Operating Systems

Lecture 2.4

Linux Special Files and Terminal Drivers

Instructor: Muhammad Arif Butt, PhD

Lecture Agenda

- Overview of Linux Special Files
- Device Files vs Regular files
- Character and Block Special Files
- Creating your own Special Files
- Overview of Terminal Devices
- Writing to someone else Terminal
- Reading from someone else Terminal
- What are Terminal Attributes
- Categories of TTY Driver Processing
- Canonical vs Non-Canonical Input Modes
- Accessing and Modifying Terminal Attributes
- Programmatically Setting Terminal Attributes

Linux Special Files

What are Special Files in Linux?

- Special files are a fundamental concept in Unix/Linux systems that provide interfaces to hardware devices and kernel services. They are located inside /dev/ directory.

```
crw-rw-rw- 1 root root 1, 3 date /dev/null
brw-rw---- 1 root disk 8, 0 date /dev/sda
```

  - **Character Special Files** represents hardware devices that reads or writes a serial stream of data bytes connected via serial/parallel port. Examples of such devices are terminal devices, sound cards, and tape drives
  - **Block Special Files** represents hardware devices that reads or writes data in fixed size blocks (buffered devices), and unlike serial devices they provide random access to data stored on the device. Examples of such devices are HDD, SSD, USB and cdrom

Character Special Files:

- ➢ Sequential access (one character at a time)
- ➢ Stream oriented (Data flows as a continuous stream rather than discrete blocks)
- ➢ Used for devices like keyboard, mouse, serial ports, sound cards
- ➢ No buffering by the kernel

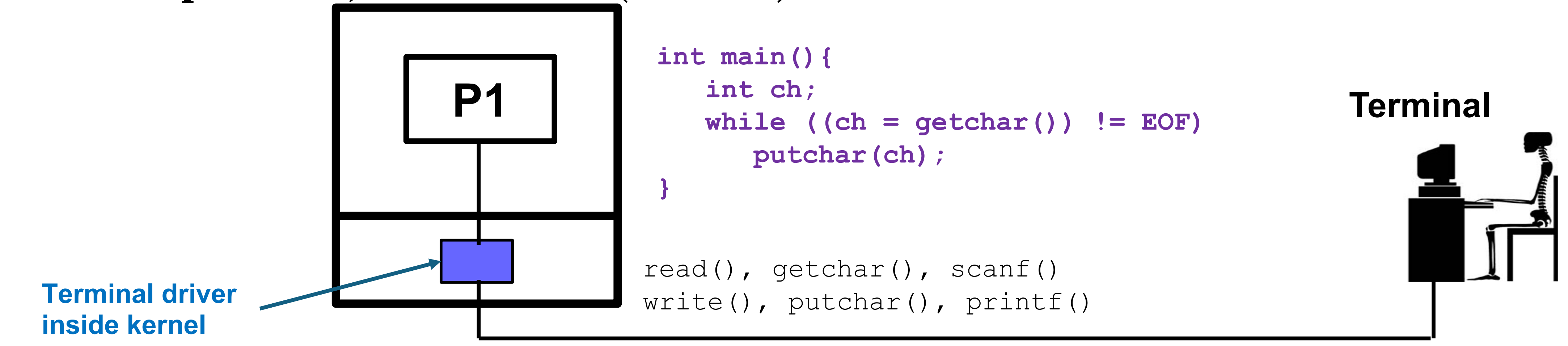
Block Special Files:

- ➢ Random access (one block 512B or 4 KiB at a time)
- ➢ Block oriented (Data transfer occur in block-sized chunks rather than individual bytes)
- ➢ Used for storage devices like hard drives, USB drives, SSDs
- ➢ Kernel provides buffering and caching

Block devices are ideal for storage systems (SSDs, USB drives), while character devices are suited for communication and streaming interfaces

Overview of Device Files

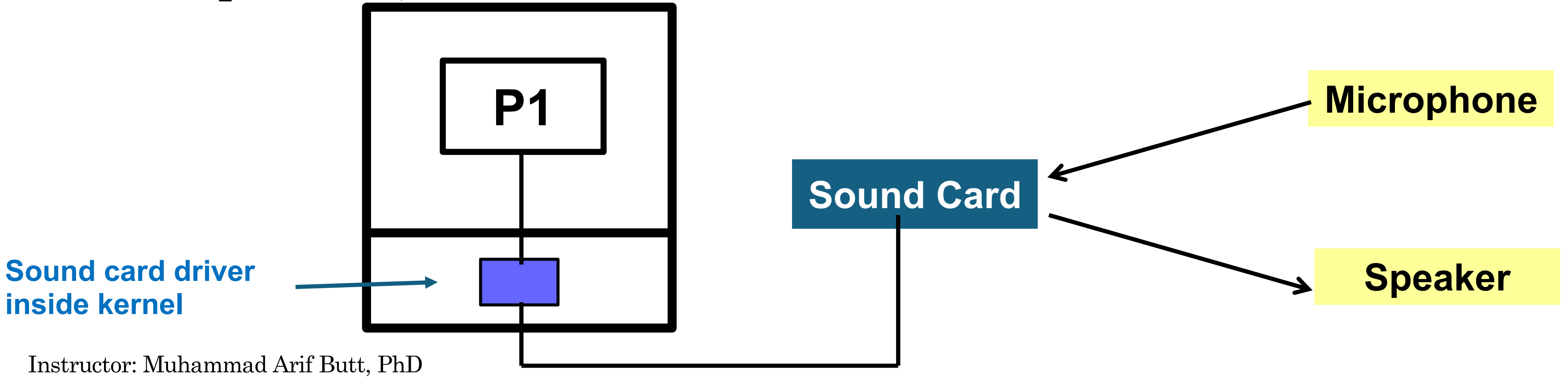
For a process, a terminal (kb/vdo)is a source/destination of data



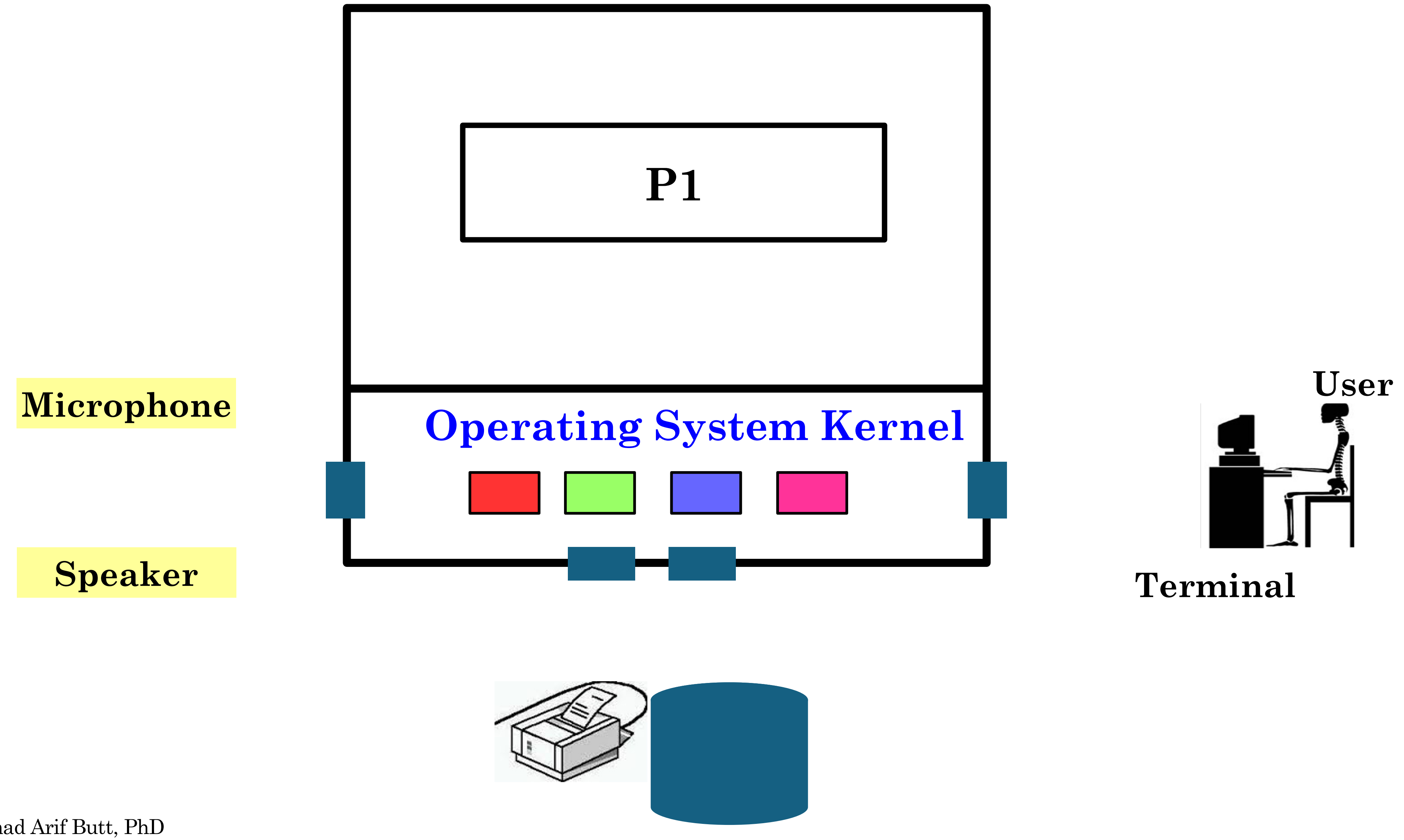
```
int main(){
    int ch;
    while ((ch = getchar()) != EOF)
        putchar(ch);
}
```

```
read(), getchar(), scanf()
write(), putchar(), printf()
```

For a process, a sound card is a source/destination of data



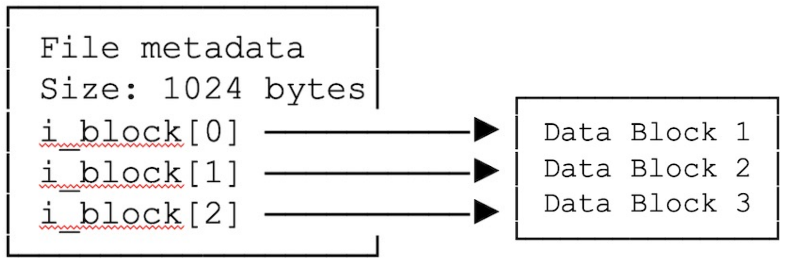
Overview of Devices Files (cont...)



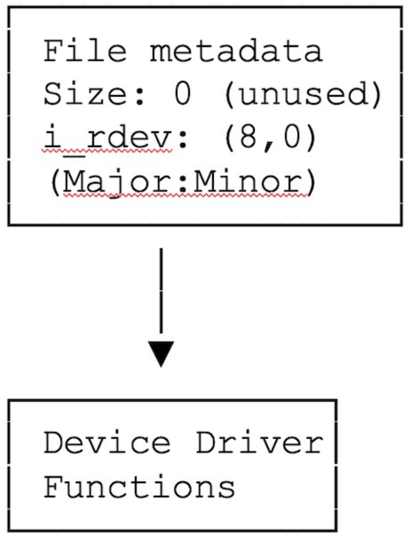
Device Files vs Regular Files

```
# Regular File
-rw-r--r-- 1 user user 1048576 Aug 14 10:20 /home/user/document.txt
|                      |       |
|                      |       └─ Timestamp
|                      └─ File size (1MB)
└─ File type and permissions (- = regular file)

# Character Special File
crw-rw-rw- 1 root root 1,3 Aug 14 10:20 /dev/null
|                      | |
|                      | └─ Minor number (3)
|                      └─ Major number (1)
└─ File type (c = character device)

# Block Special File
brw-rw---- 1 root disk 8,0 Jul 28 19:40 /dev/sda
|                      | |
|                      | └─ Minor number (0)
|                      └─ Major number (8)
└─ File type (b = block device)
```

Regular File Inode:



Device File Inode:



Device Files vs Regular Files (cont…)

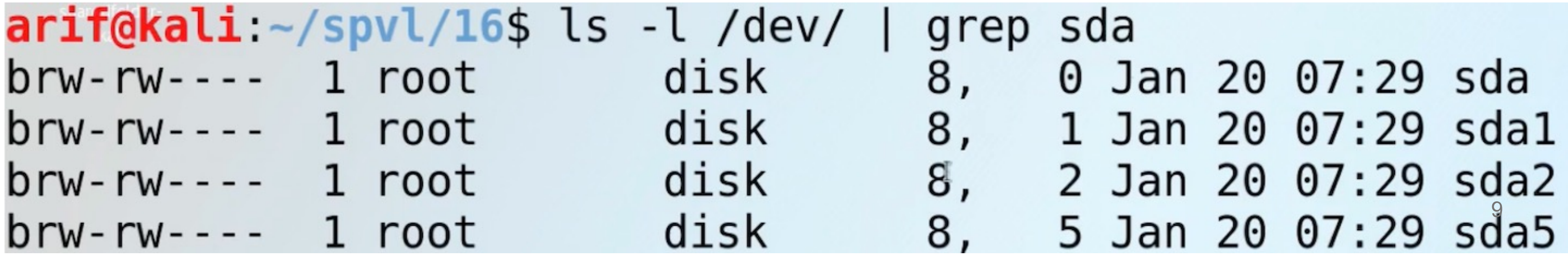
- A regular file is a container, while a device file is a connection.
- The inode block of a regular file contains pointer that points to its data blocks, while the inode block of a device file contains pointer that points to a function inside the kernel called the device driver.
- When you see the long listing, a regular file shows its size while a device file displays the major and minor number of the device driver at the place of size when you see its long listing.

The Linux kernel maintains separate tables for character and block devices, indexed by major numbers. Each table entry points to a device structure (like struct cdev for character devices) which contains a file operations table with function pointers to the actual device driver functions. The minor number is passed to these driver functions to identify the specific device instance.
Device files can be copied, moved, and removed like regular files since they are just filesystem entries containing major:minor numbers. Copying creates a new entry with identical device numbers. Removing a device file only removes the filesystem entry and does not affect the device driver or the actual hardware device."

Device Numbers

Every special file has two important numbers (instead of file size) associated with it.

- The major number (8 bits) identifies the device driver
  - 1 = Memory devices (`/dev/null, /dev/zero, /dev/random`)
  - 3 = IDE hard disk (first controller)
  - 4 = TTY devices (`/dev/tty0, /dev/ttyS0`)
  - 8 = SCSI disk devices
- The minor number (8 bits) identifies the specific device instance withing a driver
  - `/dev/sda` = (8,0) - First SCSI disk
  - `/dev/sda1` = (8,1) - First partition of first SCSI disk
  - `/dev/sda2` = (8,2) - Second partition of first SCSI disk
  - `/dev/sdb` = (8,16) - Second SCSI disk
  - `/dev/sdb1` = (8,17) - First partition of second SCSI disk



```
arif@kali:~/spvl/16$ ls -l /dev/ | grep sda
brw-rw----   1 root      disk     8,   0 Jan 20 07:29 sda
brw-rw----   1 root      disk     8,   1 Jan 20 07:29 sda1
brw-rw----   1 root      disk     8,   2 Jan 20 07:29 sda2
brw-rw----   1 root      disk     8,   5 Jan 20 07:29 sda5
```

Character Special vs Block Special Files

1. **Data Access Method:**
   - *Character Files:* Data is accessed sequentially, one character (byte) at a time
   - *Block Files:* Data is accessed in fixed-size blocks (typically 512 bytes, 1KB)
2. **Buffering Strategy:**
   - *Character Files:* Unbuffered or minimally buffered
   - *Block Files:* Heavily buffered through the kernel's buffer cache for performance optimization
3. **Random Access Capability:**
   - *Character Files:* Typically sequential access only (tape drives, serial ports)
   - *Block Files:* Support random access, can seek to any position (hard disks, ssd)
4. **I/O Operations:**
   - *Character Files:* Use read/write system calls directly
   - *Block Files:* Can use both direct I/O and memory mapped I/O through kernel's page cache

Character Special vs Block Special Files (cont...)

Examples of Character Special Files:

- **`/dev/null`** accepts and discards all input, returns EOF on reads (the "black hole" device)
- **`/dev/zero`** provides infinite stream of null bytes (0x00) for creating empty files or clearing memory
- **`/dev/random`** generates cryptographically secure random numbers
- **`/dev/tty`** represents the controlling terminal for the current process
- **`/dev/pts/0`** represents the controlling terminal for the current process

Examples of Block Special Files:

- **`/dev/sda`** primary storage devices like SATA/SCSI hard drives and SSDs
- **`/dev/hda`** primary storage devices like IDE hard drives and SSDs
- **`/dev/loop0`** loopback devices that make regular files accessible as block devices for mounting
- **`/dev/nvme0n1`** Non-Volatile-Memory-Express solid-state drives (0n1 specify controller and namespace)
- **`/dev/mmcblk0`** Multi-Media-Card block storage device for (SD cards)

Creating Special Files

- In Linux, the `mknod` command is used to create special files. The command takes four arguments:

```
mknod <filename> <type> <major> <minor>
```

  - The `<type>` specifies the file type (character special [c], block special [b], named pipe [p])
  - The major number identifies the device driver while the minor number identifies the specific device instance used by the driver to distinguish between multiple devices

```
stat /dev/sda /dev/null                [display device numbers of devices]
file /dev/sda /dev/null                [show device type]
lsblk                                  [show all block devices]
sudo mknod /tmp/mynull c 1 3           [create character device]
sudo mknod /tmp/mydisk b 8 0           [create block device]
ls -l /tmp/mynull /tmp/mydisk          [verify]
```

- **Important Notes:**
  - The Linux kernel maintains separate tables for character and block devices, indexed by major numbers and each table entry points to the device driver function of that specific device. The minor number is passed to the device driver function to differentiate the device instance.
  - Creating a device file doesn't create the actual device or driver
  - Device files are just interfaces, the real functionality comes from kernel drivers

Demonstration

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Terminal Devices

Terminal Devices

- Terminal Devices are character special files that provide text I/O interfaces. Like other device files, they are located inside `/dev/` directory, and allow communication between users and the Linux system.

```
$ ls -l /dev/tty3 /dev/pts/1
crw------- 1 root tty   4,3   Aug 14 10:20 /dev/tty3
crw--w---- 1 user tty   136,1   Aug 14 10:25 /dev/pts/1
```

- Both are character devices (start with c), having different major numbers: 4 (tty) vs 136 (pts).
- Every terminal that we open on our system has an associated character special file. To check the terminal you are currently on, use the following command:

```
$ tty
```

Terminal Devices (cont...)

- Controlling Terminals are text mode only devices (**/dev/ttyN**) have direct hardware access. Can be accessed via Ctro+Alt+F1 through F6. They have a major number of 4.
- Pseudo-terminals are virtual terminals **(/dev/pts/N**) used inside GUI terminals. Created dynamically when terminal emulator opens. Has a major number of 136.
- The **/dev/tty** is a dynamic alias that always refers to the controlling terminal of the current process. Always resolves to the real TTY or PTS device.
- You can write to a specific terminal, provided you have write permissions:

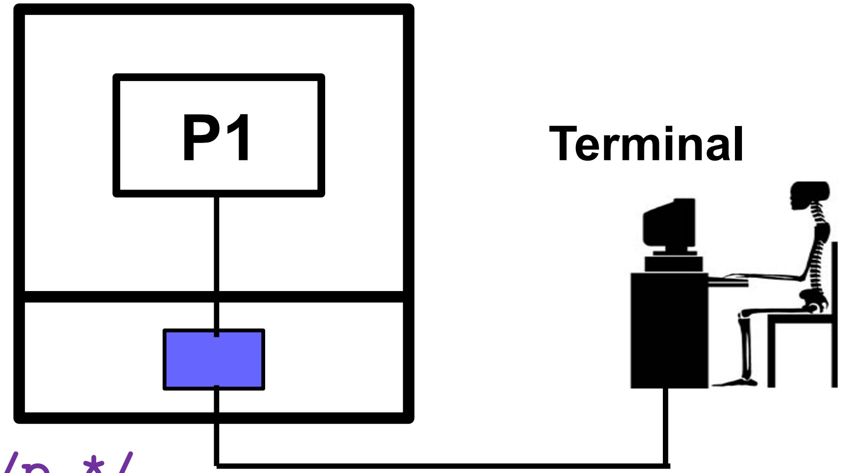
```
$ echo "Hello students" > /dev/pts/1
$ cp friends.txt /dev/pts/1
```

- You can read from a specific terminal, provided you have read permissions:

```
$ read input < /dev/tty
$ echo "You entered: $input”
```

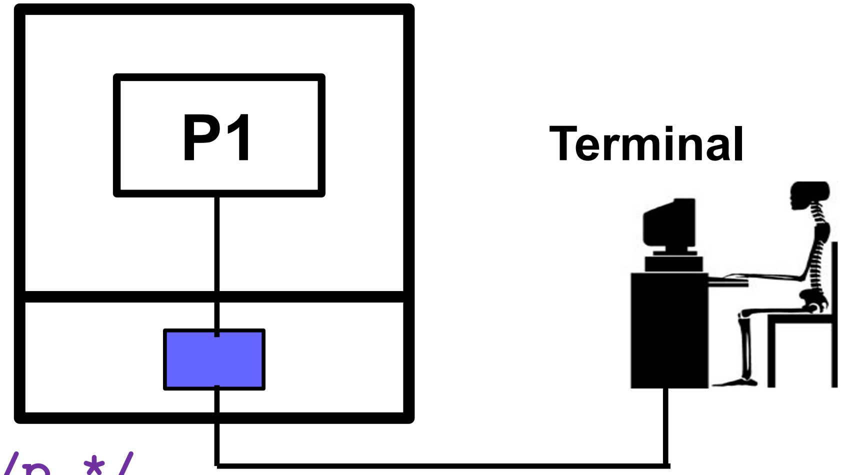
Writing on someone else Terminal

```
// A terminal is just a file supporting which you can read and write regular i/o
// usage: $./mywrite /dev/pts/N

int main(int argc, char *argv[]){
   if (argc != 2 ){
       fprintf(stderr,"usage: ./a.out ttyname\n");
       exit(1);
   }
   int fd = open(argv[1], 1); /* open terminal file for o/p */

   char  buf[512];  /* loop until EOF on input */
   while(fgets(buf, 512, stdin) != NULL ) /*read keyboard of character special file*/
      if (write(fd, buf, strlen(buf)) == -1 ) /*write vdu of character special file*/
          break;
   close(fd); /*close the file once you are out of loop*/
   return 0;
}
```

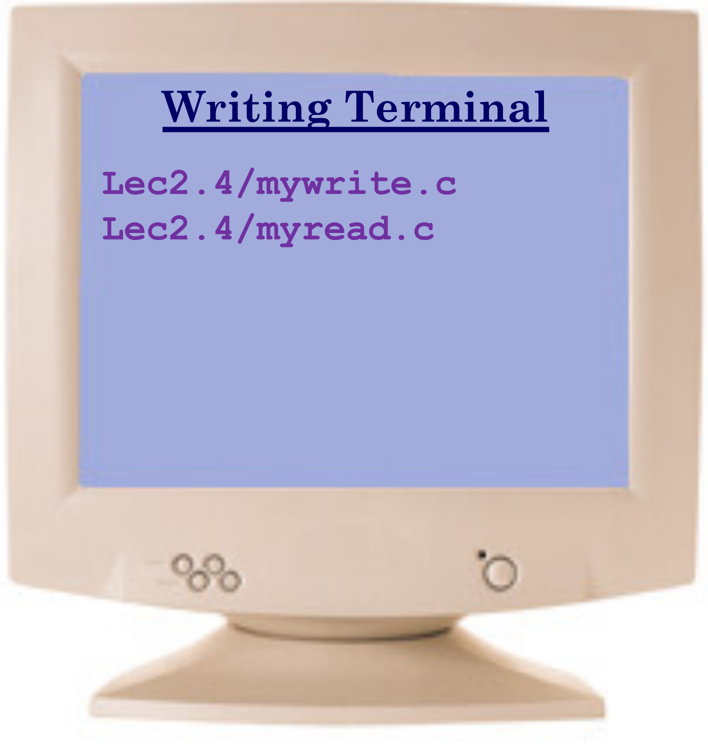


Reading from someone else Terminal

```
// A terminal is just a file supporting which you can read and write regular i/o
// usage: $./myread /dev/pts/N

int main(int argc, char *argv[]){
   if (argc != 2 ){
       fprintf(stderr,"usage: ./a.out ttyname\n");
       exit(1);
   }
   int fd = open(argv[1], 0); /* open terminal file for i/p */
   char  buf[512];  /* loop until EOF on input */
   while((n=read(fd, buf, sizeof(buf)-1))>0){
       buf[n] = '\0';
       printf("%s", buf);
       fflush(stdout);
   }
   close(fd); /*close the file once you are out of loop*/
   return 0;
}
```



Demonstration



Writing Terminal

```
Lec2.4/mywrite.c
Lec2.4/myread.c
```

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Accessing and Modifying Terminal Attributes

What are Terminal Attributes?

- Linux terminal attributes are configuration settings that control how the TTY driver processes input and output data flowing between the keyboard, programs, and vdu.
- The **`stty`** command is a Linux utility that displays and modifies terminal line settings and attributes for the current TTY device. The attributes can have thre types of values: Numerical, Character and Boolean values.

```
$ stty -a              # Show all settings in readable format
$ stty -g              # Show settings in stty-readable format
$ stty sane             # Reset to sensible defaults
```

Four Categories of TTY Driver Processing

- **Input Processing** transforms keyboard characters before they reach the process, for example `icrnl` converts carriage return (Enter key) to newline character.
- **Output Processing** transforms characters from the process before displaying them on screen, for example `onlcr` converts newline characters to carriage return + newline.
- **Control Processing** defines how characters are transmitted and represented, for example `cs8` sets 8-bit character size for data transmission.
- **Local Processing** controls how the TTY driver handles characters while they remain buffered in the driver, for example `icanon` enables line-by-line input and `echo` displays typed characters on screen

Canonical vs Non-Canonical Input Modes

Canonical Mode (Default Mode)

- Line-buffered input: Data processed line by line
- Enter key required: Input available only after pressing Enter
- Line editing enabled: Backspace, delete, Ctrl+U work
- Buffering location: Inside the TTY driver program

Non-Canonical Mode

- Character by character input: Data available immediately
- No Enter key required: Each key press is processed instantly
- Line editing disabled: No backspace or editing word
- No Buffering: Direct character by character processing

```
$ stty -a | grep icanon
```

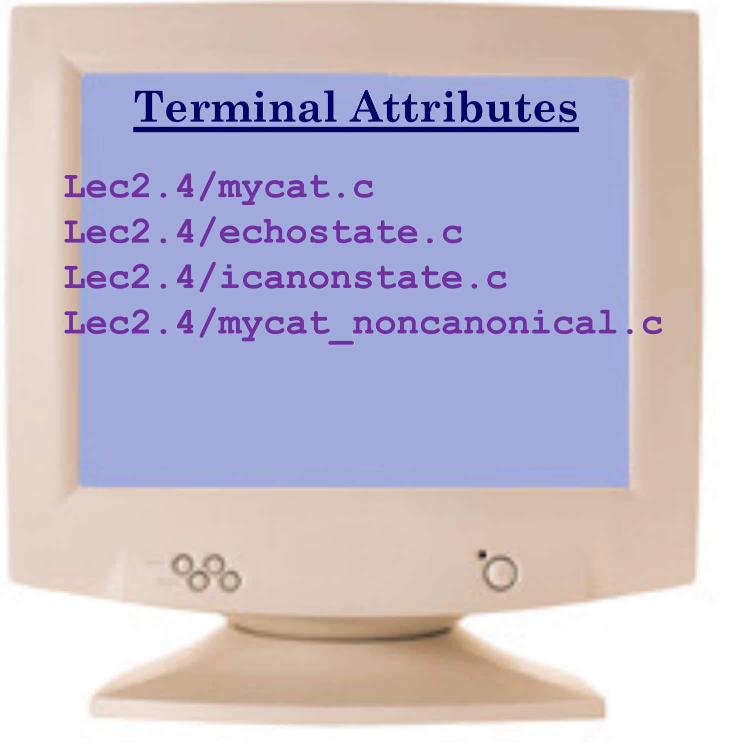
- Enable canonical mode, and understand by running the `cat` command:

```
    $ stty icanon
    $ cat
```

- Disable canonical mode, and understand by running the `cat` command:

```
    $ stty -icanon
    $ cat
```

Demonstration



Terminal Attributes

```
Lec2.4/mycat.c
Lec2.4/echostate.c
Lec2.4/icanonstate.c
Lec2.4/mycat_noncanonical.c
```

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Programmatically Setting Terminal

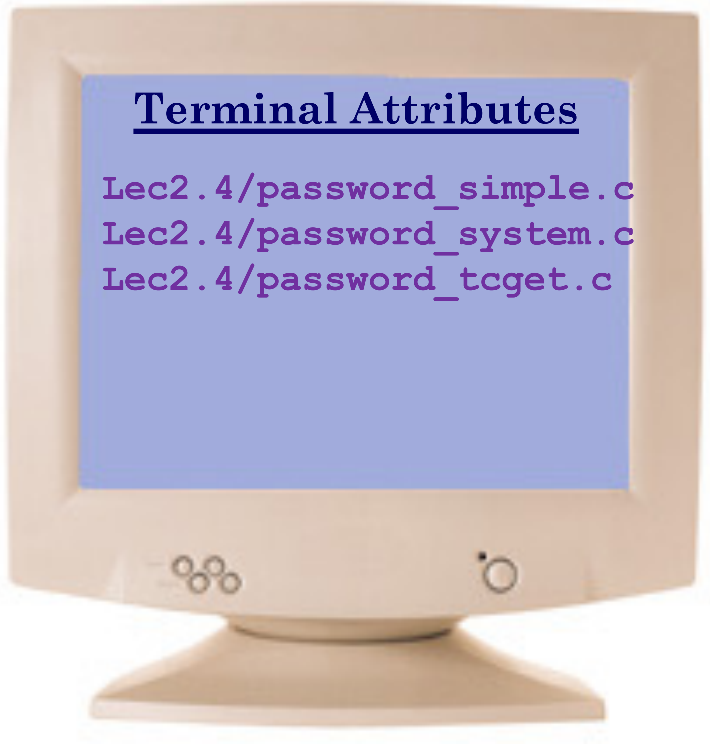
Three ways to get/set the attributes of terminal driver from a C program:

- Use `system()` library call
- Use `tcgetattr()` and `tcsetattr()` library calls
- Use `ioctl()` system call

Three steps to change the attributes of a terminal driver:

- Get the attributes from the driver
- Modify the attribute(s) you need to change
- Send these revised attributes back to the driver

Demonstration



Terminal Attributes

```
Lec2.4/password_simple.c
Lec2.4/password_system.c
Lec2.4/password_tcget.c
```

GitHub Code Repository Link: https://github.com/arifpucit/OS-Codes

Bonus
Non-Blocking I/O Models

UNIX I/O Models

There exist five different I/O models in all UNIX based systems:

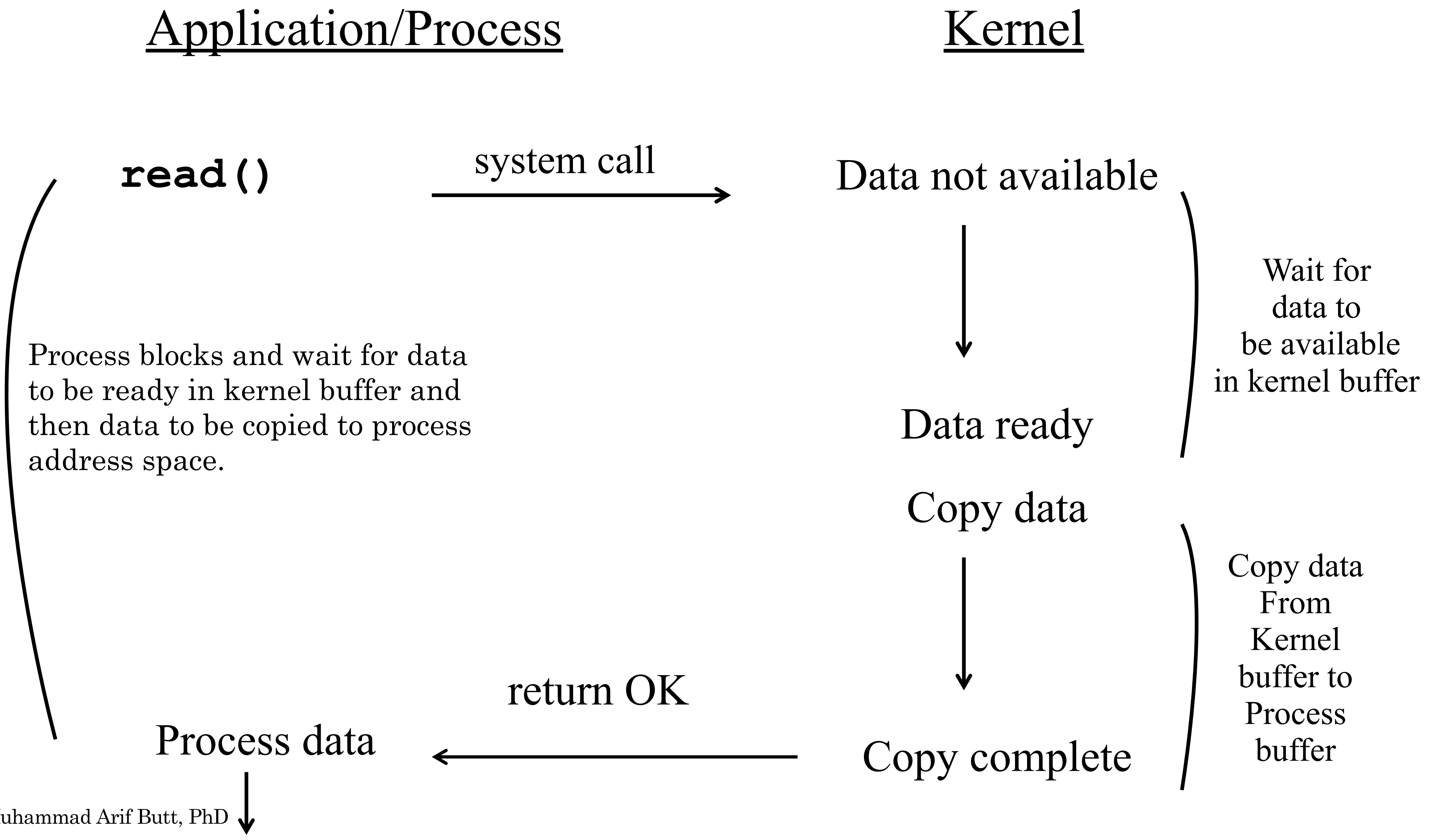
- Blocking I/O model
- Non-Blocking I/O model
- Multiplexed I/O model
- Signal Driven I/O model
- Asynchronous I/O model

Model	**Blocks on syscall?**	**Blocks on data copy?**	**Notification**
Blocking	Yes	Yes	None needed
Non-blocking	No	Yes	Polling required
Multiplexing	Yes (on select/poll)	Yes	select/poll returns
Signal-driven	No	Yes	SIGIO signal
Asynchronous	No	No	Completion notification

Before I give you an overview of each of these, remember there are normally, two distinct phases for an input operation:

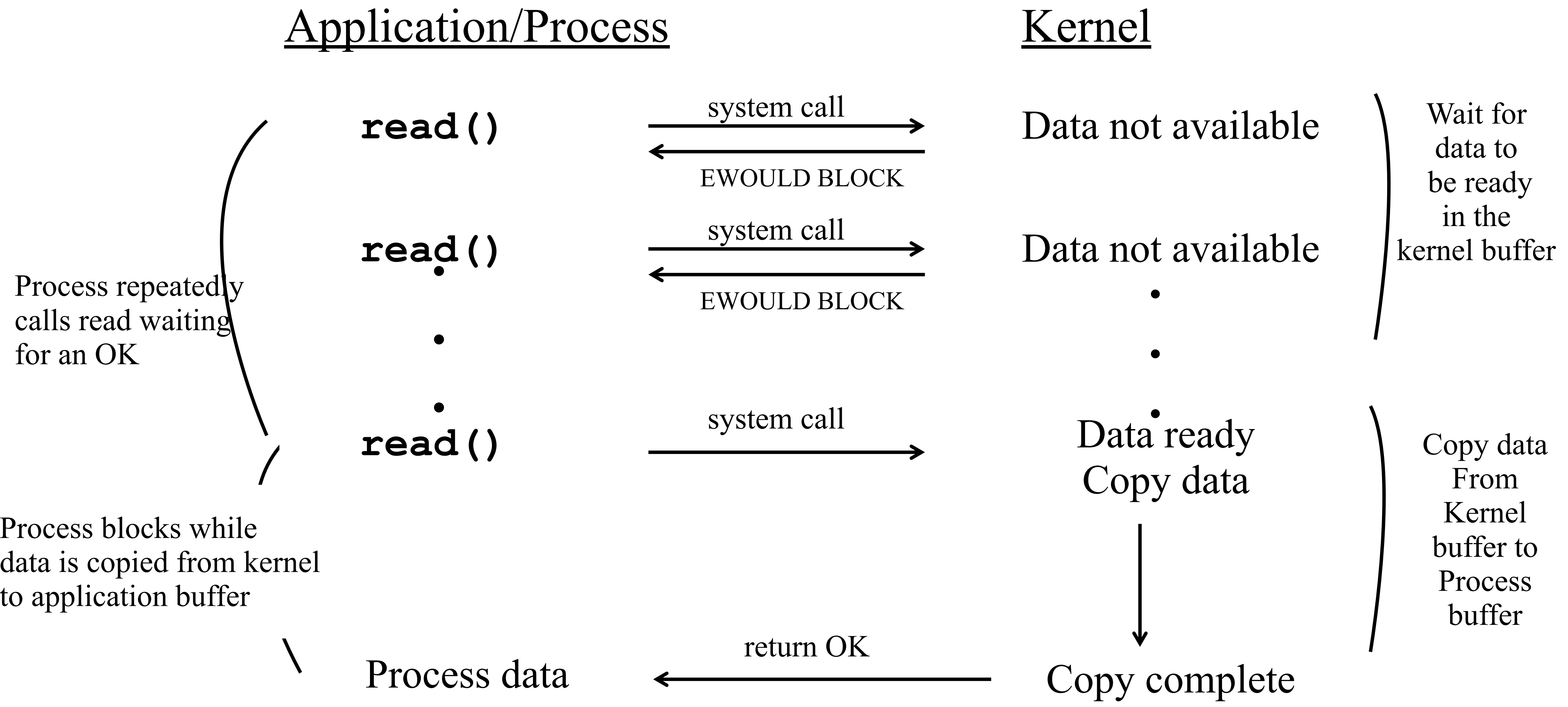
- Blocks on syscall: Wait for data to be ready in the kernel buffer
- Blocks on data copy: Wait for data to be copied from kernel buffer to process buffer

Blocking I/O Model

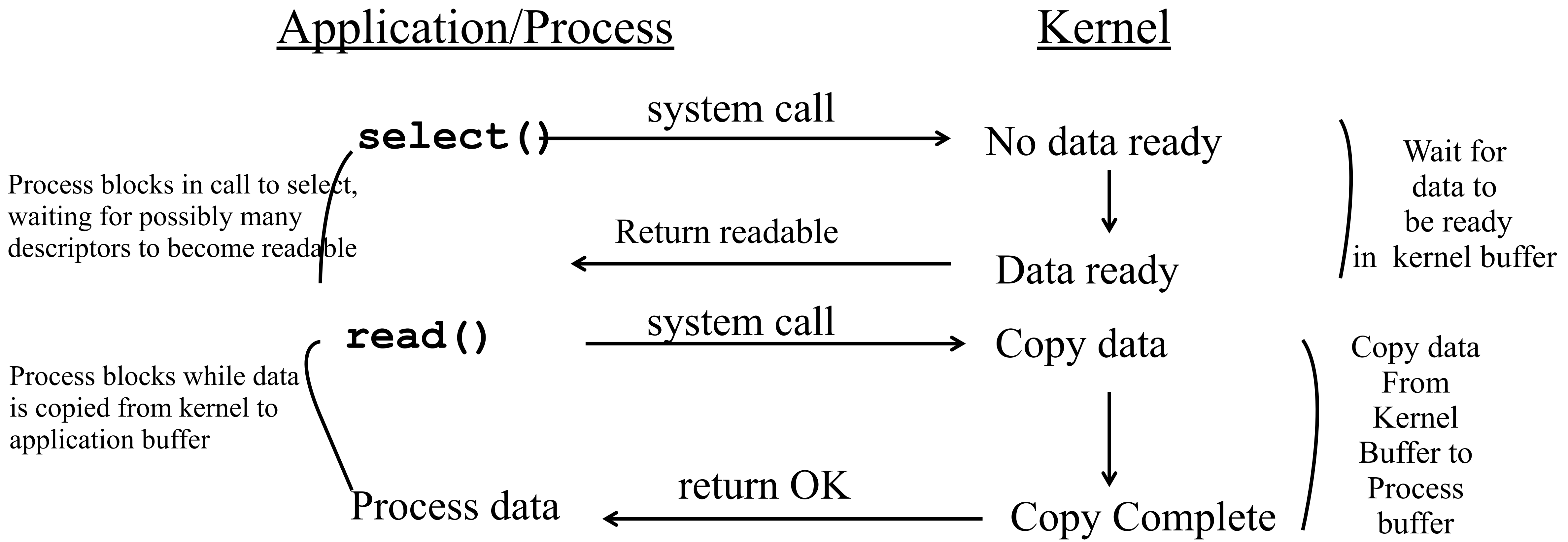


Process blocks and wait for data to be ready in kernel buffer and then data to be copied to process address space.

Non-Blocking I/O Model

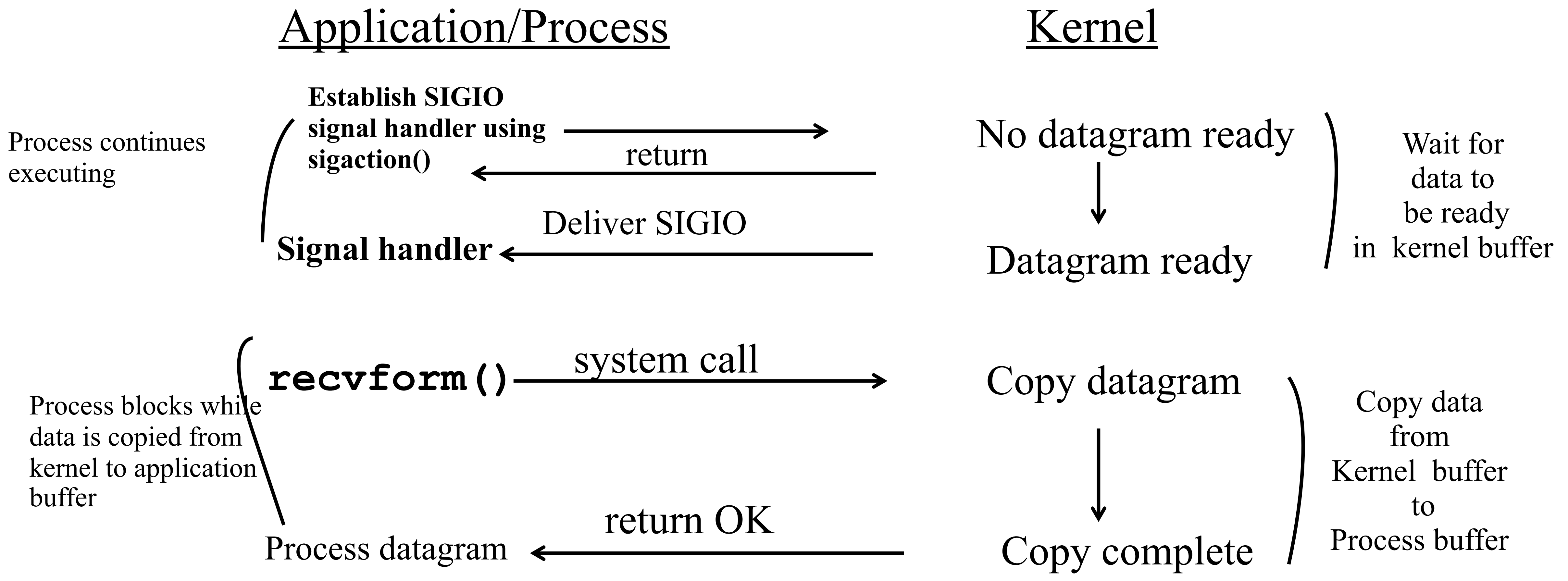

Application/Process
Kernel
read()
system call
EWOULD BLOCK
Data not available
read()
system call
EWOULD BLOCK
Data not available
Wait for data to be ready in the kernel buffer
Process repeatedly calls read waiting for an OK
read()
system call
Data ready
Copy data
Copy data From Kernel buffer to Process buffer
Process blocks while data is copied from kernel to application buffer
return OK
Process data
Copy complete

Multiplexed I/O Model

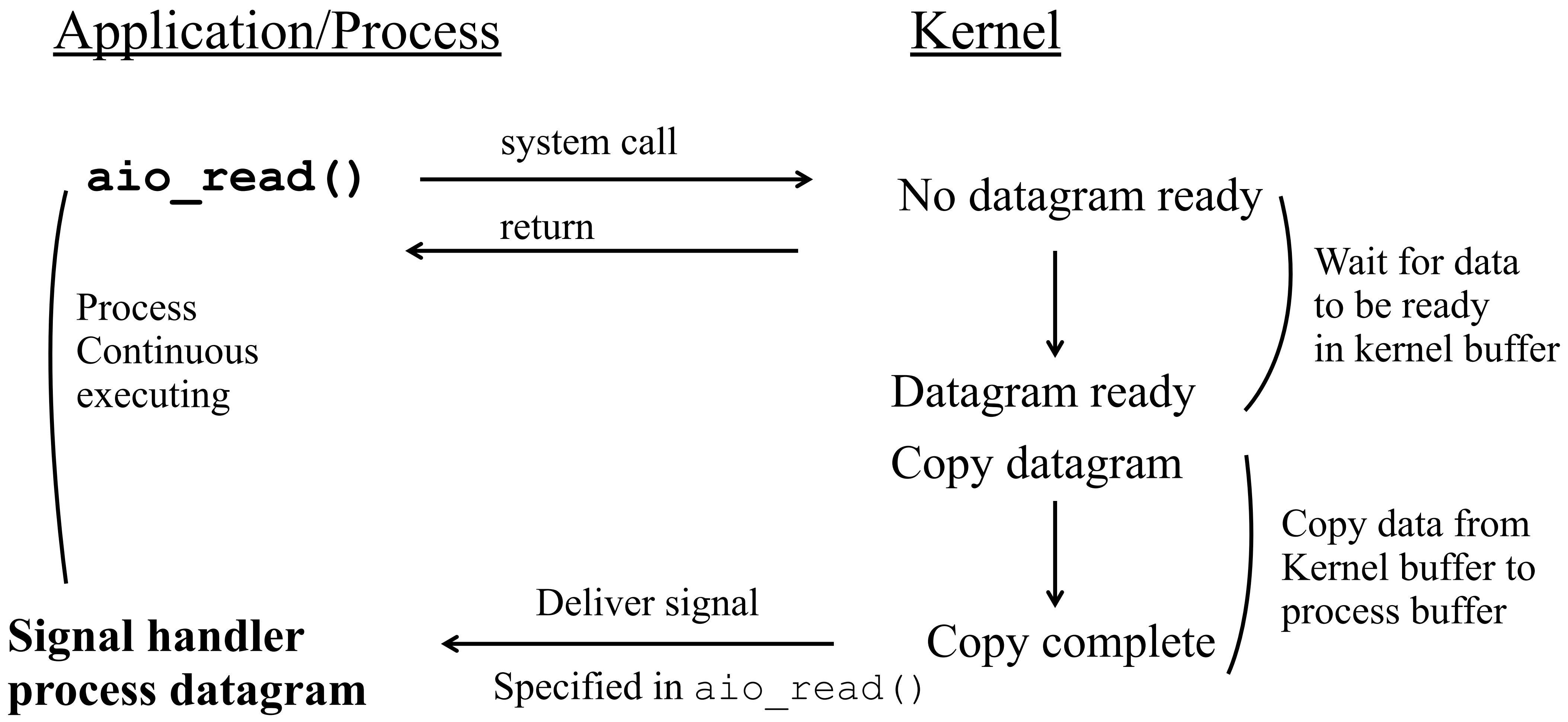


Application/Process
Kernel
select()
system call
No data ready
Process blocks in call to select, waiting for possibly many descriptors to become readable
Return readable
Data ready
Wait for data to be ready in kernel buffer
read()
system call
Copy data
Process blocks while data is copied from kernel to application buffer
Copy data From Kernel Buffer to Process buffer
Process data
return OK
Copy Complete

Signal Driven I/O Model


Application/Process
Kernel
Process continues executing
Establish SIGIO signal handler using sigaction()
return
No datagram ready
Datagram ready
Wait for data to be ready in kernel buffer
Deliver SIGIO
Signal handler
recvform()
system call
Copy datagram
Process blocks while data is copied from kernel to application buffer
Copy data from Kernel buffer to Process buffer
return OK
Process datagram
Copy complete

Asynchronous I/O Model


Application/Process
Kernel
aio_read()
system call
return
No datagram ready
Wait for data to be ready in kernel buffer
Process Continuous executing
Datagram ready
Copy datagram
Copy data from Kernel buffer to process buffer
Deliver signal
Specified in aio_read()
Copy complete
Signal handler process datagram

To Do

- Watch OS video on Character and Block Special Files:
  https://www.youtube.com/watch?v=MOP6sfXVKcY&list=PL7B2bn3G_wfBuJ_WtHADcXC44piWLRzr8&index=26

- Watch SP video on Terminal Devices:
  https://www.youtube.com/watch?v=t5sC6G73oo4&list=PL7B2bn3G_wfC-mRpG7cxJMnGWdPAQTViW&index=17

Coming to office hours does NOT mean that you are academically weak!