UNIX Command Shell Operation

- UNIX command shells generally follow this process:
  1. Wait for a command to be entered on the shell's standard input (usually entered by a user on the console, but not always!)
  2. Tokenize the command into an array of tokens
  3. If **tokens[0]** is an internal shell command (e.g. **history** or **export**) then handle the internal command, then go back to 1.
  4. Otherwise, **fork()** off a child process to execute the program. **wait()** for the child process to terminate, then go back to 1.

- Child process:
  1. If the parsed command specifies any redirection, modify **stdin/stdout/stderr** based on the command, and remove these tokens from the tokenized command
  2. **execve()** the program specified in **tokens[0]**, passing tokens as the program's arguments
  3. If we got here, execution failed (e.g. file not found)! Report error.

Command Shell and Child Process

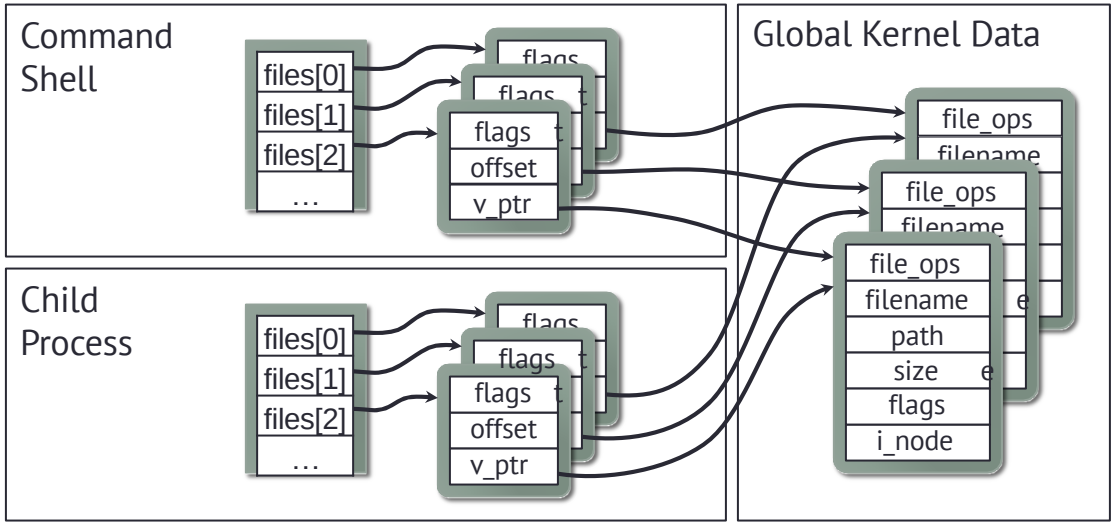
- How does the child process output to the command shell's standard output?
- How does it get the shell's stdin?

- When a UNIX process is forked, it is a near-identical copy of the parent process
  - Only differences: process ID and parent process ID

- Specifically, the child process has the same files open as the parent process
  - And they have the exact same file descriptors

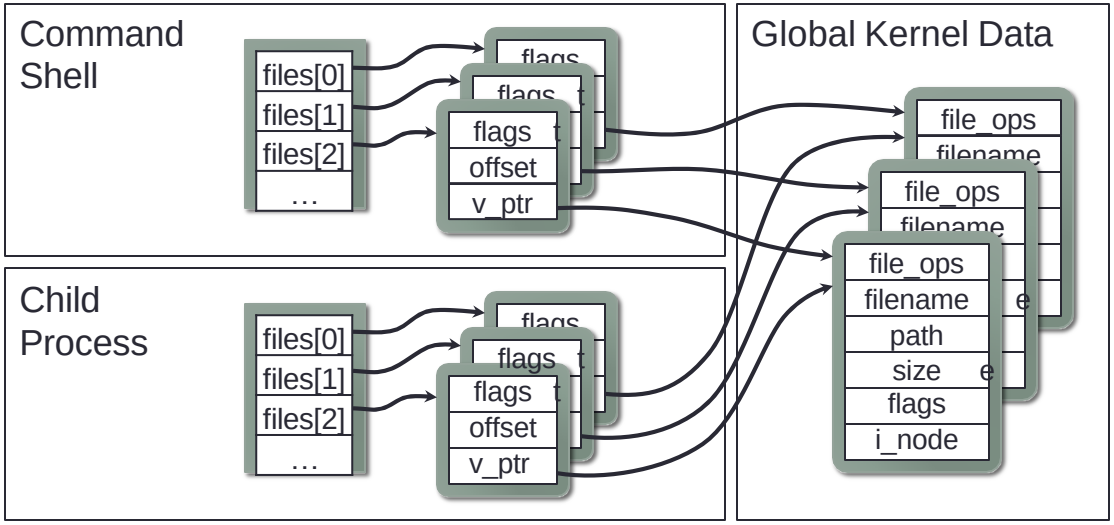
Command Shell and Child Process

When child process reads stdin and writes stdout/stderr, it writes the exact same files that the command-shell has as **stdin/stdout/stderr**



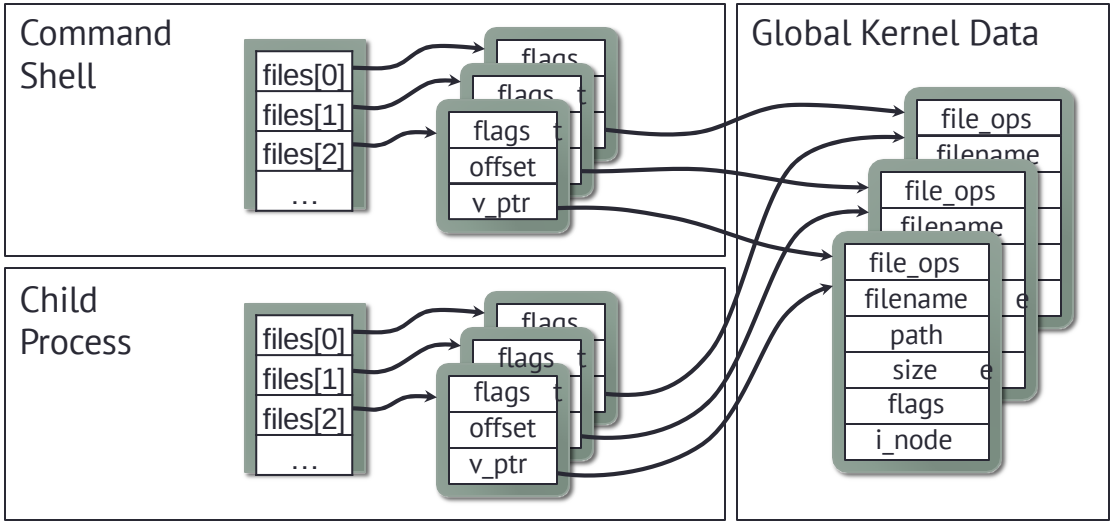
Command Shell and Child Process

- If command redirects e.g., the output to a file, clearly can't have the command- shell process do it before forking
  - Would work fine for the child process, but the command-shell's I/O state would be broken for subsequent commands



Command Shell and Child Process

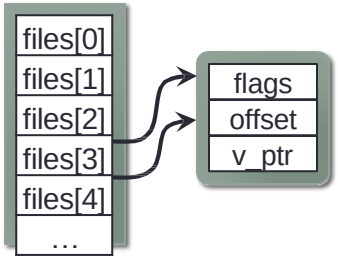
- Child process must set up **stdin, stdout,** and **stderr** before it executes the actual program
- How does a process change what file is referenced by a given file descriptor?
  - Process must ask the kernel to modify the file descriptors



Manipulating File Descriptors

- UNIX provides two system calls: **dup()** and **dup2()**

- **`int dup(int filedes)`**
  - Duplicates the specified file descriptor, returning a new, previously unused file descriptor
- Note that the internal **`file`** struct is **not** duplicated, only the pointer to the file struct!
- Implication:
  - Reads, writes and seeks through both file descriptors affect a single shared file-offset value
- Even though the one file has two descriptors, should call **close()** on each descriptor
  - Remember: each process has a maximum number of open files
  - (Kernel won't free the file struct until it has no more references)



Manipulating File Descriptors

- **`int dup2(int filedes, int filedes2)`**
  - Duplicates the specified file descriptor into the descriptor specified by **`filedes2`**
  - If filedes2 is already an open file, it is closed before **`dup2()`** duplicates **`filedes`**
  - (Unless **`filedes == filedes2`**, in which case nothing is closed)
- This function allows the command-shell's child process to redirect standard input and output
  - e.g. to replace **`stdout`** with a file whose descriptor is in **`fd`**:

    ```
    dup2(fd, STDOUT_FILENO);
    ```
- As before, the file descriptor that was duplicated should be closed to keep from leaking descriptors

  ```
  close(fd);
  ```

Manipulating File Descriptors

- Previous example:

```
grep Allow < logfile.txt > output.txt
```

- After command shell forks off a child process, the child can execute code like this, before it starts grep:

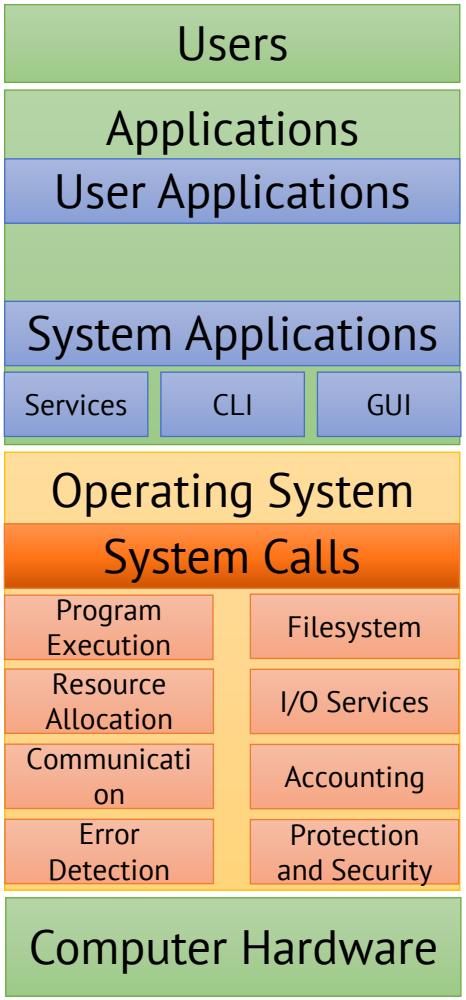
```
int in_fd, out_fd;

in_fd = open("logfile.txt", O_RDONLY);
/* Replace stdin */
dup2(in_fd, STDIN_FILENO);
close(in_fd);

out_fd = open("output.txt", O_CREAT | O_TRUNC | O_WRONLY, 0);
/* Replace stdout */
dup2(out_fd, STDOUT_FILENO);
close(out_fd);
```

Operating System Components

- Operating systems commonly provide these components:
- Applications can't access operating system state or code directly
  - OS code and state are stored in kernel space
  - Must be in kernel mode to access this data
  - Application code and state is in user space
- **How does the application interact with the OS?**



Operating Modes and Traps

- Typical solution: application issues a **trap** instruction
  - A trap is an intentional software-generated exception or interrupt
  - (as opposed to a fault; e.g. a divide-by-zero or general protection fault)
- During OS initialization, the kernel provides a handler to be invoked by the processor when the trap occurs
  - When the trap occurs, the processor can change the current protection level (e.g. switch from user mode to kernel mode)
- Benefits:
  - Applications can invoke operating system subroutines without having to know what address they live at
  - (and the location of OS routines can change when the OS is upgraded)
  - OS can verify applications' requests and prevent misbehavior
    - e.g. check arguments to system calls, verify process permissions, etc.

x86: Operating Modes and Traps

- On x86, programs use the int instruction to cause a software interrupt (a.k.a. software trap)
  - e.g. int $0x80 is used to make Linux system calls
- On x86, different operating modes have different stacks
  - Ensures the kernel won't be affected by misbehaving programs
- Arguments to the interrupt handler are passed in registers
  - Results also come back in registers

- Example:

```
movl $20, %eax # Get PID of current process
# Now %eax holds the PID of the current process

int $0x80      # Invoke system call!
```

- The operating systems provide wrappers for this

```
int pid = getpid(); /* Does the syscall for us */
```

x86: Interrupt Descriptor Table

- x86 uses an Interrupt Descriptor Table (IDT) to specify handlers for various interrupts
  - x86 supports 256 interrupts, so the IDT contains 256 entries
- In protected mode, each interrupt descriptor is 8 bytes
  - Specifies the address of the handler (6 bytes), plus flags (2 bytes)
  - (Privilege level that the handler runs at is specified elsewhere in the x86 architecture; will revisit this topic in the future)
- When a program issues an **`int n`** instruction:
  - Processor retrieves entry n in the Interrupt Descriptor Table
  - If the caller has at least the required privilege level (specified in the IDT entry), the processor transfers control to the handler
  - (if privilege level changes, the stack will be switched as well)
  - The handler runs at the privilege level it requires (e.g. kernel mode)

x86: Interrupt Descriptor Table

- When interrupt handler is finished, it issues an **`iret`**
  - Performs the same sequence of steps in reverse, ending up back at the caller's location and privilege level
    - (if the privilege level changes, the stack will be switched again)
- Location of Interrupt Descriptor Table can be set with the **`lidt`** x86 instruction
  - Can only be executed in kernel mode (level 0)
  - Operating system can configure the IDT to point to its entry-point(s)
- If you're wondering:
  - x86 processors start out in kernel mode when reset
  - Allows OS to perform initial processor configuration (interrupt handlers, virtual memory, etc.) before switching to user mode

Exceptional Control Flow

- **Exceptional** control flow is very important for many other operating system facilities
- Already saw **traps**
  - Intentional, software-generated exceptions
  - Frequently used to invoke operating system services
  - Returns to next instruction
- **Interrupts** are caused by hardware devices connected to the processor
  - Signals that a device has completed a task, etc.
  - The current operation is interrupted, and control switches to the OS
  - OS handles interrupt, then go back to what was being done before
  - Returns to next instruction

Exceptional Flow Control

- **Faults** are (usually unintentional) exceptions generated by attempting to execute a specific instruction
  - Signals a potentially recoverable error
    - e.g. a page fault generated by the MMU
    - e.g. a divide-by-zero fault generated by the ALU
  - Returns to the current instruction, if the OS can recover from fault!

- **Aborts** are nonrecoverable hardware errors
  - Often used to indicate severe hardware errors
    - e.g. memory parity errors, system bus errors, cache errors, etc.
  - Doesn’t return to interrupted operation

x86 Aborts

- x86 machine-check exception is an abort
  - Signaled when hardware detects a fatal error
- x86 also has a double-fault abort
- Scenario:
  - User program is running merrily along… then an interrupt occurs!
  - CPU looks in Interrupt Descriptor Table (configured by the OS) to dispatch to the interrupt handler
  - When CPU attempts to invoke the handler, a fault occurs!
    - e.g. a general protection fault, because the handler address was wrong
  - Double-fault indicates that a fault occurred during another fault
- Another scenario that causes a double-fault:
  - User program causes a page fault…
  - When CPU tries to invoke page-fault handler, it causes another page-fault to occur.

x86 Aborts

- Double-faults are signs of operating system bugs
  - It should be impossible for a user-mode program to cause one
- Of course, x86 allows operating systems to register a double-fault handler in the IDT (interrupt 8)
- If a fault occurs while attempting to invoke the double-fault handler, a triple- fault occurs (!!!)
- At this point, the CPU gives up.
  - Enters a shutdown state that can only be cleared by a hardware reset, or (in some cases) by certain kinds of specific interrupts

Multitasking and Hardware Timer

- Certain kinds of multitasking rely on hardware interrupts
- Early multitasking operating systems provided cooperative multitasking
  - Each process voluntarily relinquishes the CPU, e.g. when it blocks for an IO operation to complete, when it yields, or terminates
  - (This would implicitly occur when certain system calls were made)
- Problem:
  - Operating system is susceptible to badly designed programs that never relinquish the CPU (e.g. buggy or malicious program)
- A number of OSes only provided cooperative multitasking:
  - Windows before Win95
  - MacOS before MacOS X
  - Many older operating systems (but generally not UNIX variants!)

Multitasking and Hardware Timer

- Solution: incorporate a **hardware timer** into the computer
- OS configures the timer to trigger after a specific amount of time
  - When time runs out, the hardware timer fires an interrupt
  - OS handles the interrupt by performing a context-switch to another process, then starting the timer again
- Operating systems that can interrupt the currently running process provide **preemptive multitasking**
- Obviously, all hardware timer configuration must require kernel-mode access
  - Otherwise, a process could easily take all the time it wants
- Virtually all widely used OSes now have preemption
  - (All UNIX variants have basically always had preemption)