Lecture 16 - fork() and exec() (last updated: 11/03/22)

In UNIX, each instance of a running program is called a process, uniquely identified by a positive number called the process ID (PID). C programs can obtain their PID at runtime using the getpid() system call.

Processes may spawn more processes, which we call "child" processes. The process that spawned the child is known as the child's "parent." A process can obtain the PID of its parent (PPID) using the getppid() system call. All processes "descend" from the "init" process, whose PID is 1 and is the first process to run when a UNIX system boots.

In this lecture, we will learn about how processes are created using fork(), managed using waitpid(), and made to execute other programs using exec().

Spawning processes with fork()

A process spawns a child process using the fork() system call:

```
pid_t fork(void);
```

It forms the child process's memory address space by duplicating the parent's address space; after returning from the fork() system call, both processes will resume execution from the same place in the code.

The parent and the child will be able to distinguish themselves from each other using the return value of fork(), which will return 0 to the child process, and the PID of the child process to the parent process:

```
pid_t pid = fork();

// Both the parent and child will resume execution here.

if (pid == 0)
    // Child process
    printf("This is the child, my PID is %d\n", getpid());
else
    // Parent process
    printf("This is the parent, my child's PID is %d\n", pid);

// Both the parent and child will print out "Hello"
printf("Hello\n");
```

Note that since BOTH the parent and the child will resume execution from the point where fork() returns, "Hello" is printed twice in the above example!

The parent and child will execute concurrently after returning from fork(), meaning there is no guarantee of their execution order relative to one another. In other words, the pace and progress of their execution is independent from each other. In general, all processes execute concurrently with one another unless they use some kind of OS-facilitated synchronization mechanism (including but not limited to waitpid()).

After a parent spawns a child process, they will no longer share any memory (unless they explicitly opt into shared memory; beyond the scope of this class). This means that they will have separate copies of the same local and global variables; modifications to memory made by the child process will not be visible

to the parent process (and vice versa). It also means that they will have separate stacks and heaps: both parent and child will separately roll up their stacks as they return from each function call, and they will also need to separately free() any heap memory they've allocated.

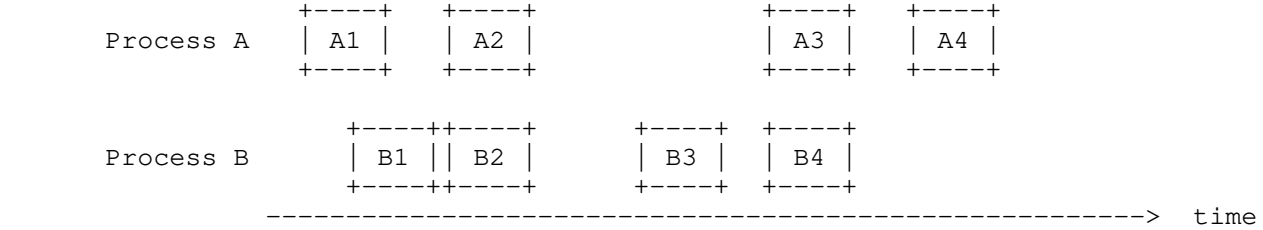
A more subtle point is that the file descriptors are now also duplicated between parent and child processes, so they will each need to close() their copy of the same file descriptor. This is important if, say, the child inherits a copy of a socket file descriptor from the parent; both the parent and child will need to close() that file descriptor in order to terminate the socket connection.

Note that a single parent process may spawn multiple children by calling fork(); similarly, child processes can spawn children of their own by calling fork(). The parent-child relationships form a process family tree rooted at the init process, the ancestor of all processes on a UNIX system.

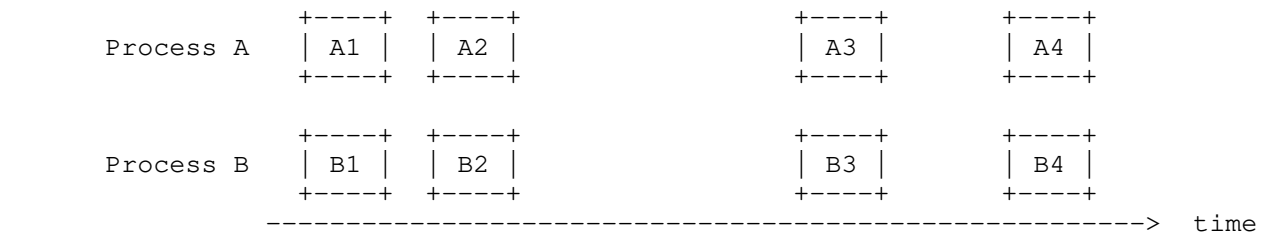
Concurrency vs parallelism (optional)

Editorial note: the meaning of "concurrent" here should not be confused with "parallel!" "Concurrent" refers to the lack of synchronization between processes, while "parallel" describes processes that are physically executing at the same time. Parallel processes often happen to be unsynchronized and are therefore concurrent, so concurrency is often used as a model of parallelism.

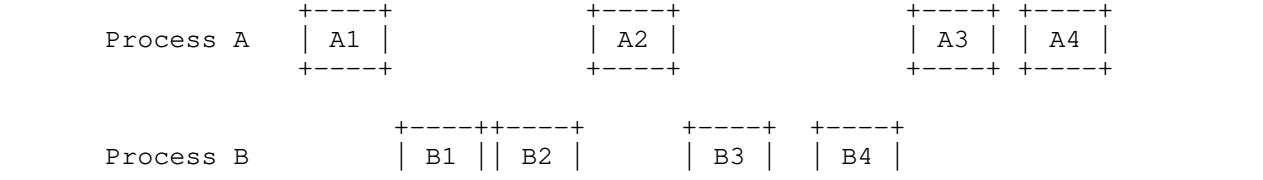
For example, the following diagram illustrates two parallel processes, A and B, which each concurrently execute four steps, depicted as events A1–A4 and B1–B4:

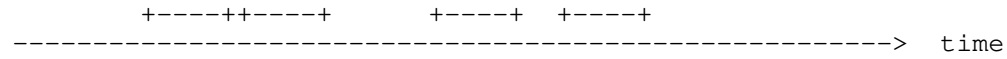


It's possible to have parallel computation without concurrency, where two parallel processes execute in lockstep:



It's also possible to have concurrent computation without parallelism, such as when two processes' executions are interleaved, but never take place at the same time:





Note that the interleaving of the executions is arbitrary, since the execution order of concurrent computation is non-deterministic.

UNIX processes execute concurrently, but that does not necessarily mean they are parallel. For example, if your computer only has a single CPU, only one process can execute at a time, ruling out any possibility for true parallelism.

Exit status codes and waitpid()

When a process terminates, it does so with an integer known as the exit status code. By convention, an exit status code of 0 indicates success, while a non-zero code indicates some kind of error (whose specific meaning is determined by each application).

In particular, when a process returns from its program's main() function, the return value of main() is used as the exit status. A process may also immediately terminate at any point during its execution by calling the exit() function, passing the status code as an argument to exit().

So, the following two programs behave equivalently; one returns from main():

```
int main(void) {
    return 3;   // Exit with exit status code of 3
}
```

while the other calls exit() explicitly:

```
int main(void) {
    exit(3);    // Exit with exit status code of 3
    return 0;   // Never reached!
}
```

When a process terminates, it is the parent's responsibility to handle its exit status code. In general, parent processes should manage the lifetimes of their children and live long enough to witness their termination; in UNIX parlance, this is called "reaping" the child.

Parent processes reap their children using the waitpid() system call:

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

waitpid() can block the parent process until it successfully reaps a child. The first parameter of waitpid() specifies which process to wait for; the parent may pass the PID of a specific child process to only wait for that process, or pass the number -1 to wait for any of its child processes. When waitpid() returns, it will return the PID of the child process it reaped.

The second parameter, wstatus, tells waitpid() where to write the terminated child's exit status code. For example:

```
pid_t pid = fork();

if (pid == 0) {
    // Child process
    // ...
```

```
    } else {
        // Parent process
        int wstatus;
        waitpid(pid, &wstatus, 0);
        printf("Child (PID = %d) exited with: %d\n", pid, WEXITSTATUS(wstatus));
    }
```

Note that we use the WEXITSTATUS() macro to read the status code from wstatus (which also holds other information about why and how the child terminated). Passing a NULL pointer as wstatus tells waitpid() to discard the status code.

The last parameter of waitpid() allows the caller to customize the blocking behavior of waitpid(), where the default option, 0, tells waitpid() to block until a child terminates. However, specifying the WNOHANG option tells waitpid() not to block if no child has terminated. This option is useful for when the parent wants to reap any terminated children without blocking and waiting on them to terminate. A common pattern is to call waitpid() with WNOHANG in a loop, to reap any children encountered, e.g.:

```
    pid_t pid;
    while ((pid = waitpid(-1, NULL, WNOHANG)) > 0)
        printf("Reaped child (PID = %d)\n", pid);
```

waitpid() returns 0 if it did not reap any child process, terminating the loop and allowing the parent to resume other tasks.

A child process that has terminated but has not yet been reaped is called a "zombie process." The OS needs to keep around some metadata about the zombie process to report to the parent if and when it is reaped, such as the PID and the exit status code. Though this metadata consumes a minimal amount of the system's resources, it can add up as the system maintains metadata for a large number of zombies, so parent processes mustn't forget to reap their children!

However, when a parent process terminates before its children do, the child process is said to have become an "orphan process." All orphan processes are adopted by the init process and automatically reaped when they terminate. Unlike zombie processes, orphan processes are not inherently bad, and are actually quite common in UNIX systems. In particular, long-running processes, called "daemons," often run as orphan processes, acting as servers or performing background tasks (e.g., taking periodic file system snapshots).

Executing programs with exec()

Although fork() creates a new process, it doesn't change the program that the process is running: the child process is just a duplicate of the parent. To complement fork(), UNIX also provides a family of system calls, generally referred to as exec(), which replace the program of the running process. (Technically, there is only one system call, execve(); all the other exec() functions are just wrappers for execve().)

While the usage and behavior of each exec() system call differs slightly from the others, they all accomplish the same goal: they allow processes to load a program from the file system and execute it. In this course, we will discuss the execv() variant (the 'v' stands for "vector", the same as in "argv"), whose type signature is as follows (with const qualifiers omitted for clarity):

```
    int execv(char *pathname, char **argv);
```

pathname tells execv() what program to execute, and argv tells it what arguments to execute that program with. For example, here's how to use execv() to execute the echo program (whose full path is /usr/bin/echo) with the argument "hello":

```
char *argv[] = { "/usr/bin/echo", "hello", NULL };
// argv must be null-terminated with a NULL pointer!

execv(argv[0], argv);

// UNREACHABLE
```

When called successfully, execv() will not return (nor will any of its cousins in the exec() family); instead, the running process will transform into the echo program and print "hello". Although that process will retain its PID, parent, and most of its file descriptors, its memory address space will be overwritten as the echo program is loaded into memory: any local variables, global variables, heap-allocated memory, and even string constants that existed before the exec() call will be destroyed in the process.

Note that in the example above, we reuse argv[0] as the pathname, because by convention, the argv[0] of a program should be the file path of the program. While it's also possible to specify a different argv[0] than the pathname, doing so may confuse programs that assume argv[0] is set according to convention.

A common pattern is to combine fork() and exec(), where the parent process waits for its child to execute another program:

```
pid_t pid = fork();

if (pid == 0) {
    // Child process

    char *argv[] = { "/usr/bin/echo", "hello", NULL };
    execv(argv[0], argv);

    die("exec");
} else {
    // Parent process

    waitpid(pid, NULL, 0);
}
```

In fact, this is what shell programs like Bash and Zsh do! They read in your command from stdin, fork() and exec() a child process to run your command, and waitpid() for that child to terminate before prompting you for another command.