CSC209: Software Tools and Systems Programming

Week 8: Processes, Forking, and File Descriptors [1]

Kianoosh Abbasi

[1] Slides are mostly taken from Andi Bergen's in summer 2021.

Systems Programming

- Now we will shift to the *systems-level* aspect of the course, which requires awareness of the hardware and operating system being used
  - Unlike Java or Python, which act as a "translation layer" that make all platforms appear similar
- When working on tutorials/assignments, think about what we're learning about the underlying system.

System Calls vs. Library Calls

- System calls are the interface by which programs request services from the operating system kernel
- Standard C library functions (e.g., string library functions) are **not** system calls
  - Some serve as "wrappers" around system calls (e.g., `fopen()` calls `open()`)

System Calls and Portability

- Usage of C standard library functions is *portable*
  - But need to recompile on different platforms
- Usage of system calls is **not** portable, unless you
  - Use (e.g., POSIX-compliant) system calls supported by multiple operating systems (see `CONFORMING TO` heading in system call man pages)
  - Write separate implementations of OS-dependent code, compile multiple platform-specific executables (using C preprocessor macros)

Error-Checking

- All system calls and some library functions use `errno` to return error values
  - See `man errno` for complete list of error names defined in `errno.h`

Q: How many systems programmers does it take to change a light bulb?

A: Just one, but they will keep changing it until it returns 0.

Simplistic use of system calls is not suitable for proper error handling. Example below demonstrates proper usage of `read()` to read `len` bytes.

```
ssize_t ret;
while (len != 0 && (ret = read(fd, buf, len)) != 0) {
    if (ret == -1) {
        if (errno == EINTR)
            continue;
        perror("read");
        break;
    }
    len -= ret;
    buf += ret;
}
```

Processes

- A *process* is an instance of an executing *program*
- Executing multiple instances of the same program launches multiple processes
  - e.g., run multiple instances of Notepad
- A single instance of a program may launch multiple processes
  - Firefox/Chrome run one-process-per-tab
  - Assignment 2

Process Memory

Each process has its own memory space, including its own stack and heap.

A process cannot access the variables/memory of another process.

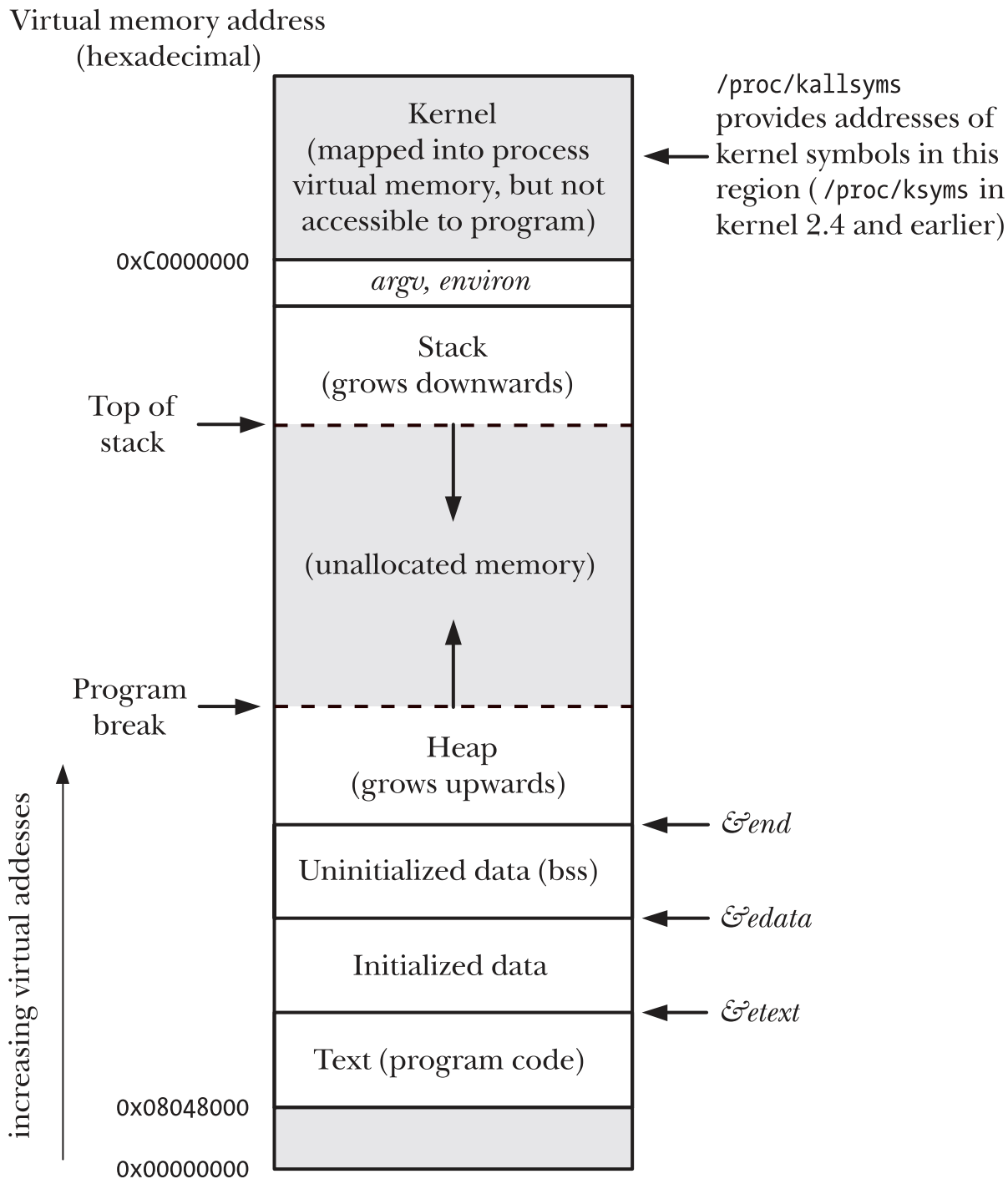


Figure 6-1: Typical memory layout of a process on Linux/x86-32

Process Creation

- In UNIX-like systems, processes are created with the `fork()` system call (next week)
- The process that calls `fork()` is the *parent* of the newly-created *child* process
- Try `pstree` from a Bash shell to print the tree of currently-running processes

Process Identifiers

- Each process has a PID (process identifier)
- The first process created when the system boots up is the `init` process, with PID 1
  - On Ubuntu, the `init` process is `systemd`
  - On macOS, the `init` process is `launchd`

Program Exit Status vs. Process Exit Status

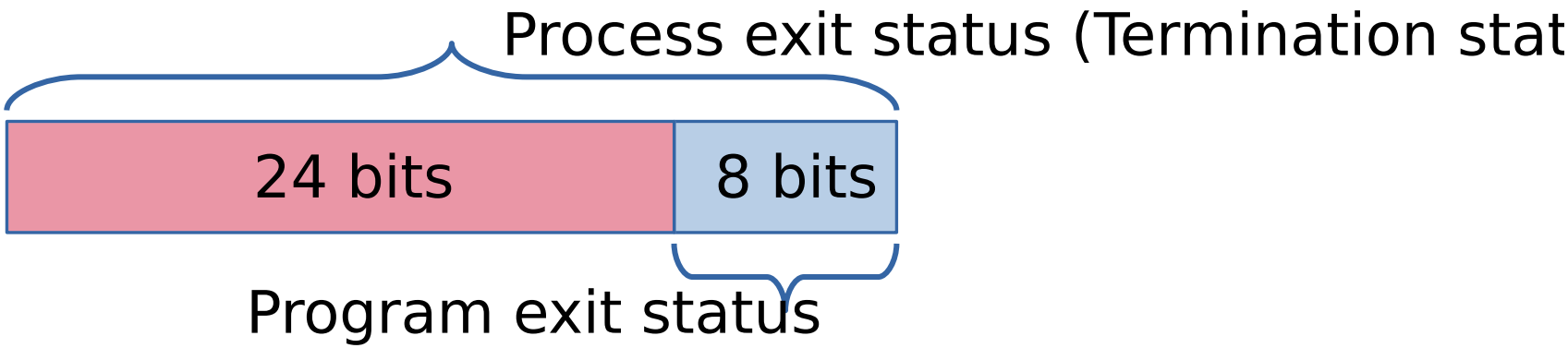
Every program reports an *exit status* upon completion/termination. This is done via `exit()`.

```
void exit (int status)
```

Here, `status` is the *program's exit status*, which becomes **part** of the *process' exit status*.

Source: The GNU C Library

Obtaining the Exit Status



A process exit status is saved to be reported back to the parent process via `wait` *or* `waitpid`*. If the program exited, this status includes as its low-order 8 bits the program exit status.*

Source: The GNU C Library

Exit vs. Return

Inside `main()`, `return` and `exit()` are nearly equivalent (save some edge cases):

1. The return value of `main()` is the *program exit status* passed to `exit()`.
2. `exit()` performs some cleanup (e.g., flush `stdio` streams) and calls `_exit()`.
3. `_exit()` sets the *process exit status*, or *termination status*, and terminates the process.

Outside of `main()`, use `exit()` to terminate the process.

Try:

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    printf("Hi");
    exit(0); // equivalent to "return 0;"
}
```

vs.

```c
#include <stdio.h>
#include <unistd.h>
int main() {
    printf("Hi");
    _exit(0);
}
```

What's the difference?

Exit Status Conventions

Return 0 on success, any other value on error.

But if you're a real GNU/Linux geek...

> *A general convention reserves status values 128 and up for special purposes. In particular, the value 128 is used to indicate failure to execute another program in a subprocess.*

Question

How do we obtain the program exit status from the process exit status?

Answer

Use macros defined in `wait.h` (see `man 2 wait`), e.g.,

- `WIFEXITED(status)` to see if process terminated normally or abnormally
- `WEXITSTATUS(status)` to obtain program exit status

The exec Functions

The exec functions *load a new program* into the current process image. The process retains its original PID.

The functions differ mainly in how they are called, e.g.,

```
char* args[] = {"ls", NULL};
execve("/bin/ls", args, NULL);
```

vs.

```
execle("/bin/ls", "ls", NULL, NULL);
```

Table 27-1: Summary of differences between the *exec()* functions

Function	**Specification of program file (–, *p*)**	**Specification of arguments (*v*, *l*)**	**Source of environment (*e*, –)**
execve()	pathname	array	*envp* argument
execle()	pathname	list	*envp* argument
execlp()	filename + `PATH`	list	caller's *environ*
execvp()	filename + `PATH`	array	caller's *environ*
execv()	pathname	array	caller's *environ*
execl()	pathname	list	caller's *environ*

Also: `execve()` is a system call; the others are C standard library functions

Exec and the Shell

A shell can use `fork` and `exec` to execute other programs:

1. Shell process `p` waits for keyboard input.
2. You type `ls`.
3. Shell forks child process `c`.
4. Process `c` uses an `exec` function to run `ls`.
5. Process `p` calls `wait` to wait for `c` to terminate, and then prints new prompt.

Exec and File Descriptors

When a program calls `exec`, the new program still retains the file descriptors of the original process.

But the `FILE *` variables are gone, upon replacing the process image with the new program.

So the new program must either:

1. Perform low-level I/O using the `read()` or `write()` system calls (you will do this with *pipes* on Assignment 2); or
2. Use `fdopen()` to associate a new buffered file stream with an existing open file descriptor.

Low-Level I/O

When we want to do low-level I/O, to bypass the buffering and abstractions provided by the C standard library, we must use *system calls*, namely:

- `open()`
- `close()`
- `read()`
- `write()`

Many different flags, errors, corner cases, etc. to consider: See `man 2 XXX`, where `XXX` is the name of the system call.

Repeated for emphasis: Proper usage of low-level I/O often requires looping and handling error conditions.

```
ssize_t ret;
while (len != 0 && (ret = read(fd, buf, len)) != 0) {
    if (ret == -1) {
        if (errno == EINTR)
            continue;
        perror("read");
        break;
    }
    len -= ret;
    buf += ret;
}
```

Low-Level I/O and File Descriptors

- Low-level I/O is done using *file descriptors*, which are integer values that serve as indices for open files
- Each process has its own *file descriptor table*
  - File descriptor `N` in *process A* can refer to a different file than file descriptor `N` in *process B*

File Descriptors vs. File Descriptions

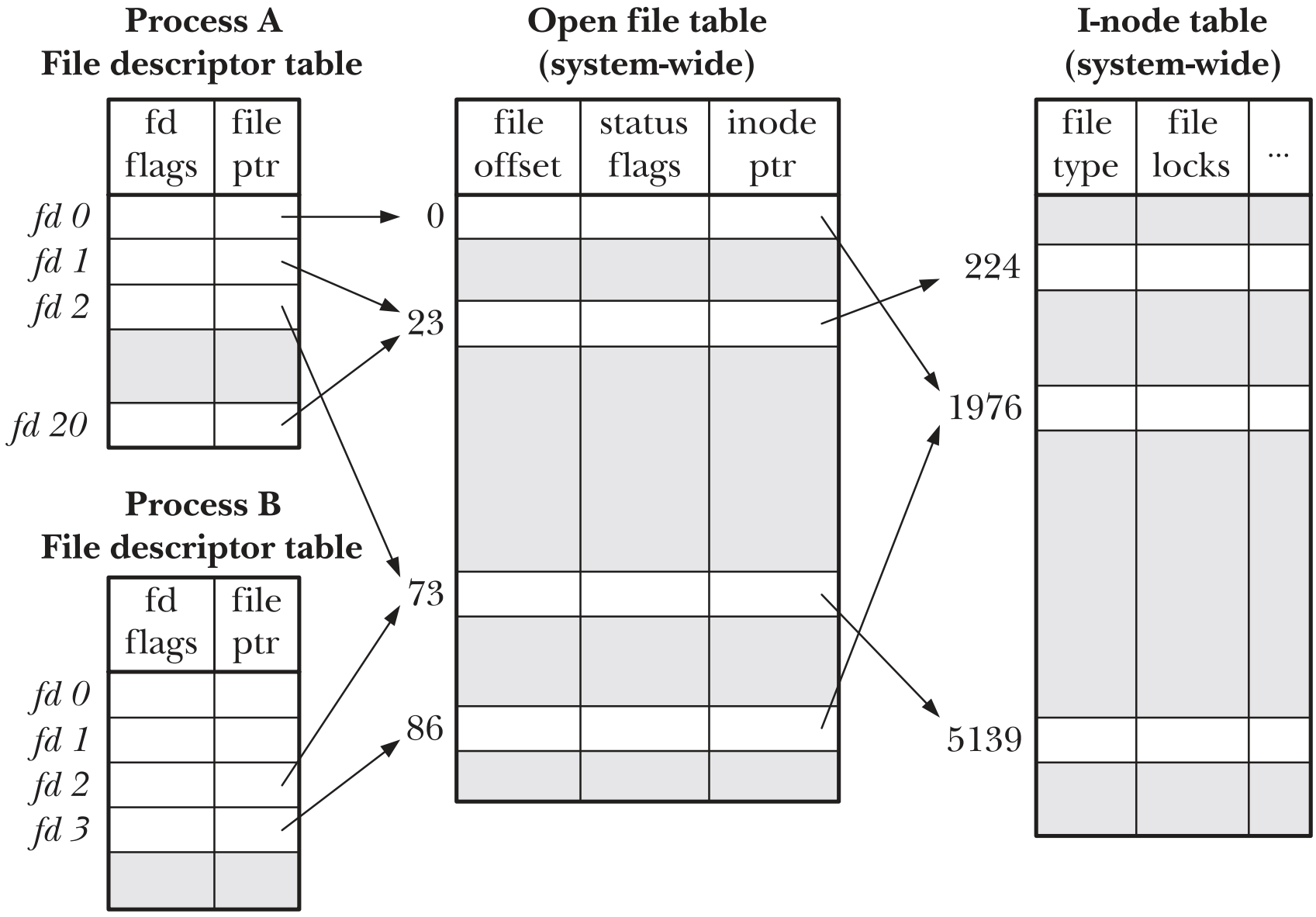


Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

Exec and File Descriptors

When a program calls `exec`, the new program still retains the file descriptors of the original process.

But the `FILE *` variables are gone, upon replacing the process image with the new program.

So the new program must either:

1. Perform low-level I/O using the `read()` or `write()` system calls; or
2. Use `fdopen()` to associate a new buffered file stream with an existing open file descriptor.

dup and dup2

```
int dup(int oldfd)

int dup2(int oldfd, int newfd)
```

- `dup` returns a new FD that refers to the same file as `oldfd`
- `dup2` does the same, but lets you specify the value of new FD
  - `dup2` first closes `newfd` if already in use

Output Redirection with dup2

```c
int main (void)
{
    int fd = open("lsout", O_WRONLY | O_CREAT, 0600);
    if (fd == -1) {
        perror("open");
        exit(1);
    }

    dup2(fd, STDOUT_FILENO) ;
    execl("/bin/ls", "ls", "-l", (char *)NULL);
    perror("execl");
    return 1;
}
```

Figures Credit

Figure 5-2 and Table 27-1 are from The Linux Programming Interface by Michael Kerrisk.

Extra Slides

Shell Skeleton Code

```
while (1) { // Infinite
    print_prompt();
    read_command(command, parameters);
    if (fork ()) { // Parent
        wait(&status);
    } else {
        execve(command, parameters, NULL);
    }
}
```

Security Implications of Exec Functions

Using `execlp` and `execvp` can be very dangerous when used improperly. Can you find out why?

Hint: See Section 27.2.1 of The Linux Programming Interface by Michael Kerrisk.