CSE 410: Systems Programming

The Process Environment

Ethan Blanton

Department of Computer Science and Engineering
University at Buffalo

Last Time

- Programs *vs.* processes
- ELF
- Process segments (text, data, BSS)
- Heap and stack

The Process Environment

In addition to its memory, a process has a complex environment.

Kernel services:

- System calls
- Filesystem
- Signals

Its lifecycle:

- Creation
- Execution of a new program
- Destruction

Kernel Services

The kernel performs services on behalf of processes.

Recall that POSIX systems provide:

- Memory isolation
- The illusion of a dedicated CPU

To enforce this, hardware assistance is required.

Only the kernel can configure those hardware features!

Therefore, processes must request access to shared resources from the kernel.

This is accomplished via system calls.

Process Lifecycle

Process memory spaces must be created by the kernel.

Therefore, the process must be created by the kernel.

Once created, the process must execute some program.

When finished, the process's resources must be cleaned up:

- Memory
- Files
- Other shared resources

Kernel/Userspace Separation

The kernel manages all shared resources in a POSIX system.

- Memory
- Files
- Hardware devices (mouse, keyboard, display)
- …

The kernel runs in supervisor mode[1] to give it access to these.

Processes run in user mode in an environment created by the kernel that we call userspace.

[1] This term varies from architecture to architecture. On x86_64, we often say “ring 0”.

Protection Domains

Supervisor mode and user mode are protection domains.

Moving between protection domains requires hardware assistance.

Therefore, system calls cannot be simple functions.

On our x86_64 Linux system, system calls are accessed via a software interrupt.

This is a hardware-supported feature.

Invoking a Function

A normal function call involves:

- Placing function arguments in particular registers or on the stack in known positions
- Placing the current program counter on the stack
- Changing the program counter to the first instruction of the called function

When the function completes, it:

- Places its return value in a particular register
- Retrieves the previous program counter from the stack
- Changes the program counter to the calling location

Invoking a System Call

A system call has a special invocation:

- The system call number is placed in a particular register
- The system call arguments are placed in other registers
- The the syscall processor instruction is invoked

Then the CPU hardware:

- Changes protection domains
- Jumps to a well-known location

The system call executes, and then:

- Places its return value in a particular register
- Invokes the sysret processor instruction

The CPU hardware:

- Changes back to user mode
- Jumps to the calling function

System Calls on Other Platforms

Note that system calls used specific processor instructions.

Different processors, and different models of compatible processors, may use different instructions.

For example, x86 32-bit uses int 0x80 or sysenter.

In addition, different operating systems may be different!

Crossing Protection Domains

The kernel memory map differs from a process memory map.

If a system call (*e.g.*, `read()`) passes a pointer to the kernel, the kernel must do extra work to use it.

- It must check that the pointer is mapped in the process
- It must ensure that the entire buffer is valid
- It may have to check that the process can write at that address

This prevents system crashes and security exploits.

System Call Functions

If system calls are so complicated, how can a process call a system call like `write()` directly?

System Call Functions

If system calls are so complicated, how can a process call a system call like `write()` directly?

The C library provides wrapper functions for system calls.

These wrappers:

- Set up the appropriate registers
- Call the necessary processor instructions
- Retrieve the return value
- Return normally

This is purely for convenience.

Process Creation

UNIX historically had only one way to create a new process: the `fork()` system call.

`fork()` duplicates the calling process by (among other things):

- Creating a new process ID (PID) and kernel structures
- Creating a new memory space for the new process
- Copying the entire contents of the current process into the new memory space
- Returning execution from the `fork()` call in both processes

In the original process, `fork()` returns the new PID.

In the new process, `fork()` returns zero.

Process Families

Every POSIX process[2] has a parent process.

A process may have child processes if it has called `fork()` or `posix_spawn()`.

If a process's parent process terminates, it becomes an orphan.

An orphaned process will be adopted by `init`.

The family of all processes forms a tree.

[2] Except the special process `init`, which always has PID 1.

The fork()/exec() Model

Note that a forked process must run the same program as its creator!

POSIX also provides a system call to execute a program: exec()

exec() replaces the current process image with a new program.

The fork()/exec() model has advantages and disadvantages.

Many systems provide a single call to:

- Create a new process
- Execute a new program in that process

Modern POSIX systems provide posix_spawn() for this purpose.

exec()

The exec() system call is actually a whole family of calls that load a named executable file.

```
execl("/bin/ls", "ls", "-F", "/", NULL);
```

Output:

```
afs/
bin/
boot/
dev/
etc/
...
```

Fork in Action

```
pid_t pid = fork();

if (pid == 0) {
    puts("In child");
} else {
    printf("In parent, child PID = %d\n", pid);
}
```

Fork in Action

```
pid_t pid = fork();

if (pid == 0) {
    puts("In child");
} else {
    printf("In parent, child PID = %d\n", pid);
}
```

Output:

```
In parent, child PID = 9095
In child
```

Fork in Action

```
In parent, child PID = 9095
In child
```

Note that it appears that both branches of the if were taken.

Fork in Action

```
In parent, child PID = 9095
In child
```

Note that it appears that both branches of the if were taken.

In fact, both branches were taken.

…but only one of them in each of two processes.

Note that the order here is not predictable.

Process Termination

A process terminates when:

- It calls the system call `exit()`
- It returns from `main()`
- It recieves and and fails to catch certain signals (*e.g.*, `SIGSEGV`; more on signals later!)

In the first two cases, it returns a chosen value:

- The integer argument to `exit()`
- The integer return value of `main()`

In the third case, it returns a special value indicating that it was killed by a signal (and which signal).

Detecting Process Termination

The `wait()` family of system calls allows a program to detect process termination.

A process can `wait()` for any of its child processes.

- This is called reaping.
- If a process terminates and is not reaped, it becomes a zombie.
- Zombie processes consume (minimal) system resources.
- Orphan processes will be reaped by `init`.

Wait in Action

```
pid_t pid = fork();
if (pid == 0) {
    puts("In child");
    exit(42);
} else {
    int status;
    waitpid(pid, &status, 0);
    printf("Child exited with status %d\n",
       WEXITSTATUS(status));
}
```

Output:

```
In child
Child exited with status 42
```

Wait in Action

```
In child
Child exited with status 42
```

This order is deterministic.

The call to `waitpid()` will not return until the child terminates.

Other Environmental Features

A process's environment also includes:

- A current working directory
- Environment variables
- Open files

These are maintained in cooperation with the kernel.

Current Working Directory

Every process has a current working directory.

All relative file paths are with respect to this directory.

This directory can be:

- set with the system call `chdir()`
- retrieved with `getcwd()`.

There is also a function `getwd()`, but it is dangerous and should not be used.

Environment Variables

Every process has environment variables.

- They are stored in a global array called `environ`
- A single variable can be retrieved by `getenv()`
- A variable can be set with `setenv()`

`environ` is duplicated by the kernel on `fork()`

Environment Variables in Action

```
char *homedir = getenv("HOME");
puts(homedir);
```

Output:

```
/home/elb
```

Open Files

The kernel maintains open files for every process.

Each open file is identified by an integer file descriptor.

The position of the most recent read or write is maintained for each file descriptor.

File descriptors are duplicated on fork.
(This is how both the child and parent wrote in the `fork()` example!)
These duplicated descriptors share their position information.

A file descriptor can be optionally closed on `exec()`.

Summary

- The kernel manages shared resources
- Userspace and the kernel are in different protection domains
- Processes request services from the kernel using system calls
- UNIX processes are created with `fork()`
- The `exec()` system call loads a new program
- The kernel manages other state for processes, such as:
  - The current directory
  - Environment variables
  - Open files

Next Time …

- Dynamic Allocator Project

References I

Required Readings

[1] Randal E. Bryant and David R. O'Hallaron. *Computer Science: A Programmer's Perspective*. Third Edition. Chapter 8: 8.2, 8.4. Pearson, 2016.

License

To retrieve a copy of this material, or related materials, see https://www.cse.buffalo.edu/~eblanton/.