



# Basic OS Programming Abstractions

---

Don Porter  
CSE 306

# Recap



- ✦ We've introduced the idea of a process as a container for a running program
- ✦ And we've discussed the hardware-level mechanisms to transition between the OS and applications (interrupts)
- ✦ This lecture: Introduce key OS APIs
  - ✦ Some may be familiar from lab 1
  - ✦ Others will help with lab 2

# Outline



- ✦ Files and File Handles
- ✦ Inheritance
- ✦ Pipes
- ✦ Sockets
- ✦ Signals
- ✦ Synthesis Example: The Shell

# 2 Ways to Refer to a File



- ✦ Path, or hierarchical name, of the file
  - ✦ Absolute: “/home/porter/foo.txt”
    - ✦ Starts at system root
  - ✦ Relative: “foo.txt”
    - ✦ Assumes file is in the program’s current working directory
- ✦ Handle to an open file
  - ✦ Handle includes a cursor (offset into the file)

# Path-based calls



- ✦ Functions that operate on the directory tree
  - ✦ Rename, unlink (delete), chmod (change permissions), etc.
- ✦ Open – creates a handle to a file
  - ✦ `int open (char *path, int flags, mode_t mode);`
    - ✦ Flags include `O_RDONLY`, `O_RDWR`, `O_WRONLY`
    - ✦ Permissions are generally checked only at open
  - ✦ `Opendir` – variant for a directory



# Handle-based calls



- ✦ `ssize_t read (int fd, void *buf, size_t count)`
  - ✦ Fd is the handle
  - ✦ Buf is a user-provided buffer to receive count bytes of the file
  - ✦ Returns how many bytes read
- ✦ `ssize_t write(int fd, void *buf, size_t count)`
  - ✦ Same idea, other direction
- ✦ `int close (int fd)`
  - ✦ Close an open file

# Example



```
char buf[9]; // stack allocate a char buffer

int fd = open ("foo.txt", O_RDWR);

ssize_t bytes = read(fd, buf, 8);

if (bytes != 8) // handle the error

memset (buf, "Awesome", 7);

buf[7] = '\0';

bytes = write(fd, buf, 8);

if (bytes != 8) // error

close(fd);
```

# But what is a handle?



- ✦ A reference to an open file or other OS object
  - ✦ For files, this includes a cursor into the file
- ✦ In the application, a handle is just an integer
  - ✦ This is an offset into an OS-managed table



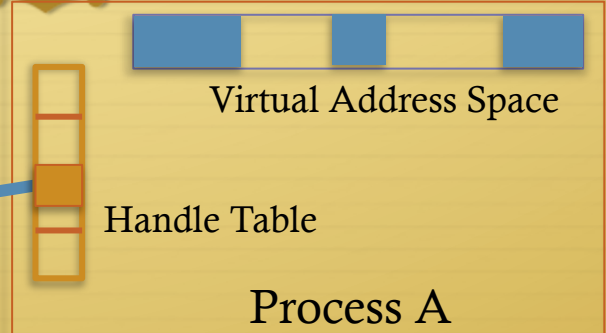
# View

Handles can be shared

Hello!

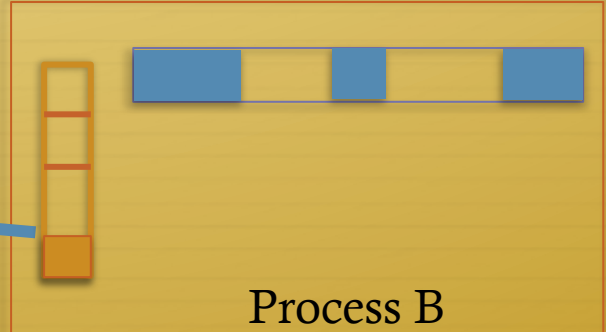
Foo.txt inode

50

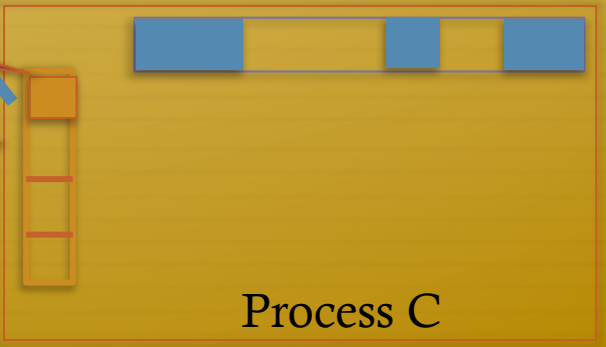


Handle indices are process-specific

20



Handle Table



# Handle Recap



- ✦ Every process has a table of pointers to kernel handle objects
  - ✦ E.g., a file handle includes the offset into the file and a pointer to the kernel-internal file representation (inode)
- ✦ Application's can't directly read these pointers
  - ✦ Kernel memory is protected
  - ✦ Instead, make system calls with the indices into this table
  - ✦ Index is commonly called a handle

# Rearranging the table



- ✦ The OS picks which index to use for a new handle
- ✦ An application explicitly copy an entry to a specific index with `dup2(old, new)`
  - ✦ Be careful if new is already in use...

# Other useful handle APIs



- ✦ We've seen mmap already; can map part or all of a file into memory
- ✦ seek() – adjust the cursor position of a file
  - ✦ Like rewinding a cassette tape

# Outline



- ✦ Files and File Handles
- ✦ Inheritance
- ✦ Pipes
- ✦ Sockets
- ✦ Signals
- ✦ Synthesis Example: The Shell

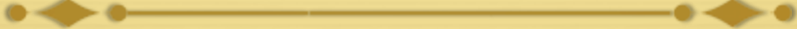


# Inheritance



- ✦ By default, a child process gets a copy of every handle the parent has open
  - ✦ Very convenient
  - ✦ Also a security issue: may accidentally pass something the program shouldn't
- ✦ Between `fork()` and `exec()`, the parent has a chance to clean up handles it doesn't want to pass on
  - ✦ See also `CLOSE_ON_EXEC` flag

# Standard in, out, error



- ✦ Handles 0, 1, and 2 are special by convention
  - ✦ 0: standard input
  - ✦ 1: standard output
  - ✦ 2: standard error (output)
- ✦ Command-line programs use this convention
  - ✦ Parent program (shell) is responsible to use `open/close/dup2` to set these handles appropriately between `fork()` and `exec()`

# Example



```
int pid = fork();  
if (pid == 0) {  
    int input = open ("in.txt", O_RDONLY);  
    dup2(input, 0);  
    exec ("grep", "quack");  
}  
  
//...
```

# Outline



- ✦ Files and File Handles
- ✦ Inheritance
- ✦ Pipes
- ✦ Sockets
- ✦ Signals
- ✦ Synthesis Example: The Shell

# Pipes



- ✦ Stream of bytes between two processes
- ✦ Read and write like a file handle
  - ✦ But not anywhere in the hierarchical file system
  - ✦ And not persistent
  - ✦ And no cursor or seek()-ing
  - ✦ Actually, 2 handles: a read handle and a write handle
- ✦ Primarily used for parent/child communication
  - ✦ Parent creates a pipe, child inherits it



# Example



```
int pipe_fd[2];

int rv = pipe(pipe_fd);

int pid = fork();

if (pid == 0) {

    close(pipe_fd[1]); //Close unused write end

    dup2(pipe_fd[0], 0); // Make the read end stdin

    exec("grep", "quack");

} else {

    close (pipe_fd[0]); // Close unused read end ...
```

# Sockets



- ✦ Similar to pipes, except for network connections
- ✦ Setup and connection management is a bit trickier
  - ✦ A topic for another day (or class)

# Select



- ✦ What if I want to block until one of several handles has data ready to read?
- ✦ Read will block on one handle, but perhaps miss data on a second...
- ✦ Select will block a process until a handle has data available
  - ✦ Useful for applications that use pipes, sockets, etc.

# Outline



- ✦ Files and File Handles
- ✦ Inheritance
- ✦ Pipes
- ✦ Sockets
- ✦ Signals
- ✦ Synthesis Example: The Shell

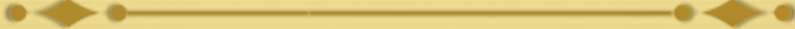
# Signals



- ✦ Similar concept to an application-level interrupt
  - ✦ Unix-specific (more on Windows later)
- ✦ Each signal has a number assigned by convention
  - ✦ Just like interrupts
- ✦ Application specifies a handler for each signal
  - ✦ OS provides default
- ✦ If a signal is received, control jumps to the handler
  - ✦ If process survives, control returns back to application



# Signals, cont.



- ✦ Can occur for:
  - ✦ Exceptions: divide by zero, null pointer, etc.
  - ✦ IPC: Application-defined signals (USR1, USR2)
  - ✦ Control process execution (KILL, STOP, CONT)
- ✦ Send a signal using `kill(pid, signo)`
  - ✦ Killing an errant program is common, but you can also send a non-lethal signal using `kill()`
- ✦ Use `signal()` or `sigaction()` to set the handler for a signal

# How signals work



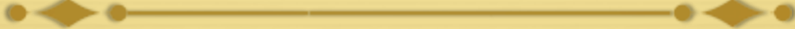
- ✦ Although signals appear to be delivered immediately...
  - ✦ They are actually delivered lazily...
  - ✦ Whenever the OS happens to be returning to the process from an interrupt, system call, etc.
- ✦ So if I signal another process, the other process may not receive it until it is scheduled again
- ✦ Does this matter?

# More details



- ✦ When a process receives a signal, it is added to a pending mask of pending signals
  - ✦ Stored in PCB
- ✦ Just before scheduling a process, the kernel checks if there are any pending signals
  - ✦ If so, return to the appropriate handler
  - ✦ Save the original register state for later
  - ✦ When handler is done, call sigreturn() system call
    - ✦ Then resume execution

# Meta-lesson



- ✦ Laziness rules!
  - ✦ Not on homework
  - ✦ But in system design
- ✦ Procrastinating on work in the system often reduces overall effort
  - ✦ Signals: Why context switch immediately when it will happen soon enough?

# Language Exceptions



- ✦ Signals are the underlying mechanism for Exceptions and catch blocks
- ✦ JVM or other runtime system sets signal handlers
  - ✦ Signal handler causes execution to jump to the catch block



# Windows comparison



- ✦ Exceptions have specific upcalls from the kernel to ntdll
- ✦ IPC is done using Events
  - ✦ Shared between processes
  - ✦ Handle in table
  - ✦ No data, only 2 states: set and clear
  - ✦ Several variants: e.g., auto-clear after checking the state

# Outline



- ✦ Files and File Handles
- ✦ Inheritance
- ✦ Pipes
- ✦ Sockets
- ✦ Signals
- ✦ Synthesis Example: The Shell

# Shell Recap



- ✦ Almost all ‘commands’ are really binaries
  - ✦ /bin/ls
- ✦ Key abstraction: Redirection over pipes
  - ✦ ‘>’, ‘<’, and ‘|’ implemented by the shell itself

# Shell Example



✦ Ex: `ls | grep foo`

✦ Implementation sketch:

✦ Shell parses the entire string

✦ Sets up chain of pipes

✦ Forks and exec's 'ls' and 'grep' separately

✦ Wait on output from 'grep', print to console

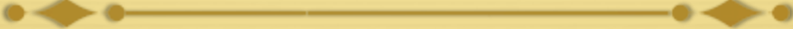
# What about Ctrl-Z?



- ✦ Shell really uses `select()` to listen for new keystrokes
  - ✦ (while also listening for output from subprocess)
- ✦ Special keystrokes are intercepted, generate signals
  - ✦ Shell needs to keep its own “scheduler” for background processes
  - ✦ Assigned simple numbers like 1, 2, 3
- ✦ ‘fg 3’ causes shell to send a `SIGCONT` to suspended child



# Other hints



- ✦ `splice()`, `tee()`, and similar calls are useful for connecting pipes together
  - ✦ Avoids copying data into and out-of application

# Summary



- ✦ Understand how handle tables work
  - ✦ Survey basic APIs
- ✦ Understand signaling abstraction
  - ✦ Intuition of how signals are delivered
- ✦ Be prepared to start writing your shell in lab 2!