← Back to all articles

Published on 2024-11-10

Way too many ways to wait on a child process with a timeout

Unix Signals C Linux FreeBSD Illumos MacOS

Table of contents

Windows is not covered at all in this article.

Discussions: /r/programming, HN, Lobsters

I often need to launch a program in the terminal in a retry loop. Maybe because it's flaky, or because it tries to contact a remote service that is not available. A few scenarios:

- ssh to a (re)starting machine.
- psql to a (re)starting database.
- Ensuring that a network service started fine with netcat .
- File system commands over NFS.

It's a common problem, so much so that there are two utilities that I usually reach for:

- timeout from GNU coreutils, which launches a command with a timeout (useful if the command itself does not have a --timeout option).
- eb which runs a command with a certain number of times with an exponential backoff. That's useful to avoid hammering a server with connection attempts for example.

This will all sound familiar to people who develop distributed systems: they have long known that this is best practice to retry

an operation:

- With a timeout (either constant or adaptive).
- A bounded number of times e.g. 10.
- With a waiting time between each retry, either a constant one or a increasing one e.g. with exponential backoff.
- With jitter, although this point also seemed the least important since most of us use non real-time operating systems which introduce some jitter anytime we sleep or wait on something with a timeout. The AWS article makes a point that in highly contended systems, the jitter parameter is very important, but for the scope of this article I'll leave it out.

This is best practice in distributed systems, and we often need to do the same on the command line. But the two aforementioned tools only do that partially:

- `timeout` does not retry.
- `eb` does not have a timeout.

So let's implement our own that does both! As we'll see, it's much less straightforward, and thus more interesting, than I thought. It's a whirlwind tour through Unix deeps. If you're interested in systems programming, Operating Systems, multiplexed I/O, data races, weird historical APIs, and all the ways you can shoot yourself in the foot with just a few system calls, you're in the right place!

What are we building?

I call the tool we are building ueb for: micro exponential backoff. It does up to 10 retries, with a waiting period in between that starts at an arbitrary 128 ms and doubles every retry. The timeout for the subprocess is the same as the sleep time, so that it's adaptive and we give the subprocess a longer and longer time to finish successfully. These numbers would probably be exposed as command line options in a real polished program, but there's no time, what have to demo it:

```
# This returns immediately since it succeeds on the first try.
$ ueb true

# This retries 10 times since the command always fails, waiting
more and more time between each try, and finally returns the last
exit code of the command (1).
$ ueb false

# This retries a few times (~ 4 times), until the waiting time
exceeds the duration of the sub-program. It exits with `0` since
from the POV of our program, the sub-program finally finished in
its alloted time.
$ ueb sleep 1



# Run a program that prints the date and time, and exits with a
random status code, to see how it works.
$ ueb sh -c 'date --iso-8601=ns; export R=$(($RANDOM % 5)); echo
$R; exit $R'
2024-11-10T15:48:49,499172093+01:00
4
2024-11-10T15:48:49,628818472+01:00
3
```

```
2024-11-10T15:48:49,886557676+01:00
4
2024-11-10T15:48:50,400199626+01:00
3
2024-11-10T15:48:51,425937132+01:00
2
2024-11-10T15:48:53,475565645+01:00
2
2024-11-10T15:48:57,573278508+01:00
1
2024-11-10T15:49:05,767338611+01:00
0

# Some more practical examples.
$ ueb ssh <some_ip>
$ ueb createdb my_great_database -h 0.0.0.0 -U postgres
```

If you want to monitor the retries and the sleeps, you can use strace or dtrace:

```
$ strace ueb sleep 1
```

Note that the sub-command should be idempotent, otherwise we might create a given resource twice, or the command might have succeeded right after our timeout triggered but also right before we killed it, so our program thinks it timed out and thus need to be retried. There is this small data race window, which is completely fine if the command is idempotent but will erroneously retry the command to the bitter end otherwise. There is also the case where the sub-command does stuff over the network for example creating a resource, it succeeds, but the ACK is never received due to network issues. The sub-command will think it failed and retry. Again,

fairly standard stuff in distributed systems but I thought it was worth mentioning.

So how do we implement it?

Immediately, we notice something: even though there are a bazillion ways to wait on a child process to finish (`wait`, `wait3`, `wait4`, `waitid`, `waitpid`), none of them take a timeout as an argument. This has sparked numerous questions online (1, 2), with in my opinion unsatisfactory answers. So let's explore this rabbit hole.

We'd like the pseudo-code to be something like:

```
 1  wait_ms := 128
 2
 3  for retry in 0..<10:
 4      child_pid := run_command_in_subprocess(cmd)
 5
 6      ret := wait_for_process_to_finish_with_timeout_ms(child_pid,
wait_ms)
 7      if (did_process_finish_successfully(ret)):
 8          exit(0)
 9
10      // In case of a timeout, we need to kill the child process
and retry.
11      kill(child_pid, SIGKILL)
12
13      // Reap zombie process to avoid a resource leak.
14      waitpid(child_pid)
```

```
15
16      sleep_ms(wait_ms);
17
18      wait_ms *= 2;
19
20  // All retries exhausted, exit with an error code.
21  exit(1)
```

There is a degenerate case where the give command to run is wrong (e.g. typo in the parameters) or the executable does not exist, and our program will happily retry it to the bitter end. But there is solace: this is bounded by the number of retries (10). That's why we do not retry forever.

First way: old-school sigsuspend

That's how timeout from coreutils implements it. This is quite simple on paper:

1. We opt-in to receive a SIGCHLD signal when the child processes finishes with: signal(SIGCHLD, on_chld_signal) where on_chld_signal is a function pointer we provide. Even if the signal handler does not do anything in this case.
2. We schedule a SIGALARM signal with alarm or more preferably setitimer which can take a duration in microseconds whereas alarm can only handle seconds. There's also timer_create/timer_settime which handles nanoseconds. It depends what the OS and hardware support.
3. We wait for either signal with sigsuspend which suspends the

program until a given set of signals arrive.

4. We should not forget to wait on the child process to avoid leaving zombie processes behind.

The reality is grimmer, looking through the timeout implementation:

- We could have inherited any signal mask from our parent so we need to explicitly unblock the signals we are interested in.
- Signals can be sent to a process group we need to handle that case.
- We have to avoid entering a 'signal loop'.
- Our process can be implicitly multi-threaded due to some timer_settime implementations, therefore a SIGALRM signal sent to a process group, can be result in the signal being sent multiple times to a process (I am directly quoting the code comments from the timeout program here).
- When using timer_create, we need to take care of cleaning it up with timer_delete, lest we have a resource leak when retrying.
- The signal handler may be called concurrently and we have to be aware of that.
- Depending on the timer implementation we chose, we are susceptible to clock adjustments for example going back. E.g. setitimer only offers the CLOCK_REALTIME clock option for counting time, which is just the wall clock. We'd like something like CLOCK_MONOTONIC or CLOCK_MONOTONIC_RAW (the latter being Linux specific).

So... I don't *love* this approach:

- I find signals hard. It's basically a global goto to a completely different location.
- A signal handler is forced to use global mutable state, which is better avoided if possible, and it does not play nice with threads.
- Lots of functions are not 'signal-safe', and that has led to security vulnerabilities in the past e.g. in ssh. In short, non-atomic operations are not signal safe because they might be suspended in the middle, thus leaving an inconsistent state behind. Thus, we have to read documentation very carefully to ensure that we only call signal safe functions in our signal handler, and cherry on the cake, that varies from platform to platform, or even between libc versions on the same platform.
- Signals do not compose well with other Unix entities such as file descriptors and sockets. For example, we cannot poll on signals. There are platform specific solutions though, keep on reading.
- Different signals have different default behaviors, and this gets inherited in child processes, so you cannot assume anything in your program and have to be very defensive. Who knows what the parent process, e.g. the shell, set as the signal mask? If you read through the whole implementation of the timeout program, a lot of the code is dedicated to setting signal masks in the parent, forking, immediately changing the signal mask in the child and the parent, etc. Now, I believe modern Unices offer more control than fork() about what signal mask the child should be created with, so maybe it got better. Still, it's a lot of stuff to know.

- They are many libc functions and system calls relating to signals and that's a lot to learn. A non-exhaustive list e.g. on Linux: kill(1), alarm(2), kill(2), pause(2), sigaction(2), signalfd(2), sigpending(2), sigprocmask(2), sigsuspend(2), bsd_signal(3), killpg(3), raise(3), siginterrupt(3), sigqueue(3), sigsetops(3), sigvec(3), sysv_signal(3), signal(7). Oh wait, I forgot sigemptyset(3) and sigaddset(3). And I'm sure I forgot about a few!

So, let's stick with signals for a bit but simplify our current approach.

Second way: sigtimedwait

Wouldn't it be great if we could wait on a signal, say, SIGCHLD, with a timeout? Oh look, a system call that does exactly that *and* is standardized by POSIX 2001. Cool! I am not quite sure why the timeout program does not use it, but we sure as hell can. My only guess would be that they want to support old Unices pre 2001, or non POSIX systems.

Anyways, here's a very straightforward implementation:

```
1  #define _GNU_SOURCE
2  #include <errno.h>
3  #include <signal.h>
4  #include <stdint.h>
5  #include <sys/wait.h>
6  #include <unistd.h>
7
```

```c
void on_sigchld(int sig) { (void)sig; }

int main(int argc, char *argv[]) {
  (void)argc;
  signal(SIGCHLD, on_sigchld);

  uint32_t wait_ms = 128;

  for (int retry = 0; retry < 10; retry += 1) {
    int child_pid = fork();
    if (-1 == child_pid) {
      return errno;
    }

    if (0 == child_pid) { // Child
      argv += 1;
      if (-1 == execvp(argv[0], argv)) {
        return errno;
      }
      __builtin_unreachable();
    }

    sigset_t sigset = {0};
    sigemptyset(&sigset);
    sigaddset(&sigset, SIGCHLD);

    siginfo_t siginfo = {0};
```

```
35
36      struct timespec timeout = {
37          .tv_sec = wait_ms / 1000,
38          .tv_nsec = (wait_ms % 1000) * 1000 * 1000,
39      };
40
41      int sig = sigtimedwait(&sigset, &siginfo, &timeout);
42      if (-1 == sig && EAGAIN != errno) { // Error
43        return errno;
44      }
45      if (-1 != sig) { // Child finished.
46        if (WIFEXITED(siginfo.si_status) && 0 ==
WEXITSTATUS(siginfo.si_status)) {
47          return 0;
48        }
49      }
50
51      if (-1 == kill(child_pid, SIGKILL)) {
52        return errno;
53      }
54
55      if (-1 == wait(NULL)) {
56        return errno;
57      }
58
59      usleep(wait_ms * 1000);
60      wait_ms *= 2;
```

```
61    }
62    return 1;
63  }
```

I like this implementation. It's pretty easy to convince ourselves looking at the code that it is obviously correct, and that's a very important factor for me.

We still have to deal with signals though. Could we reduce their imprint on our code?

Third approach: Self-pipe trick

This is a really nifty, quite well known trick at this point, where we bridge the world of signals with the world of file descriptors with the pipe(2) system call.

Usually, pipes are a form of inter-process communication, and here we do not want to communicate with the child process (since it could be any program, and most programs do not get chatty with their parent process). What we do is: in the signal handler for SIGCHLD , we simply write (anything) to our own pipe. We know this is signal-safe so it's good.

And you know what's cool with pipes? They are simply a file descriptor which we can poll . With a timeout. Nice! Here goes:

```
1  #define _GNU_SOURCE
2  #include <errno.h>
3  #include <poll.h>
```

```c
#include <signal.h>
#include <stdint.h>
#include <sys/wait.h>
#include <unistd.h>

static int pipe_fd[2] = {0};
void on_sigchld(int sig) {
  (void)sig;
  char dummy = 0;
  write(pipe_fd[1], &dummy, 1);
}

int main(int argc, char *argv[]) {
  (void)argc;

  if (-1 == pipe(pipe_fd)) {
    return errno;
  }

  signal(SIGCHLD, on_sigchld);

  uint32_t wait_ms = 128;

  for (int retry = 0; retry < 10; retry += 1) {
    int child_pid = fork();
    if (-1 == child_pid) {
      return errno;
```

```
31      }
32
33      if (0 == child_pid) { // Child
34        argv += 1;
35        if (-1 == execvp(argv[0], argv)) {
36          return errno;
37        }
38        __builtin_unreachable();
39      }
40
41      struct pollfd poll_fd = {
42          .fd = pipe_fd[0],
43          .events = POLLIN,
44      };
45
46      // Wait for the child to finish with a timeout.
47      poll(&poll_fd, 1, (int)wait_ms);
48
49      kill(child_pid, SIGKILL);
50      int status = 0;
51      wait(&status);
52      if (WIFEXITED(status) && 0 == WEXITSTATUS(status)) {
53        return 0;
54      }
55
56      char dummy = 0;
57      read(pipe_fd[0], &dummy, 1);
```

```
58
59      usleep(wait_ms * 1000);
60      wait_ms *= 2;
61    }
62    return 1;
63  }
```

So we still have one signal handler but the rest of our program does not deal with signals in any way (well, except to kill the child when the timeout triggers, but that's invisible).

There are a few catches with this implementation:

- Contrary to sigtimedwait, poll does not give us the exit status of the child, we have to get it with wait. Which is fine.
- In the case that the timeout fired, we kill the child process. However, the child process, being forcefully ended, will result in a SIGCHLD signal being sent to our program. Which will then trigger our signal handler, which will then write a value to the pipe. So we need to unconditionally read from the pipe after killing the child and before retrying. If we only read from the pipe if the child ended by itself, that will result in the pipe and the child process being desynced.
- In some complex programs, we'd have to use ppoll instead of poll. ppoll prevents a set of signals from interrupting the polling. That's to avoid some data races (again, more data races!). Quoting from the man page for pselect which is analogous to ppoll:

> The reason that pselect() is needed is that if one wants to wait for either a signal or for a file descriptor to become ready, then an atomic test is needed to prevent race conditions. (Suppose the signal handler sets a global flag and returns. Then a test of this global flag followed by a call of select() could hang indefinitely if the signal arrived just after the test but just before the call. By contrast, pselect() allows one to first block signals, handle the signals that have come in, then call pselect() with the desired sigmask, avoiding the race.)

So, this trick is clever, but wouldn't it be nice if we could avoid signals *entirely*?

A simpler self-pipe trick

An astute reader pointed out that this trick can be simplified to not deal with signals at all and instead leverage two facts:

- A child inherits the open file descriptors of the parent (including the ones from a pipe)
- When a process exits, the OS automatically closes its file descriptors

Behind the scenes, at the OS level, there is a reference count for a file descriptor shared by multiple processes. It gets decremented

when doing close(fd) or by a process terminating. When this count reaches 0, it is closed for real. And you know what system call can watch for a file descriptor closing? Good old poll !

So the improved approach is as follows:

1. Each retry, we create a new pipe.
2. We fork.
3. The parent closes the write end pipe and the child closes the read end pipe. Effectively, the parent owns the read end and the child owns the write end.
4. The parent polls on the read end.
5. When the child finishes, it automatically closes the write end which in turn triggers an event in poll .
6. We cleanup before retrying (if needed)

So in a way, it's not really a *self*-pipe, it's more precisely a pipe between the parent and the child, and nothing gets written or read, it's just used by the child to signal it's done when it closes its end. Which is a useful approach for many cases outside of our little program.

Here is the code:

```
1  #define _GNU_SOURCE
2  #include <errno.h>
3  #include <poll.h>
4  #include <stdint.h>
5  #include <sys/wait.h>
```

```c
#include <unistd.h>

int main(int argc, char *argv[]) {
  (void)argc;

  uint32_t wait_ms = 128;

  for (int retry = 0; retry < 10; retry += 1) {
    int pipe_fd[2] = {0};
    if (-1 == pipe(pipe_fd)) {
      return errno;
    }

    int child_pid = fork();
    if (-1 == child_pid) {
      return errno;
    }

    if (0 == child_pid) { // Child
      // Close the read end of the pipe.
      close(pipe_fd[0]);

      argv += 1;
      if (-1 == execvp(argv[0], argv)) {
        return errno;
      }
      __builtin_unreachable();
```

```
33      }
34
35      // Close the write end of the pipe.
36      close(pipe_fd[1]);
37
38      struct pollfd poll_fd = {
39          .fd = pipe_fd[0],
40          .events = POLLHUP | POLLIN,
41      };
42
43      // Wait for the child to finish with a timeout.
44      poll(&poll_fd, 1, (int)wait_ms);
45
46      kill(child_pid, SIGKILL);
47      int status = 0;
48      wait(&status);
49      if (WIFEXITED(status) && 0 == WEXITSTATUS(status)) {
50        return 0;
51      }
52
53      close(pipe_fd[0]);
54
55      usleep(wait_ms * 1000);
56      wait_ms *= 2;
57    }
58    return 1;
59  }
```

Voila, no signals and no global state!

Fourth approach: Linux's signalfd

This is a short one: on Linux, there is a system call that does exactly the same as the self-pipe trick: from a signal, it gives us a file descriptor that we can poll. So, we can entirely remove our pipe and signal handler and instead poll the file descriptor that signalfd gives us.

Cool, but also....Was it really necessary to introduce a system call for that? I guess the advantage is clarity.

I would prefer extending poll to support things other than file descriptors, instead of converting everything a file descriptor to be able to use poll.

Ok, next!

Fifth approach: process descriptors

Recommended reading about this topic: 1 and 2.

In the recent years (starting with Linux 5.3 and FreeBSD 9), people realized that process identifiers (pids) have a number of problems:

- PIDs are recycled and the space is small, so collisions will happen. Typically, a process spawns a child process, some work happens, and then the parent decides to send a signal to the PID of the child. But it turns out that the child already terminated

(unbeknownst to the parent) and another process took its place with the same PID. So now the parent is sending signals, or communicating with, a process that it thinks is its original child but is in fact something completely different. Chaos and security issues ensue. Now, in our very simple case, that would not really happen, but perhaps the root user is running our program, or, imagine that you are implementing the init process with PID 1, e.g. systemd: you can kill any process on the machine! Or think of the case of re-parenting a process. Or sending a certain PID to another process and they send a signal to it at some point in the future. It becomes hairy and it's a very real problem.

- Data races are hard to escape (see the previous point).
- It's easy to accidentally send a signal to all processes with `kill(0, SIGKILL)` or `kill(-1, SIGKILL)` if the developer has not checked that all previous operations succeeded. This is a classic mistake:

```
1  int child_pid = fork();  // This fork fails and returns -1.
2  ... // (do not check that fork succeeded);
3  kill(child_pid, SIGKILL); // Effectively: kill(-1, SIGKILL)
```

And the kernel developers have worked hard to introduce a better concept: process descriptors, which are (almost) bog-standard file descriptors, like files or sockets. After all, that's what sparked our whole investigation: we wanted to use `poll` and it did not work on a PID. PIDs and signals do not compose well, but file descriptors do. Also, just like file descriptors, process descriptors are per-process. If I open a file with `open()` and get the file descriptor `3`, it is scoped to my process. Another process can `close(3)` and it will refer to their own file descriptor, and

not affect my file descriptor. That's great, we get isolation, so bugs in our code do not affect other processes.

So, Linux and FreeBSD have introduced the same concepts but with slightly different APIs (unfortunately), and I have no idea about other OSes:

- A child process can be created with clone3(..., CLONE_PIDFD) (Linux) or pdfork() (FreeBSD) which returns a process descriptor which is almost like a normal file descriptor. On Linux, a process descriptor can also be obtained from a PID with pidfd_open(pid) e.g. after a normal fork was done (but there is a risk of a data race in some cases!). Once we have the process descriptor, we do not need the PID anymore.
- We wait on the process descriptor with poll(..., timeout) (or select , or epoll , etc).
- We kill the child process using the process descriptor with pidfd_send_signal (Linux) or close (FreeBSD) or pdkill (FreeBSD).
- We wait on the zombie child process again using the process descriptor to get its exit status.

And voila, no signals! Isolation! Composability! (Almost) No PIDs in our program! Life can be nice sometimes. It's just unfortunate that there isn't a cross-platform API for that.

Here's the Linux implementation:

```
1  #define _GNU_SOURCE
2  #include <errno.h>
```

```c
#include <poll.h>
#include <stdint.h>
#include <sys/syscall.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  (void)argc;

  uint32_t wait_ms = 128;

  for (int retry = 0; retry < 10; retry += 1) {
    int child_pid = fork();
    if (-1 == child_pid) {
      return errno;
    }

    if (0 == child_pid) { // Child
      argv += 1;
      if (-1 == execvp(argv[0], argv)) {
        return errno;
      }
      __builtin_unreachable();
    }

    // Parent.

```

```
30      int child_fd = (int)syscall(SYS_pidfd_open, child_pid, 0);
31      if (-1 == child_fd) {
32        return errno;
33      }
34
35      struct pollfd poll_fd = {
36          .fd = child_fd,
37          .events = POLLHUP | POLLIN,
38      };
39      // Wait for the child to finish with a timeout.
40      if (-1 == poll(&poll_fd, 1, (int)wait_ms)) {
41        return errno;
42      }
43
44      if (-1 == syscall(SYS_pidfd_send_signal, child_fd, SIGKILL,
NULL, 0)) {
45        return errno;
46      }
47
48      siginfo_t siginfo = {0};
49      // Get exit status of child & reap zombie.
50      if (-1 == waitid(P_PIDFD, (id_t)child_fd, &siginfo,
WEXITED)) {
51        return errno;
52      }
53
54      if (WIFEXITED(siginfo.si_status) && 0 ==
WEXITSTATUS(siginfo.si_status)) {
```

```
55        return 0;
56      }
57
58      wait_ms *= 2;
59      usleep(wait_ms * 1000);
60
61      close(child_fd);
62    }
63  }
```

A small note: To poll a process descriptor, Linux wants us to use POLLIN whereas FreeBSD wants us to use POLLHUP. So we use POLLHUP | POLLIN since there are no side-effects to use both.

Another small note: a process descriptor, just like a file descriptor, takes up resources on the kernel side and we can reach some system limits (or even the memory limit), so it's good practice to close it as soon as possible to free up resources. For us, that's right before retrying. On FreeBSD, closing the process descriptor also kills the process, so it's very short, just one system call. On Linux, we need to do both.

Sixth approach: MacOS's and BSD's kqueue

It feels like cheating, but MacOS and the BSDs have had kqueue for decades which works out of the box with PIDs. It's a bit similar to poll or epoll on Linux:

```
1  #include <errno.h>
2  #include <signal.h>
```

```c
#include <stdint.h>
#include <sys/event.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  (void)argc;

  uint32_t wait_ms = 128;
  int queue = kqueuex(KQUEUE_CLOEXEC);

  for (int retry = 0; retry < 10; retry += 1) {
    int child_pid = fork();
    if (-1 == child_pid) {
      return errno;
    }

    if (0 == child_pid) { // Child
      argv += 1;
      if (-1 == execvp(argv[0], argv)) {
        return errno;
      }
      __builtin_unreachable();
    }

    struct kevent change_list = {
        .ident = child_pid,
```

```
30          .filter = EVFILT_PROC,
31          .fflags = NOTE_EXIT,
32          .flags = EV_ADD | EV_CLEAR,
33      };
34
35      struct kevent event_list = {0};
36
37      struct timespec timeout = {
38          .tv_sec = wait_ms / 1000,
39          .tv_nsec = (wait_ms % 1000) * 1000 * 1000,
40      };
41
42      int ret = kevent(queue, &change_list, 1, &event_list, 1,
&timeout);
43      if (-1 == ret) { // Error
44        return errno;
45      }
46      if (1 == ret) { // Child finished.
47        int status = 0;
48        if (-1 == wait(&status)) {
49          return errno;
50        }
51        if (WIFEXITED(status) && 0 == WEXITSTATUS(status)) {
52          return 0;
53        }
54      }
55
```

```
56      kill(child_pid, SIGKILL);
57      wait(NULL);
58
59      change_list = (struct kevent){
60          .ident = child_pid,
61          .filter = EVFILT_PROC,
62          .fflags = NOTE_EXIT,
63          .flags = EV_DELETE,
64      };
65      kevent(queue, &change_list, 1, NULL, 0, NULL);
66
67      usleep(wait_ms * 1000);
68      wait_ms *= 2;
69    }
70    return 1;
71  }
```

The only surprising thing, perhaps, is that a kqueue is stateful, so once the child process exited by itself or was killed, we have to remove the watcher on its PID, since the next time we spawn a child process, the PID will very likely be different. kqueue offers the flag EV_ONESHOT, which automatically deletes the event from the queue once it has been consumed by us. However, it would not help in all cases: if the timeout triggers, no event was consumed, and we have to kill the child process, which creates an event in the queue! So we have to always consume/delete the event from the queue right before we retry, with a second kevent call. That's the same situation as with the self-pipe approach where we unconditionally read from the pipe to 'clear' it before retrying.

I love that kqueue works with every kind of Unix entity: file descriptor, pipes, PIDs, Vnodes, sockets, etc. Even signals! However, I am not sure that I love its statefulness. I find the poll API simpler, since it's stateless. But perhaps this behavior is necessary for some corner cases or for performance to avoid the linear scanning that poll entails? It's interesting to observe that Linux's epoll went the same route as kqueue with a similar API, however, epoll can only watch plain file descriptors.

A parenthesis: libkqueue

kqueue is only for MacOS and BSDs....Or is it?

There is this library, libkqueue, that acts as a compatibility layer to be able to use kqueue on all major operating systems, mainly Windows, Linux, and even Solaris/Illumos!

So...How do they do it then? How can we, on an OS like Linux, watch a PID with the kqueue API, when the OS does not support that functionality (neither with poll or epoll)? Well, the solution is actually very simple:

- On Linux 5.3+, they use pidfd_open + poll/epoll. Hey, we just did that a few sections above!
- On older versions of Linux, they handle the signals, like GNU's timeout. It has a number of known shortcomings which is testament to the hardships of using signals. To just quote one piece:

> Because the Linux kernel coalesces SIGCHLD (and other signals), the only way to reliably determine if a monitored process has exited, is to loop through all PIDs registered by any kqueue when we receive a SIGCHLD. This involves many calls to waitid(2) and may have a negative performance impact.

Another parenthesis: Solaris/Illumos's ports

So, if it was not enough that each major OS has its own way to watch many different kinds of entities (Windows has its own thing called I/O completion ports, MacOS & BSDs have kqueue , Linux has epoll), Solaris/Illumos shows up and says: Watch me do my own thing. Well actually I do not know the chronology, they might in fact have been first, and some Illumos kernel developers (namely Brian Cantrill in the fabulous Cantrillogy) have admitted that it would have been better for everyone if they also had adopted kqueue .

Anyways, their own system is called port (or is it ports?) and it looks so similar to kqueue it's almost painful. And weirdly, they support all the different kinds of entities that kqueue supports *except* PIDs! And I am not sure that they support process descriptors either e.g. pidfd_open . However, they have an extensive compatibility layer for Linux so perhaps they do there.

Seventh approach: Linux's io_uring

io_uring is the last candidate to enter the already packed ring (eh) of different-yet-similar ways to do 'I/O multiplexing', meaning to wait with a timeout on various kinds of entities to do interesting 'stuff'. We queue a system call e.g. wait, as well as a timeout, and we wait for either to complete. If wait completed first and the exit status is a success, we exit. Otherwise, we retry. Familiar stuff at this point. io_uring essentially makes every system call asynchronous with a uniform API. That's exactly what we want! io_uring only exposes waitid and only in very recent versions, which is completely fine.

Incidentally, this approach is exactly what liburing does in a unit test.

Alternatively, we can only queue the waitid and use io_uring_wait_cqe_timeout to mimick poll(..., timeout):

```c
#define _DEFAULT_SOURCE
#include <liburing.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  (void)argc;

  struct io_uring ring = {0};
  if (io_uring_queue_init(2, &ring,
                          IORING_SETUP_SINGLE_ISSUER |
                              IORING_SETUP_DEFER_TASKRUN) < 0)
{
```

```
13      return 1;
14    }
15
16    uint32_t wait_ms = 128;
17
18    for (int retry = 0; retry < 10; retry += 1) {
19      int child_pid = fork();
20      if (-1 == child_pid) {
21        return errno;
22      }
23
24      if (0 == child_pid) { // Child
25        argv += 1;
26        if (-1 == execvp(argv[0], argv)) {
27          return errno;
28        }
29        __builtin_unreachable();
30      }
31
32      struct io_uring_sqe *sqe = NULL;
33
34      // Queue `waitid`.
35      sqe = io_uring_get_sqe(&ring);
36      siginfo_t si = {0};
37      io_uring_prep_waitid(sqe, P_PID, (id_t)child_pid, &si,
WEXITED, 0);
38      sqe->user_data = 1;
```

```
39
40      io_uring_submit(&ring);
41
42      struct __kernel_timespec ts = {
43          .tv_sec = wait_ms / 1000,
44          .tv_nsec = (wait_ms % 1000) * 1000 * 1000,
45      };
46      struct io_uring_cqe *cqe = NULL;
47
48      int ret = io_uring_wait_cqe_timeout(&ring, &cqe, &ts);
49
50      // If child exited successfully: the end.
51      if (ret == 0 && cqe->res >= 0 && cqe->user_data == 1 &&
52          WIFEXITED(si.si_status) && 0 ==
WEXITSTATUS(si.si_status)) {
53        return 0;
54      }
55      if (ret == 0) {
56        io_uring_cqe_seen(&ring, cqe);
57      } else {
58        kill(child_pid, SIGKILL);
59        // Drain the CQE.
60        ret = io_uring_wait_cqe(&ring, &cqe);
61        io_uring_cqe_seen(&ring, cqe);
62      }
63
64      wait(NULL);
```

```
65
66      wait_ms *= 2;
67      usleep(wait_ms * 1000);
68    }
69    return 1;
70  }
```

The only difficulty here is in case of timeout: we kill the child directly, and we need to consume and discard the waitid entry in the completion queue. Just like kqueue.

One caveat for io_uring: it's only supported on modern kernels (5.1+).

Another caveat: some cloud providers e.g. Google Cloud disable io_uring due to security concerns when running untrusted code. So it's not ubiquitous.

Eigth approach: Threads

Readers have pointed out that threads are also a solution, albeit a suboptimal one. Here's the approach:

1. Spawn a thread, it will be in charge of spawning the child process, storing the child PID in a global thread-safe variable (e.g. protected by a mutex). It then waits on the child in a blocking way.
2. If the child exits, wait will return the status, which is also written in a global thread-safe variable, and the thread ends.

3. In the main thread, wait on the other thread with a timeout, e.g. with pthread_timedjoin_np .
4. If the child did not exit successfully, this is the same as usual: kill, wait, sleep, and retry.

If the threads library supports returning a value from a thread, like pthread or C11 threads do, that could be used to return the exit status of the child to simplify the code a bit.

Also, we could make the thread spawning logic a bit more efficient by not spawning a new thread for each retry, if we wanted to. Instead, we communicate with the other thread with a queue or such to instruct it to spawn the child again. It's more complex though.

Now, this approach works but is kind of cumbersome (as noted by the readers), because threads interact in surprising ways with signals (yay, another thing to watch out for!) so we may have to set up signal masks to block/ignore some, and we must take care of not introducing data-races due to the global variables.

Unless the problem is embarassingly parallel and the threads share nothing (e.g.: dividing an array into pieces and each thread gets its own piece to work on), I am reminded of the adage: "You had two problems. You reach out for X. You now have 3 problems". And threads are often the X.

Still, it's a useful tool in the toolbox.

Nineth approach: Active polling.

That's looping in user code with micro-sleeping to actively poll on the child status in a non-blocking way, for example using wait(..., WNOHANG). Unless you have a very bizzare use case and you know what you are doing, please do not do this. This is unnecessary, bad for power consumption, and all we achieve is noticing late that the child ended. This approach is just here for completeness.

Conclusion

I find signals and spawning child process to be the hardest parts of Unix. Evidently this is not a rare opinion, looking at the development in these areas: process descriptors, the various expansions to the venerable fork with vfork, clone, clone3, clone6, a bazillion different ways to do I/O multiplexing, etc.

So what's the best approach then in a complex program? Let's recap:

- If you need maximum portability and are a Unix wizard, you can use sigsuspend.
- If you are not afraid of signals, want a simpler API that still widely supported, and the use case is very specific (like ours), you can use sigtimedwait.
- If you favor correctness and work with recent Linux and FreeBSD versions, you can use process descriptors with shims to get the same API on both OSes. That's probably my favorite option if it's applicable.
- If you only care about MacOS and BSDs (or accept to use libkqueue on Linux), you can use kqueue because it works out of the box with PIDs, you avoid signals completely, and it's used in all the big libraries out of there e.g. libuv.

- If you only care about bleeding edge Linux, are already using `io_uring` in your code, and are bold enough to add `wait` support to `io_uring`, you can use `io_uring` (once you have merged it in mainline Linux!).
- If you only care about Linux and are afraid of using `io_uring`, you can use `signalfd` + `poll`.

I often look at complex code and think: what are the chances that this is correct? What are the chances that I missed something? Is there a way to make it simplistic that it is obviously correct? And how can I limit the blast of a bug I wrote? Will I understand this code in 3 months? When dealing with signals, I was constantly finding weird corner cases and timing issues leading to data races. You would not believe how many times I got my system completely frozen while writing this article, because I accidentally fork-bombed myself or simply forgot to reap zombie processes.

And to be fair to the OS developers that have to implement them: I do not think they did a bad job! I am sure it's super hard to implement! It's just that the whole concept and the available APIs are very easy to misuse. It's a good illustration of how a good API, the right abstraction, can enable great programs, and a poor API, the wrong abstraction, can be the root cause of various bugs in many programs for decades.

And OS developers have noticed and are working on new, better abstractions!

Process descriptors seem to me so straightforward, so obviously correct, that I would definitely favor them over signals. They simply remove entire classes of bugs. If these are not available to

me, I would perhaps use kqueue instead (with libkqueue emulation when necessary), because it means my program can be extended easily to watch for over types of entities and I like that the API is very straightforward: one call to create the queue and one call to use it.

Finally, I regret that there is so much fragmentation across all operating systems. Perhaps io_uring will become more than a Linuxism and spread to Windows, MacOS, the BSDs, and Illumos in the future?

Addendum: The code

The code is available here. It does not have any dependencies except libc (well, and libkqueue for kqueue.c). All of these programs are in the worst case 27 KiB in size, with debug symbols enabled and linking statically to musl. They do not allocate any memory themselves. For comparison, eb has 24 dependencies and is 1.2 MiB! That's roughly 50x times more.

Back to all articles

This blog is open-source! If you find a problem, please open a Github issue. The content of this blog as well as the code snippets are under the BSD-3 License which I also usually use for all my personal projects. It's basically free for every use but you have to mention me as the original author.