# System Programming

## Processes, memory and communication

Thomas Ropars

thomas.ropars@univ-grenoble-alpes.fr

2025

The slides are available at:

`https://m1-mosig-os.gitlab.io/`

# References

Main references:

- *Advanced Programming in the Unix Environment* by R. Stevens
- *The Linux Programming Interface* by M. Kerrish
- *Operating Systems: Three Easy Pieces* by R. Arpaci-Dusseau and A. Arpaci-Dusseau (introduction chapter)
- *Computer Systems: A Programmer's Perspective* by R. Bryant and D. O'Hallaron

The content of these lectures is inspired by:

- The lecture notes of Prof. J.F. Mehaut.
- The lecture notes of R. Lachaize.

# Agenda

What is an operating system?

Unix File System

The Shell

Processes

Inputs/Outputs

# Agenda

What is an operating system?

Unix File System

The Shell

Processes

Inputs/Outputs

# Purpose of an operating system

Operations run by a program:

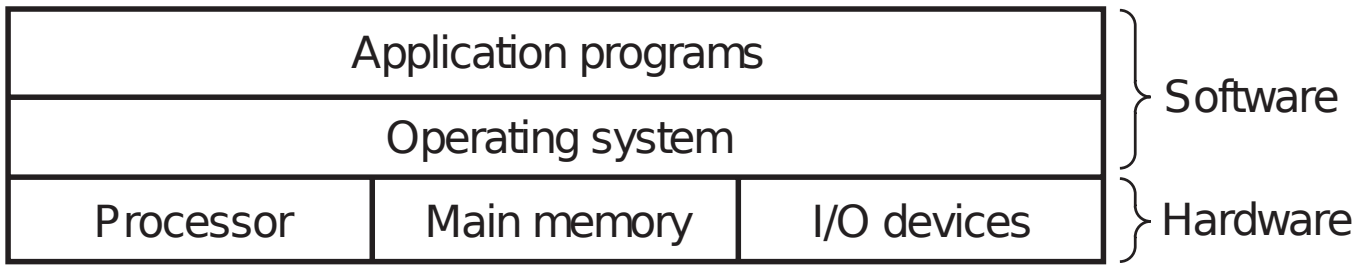
- Executing instructions
- Reading/writing to memory
- Reading/writing files
- Accessing devices

The operating system is here to make it easier to write and run programs, and to manage resources.

# What is an operating system?

Figure by R. Bryant and D. O'Hallaron

The operating system is a layer of software interposed between the application program and the hardware




Two main roles:

- Virtualization
- Resource management

# Virtualization

Transforms the physical resources into virtualized resources:

- It hides the low-level interface of the hardware and provides higher-level abstractions:

# Virtualization

Transforms the physical resources into virtualized resources:

- It hides the low-level interface of the hardware and provides higher-level abstractions:
  - easy to use
  - more general (hides the differences between different hardwares)
  - powerful
  - prevents programs from misusing the hardware

It provides an API (Application Programming Interface) that allows user programs to interact with OS services:

- A set of libraries
- System calls

# Resource management

The OS allows several programs to run on the machine at the same time.

Programs can access resources (CPU, memory, disks, etc.) at the same time. The OS should ensure:

- fairness
- efficiency
- security

# A bit of history

## UNIX

- Created by Ken Thompson from Bell Labs in 1969.
- Made available *for free* to universities in the 70's
- Written in C
- A large set of command to run in a shell (command-line interpreter)
- UNIX systems: Open BSD, Free BSD, Linux, MAC OS X, etc

## Main ideas

- Multi-tasks and Multi-users
- Modular design (set of simple tools)
- A unified file system
  - *Everything is a file*
- Cooperating processes
  - *Inexpensive process spawning and easy Inter-Process Communication*

# A bit of history

## POSIX

- In the 80's, UNIX vendors start adding incompatible features
- IEEE specification of the API of operating systems (1988)
- Portable Operating System Interface (+X for UNIX)

## Linux

- Implementation of a UNIX system from scratch
- By Linus Torvalds
- Released as free software in 1991
- Used by big internet companies (Google, Amazon, Facebook, etc.)

# Agenda

What is an operating system?

Unix File System

The Shell

Processes

Inputs/Outputs

# The file system

On a Unix system, the file system is an important building block:

- The file system is the main means of communication[1]

The file system is a unified tree.

- The root directory is called "/"

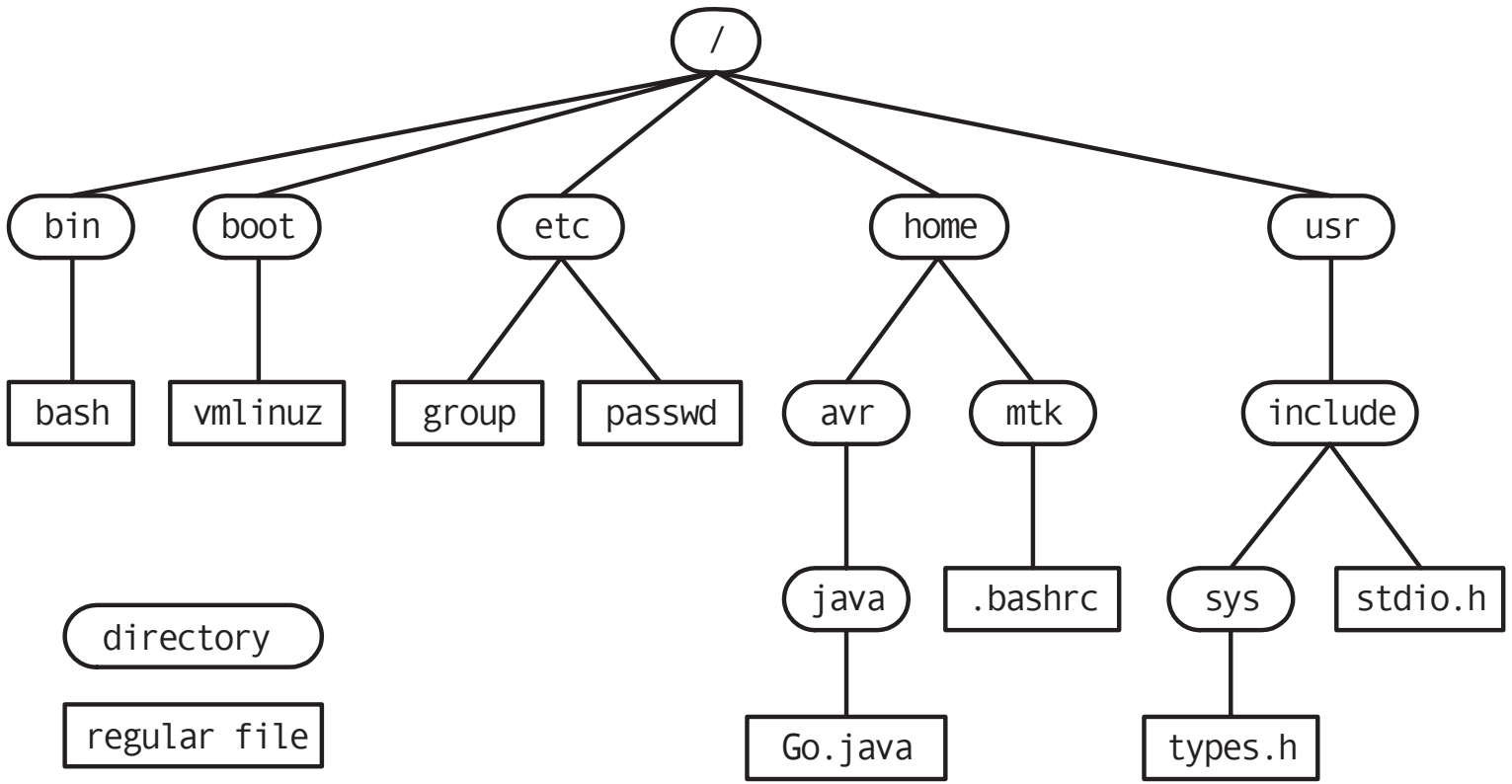
[1] Quoting D. Ritchie and K. Thompson

# File types

It contains different types of objects:

- Regular files
- Directories
- Device files
- Pipes
- Symbolic links
- ...

# The file system tree



# About the file system

## Filename

- Any character can appear in a filename except "/".
- It is not recommended to have white spaces
- Recommended to avoid exotic characters:
  - Portable filename character set: [-._a-zA-Z0-9]

## Default filenames

Every directory contains at least two entries:

- "." (dot): A link to the current directory
- ".." (dot dot): A link to the parent directory

# About the file system

## Pathname

- A series of filenames separated by slashes (/)
  - All but the last of these component filenames identifies a directory
  - The last component of a pathname may identify any type of file, including a directory
- An absolute path starts with "/":
  - Specifies a location with respect to the root directory
  - `/home/avr/java`
- A relative path :
  - Specifies a location with respect to the current directory
  - `../../../usr/include`

# More vocabulary

- Working directory: The directory from which relative paths are interpreted.
  - Every process has a working directory.

- Home directory: The working dir at the time the user logs in.

# Conventional directory layout

man hier

Some conventions exists for file system organization. Here is a non-exhaustive list:

- /: The top level directory referred to as root. Contains all files in the file system.
- /bin: Stands for binaries and contains certain fundamental utilities
- /dev: Stands for devices. Contains files that represent input/output devices
- /etc: Stands for "et cetera". Contains system-wide configuration files and system databases.
- /home: Contains user home directories on Linux

# Conventional directory layout

- /lib: Contains the shared libraries needed by programs in /bin.
- /media: Default mount point for removable devices, such as USB sticks.
- /proc: procfs virtual file system showing information about processes as files
- /tmp: A place for temporary files not expected to survive a reboot
- /usr: Stands for "user file system". It holds executables, libraries, and shared resources that are not system critical.
- /var: Stands for variable. A place for files that may change often (ex: log files).

# Agenda

What is an operating system?

Unix File System

The Shell

Processes

Inputs/Outputs

# Presentation

User interface for OS services:

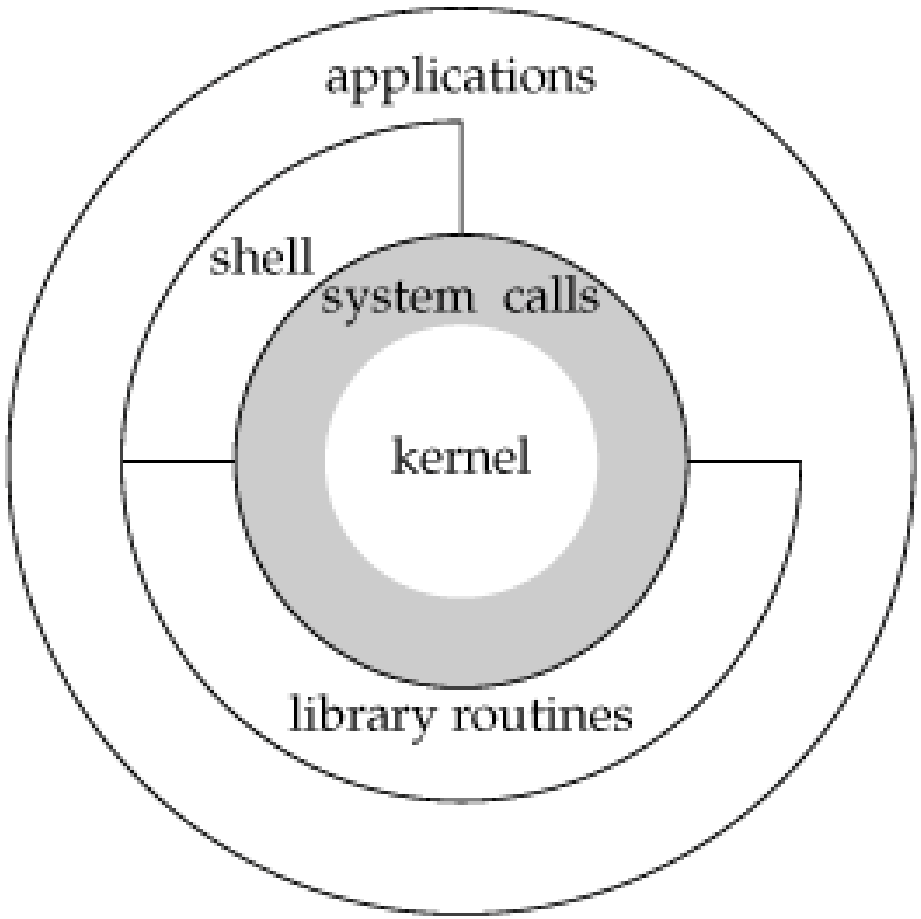
- Command-line interpreter
- Scripting language (to execute a set of commands)

## Bash

- Default shell on most Linux systems
- Extends the Bourne Shell
- Many other shells exist

# A view of a UNIX system

Figure by R. Stevens



# Shell functionalities

- Piping (|)
- Control structures (if, for)
- Variables
- Filename wildcarding (*)
- I/O redirection (<, >)
- ...

# Some useful commands[1]

- Manipulating files and dirs
  - **ls** (listing directory content), **cd** (changing working directory), **mkdir** (create dir), **cp** (copy file/directory), **mv** (move file/directory), **rm** (delete file/directory)
- Working with files
  - **less** (view file content), **cat** (concatenate files), **head/tail** (print first/last part of a file), **file** (check file type)
- Filtering
  - **uniq** (omit duplicate lines), **wc** (count lines/words), **grep** (print lines matching a pattern)
- Documentation
  - **man** (display documentation)

[1] For more information: `http://linuxcommand.org`

# Some examples

```
man uniq
```

# Some examples

`man uniq`

- Displays the documentation of the `uniq` command

`ls *.txt`

# Some examples

`man uniq`

- Displays the documentation of the `uniq` command

`ls *.txt`

- Lists all files in current directory with suffix `.txt`

`grep malloc *.txt`

# Some examples

`man uniq`

- Displays the documentation of the `uniq` command

`ls *.txt`

- Lists all files in current directory with suffix `.txt`

`grep malloc *.txt`

- Prints lines containing the word `malloc` from files with suffix `.txt` in current directory

# Some examples

```
ls | wc -l
```

# Some examples

```
ls | wc -l
```

- Counts the number of entries in the current directory
- Creation of a pipeline: the standard output of the first command is redirected to the standard input of the second command
- 2 processes are created

```
cat file1.txt | uniq
```

# Some examples

`ls | wc -l`

- Counts the number of entries in the current directory
- Creation of a pipeline: the standard output of the first command is redirected to the standard input of the second command
- 2 processes are created

`cat file1.txt | uniq`

- Outputs the content of `file1.txt` while removing adjacent duplicated lines

# About inputs and outputs

see `man stdin`

Each program in a Unix system has 3 streams opened when it is started:

- *stdin*
  - The standard input
  - Identified by the file descriptor 0
  - Associated with the keyboard by default in the shell
- *stdout*
  - The standard output
  - Identified by the file descriptor 1
  - Linked to the screen by default in the shell
- *stderr*
  - The standard error
  - Identified by the file descriptor 2
  - Linked to the screen by default in the shell

# Some examples

```
head file1.txt > file2.txt
```

# Some examples

```
head file1.txt > file2.txt
```

- Writes the 10 first lines of file1.txt in file2.txt (file2.txt is overwritten)
- `stdout` is redirected to `file2.txt`

```
ls -l /usr/bin >> file2.txt
```

# Some examples

```
head file1.txt > file2.txt
```

- Writes the 10 first lines of file1.txt in file2.txt (file2.txt is overwritten)
- `stdout` is redirected to `file2.txt`

```
ls -l /usr/bin >> file2.txt
```

- Appends the result of the `ls` command to file2.txt

```
ls -l /bin/usr 2> error.txt
```

# Some examples

```
head file1.txt > file2.txt
```

- Writes the 10 first lines of file1.txt in file2.txt (file2.txt is overwritten)
- `stdout` is redirected to `file2.txt`

```
ls -l /usr/bin >> file2.txt
```

- Appends the result of the `ls` command to file2.txt

```
ls -l /bin/usr 2> error.txt
```

- Writes the error messages to file `error.txt`
- `stderr` is redirected to `error.txt`

```
grep error /var/log/* >output.txt 2>&1
```

# Some examples

```
head file1.txt > file2.txt
```

- Writes the 10 first lines of file1.txt in file2.txt (file2.txt is overwritten)
- `stdout` is redirected to `file2.txt`

```
ls -l /usr/bin >> file2.txt
```

- Appends the result of the `ls` command to file2.txt

```
ls -l /bin/usr 2> error.txt
```

- Writes the error messages to file `error.txt`
- `stderr` is redirected to `error.txt`

```
grep error /var/log/* >output.txt 2>&1
```

- Redirects `stdout` to file `output.txt` and `stderr` to `stdout`

# More examples

```
grep error /var/log/* 2> /dev/null
```

# More examples

```
grep error /var/log/* 2> /dev/null
```

- Suppresses error messages
- `/dev/null` is a special file. It is a system device that accepts input and does nothing with it.

```
ls /bin /usr/bin | sort | uniq > /tmp/exec_list.txt
```

# More examples

```
grep error /var/log/* 2> /dev/null
```

- Suppresses error messages
- `/dev/null` is a special file. It is a system device that accepts input and does nothing with it.

```
ls /bin /usr/bin | sort | uniq > /tmp/exec_list.txt
```

- Lists content of the two directories, sorts the result, removes the duplicated entries and stores the results in a temporary file.

# Agenda

What is an operating system?

Unix File System

The Shell

Processes

Inputs/Outputs

# About processes

A process is an instance of a running program

- The program itself is just a file
  - The set of instructions to execute
- The process is an operating system abstraction of a running program

Commands to know which programs are running on the system:

- `ps`: Lists the active processes
- `top`: Dynamic list of processes with resource usage

# About processes

The operating system provides to an instance of a program the illusion that it is the only one running on the system.

- It appears to have exclusive access to the processor, the memory, and the I/O devices.
- The processor appears to execute the instructions in the program, one after the other, without interruption.

# About processes

The operating system provides to an instance of a program the illusion that it is the only one running on the system.

- It appears to have exclusive access to the processor, the memory, and the I/O devices.
- The processor appears to execute the instructions in the program, one after the other, without interruption.

- However, in practice, there might be many programs running concurrently on the system.
  - Web browser, email client, text editor, compiler, system services (init, syslog, graphical interface, ...)
  - Multiple instances of the same program might run concurrently (`emacs file1.txt` + `emacs file2.txt`)

# About memory

## Physical memory

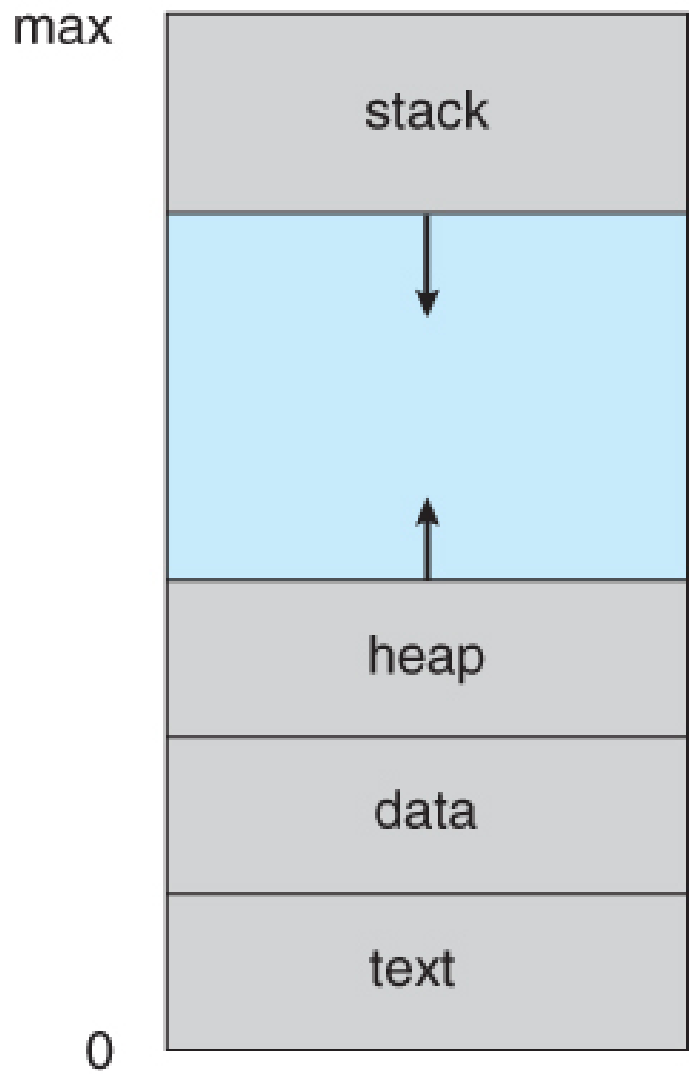
- Physical memory is an array of bytes
  - Area to read and write data
- **A process does not access physical memory directly**

## Virtual memory

- Each process is provided with its own virtual address space
- The OS translates virtual addresses into physical addresses
- A memory reference within one running program does not affect the address space of other processes (or the OS itself)

# Process virtual address space

Figure by Silberschatz et al



# Process virtual address space

- text section: comprises the compiled program code, read from the executable file.
- data section: stores global and static variables, allocated and initialized prior to executing main
- heap: Expends and contracts as a result of dynamic memory allocation
- stack: Space on the stack is reserved for local variables when they are declared, and the space is freed up when the variables go out of scope

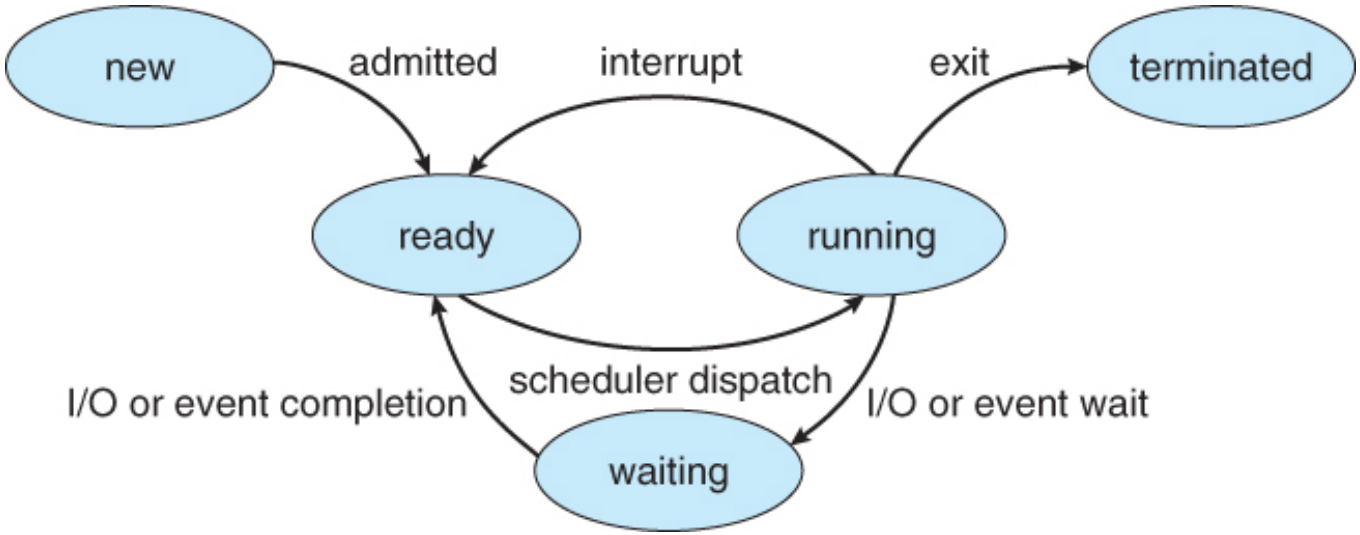
# Process resources

A set of resources is associated with each process:

- A unique process id
- A process state
- A program counter
  - It point to the next instruction to be executed by the process
- CPU registers
- Credentials
- Information about virtual memory
- Information about open files
- ...

# Process state

Figure by Silberschatz et al




- new: The process is in the stage of being created
- running: currently executing
- ready: can run, but kernel has chosen a different process to run
- waiting: The process cannot run, because it is waiting for some resource to become available or for some event to occur. (ex: keyboard input)
- terminated: The process has terminated

# Process control

3 sets of functions:

- fork: process creation
- exec: load new executable file
- wait: wait for process termination

# Creating a new process

- **`pid_t fork(void)`**
  - Creates a new process (child process) that is identical to the calling process (parent process)
  - Returns 0 to the child process
  - Returns the child's pid to the parent process

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

  - **`fork`** is interesting (and often confusing) because it is called once but returns twice

# Understanding fork

***Process n***

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = m

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

***Child Process m***

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = 0

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

`hello from parent` ***Which one is first?*** `hello from child`

# Fork: additional details

- Parent and child both run the same code
  - Distinguish parent from child by return value from **`fork`**
- Start with the same state, but each has private copy
  - Memory address space
  - Environment variables
  - List of currently open files
  - Signal handlers, signal mask and list of pending signals
- Scheduling non-determinism
  - Who runs first after a fork? Parent? Child? (Both?)
  - No order imposed/specified

# Ending a process

- **`void exit(int status)`**
  - Exits a process (current process ceases to exist)
    - By convention, status of 0 is success (non-zero means error)
  - **`atexit()`** allows programmer to register function to be executed upon exit

```
void cleanup(void) {
   printf("cleaning up\n");
   ... // clean things up
   printf("done\n");
}

int main() {
   atexit(cleanup);
   ... // do useful things
   exit(0);
}
```

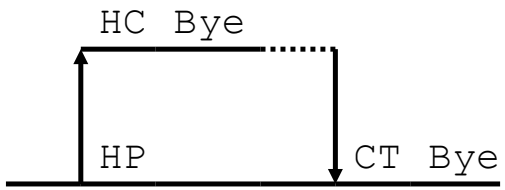
# Synchronizing with child processes

- **`int wait(int *child_status)`**
  - suspends current process until one of its children terminates
  - return value is the **`pid`** of the child process that terminated
  - if **`child_status != NULL`**, then the variable it points to will be set to a status indicating why the child process terminated

# Synchronizing with child processes
# `wait`: example #1

```
int main() {
   int child_status;

   if (fork() == 0) {
      printf("HC: hello from child\n");
   }
   else {
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
   exit();
}
```



# Synchronizing with child processes
# `wait`: details and example #2

- If multiple children completed, will take in arbitrary order
- Can use macros **`WIFEXITED`** and **`WEXITSTATUS`** to get information about exit status (see **`man 2 wait`** for details)

```
int main()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

# Synchronizing with child processes
# Waiting for a specific process

- **`waitpid(pid, &status, options)`**
  - suspends current process until specific process terminates
  - various options – see man page for details

```
int main()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++) {
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    }
    for (i = 0; i < N; i++) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status)) /* Child terminated due to call to exit */
            printf("Child %d terminated with exit status %d\n",
                    wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

# Zombies

- Idea
  - When a process terminates, it still consumes system resources
    - Various tables maintained by OS
  - Called a “zombie”
    - Living corpse, half alive and half dead
- Reaping
  - Performed by parent on terminated child via **`wait`** or **`waitpid`**
  - Parent is given exit status information
  - Kernel discards process
- What if parent doesn’t reap?
  - If any parent terminates without reaping a child, then child will be reaped by a special OS process (often called **`init`**)
  - So, we only need explicit reaping in long-running processes
    - e.g., shells and servers

# Loading and running programs

- **`int execve(char *filename, char *argv[], char *envp);`**

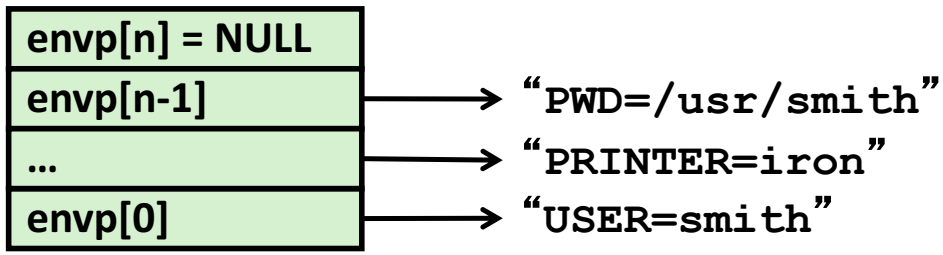
- Completly reconfigures the image of the calling process

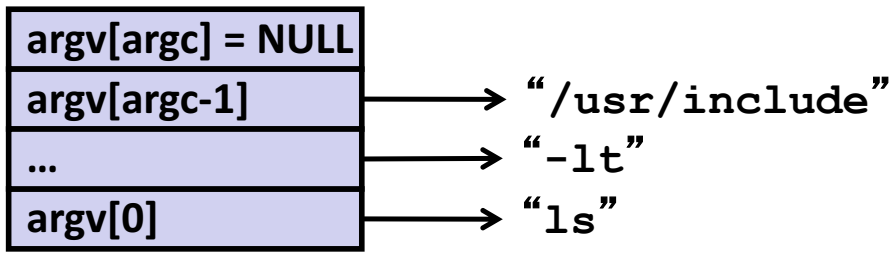
- Loads and runs
  - Executable **`filename`**
  - With argument list **`argv`** (NULL terminated)
  - By convention, **`argv0`** should be the name of the executable file
  - And environment variable list **`envp`**
- **Does not return!** (unless error)

- Overwrites process memory address space, keeps pid

- Environment variables
  - “name=value” strings

# `Execve` arguments: example



# `execl` and `exec` family

- `int execl(char *path, char *arg0, char *arg1, ..., 0)`

- Loads and runs executable at path with args **`arg0`**, **`arg1`**, ...)
  - **`path`** is the complete path of an executable object file
  - By convention, **`arg0`** is the executable object file
  - "Real" arguments to the program start with **`arg1`**, etc.
  - **List of args is terminated by a `(char*)0` argument**
  - Environment (implicitly) taken from **`char **environ`** (an automatically defined global variable), which points to an array of "name=value" strings

# `execl` and `exec` family (continued)

- Family of functions includes **`execv, execve, execvp, execl, execle, execlp`**
- **These functions only differ in terms of interface; their purpose is the same (loading and running a new process, in the context of the current process)**
- Usually, **`execve`** is the only system call
  - The other functions are implemented as library wrappers

- **Semantics of the suffixes**
  - **`v`** (vector): pass arguments as (**`NULL`**-terminated) array of pointers
  - **`l`** (list): pass arguments as (**`NULL`**-terminated) list of pointers (i.e., the function has a variable number of arguments)
  - **`p`** (path): search executable file name in the **`PATH`** (otherwise, full path must be provided)
  - **`e`** (environment): explicitly provide new environment (otherwise, use the current environment pointed by **`environ`**)

# `execl`: example

```
int main() {
   int res;
   if (fork() == 0) {
      res = execl("/usr/bin/cp", "cp", "foo", "bar", 0);
      if (res < 0) {
          printf("error: execl failed\n");
          exit(-1);
      }
   }
   wait(&res);
   if (WIFEXITED(res) && (WEXITSTATUS(res) == 0)) {
      printf("copy completed successfully\n");
   } else {
      printf("copy failed\n");
   }
   exit(0);
}
```

# Agenda

What is an operating system?

Unix File System

The Shell

Processes

Inputs/Outputs

# Note

From this point on, most of the slides are taken from "Lecture 0: Unix I/O" of the OS course.

- Full set of slides available on the web site:
  `https://m1-mosig-os.gitlab.io/`

# Unix files

- A Unix *file* is a sequence of *m* bytes:
  - $B_0, B_1, \ldots, B_k, \ldots, B_{m-1}$

- All I/O devices are represented as files:
  - **`/dev/sda2`** (**`/usr`** disk partition)
  - **`/dev/tty2`** (terminal)

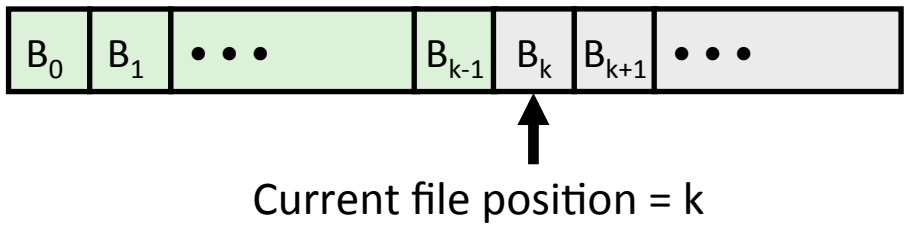
- Even the kernel sometimes represented as a file:
  - **`/dev/kmem`** (kernel memory image)
  - **`/proc`** (kernel data structures)

# Unix file types

- Regular file
  - File containing user/app data (binary, text, whatever)
  - OS does not know anything about the format
    - Other than “sequence of bytes”, akin to main memory
- Directory file
  - A file that contains the names and locations of other files
- Character special and block special files
  - Terminals (character special) and disks (block special)
- FIFO (named pipe)
  - A file type used for inter-process communication (details later)
- Socket
  - A file type used for network communication between processes

# Unix I/O

- Key Features
  - Elegant mapping of files to devices allows kernel to export simple interface called Unix I/O
  - Important idea: All input and output is handled in a consistent and uniform way
- Basic Unix I/O operations (system calls):
  - Opening and closing files
    - **`open()`** and **`close()`**
  - Reading and writing a file
    - **`read()`** and **`write()`**
  - Changing the ***current file position*** (seek)
    - indicates next offset into file to read or write
    - **`lseek()`**



# Opening files

- Opening a file informs the kernel that you are getting ready to access that file

```
int fd;   /* file descriptor */

if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {
   perror("open");
   exit(1);
}
```

- Returns a small identifying integer *file descriptor*
  - **`fd == -1`** indicates that an error occurred
- Each process created by a Unix shell begins life with three open files associated with a terminal:
  - 0: standard input
  - 1: standard output
  - 2: standard error

# Closing files

- Closing a file informs the kernel that you are finished accessing that file

```
int fd;     /* file descriptor */
int retval; /* return value */

if ((retval = close(fd)) < 0) {
   perror("close");
   exit(1);
}
```

- Closing an already closed file is a recipe for disaster in threaded programs (more details on this later)

- Moral: Always check return codes, even for seemingly benign functions such as **`close()`**

# Reading files

- Reading a file copies bytes from the current file position to memory, and then updates file position

```
char buf[512];
int fd;       /* file descriptor */
int nbytes;   /* number of bytes read */

/* Open file fd ...  */
/* Then read up to 512 bytes from file fd */
if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {
   perror("read");
   exit(1);
}
```

- Returns number of bytes read from file **`fd`** into **`buf`**
  - Return type **`ssize_t`** is signed integer (unlike **`size_t`**)
  - **`nbytes < 0`** indicates that an error occurred
  - ***Short counts*** (**`nbytes < sizeof(buf)`** ) are possible and are not errors!

# Writing files

- Writing a file copies bytes from memory to the current file position, and then updates current file position

```
char buf[512];
int fd;       /* file descriptor */
int nbytes;   /* number of bytes read */

/* Open the file fd ... */
/* Then write up to 512 bytes from buf to file fd */
if ((nbytes = write(fd, buf, sizeof(buf)) < 0) {
   perror("write");
   exit(1);
}
```

- Returns number of bytes written from **`buf`** to file **`fd`**
  - **`nbytes < 0`** indicates that an error occurred
  - As with reads, short counts are possible and are not errors!

# Simple Unix I/O example

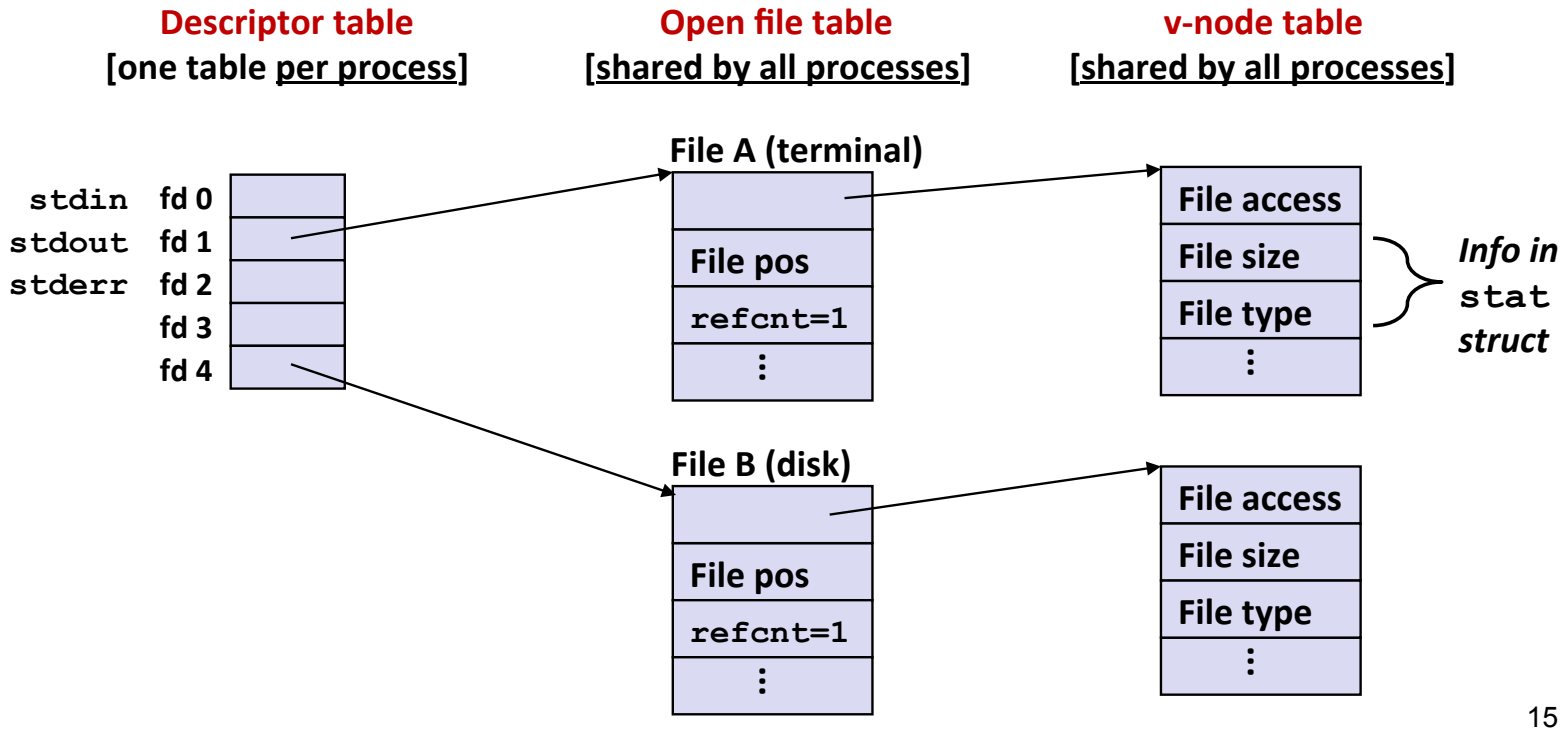
- Copying standard input to standard output, one byte at a time

```
int main(void)
{
    char c;
    int len;

    while ((len = read(0 /*stdin*/, &c, 1)) == 1) {
        if (write(1 /*stdout*/, &c, 1) != 1) {
            exit(20);
        }
    }
    if (len < 0) {
        printf ("read from stdin failed");
        exit (10);
    }
    exit(0);
}
```

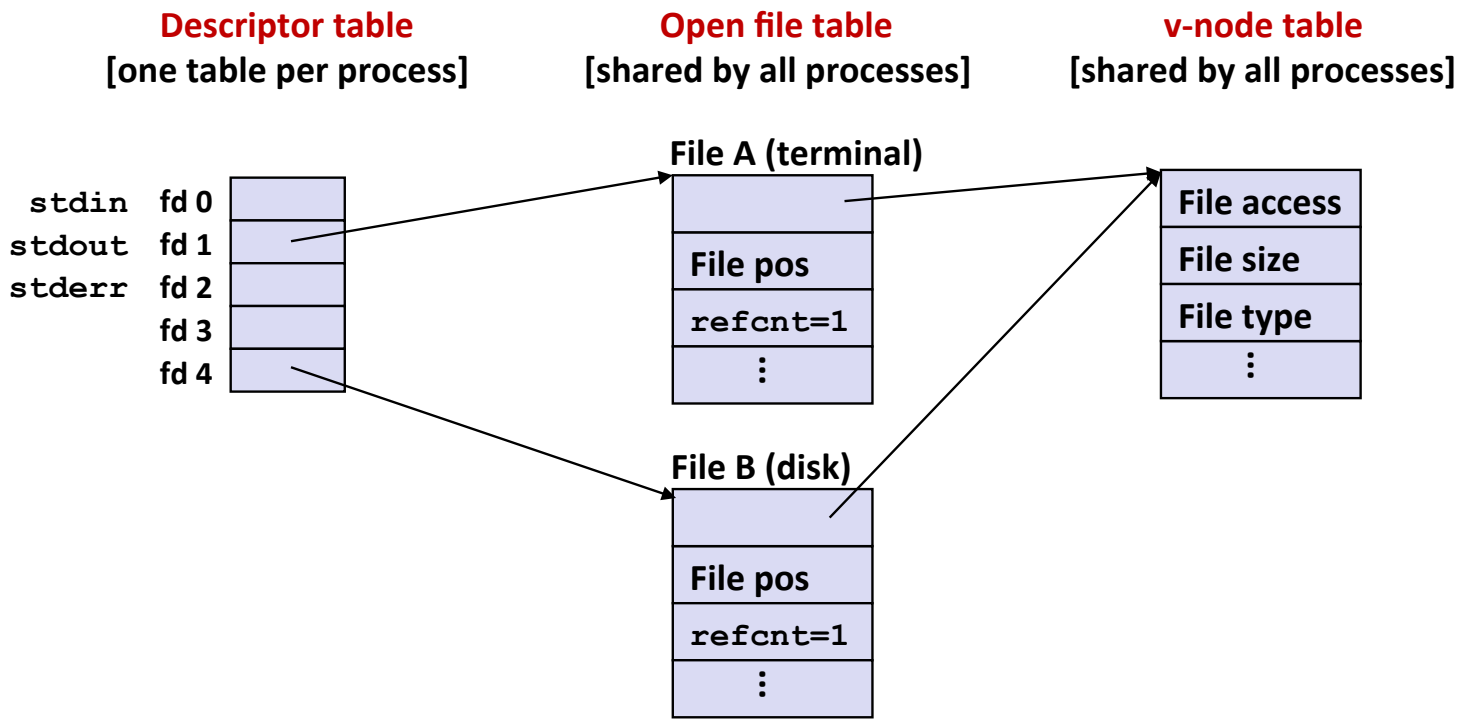
# How a Unix kernel represents open files

- Two descriptors referencing two distinct open disk files.
- Descriptor 1 (stdout) points to terminal, and descriptor 4 points to open disk file



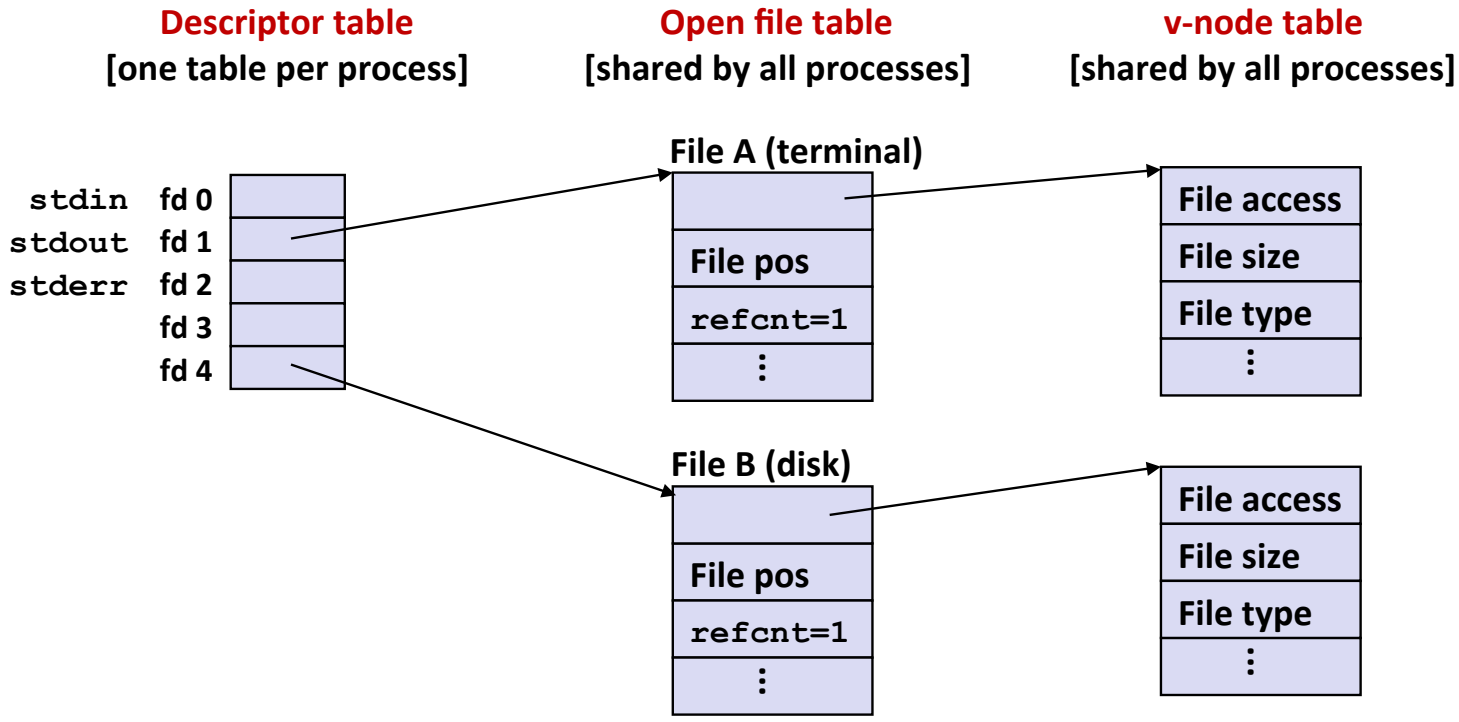
# File sharing

- 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



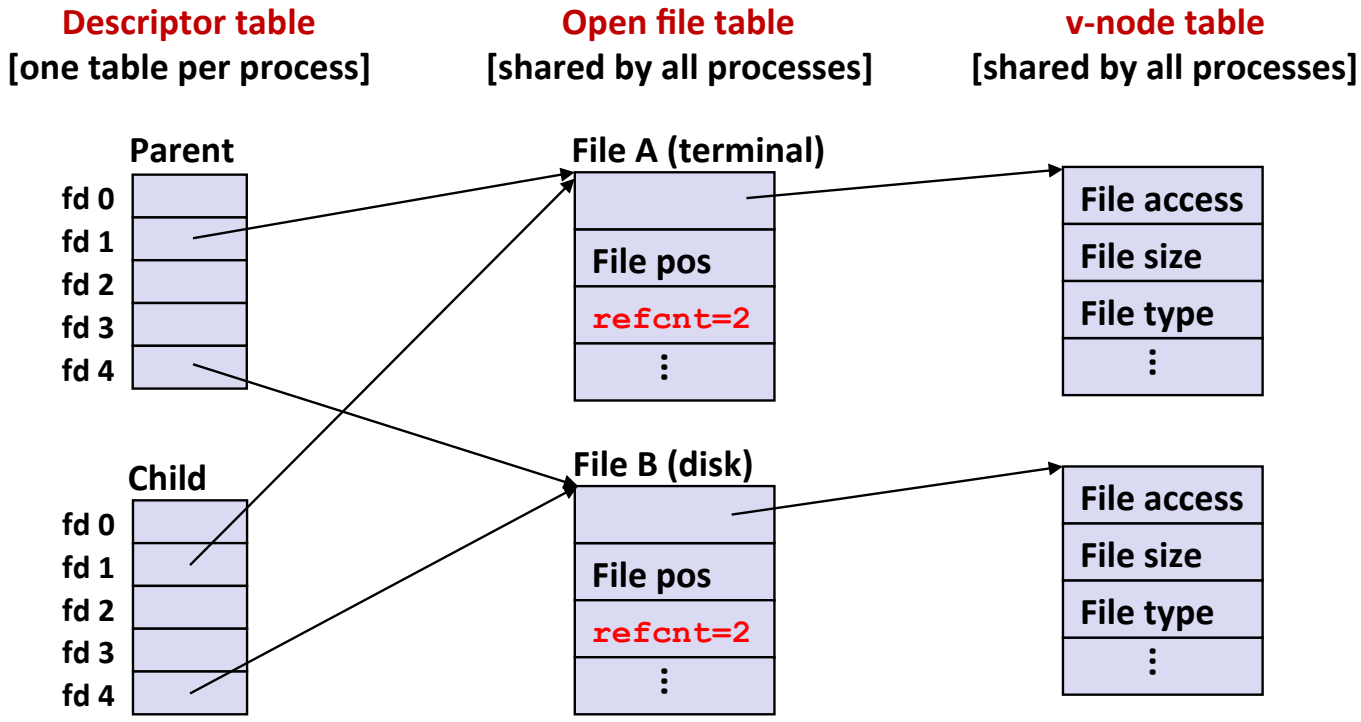
# How processes share files
# What happens upon fork

- A child process inherits its parent's open files
  - Note: situation unchanged by **`exec`** functions
- *Before* **`fork`** call:



# How processes share files
# What happens upon fork

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



# I/O redirection

- Question: How does a shell implement I/O redirection?

```
ls > foo.txt
```

- Answer: By calling the **`dup2(oldfd, newfd)`** function
  - Copies (per-process) descriptor table entry **`oldfd`** to entry **`newfd`**

**Descriptor table**
***before*** **`dup2(4,1)`**

| | |
|---|---|
| fd 0 | |
| fd 1 | a |
| fd 2 | |
| fd 3 | |
| fd 4 | b |

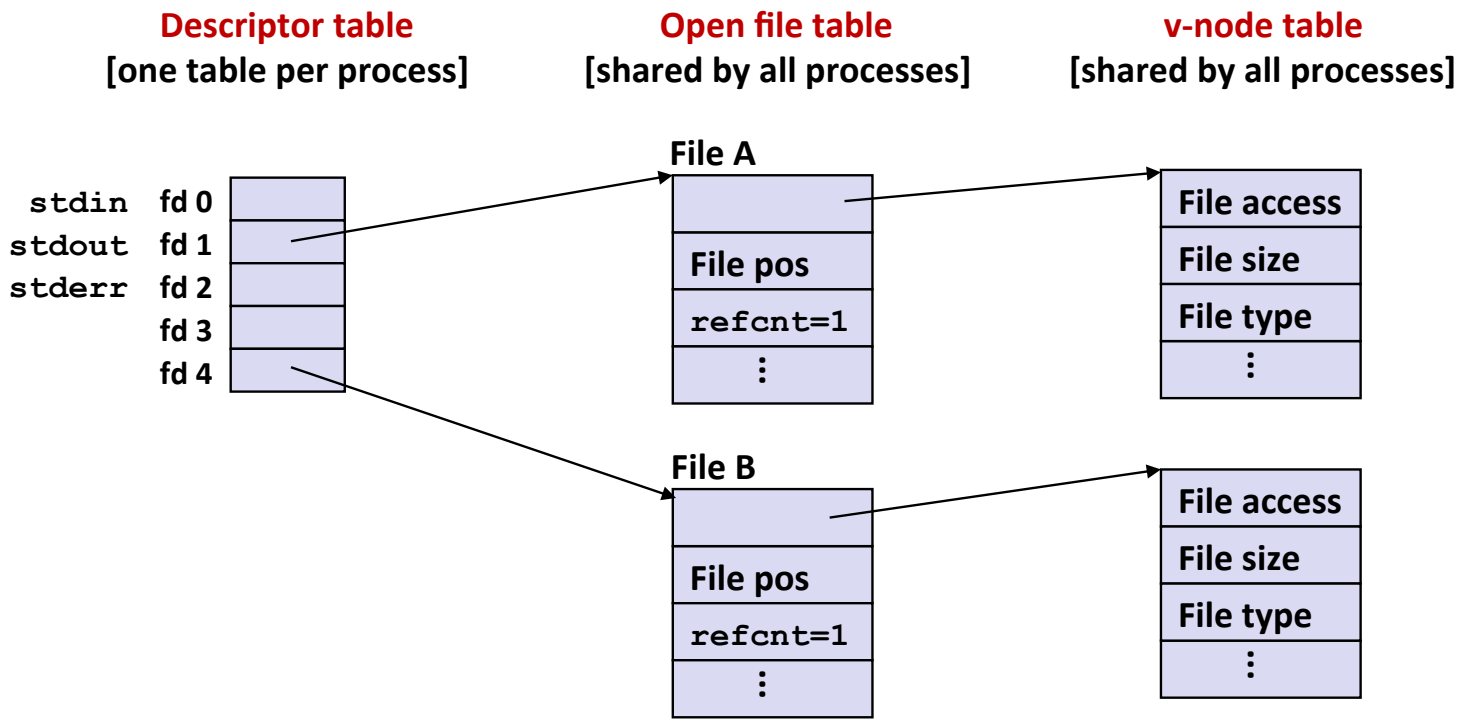


**Descriptor table**
***after*** **`dup2(4,1)`**

| | |
|---|---|
| fd 0 | |
| fd 1 | b |
| fd 2 | |
| fd 3 | |
| fd 4 | b |

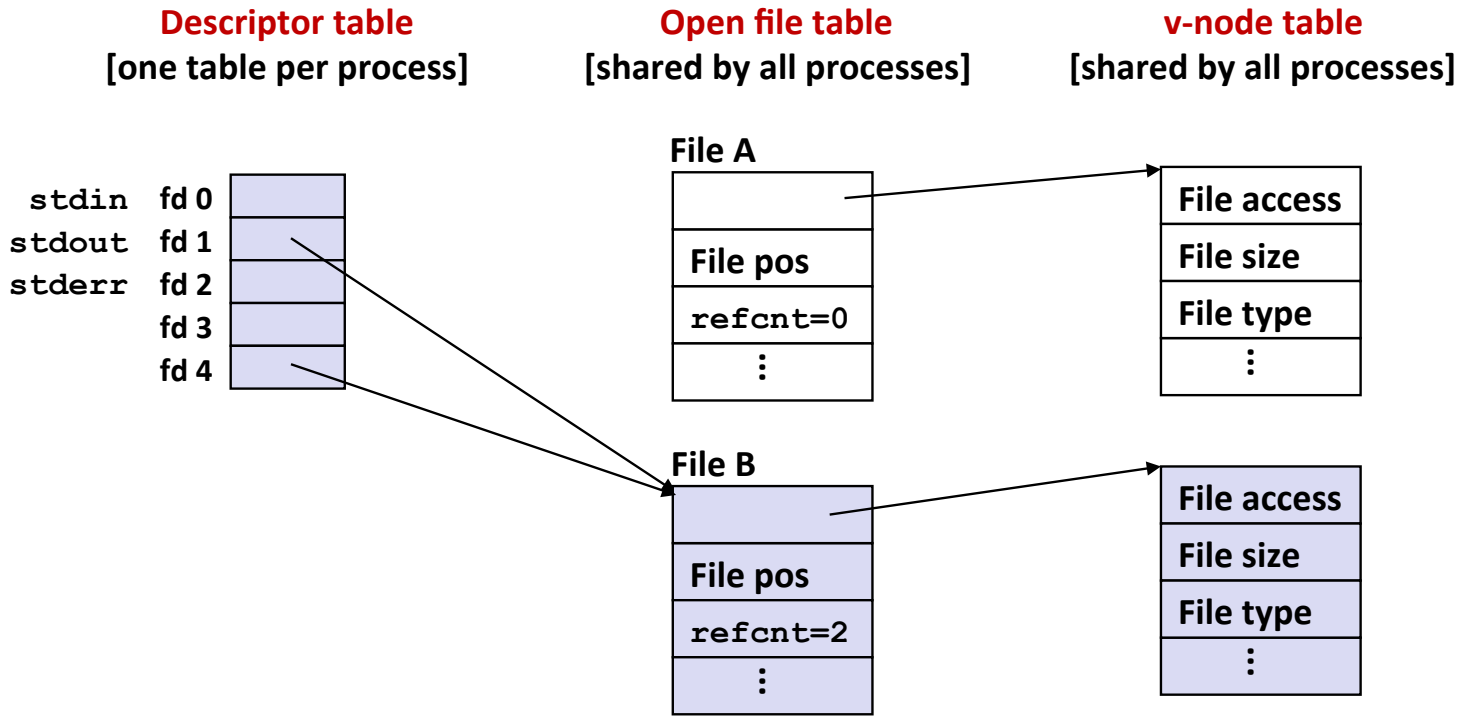
# I/O redirection example

- Step #1: open file to which stdout should be redirected
  - Happens in child executing shell code, before calling **`exec`**



# I/O redirection example (continued)

- Step #2: call **`dup2(4,1)`**
  - causes fd=1 (stdout) to refer to disk file pointed at by fd=4
  - (then fd=4 can be closed)



# Standard I/O functions

- The C standard library (**`libc`**) contains a collection of higher-level *standard I/O* functions

- Examples:
  - Opening and closing files (**`fopen`** and **`fclose`**)
  - Reading and writing bytes (**`fread`** and **`fwrite`**)
  - Reading and writing text lines (**`fgets`** and **`fputs`**)
  - Formatted reading and writing (**`fscanf`** and **`fprintf`**)

# Standard I/O streams

- Standard I/O models open files as *streams*
  - Abstraction for a file descriptor and a buffer in user memory.
- C programs begin life with three open streams (defined in **`stdio.h`**)
  - **`stdin`** (standard input)
  - **`stdout`** (standard output)
  - **`stderr`** (standard error)

```
#include <stdio.h>
extern FILE *stdin;  /* standard input  (descriptor 0) */
extern FILE *stdout; /* standard output (descriptor 1) */
extern FILE *stderr; /* standard error  (descriptor 2) */

int main() {
    fprintf(stdout, "Hello, world\n");
}
```

# Unix pipes

- Pipes are a mechanism for inter-process communication (IPC)
- A pipe is essentially a (unidirectional) buffer that can be used for data exchange between a producer process and a consumer process
- Available at two levels: command line interface and programmatic interface
- Command line interface (shell)
  - Example : **`cat *.c | grep var`**
    - Creates two processes: P1 running **`cat *.c`** and P2 running **`grep var`**
    - Connects (redirects) P1's standard output to the pipe's input and the pipe's output to P2's standard input

# Unix pipes
# Programmatic interface

- User programs (not just shells) can create and interact with pipes through system calls
- A pipe is seen as a special kind of file
- The only way to share a pipe between processes is through inheritance of open files
- Typical usages:
  - Parent creates pipe then creates child then communicates with child through pipe (see following example)
  - Parent creates pipe, then create child1 and child2, then child1 and child2 communicate through pipe

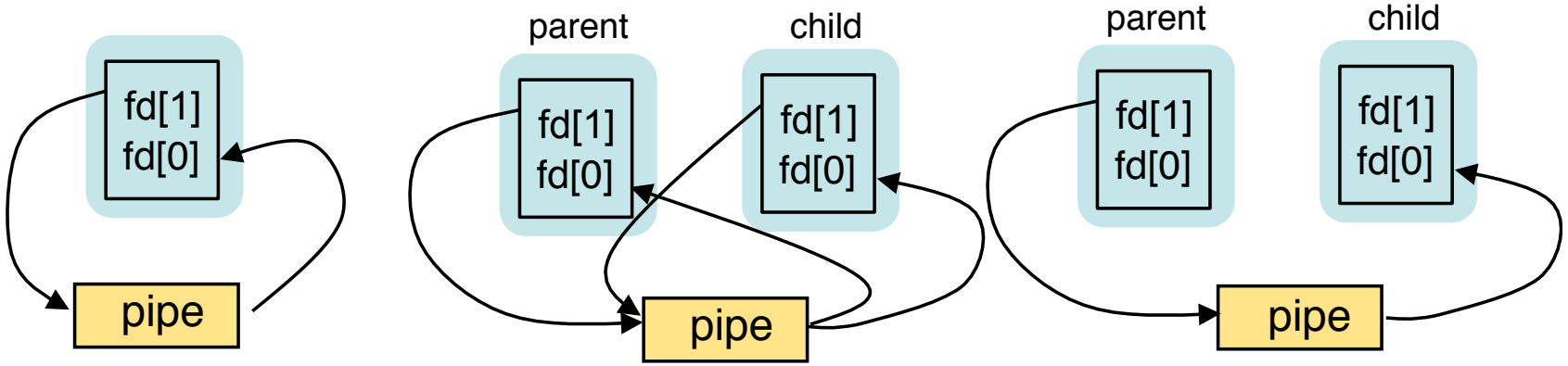
# Unix pipes
# Programmatic interface (continued)

**Pipe creation**: `int pipe(int filedes[2])`

```
int fd[2]; pipe(fd);
```

If the call succeeds, a pipe is created and the **`fd`** array is updated with the file descriptors of the **pipe's output** (in **`fd[0]`**) and the **pipe's input** (in **`fd[1]`**)
If the call fails, -1 is returned.

The pipe can then be transmitted through inheritance and used for communication. Each process will typically use only one side of the pipe and should close the other side.




after **`pipe(fd)`**

after **`fork()`**
(copied descriptors)

after closing unused descriptors

# Unix pipe example

```
#include ...
#define BUFSIZE 10
int main(void) {
  char bufin[BUFSIZE] = "empty";
  char bufout[BUFSIZE] = "hello";
  int bytesin, bytesout;   pid_t childpid;
  int fd[2];

  pipe(fd);
  bytesin = strlen(bufin);
  childpid = Fork();
  if (childpid  != 0) {           /* parent */
    close(fd[0]);
    bytesout = write(fd[1], bufout, strlen(bufout)+1);
    printf("[%d]: wrote %d bytes\n", getpid(), bytesout);
  } else {                        /* child */
    close(fd[1]);
    bytesin = read(fd[0], bufin, BUFSIZE);
    printf("[%d]: read %d bytes, my bufin is {%s} \n »,
           getpid(), bytesin, bufin);
  }
  exit(0);
}
```

```
<unix>./parentwritepipe
[29196]:wrote 6 bytes
[29197]: read 6 bytes, my bufin is {hello}
<unix>
```

# Unix pipes
# Additional details

- Pipes are unidirectional (i.e., one-way communication), with first-in-first-out semantics
  - If two-way communication is needed, use a pair of pipes

- Pipes are not persistent

- **Automatic producer-consumer synchronization**
  - A reader will block if the pipe is empty but has at least one writer (i.e., the pipe input is still open)
  - If the pipe is empty and has no remaining writer, **`read`** will return 0
  - A writer will block if pipe is full but has at least one reader (i.e., the pipe output is still open)
  - A **`write`** to a pipe with a closed output will trigger an error
  - So, for correct operation, it is important for each process to close the unused side(s) of a given pipe

# Unix pipes
# Additional details (continued)

- A call to write on a pipe with less than **`PIPE_BUF`** bytes (4096 bytes on Linux) is an atomic operation
- A call to write on a pipe with more than **`PIPE_BUF`** bytes is not necessarily atomic (i.e., the written data may get interleaved with the data of other writes)
- **`lseek`** does not work on pipes
- See **`man 7 pipe`** for details