CS 241: Systems Programming
Lecture 27. System Calls II

Spring 2020

Prof. Stephen Checkoway

Creating a new process

Two schools of thought

- Windows way: single system call
    - `CreateProcess("calc.exe", /* other params */)`
- Unix way: two (or more) system calls
    - Create a copy of the currently running process: `fork()`
    - Transform the copy into a new process: `execve("/usr/bin/bc", args, env)`

Process IDs

Every Unix process has a unique identifier

- Integer, used to index into a kernel process table
- `$ ps ax` # Print a list of all running processes and their PIDs

```
pid_t getpid(void);
```

Every process has a parent process

- processes are "reparented" to the `init` process if your parent already died

```
pid_t getppid(void);
```

Running another program

```
int execve(char const *path, char *const argv[],
           char *const envp[]);
```

- Last element of `argv[]` and `envp[]` must be `0` (`NULL`)
- If successful, `execve` won't return, instead, the OS will remove all of the process's code and data and load the program from `path` in its place and start running that
- The PID of the process doesn't change
- The open file descriptors remain open (unless marked close on exec)
- Returns `-1` and sets `errno` on error

The types of argv and envp

`execve(path, argv, envp)` does not modify its arguments

For historical reasons, `argv` and `envp` have type

- `char *const[]` — this is a constant pointer to `char *`
- We really want `char const *const[]` which is a constant pointer to `char const *`
- Normally, we pass a `char *argv[]` array (no `const`)

The types of argv and envp

We can deal with this in one of two ways

- For historical reasons, we can assign string literals to `char *`

  ```
  char *s = "foo"; // normally char const *s = "foo";
  ```

- We can **cast** a `char const *` to a `char *`

  ```
  char const *t = /* … */;
  char *u = (char *)t;
  ```

- If you omit the cast, you get a compiler warning; compiler warnings should not be ignored

```c
#include <err.h>
#include <stdlib.h>
#include <unistd.h>

void run_with_args(char const *program) {
  char *args[] = {
    (char *)program,        // argv[0]
    "This is one argument", // argv[1]
    "two",                  // argv[2]
    "three",                // argv[3]
    0,                      // argv[4] is NULL, end of args
  };
  char *env[] = { 0 }; // Empty environment.
  execve(program, args, env);
  err(EXIT_FAILURE, "%s", args[0]);
}

int main(int argc, char *argv[]) {
  run_with_args(argc == 1 ? "/bin/echo" : argv[1]);
}
```

exec(3) family

```
int execl(const char *path, const char *arg0, ...,
          (char *)0);
int execle(const char *path, const char *arg0, ...,
           (char *)0, char *const envp[]);
int execlp(const char *program, const char *arg0, ...,
           (char *)0);
int execv(const char *path, char *const argv[]);
int execvp(const char *program, char *const argv[]);
```

- `execl`, `execle`, `execlp` take 0-terminated variable number of arguments
- The `argv` and `envp` arrays must be 0-terminated
- execlp and execvp search PATH for the program
- glibc has an execvpe which is like execve but searches the PATH

Which of the following statements about `execve()` is false?

A. If execve() is successful, the new program replaces the calling program.

B. The file descriptors that were open before execve() are open in the new program (except for those marked as close on exec).

C. If `execve()` has an error, it returns -1 and sets **`errno`**.

D. If `execve()` is successful, it returns 0.

Creating a new process

```
#include <unistd.h>
#include <sys/types.h>

pid_t fork(void);
```

Creates an (almost) identical copy of the running program with one big exception

- Returns `0` to the child but PID of child to the parent
- `-1` on error and sets `errno`

This includes a copy of memory, code, file descriptors and most other bit of process state (but not all)

```
#include <sys/types.h>
#include <sys/wait.h>
#include <err.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void whoami(char const *str) {
  pid_t self = getpid();
  pid_t parent = getppid();
  printf("%s: pid=%d ppid=%d\n",
   str, self, parent);
}
```

```
int main(void) {
  whoami("Prefork");
  pid_t pid = fork();
  if (pid < 0)
    err(EXIT_FAILURE, "fork");
  if (pid == 0) {
    whoami("Child");
  } else {
    whoami("Parent");
    int status;
    wait(&status);
  }
  return 0;
}
```

```
#include <sys/types.h>
```

```
Prefork: pid=48627 ppid=28834
Parent: pid=48627 ppid=28834
Child: pid=48628 ppid=48627
```

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

void whoami(char const *str) {
  pid_t self = getpid();
  pid_t parent = getppid();
  printf("%s: pid=%d ppid=%d\n",
   str, self, parent);
}
```

```
int main(void) {
  whoami("Prefork");
  pid_t pid = fork();
  if (pid < 0)
    err(EXIT_FAILURE, "fork");
  if (pid == 0) {
    whoami("Child");
  } else {
    whoami("Parent");
    int status;
    wait(&status);
  }
  return 0;
}
```

fork/exec

Usually used together

`fork` to create a duplicate process

`exec` (one of the exec family that is) to run a new program

`fork` and `exec` both preserve file descriptors

- This is how bash operates: it forks, sets file descriptors, and execs

After a `fork`, you have two copies of a program, the parent and the child, and...

A. Either the parent or the child must call `exec()` immediately

B. The parent gets a PID and the child gets a 0 as return values

C. The child gets a PID and the parent gets a 0 as return values

D. Both parent and child get PIDs as the return values

E. Both parent and child must call `exec` to proceed

Process exit status

Can wait for a child process to die (or be stopped, e.g., by a debugger)

```
#include <sys/wait.h>

int status;
pid_t pid = wait(&status);
```

Suspends execution until child terminates, returns the PID of the child

Checking exit status

Use macros to examine exit status

`WIFEXITED``(status)`

- True if the process terminated normally

`WEXITSTATUS``(status)`

- Returns actual return/exit value if **`WIFEXITED`**`(status)` is true

`WIFSIGNALED``(status)`

- True if the process was terminated by a signal (e.g., **`SIGINT`** from ctrl-C)

`WTERMSIG``(status)`

- Returns the signal that terminated the process if **`WIFSIGNALED`**`(status)`

strace(1)

strace is a Linux program that prints out the system calls a program uses

- `-e trace=open,openat,close,read,write` will trace those system calls
- `-f` will trace children too
- `-s size` will show up to `size` bytes of strings

```
$ strace -e trace=open,openat,close,read,write cat Makefile
...
openat(AT_FDCWD, "Makefile", O_RDONLY)  = 3
read(3, "CC := clang\nCFLAGS := -Wall -std"..., 1048576) = 176
write(1, "CC := clang\nCFLAGS := -Wall -std"..., 176) = 176
read(3, "", 1048576)                    = 0
close(3)                                = 0
...
```

In-class exercise

https://checkoway.net/teaching/cs241/2020-spring/exercises/Lecture-27.html

Grab a laptop and a partner and try to get as much of that done as you can!