What is the difference between using _exit() & exit() in a conventional Linux fork-exec?

29

11

I've been trying to figure out how the fork-exec mechanism is used inside Linux. Everything was going on according to the plan until some web pages started to confuse me.

It is said that a child process should strictly use `_exit()` instead of a simple `exit()` or a normal return from `main()`.

As I know, Linux shell fork-execs every one of the external commands; assuming what I said above is true, the conclusion is that none of these external commands nor any other execution happening inside the Linux shell can do normal return!

Wikipedia & some other webpages claim we've got to use `_exit()` just to prevent a child process causing deletion of parent's temporary files while a probable double flushing of stdio buffers may happen. though I understand the former, I have no clues how a double flushing of buffers could be harmful to a Linux system.

I've spent my whole day on this... Thanks for any clarification.

c linux return fork exit

share improve this question

edited Mar 25 '11 at 5:16

jschmier

10.8k 3 34 61

asked Mar 24 '11 at 17:05

ned1986zha

170 1 2 5

Duplicate of stackoverflow.com/questions/2329640/… . Also related: stackoverflow.com/questions/3657667/exit-functions-in-c – Adam Rosenfield Mar 25 '11 at 5:26

add a comment

2 Answers

active oldest votes

29

You should use `_exit` (or its synonym `_Exit`) to abort the child program when the `exec` fails, because in this situation, the child process may interfere with the parent process' external data (files) by calling its `atexit` handlers, calling its signal handlers, and/or flushing buffers.

For the same reason, you should also use `_exit` in any child process that does not do an `exec`, but those are rare.

In all other cases, just use `exit`. As you partially noted yourself, *every* process in Unix/Linux (except one, `init`) is the child of another process, so using `_exit` in every child process would mean that `exit` is useless outside of `init`.

```
switch (fork()) {
  case 0:
    // we're the child
    execlp("some", "program", NULL);
    _exit(1);  // <-- HERE
  case -1:
    // error, no fork done ...
  default:
    // we're the parent ...
}
```

share improve this answer

edited Mar 24 '11 at 17:36

answered Mar 24 '11 at 17:29

Fred Foo

229k 34 435 610

TONS of Thanks. I feel much better now; though still twice flushing of a buffer being problematic, has no meaning for me. – ned1986zha Mar 24 '11 at 18:11

3 @ned1986zha: At the time of the `fork()`, there may be data in the stdio buffers. If both the parent and the child flush those buffers, that data will appear twice in the output. The problem doesn't exist if `exec()` succeeds, because in that case the newly exec'd process starts with fresh stdio buffers. – caf Mar 24 '11 at 22:10

3 @ned1986zha: Yes - the `stdio` buffer is a C library mechanism. There may be OS-provided buffering as well - but that doesn't matter, because the `fork()` won't duplicate that. – caf Mar 25 '11 at 8:30

1 @Bin: As the comments above say, `_exit()` ensures that the C-library-level (userspace stdio) buffers aren't flushed - because those buffers get duplicated by `fork()` (since they're userspace, the kernel isn't aware of them). Any kernel-level buffers would still get flushed, but that's OK, because kernel-level buffers aren't duplicated by `fork()`. – caf Jul 26 at 0:13

1 @Bin: When the process is terminated by the kernel, the file descriptors get closed but the stdio buffers aren't flushed - because the kernel doesn't know anything about those buffers, they're entirely a creation of the userspace C library. – caf Jul 26 at 13:56

show **5** more comments

8

`exit()` flushes io buffers and does some other things like run functions registered by `atexit()`. `exit()` invokes `_end( )`

`_exit()` just ends the process without doing that. You call `_exit()` from the parent process when creating a daemon for example.

Ever notice that `main()` is a function? Ever wonder what called it in the first place? When a c program runs the shell you are running in provides the executable path to 'exec' system call and the control is passed to kernel which in turn calls the startup function of every executable `_start()`, calls your `main()`, when `main()` returns it then calls `_end()` Some implementations of C use slightly different names for `_end()` & `_start()` ...

`exit()` and `_exit()` invoke `_end()`

Normally - for every `main()` there should be one & only one `exit()` call. (or return at the end of `main()`)