Outline

- System Calls: fork, exec, exit, wait, open, read, write, close, dup, pipe
- Case Study: Unix/xv6 shell (simplified)

System Calls

- UNIX's `fork()` and `exec()` versus Windows' `CreateProcess()`
- UNIX process inheritance and file sharing

Case study: Unix/xv6 shell (simplified)

```
while (1) {
    write (1, "$ ", 2);   // 1 = STDOUT_FILENO
    readcommand (0, command, args); // parse user input, 0 = STDIN_FILENO
    if ((pid = fork ()) == 0) {  // child?
 exec (command, args, 0);
    } else if (pid > 0) {  // parent?
 wait (0);   // wait for child to terminate
    } else {
 perror ("Failed to fork\n");
    }
}
```

- why call "wait"? to wait for the child to terminate and collect its exit status. (if child finishes, child becomes a zombie until parent calls wait.). Zombies are required as a parent may need the exit status later. What happens if the parent exits before the child? Attach to init process.
- Example:

  ```
  $ ls
  ```

- How does the shell implement:

  ```
  $ ls > tmp1
  ```

  just before exec insert:

  ```
  close(1);
  creat("tmp1", 0666);   // fd will be 1
  ```

  The kernel always uses the first free file descriptor, 1 in this case. Could use `dup2()` to clone a file descriptor to a new number.

- Good illustration for why fork + exec vs. CreateProcess on Windows. (CreateProcess takes 10 arguments.)

  The split of process creation into fork and exec turns out to have been an inspired choice, though that might not have been clear at the time; see today's assigned paper.

- What if you run the shell itself with redirection?

  ```
  $ sh < script > tmp1
  ```

  If for example the file `script` contains

  ```
  echo one
  echo two
  ```

  FD inheritance makes this work well.
- Discuss how the same interface can be used for both devices and files; and how the OS now needs to keep track of the file-offset internally due to this interface.
- Discuss how the offset is a property of the resource, and not of the file-descriptor. Being a property of the resource allows sharing of offsets (thus producing interleaving effects). Separate offsets would preclude interleaving completely. On the other hand, separate offsets can still be created with the current abstractions.
- What if we want to redirect multiple FDs (stdout, stderr) for programs that print to both?

  ```
  $ ls f1 f2 nonexistant-f3 > tmp1 2>&1
  ```

  after creat, insert:

  ```
  close(2);
  creat("tmp1", 0666);   // fd will be 2
  ```

  why is this bad? illustrate what's going on with file descriptors. better:

  ```
  close(2);
  dup(1);         // fd will be 2
  ```

(Read Section 3 of "Advanced Programming in the UNIX environment" by W. R. Stevens, esp. Section 3.10) or in bourne shell syntax,

```
$ ls f1 f2 nonexistant-f3 > tmp1 2>&1
```

Read Chapter 3 of *Advanced Programming in the UNIX Environment* by W. Richard Stevens for a detailed understanding of how file descriptors are implemented. In particular, read Section 3.10 to understand how file sharing works.

- how to run a series of programs on some data?

```
$ sort < file.txt > tmp1
$ uniq tmp1 > tmp2
$ wc tmp2
$ rm tmp1 tmp2
```

can be more concisely done as:

```
$ sort < file.txt | uniq | wc
```

Draw a figure with boxes as programs, and each box containing two ports (STDIN/0 and STDOUT/1). Advantages of second option: no temporary space, no extra reads/writes to the disk, scheduling visibility to OS

- A pipe is a one-way communication channel. Here is a simple example:

```
int fdarray[2];
char buf[512];
int n;

pipe(fdarray);
write(fdarray[1], "hello", 5);
n = read(fdarray[0], buf, sizeof(buf));
// buf[] now contains 'h', 'e', 'l', 'l', 'o'
```

- file descriptors are inherited across `fork()`, so this also works:

```
int fdarray[2];
char buf[512];
int n, pid;

pipe(fdarray);
pid = fork();
if(pid > 0){
  write(fdarray[1], "hello", 5);
} else {
  n = read(fdarray[0], buf, sizeof(buf));
}
```

- How does the shell implement pipelines (i.e., cmd 1 | cmd 2 |..)? We want to arrange that the output of cmd 1 is the input of cmd 2. The way to achieve this goal is to manipulate stdout and stdin.
- The shell creates processes for each command in the pipeline, hooks up their stdin and stdout, and waits for the last process of the pipeline to exit. Here's a sketch of what the shell does, in the child process of the `fork()` we already have, to set up a pipe:

```
int fdarray[2];

  if (pipe(fdarray) < 0) panic ("error");
if ((pid = fork ()) == 0) {  child (left end of pipe)
   close (1);
   tmp = dup (fdarray[1]);   // fdarray[1] is the write end, tmp will be 1
   close (fdarray[0]);       // close read end
   close (fdarray[1]);       // close fdarray[1]
   exec (command1, args1, 0);
} else if (pid > 0) {        // parent (right end of pipe)
   close (0);
   tmp = dup (fdarray[0]);   // fdarray[0] is the read end, tmp will be 0
   close (fdarray[0]);
   close (fdarray[1]);       // close write end
   exec (command2, args2, 0);
} else {
   printf ("Unable to fork\n");
       }
```

- Who waits for whom? (draw a tree of processes)
- Why close read-end and write-end? ensure that every process starts with 3 file descriptors, and that reading from the pipe returns end of file after the first command exits.