Lecture 05: Understanding `execvp`

Principles of Computer Systems

Fall 2019

Stanford University

Computer Science Department

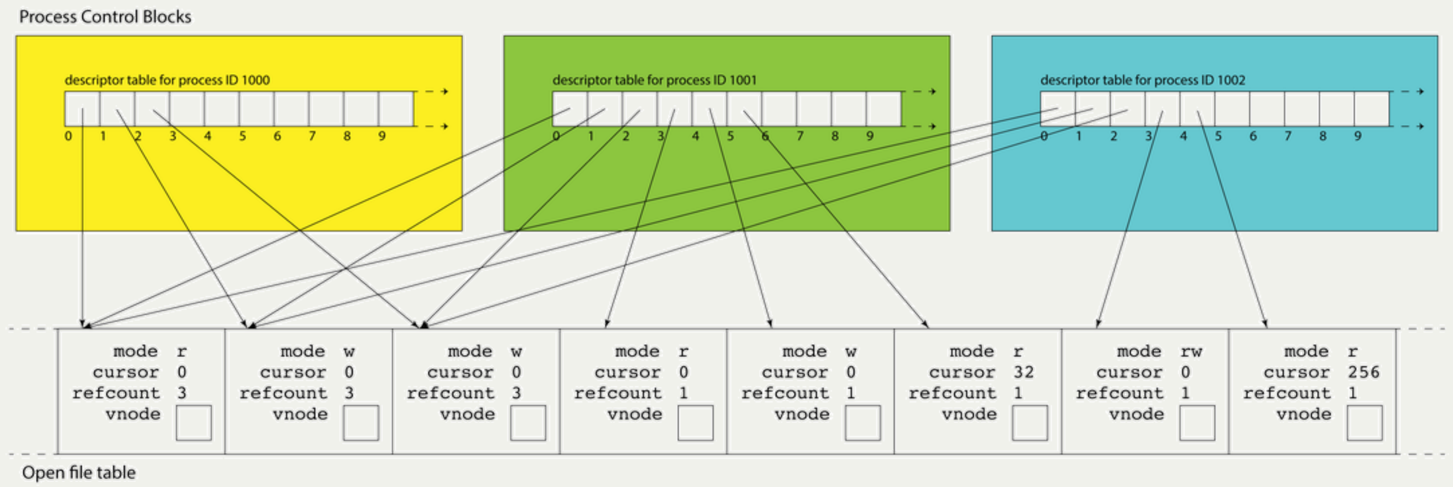
Instructors: Chris Gregg and

Phil Levis

PDF of this presentation

Lecture 05: Question from yesterday: where is the cursor stored for a file?

Diagram from last lecture:


Process Control Blocks
descriptor table for process ID 1000
descriptor table for process ID 1001
descriptor table for process ID 1002
0 1 2 3 4 5 6 7 8 9
mode r cursor 0 refcount 3 vnode
mode w cursor 0 refcount 3 vnode
mode w cursor 0 refcount 3 vnode
mode r cursor 0 refcount 1 vnode
mode w cursor 0 refcount 1 vnode
mode r cursor 32 refcount 1 vnode
mode rw cursor 0 refcount 1 vnode
mode r cursor 256 refcount 1 vnode
Open file table

- In the last lecture, the question came up about multiple processes pointing to the open file table, and what happens to the cursor.
  - Every time there is an **`open()`** system call, a new entry is generated for the open file table.
  - If a process **`fork`**s, both the parent and child share a pointer to the same open file entry, and therefore their cursors will be the same (i.e., if one reads a line, the other will read the next line). This is also true with the **`dup`** and **`dup2`** system calls, which Phil mentioned last time, and which we will learn about (briefly) in lab and in more detail next week.

Assignment 2: The Unix v6 Filesystem

- Your second assignment of the quarter is to write a program in C (not C++) that can read from a 1970s-era Unix version 6 filesystem.
  - The test data you are reading from are literally bit-for-bit representations of a Unix v6 disk.
  - You will leverage all of the information covered in the file system lecture from Lecture 3 to write the program, and for more detailed information, see Section 2.5 of the Salzer and Kaashoek textbook.
- You will primarily be writing code in four different files (and we suggest you tackle them in this order):
  - **`inode.c`**
  - **`file.c`**
  - **`directory.c`**
  - **`pathname.c`**
- Because the program is in C, you will have to rely on arrays of structs, and low-level data manipulation, as you don't have access to any C++ standard template library classes.

Assignment 2: The Unix v6 Filesystem

- The basic idea of the assignment is to write code that will be able to locate and read files in the file system. You will, for example, be using a function we've written for you to read sectors from the disk image:

```
/**
 * Reads the specified sector (e.g. block) from the disk.  Returns the number of bytes read,
 * or -1 on error.
 */
int diskimg_readsector(int fd, int sectorNum, void *buf);
```

- Sometimes, **`buf`** will be an array of **`inode`**s, sometimes it will be a buffer that holds actual file data. In any case, the function will *always* read **`DISKIMG_SECTOR_SIZE`** number of bytes, and it is your job to determine the relevance of those bytes.
- It is critical that you carefully read through the header files for this assignment (they are actually relatively short). There are key constants (e.g., **`ROOT_INUMBER`**, **`struct direntv6`**, etc.) that are defined for you to use, and reading them will give you an idea about where to start for parts of the assignment.

Assignment 2: The Unix v6 Filesystem

- One function that can be tricky to write is the following:

```c
/**
 * Gets the location of the specified file block of the specified inode.
 * Returns the disk block number on success, -1 on error.
 */
int inode_indexlookup(struct unixfilesystem *fs, struct inode *inp, int blockNum);
```

- The **`unixfilesystem`** struct is defined and initialized for you.
- The **`inode`** struct will be populated already
- The **`blockNum`** is the number, in linear order, of the block you are looking for in a particular file.

- For example:
  - Let's say the **`inode`** indicates that the file it refers to has a size of 180,000 bytes. And let's assume that **`blockNum`** is **`302`**.
  - This means that we are looking for the 302nd block of data in the file referred to by **`inode`**.
  - Recall that blocks are 512 bytes long.
  - How would you find the block index (i.e., sector index) of the 302nd block in the file?

Assignment 2: The Unix v6 Filesystem

- For example:
  - Let's say the **`inode`** indicates that the file it refers to has a size of 180,000 bytes. And let's assume that **`blockNum`** is **`302`**.
  - This means that we are looking for the 302nd block of data in the file referred to by **`inode`**.
  - Recall that blocks are 512 bytes long
  - How would you find the block index (i.e., sector index) of the 302nd block in the file?

a. Determine if the file is large or not
b. If it isn't large, you know you only have direct addressing.
c. If it is large (this file is), then you have indirect addressing.
d. The 302nd block is going to fall into the *second* indirect block, because each block has 256 block numbers (each block number is an **`unsigned short`**, or a **`uint16_t`**).
e. You, therefore, need to use **`diskimg_readsector`** to read the sector listed in the 2nd block number (which is in the **`inode`** struct), then extract the (302 % 256)th short from that block, and return the value you find there.
f. If the block number you were looking for happened to fall into the 8th inode block, then you would have a further level of indirection for a doubly-indirect lookup.

Assignment 2: The Unix v6 Filesystem

- For the assignment, you will also have to search through directories to locate a particular file.
  - You *do not* have to follow symbolic links (you can ignore them completely)
  - You do need to consider directories that are longer than 32 files long (because they will take up more than two blocks on the disk), *but* this is not a special case! You are building generic functions to read files, so you can rely on them to do the work for you, even for directory file reading.
  - Don't forget that a filename is limited to 14 characters, and if it is exactly 14 characters, there is *not* a trailing `'\0'` at the end of the name (this to conserve that one byte of data!) So...you might want to be careful about using **`strcmp`** for files (maybe use **`strncmp`**, instead?)

- This is a relatively advanced assignment, with a lot of moving parts.
- Start early!
- Come to office hours or ask Piazza questions.
- Remember: CAs *won't* look at your code, so you must formulate your questions to be conceptual enough that they can be answered.

Lecture 05: Continuing from last time: brief review of fork()

- As Phil discussed last time, the **`fork()`** system call is used to *spawn* a *child* process. Once you call **`fork()`** you now have two *identical* processes that each continue at the next line of code. The *only* difference is that the return value from **`fork()`** is 0 if the process is the child, and it is the child's process id (pid) if the process is the parent. Otherwise, *everything* else is the same, and a complete copy of the memory system is made for the child. Let's investigate this copy a bit more:

```c
int main(int argc, char *argv[]) {
    char str[128];
    strcpy(str, "SpongeBob");
    printf("str's addres is %p\n", str);
    pid_t pid = fork();
    if (pid == 0) { // child
        printf("I am the child. str's address is %p\n", str);
        strcpy(str, "SquarePants");
        printf("I am the child and I changed str to %s. str's address is still %p\n", str, str);
    } else {
        printf("I am the parent. str's address is %p\n", str);
        printf("I am the parent, and I'm going to sleep for 2 seconds.\n");
        sleep(2);
        printf("I am the parent. I just woke up. str's address is %p, and it's value is %s\n", str, str);
    }

    return 0;
}
```

Lecture 05: Continuing from last time: brief review of fork()

Output:

```
$ ./fork-copy
str's addres is 0x7ffe092639d0
I am the parent. str's address is 0x7ffe092639d0
I am the parent, and I'm going to sleep for 2 seconds.
I am the child. str's address is 0x7ffe092639d0
I am the child and I changed str to SquarePants. str's address is still 0x7ffe092639d0
I am the parent. I just woke up. str's address is 0x7ffe092639d0, and it's value is SpongeBob
```

Lecture 05: Continuing from last time: brief review of fork()

Output:

```
$ ./fork-copy
str's addres is 0x7ffe092639d0
I am the parent. str's address is 0x7ffe092639d0
I am the parent, and I'm going to sleep for 2 seconds.
I am the child. str's address is 0x7ffe092639d0
I am the child and I changed str to SquarePants. str's address is still 0x7ffe092639d0
I am the parent. I just woke up. str's address is 0x7ffe092639d0, and it's value is SpongeBob
```

What? How can two processes have the same pointer but the value the pointer points to has different values in each process?

Lecture 05: Continuing from last time: brief review of fork()

Output:

```
$ ./fork-copy
str's addres is 0x7ffe092639d0
I am the parent. str's address is 0x7ffe092639d0
I am the parent, and I'm going to sleep for 2 seconds.
I am the child. str's address is 0x7ffe092639d0
I am the child and I changed str to SquarePants. str's address is still 0x7ffe092639d0
I am the parent. I just woke up. str's address is 0x7ffe092639d0, and it's value is SpongeBob
```

What? How can two processes have the same pointer but the value the pointer points to has different values in each process?

Virtualization. There is a translation that happens between a process and the OS and hardware. For both processes above, the pointer value is `0x7ffe092639d0`. But, in *physical* memory, there has been a translation, so that there are actually two different memory locations. In other words: a pointer does not necessarily reflect the physical memory address.

Lecture 05: Continuing from last time: brief review of fork()

Output:

```
$ ./fork-copy
str's addres is 0x7ffe092639d0
I am the parent. str's address is 0x7ffe092639d0
I am the parent, and I'm going to sleep for 2 seconds.
I am the child. str's address is 0x7ffe092639d0
I am the child and I changed str to SquarePants. str's address is still 0x7ffe092639d0
I am the parent. I just woke up. str's address is 0x7ffe092639d0, and it's value is SpongeBob
```

Your next question might be: wait, isn't that expensive? Copying everything owned by a process when it forks? Yes, and no.

- First: creating new processes does have some overhead, so there are some inefficiencies.
- Second: until one of the processes makes a change to a memory location, both pointers actually *do* point to the same place in physical memory, and only after a modification (when the processes need to see different values) is there a copying of the memory necessary to make the change. This is completely hidden from the process, which behaves as you are used to in any C or C++ program.

Lecture 05: More on Multiprocessing

- **Synchronizing between parent and child using `waitpid`**
  - `waitpid` can be used to temporarily block one process until a child process exits.

```
pid_t waitpid(pid_t pid, int *status, int options);
```

  - The first argument specifies the wait set, which for the moment is just the id of the child process that needs to complete before `waitpid` can return.
  - The second argument supplies the address of an integer where termination information can be placed (or we can pass in `NULL` if we don't care for the information).
  - The third argument is a collection of bitwise-or'ed flags we'll study later. For the time being, we'll just go with 0 as the required parameter value, which means that `waitpid` should only return when a process in the supplied wait set exits.
  - The return value is the pid of the child that exited, or `-1` if `waitpid` was called and there were no child processes in the supplied wait set.

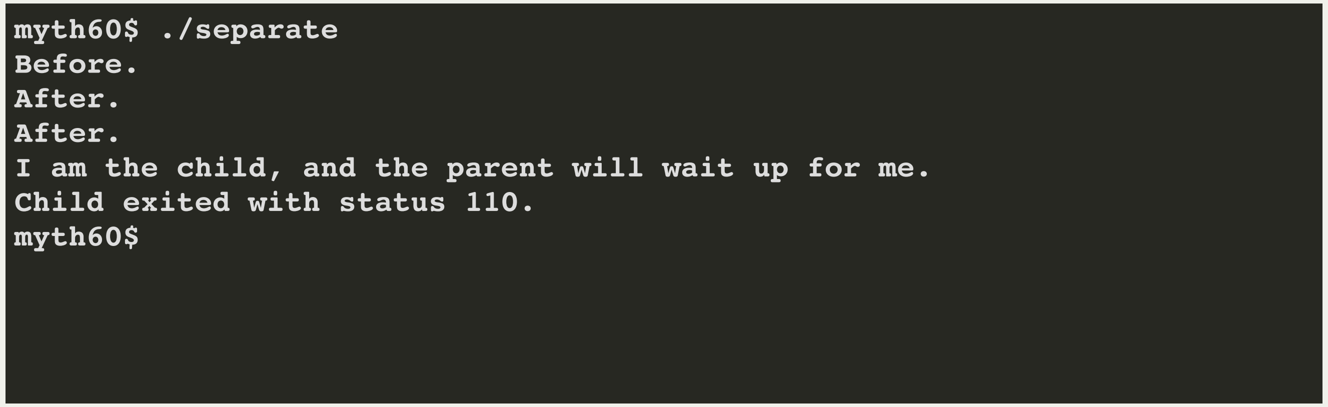
Lecture 05: More on Multiprocessing

- **Synchronizing between parent and child using `waitpid`**
  - Consider the following program, which is more representative of how **`fork`** really gets used in practice (full program, with error checking, is right here):

```
int main(int argc, char *argv[]) {
  printf("Before.\n");
  pid_t pid = fork();
  printf("After.\n");
  if (pid == 0) {
    printf("I am the child, and the parent will wait up for me.\n");
    return 110; // contrived exit status
  } else {
    int status;
    waitpid(pid, &status, 0)
    if (WIFEXITED(status)) {
      printf("Child exited with status %d.\n", WEXITSTATUS(status));
    } else {
      printf("Child terminated abnormally.\n");
    }
    return 0;
  }
 }
```

Lecture 05: More on Multiprocessing

- **Synchronizing between parent and child using `waitpid`**
  - The output is the same every single time the above program is executed.



```
myth60$ ./separate
Before.
After.
After.
I am the child, and the parent will wait up for me.
Child exited with status 110.
myth60$
```

  - The implementation directs the child process one way, the parent another.
  - The parent process correctly waits for the child to complete using **`waitpid`**.
  - The parent lifts child exit information out of the **`waitpid`** call, and uses the **`WIFEXITED`** macro to examine some high-order bits of its argument to confirm the process exited normally, and it uses the **`WEXITSTATUS`** macro to extract the lower eight bits of its argument to produce the child return value (which we can see is, and should be, 110).
  - The **`waitpid`** call also donates child process-oriented resources back to the system.

Lecture 05: More on Multiprocessing

- **Spawning and synchronizing with multiple child processes**
  - A parent can call **`fork`** multiple times, provided it reaps the child processes (via **`waitpid`**) once they exit. If we want to reap processes as they exit without concern for the order they were spawned, then this does the trick (full program checking right here):

```
int main(int argc, char *argv[]) {
  for (size_t i = 0; i < 8; i++) {
    if (fork() == 0) exit(110 + i);
  }
  while (true) {
    int status;
    pid_t pid = waitpid(-1, &status, 0);
    if (pid == -1) { assert(errno == ECHILD); break; }
    if (WIFEXITED(status)) {
      printf("Child %d exited: status %d\n", pid, WEXITSTATUS(status));
    } else {
      printf("Child %d exited abnormally.\n", pid);
    }
  }
  return 0;
}
```

Lecture 05: More on Multiprocessing

- **Spawning and synchronizing with multiple child processes**
  - Note we feed a -1 as the first argument to **`waitpid`**. That -1 states we want to hear about **any** child as it exits, and pids are returned in the order their processes finish.
  - Eventually, all children exit and **`waitpid`** correctly returns -1 to signal there are no more processes under the parent's jurisdiction.
  - When **`waitpid`** returns -1, it sets a global variable called **`errno`** to the constant **`ECHILD`** to signal **`waitpid`** returned -1 because all child processes have terminated. That's the "error" we want.

```
myth60$ ./reap-as-they-exit
Child 1209 exited: status 110
Child 1210 exited: status 111
Child 1211 exited: status 112
Child 1216 exited: status 117
Child 1212 exited: status 113
Child 1213 exited: status 114
Child 1214 exited: status 115
Child 1215 exited: status 116
myth60$
```

```
myth60$ ./reap-as-they-exit
Child 1453 exited: status 115
Child 1449 exited: status 111
Child 1448 exited: status 110
Child 1450 exited: status 112
Child 1451 exited: status 113
Child 1452 exited: status 114
Child 1455 exited: status 117
Child 1454 exited: status 116
myth60$
```

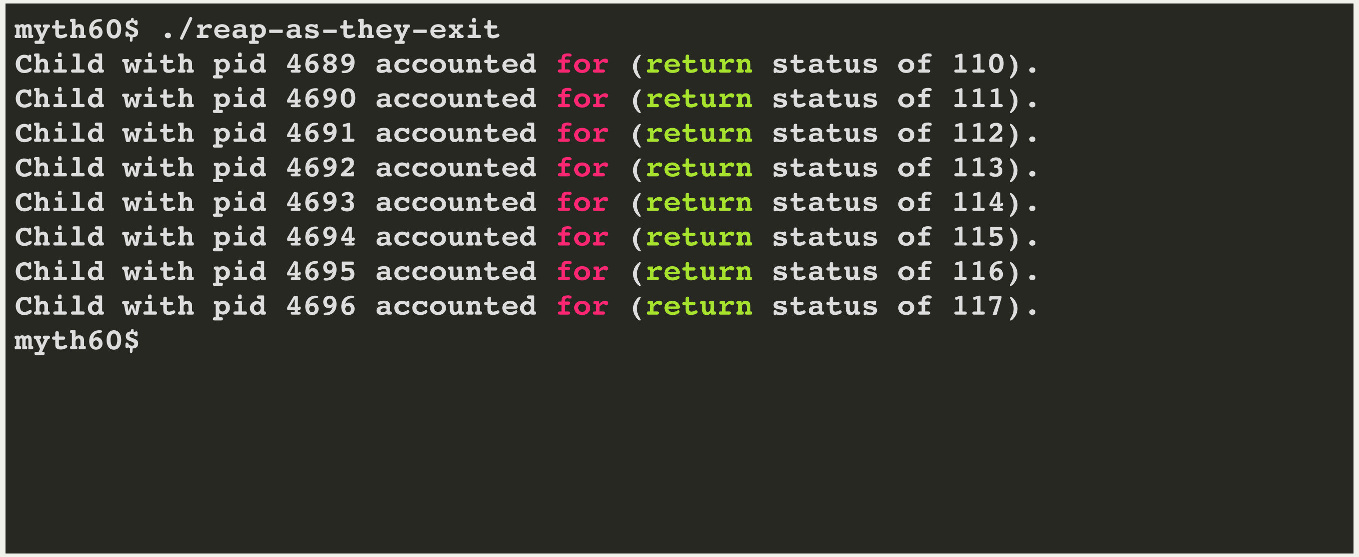
Lecture 05: More on Multiprocessing

- **Spawning and synchronizing with multiple child processes**
  - We can do the same thing we did in the first program, but monitor and reap the child processes in the order they are forked.
  - Check out the abbreviated program below (full program with error checking right here):

```
int main(int argc, char *argv[]) {
  pid_t children[8];
  for (size_t i = 0; i < 8; i++) {
    if ((children[i] = fork()) == 0) exit(110 + i);
  }
  for (size_t i = 0; i < 8; i++) {
    int status;
    pid_t pid = waitpid(children[i], &status, 0);
    assert(pid == children[i]);
    assert(WIFEXITED(status) && (WEXITSTATUS(status) == (110 + i)));
    printf("Child with pid %d accounted for (return status of %d).\n",
           children[i], WEXITSTATUS(status));
  }
  return 0;
}
```

Lecture 05: More on Multiprocessing

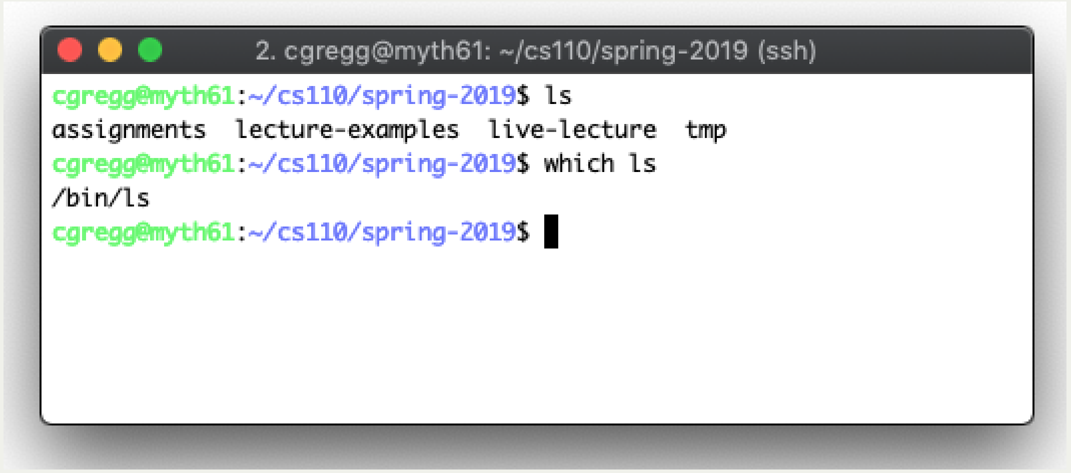
- **Spawning and synchronizing with multiple child processes**
  - This version spawns and reaps processes in some first-spawned-first-reaped manner.
  - The child processes aren't required to exit in FSFR order.
  - In theory, the first child thread could finish last, and the reap loop could be held up on its very first iteration until the first child really is done. But the process zombies—yes, that's what they're called—are reaped in the order they were forked.
  - Below is a sample run of the `reap-in-fork-order` executable. The pids change between runs, but even those are guaranteed to be published in increasing order.



```
myth60$ ./reap-as-they-exit
Child with pid 4689 accounted for (return status of 110).
Child with pid 4690 accounted for (return status of 111).
Child with pid 4691 accounted for (return status of 112).
Child with pid 4692 accounted for (return status of 113).
Child with pid 4693 accounted for (return status of 114).
Child with pid 4694 accounted for (return status of 115).
Child with pid 4695 accounted for (return status of 116).
Child with pid 4696 accounted for (return status of 117).
myth60$
```

Lecture 05: New system call: `execvp`

- It is possible to have a **`fork`**ed process simply do other work that you program. In other words, you have two processes, each doing work concurrently, and you've programmed the code for both processes. These are the examples we've seen so far.
- However, this is actually *not* the most common use for **`fork`**. Most often, a programmer wants to run a *completely separate program*, but wants to maintain control over the program, and may also (quite frequently) want to send data to the program through **`stdin`** and capture the output of the program through its **`stdout`**.
- This is what your shell does whenever you launch a program. The shell is a program, and when you type a command, it executes that program, and waits for it to end.



```
2. cgregg@myth61: ~/cs110/spring-2019 (ssh)
cgregg@myth61:~/cs110/spring-2019$ ls
assignments  lecture-examples  live-lecture  tmp
cgregg@myth61:~/cs110/spring-2019$ which ls
/bin/ls
cgregg@myth61:~/cs110/spring-2019$ 
```

- In the screenshot to the left, the terminal shell is a program, and after you type ls, the shell runs the **`ls`** program, located at **`/bin/ls`**.
- The shell waits for **`ls`** to finish, and then reprompts for another command.

Lecture 05: New system call: `execvp`

- Enter the **`execvp`** system call!
  - **`execvp`** effectively reboots a process to run a different program from scratch. Here is the prototype:

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

    - **`path`** identifies the name of the executable to be invoked.
    - **`argv`** is the argument vector that should be funneled through to the new executable's **`main`** function.
    - For the purposes of CS110, **`path`** and **`argv[0]`** end up being the same exact string.
    - If **`execvp`** fails to cannibalize the process and install a new executable image within it, it returns -1 to express failure.
    - If **`execvp`** succeeds, it never returns in the calling process. #deep
  - **`execvp`** has many variants (**`execle`**, **`execlp`**, and so forth. Type **`man execvp`** to see all of them). We generally rely on **`execvp`** in this course.

Lecture 05: New system call: `execvp`

- First example using `execvp`? An implementation `mysystem` to emulate the behavior of the libc function called `system`.
  - Here we present our own implementation of the `mysystem` function, which executes the supplied `command` as if we typed it out in the terminal ourselves, ultimately returning once the surrogate `command` has finished.
  - If the execution of `command` exits normally (either via an `exit` system call, or via a normal return statement from `main`), then our `mysystem` implementation should return that exact same exit value.
  - If the execution exits abnormally (e.g. it segfaults), then we'll assume it aborted because some signal was ignored, and we'll return that negative of that signal number (e.g. -11 for `SIGSEGV`).

Lecture 05: New system call: `execvp`

- Here's the implementation, with minimal error checking (the full version is right here):

```c
static int mysystem(const char *command) {
  pid_t pid = fork();
  if (pid == 0) {
    char *arguments[] = {"/bin/sh", "-c", (char *) command, NULL};
    execvp(arguments[0], arguments);
    printf("Failed to invoke /bin/sh to execute the supplied command.");
    exit(0);
  }
  int status;
  waitpid(pid, &status, 0);
  return WIFEXITED(status) ? WEXITSTATUS(status) : -WTERMSIG(status);
}
```

- Instead of calling a subroutine to perform some task and waiting for it to complete, **`mysystem`** spawns a **child process** to perform some task and waits for it to complete.
- We don't bother checking the return value of **`execvp`**, because we know that if it returns at all, it returns a -1. If that happens, we need to handle the error and make sure the child process terminates, via an exposed **`exit(0)`** call.
- Why not call **`execvp`** inside parent and forgo the child process altogether? Because **`execvp`** would consume the calling process, and that's not what we want.

Lecture 05: New system call: `execvp`

- Here's a test harness that we can run to confirm our **`mysystem`** implementation is working as expected:

```c
static const size_t kMaxLine = 2048;
int main(int argc, char *argv[]) {
  char command[kMaxLine];
  while (true) {
    printf("> ");
    fgets(command, kMaxLine, stdin);
    if (feof(stdin)) break;
    command[strlen(command) - 1] = '\0'; // overwrite '\n'
    printf("retcode = %d\n", mysystem(command));
  }

  printf("\n");
  return 0;
}
```

- **`fgets`** is a somewhat overflow-safe variant on **`scanf`** that knows to read everything up through and including the newline character.
  - The newline character is retained, so we need to chomp that newline off before calling **`mysystem`**.