This is a short story about a non-trivial bug in the processx package,  
and how I fixed it. It is a good showcase of the some debugging tools.

**The bug**

processx is an R package to start and manage  
external processes. It is used by the callr  
package to run code in another R session.

> fun <- function() {

> parallel::mclapply(1:2, function(x) x)

> }

> env <- c(

> callr::rcmd\_safe\_env(),

> PROCESSX\_NOTIFY\_OLD\_SIGCHLD = "true"

> )

> tmp <- callr::r(fun, env = env, show = TRUE)

> #> Error while shutting down parallel: unable to terminate some child processes

**A red herring**

The error message is exactly the same there, but the reasons are very  
different. That issue is about an interference between processx and the  
parallel package. This one must be something else, because callr does not  
load processx (or any other R package) in the R subprocess it creates:

callr::r(function() loadedNamespaces())

## [1] "compiler" "graphics" "utils" "grDevices"

## [5] "stats" "datasets" "methods" "base"

**About SIGCHLD signals**

To understand what is going on here, we need to know a bit about signals  
and subprocesses. A signal is an asynchronous notification, sent to a  
process by a(nother) process or the operating system. A signal means  
that an important event has happened, and the process’s normal execution  
is interrupted. For some signal types, the process gets a chance to handle  
the signal gracefully and then continue execution. The SIGCHLD signal is  
such a signal. It is sent by the operating system when a subprocess of  
the (parent) process has finished its execution. When the parent process  
receives SIGCHLD, the subprocess has already finished, but it is still  
in the OS’s process table. It is dead but still hanging around:

To completely eliminate the subprocess, the parent process needs to either  
read out its exit status, or tell the OS that it is not interested in the  
exit status. The parent process can also pre-emptively tell the OS that  
it is not interested in the exit status of its subprocesses, and in this  
case no SIGCHLD signals are delivered to it at all.

**Of course it is my bad**

To make sure that this issue is specific to processx or callr, I tried if  
a subprocess started with system() has the same issue, and it does not:

system("R -q -e 'parallel::mclapply(1:2, function(x) x)'")

Whereas processx/callr does:

tmp <- callr::r(

function() parallel::mclapply(1:2, function(x) x),

show = TRUE

)

## Error while shutting down parallel: unable to terminate some child processes

Still, it is possible (however unlikely ) that the bug  
is in the parallel package, e.g. because it does not set up the signal  
handler for SIGCHLD properly, when the R session was started by  
processx/callr.

The parallel package sets up a  
finalizer script.  
This finalizer tries to eliminate all subprocesses that were started by  
parallel, and if it fails to do that, it emits the error message that we  
see above.

When eliminating subprocesses, parallel sends them a SIGKILL signal, which  
is not possible to catch and handle, so it is sure that their execution  
has finished, and they are in a zombie state. In fact, mclapply() already  
tries to clean up the subprocesses it has started, so they are probably  
already in the zombie state when mclapply() returns. This is  
easy to check with the ps package:

tmp <- callr::r(

function() {

parallel::mclapply(1:2, function(x) x)

print(sapply(ps::ps\_children(ps::ps\_handle()), ps::ps\_status))

},

show = TRUE

)

## [1] "zombie" "zombie"

## Error while shutting down parallel: unable to terminate some child processes

Indeed, both subprocesses of the callr R process are zombies.  
Clearly, the callr R process did not receive or did not handle their  
SIGCHLD signals. To make sure that the SIGCHLD signal handler is  
properly set up in parallel, we need to debug parallel’s C code, in the  
callr subprocess.

We will use the lldb debugger here, as that is  
the default on macOS. gdb has similar functionality, you probably want to  
use gdb for gcc and Linux.

To debug the callr subprocess, we tell lldb to wait for a process called R:

❯ lldb -w -n R

(lldb) process attach --name "R" --waitfor

Then in another terminal, we run callr::r():

callr::r(function() parallel::mclapply(1:2, function(x) x))

As soon as the callr subprocess starts, lldb will stop it (using a SIGSTOP  
signal that cannot be caught):

❯ lldb -w -n R

(lldb) process attach --name "R" --waitfor

Process 4196 stopped

\* thread #1, queue = 'com.apple.main-thread', stop reason = signal SIGSTOP

frame #0: 0x00007fff74b75a4b libsystem\_info.dylib`xdrmem\_getlong\_aligned + 75

libsystem\_info.dylib`xdrmem\_getlong\_aligned:

-> 0x7fff74b75a4b <+75>: callq 0x7fff74b759f0 ; \_OSSwapInt32

0x7fff74b75a50 <+80>: movq -0x18(%rbp), %rdx

0x7fff74b75a54 <+84>: movl %eax, (%rdx)

0x7fff74b75a56 <+86>: movq -0x10(%rbp), %rdx

Target 0: (R) stopped.

Executable module set to "/Library/Frameworks/R.framework/Resources/bin/exec/R".

Architecture set to: x86\_64h-apple-macosx-.

(lldb)

(lldb) b mc\_fork

Breakpoint 1: no locations (pending).

WARNING: Unable to resolve breakpoint to any actual locations.

(lldb) b mc\_cleanup

Breakpoint 2: no locations (pending).

WARNING: Unable to resolve breakpoint to any actual locations.

Ideally, we would set a break point on the setup\_sig\_handler() function,  
that sets up the signal handler, but this function is optimized out by  
the compiler. Lets continue running the process:

(lldb) c

Process 4196 resuming

1 location added to breakpoint 1

1 location added to breakpoint 2

Process 4196 stopped

\* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1

frame #0: 0x000000010e89ea10 parallel.so`mc\_fork

parallel.so`mc\_fork:

-> 0x10e89ea10 <+0>: pushq %rbp

0x10e89ea11 <+1>: movq %rsp, %rbp

0x10e89ea14 <+4>: pushq %r15

0x10e89ea16 <+6>: pushq %r14

Target 0: (R) stopped.

(lldb)

lldb stops the process at mc\_fork(). parallel is just about to start a  
subprocess here. We step throught mc\_fork() a bit (there is not easy way  
to step until the end of a function in lldb, AFAICT). We go on until we  
see that sigaction(2) is called, this should set up the signal handler:

(lldb) n

Process 4196 stopped

\* thread #1, queue = 'com.apple.main-thread', stop reason = instruction step over

frame #0: 0x000000010e89eac9 parallel.so`mc\_fork + 185

parallel.so`mc\_fork:

-> 0x10e89eac9 <+185>: callq 0x10e8a089a ; symbol stub for: sigaction

0x10e89eace <+190>: movl $0x80000, -0x58(%rbp) ; imm = 0x80000

0x10e89ead5 <+197>: leaq -0x58(%rbp), %rsi

0x10e89ead9 <+201>: leaq -0x44(%rbp), %rdx

Target 0: (R) stopped.

(lldb) n

Process 4196 stopped

\* thread #1, queue = 'com.apple.main-thread', stop reason = instruction step over

frame #0: 0x000000010e89eace parallel.so`mc\_fork + 190

parallel.so`mc\_fork:

-> 0x10e89eace <+190>: movl $0x80000, -0x58(%rbp) ; imm = 0x80000

0x10e89ead5 <+197>: leaq -0x58(%rbp), %rsi

0x10e89ead9 <+201>: leaq -0x44(%rbp), %rdx

0x10e89eadd <+205>: movl $0x1, %edi

Target 0: (R) stopped.

If you look at the source code, sigaction(2) is called from  
setup\_sig\_handler(), but the compiler inlined that function.  
Now that lldb has resolved the break points to the loaded parallel.so  
shared lib, we can double check that:

(lldb) image lookup -n setup\_sig\_handler

(lldb)

Never mind, sigaction(2) was already called, so the signal handler  
should be set up. Let’s check it. We can call sigaction(2) to query the  
current SIGCHLD handler:

(lldb) call (void\*) malloc(sizeof(struct sigaction)) (void \*) $0 = 0x00007fe6c7e305f0

(lldb) call (int) \_\_sigaction(20, NULL, $0)

(int) $1 = 0

(lldb) p ((struct sigaction \*)$0)->\_\_sigaction\_u (\_\_sigaction\_u) $2 = {

\_\_sa\_handler = 0x000000010e8a0750 (parallel.so`parent\_sig\_handler)

\_\_sa\_sigaction = 0x000000010e8a0750 (parallel.so`parent\_sig\_handler)

}

20 is the number of SIGCHLD, see man signal, and for the (platform  
dependent) structure of struct sigaction see man sigaction.  
In any case, parallel.so’s parent\_sig\_handler is indeed set up properly.  
Now we tell lldb to stop on receiving a SIGCHLD signal:

(lldb) process handle SIGCHLD --notify true --pass true --stop true NAME PASS STOP NOTIFY

=========== ===== ===== ======

SIGCHLD true true true

and we are ready to continue the process. It will stop again at mc\_fork(),  
because we are starting two subprocesses in mclapply(). Then we continue  
again:

(lldb) c

Process 4196 resuming

Process 4196 stopped

\* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1

frame #0: 0x000000010e89ea10 parallel.so`mc\_fork

parallel.so`mc\_fork:

-> 0x10e89ea10 <+0>: pushq %rbp

0x10e89ea11 <+1>: movq %rsp, %rbp

0x10e89ea14 <+4>: pushq %r15

0x10e89ea16 <+6>: pushq %r14

Target 0: (R) stopped.

(lldb) c

Process 4196 resuming

Process 4196 stopped

\* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 2.1

frame #0: 0x000000010e89e680 parallel.so`mc\_cleanup

parallel.so`mc\_cleanup:

-> 0x10e89e680 <+0>: pushq %rbp

0x10e89e681 <+1>: movq %rsp, %rbp

0x10e89e684 <+4>: pushq %r15

0x10e89e686 <+6>: pushq %r14

Target 0: (R) stopped.

(lldb)

Interestingly, we got to mc\_cleanup, even though the SIGCHLD signal(s)  
should have arrived first. From another R session, we can check that the  
subprocesses of parallel are zombies already:

❯ sapply(ps::ps\_children(ps::ps\_handle(4196L)), ps::ps\_status)

[1] "zombie" "zombie"

We can continue still, and hope that the SIGCHLDs will still arrive,  
but they won’t:

(lldb) c

Process 4196 resuming

Process 4196 stopped

\* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 2.1

frame #0: 0x000000010e89e680 parallel.so`mc\_cleanup

parallel.so`mc\_cleanup:

-> 0x10e89e680 <+0>: pushq %rbp

0x10e89e681 <+1>: movq %rsp, %rbp

0x10e89e684 <+4>: pushq %r15

0x10e89e686 <+6>: pushq %r14

Target 0: (R) stopped.

(lldb) c

Process 4196 resuming

Process 4196 exited with status = 0 (0x00000000)

(lldb)

Clearly, the signal handler is properly set up in parallel, but the  
signals are not delivered to the process. No doubt, this is a bug in  
the processx or callr code.

**Some hypotheses**

There are very few possible reasons for the OS not sending out SIGCHLD  
signals. The first is that the parent process explicitly tells the OS that  
it is not interested. This is done by setting the signal handler to  
SIG\_IGN with sigaction(2). This is unlikely to be the case for us,  
since we saw that parallel did set up a signal handler properly. By  
searching the R source code and the source code of processx and callr for  
SIG\_IGN, it is obvious that none of them does this.

The other reason is that the SIGCHLD signal is blocked by the process.  
Blocking a signal means that the process tells the OS, that it is currently  
not ready to process it, and it should be delivered later, when the process  
has unblocked the signal. If SIGCHLD was blocked in the callr subprocess,  
that would be a good explanation for the OS not sending it.

**Validating the hypothesis**

The sigprocmask(2) system call can be used to query or manipulate the  
set of blocked signals. So we can re-run our callr::r() reprex, with  
lldb on the callr subprocess again, and examine the state of the  
SIGCHLD signal:

(lldb) call (void\*) malloc(sizeof(sigset\_t))

(void \*) $3 = 0x00007f92704c4300

(lldb) call (int) sigprocmask(0, NULL, $3)

(int) $4 = 0

(lldb) p (int) sigismember((const sigset\_t\*) $3, 20)

(int) $7 = 1

SIGCHLD is indeed blocked! We got it!

**Fixing the bug**

Clearly, if base::system() works and callr::r() and processx::run()  
do not, that means that processx causes the SIGCHLD to be blocked in the  
subprocess.

When processx starts a subprocess, it first calls fork(2) to create a  
copy of the current process, and then execvp(3) to replace that with  
another executable. fork() creates an identical copy, i.e. the signal  
handlers and the blocked signals are the same in the subprocess.  
execvp(3) resets the signal handlers to their defaults, which is great,  
but apparently, it does not reset the set of blocked signals. So if  
SIGCHLD was blocked when processx called fork(2) then it will be blocked  
in the subprocess as well. Indeed,

Processx starting point:

processx\_\_block\_sigchld();

pid = fork();

if (pid == -1) { /\* ERROR \*/

err = -errno;

if (signal\_pipe[0] >= 0) close(signal\_pipe[0]);

if (signal\_pipe[1] >= 0) close(signal\_pipe[1]);

if (cpty) close(pty\_master\_fd);

processx\_\_unblock\_sigchld();

R\_THROW\_SYSTEM\_ERROR\_CODE(err, "Cannot fork when running '%s'",

ccommand);

}

/\* CHILD \*/

if (pid == 0) {

if (cpty) close(pty\_master\_fd);

processx\_\_child\_init(handle, pipes, num\_connections, ccommand, cargs,

signal\_pipe[1], cstdin, cstdout, cstderr,

pty\_name, cenv, &options, ctree\_id);

R\_THROW\_SYSTEM\_ERROR("Cannot start child process when running '%s'",

ccommand);

}

processx blocks SIGCHLD before forking, for simplicity, so it does not  
have to worry about concurrency. processx\_\_child\_init() does a number  
of things, but it does *not* unblock this signal, so it stays blocked.

processx\_\_child\_init():

/\* CHILD \*/

if (pid == 0) {

/\* LCOV\_EXCL\_START \*/

if (cpty) close(pty\_master\_fd);

processx\_\_unblock\_sigchld();

processx\_\_child\_init(handle, pipes, num\_connections, ccommand, cargs,

signal\_pipe[1], cstdin, cstdout, cstderr,

pty\_name, cenv, &options, ctree\_id);

R\_THROW\_SYSTEM\_ERROR("Cannot start child process when running '%s'",

ccommand);

/\* LCOV\_EXCL\_STOP \*/

}

**Conclusion**

Debugging is hard, but debuggers like lldb help you greatly. It is worth  
to invest some time into learning what they can do: wait for a process to  
attach to, call functions and system calls, catch signals, and a lot more.

The fact that the signal mask is *not* reset at execvp(3) is of course  
documented in the manual page of execve(2) (on macOS). This is the system  
call that execvp(3) uses internally. I guess another lesson is to  
read the manual…