

# Forking, Part 1: Introduction

Thomas Liu edited this page on Jan 25 · 3 revisions

## A word of warning

Process forking is a very powerful (and very dangerous) tool. If you mess up and cause a fork bomb (explained later on this page), **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.

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.**

## What does fork do?

The `fork` system call clones the current process to create a new process. It creates a new process (the child process) by duplicating the state of the existing process with a few minor differences (discussed below). The child process does not start from main. Instead it returns from `fork()` just as the parent process does.

## What is the simplest `fork()` example?

Here's a very simple example...

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

## Why does this example print 42 twice?

The following program prints 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();
}
```

Edit

New Page

▼ Pages 51

Find a Page...

[Home](#)

[#Example Markdown](#)

[#Informal Glossary](#)

[#Piazza: When And How to Ask For Help](#)

[C Programming, Part 1: Introduction](#)

[C Programming, Part 2: Text Input And Output](#)

[C Programming, Part 3: Common Gotchas](#)

[C Programming, Part 4: Debugging](#)

[Deadlock, Part 1: Resource Allocation Graph](#)

[Deadlock, Part 2: Deadlock Conditions](#)

[File System, Part 1: Introduction](#)

[File System, Part 2: Files are inodes \(everything else is just data...\)](#)

[File System, Part 3: Permissions](#)


[File System, Part 4: Working with directories](#)

[File System, Part 5: Virtual file systems](#)


Show 36 more pages...


Clone this wiki locally


https://github.com/angrave/SystemPr


 Clone in Desktop

<>









```
    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 in 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.

## How do you write code that is different for the parent and child process?

Check the return value of `fork()`. Return value -1= failed; 0= in child process; positive = in parent process (and the return value is the child process id). Here's one way to remember which is which:

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()`. 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
}
```

## What is a fork bomb ?

A 'fork bomb' is when you attempt to create an infinite number of processes. A simple example is shown below:

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

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. Comment: 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 student coding errors.

Angrave suggests that the Matrix trilogy, where the machine and man finally work

together to defeat the multiplying Agent-Smith, was a cinematic plot based on an AI-driven fork-bomb.

# How does the parent process wait for the child to finish?

Use `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);
}
```

# Can I make the child process execute another program?

Yes. 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.

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.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", "ls","-alh", (char *) NULL);

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

# A simpler way to execute another program

Use `system` !!! Here is how to use it:

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

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

The `system` call would fork, exec the command passed by parameter and the parent process would 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. This may be useful or it may not be, use with caution.

## What is the silliest fork example?

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;
}
```

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;
}
```

## What is different in the child process than the parent process?

The key differences include:

- The process id returned by `getpid()` . The parent process id returned by `getppid()` .

- The parent is notified via a signal 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](#)

# Do child processes share open filehandles?

Yes! In fact both processes use the same underlying kernel file descriptor. For example if one process rewinds the random access position back to the beginning of the file, then both processes are affected.

Both child and parent should `close` (or `fclose` ) their file descriptors or file handle respectively.

## How can I find out more?

Read the man pages!

- [fork](#)
- [exec](#)
- [wait](#)

Legal and Licensing information: Unless otherwise specified, submitted content to the wiki must be original work (including text, java code, and media) and you provide this material under a [Creative Commons License](#). If you are not the copyright holder, please give proper attribution and credit to existing content and ensure that you have license to include the materials.

