**Linux process states**

**ptrace tutorial part #1**

11 December 2012

Recently I've been experimenting with Linux's [ptrace(2)](http://www.kernel.org/doc/man-pages/online/pages/man2/ptrace.2.html) syscall. Unfortunately, there isn't any kind of "official" documentation for it and the manual page is quite poor. There are some other attempts to document it, for example [in the strace sources](http://strace.git.sourceforge.net/git/gitweb.cgi?p=strace/strace;a=blob_plain;f=README-linux-ptrace;hb=HEAD) or in [several](http://www.linuxjournal.com/article/6100) [introductory](https://mikecvet.wordpress.com/2010/08/14/ptrace-tutorial/) [tutorials](http://www.secretmango.com/jimb/Whitepapers/ptrace/ptrace.html) online, but nothing explains how ptrace works from the ground up.

Ptrace was always treated by kernel developers as a half-baked hack and not a well-designed debugging interface. Everyone agrees that it's suboptimal and alternatives like [perf trace](http://kernelnewbies.org/Linux_3.7#head-5f16e5d3f07c2a1601c98f4b81e522c12f6cae7b) and [utrace](http://lwn.net/Articles/224772/) have been proposed. Perf is very young and utrace is not a favourite technology of Linus, and according to him ptrace will likely [stay with us for the foreseeable future](http://www.yarchive.net/comp/linux/utrace.html):

> ptrace is a nasty, complex part of the kernel which has  
> a long history of problems, but it's all been pretty quiet  
> in there for the the past few years.

More importantly, we're not ever going to get rid of it.

**Digging into ptrace**

Ptrace is a complex, low-level debugging facility, and the magic behind tools like strace and gdb. Importantly, ptrace doesn't require administrator rights.

Let's start our journey with the [ptrace(2) man page](http://www.kernel.org/doc/man-pages/online/pages/man2/ptrace.2.html):

[...] While being traced, the child will stop each time a signal is delivered, even if the signal is being ignored. The parent will be notified at its next wait(2) and may inspect and modify the child process while it is stopped. [...]

Okay, but what exactly does it mean for a process to be "stopped" and what is [wait(2)](http://www.kernel.org/doc/man-pages/online/pages/man2/wait.2.html) doing?

Ptrace reuses a common Unix mechanism of "stopping" and "continuing" processes. Let's forget about ptrace for a moment and dig deeper into this - the linux process state logic.

**Linux process states**

As in every Unix flavour, in Linux a process can be in a number of states. It's easiest to observe it in tools like ps or top: it's usually in the column named S. The documentation of ps describes the possible values:

PROCESS STATE CODES

R running or runnable (on run queue)

D uninterruptible sleep (usually IO)

S interruptible sleep (waiting for an event to complete)

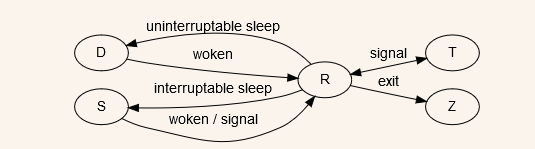
Z defunct/zombie, terminated but not reaped by its parent

T stopped, either by a job control signal or because

it is being traced

[...]

A process starts its life in an R "running" state and finishes after its parent reaps it from the Z "zombie" state.



(Sources: [linusakesson.net](http://linusakesson.net), [ufsc.br](http://www.inf.ufsc.br/%7Efernando/ine5355/linux-process.pdf), [macdesign.net](http://macdesign.net). I can't stop myself from mentioning a completely irrelevant but interesting recently introduced linux task state: TASK\_KILLABLE, discussion: [one](http://lwn.net/Articles/288056/), [two](http://www.ibm.com/developerworks/linux/library/l-task-killable/).)

**Bash and STOPPED**

We're particularly interested in the T "stopped" state, so let's have a play with that. The easiest way to see it in action is to use the shell and press CTRL+z:

$ sleep 100

^Z # Pressed CTRL+z

[1]+ Stopped

$ ps -o pid,state,command

PID S COMMAND

13224 T sleep 100

[...]

At this stage, after pressing CTRL+z, the sleep process is in T "stopped" state. It will remain in this state and won't get any CPU until we "continue" it. fg or bg commands can do the trick:

$ bg

[1]+ sleep 100 &

$ ps -o pid,state,command

PID S COMMAND

13224 S sleep 100

[...]

Yup, our sleep command is running again. Actually it's sleeping in the S "interruptable" state. If you wish to see R "running" state use something CPU-intensive instead of sleep, say: yes > /dev/null.

**SIGSTOP, SIGCONT**

When you press CTRL+z, under the hood kernel terminal driver sends a SIGSTOP signal to foreground processes. Similarly, on bg / fg bash sends a SIGCONT signal. The manual page [signal(7)](http://www.kernel.org/doc/man-pages/online/pages/man7/signal.7.html) describes the signals:

Signal Dispositions

Each signal has a current disposition, which determines how the process behaves when it is delivered the signal.

Stop Default action is to stop the process.

Cont Default action is to continue the process

if it is currently stopped.

[...]

Signal Value Action Comment

──────────────────────────────────────────────────

SIGCONT 18 Cont Continue if stopped

SIGSTOP 19 Stop Stop process

[...]

The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.

Actually, sending SIGSTOP and SIGCONT signals directly to a process will work equally well and is indistinguishable from CTRL+z and bg/fg in bash:

$ sleep 100 &

[1] 28761

$ ps -o pid,state,command

PID S COMMAND

28761 S sleep 100

[...]

$ kill -STOP 28761

[1]+ Stopped sleep 100

$ ps -o pid,state,command

PID S COMMAND

28761 T sleep 100

[...]

$ kill -CONT 28847

$ ps -o pid,state,command

PID S COMMAND

28847 S sleep 100

[...]

The behaviour is exactly the same when process gets signals from any other source - for example sending SIGSTOP to ourselves also will put the process into the T "stopped" state:

$ python -c "import os, signal; os.kill(os.getpid(), signal.SIGSTOP)"

[1]+ Stopped

**SIGCHLD and waitpid()**

Whenever a child process changes its state - either gets stopped, continues or exits - two things happen to the parent process:

* it gets a SIGCHLD signal
* a blocking [waitpid(2)](http://www.kernel.org/doc/man-pages/online/pages/man2/waitpid.2.html) (or wait) call may return

By default waitpid blocks until a selected child exits, but by setting specific flags we can also receive notifications about other state changes: a child process being stopped (flag WUNTRACED) or continued (flag WCONTINUED).

Take a look at this code:

import os

import sys

import signal

def waitpid():

(pid, status) = os.waitpid(-1,

os.WUNTRACED | os.WCONTINUED)

if os.WIFSTOPPED(status):

s = "stopped sig=%i" % os.WSTOPSIG(status)

elif os.WIFCONTINUED(status):

s = "continued"

elif os.WIFSIGNALED(status):

s = "exited signal=%i" % os.WTERMSIG(status)

elif os.WIFEXITED(status):

s = "exited status=%i" % os.WEXITSTATUS(status)

print "waitpid received: pid=%i %s" % (pid, s)

childpid = os.fork()

if childpid == 0:

# Child

os.kill(os.getpid(), signal.SIGSTOP)

sys.exit()

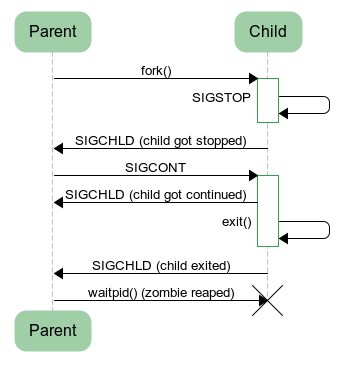
waitpid()

os.kill(childpid, signal.SIGCONT)

waitpid()

waitpid()

The control flow is described by this diagram:



The parent process doesn't install handler for SIGCHLD - this signal is ignored by default. Here's the output - the parent prints results of three waitpid calls:

$ python parent.py

waitpid received: pid=16935 stopped sig=19

waitpid received: pid=16935 continued

waitpid received: pid=16935 exited status=0

**Zombies**

A zombie process is a process that exited successfully, but its state change wasn't yet acknowledged by the parent. That is, the parent didn't call wait() / waitpid() functions.

The Z "zombie" process state is required in order to give a parent time to ask the kernel about the resources used by the deceased child, using getrusage(2). A parent informs a kernel that it's done with the child by calling waitpid.

Most often the parent doesn't really care about child process resources or exit status. In such case a common way to avoid zombies to install a SIGCHLD handler and call waitpid within it. Unfortunately, as defined by Unix, many individual signals sent to the same process with the same signal number may be coalesced into one. A single call to SIGCHLD signal handler might actually be triggered by more than one child state change. Therefore if you have more than one child process you may need to run waitpid in a loop to reap zombies, like this:

static void sigchld\_handler(int sig) {

int status;

int pid;

while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {

// `pid` exited with `status`

}

}

Alternatively, to totally avoid zombies, one can explicitly set SIGCHLD signal handler to SIG\_IGN or use SA\_NOCLDWAIT flag for sigaction (see NOTES in [waitpid(2)](http://www.kernel.org/doc/man-pages/online/pages/man2/wait.2.html#NOTES)):

signal(SIGCHLD, SIG\_IGN);

**Wrapping up process states**

Process states form an interesting mechanism that is basically creating a synchronous communication channel between a parent and a child process.

For example - if a child changes its state to "stopped" state, a parent can wait for that using waitpid. Later it can order a child to continue by sending SIGCONT.

This mechanism is not flawless - if a child goes into "stopped" state and quickly receives SIGCONT, the parent will receive SIGCHLD, but waitpid may miss the state change.

**Back to ptrace**

"Stopping" and "continuing" is the mechanism used by ptrace to controll a debugged process. First, on initialisation ptrace causes the current (debugging) process to temporarily become a parent of a debugged process (let's call it "adoption"). As a parent it will be notified about child process state changes. Next, various ptrace flags inform the kernel to put the child into "stopped" state when particular debugging events occur. When such an event is triggered the parent receives SIGCHLD, can retrieve child status via waitpid and has a chance to inspect the stopped child. When it's done, it puts child back into "running" state.

We'll see how to use ptrace to do this in the next part of this tutorial.

The way ptrace works is a huge abuse of the original Unix process model, but in practice it seems to work quite well. However this mechanism is not very efficient due to the high overhead of constant context switches between the parent and the child.

**Ptrace and security**

In the beginning I quoted:

ptrace is a nasty, complex part of the kernel which has a long history of problems

Indeed, a number of serious security issues were found in kernel ptrace code. This is so noticeable that [Ubuntu decided to disable the ability to run a ptrace](https://wiki.ubuntu.com/SecurityTeam/Roadmap/KernelHardening#ptrace_Protection) against unrelated processes by an untrusted user. You can see if your Linux has that restriction enabled by looking at the output of sysctl command:

$ /sbin/sysctl kernel.yama.ptrace\_scope

kernel.yama.ptrace\_scope = 1

With this restriction in place untrusted ptrace can be only be run against the parent's genuine children, "adoption" is not possible any more without administrator rights.