek
9. Get attributes
10. Set Attributes
11. Rename

File Related System Call

CLOSE	3	Close a file descriptor
CREAT	85	Open and possibly create a file
OPEN	2	Open and possibly create a file
OPENAT	257	Open and possibly create a file relative to a directory file descriptor
NAME_TO_HANDLE_AT	303	Obtain handle for a pathname
OPEN_BY_HANDLE_AT	304	Open file via a handle
MEMFD_CREATE	319	Create an anonymous file
MKNOD	133	Create a special or ordinary file
MKNODAT	259	Create a special or ordinary file relative to a directory file descriptor
RENAME	82	Rename a file
RENAMEAT	264	Rename a file relative to directory file descriptors
RENAMEAT	316	Rename a file relative to directory file descriptors
TRUNCATE	76	Truncate a file to a specified length
FTRUNCATE	77	Truncate a file to a specified length
FALLOCATE	285	Manipulate file space

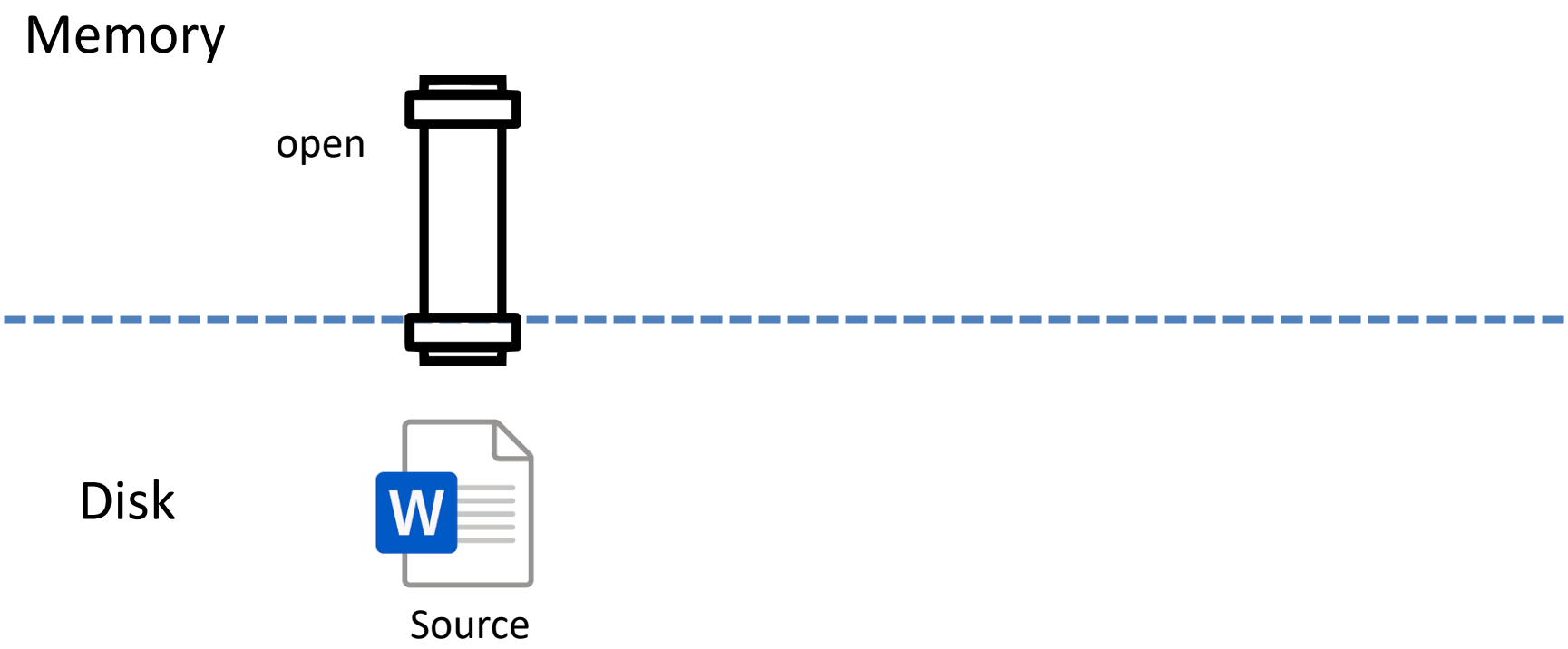
An Example Program Using File System Calls

```
/* File copy program. Error checking and reporting is minimal. */

#include <sys/types.h>                  /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);       /* ANSI prototype */

#define BUF_SIZE 4096                   /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700                /* protection bits for output file */

int main(int argc, char *argv[])
{
    int in_fd, out_fd, rd_count, wt_count;
    char buffer[BUF_SIZE];

    if (argc != 3) exit(1);             /* syntax error if argc is not 3 */
```

An Example Program Using File System Calls (cont.)

```
        /* Open the input file and create the output file */
        in_fd = open(argv[1], O_RDONLY);    /* open the source file */
        if (in_fd < 0) exit(2);             /* if it cannot be opened, exit */
        out_fd = creat(argv[2], OUTPUT_MODE);  /* create the destination file */
        if (out_fd < 0) exit(3);            /* if it cannot be created, exit */

        /* Copy loop */
        while (TRUE) {
            rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
        if (rd_count <= 0) break;           /* if end of file or error, exit loop */
            wt_count = write(out_fd, buffer, rd_count); /* write data */
            if (wt_count <= 0) exit(4);     /* wt_count <= 0 is an error */
        }

        /* Close the files */
        close(in_fd);
        close(out_fd);
        if (rd_count == 0)                  /* no error on last read */
            exit(0);
        else
            exit(5);                        /* error on last read */
    }
```

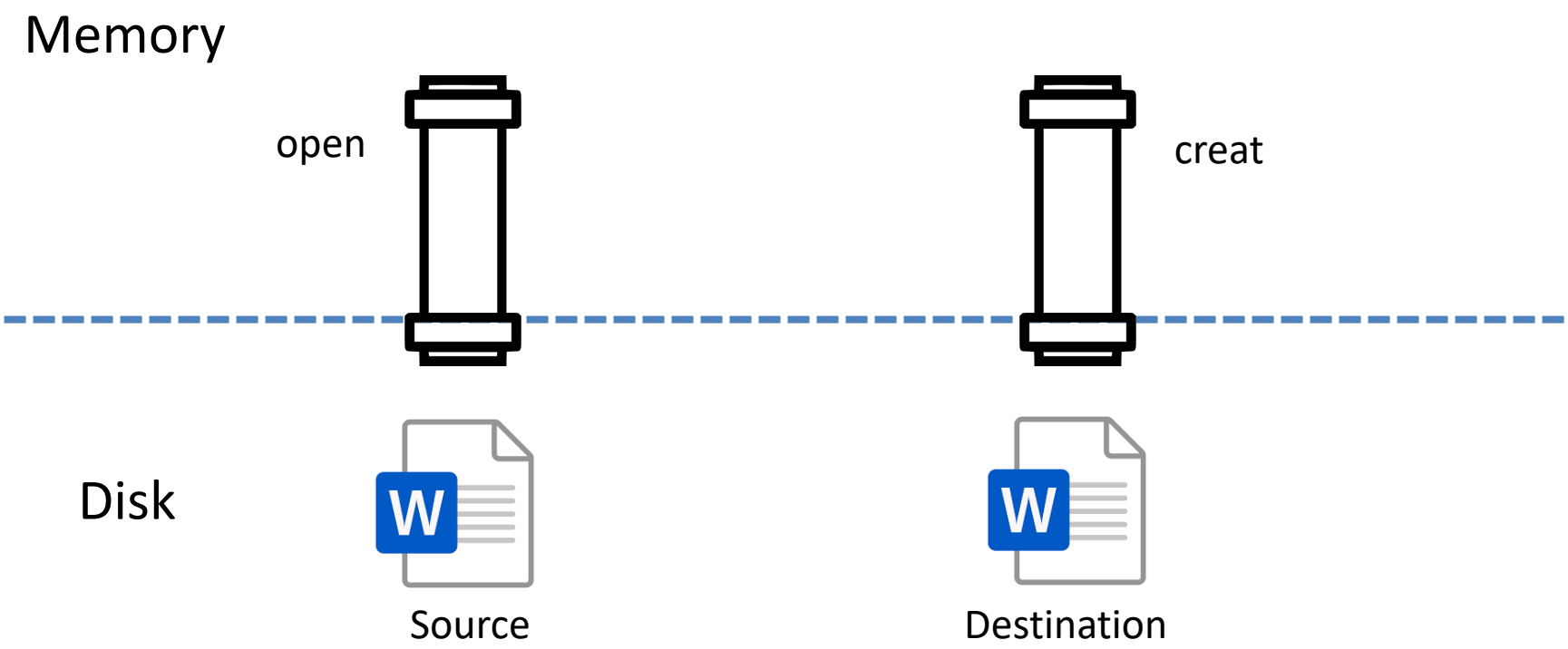
An Example Program Using File System Calls (cont.)



An Example Program Using File System Calls (cont.)

```
        /* Open the input file and create the output file */
        in_fd = open(argv[1], O_RDONLY);    /* open the source file */
        if (in_fd < 0) exit(2);             /* if it cannot be opened, exit */
        out_fd = creat(argv[2], OUTPUT_MODE);  /* create the destination file */
        if (out_fd < 0) exit(3);            /* if it cannot be created, exit */

        /* Copy loop */
        while (TRUE) {
            rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
        if (rd_count <= 0) break;           /* if end of file or error, exit loop */
            wt_count = write(out_fd, buffer, rd_count); /* write data */
            if (wt_count <= 0) exit(4);     /* wt_count <= 0 is an error */
        }

        /* Close the files */
        close(in_fd);
        close(out_fd);
        if (rd_count == 0)                  /* no error on last read */
            exit(0);
        else
            exit(5);                        /* error on last read */
    }
```

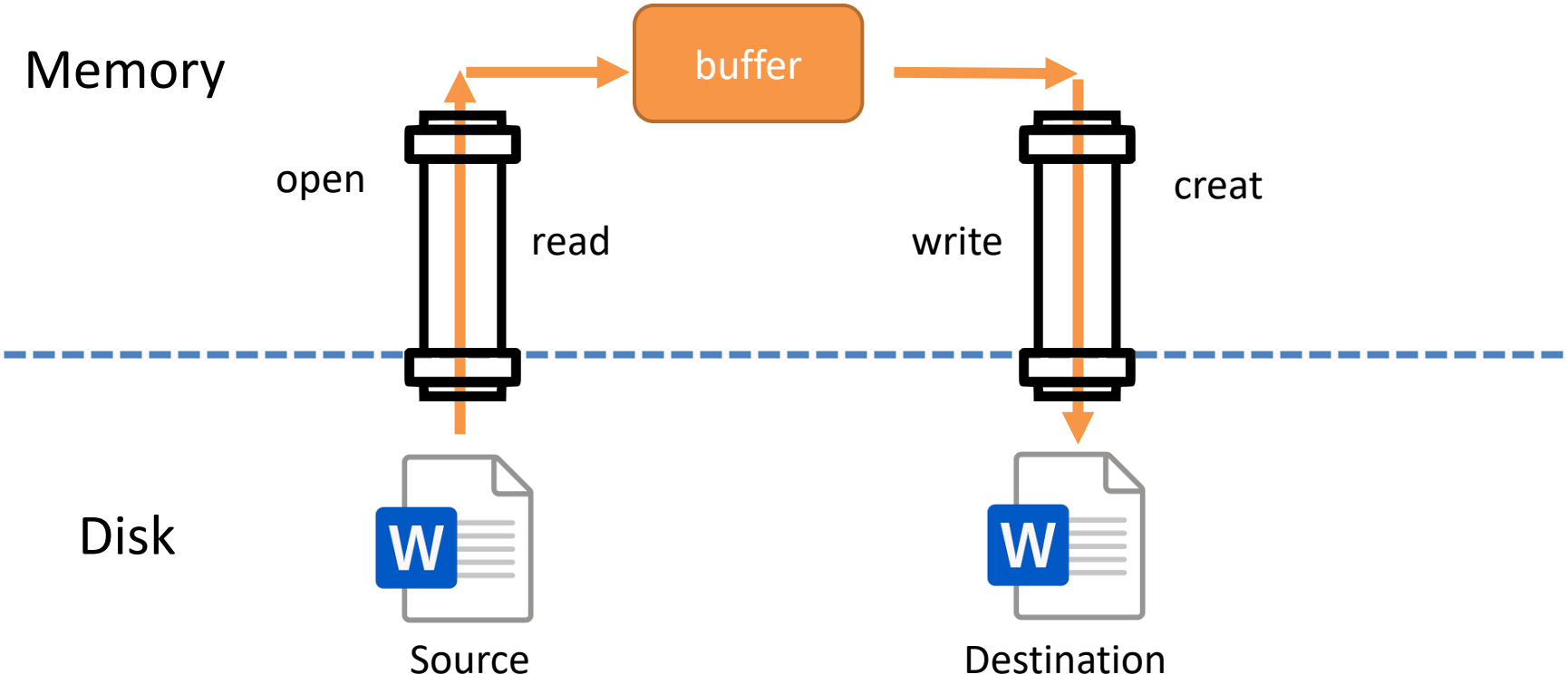
An Example Program Using File System Calls (cont.)



An Example Program Using File System Calls (cont.)

```
        /* Open the input file and create the output file */
        in_fd = open(argv[1], O_RDONLY);    /* open the source file */
        if (in_fd < 0) exit(2);             /* if it cannot be opened, exit */
        out_fd = creat(argv[2], OUTPUT_MODE);  /* create the destination file */
        if (out_fd < 0) exit(3);            /* if it cannot be created, exit */

        /* Copy loop */
        while (TRUE) {
            rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
        if (rd_count <= 0) break;           /* if end of file or error, exit loop */
            wt_count = write(out_fd, buffer, rd_count); /* write data */
            if (wt_count <= 0) exit(4);     /* wt_count <= 0 is an error */
        }

        /* Close the files */
        close(in_fd);
        close(out_fd);
        if (rd_count == 0)                  /* no error on last read */
            exit(0);
        else
            exit(5);                        /* error on last read */
    }
```

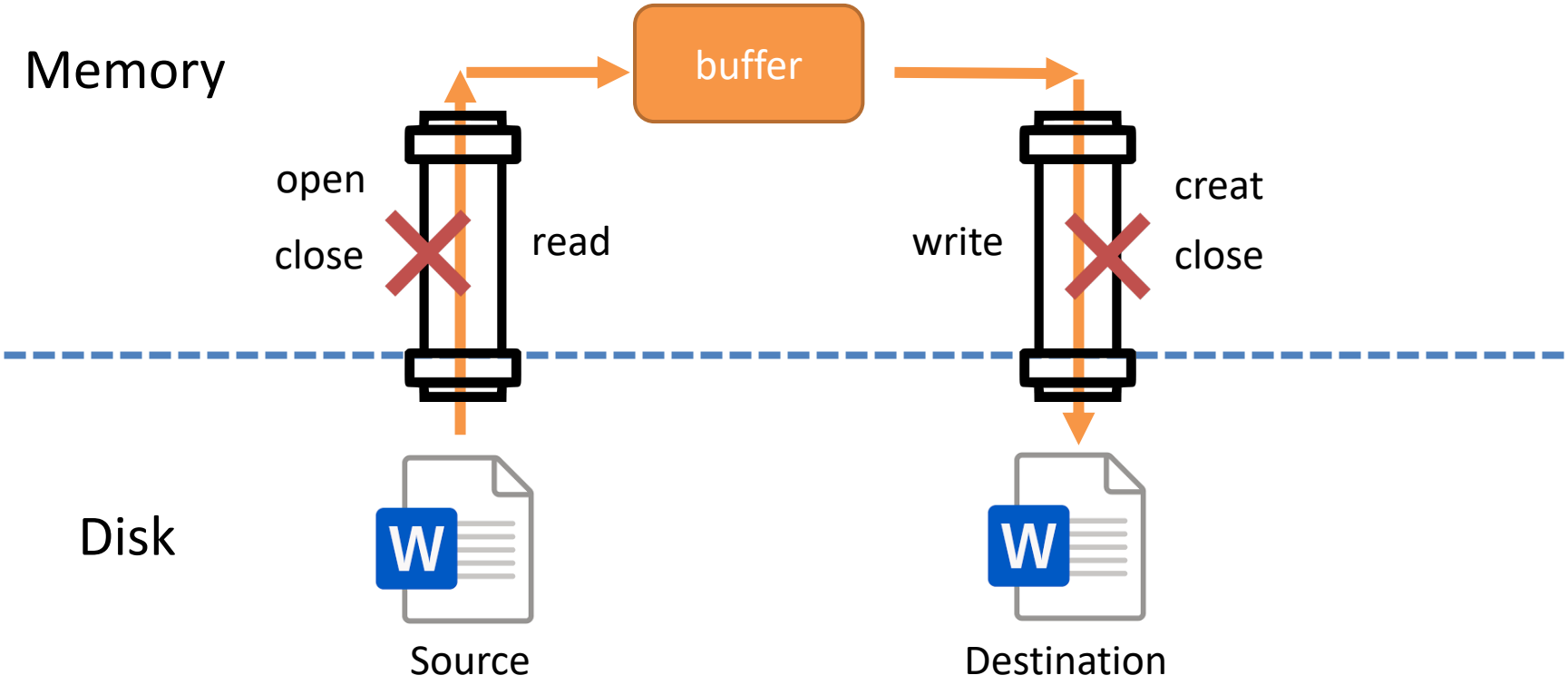
An Example Program Using File System Calls (cont.)



An Example Program Using File System Calls (cont.)

```
        /* Open the input file and create the output file */
        in_fd = open(argv[1], O_RDONLY);    /* open the source file */
        if (in_fd < 0) exit(2);             /* if it cannot be opened, exit */
        out_fd = creat(argv[2], OUTPUT_MODE);  /* create the destination file */
        if (out_fd < 0) exit(3);            /* if it cannot be created, exit */

        /* Copy loop */
        while (TRUE) {
            rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
        if (rd_count <= 0) break;           /* if end of file or error, exit loop */
            wt_count = write(out_fd, buffer, rd_count); /* write data */
            if (wt_count <= 0) exit(4);     /* wt_count <= 0 is an error */
        }

        /* Close the files */
        close(in_fd);
        close(out_fd);
        if (rd_count == 0)                  /* no error on last read */
            exit(0);
        else
            exit(5);                        /* error on last read */
    }
```

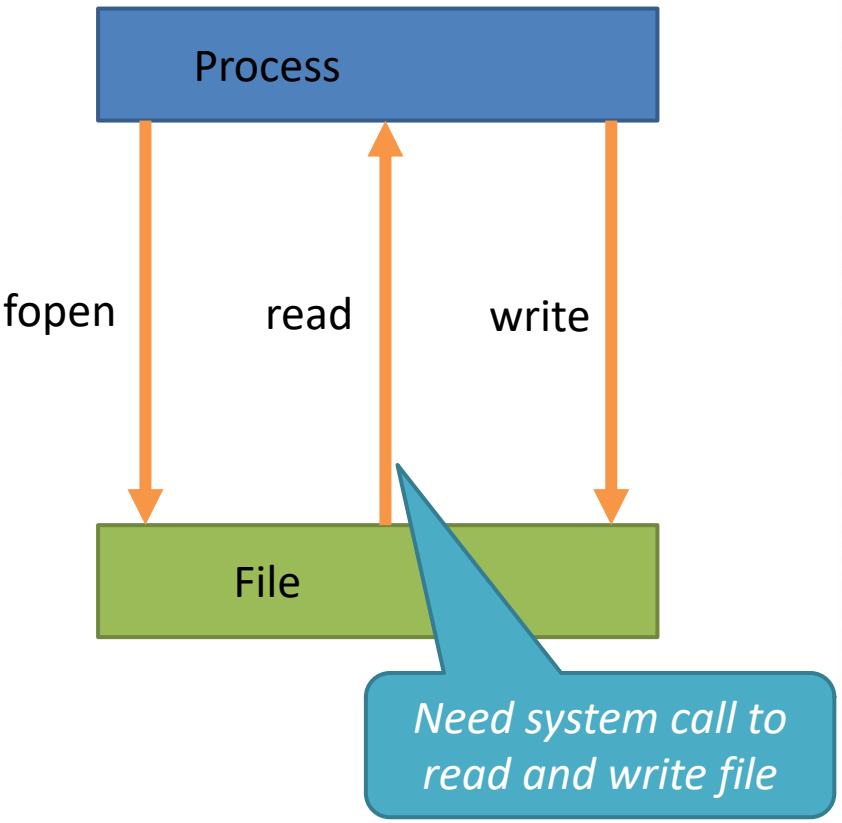
An Example Program Using File System Calls (cont.)



An Example Program Using File System Calls (cont.)

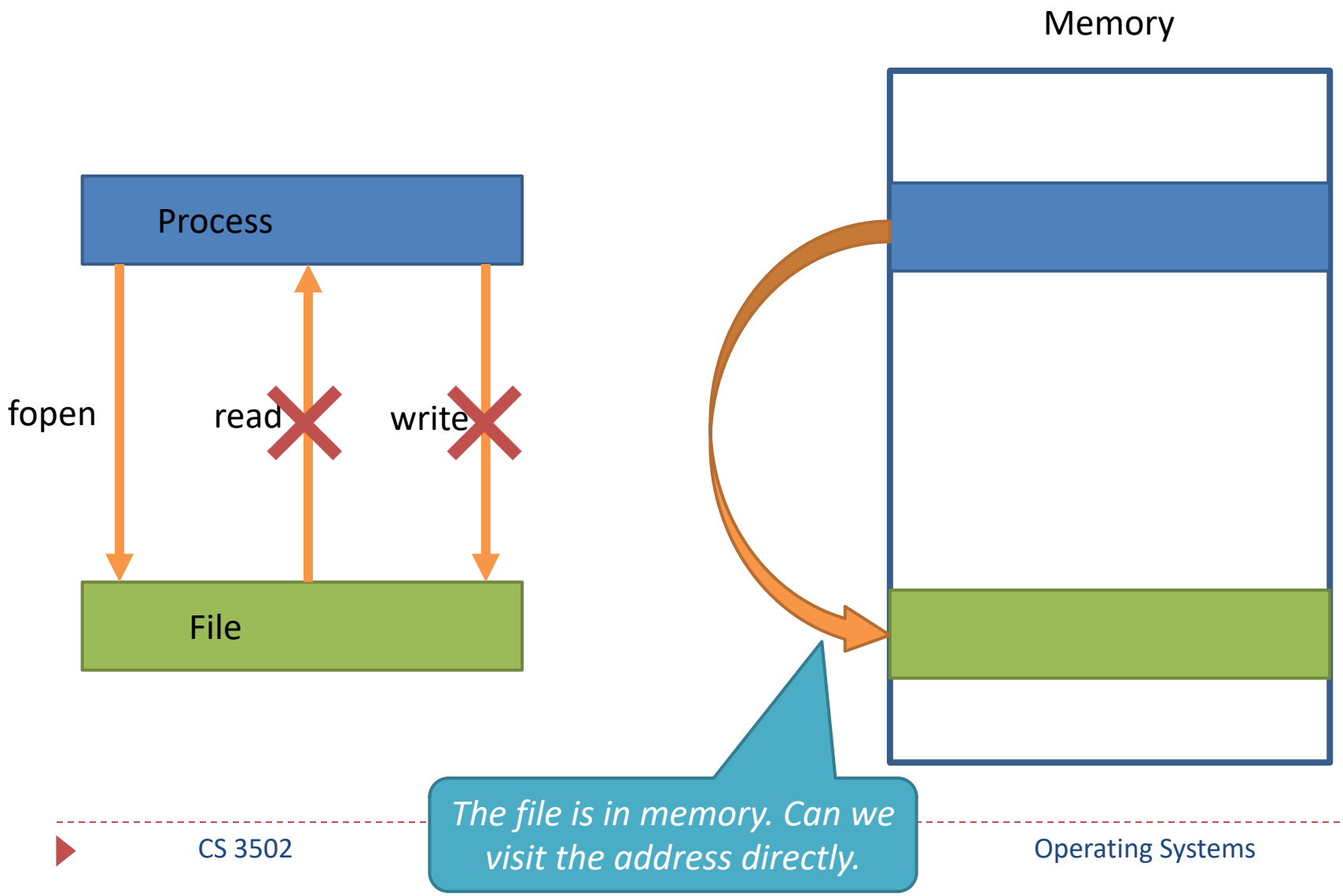
```
        /* Open the input file and create the output file */
        in_fd = open(argv[1], O_RDONLY);    /* open the source file */
        if (in_fd < 0) exit(2);             /* if it cannot be opened, exit */
        out_fd = creat(argv[2], OUTPUT_MODE);  /* create the destination file */
        if (out_fd < 0) exit(3);            /* if it cannot be created, exit */

        /* Copy loop */
        while (TRUE) {
            rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
        if (rd_count <= 0) break;           /* if end of file or error, exit loop */
            wt_count = write(out_fd, buffer, rd_count); /* write data */
            if (wt_count <= 0) exit(4);     /* wt_count <= 0 is an error */
        }

        /* Close the files */
        close(in_fd);
        close(out_fd);
        if (rd_count == 0)                  /* no error on last read */
            exit(0);
        else
            exit(5);                        /* error on last read */
    }
```

Memory-mapped Files



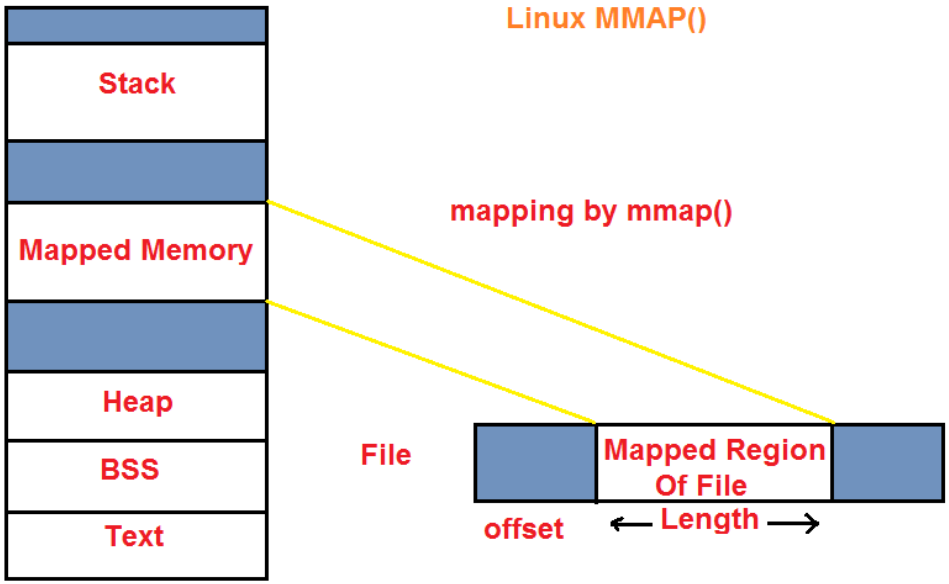

```
/* Open the input file and create the output file */
in_fd = open(argv[1], O_RDONLY);     /* open the source file */
if (in_fd < 0) exit(2);              /* if it cannot be opened, exit */
out_fd = creat(argv[2], OUTPUT_MODE);  /* create the destination file */
if (out_fd < 0) exit(3);             /* if it cannot be created, exit */

/* Copy loop */
while (TRUE) {
        rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
if (rd_count <= 0) break;            /* if end of file or error, exit loop */
        wt_count = write(out_fd, buffer, rd_count); /* write data */
        if (wt_count <= 0) exit(4);  /* wt_count <= 0 is an error */
}

/* Close the files */
close(in_fd);
close(out_fd);
if (rd_count == 0)                   /* no error on last read */
        exit(0);
else
        exit(5);                     /* error on last read */
```

Memory-mapped Files



Memory-mapped Files

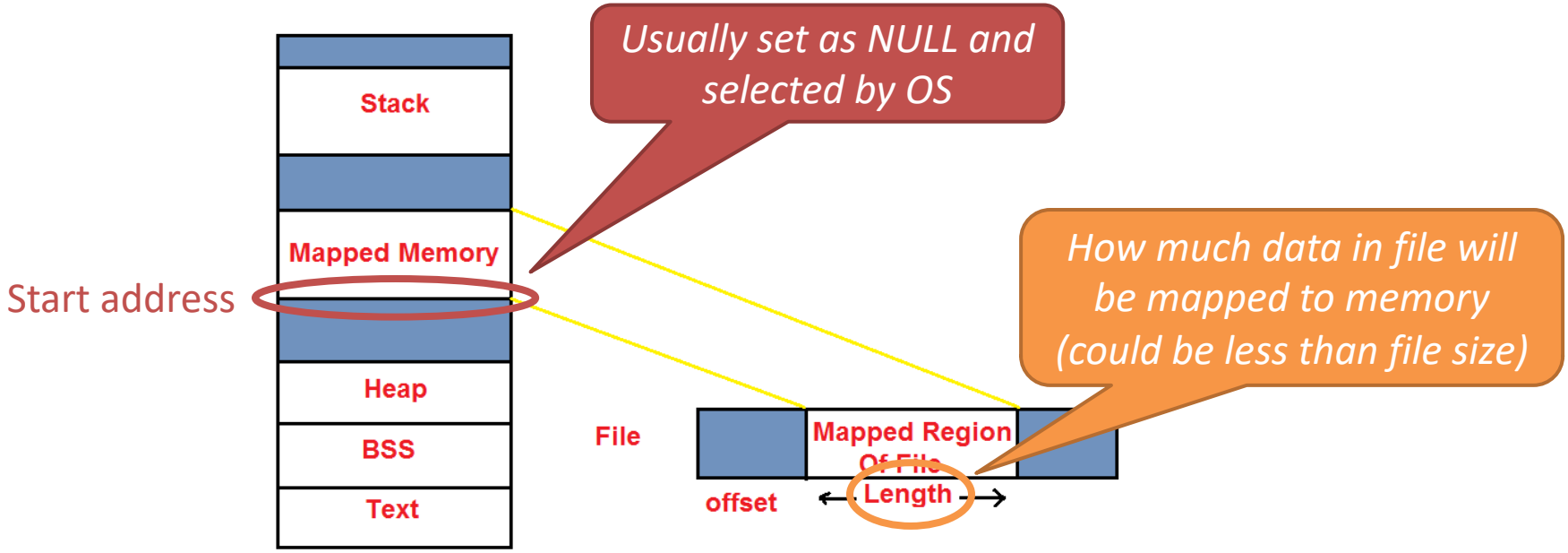
- OS provide a way (map and unmap) to map files into the address space of a running process
  - No read or write system calls are needed thereafter
- Advantages
  - Improved I/O performance and avoidance of kernel to user data copying



Memory-mapped Files

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

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

https://pubs.opengroup.org/onlinepubs/009695399/functions/mmap.html



Memory-mapped Files

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

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

https://pubs.opengroup.org/onlinepubs/009695399/functions/mmap.html

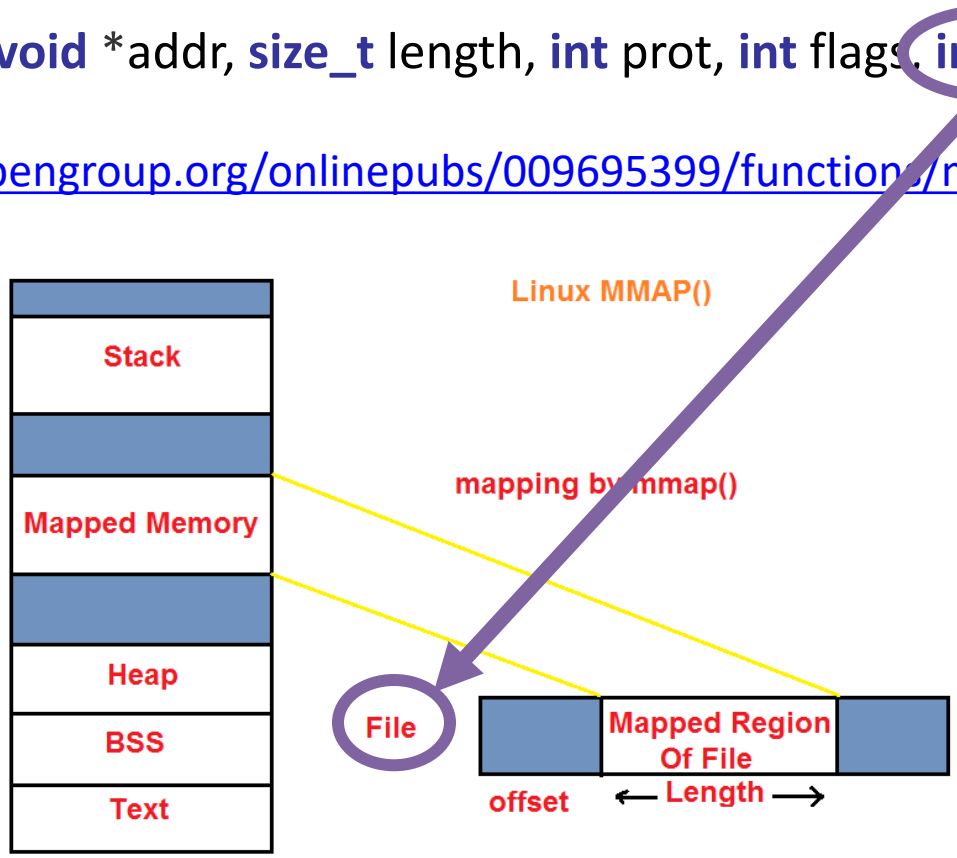
Symbolic Constant	Description
PROT_READ	Data can be read.
PROT_WRITE	Data can be written.
PROT_EXEC	Data can be executed.
PROT_NONE	Data cannot be accessed.

Symbolic Constant	Description
MAP_SHARED	Changes are shared.
MAP_PRIVATE	Changes are private.
MAP_FIXED	Interpret *addr* exactly.

Memory-mapped Files

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

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

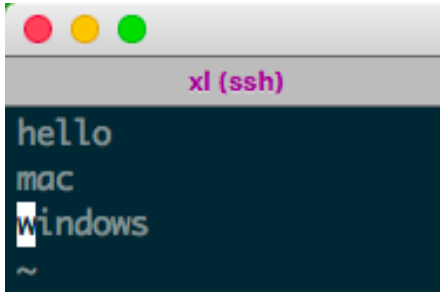
https://pubs.opengroup.org/onlinepubs/009695399/functions/mmap.html



Memory-mapped File Example

```
#include <sys/mman.h> /* for mmap and munmap */
#include <sys/types.h> /* for open */
#include <sys/stat.h> /* for open */
#include <fcntl.h> /* for open */
#include <unistd.h> /* for lseek and write */
#include <stdio.h>

int main(int argc, char **argv)
{
        int fd;
        char *mapped_mem, * p;
        int flength = 1024;
        void * start_addr = 0;

        fd = open(argv[1], O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
        flength = lseek(fd, 1, SEEK_END);
        lseek(fd, 0, SEEK_SET);

        mapped_mem = mmap(start_addr, flength, PROT_READ, MAP_PRIVATE, fd, 0);

        printf("%s\n", mapped_mem);
        close(fd);
        munmap(mapped_mem, flength);
        return 0;
}
```

text.txt



Return the mapped memory address

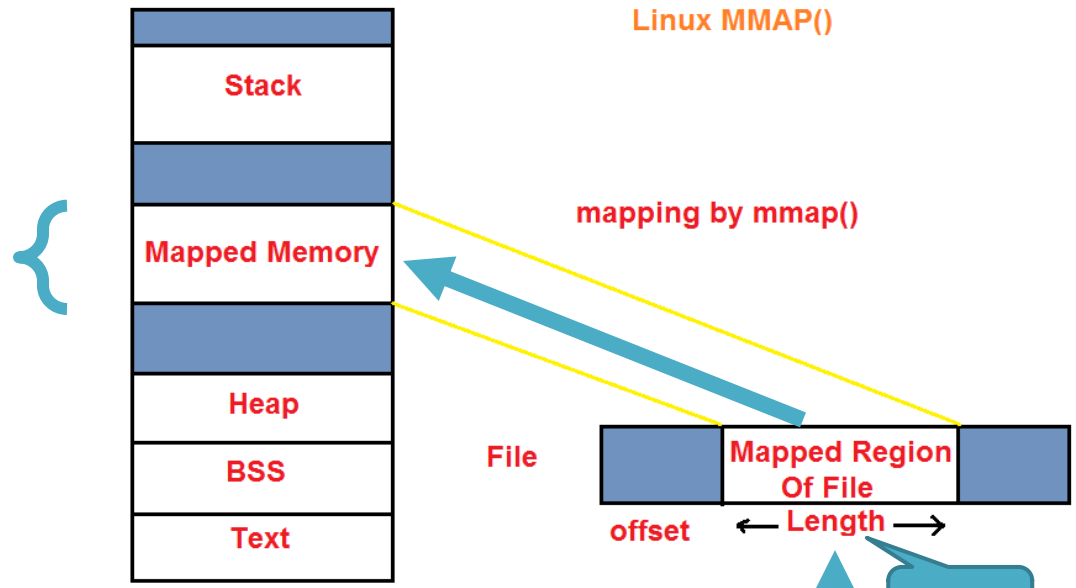
Allow read

Set private, do not allow other process to read

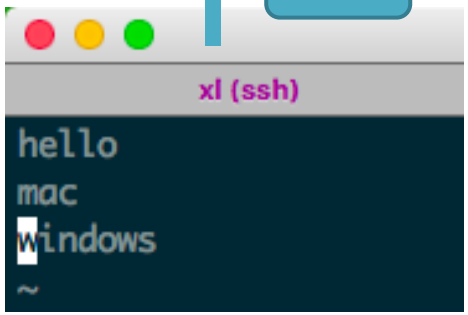
Print out the data in the memory

Memory-mapped File Example

Print out the data in this memory area



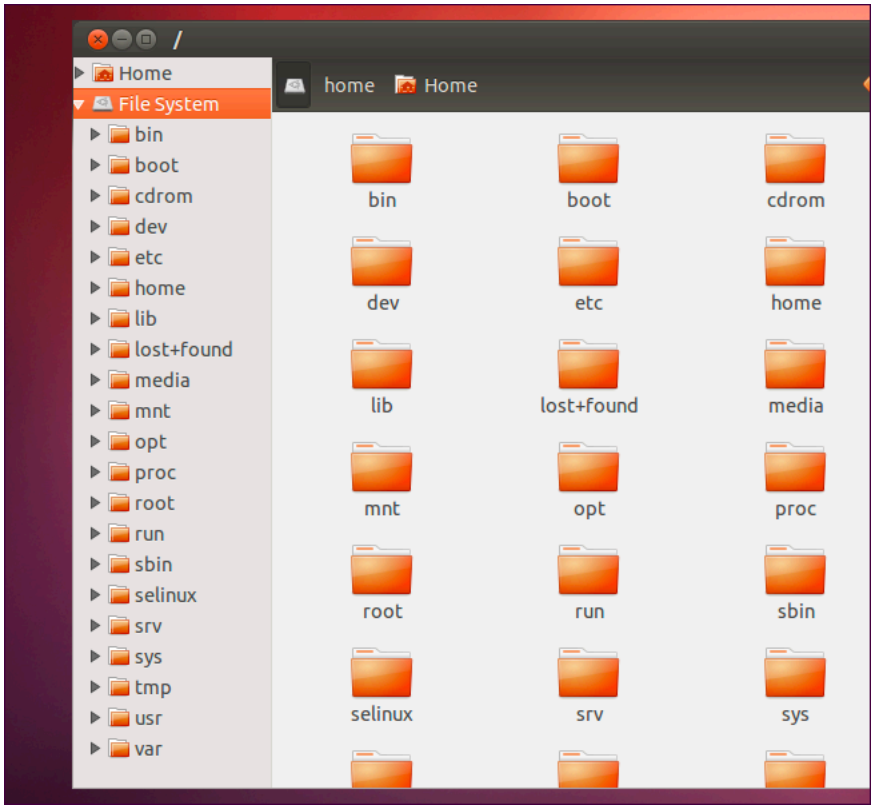

```
ksuo@centos65-pv-3 mymmap$ ./a.out text.txt
hello
mac
windows

ksuo@centos65-pv-3 mymmap$
```



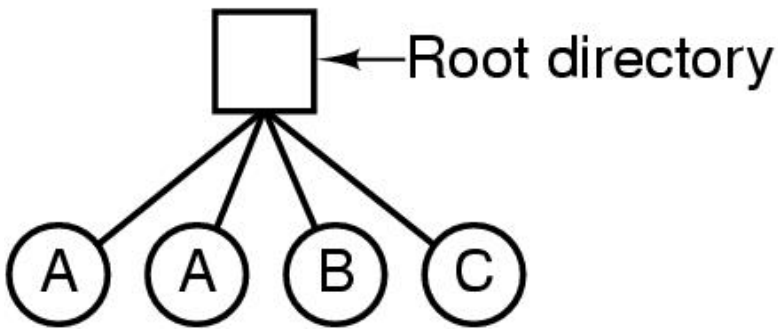
text.txt

Directory

- A directory is a file the solo job of which is to store the file names and the related information.
  - All the files, whether ordinary, special, or directory, are contained in directories.
  - The directory structure is organized as a tree and the root is root directory /



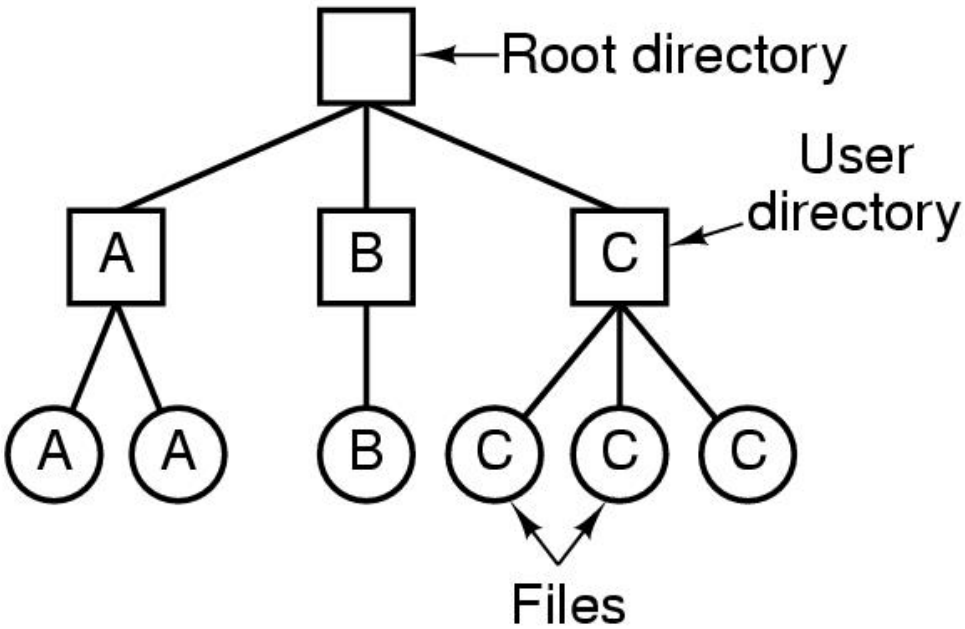
Single-Level Directory



- A single-level directory system is simple for implementation
  - contains 4 files
  - owned by 3 different people, A, B, and C

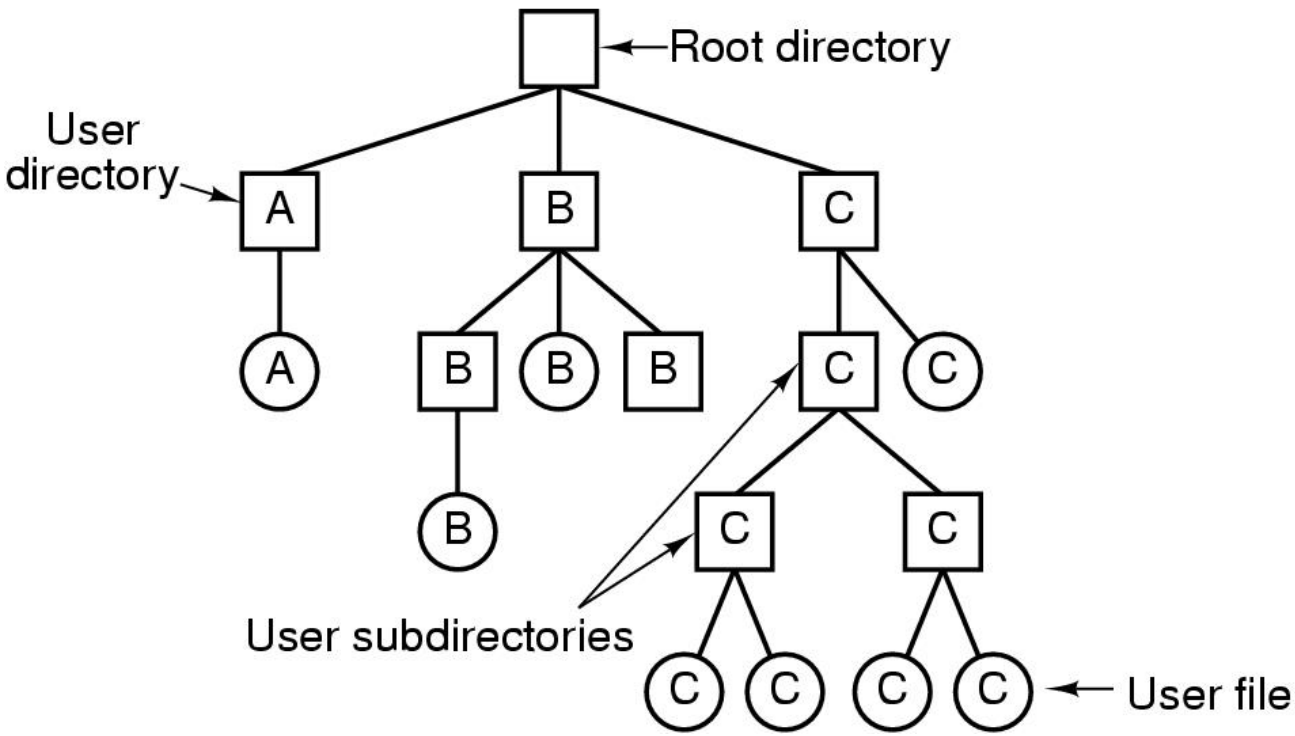
- Pros: simplicity, ability to quickly locate files
- Cons: inconvenient naming, only for single user; different users may use the same names for their files

Two-level Directory



- It requires login procedure, not allow a user to access other users' files
- Solves name collision, but no solve if user has lots of files and wants to group them in a logical way

Hierarchical Directory Systems

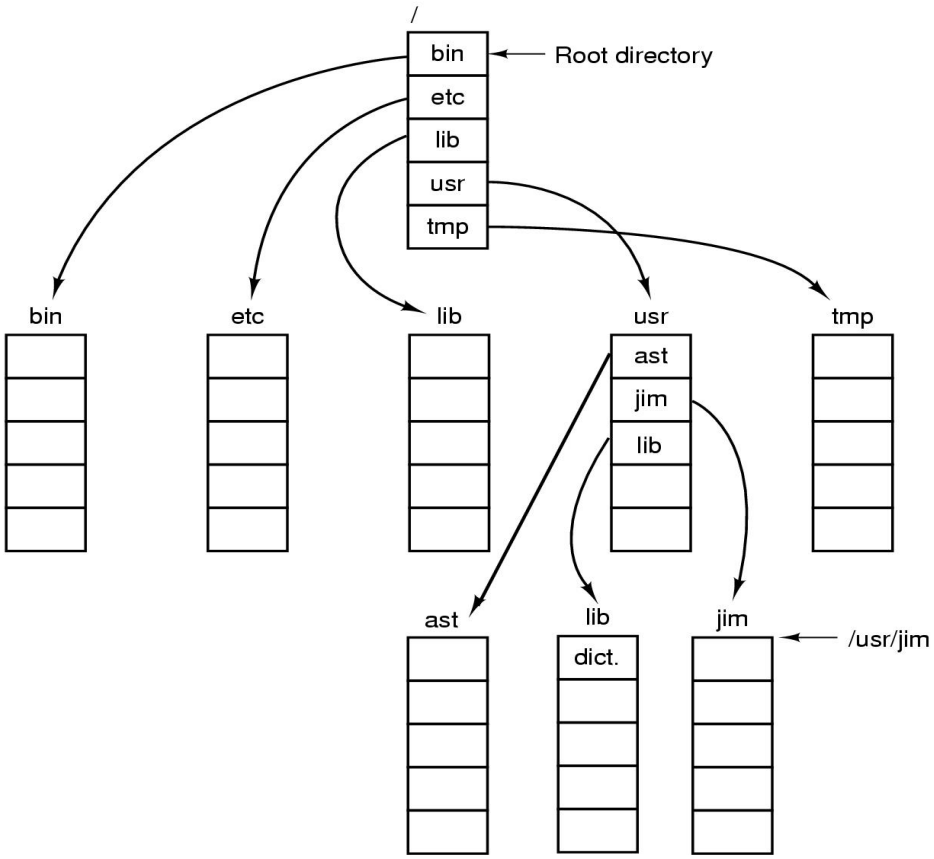


- Directory is now a tree of arbitrary height
  - Directory contains files and subdirectories

Path Names

- To access a file, the user should either:
  - Go to the directory where file resides, or
  - Specify the path where the file is
- Absolute: path of file from the root directory
  - cd /usr/data/1.txt
- Relative: path from the current working directory
  - cd ~/1.txt

A UNIX directory tree



Directory Operations

1. Create
2. Delete
3. Opendir
4. Closedir
5. Readdir
6. Rename
7. Link
8. Unlink

Directory Related System Call

MKDIR	83	Create a directory
MKDIRAT	258	Create a directory relative to a directory file descriptor
RMDIR	84	Delete a directory
GETCWD	79	Get current working directory
CHDIR	80	Change working directory
FCHDIR	81	Change working directory
CHROOT	161	Change root directory
GETDENTS	78	Get directory entries
GETDENTS64	217	Get directory entries
LOOKUP_DCOOKIE	212	Return a directory entry's path

Directory Operation Examples

- pwd: show current directory

```
ksuo@Kevins-MacBook-Pro-2017 ~/Desktop> pwd
/Users/ksuo/Desktop
```

- cd: go to next level directory

```
[root@localhost ~]# cd /usr/local/
[root@localhost local]# pwd
/usr/local
```

- cd .. : go to last level directory

```
[root@localhost ~]# cd /usr/local/lib/
[root@localhost lib]# pwd
/usr/local/lib
[root@localhost lib]# cd ./
[root@localhost lib]# pwd
/usr/local/lib
[root@localhost lib]# cd ../
[root@localhost local]# pwd
/usr/local
```

Directory Operation Examples

- mkdir: create a new directory

```
[root@localhost ~]# mkdir /tmp/test/123
mkdir: cannot create directory `/tmp/test/123': No such file or directory
[root@localhost ~]# ls /tmp/test
ls: /tmp/test: No such file or directory
```

- rmdir: remove one directory

```
[root@localhost ~]# ls /tmp/test/
123
[root@localhost ~]# rmdir /tmp/test/123/
[root@localhost ~]# ls /tmp/test/
[root@localhost ~]#
```

Conclusion

- File: abstraction of storage
  - Name, structure, type
  - Access, attribute, operation
- Memory-mapped files
- Directory: a file to organize files
  - Single-level, two-level, hierarchical
  - Path, operation