Devices, Drivers, DMA, Buffering

Computer Operating Systems, Fall 2023

Instructor: Travis McGaha

Head TAs: Nate Hoaglund & Seungmin Han

TAs:

Andy Jiang	Haoyun Qin	Kevin Bernat	Ryoma Harris
Audrey Yang	Jason hom	Leon Hertzberg	Shyam Mehta
August Fu	Jeff Yang	Maxi Liu	Tina Kokoshvili
Daniel Da	Jerry Wang	Ria Sharma	Zhiyan Lu
Ernest Ng	Jinghao Zhang	Rohan Verma	

Administrivia

- MILESTONE 1 IS DUE between Friday 11/10 – Tuesday 11/14
  - Expecting around 60% progress on this
  - Should have stand-alone PennFAT
  - Must meet with your TA in the specified window to demonstrate what you have implemented

- Recitation after lecture today will be about PennFAT

Administrivia

- I synched a bunch of grades to canvas. **PLEASE CHECK THAT THEY ARE ACCURATE**
  - All check-ins
  - Project 0 & peer-eval
- Midterm Grades Posted
  - Regrade requests open, due Saturday night @ midnight

Poll Everywhere

pollev.com/tqm

- Any questions, comments or concerns from last lecture?

Lecture Outline

- Device Drivers
  - LC4_GETC
- Stdio Buffering
  - fflush()
  - fsync()

I/O

- Reading/writing anything “beyond” memory is called I/O
  - We call the locations we read/write to I/O devices

- I/O devices include:
  - Keyboard
  - Mouse
  - Files
  - Graphics Displays
  - Networks
  - Etc.

Devices

- There are other "devices" than just the file storage
  - Auxiliary hardware that extends the functionality of the computer. The computer sends and/or receives data to communicate with the device
  - Sometimes called "Peripheral Devices"

- Examples:
  - Mouse
  - Keyboard
  - Game Controller
  - Printer
  - Network adapter
  - Projector
  - etc

Kinds of Devices

- These devices have many different functionalities and characteristics
- Block based vs character based
  - File system is block based
  - Keyboards are character based
- Shared by many processes
  - Network card, disk
- User related vs OS related
  - Keyboard vs system clock

Device Drivers

- How does a computer support these various device types?

- Each device has a *driver*: a piece of software that acts as the interface to the device. Abstracts away some of the hardware details of the device
  - Contains device specific routines for communicating with the computer and routines for controlling/configuring the device

- Your computer comes with some device drivers installed
- When you plug in a new device, your computer will start installing device drivers for that device.

IOCTL

- Input/Output Control
- The provided Linux system calls (e.g. read, write) are not enough to express the different functions a device may have.
- `int ioctl(int fd, int request, ...)`
  - Specify the file descriptor of the device you want to interact with
  - Request contains information on what you would like to do (and some other information)
  - Variadic arguments (usually a char* or void*)

Device Naming & Separation

- It can be difficult to keep track of which device is using what resources. If we use memory mapped I/O, what addresses belong to which files?

- If we have everything resident in OS memory, then it could also be difficult to manage concurrent processes accessing the same device

Everything is File

- Idea: Give each device a named file and have most requests go through the filesystem.

- The filesystem allows us to name our devices.
  - /dev/ directory contains various devices as “files”
  - For example, /dev/printer1

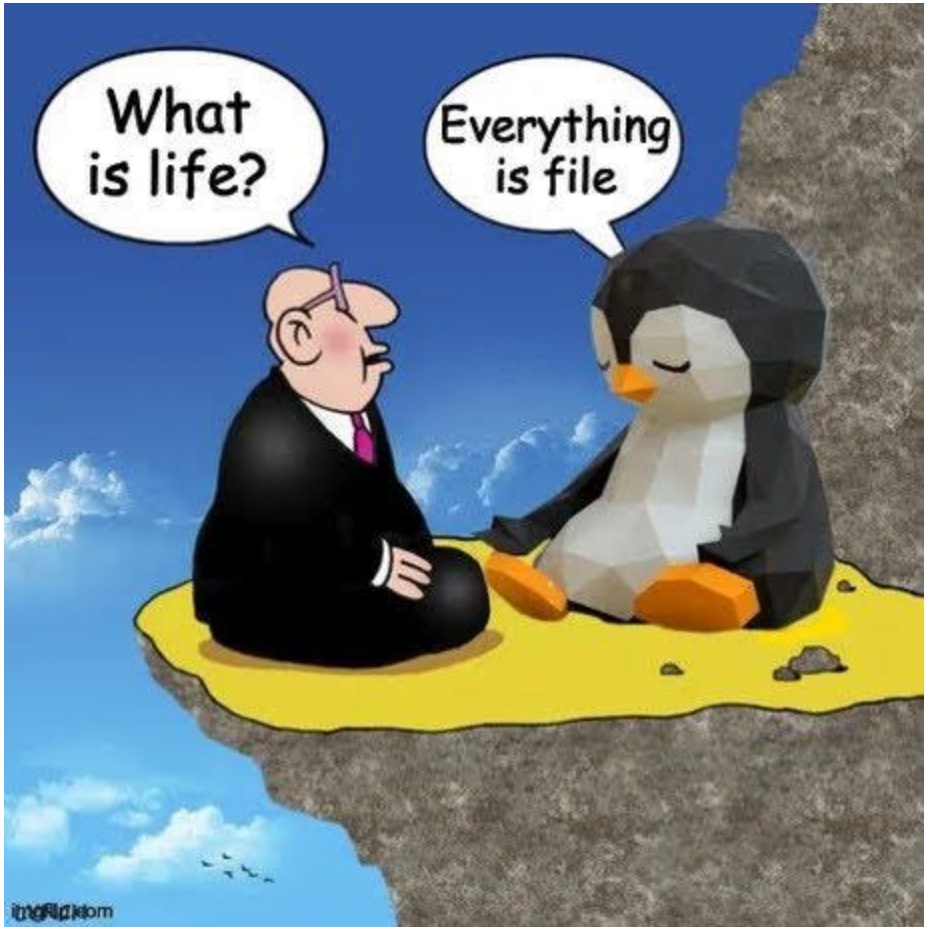
- I/O requests through the file system are already scheduled, have an order enforced, and are checked to be concurrent safe*
  - (from the filesystem level, user can still mess it up)

Everything is File

- Note: these devices are not like normal "files" as discussed previously

- These things just appear as files and can be read/written to perform some functionality.

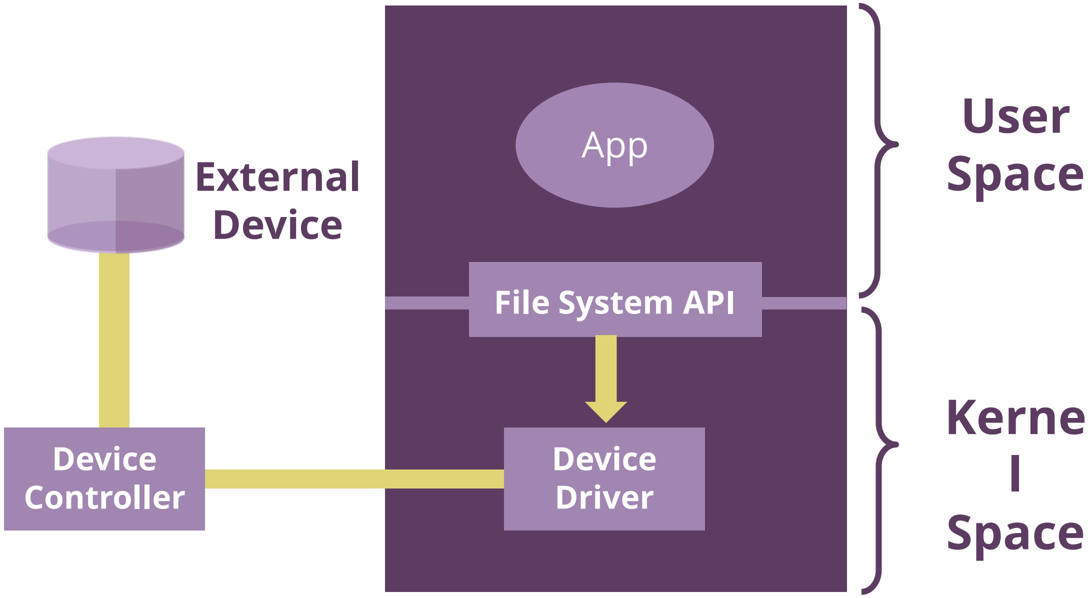
- Many things are files in linux, it provides a nice consistent interface to interact with devices.



Special Devices

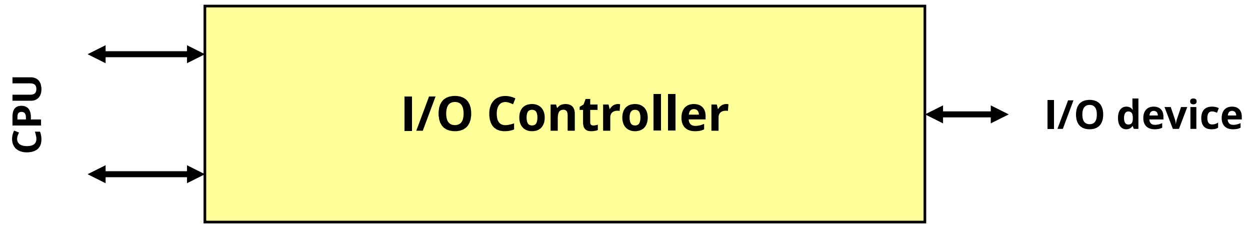
- Some special devices that exist in /dev/
- /dev/urandom and /dev/random
  - Provides bytes by the computers cryptographically secure pseudorandom number generator
- /dev/null
  - Discards anything that is written to it and reports the write as a success.
- /dev/fd/
  - Directory containing the open file descriptors for the running process
- /dev/stdin, /dev/stdout, /dev/stderr
  - Access to the process' standard streams

I/O Architecture



I/O Devices & Controllers

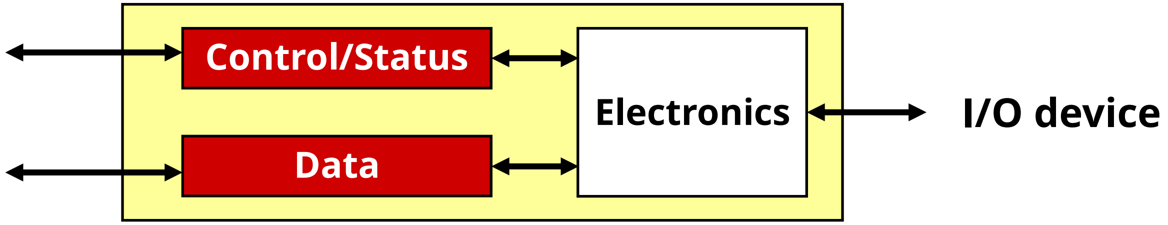
- Most I/O devices are not purely digital, they have their own hardware
  - Electro-mechanical: e.g. keyboard, mouse, disk, motor
  - Analog/digital: e.g. touchscreen, network interface, monitor, speaker, mic
- … all have digital interfaces presented by an I/O Controller
  - I/O Device (analog/digital mix) talks to controller
  - CPU (digital) talks to controller (typically through a device driver)
  - Controller acts as a translator: digital (CPU) <-> analog (device)



I/O Controller to CPU Interface

- I/O controller interface abstracts I/O device as "device registers"
  - Control/Status: may be one register or two
    - Control: lets us toggle options on the device (we won't focus on this)
    - Status: lets us know if we are data is ready to be read/written
  - Data: may be more than one register
    - The data we are reading/writing
- Example: CPU reading data from input device
  - CPU checks status register if input is available
  - Reads input the data register

Similar steps for writing.
More details later!



How can we handle I/O with code/asm?

- **Two common options**
- We could create new "I/O instructions" for the ISA
  - Designate opcode(s) for I/O
  - Register and operation encoded in instruction
- Memory-mapped I/O (Using LDR/STR for LC4)
  - Assign a memory address to each device register
  - Use conventional loads and stores
  - Hardware intercepts loads/stores to these address
  - No actual memory access performed
    (MMU and caches get more complicated as a result)
  - LC4 (and most other platforms) do this
  - This allows for the I/O code to be written in C and is more portable to other systems.

Poll Everywhere

pollev.com/tqm

- Do you see any problem with this way of getting data from a device (e.g. file/keyboard/etc.)
  - This is what we did in LC4

```
char getc() {
  while(*device_status == NOT_READY) {
    // do nothing
  }
  char user_input = *device_data;
  return user_input;
}
```

Poll Everywhere

pollev.com/tqm

- Do you see any problem with this way of getting data from a device (e.g. file/keyboard/etc.)
  - This is what we did in LC4

```
char getc() {
  while(*device_status == NOT_READY) {
    // do nothing
  }
  char user_input = *device_data;
  return user_input;
}
```

Busy waiting ☹

Poll Everywhere

pollev.com/tqm

- Do you see any problem with this way of getting data from a device (e.g. file/keyboard/etc.)
  - This is trying to make this "No hang", do not block if character is not available

```
char getc() {
  if (*device_status == NOT_READY) {
    return NOT_READY;
  }
  char user_input = *device_data;
  return user_input;
}
```

Poll Everywhere

pollev.com/tqm

- Do you see any problem with this way of getting data from a device (e.g. file/keyboard/etc.)
  - This is trying to make this "No hang", do not block if character is not available

```
char getc() {
  if (*device_status == NOT_READY) {
    return NOT_READY;
  }
  char user_input = *device_data;
  return user_input;
}
```

Busy waiting still possible… What happens if the process is blocked on waiting for input?

Interrupts

- Can instead have the hardware device interrupt the CPU to let the OS know that some I/O request is done

- Allows OS to not run blocked processes, and scheduler other processes that will utilize the CPU

Question: what?

- How do interrupts work to solve the problem we just discussed?

- If the CPU is not doing the work, then what is?

CPU vs Co-processors

- The CPU is the **C**entral **P**rocessing **U**nit
  - The set of instructions that are possible is fixed, but the exact instructions & program changes.
  - This allows the CPU to be more “general purpose”
- Our computer also has Coprocessors
  - These are hardware devices that also perform some computation to supplement the CPU.
  - Usually more specialized
  - Examples: Graphics Processing Unit (GPU), Floating Point Unit (FPU), I/O processors, network cards, sound cards, etc.
  - What these do and how they are controlled can vary a lot.

DMA

- To support co-processors, they are usually Direct Memory Access (DMA)
  - If DMA is supported, then allowed coprocessors can directly access memory independently of CPU

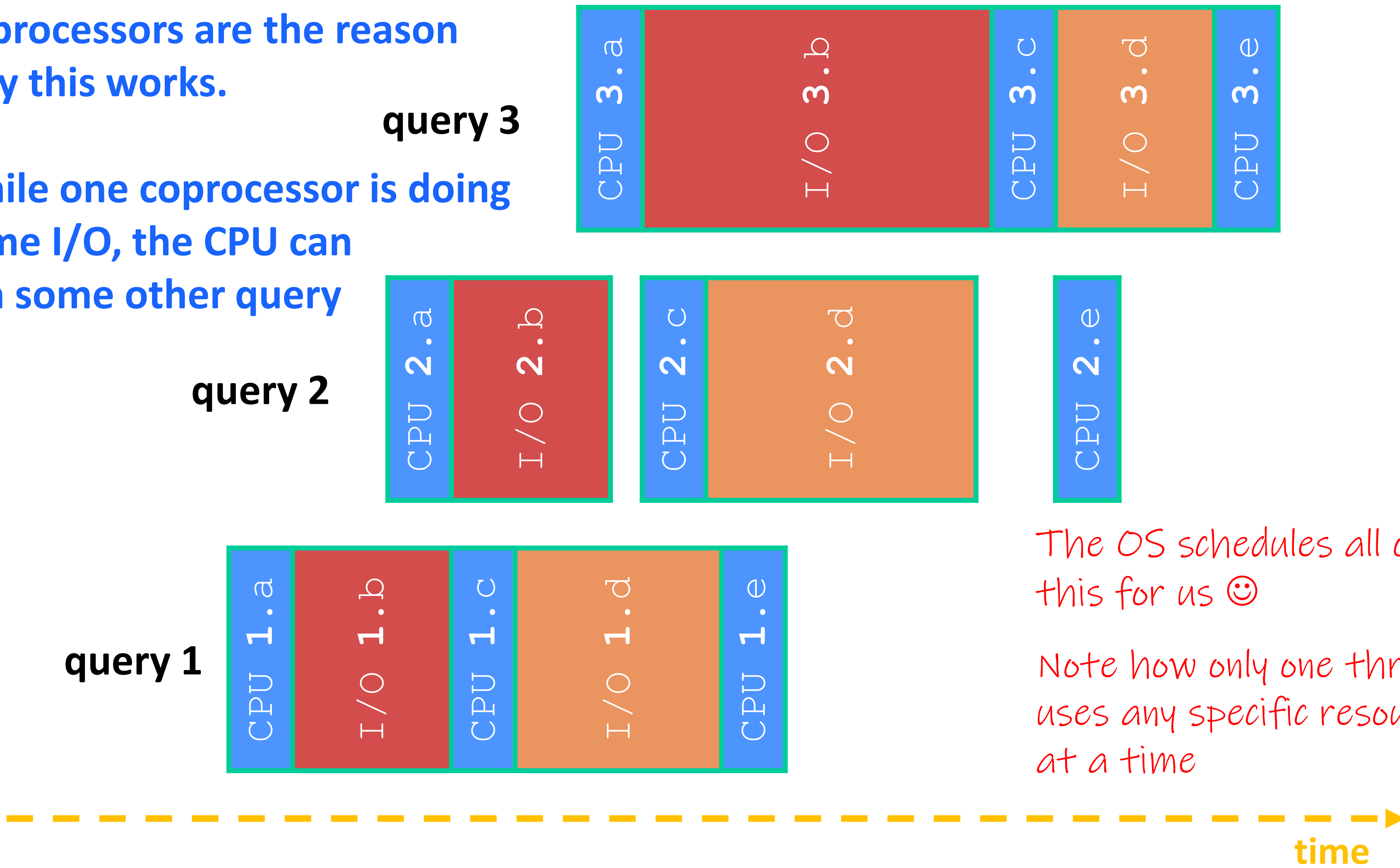
- In our I/O example, this means that an I/O request looks something like:
  - First the CPU sends a request to the I/O coprocessor for a storage medium to perform some read/write.
  - The coprocessor can fulfill this request and access memory directly to store what is read or get what needs to be written
  - The CPU does other things while the I/O request is running and eventually is interrupted by the coprocessor when the request is done.

Multi-threaded Search Engine (Execution)

*Running with 1 CPU

Remember this?
Coprocessors are the reason why this works.

While one coprocessor is doing some I/O, the CPU can run some other query



The OS schedules all of this for us ☺

Note how only one thread uses any specific resource at a time

Lecture Outline

- d

Poll Everywhere

pollev.com/tqm

- If we compile this and run it, how many times is hello printed?

```
int main() {
  if (fork() == 0) {
    write(STDOUT_FILENO, "hello", 5);
  }
  if (fork() == 0) {
    write(STDOUT_FILENO, "hello", 5);
  }
  return EXIT_SUCCESS;
}
```

Raise Your Hands

- If we compile this and run it, how many times is hello printed?

```
int main() {
  if (fork() == 0) {
    printf("hello");
  }
  if (fork() == 0) {
    printf("hello");
  }
  return EXIT_SUCCESS;
}
```

Raise Your Hands

- If we compile this and run it, how many times is hello printed?

```
int main() {
  if (fork() == 0) {
    printf("hello\n");
  }
  if (fork() == 0) {
    printf("hello\n");
  }
  return EXIT_SUCCESS;
}
```

C stdio vs POSIX

- Why are we getting these different outputs?

- Let's start with the first two. Both use different ways of writing to standard out.

  - C stdio : user level portable library for **st**andar**d** **i**nput/**o**utput. Should work on any environment that has the C standard library
    - E.g. printf, fprintf, fputs, getline, etc.

  - POSIX C API: **P**ortable **O**perating **S**ystem **I**nterface. Functions that are supported by many operating systems to support many OS-level concepts (Input/Output, networking, processes, threads…)

Buffered writing

- By default, C `stdio` uses buffering on top of POSIX:
  - When one writes with **`fwrite`**`()`, the data being written is copied into a buffer allocated by `stdio` inside your process' address space
  - As some point, once enough data has been written, the buffer will be "flushed" to the operating system.
    - When the buffer fills (often 1024 or 4096 bytes)
  - This prevents invoking the write system call and going to the filesystem too often

Buffered Writing Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  char buf[2] = {'h', 'i'};
→ FILE* fout = fopen("hi.txt", "wb");

  // read "hi" one char at a time
  fwrite(&buf, sizeof(char), 1, fout);

  fwrite(&buf+1, sizeof(char), 1, fout);

  fclose(fout);
  return EXIT_SUCCESS;
}
```

Buffered Writing Example

Arrow signifies what will be executed next

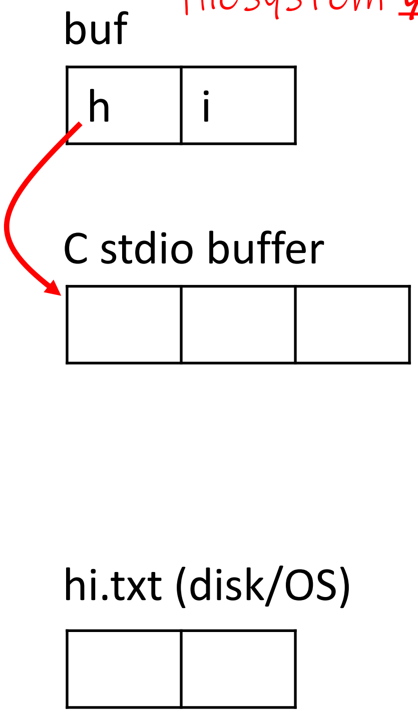
```
int main(int argc, char** argv) {
  char buf[2] = {'h', 'i'};
  FILE* fout = fopen("hi.txt", "wb");

  // read "hi" one char at a time
  fwrite(&buf, sizeof(char), 1, fout);

  fwrite(&buf+1, sizeof(char), 1, fout);

  fclose(fout);
  return EXIT_SUCCESS;
}
```

Store 'h' into buffer, so that we do not go to filesystem *yet*



Buffered Writing Example

Arrow signifies what will be executed next

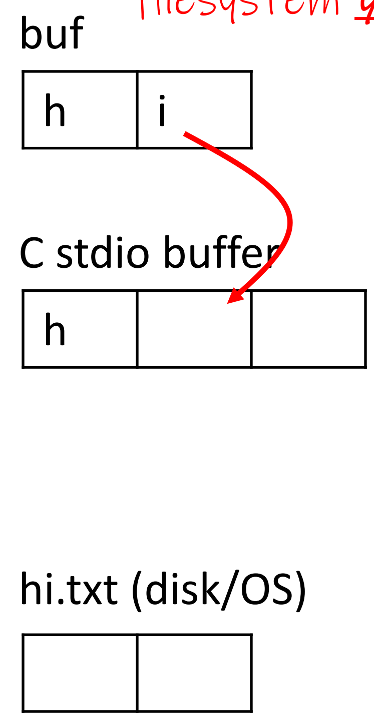
Store 'i' into buffer, so that we do not go to filesystem *yet*

```
int main(int argc, char** argv) {
  char buf[2] = {'h', 'i'};
  FILE* fout = fopen("hi.txt", "wb");

  // read "hi" one char at a time
  fwrite(&buf, sizeof(char), 1, fout);

→ fwrite(&buf+1, sizeof(char), 1, fout);

  fclose(fout);
  return EXIT_SUCCESS;
}
```



Buffered Writing Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  char buf[2] = {'h', 'i'};
  FILE* fout = fopen("hi.txt", "wb");

  // read "hi" one char at a time
  fwrite(&buf, sizeof(char), 1, fout);

  fwrite(&buf+1, sizeof(char), 1, fout);

→ fclose(fout);
  return EXIT_SUCCESS;
}
```

buf

h	i

C stdio buffer

h	i	

When we call fclose, we deallocate and flush the buffer to disk

hi.txt (disk/OS)

Buffered Writing Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  char buf[2] = {'h', 'i'};
  FILE* fout = fopen("hi.txt", "wb");

  // read "hi" one char at a time
  fwrite(&buf, sizeof(char), 1, fout);

  fwrite(&buf+1, sizeof(char), 1, fout);

  fclose(fout);
→ return EXIT_SUCCESS;
}
```

buf

h	i

hi.txt (disk/OS)

h	i

Unbuffered Writing Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  char buf[2] = {'h', 'i'};
→ int fd = open("hi.txt", O_WRONLY | O_CREAT);

  // read "hi" one char at a time
  write(fd, &buf, sizeof(char));

  write(fd, &buf+1, sizeof(char));

  close(fd);
  return EXIT_SUCCESS;
}
```

buf

h	i

hi.txt (disk/OS)

Unbuffered Writing Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  char buf[2] = {'h', 'i'};
  int fd = open("hi.txt", O_WRONLY | O_CREAT);

  // read "hi" one char at a time
→ write(fd, &buf, sizeof(char));

  write(fd, &buf+1, sizeof(char));

  close(fd);
  return EXIT_SUCCESS;
}
```

buf

h	i

hi.txt (disk/OS)

Unbuffered Writing Example

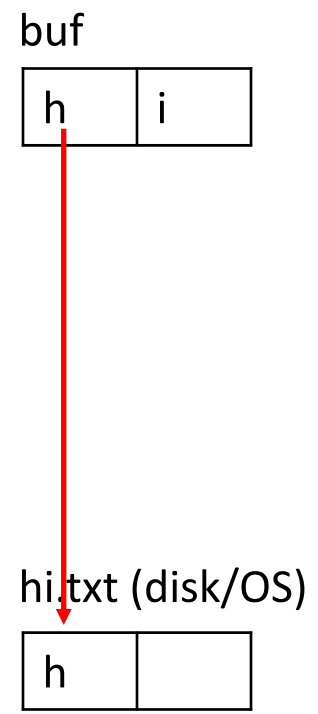
Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  char buf[2] = {'h', 'i'};
  int fd = open("hi.txt", O_WRONLY | O_CREAT);

  // read "hi" one char at a time
  write(fd, &buf, sizeof(char));

→ write(fd, &buf+1, sizeof(char));

  close(fd);
  return EXIT_SUCCESS;
}
```



Unbuffered Writing Example

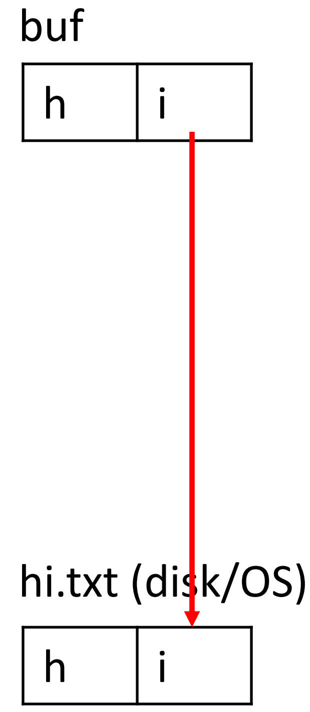
Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  char buf[2] = {'h', 'i'};
  int fd = open("hi.txt", O_WRONLY | O_CREAT);

  // read "hi" one char at a time
  write(fd, &buf, sizeof(char));

  write(fd, &buf+1, sizeof(char));

→ close(fd);
  return EXIT_SUCCESS;
}
```



Unbuffered Writing Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  char buf[2] = {'h', 'i'};
  int fd = open("hi.txt", O_WRONLY | O_CREAT);

  // read "hi" one char at a time
  write(fd, &buf, sizeof(char));

  write(fd, &buf+1, sizeof(char));

  close(fd);
→ return EXIT_SUCCESS;
}
```

buf

h	i

Two OS/File system accesses instead of one ☹

hi.txt (disk/OS)

h	i

Buffered Reading

- By default, C `stdio` uses buffering on top of POSIX:
  - When one reads with **`fread()`**, a lot of data is copied into a buffer allocated by `stdio` inside your process' address space
  - Next time you read data, it is retrieved from the buffer
    - This avoids having to invoke a system call again
  - As some point, the buffer will be "refreshed":
    - When you process everything in the buffer (often 1024 or 4096 bytes)
  - Similar thing happens when you write to a file

Buffered Reading Example

Arrow signifies what will be executed next

```c
int main(int argc, char** argv) {
  char buf[2];
  FILE* fin = fopen("hi.txt", "rb");   // ← arrow

  // read "hi" one char at a time
  fread(&buf, sizeof(char), 1, fin);

  fread(&buf+1, sizeof(char), 1, fin);

  fclose(fin);
  return EXIT_SUCCESS;
}
```

buf

hi.txt (disk/OS)

h	i

Buffered Reading Example

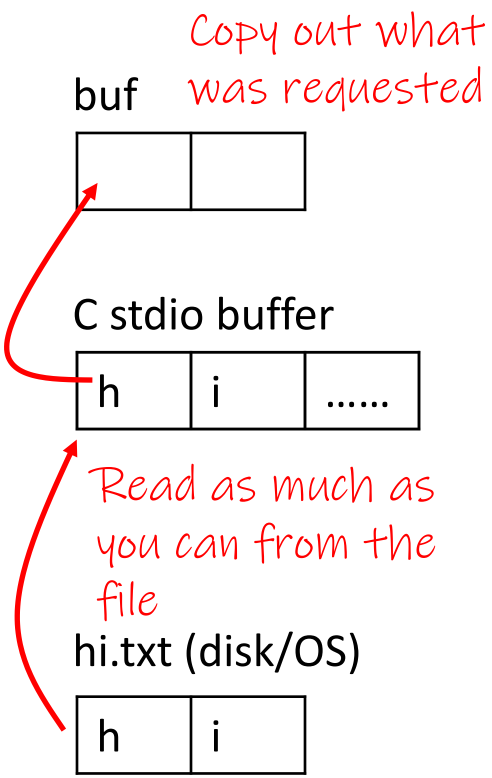
Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  char buf[2];
  FILE* fin = fopen("hi.txt", "rb");

  // read "hi" one char at a time
  fread(&buf, sizeof(char), 1, fin);    <-- (arrow)

  fread(&buf+1, sizeof(char), 1, fin);

  fclose(fin);
  return EXIT_SUCCESS;
}
```



Buffered Reading Example

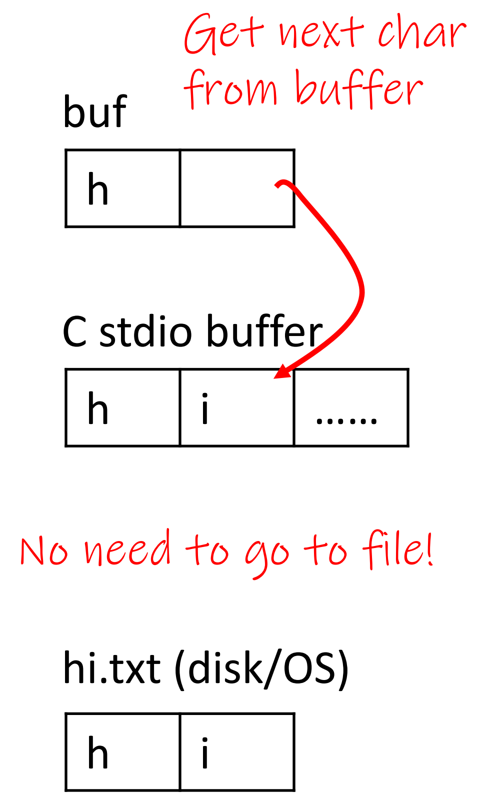
Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  char buf[2];
  FILE* fin = fopen("hi.txt", "rb");

  // read "hi" one char at a time
  fread(&buf, sizeof(char), 1, fin);

  fread(&buf+1, sizeof(char), 1, fin);

  fclose(fin);
  return EXIT_SUCCESS;
}
```



Buffered Reading Example

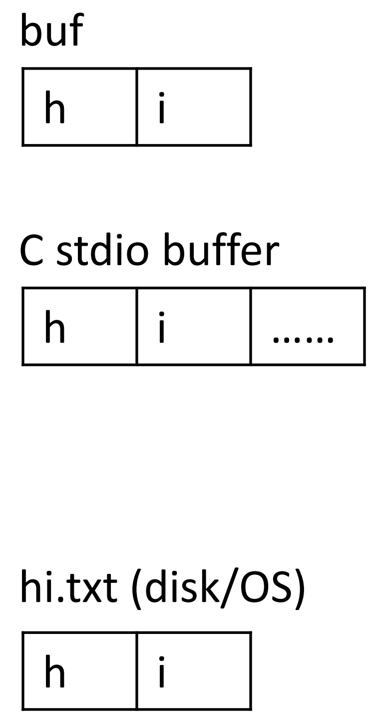
Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  char buf[2];
  FILE* fin = fopen("hi.txt", "rb");

  // read "hi" one char at a time
  fread(&buf, sizeof(char), 1, fin);

  fread(&buf+1, sizeof(char), 1, fin);

→ fclose(fin);
  return EXIT_SUCCESS;
}
```



Buffered Reading Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  char buf[2];
  FILE* fin = fopen("hi.txt", "rb");

  // read "hi" one char at a time
  fread(&buf, sizeof(char), 1, fin);

  fread(&buf+1, sizeof(char), 1, fin);

  fclose(fin);
→ return EXIT_SUCCESS;
}
```

buf

h	i

hi.txt (disk/OS)

h	i

Why NOT Buffer?

- Reliability – the buffer needs to be flushed
  - Loss of computer power = loss of data
  - "Completion" of a write (*i.e.* return from **`fwrite()`**) does not mean the data has actually been written

- Performance – buffering takes time
  - Copying data into the `stdio` buffer consumes CPU cycles and memory bandwidth
  - Can potentially slow down high-performance applications, like a web server or database (*"zero-copy"*)

- When is buffering faster? | Slower?

Many small writes
Or only writing a little

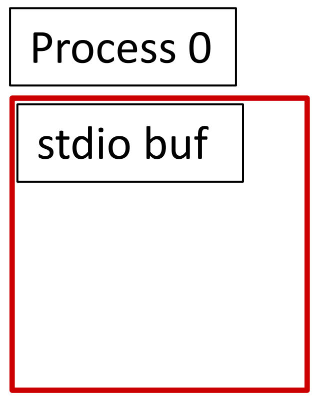
Large writes

Fork Problem Explained

Arrow signifies what will be executed next. I execute processes in parallel and "in sync" for demonstration purposes

- Remember: printf (and stdio) buffers input in the programs address space

```
int main() {
  if (fork() == 0) {
    printf("hello");
  }
  if (fork() == 0) {
    printf("hello");
  }
  return EXIT_SUCCESS;
}
```

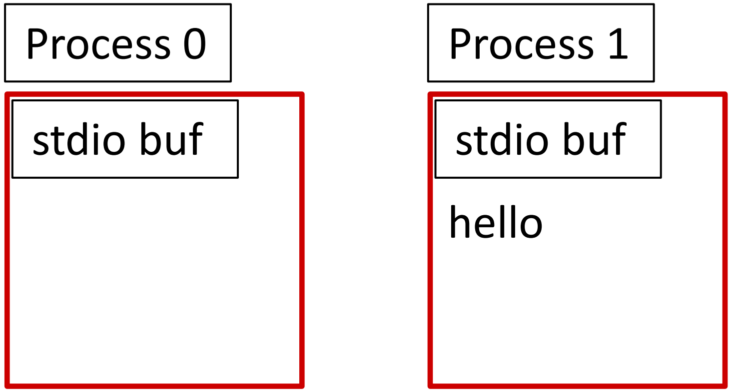


Fork Problem Explained

Arrow signifies what will be executed next.
I execute processes in parallel and "in sync" for demonstration purposes

- Remember: printf (and stdio) buffers input in the programs address space

```
int main() {
  if (fork() == 0) {
    printf("hello");
  }
→ if (fork() == 0) {
    printf("hello");
  }
  return EXIT_SUCCESS;
}
```

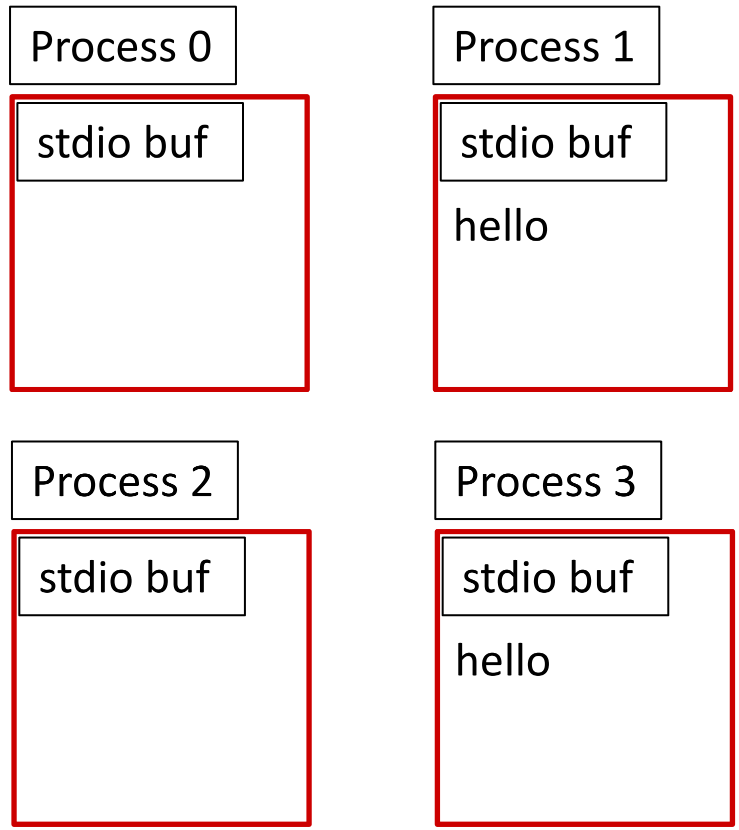


Fork Problem Explained

Arrow signifies what will be executed next.
I execute processes in parallel and "in sync" for demonstration purposes

- Remember: printf (and stdio) buffers input in the programs address space

```c
int main() {
  if (fork() == 0) {
    printf("hello");
  }
  if (fork() == 0) {
    printf("hello");
  }
  return EXIT_SUCCESS;
}
```

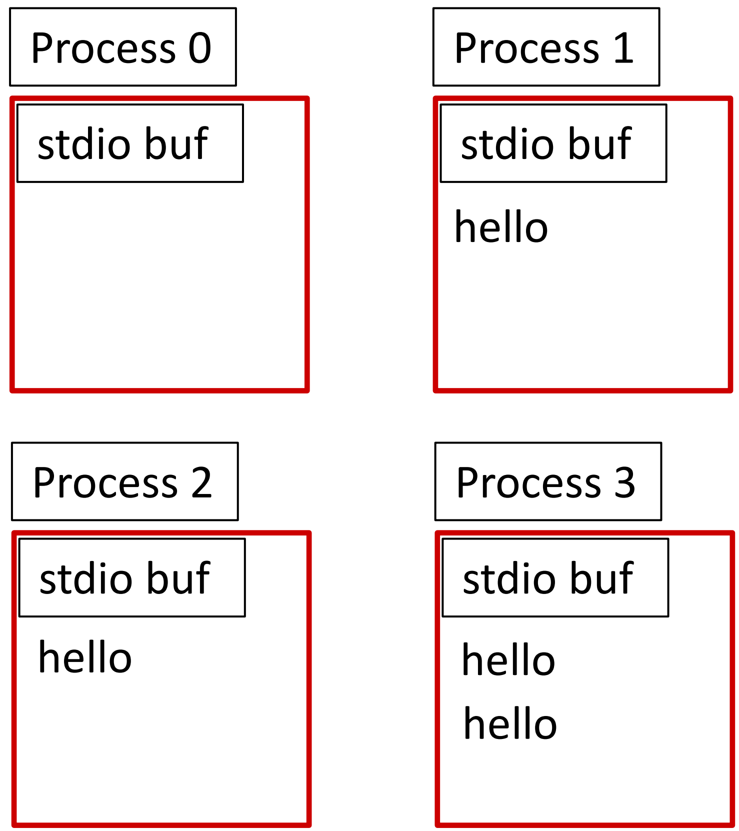


Fork Problem Explained

Arrow signifies what will be executed next.
I execute processes in parallel and "in sync" for demonstration purposes

- Remember: printf (and stdio) buffers input in the programs address space

```
int main() {
  if (fork() == 0) {
    printf("hello");
  }
  if (fork() == 0) {
    printf("hello");
  }
→ return EXIT_SUCCESS;
}
```



Hello is printed 4 times!

Fork Problem Explained (pt.2)

- Why did we get different outputs when printf printed a newline character after hello?
  - Only difference was:

```
printf("hello");
```

vs

```
printf("hello\n");
```

- All we needed to do to get the expected output was add a \n. why?

- `printf` prints to stdout and by default stdout is line buffered. Meaning it flushes the buffer on a newline character
  - If we ran ./prog > out.txt (redirect the output), we would get different output since buffering policy changes.

How to flush/modify the cstdio buffer

- For C stdio:
  - ```
    int fflush(FILE* stream);
    ```
  - Flushes the stream to the OS/filesystem
  - ```
    int setvbuf(FILE* stream, char* buf,
                int mode, size_t size);
    ```
  - Has a family of related functions like setbuf(), setbuffer(), setlinebuf();
  - Can set the stream to be unbuffered or a specified buffer

How to flush POSIX?

- When we write to a file with POSIX it is sent to the filesystem, is it immediately sent to disc? No
  - Well, we do have the block cache… so it may not be written to disc
  - Since all File I/O requests go to the file system, if another process accesses the same file, then it should see the data even if it is the block cache and not in disc.
  - If we lose power though…

How to flush POSIX to disk

- Two functions
  - ```
    int fsync(int fd);
    ```
  - Flushes all in-core data and metadata to the storage medium
  - ```
    int fdatasync(int fd);
    ```
  - Sends the file data to disk
  - Does not flush modified metadata unless necessary for data.

- C stdio is usually implemented using POSIX on posix compliant systems
  - **`fflush`** may not necessarily call **`fsync`**