2 Processes

Who needs process isolation?

Intel Marketing on Meltdown and Spectre

To understand process you need to understand the boot order. In the beginning, there is a kernel. The operating system kernel is a special piece of software. This is the piece of software that is loaded up before all of your other programs even consider getting booted up. What the kernel does is the following, abbreviated

1. The operating system executes ROM or read only code
2. The operating system then executes a `boot_loader` or `EFI` extensions nowadays
3. The boot_loader loads your kernels
4. Your kernel executes `init` to bootstrap itself from nothing
5. The kernel executes startup scripts
6. The kernel executes userland scripts, and you get to use your computer!

You don't need to know the specifics of the booting process, but there it is. When you are executing in user space the kernel provides some important operations that programs don't have to worry about.

- Scheduling Processes and threads
- Handling synchronization primitives
- Providing System Calls like `write` or `read`
- Manages virtual memory and low level binary devices like `usb` drivers
- Handles reading and understanding a filesystem
- Handles communicating over networks
- Handles communications with other processes
- Dynamically linking libraries

The kernel handles all of this stuff in kernel mode. Kernel mode gets you greater power, like executing extra CPU instructions but at the cost of one failure crashes your entire computer – ouch. That is what you are going to interacting with in this class.

More relevantly, the kernel creates the first process `init.d`. Init.d boots up things like your GUI, terminals etc. What is important is the operating system only really creates one process by default, all other processes are `fork` and `exec`'ed from that single process.

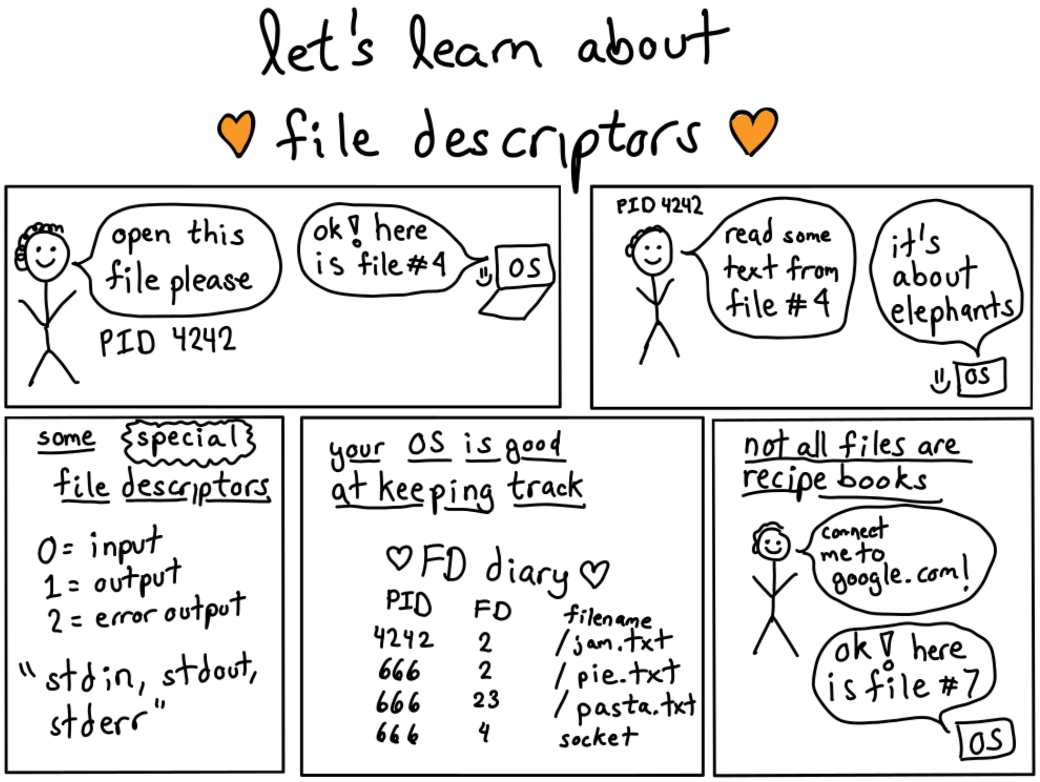

Figure 2.1: File Descriptors

File Descriptors

Although it was mentioned in the last chapter, we are going to give you a quick reminder about file descriptors. Here is a zine from Julia Evans that details it a bit [8].

As the little zine shows, the Kernel keeps track of the file descriptors and what they point to. We will see later that file descriptors need not point to actual files and the OS keeps track of them for you. Also, notice that between processes file descriptors may be reused but inside of a process they are unique. File descriptors may have a notion of position. You can read a file on disk completely because the OS keeps track of the position in the file, and that belongs to your process as well. Other file descriptors point to network sockets and various other pieces of information.

Processes

A process an instance of a computer program that may be running. Processes have a lot of things at their disposal. At the start of each program you get one process, but each program can make more processes. A program consists of the following.

- A binary format: This tells the operating system which set of bits in the binary are what – which part is executable, which parts are constants, which libraries to include etc.
- A set of machine instructions
- A number denoting which instruction to start from
- Constants
- Libraries to link and where to fill in the address of those libraries

Processes are very powerful but they are isolated! That means that by default, no process can communicate with another process. This is very important because if you have a large system (let's say the University of Illinois Engineering Workstations) then you want some processes to have higher privileges than your average user, and one certainly doesn't want the average user to be able to bring down the entire system either on purpose or accidentally by modifying a process.

As most of you have realized by now if you stuck the following code snippet into a program, the variables would not be shared between two parallel invocations of the program.

```
int secrets;
secrets++;
printf("%d\n", secrets);
```

On two different terminals, as you would guess they would both print out 1 not 2. Even if we changed the code to do something really hacky, there would be no way to change another process' state unintentionally. There are ways of changing it purposefully though.

Process Contents

Memory Layout

When a process starts, it gets its own address space. Each process gets the following.

- **A Stack**. The stack is the place where automatic variable and function call return addresses are stored. Every time a new variable is declared, the program moves the stack pointer down to reserve space for the variable. This segment of the stack is Writable but not executable. If the stack grows too far – meaning that it either grows beyond a preset boundary or intersects the heap – you will get a stackoverflow most likely resulting in a SEGFAULT. **The stack is statically allocated by default meaning that there is only a certain amount of space to which one can write**

- **A Heap**. The heap is a contiguous, expanding region of memory [5]. If you want to allocate a large object, it goes here. The heap starts at the top of the text segment and grows upward, meaning sometimes when you call `malloc` that it asks the operating system to push the heap boundary upward. More on that in the memory allocation chapter. This area is also writable but not executable. One can run out of heap memory if the system is constrained or if you run out of addresses. This is more common on a 32bit system.

- **A Data Segment** or as we will tried to refer to it as the Initialized Segement. This contains all of your globals and any other static duration variables. This section starts at the end of the text segment and is static in size because the amount of globals is known at compile time. This section is be writable [10, P. 124]. Most notably, this section contains variables that were initialized with a static initializer like

  ```
  int global = 1;
  ```

  But not those that are default initialized, that is the next segment.

- **A BSS Segment** or the Basic Service Segment. This contains all of your globals and any other static duration variables that are implicitly zeroed out like.

  ```
  int assumed_to_be_zero;
  ```

  It is not an error to assume that this will be zero because otherwise we'd have a security risk from other processes. They just get put in a different section to speed up process start up time. This section starts at the end of the data segment and is also static in size because the amount of globals is known at compile time. At this point the BSS and data segment are combined and confusingly referred to as the data segment [10, P. 124]. From now on for convenience, we will adopt that the data segment is both the data segment

- **A Text Segment**. This is where all your code is stored – all the 1's and 0's. The program counter moves through this segment executing instructions and moving down the next instruction. It is important to note that this is the only executable section of the code created by default. If you try to change the code while it's running, most likely you will segfaults. There are ways around it, but we won't be exploring those in this course. Why doesn't it start at zero? Because of Address Space Layout Randomization. It is outside the scope of this class but know that it exists. The address can be fixed if compiled with the DEBUG flag though.

Other Contents

To keep track of all these processes, your operating system gives each process a number and that process is called the PID, process ID. Processes also have a `ppid` which is short for parent process id. Every process has a parent, that parent could be `init.d`.

Processes could also contain

- Running State - Whether a process is getting ready, running, stopped, terminated etc. (more on this during the Scheduling chapter).
- File Descriptors - List of mappings from integers to real devices (files, usb sticks, sockets)
- Permissions - What `user` the file is running on and what `group` the process belongs to. The process can then only do this admissible to the `user` or `group` like opening a file that the `user` has made exclusives. There are tricks to make a program not be the user who started the program i.e. `sudo` takes a program that a `user` starts and executes it as `root`.
- Arguments - a list of strings that tell your program what parameters to run under
- Environment List - a list of strings in the form `NAME=VALUE` that one can modify.

If we want to give you the technical definition, all POSIX says a process needs is a thread and address space, but most kernel developers and users know that really isn't enough [6].

Intro to Fork

A word of warning

Process forking is a powerful and dangerous tool. If you mess up and cause a fork bomb, **you can bring down the entire system**. To reduce the chances of this, limit your maximum number of processes to a small number e.g 40 by typing `ulimit -u 40` into a command line. Note, this limit is only for the user, which means if you fork bomb, then you won't be able to kill all of the processes you just created since calling `killall` requires your shell to `fork()` ... ironic right? One solution is to spawn another shell instance as another user (for example root) before hand and kill processes from there. Another is to use the built in `exec` command to kill all the user processes (careful you only have one shot at this). Finally you could reboot the system, but you only have one shot at this with the exec function. When testing fork() code, ensure that you have either root and/or physical access to the machine involved. If you must work on fork() code remotely, remember that **kill -9 -1** will save you in the event of an emergency.

TL;DR: Fork can be **extremely** dangerous if you aren't prepared for it. **You have been warned.**

Fork Functionality

The `fork` system call clones the current process to create a new process. It creates a new process called the child process by duplicating the state of the existing process with a few minor differences. The child process does not start from main. Instead it executes the next line after the `fork()` just as the parent process does. Just as a side remark, in older UNIX systems, the entire address space of the parent process was directly copied regardless of whether the resource was modified or not. These days, kernel performs copy-on-write, which saves a lot of resources, while being very time efficient [7, Copy-on-write section]. Here's a very simple example

```
printf("I'm printed once!\n");
fork();
// Now there are two processes running if fork succeeded
// and each process will print out the next line.
printf("You see this line twice!\n");
```

Here is a simple example of this address space cloning. The following program may print out 42 twice - but the `fork()` is after the `printf`!? Why?

```
#include <unistd.h> /*fork declared here*/
#include <stdio.h> /* printf declared here*/
int main() {
  int answer = 84 >> 1;
  printf("Answer: %d", answer);
  fork();
  return 0;
}
```

The `printf` line *is* executed only once however notice that the printed contents is not flushed to standard out. There's no newline printed, we didn't call `fflush`, or change the buffering mode. The output text is therefore still in process memory waiting to be sent. When `fork()` is executed the entire process memory is duplicated including the buffer. Thus the child process starts with a non-empty output buffer which will be flushed when the program exits. We also say may because the contents may be unwritten given a bad program exit as well.

To write code that is different for the parent and child process, check the return value of `fork()`. If `fork()` returns -1, that implies something went wrong in the process of creating a new child. One should check the value stored in *errno* to determine what kind of error occurred. Commons one include `EAGAIN` and `ENOMEM` Which are essentially try again and out of memory. Similarly, a return value of 0 indicates that we are in the child process, while a positive integer shows that we are in parent process. The positive value returned by `fork()` gives as the process id (*pid*) of the child.

Another way to remember which is which is that the child process can find its parent - the original process that was duplicated - by calling `getppid()` - so does not need any additional return information from `fork()`. In the POSIX word, you only have one parent process. The parent process however can only find out the id of the new child process from the return value of `fork`:

```
pid_t id = fork();
if (id == -1) exit(1); // fork failed
if (id > 0) {
  // I'm the original parent and
  // I just created a child process with id 'id'
  // Use waitpid to wait for the child to finish
} else {// returned zero
  // I must be the newly made child process
}
```

A slightly silly example is shown below. What will it print? Try it with multiple arguments to your program.

```
#include <unistd.h>
#include <stdio.h>
int main(int argc, char **argv) {
  pid_t id;
  int status;
  while (--argc && (id=fork())) {
    waitpid(id,&status,0); /* Wait for child*/
  }
  printf("%d:%s\n", argc, argv[argc]);
  return 0;
}
```

Another example is below. This is the amazing parallel apparent-O(N) *sleepsort* is today's silly winner. First published on 4chan in 2011. A version of this awful but amusing sorting algorithm is shown below.

```
int main(int c, char **v) {
  while (--c > 1 && !fork());
  int val = atoi(v[c]);
  sleep(val);
  printf("%d\n", val);
  return 0;
}
```

The algorithm isn't actually O(N) because of how the system scheduler works.

What is a fork bomb?

A fork bomb is what we warned you about earlier. A 'fork bomb' is when you attempt to create an infinite number of processes. This will often bring a system to a near-standstill as it attempts to allocate CPU time and memory to a very large number of processes that are ready to run. System administrators don't like fork-bombs and may set upper limits on the number of processes each user can have or may revoke login rights because it creates a disturbance in the force for other users' programs. You can also limit the number of child processes created by using `setrlimit()`. fork bombs are not necessarily malicious - they occasionally occur due to coding errors. Below is a simple example that is malicious.

```
while (1) fork();
```

There may even be subtle forkbombs that occur when you are being careless while coding. Can you spot the fork bomb here?

```
#include <unistd.h>
#define HELLO_NUMBER 10

int main(){
  pid_t children[HELLO_NUMBER];
  int i;
  for(i = 0; i < HELLO_NUMBER; i++){
    pid_t child = fork();
    if(child == -1){
      break;
    }
    if(child == 0){ //I am the child
      execlp("ehco", "echo", "hello", NULL);
    }
    else{
      children[i] = child;
    }
  }

  int j;
  for(j = 0; j < i; j++){
    waitpid(children[j], NULL, 0);
  }
  return 0;
}
```

We misspelled `ehco`, so we can't `exec` it. What does this mean? Instead of creating 10 processes we just created **1024 processes, fork bombing our machine. How could we prevent this? Put an exit right after exec so in case exec fails we won't end up fork bombing our machine.** There are various other ways. What if we removed the `echo` binary? What if the binary itself creates a forkbomb.

Signals

We won't fully get into signals until the very end of the course, but it is relevant to bring it up now because various semantics with fork and a few other function calls detail what a signal is.

A signal can be thought of as a software interrupt. This means that a process that receives a signal stops execution of the current program and makes the program respond to the signal.

There are various signals defined by the operating system, two of which you may already know: SIGSEGV and SIGINT. One is caused by an illegal memory access and one is sent by a user wanting to terminate a program. Each case, the program jumps from what it is doing to the signal handler. If no signal handler is supplied by the program a default handler is executed – for example terminating, ignoring.

Here is an example of a simple user defined signal handler.

```c
void handler(int signum) {
  write(1, "signaled!", 9);
  // we don't need the signum because we are only catching SIGINT
  // if you want to use the same piece of code for multiple
  // signals, check the signum
}
int main() {
  signal(SIGINT, handler);
  while(1) ;
  return 0;
}
```

A signal can be in the generated, sent, pending, and received state. These refer to when a process generates, the kernel sends, the kernel is about to deliver, and when the kernel delivers a signal which all take a bit of time to do. The terminology is important because fork and exec require different operations based on what state the signal is in.

Just to note, It is generally poor programming practice to use signals in program logic. Meaning send a signal to perform a certain operation. Signals have no timeframe of delivery and no assurance that they will be delivered. If you need to communicate between two processes, there are other ways of doing so.

If you want to read more, feel free to skip ahead to that particular chapter and read it over. It isn't very long and gives you the long and short about how to deal with signals in processes.

POSIX Fork Detailings

POSIX determines what the standards of fork are [4]. You can read the previous citation, but it can be quite verbose. Here are a summary of what is relevant.

1. Fork will return a non-negative integer on success
2. A child will inherit any open file descriptors of the parent. That means if a parent reads a half way of the file and forks, the child will start at that offset. Any other flags are also carried over.
3. Pending signals are not inherited. This means that if a parent has a pending signal and creates a child, the child will not receive that signal unless another process signals the child.
4. The process will be created with one thread (more on that later, the general consensus is to not fork and pthread at the same time).
5. Since we have copy on write, read-only memory addresses are shared between processes

6. If you set up certain regions of memory, they are shared between processes.
7. Signal handlers are inherited but can be changed.
8. Current working directory is inherited but can be changed.
9. Environment variables are inherited but can be changed.

Key differences between the parent and the child include:

- The process id returned by `getpid()`. The parent process id returned by `getppid()`.
- The parent is notified via a signal, SIGCHLD, when the child process finishes but not vice versa.
- The child does not inherit pending signals or timer alarms. For a complete list see the fork man page
- The child has its own set of environment variables

Waiting and Execing

If the parent process wants waits for the child to finish, `waitpid` (or `wait`).

```
pid_t child_id = fork();
if (child_id == -1) {perror("fork"); exit(EXIT_FAILURE);}
if (child_id > 0) {
  // We have a child! Get their exit code
  int status;
  waitpid( child_id, &status, 0 );
  // code not shown to get exit status from child
} else {// In child ...
  // start calculation
  exit(123);
}
```

`wait` is a simpler version of `waitpid`. `wait` accepts a pointer to an integer and waits on any child process. After the first one changes state `wait` returns. `waitpid` is similar to `wait` but it has a few differences. First, you *can* wait on a specific process, or you can pass in special values for the `pid` to do different things (check the man pages). The last parameter to waitpid is an option parameter. The options are listed below

WNOHANG - Return whether or not the searched process is exited

WNOWAIT - Wait, but leave the child waitable by another wait call

WEXITED - Wait for exited children

WSTOPPED - Wait for stopped children

WCONTINUED - Wait for continued children

Exit statuses or the value stored in the integer pointer for both of the calls above are explained below.

Exit statuses

To find the return value of `main()` or value included in `exit()`), Use the `Wait macros` - typically you will use `WIFEXITED` and `WEXITSTATUS` . See `wait/waitpid` man page for more information.

```
int status;
pid_t child = fork();
if (child == -1) return 1; //Failed
if (child > 0) {/* I am the parent - wait for the child to finish */
  pid_t pid = waitpid(child, &status, 0);
  if (pid != -1 && WIFEXITED(status)) {
    int low8bits = WEXITSTATUS(status);
    printf("Process %d returned %d" , pid, low8bits);
  }
} else {/* I am the child */
  // do something interesting
  execl("/bin/ls", "/bin/ls", ".", (char *) NULL); // "ls ."
}
```

A process can only have 256 return values, the rest of the bits are informational, and the information is extracte with bit shifting. But, The kernel has an internal way of keeping track of signaled, exited, or stopped. That API is abstracted so that that the kernel developers are free to change at will. Remember that these macros only make sense if the precondition is met. Meaning that a process' exit status won't be defined if the process isn't signaled. The macros will not do the checking for you, so it's up to the programmer to make sure the logic checks out. As an example above, you should use the `WIFSTOPPED` to check if a process was stopped and then the `WSTOPSIG` to find the signal that stopped it. As such there is no need to memorize the following, this is just a high level overview of how information is stored inside the status variables. From `sys/wait.h` of an old Berkeley kernel[1]:

```
/* If WIFEXITED(STATUS), the low-order 8 bits of the status. */
#define _WSTATUS(x) (_W_INT(x) & 0177)
#define _WSTOPPED 0177 /* _WSTATUS if process is stopped */
#define WIFSTOPPED(x) (_WSTATUS(x) == _WSTOPPED)
#define WSTOPSIG(x) (_W_INT(x) >> 8)
#define WIFSIGNALED(x) (_WSTATUS(x) != _WSTOPPED && _WSTATUS(x) != 0)
#define WTERMSIG(x) (_WSTATUS(x))
#define WIFEXITED(x) (_WSTATUS(x) == 0)
```

There is a convention about exit codes. If the process exited normally and everything was successful, then a zero should be returned. Beyond that, there isn't too many conventions except the ones that you place on yourself. If you know how the program you spawn is going to interact, you may be able to make more sense of the 256 error codes. You could in fact write your program to return 1 if the program went to stage 1 (like writing to a file) 2 if it did something else etc... But none of the unix programs are designed to follow that for simplicity sake.

Zombies and Orphans

It is good practice to wait on your children! If you don't wait on your children they become zombies. Zombies occur when a child terminates and then take up a spot in the kernel process table for your process. The process table keeps information about that process like pid, status, how it was killed. The only way to get rid of a zombie is to wait on your children. If you never wait on your children, and the program is long running, you may lose the ability to fork.

You don't always need to wait for your children! Your parent process can continue to execute code without having to wait for the child process. If a parent dies without waiting on its children, a process can orphan its children. Once a parent process completes, any of its children will be assigned to `init` - the first process with pid of 1. Thus these children would see `getppid()` return a value of 1. These orphans will eventually finish and for a brief moment become a zombie. The init process automatically waits for all of its children, thus removing these zombies from the system.

Extra: How can I asynchronously wait for my child using SIGCHLD?

Warning: This section uses signals which we have not yet fully introduced. The parent gets the signal SIGCHLD when a child completes, so the signal handler can wait on the process. A slightly simplified version is shown below.

```
pid_t child;

void cleanup(int signal) {
  int status;
  waitpid(child, &status, 0);
  write(1,"cleanup!\n",9);
}
int main() {
  // Register signal handler BEFORE the child can finish
  signal(SIGCHLD, cleanup); // or better - sigaction
  child = fork();
  if (child == -1) {exit(EXIT_FAILURE);}

  if (child == 0) {/* I am the child!*/
    // Do background stuff e.g. call exec
  } else {/* I'm the parent! */
    sleep(4); // so we can see the cleanup
    puts("Parent is done");
  }
  return 0;
}
```

The above example however misses a couple of subtle points.

1. More than one child may have finished but the parent will only get one SIGCHLD signal (signals are not queued)
2. SIGCHLD signals can be sent for other reasons (e.g. a child process is temporarily stopped)
3. It uses the deprecated `signal` code

A more robust code to reap zombies is shown below.

```
void cleanup(int signal) {
  int status;
  while (waitpid((pid_t) (-1), 0, WNOHANG) > 0) {

  }
}
```

exec

To make the child process execute another program, use one of the `exec` functions after forking. The `exec` set of functions replaces the process image with the the process image of what is being called. This means that any lines of code after the `exec` call are replaced. Any other work you want the child process to do should be done before the `exec` call. The naming schemes can be shortened mnemonically.

1. e – An array of pointers to environment variables is explicitly passed to the new process image.
2. l – Command-line arguments are passed individually (a list) to the function.
3. p – Uses the PATH environment variable to find the file named in the file argument to be executed.
4. v – Command-line arguments are passed to the function as an array (vector) of pointers.

An example of this code is below. This code executes `ls`

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

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

int main(int argc, char**argv) {
  pid_t child = fork();
  if (child == -1) return EXIT_FAILURE;
  if (child) {/* I have a child! */
    int status;
    waitpid(child , &status ,0);
    return EXIT_SUCCESS;

  } else {/* I am the child */
    // Other versions of exec pass in arguments as arrays
    // Remember first arg is the program name
    // Last arg must be a char pointer to NULL

    execl("/bin/ls", "/bin/ls", "-alh", (char *) NULL);

    // If we get to this line, something went wrong!
    perror("exec failed!");
  }
}
```

Try to decode the following example

```
#include <unistd.h>
#include <fcntl.h> // O_CREAT, O_APPEND etc. defined here

int main() {
  close(1); // close standard out
  open("log.txt", O_RDWR | O_CREAT | O_APPEND, S_IRUSR | S_IWUSR);
  puts("Captain's log");
  chdir("/usr/include");
  // execl( executable, arguments for executable including program name
     and NULL at the end)

  execl("/bin/ls", /* Remaining items sent to ls*/ "/bin/ls", ".", (char
     *) NULL); // "ls ."
  perror("exec failed");
  return 0;
}
```

The example writes "Captain's Log" to a file then prints everything in /usr/include to the same file. There's no error checking in the above code (we assume close,open,chdir etc works as expected).

1. `open` – will use the lowest available file descriptor (i.e. 1) ; so standard out now goes to the log file.
2. `chdir` – Change the current directory to /usr/include
3. `execl` – Replace the program image with /bin/ls and call its main() method
4. `perror` – We don't expect to get here - if we did then exec failed.
5. We need the return zero because compilers complain if we don't have it.

POSIX Exec Detailings

POSIX details all of the semantics that exec needs to cover [3]. What you need to know is the following bullet points.

1. File descriptors are preserved after an exec. That means if you open a file, and you forget to close it, it remains open in the child. This is a problem because usually the child doesn't know about those file descriptors and they take up a slot in the file descriptor table and could possible prevent other processes from accessing the file. The one exception to this is if the file descriptor has the Close Exec flag set (more on how to set that later).
2. Various signal semantics. The exec'ed processes preserve the signal mask and the pending signal set, but does not preserve the signal handlers because it is a different program.
3. Environment variables are preserved unless using an environ version of exec
4. The operating system may open up 0, 1, 2 – stdin, stdout, stderr, if they are closed after exec, most of the time they leave them closed.
5. The exec'ed process runs as the same PID and has the same parent and process group as the previous process.
6. The exec'ed process is run on the same user and group with the same working directory

Shortcuts

`system` pre-packs the above code [9, P. 371]. Here is how to use it:

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

int main(int argc, char**argv) {
  system("ls"); // execl("/bin/sh", "/bin/sh", "-c", "\\"ls\\"")
  return 0;
}
```

The `system` call will fork, execute the command passed by parameter and the original parent process will wait for this to finish. This also means that `system` is a blocking call. The parent process can't continue until the process started by `system` exits. Also, `system` actually creates a shell which is then given the string, which is more overhead than just using `exec` directly. The standard shell will use the `PATH` environment variable to search for a filename that matches the command. Using system will usually be sufficient for many simple run-this-command problems but can quickly become limiting for more complex or subtle problems, and it hides the mechanics of the fork-exec-wait pattern so we encourage you to learn and use `fork` `exec` and `waitpid` instead. Not only that, it tends to be a huge security risk. By allowing someone to access a shell version of the environment, you can reach all sorts of problems

```c
int main(int argc, char**argv) {
  char *to_exec = asprintf("ls %s", argv[1]);
  system(to_exec);
}
```

Passing something along the lines of argv[1] = "; sudo su" is a huge security risk.

The fork-exec-wait Pattern

A common programming pattern is to call `fork` followed by `exec` and `wait`. The original process calls fork, which creates a child process. The child process then uses exec to start execution of a new program. Meanwhile the parent uses `wait` (or `waitpid`) to wait for the child process to finish.

```c
#include <unistd.h>

int main() {
  pid_t pid = fork();
  if (pid < 0) {// fork failure
    exit(1);
  } else if (pid > 0) {// I am the parent
    int status;
    waitpid(pid, &status, 0);
  } else {// I am the child
    execl("/bin/ls", "/bin/ls", NULL);
    exit(1); // For safety.
  }
}
```

You may ask why we didn't just execute ls directly. The reason is that now we have a monitor program aka our parent that can do other things. It can proceed afterwards and execute another function, or it can also modify the state of the system or read the output of the functino call.

Environment Variables

Environment variables are variables that the system keeps for all processes to use. Your system has these set up right now! In Bash, you can check some of these

```
$ echo $HOME
/home/bhuvy
$ echo $PATH
/usr/local/sbin:/usr/bin:...
```

How would you get and set these in C/C++? You can use the `getenv` and `setenv` function respectively.

```c
char* home = getenv("HOME"); // Will return /home/bhuvy
setenv("HOME", "/home/bhuvan", 1 /*set overwrite to true*/ );
```

Environment variables are important because they are inherited between processes and can be used the specify a standard set of behaviors [2], though you don't need to memorize the options. Another security related concern is that environment variables cannot be read by an outside process whereas argv can be.

Further Reading

Read the man pages and the posix groups above!

- fork
- exec
- wait

Topics

- Correct use of fork, exec and waitpid
- Using exec with a path
- Understanding what fork and exec and waitpid do. E.g. how to use their return values.
- SIGKILL vs SIGSTOP vs SIGINT.

- What signal is sent when you press CTRL-C
- Using kill from the shell or the kill POSIX call.
- Process memory isolation.
- Process memory layout (where is the heap, stack etc; invalid memory addresses).
- What is a fork bomb, zombie and orphan? How to create/remove them.
- getpid vs getppid
- How to use the WAIT exit status macros WIFEXITED etc.

Questions/Exercises

- What is the difference between execs with a p and without a p? What does the operating system
- How do you pass in command line arguments to `execl*`? How about `execv*`? What should be the first command line argument by convention?
- How do you know if `exec` or `fork` failed?
- What is the `int *status` pointer passed into wait? When does wait fail?
- What are some differences between `SIGKILL`, `SIGSTOP`, `SIGCONT`, `SIGINT`? What are the default behaviors? Which ones can you set up a signal handler for?
- What signal is sent when you press `CTRL-C`?
- My terminal is anchored to PID = 1337 and has just become unresponsive. Write me the terminal command and the C code to send `SIGQUIT` to it.
- Can one process alter another processes memory through normal means? Why?
- Where is the heap, stack, data, and text segment? Which segments can you write to? What are invalid memory addresses?
- Code me up a fork bomb in C (please don't run it).
- What is an orphan? How does it become a zombie? How do I be a good parent?
- Don't you hate it when your parents tell you that you can't do something? Write me a program that sends `SIGSTOP` to your parent.
- Write a function that fork exec waits an executable, and using the wait macros tells me if the process exited normally or if it was signaled. If the process exited normally, then print that with the return value. If not, then print the signal number that caused the process to terminate.

Bibliography

[1] Source to sys/wait.h. URL `http://unix.superglobalmegacorp.com/Net2/newsrc/sys/wait.h.html`.

[2] Environment variables, Jul 2018. URL `https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap08.html`.

[3] exec, Jul 2018. URL `https://pubs.opengroup.org/onlinepubs/9699919799/functions/exec.html`.

[4] fork, Jul 2018. URL `https://pubs.opengroup.org/onlinepubs/9699919799/functions/fork.html`.

[5] Overview of malloc, Mar 2018. URL `https://sourceware.org/glibc/wiki/MallocInternals`.

[6] Definitions, Jul 2018. URL `http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap03.html#tag_03_210`.

[7] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005. ISBN 0596005652.

[8] Julia Evans. File descriptors, Apr 2018. URL `https://drawings.jvns.ca/file-descriptors/`.

[9] Larry Jones. Wg14 n1539 committee draft iso/iec 9899: 201x, 2010.

[10] Peter Van der Linden. *Expert C programming: deep C secrets*. Prentice Hall Professional, 1994.