Unix I/O

CS 105: Computer Systems
Lecture 17

Melissa O'Neill

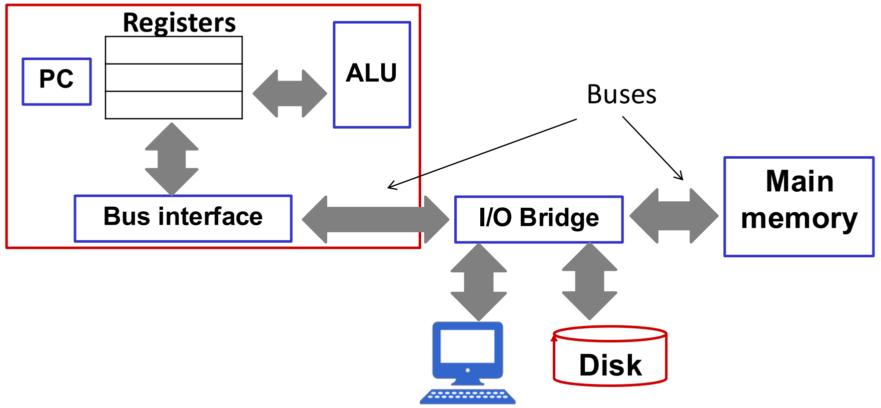
March 30, 2026

Learning Goals

- **Understand how Unix-based operating systems represent files**
- **Contrast Unix I/O and the C Standard I/O library, particularly investigating the impact of buffered I/O**
- **Reason about how open files are shared between parent and child processes**

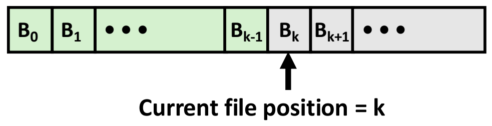
Recall: system architecture

- **Registers: fast**
- **Main memory: slow**
- **I/O devices (hard disk, `stdin`, etc.): really, really, really slow**



Unix I/O Overview

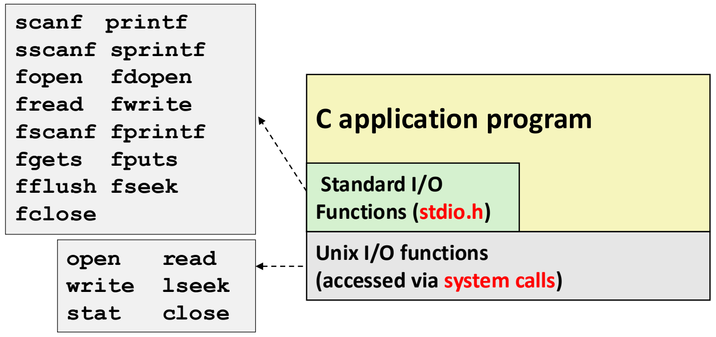
- **Elegant mapping of files to devices allows kernel to export a simple interface called Unix I/O**
  - **Key idea**: All input and output is handled in a consistent and uniform way
- **A Unix *file* is a sequence of *m* bytes:**



- **Each file has a *type* indicating its role in the system:**
  - ***Regular file:*** Contains arbitrary data
  - ***Directory:*** Index for a related group of files
  - ***Socket:*** For communicating with a process on another machine
  - Symbolic link
  - *etc.*

Standard I/O *vs.* Unix I/O

- **C Standard I/O *implemented using* low-level Unix I/O**



```
scanf  printf
sscanf sprintf
fopen  fdopen
fread  fwrite
fscanf fprintf
fgets  fputs
fflush fseek
fclose
```

```
open   read
write  lseek
stat   close
```

C application program
Standard I/O Functions (stdio.h)
Unix I/O functions (accessed via system calls)

- **Standard I/O is *buffered*, Unix I/O is not**

Unix I/O Interface: system level (NOT buffered!)

- **Opening and closing files**
  - `open()` and `close()`
- **Reading and writing a file**
  - `read()` and `write()`
  - After read or write, **file position** is updated
- **Changing the current position in a file (optional)**
  - `lseek()`
- **Big idea: file descriptors**
  - Small integers refer to open files
  - An ID, much like PIDs, but start at 0 and count forwards

Opening Files – Unix I/O

- **Opening a file informs the kernel that you are getting ready to access that file**
- **Example using Unix I/O:**

```
int fd; /* file descriptor */
fd = open("test.dat", O_RDONLY); /* read only */
```

- **Typically, each process created by a Linux shell begins with three open files associated with a terminal:**
  - 0: standard input (**stdin**)
  - 1: standard output (**stdout**)
  - 2: standard error (**stderr**)

Opening Files – Standard I/O

- **Standard I/O opens a file with `fopen`**

```
FILE *p; /* pointer to FILE object */
p = fopen("test.dat","r"); /* read only */
```

filename
will check current directory
better to use full pathname
(big source of confusion)

mode
"r" read
"w" write
"a" append
etc.

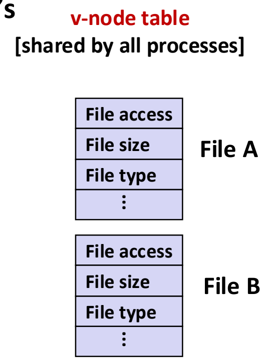
- **Standard I/O models open files as streams**
  - Abstraction for a file descriptor and a ***buffer*** in memory
- **`fopen` returns `FILE*`, can use to find the file descriptor**
  - Check `p == NULL` to see if an error occurred

`fileno(p) → fd`

Kernel data structures

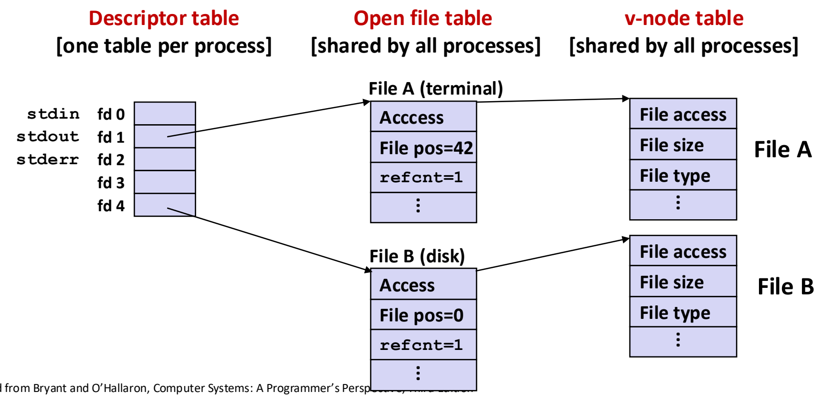
- The OS kernel maintains information about
  - currently open files
  - each process' connections to those files

1. Each open file has an entry in the kernel's v-node table
   - Entry might describe file on disk, the terminal, etc.



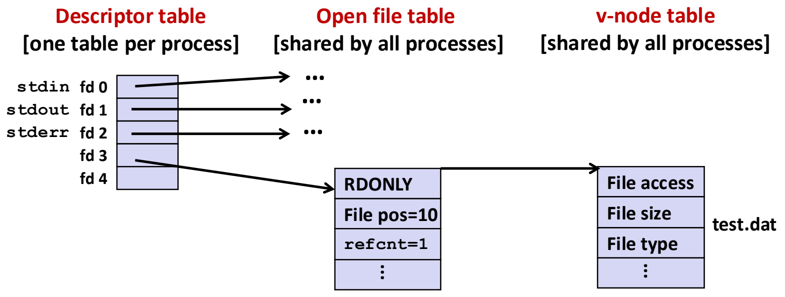
Kernel data structures

2. Open file table has an entry for each *instance* of an open file
   - Entry includes access mode, file position, and reference count (refcnt)
3. A descriptor table per process links descriptors to open files



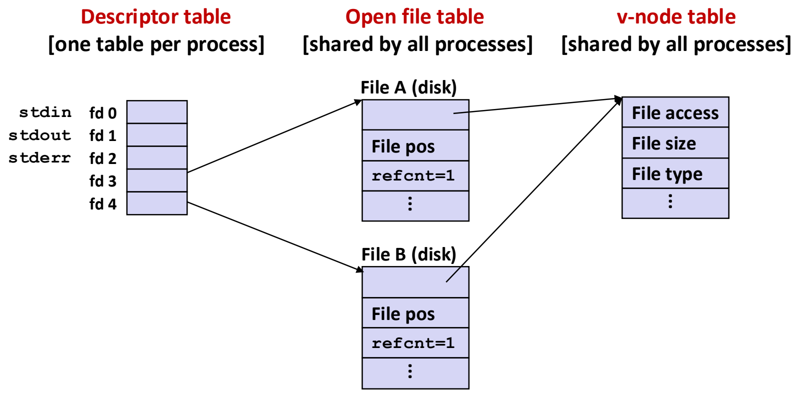
Exercise

- Consider the state of open files below. Suppose another `fopen` call happens that also opens `test.dat` for reading.
- Update the diagram below: how could descriptor table and open file table look *after* the `fopen`? *Hint: a newly opened file starts at file position 0.*



Open-file sharing within a process

- Two distinct descriptors sharing the same disk file through two distinct open file table entries
  - E.g., Calling `open` twice with the same `filename` argument

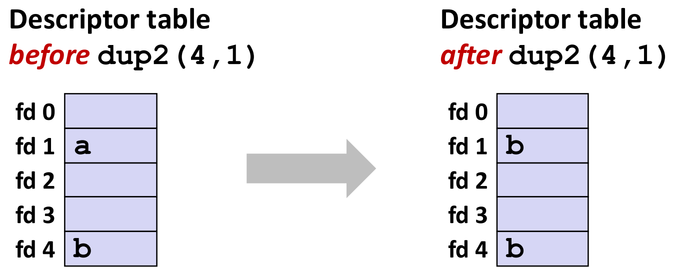


Redirecting I/O

```
%> objdump -d myProg > out.txt
```

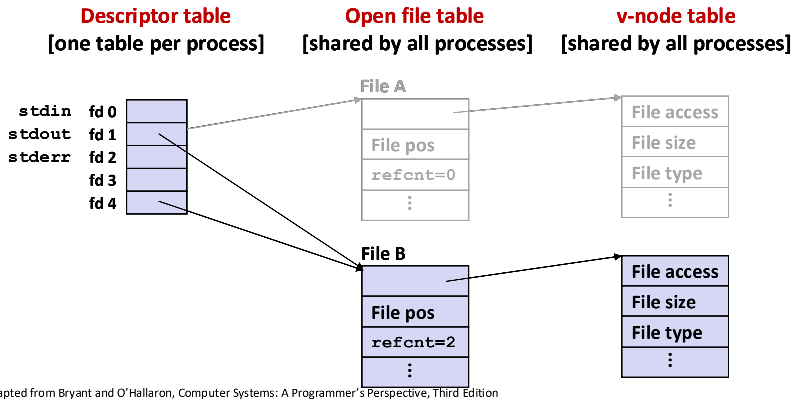
```
…
movq %rsp, %rdi
…
```

- How does a shell implement I/O redirection?
  - By calling the `dup2(oldfd, newfd)` function
    Copies descriptor table entry `oldfd` to entry `newfd`



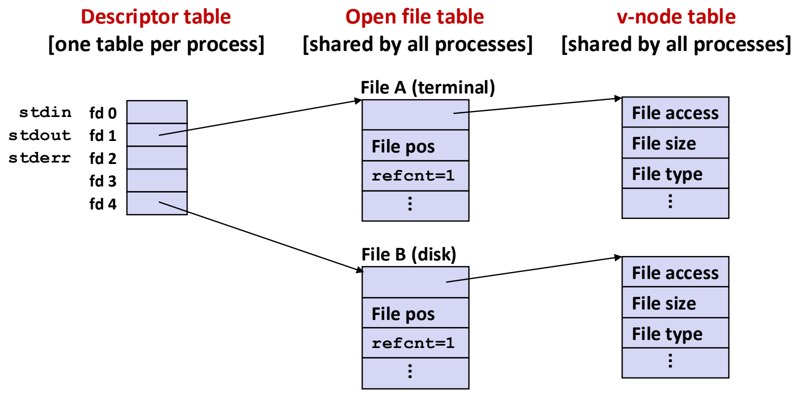
I/O Redirection Example

- Step #1: open the file to which `stdout` should be redirected
- Step #2: call `dup2(4,1)`
  - cause fd=1 (stdout) to refer to disk file pointed at by fd=4



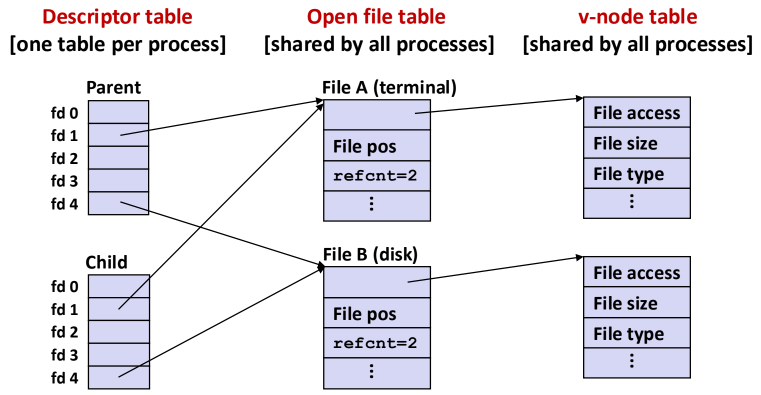
How Processes Share Files: `fork`

- A child process inherits its parent's open files
- *Before* `fork` call:



How Processes Share Files: `fork`

- A child process inherits its parent's open files
- *After* `fork`:
  - Child's table same as parent's, and +1 to each refcnt

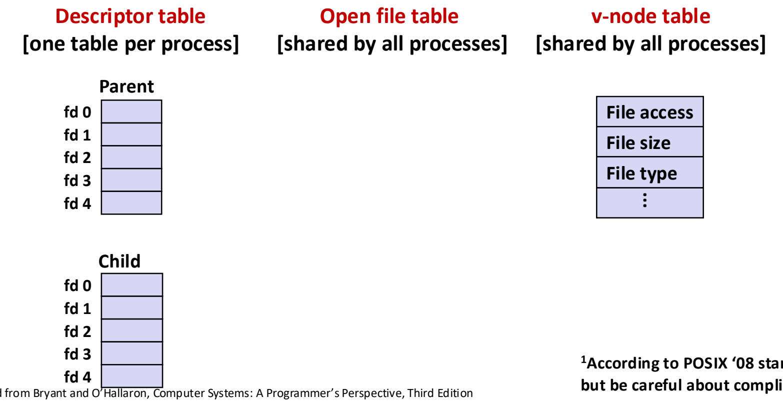


Example: Unix write() and fork()

```
/* includes omitted for space*/

int main()
{
  int childStatus;
  pid_t pval=0;

  int fd = open("output.txt", O_WRONLY|O_CREAT, 0644);
  /* error-checking omitted for space */

  pval = fork(); /* fork a child process */

  if (pval==0){ /* Child */
    if(write(fd, "Hello from child\n",18) != 18)
      perror("Error writing");
  }
  else { /* Parent */
    if(write(fd, "Hello... from parent\n",22) != 22)
      perror("Error writing");
    wait(&childStatus); /* wait for child to finish */
  }
  return 0;
}
```

Exercise

- Suppose the `open` yields `fd=3` with file position 0
  - Complete the drawing of the descriptor and open file tables below showing after both child and parent finish writing.
    You can omit parts for stdin, stdout, stderr.
    Assume a `write` updates file position immediately[1]



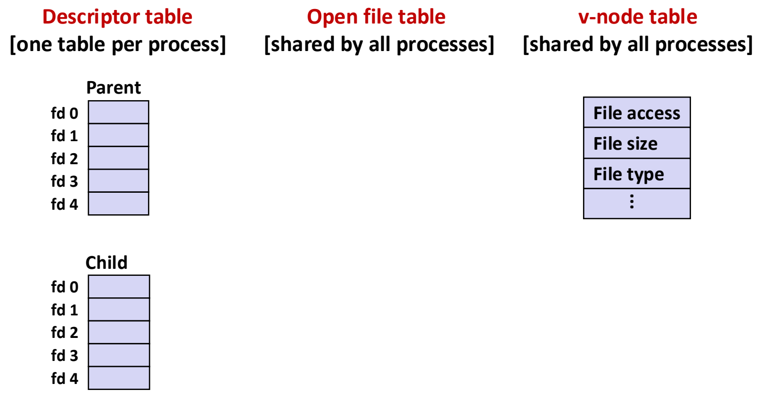
[1]According to POSIX '08 standard, but be careful about compliance

Example: Unix write() and fork() -- take 2

```
/* includes omitted for space*/

int main()
{
  int childStatus;
  pid_t pval=0;

  pval = fork(); /* fork a child process */

  int fd = open("output.txt", O_WRONLY|O_CREAT, 0644);
  /* error-checking omitted for space */

  if (pval==0){ /* Child */
    if(write(fd, "Hello from child\n",18) != 18)
      perror("Error writing");
  }
  else { /* Parent */
    if(write(fd, "Hello... from parent\n",22) != 22)
      perror("Error writing");
    wait(&childStatus); /* wait for child to finish */
  }
  return 0;
}
```

Exercise

- **The "take 2" version of the code calls `fork` *before* the `open`**
  - How will your drawing change? What is it like after the processes write?
  - Thoughts on the implications of this code?



Buffered I/O Example: `printf` to console

```
#include <stdio.h>
#include <unistd.h>

int main() {
  printf("Time to sleep...");
  sleep(5); /* sleep for 5 sec */

  printf("\nDone!\n"); /* \n is newline */
  return 0;
}
                                        print.c
```

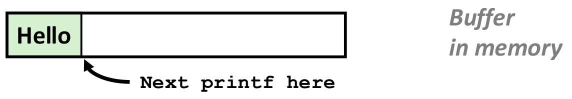
- **When will "Time to sleep…" appear on the console?**
  - Before sleeping?
  - After sleeping?

Buffered I/O: Motivation

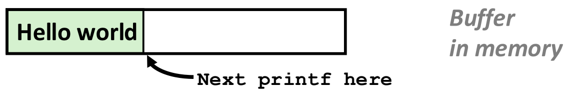
- **Applications often read/write a few characters at a time**
  - Using Unix I/O *expensive*… `read` and `write` to/from disk, `stdin/stdout`, etc. are **system calls** that can take > 10,000 clock cycles
- **Observation: reading/writing a single byte is not much cheaper than reading/writing a large block of data**

Example: Buffered Write

- **Accumulate writes to a buffer in memory**
  - Library function (e.g., `printf`) writes content to a fixed-size array in main memory (a "buffer").
  - After `printf("Hello ")`



  - After `printf("world")`



- **Standard I/O function uses Unix I/O to write buffer**
  - When buffer is full (or sooner), **Standard I/O library uses Unix `write`** to *flush* the buffer to the output device all at once (e.g., to disk or `stdout`)

Buffered writes: when is buffer *flushed*?

- **Three choices of when buffer is written to output**
  - Unbuffered
  - Block buffered – flush after some number of bytes/characters
  - Line buffered – flush on "enter" or newline (`\n`)
- **Common defaults**
  - If file is interactive device: **line buffered**
  - Other files: **block buffered**
  - Error streams tend to be **unbuffered**
- **Can also force a flush after a write using `fflush`**

Example: `printf` to console (revisited)

```
#include <stdio.h>
#include <unistd.h>

int main() {
  printf("Time to sleep...");
  sleep(5); /* sleep for 5 sec */

  printf("\
  return 0;
}
```

Only difference is newline!

```
#include <stdio.h>
#include <unistd.h>

int main() {
  printf("Time to sleep...\n");
  sleep(5); /* sleep for 5 sec */

  printf("\nDone!\n"); /* \n is newline */
  return 0;
}
                                     printnl.c
```

Exercise: buffered read

Consider the code on the next slide. Note that `fscanf` is a standard I/O function for reading from a file. After reading the code, answer the following questions.

1. What could the program print for a file containing: `abcdef`
 (Assume block buffering is being used, buffer size > 100.)

2. Suppose we turned *off the use of buffering* in the program (using the function `setvbuf`). Now what could the program print, assuming the same input file?

```
/* include statements omitted */
int main(int argc, char *argv[])
{
  FILE* p;
  int sleeptime = getpid() & 0x1; /* how long to sleep */
  char c1, c2;
  char *fname = argv[1];
  p = fopen(fname, "r"); /* open file for reading */

  fscanf(p, "%c", &c1); /* read first char from the file */

  if (fork()) { /* Parent */
    sleep(sleeptime);
    fscanf(p, "%c", &c2); /* read another char */

    printf("Parent: c1 = %c, c2 = %c\n", c1, c2);

  } else { /* Child */
    sleep(1-sleeptime);
    fscanf(p, "%c", &c2); /* read another char */
    printf("Child: c1 = %c, c2 = %c\n", c1, c2);
  }
  return 0;
}
```