(content added after midterm)

NOTES: Operating Systems

notes/operating-systems.md

Operating Systems

- Virtualization: every program can operate as if it's the only program running
- Concurrency: multi-processor distributed systems
- Persistency: storage, filesystem

Processes

- A process is an instance of a running program afforded by the abstract machine by the OS
- A process is stateful—it is defined by the program counter, all CPU registers, and all memory
- Processes get local control flow—to them, it appears they get the whole CPU
- Processes get private address spaces—they seem to have all of the memory

  Note: Multi-Core Computers with more than one CPU (core) can run multiple processes at the same time. Each core runs one process

Note: Managing Processes in Bash

```
top # show truncated list of processes, kind of like task manager
ps # list processes
ps x # list processes started by you
ps ax # list all processes
kill <id> # kill a process
kill -9 <id> # force kill a process, even if it ignores SIGINT and > SIGTERM
```

Stopping Processes

- A process can be stopped by an asynchronous interrupt—can happen at any time
- A process can be stopped by a syncronous exception, which is invoked by certain instructions
  - TRAP by specific instructions
  - FAULT by illegal instructions (i.e. divide by zero)
  - ABORT by the OS
- The OS cannot save program state after an interrupt because it needs to use registers itself. The program also should not be responsible for saving its own state because it could be interrupted at any time. Instead, the hardware (CPU) and OS collaborate to preserve the register content. Exception handling is defined by the architecture.
  - Usually, state is saved to the stack

Concurrency

- The OS switches contexts whenever a process is swapped from foreground to background
  - e.g. User Context A → Kernel Context → User Context B

Scheduling

- Every so often (usually every ms), the OS sends the current program a **system timer interrupt**
- After the system timer interrupt, the OS reevaluates which process to continue

Privilege

- CPU has various operational modes
  - User mode: normal program execution
  - Kernel mode: OS execution
- Certain instructions are only allowed in higher privileged modes
- There are special instructions for mode transition
- Memory is segmented into user and kernel space
  - OS can access all memory
  - User programs can only access their own memory

Kernel Memory

- The kernel has its own memory and its own stack
- When the CPU changes between user/kernel mode, the stack pointer changes to the kernel or user stack

System Calls

- A system call tells the OS to perform an action for the program
- ex. `read`, `write`, `open`, `close`, `fork`, `exec`, `wait`, `exit`
- System calls are an abstraction for the kernel, and allow the program to be portable from one OS to another

  ## Note: C Library Calls vs System Calls

  - There are two ways to open a file: `fopen` and `open`
  - `fopen` is a Portable C function, and is much more efficient since it > implements caching
  - `open` is just a wrapper for the Unix system call

Creating a New Process

- To create a new process, use `fork` followed by `execve`

Fork

`fork()` makes a copy of the current process (the new process will be in the middle of execution)

- Returns the copy's PID to the parent process
- Returns 0 to the child process (this is how you tell if you're the copy or the parent)
- It is undefined which process will run first

Execve

`int execve(char* prog, char** argv, char** env)`

- replaces the current process with a new one
- `prog` is name of the file to execute
- `argv` is an array of arguments
- `env` is an array of environment variables
  - Access environment variables with the `environ` global variable

Waiting for a Child Process to Exit

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

- `waitpid` waits for a child process to exit
- Use WEXITSTATUS(status) to get the value that main returned
- `waitpid` also recycles the PID of the child process, allowing it to be re-used
    - Without `waitpid` being called, **zombie processes** may be created—processes that exited, but the OS still thinks their PID is being used
    - `waitpid` must *reap* zombie processes before their PIDs can be recycled
    - You can use `waitpid(-1, ...)` to wait on all children
        - Equivalent syntax: `pid_t wait(int* status)`
- Without using `waitpid`, a child process can continue even if its parent has exited
    - Child processes without parent processes are called **orphans**
    - Orphan processes are adopted by `init` (PID #1)

Files

- Everything in Unix is a file—terminals, disks, printers, monitors
- A **file descriptor** is an integer that represents an open file
- File descriptors are shared between threads, but not processes
- Important: `execve` does NOT close file descriptors

Opening Files

```
int open(const char* path, int flags)
```

- `flags`: `O_RDONLY`, `O_WRONLY`, or `O_RDWR`

```
int close(int fd)
```

- Writes **EOF** to the file, and closes the file descriptor

Reading and Writing Files

```
ssize_t read(int fd, void* buf, size_t n)
```

- Puts `n` bytes into `buf`, or less if **EOF** is reached
- Returns the number of bytes read, or -1 for error

```
ssize_t write(int fd, const void* buf, size_t n)
```

- Writes to `fd`, using up to `n` bytes from `buf`
- Returns the number of bytes written, or -1 for error

`pipe`

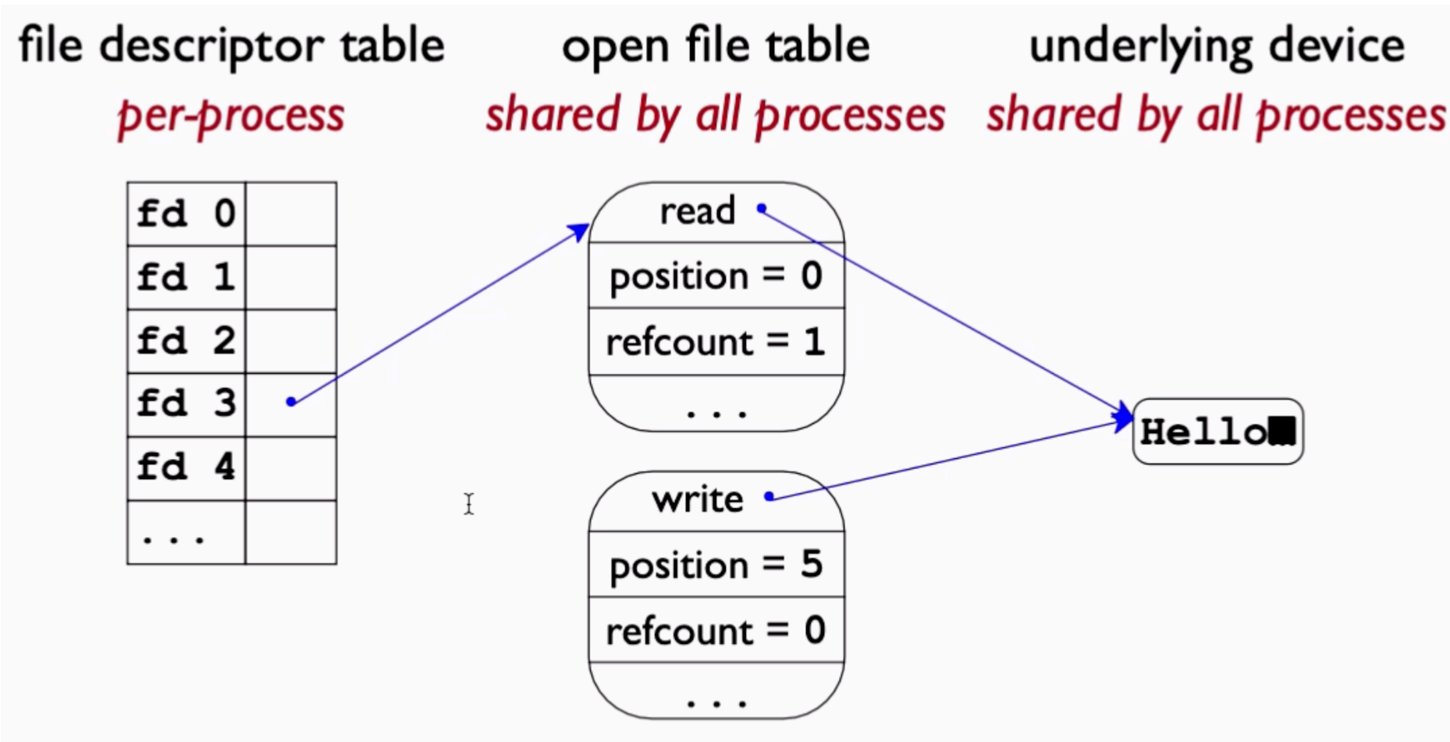
- `pipe` creates an unnamed "file" that only exists in memory
- `pipe` takles an `int[2]` "out" parameter. First argument will be filled with the file descriptor for reading, and the second argument will be filled with the file descriptor for writing
- Pipes have finite size (about 4KB) and can get full. It cannot receive more writes until some data is read

Sharing data between processes

- `pipe` can be used to communicate between parent and child processes
- If you call `fork`, the file descriptors that `pipe` returns will be shared between the two processes

Open File Table

- The **Open File Table** is shared by all processes, and keeps track of position and refcount. This is how pipe can be shared



- `refcount` is used to keep track of how many file descriptors are pointing to the file; this is how it knows when to close the file
  - If you forget to call `close`, the file will not be closed until the program exits

Standard I/O

- All processes start with at least three file descriptors
  - 0: `stdin`
  - 1: `stdout`
  - 2: `stderr`

Redirecting I/O

- You can use `dup2` to redirect file descriptors

```
int dup2(int oldfd, int newfd)
```

- `dup2` closes `newfd` if it is open, and makes it refer to the same open file as `oldfd`

Unix I/O vs C Stdlib I/O

- Cstdlib I/O (`fread`, etc) is about 10x faster than Unix I/O (`read`, etc)

- Every time a Unix I/O function is called, a system call is made, requiring a context switch to the kernel and back
  - `fread` gets around this by doing something like `memcpy` to store the entire file in a buffer, so subsequent `fread`s don't need to do system calls

  Note:

  - This is why `fwrite` doesn't write until `\n` is reached, since all writes before it's reached are stored in the buffer
  - You can choose the buffer mode to choose when to flush the buffer—on every write, when out of space, or on newline