# Ruby - Feature #19322

## Support spawning "private" child processes

01/07/2023 07:40 AM - kjtsanaktsidis (KJ Tsanaktsidis)

| | |
|---|---|
| **Status:** | Open |
| **Priority:** | Normal |
| **Assignee:** | |
| **Target version:** | |

**Description**

# Background

The traditional Unix process APIs (fork etc) are poorly isolated. If a library spawns a child process, this is not transparent to the program using the library. Any signal handler for SIGCHLD in the program will be called when the spawned process exits, and even worse, if the parent calls Process.waitpid2(-1), it will consume the returned status code, stealing it from the library!

Unfortunately, the practice of responding to SIGCHLD by calling waitpid2(-1) in a loop is a pretty common unixism. For example, Unicorn does it here. In short, there is no reliable way for a gem to spawn a child process in a way that can't (unintentionally) be interfered with by other parts of the program.

# Problem statement

Consider the following program.

```
# Imagine this part of the program is in some top-level application event loop
# or something - similar to how Unicorn works. It detects child processes exiting
# and takes some action (possibly restarting a crashed worker, for example).
Signal.trap(:CHLD) do
  loop do
    begin
      pid, status = Process.waitpid2 -1
      puts "Signal handler reaped #{pid} #{status.inspect}"
    rescue Errno::ECHILD
      puts "Signal handler reaped nothing"
      break
    end
  end
end

# Imagine that _this_ part of the program is buried deep in some gem. It knows
# nothing about the application SIGCHLD handling, and quite possibly the application
# author might not even know this gem spawns a child process to do its work!
require 'open3'
loop do
  o, status = Open3.capture2("/bin/sh", "-c", "echo 'hello'")
  puts "ran command, got #{o.chomp} #{status.inspect}"
end
```

In current versions of Ruby, *some* loop iterations will function correctly, and print something like this. The gem gets the Process::Status object from its command and can know if e.g. it exited abnormally.

```
ran command, got ohaithar #<Process::Status: pid 1153687 exit 0>
Signal handler reaped nothing
```

However, other iterations of the loop print this. The signal handler runs and calls Process.waitpid2(-1) before the code in open3 can do so. Then, the gem code does not get a Process::Status object! This is also potentially bad for the application; it reaped a child process it didn't even know existed, and it might cause some surprising bugs if the application author didn't know this was a possibility.

```
Signal handler reaped 1153596 #<Process::Status: pid 1153596 exit 0>
Signal handler reaped nothing
ran command, got ohaithar nil
```

We would like a family of APIs which allow a gem to spawn a child process and guarantees that the gem can wait on it. Some concurrent call to Process.waitpid2(-1) (or even Process.waitpid2($some_lucky_guess_for_the_pid)) should not steal the status out from underneath the code which created the process. Ideally, we should even suppress the SIGCHLD signal to avoid the application signal handler needlessly waking up.

## Proposed Ruby-level APIs.

I propose we create the following new methods in Ruby.

- Process.spawn_private
- Process.fork_private

These methods behave identically to their non-_private versions in all respect, except instead of returning a pid, they return an object of type Process::PrivateHandle.

Process::PrivateHandle would have the following methods:

- pid() - returns the pid for the created process
- wait() - blocks the caller until the created process has exited, and returns a Process::Status object. If the handle has *already* had #wait called on it, it returns the same Process::Status object as was returned then immediately. This is unlike Process.waitpid and friends, which would raise an ECHILD in this case (or, in the face of pid wraparound, potentially wait on some other totally unrelated child process with the same pid).
- wait_nonblock() - if the created process has exited, behaves like #wait; otherwise, it returns a Process::Status object for which #exited? returns false.
- kill(...) - if the created process has not been reaped via a call to #wait, performs identically to Process.kill ..., pid. Otherwise, if the process *has* been reaped, raises Errno::ESRCH immediately without issuing a system call. This ensures that, if pids wrap around, that the wrong process is not signaled by mistake.

A call to Process.wait, Process.waitpid, or Process.waitpid2 will *never* return a Process::Status for a process started with a _private method, even if that call is made with the pid of the child process. The *only* way to reap a private child process is through Process::PrivateHandle.

The implementation of IO.popen, Kernel#system, Kernel#popen, backticks, and the Open3 module would be changed to use this private process mechanism internally, although they do not return pids so they do not need to have their interfaces changed. (note though - I don't believe Kernel#system suffers from the same problem as the open3 example above, because it does not yield the GVL nor check interrupts in between spawning the child and waiting on it)

## Implementation strategy

I believe this can be implemented, in broad strokes, with an approach like this:

- Keep a global table mapping pids -> handles for processes created with fork_private or spawn_private.
- When a child process is waited on, consult the handle table. If there is a handle registered, and the wait call was made without the handle, do NOT return the reaped status. Instead, save the status against the handle, and repeat the call to waitpid.
- If the wait call *was* made with the handle, we can return the
- Once a handle has had the child status saved against it, it is removed from the table.
- A subsequent call to wait on that pi the handle will look up the saved information and return it without making a system call.

In fact, most of the infrastructure to do this correctly is already in place - it was added by @k0kubun (Takashi Kokubun) and @normalperson (Eric Wong) four years ago - https://bugs.ruby-lang.org/issues/14867. MJIT had a similar problem to the one described in this issue; it needs to fork a C compiler, but if the application performs a Process.waitpid2(-1), it could wind up reaping the gcc process out from underneath mjit. This code has changed considerably over the course of last year, but my understanding is that mjit still uses this infrastructure to protect its Ruby child-process from becoming visible to Ruby code.

In any case, the way waitpid works *currently*, is that...

- Ruby actually does all calls to waitpid as WNOHANG (i.e. nonblocking) internally.
- If a call to waitpid finds no children, it blocks the thread, representing the state in a structure of type struct waitpid_state.
- Ruby also keeps a list of all waitpid_state's that are currently being waited for, vm->waiting_pids and vm->waiting_grps.
- These structures are protected with a specific mutex, vm->waitpid_lock.
- Ruby internally uses the SIGCHLD signal to reap the dead children, and then find a waiting call to waitpid (via the two lists) to actually dispatch the reaped status to.
- If some caller is waiting for a specific pid, that *always* takes priority over some other caller that's waiting for a pid-group (e.g. -1).

mjit's child process is protected, because:

- When mjit forks, it uses a method rb_mjit_fork to do so.
- That calls the actual fork implementation *whilst still holding* vm->waitpid_lock
- Before yielding the lock, it inserts an entry in vm->waiting_pids saying that mjit is waiting for the just-created child.
- Since direct waits for pids always take precedence over pid-groups, this ensures that mjit will always reap its own children.

I believe this mechanism can be extended and generalised to power the proposed API, and mjit could itself use that rather than having mjit-specific handling in process.c.

## POC implementation

I sketched out a *very* rough POC to see if what I said above would be possible, and I think it is:

https://github.com/ruby/ruby/commit/6009c564b16862001535f2b561f1a12f6e7e0c57

The following script behaves how I expect with this patch:

```
pid, h = Process.spawn_private "/bin/sh", "-c", "sleep 1; exit 69"
puts "pid -> #{pid}"
puts "h -> #{h}"

# should ESRCH.
sleep 2
begin
    Process.waitpid2 -1
rescue => e
    puts "waitpid err -> #{e}"
end
wpid, status = h.wait
puts "wpid -> #{wpid}"
puts "status -> #{status.inspect}"

ktsanaktsidis@lima-linux1 ruby % ./tool/runruby.rb -- ./tst1.rb
pid -> 1154105
h -> #<Process::PrivateHandle:0x0000ffff94014098>
waitpid err -> No child processes
wpid -> 1154105
status -> #<Process::Status: pid 1154105 exit 4>
```

The child process can be waited on with the handle, and the call to waitpid2(-1) finds nothing.

## Previous idea: OS-specific handles

My first version of this proposal involved a similar API, but powering it with platform-specific concepts available on Linux, Windows, and FreeBSD which offer richer control than just pids & the wait syscall. In particular, I had believed that we could use the clone syscall in Linux to create a child process which:

- Could be referred to by a unique file descriptor (a pidfd) which would be guaranteed never to be re-used (unlike a pid),
- Would not generate a signal when it exited (i.e. no SIGCHLD).
- Could not be waited on by an unspecting to waitpid (except if a special flag __WCLONE as passed).

Unfortunately, when I tried to implement this, I ran into a pretty serious snag. It is possible to create such a process - BUT, when the process exec's, it goes *back* to "raise-SIGCHLD-on-exit" and "allow-waiting-without-__WCLONE" modes. I guess this functionality in the clone syscall is really designed to power threads in Linux, rather than being a general-purpose "hidden process" API.

So, I don't think we should use pidfds in this proposal.

## Motivation

My use-case for this is that I'm working on a perf-based profiling tool for Ruby. To get around some Linux capability issues, I want my profiler gem (or CRuby patch, whatever it winds up being!) to fork a privileged helper binary to do some eBPF twiddling. But, if you're profiling e.g. a Unicorn master process, the result of that binary exiting might be caught by Unicorn itself, rather than my (gem | interpreter feature).

In my case, I'm so deep in linux specific stuff that just calling clone(2) from my extension is probably fine, but I had enough of a look at this process management stuff I thought it would be worth asking the question if this might be useful to other, more normal, gems.

## History

#### #1 - 01/07/2023 08:59 AM - nobu (Nobuyoshi Nakada)

Already possible solution would be a daemon process:

```
IO.popen("-", "r+") do |childio|
  if childio
    # In parent process
    Process.wait(childio.pid)
    # `Process.wait` no longer consume the returned status code.

    # Wait the grandchild process to finish
    childio.read
  elsif Process.fork
    # In child process
    exit
  else
    # In grandchild process
    do_something(STDIN, STDOUT)
  end
end
```

#### #2 - 01/07/2023 09:10 AM - kjtsanaktsidis (KJ Tsanaktsidis)

I did think about something in that shape @nobu (Nobuyoshi Nakada) for unsupported systems, but I think there are two problems -

- SIGCHLD will still be received, which is undesirable on its own
- if a different thread is running Process.waitpid2(-1) concurrently, there is no guarantee who will reap the intermediate parent process - their call or our call to waitpid.

I guess we might not really care if we miss the status of the intermediate (it can't really fail, and we'll know it's dead because we'll get ECHILD out of waitpid2 I think*), but the application might not be expecting a random pid it wasn't tracking to fall out of waitpid.

(*) unless there's pid reuse in the meanwhile (which *does* seem highly unlikely but possible)

#### #3 - 01/07/2023 09:16 AM - nobu (Nobuyoshi Nakada)

I'm not very positive to implement platform specific methods, and rather suggest to create a gem as the first step.

#### #4 - 01/07/2023 09:24 AM - kjtsanaktsidis (KJ Tsanaktsidis)

Apologies if I wasn't clear, but I definitely don't intend for the proposed interface to be platform specific. It would make use of clone/pdfork if they were available, but the fallback implementation (either the "proxy everything through an intermediate server" one or the "trap all calls to waitpid2 and lie about them" one) would be used elsewhere.

On all platforms the observed behaviour of Process::Handle should be the same I think.

I can try an implementation in a gem, the fallback should be possible from a gem by monkey patching the relevant process methods I think

#### #5 - 01/11/2023 07:56 AM - kjtsanaktsidis (KJ Tsanaktsidis)

*- Description updated*

#### #6 - 01/11/2023 08:01 AM - kjtsanaktsidis (KJ Tsanaktsidis)

So I ran into a pretty serious snag when trying to implement my idea with Linux pidfds - you can make a hidden process which is unwaitable & doesn't raise SIGCHLD, but if that process exec's, those special properties go away and it goes back to behaving just like a normal child process.

That led me to look more carefully at Ruby's current handling of SIGCHLD/waitpid, and I think there's room in there to implement the API I proposed without leaning on any new system API's beyond SIGCHLD & waitpid. Plus, my proposal would clean up a bit of special-casing for mjit which is currently floating around inside process.c.

WDYT @nobu (Nobuyoshi Nakada)?

#### #7 - 01/11/2023 07:38 PM - Eregon (Benoit Daloze)

IMHO this sounds like some code is doing bad stuff and not properly caring about its own resources.

In the example you shown, I believe it's none of Unicorn's business to reap arbitrary processes, it doesn't compose (I could be wrong, but this seems a general rule when it come to resources of a program: don't mess with what you don't own).
Unicorn should keep a list of pid subprocesses it created, and only do something on the Signal.trap(:CHLD) do if it's one of these pids.
I think it's a very frequent pattern to track pids of subprocesses and connect to trap handlers.

IMHO the proposed API are way too big and invasive to workaround a bad library which does Process.waitall.

The proposed workaround in https://bugs.ruby-lang.org/issues/19322#note-1 doesn't seem too bad.

#### #8 - 01/11/2023 07:39 PM - Eregon (Benoit Daloze)

Also the hacks for MJIT in process.c are already infamous, let's not add on top of it and force every Ruby implementation to have such complexity please.

#### #9 - 01/12/2023 12:11 AM - Anonymous

"Eregon (Benoit Daloze) via ruby-core" ruby-core@ml.ruby-lang.org wrote:

> IMHO this sounds like some code is doing bad stuff and not
> properly caring about its own resources.
>
> In the example you shown, I believe it's none of Unicorn's
> business to reap arbitrary processes, it doesn't compose (I
> could be wrong, but this seems a general rule when it come to
> resources of a program: don't mess with what you don't own).
> Unicorn should keep a list of pid subprocesses it created, and
> only do something on the Signal.trap(:CHLD) do if it's one
> of these pids.

There'd be lots of zombies if unicorn did what you propose
(at least for non-MJIT-Rubies).

KJ: do you need to care about the exit status?
Or just whether or not a process has exited?

If it's only the latter, turning FD_CLOEXEC off on the write
end of a pipe would let you IO.select/poll/epoll_wait on the
read end to detect when the child+descendents are all dead:

```
r, w = IO.pipe
Process.spawn ..., w => w # share `w' with all descendents
w.close
IO.select([r, ...], ...)
```

I've started using the above pattern in tests for setsid daemons
lately.

---



#### #10 - 01/12/2023 02:20 AM - kjtsanaktsidis (KJ Tsanaktsidis)

> In the example you shown, I believe it's none of Unicorn's business to reap arbitrary processes

Firstly, I do want to note that I don't think this is just a Unicorn problem. This is the "classic unix" way of writing a preforking pool of workers of any kind, and I'm sure similar code exists in many deployed Ruby applications.

There's no way, when responding to a SIGCHLD, to know what child died until *after* you actually reap it and steal its status code. If Unicorn (as an example) wanted to avoid reaping children it does not own, it would need to perform O(N) waitpid system calls to wait on each of its known children and see if they've exited. Alternatively, it could do the pass-down-a-pipe trick that @normalperson (Eric Wong) pointed out above, but then you can't get the exit status.

> it doesn't compose

That's the problem with the entire UNIX process API - it doesn't compose! Subprocesses exiting raise signals that run in other parts of the program, other parts of the program can accidentally wait for your subprocesses and steal the exit status from you, pid re-use means that a gem can't necessarily even *tell* that its subprocess exited (is there now some new process with the same pid? you can't know unless you know for sure you didn't reap the previous one).

The new APIs I've proposed here (spawn_private and fork_private) *do* compose - when a subprocess is created, it can only be reaped by using the unique handle which came from its creation, and not from other random parts of a potentially very large application. My hope is that these APIs let gems (and Ruby itself, e.g. mjit) treat the spawning of subprocesses as an opaque implementation detail.

> let's not add on top of it and force every Ruby implementation to have such complexity please.

I think other ruby implementations already need this complexity if they want threadsafe implementations of things like Process.system. For example, this program will quickly throw an exception under Truffleruby:

```
t1 = Thread.new do
    loop do
        pid, status = Process.waitpid2 -1
        puts "Reaped pid #{pid} status #{status.inspect}"
    rescue Errno::ECHILD
    end
end

t2 = Thread.new do
    loop do
        child_success = system "/bin/sh -c 'exit 1'"
        puts "Child success? #{child_success}"
    end
end

t2.join
t1.join

% ruby bad.rb
Reaped pid 8781 status #<Process::Status: pid 8781 exit 1>
#<Thread:0x158 bad.rb:9 run> terminated with exception (report_on_exception is true):
<internal:core> core/errno.rb:48:in `handle': No child processes - No child process: 8781 (Errno::ECHILD)
 from <internal:core> core/truffle/process_operations.rb:150:in `block in wait'
 from <internal:core> core/truffle/ffi/pointer.rb:255:in `new'
 from <internal:core> core/truffle/process_operations.rb:145:in `wait'
 from <internal:core> core/process.rb:591:in `wait'
 from <internal:core> core/kernel.rb:593:in `system'
 from bad.rb:11:in `block (2 levels) in <main>'
 from <internal:core> core/kernel.rb:407:in `loop'
 from bad.rb:10:in `block in <main>'
<internal:core> core/errno.rb:48:in `handle': No child processes - No child process: 8781 (Errno::ECHILD)
 from <internal:core> core/truffle/process_operations.rb:150:in `block in wait'
 from <internal:core> core/truffle/ffi/pointer.rb:255:in `new'
 from <internal:core> core/truffle/process_operations.rb:145:in `wait'
 from <internal:core> core/process.rb:591:in `wait'
 from <internal:core> core/kernel.rb:593:in `system'
 from bad.rb:11:in `block (2 levels) in <main>'
 from <internal:core> core/kernel.rb:407:in `loop'
 from bad.rb:10:in `block in <main>'
```

It actually works under CRuby, because the direct wait for a specific pid always takes precedence over the wait on -1, and there is no interrupt check between when the child process is spawned and when waitpid is called in the system implementation.

> KJ: do you need to care about the exit status?

I doubt I specifically need it for my use-case (the parent/child process already share a socketpair, and the parent would notice if it closed), but I kind of thought Ruby should offer non-hacky APIs for the use-case of "child processes in gems" in general, so I still wrote up my proposalj.

## Summary:

Really, I think there are three ways of looking at this issue:

1. Programs doing waitpid -1 are bad and wrong, nobody should ever do that, if any code in your program does this anywhere, then Ruby should no longer make any guarantees about subprocess management working correctly in the entire process.
2. Programs doing waitpid -1 are widely deployed, it would be good if, when writing gems, there were APIs we could use which offer better isolation and composibility than the classic unix APIs, so that our gems work no matter what their containing processes are doing.
3. Gems should never be spawning child processes anyway.

My thinking on this issue is camp 2. Like it or not (and really, I don't like it), waitpid -1 has been part of the unix way of doing preforking worker pools since approximately forever, and it would be good if programs such programs could use gems without carefully checking whether they spawn any subprocesses in their implementation.

Perhaps some more data needs to be gathered on just how common waitpid -1 actually is? If people think this is something that moves the needle on this discussion, I'm happy to do some research on the topic.

#### #11 - 01/12/2023 04:41 AM - Anonymous

"kjtsanaktsidis (KJ Tsanaktsidis) via ruby-core" ruby-core@ml.ruby-lang.org wrote:

> 1. Programs doing waitpid -1 are widely deployed, it would

> be good if, when writing gems, there were APIs we could use which offer better isolation and composibility than the classic unix APIs, so that our gems work no matter what their containing processes are doing.

> My thinking on this issue is camp 2. Like it or not (and really, I don't like it), waitpid -1 has been part of the unix way of doing preforking worker pools since approximately forever, and it would be good if programs such programs could use gems without carefully checking whether they spawn any subprocesses in their implementation.

Same here. I think the process.c stuff I worked on for MJIT can be extended easily to support registering per-PID callbacks:

```
Process.wait(pid) { |wpid, status| ... }
```

(But I'll let you or somebody else interested implement it)

> Perhaps some more data needs to be gathered on just how common waitpid -1 actually is? If people think this is something that moves the needle on this discussion, I'm happy to do some research on the topic.

Pretty common if Process.waitall exists. Breaking any common use case is unacceptable to me. But I'm of a minority opinion.

---



#### #12 - 01/19/2023 01:08 AM - kjtsanaktsidis (KJ Tsanaktsidis)

Hey @nobu (Nobuyoshi Nakada), @Eregon (Benoit Daloze) - any further thoughts on this?

@nobu (Nobuyoshi Nakada) - I changed the proposal not to depend on any new platform-specific process management APIs, but instead to leverage the existing code for managing process waits in process.c. However, I don't think my idea can be a gem, because it needs to tightly integrate with the implementation of Process.waitpid to make sure calls to Process.waitpid -1 don't steal the exit status of spawned programs.

@Eregon (Benoit Daloze) - I realise that having Process.spawn_private work in other Ruby implementations requires that they keep track of all (ruby-spawned) child processes and deliver the right exit statuses to the right waiters. However, I think if they want Cruby compatible handling of how waitpid -1 and Process.system interact today, they already need to be doing this tracking. Is Cruby-compatible handling of waitpid -1 actually a goal of Truffleruby? If so, I'm happy to try and contribute a patch for Truffleruby to implement this private_spawn stuff there as well (although I have zero experience with Truffleruby!)

#### #13 - 02/06/2023 07:42 PM - Eregon (Benoit Daloze)

I meant to reply to this earlier but could not.

Right, in the SIGCHILD handler it's not possible to know the pid from Ruby's trap (it might be possible with siginfo_t of sigaction() but that's platform-dependent).

The typical way to care about a resource is let the caller both allocate and release it. So things like Process.wait fork {}.
That would not work as-is for Unicorn and similar use cases since it doesn't want to wait for that child process on the main thread.
There are multiple solutions:

- There is Process.detach(pid). That creates one thread per pid, if that's too much overhead one could make their own with WNOHANG and sleep, then it's just 1 extra thread.
- Alternatively, just do the Process.wait fork {} in a thread, and it's even simpler and easier to handle a child process terminating.
- The pipe trick, I suppose this could be used with one pipe per pid, then it's also easy to detect which process is ready to be waited on.
- Isn't one of the main points of process groups to deal with such a case? I guess we'd need to place the forks in a new process group and then we could wait on that whole process group (Process.wait -group). Maybe with an extra child process in between to setup the process group or so.
- Maybe io_uring or similar API can wait for any of multiple processes to terminate? Probably not portable enough though.

> Like it or not (and really, I don't like it), waitpid -1 has been part of the unix way of doing preforking worker pools since approximately forever,

Is there any reason to do it that way, that none of the solutions above addresses?
I suppose that's easier but also more hacky, doesn't compose and breaks other places in the code waiting for processes.

From your summary, I'm for 1, which I see as proper resource management: release what you own/allocated, don't release other resources you don't own.
Process.waitall doesn't compose, Process.wait pid/group composes and works well.

Also this new API wouldn't be adopted before a very long time by the many usages of Kernel#spawn/etc (far more than usages of Process.waitall).

#### #14 - 02/13/2023 01:01 AM - kjtsanaktsidis (KJ Tsanaktsidis)

> Also this new API wouldn't be adopted before a very long time by the many usages of Kernel#spawn/etc (far more than usages of Process.waitall).

This seems like a fairly compelling argument actually. You're right, I want to write some code that I can use safely in Unicorn, but there's probably a large body of existing code calling Process#spawn et al that people are also running inside forking servers and is "mostly working". So maybe I should have a look at solving this on the Unicorn side rather than the Ruby side.

I wonder, then, if we should aim to *deprecate* Process.waitall and friends, given that it can intefere with standard library modules like open3.

- There is Process.detach(pid). That creates one thread per pid, if that's too much overhead one could make their own with WNOHANG and sleep, then it's just 1 extra thread.
- Alternatively, just do the Process.wait fork {} in a thread, and it's even simpler and easier to handle a child process terminating.

I think it'd be nice to avoid having to make threads just to manage subprocesses through blocking API calls if possible

- The pipe trick, I suppose this could be used with one pipe per pid, then it's also easy to detect which process is ready to be waited on.

Could definitely work, and seems like the best solution for Unicorn specifically, all things considered.

- Isn't one of the main points of process groups to deal with such a case? I guess we'd need to place the forks in a new process group and then we could wait on that whole process group (Process.wait -group). Maybe with an extra child process in between to setup the process group or so.

This was actually quite interesting, I had a look at this. The extra child process would need to be a sibling to the processes to be waited on (because the parent can only wait on children, not grandchildren). Then, when forking processes, each fork could call setpgid(2) to move itself into that sibling's process group; then, as you say, Process.wait -group could be used to wait for only those subprocesses that joined the group.

Unfortunately there's a race condition - a fork could crash after forking and before calling setpgid(2), in which case it would never get reaped.

- Maybe io_uring or similar API can wait for any of multiple processes to terminate?

Linux and FreeBSD have pidfds which can do this, and I think WaitForMultipleObjects can do this on Windows, but I couldn't find any equivalent for MacOS or OpenBSD unfortunately.