4 Processes

Who needs process isolation?

Intel Marketing on Meltdown and Spectre

To understand what a process is, you need to understand what an operating system is. An operating system is a program that provides an interface between hardware and user software as well as providing a set of tools that the software can use. The operating system manages hardware and gives user programs a uniform way of interacting with hardware as long as the operating system can be installed on that hardware. Although this idea sounds like it is the end-all, we know that there are many different operating systems with their own quirks and standards. As a solution to that, there is another layer of abstraction: POSIX or portable operating systems interface. This is a standard (or many standards now) that an operating system must implement to be POSIX compatible – most systems that we'll be studying are almost POSIX compatible due more to political reasons.

Before we talk about POSIX systems, we should understand what the idea of a kernel is generally. In an operating system (OS), there are two spaces: kernel space and user space. Kernel space is a power operating mode that allows the system to interact with the hardware and has the potential to destroy your machine. User space is where most applications run because they don't need this level of power for every operation. When a user space program needs additional power, it interacts with the hardware through a system call that is conducted by the kernel. This adds a layer of security so that normal user programs can't destroy your entire operating system. For the purposes of our class, we'll talk about single machine multiple user operating systems. This is where there is a central clock on a standard laptop or desktop. Other OSes relax the central clock requirement (distributed) or the "standardness" of the hardware (embedded systems). Other invariants make sure events happen at particular times too.

The operating system is made up of many different pieces. There may be a program running to handle incoming USB connections, another one to stay connected to the network, etc. The most important one is the kernel – although it might be a set of processes – which is the heart of the operating system. The kernel has many important tasks. The first of which is booting.

1. The computer hardware executes code from read-only memory, called firmware.
2. The firmware executes a bootloader, which often conforms to the Extensible Firmware Interface (`EFI`), which is an interface between the system firmware and the operating system.
3. The bootloader's boot manager loads the operating system kernels, based on the boot settings.
4. Your kernel executes `init` to bootstrap itself from nothing.
5. The kernel executes startup scripts like starting networking and USB handling.

6. The kernel executes userland scripts like starting a desktop, and you get to use your computer!

When a program is executing in user space, the kernel provides some important services to programs in User space.

- Scheduling processes and threads
- Handling synchronization primitives (futexes, mutexes, semaphores, etc.)
- Providing system calls such as `write` or `read`
- Managing virtual memory and low-level binary devices such as USB drivers
- Managing filesystems
- Handling communication over networks
- Handling communication between processes
- Dynamically linking libraries
- The list goes on and on.

The kernel creates the first process `init.d` (an alternative is system.d). *init.d* boots up programs such as graphical user interfaces, terminals, etc – by default, this is the only process explicitly created by the system. All other processes are instantiated by using the system calls `fork` and `exec` from that single process.

File Descriptors

Although these were mentioned in the last chapter, we are going to give a quick reminder about file descriptors. A zine from Julia Evans gives some more details [8].

The kernel keeps track of the file descriptors and what they point to. Later we will learn two things: that file descriptors point to more than files and that the operating system keeps track of them.

Notice that file descriptors may be reused between processes, but inside of a process, they are unique. File descriptors may have a notion of position. These are known as seekable streams. A program can read a file on disk completely because the OS keeps track of the position in the file, an attribute that belongs to your process as well.

Other file descriptors point to network sockets and various other pieces of information, that are unseekable streams.

Processes

A process is an instance of a computer program that may be running. Processes have many resources at their disposal. At the start of each program, a program gets one process, but each program can make more processes. A program consists of the following:

- A binary format: This tells the operating system about the various sections of bits in the binary – which parts are 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 powerful, but they are isolated!
That means that by default, no process can communicate with another process.

This is important because in complex systems (like the University of Illinois Engineering Workstations), it is likely that different processes will have different privileges. One certainly doesn't want the average user to be able to bring down the entire system, by either purposely or accidentally modifying a process. As most of you have realized by now, if you stuck the following code snippet into a program, the variables are unshared between two parallel invocations of the program.

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

On two different terminals, they would both print out 1 not 2. Even if we changed the code to attempt to affect other process instances, there would be no way to change another process' state unintentionally. However, there are other intentional ways to change the program states of other processes.

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 automatically allocated variables 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. This behavior is controlled by the no-execute (NX) bit, sometimes called the WX̂ (write XOR execute) bit, which helps prevent malicious code, such as `shellcode` from being run on the stack.

  If the stack grows too far – meaning that it either grows beyond a preset boundary or intersects the heap – the program will stack overflow error, most likely resulting in a SEGFAULT. **The stack is statically allocated by default; 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 a program wants to allocate an object whose lifetime is manually controlled or whose size cannot be determined at compile-time, it would want to create a heap variable.

  The heap starts at the top of the text segment and grows upward, meaning `malloc` may push the heap boundary – called the program break – upward.

  We will explore this in more depth in our chapter on memory allocation. This area is also writable but not executable. One can run out of heap memory if the system is constrained or if a program run out of addresses, a phenomenon that is more common on a 32-bit system.

- **A Data Segment**

  This segment contains two parts, an initialized data segment, and an uninitialized segment. Furthermore, the initialized data segment is divided into a readable and writable section.

  – **Initialized Data Segment** This contains all of a program's globals and any other static variables.

  This section starts at the end of the text segment and starts at a constant size because the number of globals is known at compile time. The end of the data segment is called the `program break` and can be extended via the use of brk / sbrk.

  This section is writable [10, P. 124]. Most notably, this section contains variables that were initialized with a static initializer, as follows:

```
int global = 1;
```

  – **Uninitialized Data Segment / BSS** BSS stands for an old assembler operator known as Block Started by Symbol.

  This contains all of your globals and any other static duration variables that are implicitly zeroed out.

  Example:

```
int assumed_to_be_zero;
```

  This variable will be zeroed; otherwise, we would have a security risk involving isolation from other processes.

  They 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.

  Currently, both the initialized and BSS data segments are combined and referred to as the data segment [10, P. 124], despite being somewhat different in purpose.

  – **A Text Segment**. This is where all executable instructions are stored, and is readable (function pointers) but not writable.

  The program counter moves through this segment executing instructions one after the other.

  It is important to note that this is the only executable section of the program, by default.

  If a program's code while it's running, the program most likely will SEGFAULT.

  There are ways around it, but we will not be exploring these in this course.

  Why doesn't it always start at zero? This is because of a security feature called address space layout randomization.

  The reasons for and explanation about this is outside the scope of this class, but it is good to know about its existence.

  Having said that, this address can be made constant, if a program is compiled with the DEBUG flag.

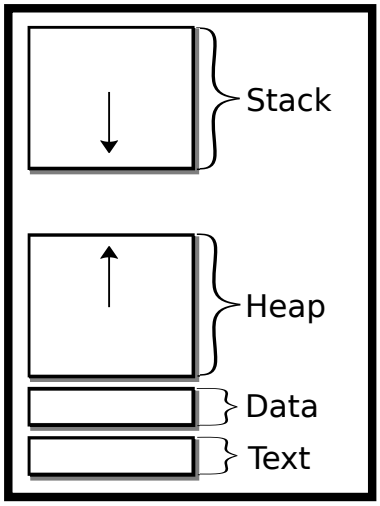

Figure 4.1: Process address space

Other Contents

To keep track of all these processes, your operating system gives each process a number called the process ID (PID). Processes are also given the PID of their parent process, called parent process ID (`PPID`). Every process has a parent, that parent could be `init.d`.

Processes could also contain the following information:

- Running State - Whether a process is getting ready, running, stopped, terminated, etc. (more on this is covered in the chapter on Scheduling).
- File Descriptors - A list of mappings from integers to real devices (files, USB flash drives, sockets)
- Permissions - What `user` the file is running on and what `group` the process belongs to. The process can then only perform operations based on the permissions given to the `user` or `group`, such as accessing files. There are tricks to make a program take a different user than who started the program i.e. `sudo` takes a program that a `user` starts and executes it as `root`. More specifically, a process has a real user ID (identifies the owner of the process), an effective user ID (used for non-privileged users trying to access files only accessible by superusers), and a saved user ID (used when privileged users perform non-privileged actions).
- Arguments - a list of strings that tell your program what parameters to run under.
- Environment Variables - a list of key-value pair strings in the form `NAME=VALUE` that one can modify. These are often used to specify paths to libraries and binaries, program configuration settings, etc.

According to the POSIX specification, a process only needs a thread and address space, but most kernel developers and users know that only these aren't enough [6].

Intro to Fork

A word of warning

Process forking is a powerful and dangerous tool. If you make a mistake resulting in a fork bomb, **you can bring down an 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 created process since calling `killall` requires your shell to `fork()`. Quite unfortunate. One solution is to spawn another shell instance as another user (for example root) beforehand and kill processes from there.

Another is to use the built-in `exec` command to kill all the user processes (you only have one attempt 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. 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, called a child process. This occurs by duplicating the state of the existing process with a few minor differences.

- The child process executes the next line after the `fork()` 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. The current behavior is for the kernel to perform a copy-on-write, which saves a lot of resources, while being time efficient [7, Copy-on-write section].

Here is a simple example:
m

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

```
#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 are 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 may be flushed when the program exits. We 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. Common errors include `EAGAIN` and `ENOENT` Which are essentially "try again – resource temporarily unavailable", and "no such file or directory".

Similarly, a return value of 0 indicates that we are operating in the context of the child process, whereas a positive integer shows that we are in the context of the parent process.

The positive value returned by `fork()` is the process id (*pid*) of the child.

A way to remember what is represented by the return value of fork 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()`. However, the parent process may have many child processes, and therefore needs to be explicitly informed of its child PIDs.

According to the POSIX standard, every process only has a single parent process.

The parent process can only know the PID of the new child process from the return value of `fork`:

m

```
pid_t id = fork();
if (id == -1) exit(1); // fork failed
if (id > 0) {
  // Original parent
  // A child process with id 'id'
  // Use waitpid to wait for the child to finish
} else {// returned zero
  // Child Process
}
```

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

m

```
#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. This sorting algorithm may fail to produce the correct output.

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

Imagine that we ran this program like so

```
$ ./ssort 1 3 2 4
```

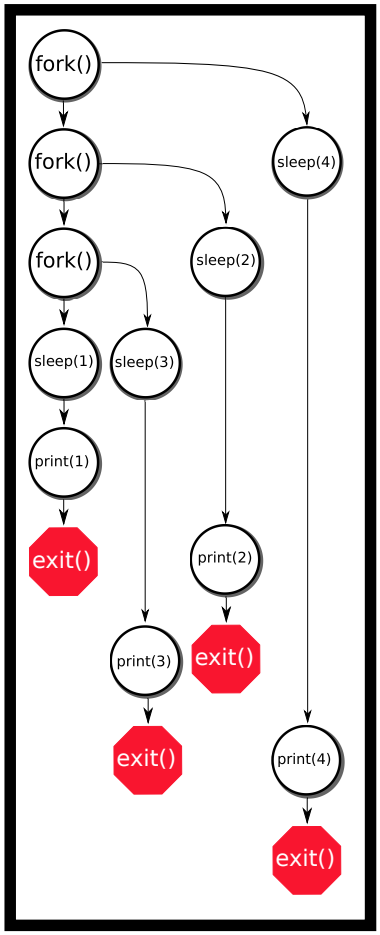

Figure 4.2: Timing of sorting 1, 3, 2, 4

The algorithm isn't actually O(N) because of how the system scheduler works. In essence, this program outsources the actual sorting to the operating system.

Fork Bomb

A 'fork bomb' is what we warned you about earlier. This occurs when there is an 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 large number of processes that are ready to run. System administrators don't like them and may set upper limits on the number of processes each user can have, or revoke login rights because they create disturbances in the Force for other users' programs. A program can limit the number of child processes created by using `setrlimit()`.

Fork bombs are not necessarily malicious - they occasionally occur due to programming errors. Below is a simple example that is malicious.

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

It is easy to cause one, if you are careless while calling fork, especially in a loop. Can you spot the fork bomb here?

```c
#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) {
      // Child
      execlp("ehco", "echo", "hello", NULL);
    }
    else{
      // Parent
      children[i] = child;
    }
  }

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

We misspelled `ehco`, so the `exec` call fails. What does this mean? Instead of creating 10 processes, we created *1024 processes, fork bombing our machine*. **How could we prevent this? Add an exit right after exec, so that if exec fails, we won't end up calling fork an unbounded number of times.** There are various other ways. What if we removed the `echo` binary? What if the binary itself creates a fork bomb?

Signals

We won't fully explore signals until the end of the course, but it is relevant to broach the subject now because various semantics related to fork and 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 the 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. The first is caused by an illegal memory access, and the second is sent by a user wanting to terminate a

program. In each case, the program jumps from the current line being executed to the signal handler. If no signal handler is supplied by the program, a default handler is executed – such as terminating the program, or ignoring the signal.

Here is an example of a simple user-defined signal handler:

```
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 has four stages in its life cycle: generated, pending, blocked, and received state. These refer to when a process generates a signal, the kernel is about to deliver a signal, the signal is blocked, and when the kernel delivers a signal, each of which requires some time to complete. Read more in the introduction to the Signals chapter.

The terminology is important because fork and exec require different operations based on the state a signal is in.

To note, it is generally poor programming practice to use signals in program logic, which is to send a signal to perform a certain operation. The reason: signals have no time frame of delivery and no assurance that they will be delivered. There are better ways to communicate between two processes.

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

POSIX Fork Details

POSIX determines the standards of fork [4]. You can read the previous citation, but do note that it can be quite verbose. Here is 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 half of the file and forks, the child will start at that offset. A read on the child's end will shift the parent's offset by the same amount. 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 create processes and threads at the same time).
5. Since we have copy on write (COW), read-only memory addresses are shared between processes.

6. If a program sets up certain regions of memory, they can be shared between processes.
7. Signal handlers are inherited but can be changed.
8. The process' current working directory (often abbreviated to CWD) 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.

Fork and FILEs

There are some tricky edge cases when it comes to using `FILE` and forking. First, we have to make a technical distinction. A **File Description** is the struct that a file descriptor points to. File descriptors can point to many different structs, but for our purposes, they'll point to a struct that represents a file on a filesystem. This file description contains elements like paths, how far the descriptor has read into the file, etc. A file descriptor points to a file description. This is important because when a process is forked, only the file descriptor is cloned, not the description. The following snippet contains only one description.

```
int file = open(...);
if(!fork) {
  read(file, ...);
} else {
  read(file, ...);
}
```

One process will read one part of the file, the other process will read another part of the file. In the following example, there are two descriptions caused by two different file handles.

```
if(!fork) {
  int file = open(...);
  read(file, ...);
} else {
  int file = open(...);
  read(file, ...);
}
```

Let's consider our motivating example.

```
$ cat test.txt
A
B
C
```

Take a look at this code, what does it do?

```
size_t buffer_cap = 0;
char * buffer = NULL;
ssize_t nread;
FILE * file = fopen("test.txt", "r");
int count = 0;
while((nread = getline(&buffer, &buffer_cap, file) != -1) {
  printf("%s", buffer);
  if(fork() == 0) {
    exit(0);
  }
  wait(NULL);
}
```

The initial thought may be that it prints the file line by line with some extra forking. It is actually undefined behavior because we didn't prepare the file descriptors. To make a long story short, here is what to do to avoid the example.

1. You as the programmer need to make sure that all of your file descriptors are prepared before forking.
2. If it is a file descriptor or an unbuffered `FILE*`, it is already prepared.
3. If the `FILE*` is open for reading and has been read fully, it is already prepared.
4. Otherwise, the `FILE*` **must** be `fflush`'ed or closed to be prepared.
5. If the file descriptor is prepared, it must unactive in the parent process if the child process is using it or vice versa. A process is using it if it is read or written or if that process *for whatever reason* calls `exit`. If a process uses it when the other process is as well, the whole application's behavior is undefined.

So how would we fix the code? We would have to flush the file before forking and refrain from using it until after the `wait` call – more on the specifics of this next section.

```
size_t buffer_cap = 0;
char * buffer = NULL;
ssize_t nread;
FILE * file = fopen("test.txt", "r");
int count = 0;
while((nread = getline(&buffer, &buffer_cap, file) != -1) {
  printf("%s", buffer);
```

```
  fflush(file);
  if(fork() == 0) {
    exit(0);
  }
  wait(NULL);
}
```

What if the parent process and the child process need to perform asynchronously and need to keep the file handle open? Due to event ordering, we need to make sure that parent process knows that the child is finished using `wait`. We'll talk about Inter-Process communication in a later chapter, but now we can use the double fork method.

```
//...
fflush(file);
pid_t child = fork();
if(child == 0) {
  fclose(file);
  if (fork() == 0) {
    // Do asynchronous work
    // Safe exit, this child doesn't know about
    // the file descriptor
    exit(0);
  }
  exit(0);
}
waitpid(child, NULL, 0);
```

Extra: Explanation of the Fork-FILE Problem

To parse the POSIX documentation, we'll have to go deep into the terminology. The sentence that sets the expectation is the following

> The result of function calls involving any one handle (the "active handle") is defined elsewhere in this volume of POSIX.1-2008, but if two or more handles are used, and any one of them is a stream, the application shall ensure that their actions are coordinated as described below. If this is not done, the result is undefined.

What this means is that if we don't follow POSIX to the letter when using two file descriptors that refer to the same description across processes, we get undefined behavior. To be technical, the file descriptor must have a "position" meaning that it needs to have a beginning and an end like a file, not like an arbitrary stream of bytes. POSIX then goes on to introduce the idea of an active handle, where a handle may be a file descriptor or a `FILE*` pointer. File handles don't have a flag called "active". An active file descriptor is one that is currently being used

for reading and writing and other operations (such as `exit`). The standard says that before a `fork` that the *application* or your code must execute a series of steps to prepare the state of the file. In simplified terms, the descriptor needs to be closed, flushed, or read to its entirety – the gory details are explained later.

> For a handle to become the active handle, the application shall ensure that the actions below are performed between the last use of the handle (the current active handle) and the first use of the second handle (the future active handle). The second handle then becomes the active handle. All activity by the application affecting the file offset on the first handle shall be suspended until it again becomes the active file handle. (If a stream function has as an underlying function one that affects the file offset, the stream function shall be considered to affect the file offset.)

Summarizing as if two file descriptors are actively being used, the behavior is undefined. The other note is that after a fork, the library code must prepare the file descriptor as if the other process were to make the file active at any time. The last bullet point concerns itself with how a process prepares a file descriptor in our case.

> If the stream is open with a mode that allows reading and the underlying open file description refers to a device that is capable of seeking, the application shall either perform an fflush(), or the stream shall be closed.

The documentation says that the child needs to perform an fflush or close the stream because the file descriptor needs to be prepared in case the parent process needs to make it active. glibc is in a no-win situation if it closes a file descriptor that the parent may expect to be open, so it'll opt for the fflush on exit because exit in POSIX terminology counts as accessing a file. That means that for our parent process, this clause gets triggered.

> If any previous active handle has been used by a function that explicitly changed the file offset, except as required above for the first handle, the application shall perform an lseek() or fseek() (as appropriate to the type of handle) to an appropriate location.

Since the child calls fflush and the parent didn't prepare, the operating system chooses to where the file gets reset. Different file systems will do different things which are supported by the standard. The OS may look at modification times and conclude that the file hasn't changed so no resets are needed or may conclude that exit denotes a change and needs to rewind the file back to the beginning.

Waiting and Executing

If the parent process wants to wait for the child to finish, it must use `waitpid` (or `wait`), both of which wait for a child to change process states, which can be one of the following:

1. The child terminated
2. The child was stopped by a signal
3. The child was resumed by a signal

Note that waitpid can be set to be non-blocking, which means they will return immediately, letting a program know if the child has exited.

```
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. Here is the behavior of `waitpid`:

1. A program *can* wait on a specific process, or it can pass in special values for the `pid` to do different things (check the man pages).
2. The last parameter to waitpid is an option parameter. The options are listed below:
3. WNOHANG - Return whetherthe searched process has exited
4. WNOWAIT - Wait, but leave the child wait-able by another wait call
5. WEXITED - Wait for exited children
6. WSTOPPED - Wait for stopped children
7. 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 a program 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) {
  // Parent, wait for child to finish
  pid_t pid = waitpid(child, &status, 0);
  if (pid != -1 && WIFEXITED(status)) {
    int exit_status = WEXITSTATUS(status);
    printf("Process %d returned %d" , pid, exit_status);
  }
```

```
} else {
  // 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 extracted with bit shifting. However, the kernel has an internal way of keeping track of signaled, exited, or stopped processes. This API is abstracted so that that the kernel developers are free to change it at will. Remember: these macros only make sense if the precondition is met. For example, a process' exit status won't be defined if the process isn't signaled. The macros will not do the checking for the program, so it's up to the programmer to make sure the logic is correct. As an example above, the program 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 a high-level overview of how information is stored inside the status variables. From the `sys/wait.h` of an old Berkeley Standard Distribution(BSD) 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 aren't too many widely accepted conventions. If a program specifies return codes to mean certain conditions, it may be able to make more sense of the 256 error codes. For example, a program could return `1` if the program went to stage 1 (like writing to a file) `2` if it did something else, etc. Usually, UNIX programs are not designed to follow this policy, for the sake of simplicity.

Zombies and Orphans

It is good practice to wait on your process' children. If a parent doesn't wait on your children they become, what are called zombies. Zombies are created when a child terminates and then takes up a spot in the kernel process table for your process. The process table keeps track of the following information about a process: PID, status, and how it was killed. The only way to get rid of a zombie is to wait on your children. If a long-running parent never waits for your children, it may lose the ability to fork.

Having said that, a program doesn'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, whose PID is 1. Therefore, 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: Asynchronously Waiting

Warning: This section uses signals which are partially introduced. The parent gets the signal SIGCHLD when a child completes, so the signal handler can wait for 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) {
    // 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;
}
```

However, the above example 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 has temporarily stopped)
3. It uses the deprecated `signal` code, instead of the more portable sigaction.

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 that of the specified program. This means that any lines of code after the `exec` call are replaced with those of the executed program. Any other work a program wants 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.

Note that if the information is passed via an array, the last element must be followed by a NULL element to terminate the array.

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

```c
#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) {
    int status;
    waitpid(child , &status ,0);
    return EXIT_SUCCESS;

  } else {
    // 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

```c
#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. work as expected).

1. `open` – will use the lowest available file descriptor (i.e. 1) ; so standard out(stdout) is now redirected 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 0;" because compilers complain if we don't have it.

POSIX Exec Details

POSIX details all of the semantics that exec needs to cover [3]. Note the following

1. File descriptors are preserved after an exec. That means if a program open a file and doesn't to close it, it remains open in the child. This is a problem because usually the child doesn't know about those file descriptors. Nevertheless, they take up a slot in the file descriptor table and could possibly prevent other processes from accessing the file. The one exception to this is if the file descriptor has the Close-On-Exec flag set (O_CLOEXEC) – we will go over setting flags later.
2. Various signal semantics. The executed processes preserve the signal mask and the pending signal set but does not preserve the signal handlers since 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 executed process runs as the same PID and has the same parent and process group as the previous process.
6. The executed process is run on the same user and group with the same working directory

Shortcuts

`system` pre-packs the above code [9, P. 371]. The following is a snippet of how to use system.

```
#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 that is then given the string, which is more overhead than 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. It also tends to be a huge security risk. By allowing someone to access a shell version of the environment, the program can run into all sorts of problems:

```
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 called privilege escalation.

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 the execution of a new program. Meanwhile, the parent uses `wait` (or `waitpid`) to wait for the child process to finish.

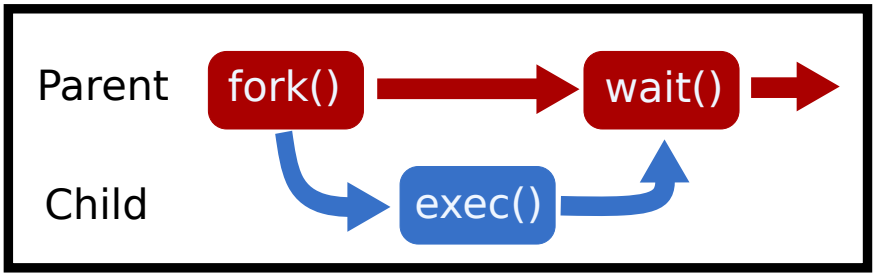

Figure 4.3: Fork, exec, wait diagram

```c
#include <unistd.h>

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

Why not execute ls directly? The reason is that now we have a monitor program – our parent that can do other things. It can proceed and execute another function, or it can also modify the state of the system or read the output of the function 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, some are already defined

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

How would a program later these in C? They can call `getenv` and `setenv` function respectively.

```
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 to specify a standard set of behaviors [2], although 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! Here are some guiding questions. Note that we aren't expecting you to memorize the man page.

- What is one reason fork may fail?
- Does fork copy all pages to the child?
- Are file descriptors cloned between parent and child?
- Are file descript**ions** cloned between parent and child?
- What is the difference between exec calls ending in an e?
- What is the difference between l and v in an exec call? How about p?
- When does exec error? What happens?
- Does wait only notify if a child has exited?
- Is it an error to pass a negative value into wait?
- How does one extract information out of the status?
- Why may wait fail?
- What happens when a parent doesn't wait on their children?
- 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 press CTRL-C at a terminal?
- 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 does a program pass in command line arguments to `execl*`? How about `execv*`? What should be the first command line argument by convention?
- How does a program 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 a program set up a signal handler for?
- What signal is sent when you press `CTRL-C`?
- My terminal is anchored to PID = 1337 and has 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 a program write to? What are invalid memory addresses?
- Code up a fork bomb in C (please don't run it).
- What is an orphan? How does it become a zombie? What should a parent do to avoid this?
- Don't you hate it when your parents tell you that you can't do something? Write a program that sends `SIGSTOP` to a parent process.
- 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.