Programming Interface
Main Points

- Creating and managing processes
  - fork, exec, wait

- Performing I/O
  - open, read, write, close

- Communicating between processes
  - pipe, dup, select, connect

- Example: implementing a shell
Shells

• A shell is a job control system
  – Allows programmer to create and manage a set of programs to do some task
  – Windows, MacOS, Linux all have shells
    • Desktop vs. Shell?

• Example: to compile a C program
  $ cc -c sourcefile1.c
  $ cc -c sourcefile2.c
  $ ln -o program sourcefile1.o sourcefile2.o
Questions

• If the compiler (cc) crashes, does the shell crash?

• If the shell crashes, does the compiler run to completion?
Basic Shell Operation

• Shells implement some commands, but primarily they launch new processes
  – `cc -c sourcefile1.c`
    Starts a new process that (a) executes “cc” and (b) is passed [-c, sourcefile1.c] as arguments.

• What system call(s) are required to create a new process running some executable?
Windows: `CreateProcess`

- System call to create a new process to run a program
  - Create and initialize the process control block (PCB) in the kernel
  - Create and initialize a new address space
  - Load the program into the address space
  - Copy arguments into memory in the address space
  - Initialize the hardware context to start execution at `"start"`
  - Inform the scheduler that the new process is ready to run
Windows CreateProcess API (simplified)

if (!CreateProcess(
    NULL,           // No module name (use command line arg)
    argv[1],        // Command line
    NULL,           // Process handle not inheritable
    NULL,           // Thread handle not inheritable
    FALSE,          // Set handle inheritance to FALSE
    0,                // No creation flags
    NULL,           // Use parent's environment block
    NULL,           // Use parent's starting directory
    &si,                // Pointer to STARTUPINFO structure
    &pi )            // Pointer to PROCESS_INFORMATION structure
) { // success
UNIX Process Management

• **fork** – system call to create a copy of the current process, and start it running
  – No arguments!

• **exec** – system call to change the program being run by the current process

• **wait** – system call to wait for a process to finish

• **signal/kill** – system calls to register a handler for a signal and to send a signal to another process
Question: What does this code print?

```c
int child_pid = fork();
if (child_pid == 0) { // I'm the child process
    printf("I am process #\%d\n", getpid());
    return 0;
} else { // I'm the parent process
    printf("I am parent of process #\%d\n", child_pid);
    return 0;
}
```
Questions

• Can UNIX fork() return an error? Why?

• Can UNIX exec() return an error? Why?

• Can UNIX wait() ever return immediately? Why?
Implementing UNIX `fork`

Steps to implement UNIX `fork`

- Create and initialize the process control block (PCB) in the kernel
- Create a new address space
- Initialize the address space with a copy of the entire contents of the address space of the parent
- Inherit the execution context of the parent (e.g., any open files)
- Inform the scheduler that the new process is ready to run
Implementing UNIX exec

- Steps to implement UNIX exec
  - Load the executable into the current address space (overwriting what’s already there)
  - Copy arguments into the address space
  - Initialize the hardware context to start execution at ``start``
UNIX I/O

• Uniformity
  – All operations on all files, devices use the same set of system calls: open, close, read, write

• Open before use
  – Open returns a handle (file descriptor) for use in later calls on the file

• Byte-oriented

• Kernel-buffered read/write

• Explicit close
  – To garbage collect the open file descriptor
UNIX File System Interface

• UNIX file open is a Swiss Army knife:
  – Open the file, return file descriptor (an int)
  – Options:
    • if file doesn’t exist, return an error
    • If file doesn’t exist, create file and open it
    • If file does exist, return an error
    • If file does exist, open file
    • If file exists but isn’t empty, nix it then open
    • If file exists but isn’t empty, return an error
    • ...

Interface Design Question

• Why not separate syscalls for open/create/exists?

```c
if (!exists(name))
    create(name);  // can create fail?
fd = open(name);    // does the file exist?
```
Implementing a Shell

char *prog, **args;
int child_pid;

// Read and parse the input a line at a time
while (readAndParseCmdLine(&prog, &args)) {
    child_pid = fork();  // create a child process
    if (child_pid == 0) {
        exec(prog, args);  // I'm the child process. Run program
        // NOT REACHED
    } else {
        wait(child_pid);    // I'm the parent, wait for child
    }
}