

Package ‘fork’

January 2, 2012

Title R functions for handling multiple processes.

Version 1.2.4

Date 2011-08-26

Author Gregory R. Warnes <greg@random-technologies-llc.com>. Financial support for some aspects of this package provided by Metrum Research Group, LLC <<http://www.metrumrg.com>>.

Description These functions provides simple wrappers around the Unix process management API calls: fork, signal, wait, waitpid, kill, and _exit. This enables construction of programs that utilize and manage multiple concurrent processes.

Maintainer Gregory R. Warnes <greg@warnes.net>

License GPL-2

OS_type unix

Repository CRAN

Date/Publication 2011-09-02 04:54:15

R topics documented:

exit	2
fork	3
getpid	6
handleSIGCLD	7
kill	9
signal	10
signame	11
wait	13

Index	15
--------------	-----------

exit	<i>Exit from a child process</i>
------	----------------------------------

Description

Exit from a child process.

Usage

```
exit(status = 0)
```

Arguments

status Integer status flag. Use 0 for a normal exit.

Details

This function is a shallow wrapper around the Unix "_exit" command, and should be used instead of `quit()` to exit from a process created via `fork`.

Value

None.

Note

The `fork` command automatically sets up an `on_exit` function that calls `exit` before evaluating the `slave` argument, so it is usually not necessary to directly call `exit()`.

Author(s)

Gregory R. Warnes <greg@warnes.net>

References

"_exit" man page

See Also

[fork](#), [getpid](#), [wait](#), [kill](#), [killall](#)

Examples

```
waittest <- function()
{
  pid = fork(NULL)
  if(pid==0)
  {
    cat("Hi, I'm the child process and I need to explicitly call exit().")
    cat("\n\n")
    exit()
  }
  else
  {
    wait(pid)
    cat("Hi, I'm the main process, and I wait()ed until the child process\n")
    cat("finished before introducing myself.\n")
  }
}

waittest()
```

fork

Create a new R process using the Unix 'fork' system call

Description

Create a new R process using the Unix 'fork' system call.

Usage

```
fork(slave)
```

Arguments

`slave` Function to be executed in the new R process. This can be NULL, see details.

Details

This function provides a thin wrapper around the Unix "fork" system call, which will create a new process which is an exact copy of the current R process, including open files and sockets including STDIN and STDOUT.

The `child` parameter should normally contain a function to be executed in the newly created process. This function will be called in the new process, and `exit()` will be called when it returns to terminate the process.

If you wish to explicitly control what happens in the child process, you can pass `child=NULL`, in which case you are responsible for managing all the details.

First, the child process must call `exit()` to terminate instead of `quit`. This is necessary to ensure that temporary files or directories created by the parent process are not removed by the child.

Second, the child process should not attempt to use STDIN to obtain commands since it shares all files open at the time of the fork with the parent. This includes STDIN, consequently, any code following the fork will be obtained *competitively* by both processes. This usually means that neither process will get a consistent picture of the following commands, since which process gets each line will be simply a matter of which process asked first.

The best way to avoid the second problem is to simply pass a function to fork using the `slave` parameter. Another way to accomplish this is to ensure that all code that needs to be executed has already been fed to the interpreter before the fork call occurs. The simplest mechanism to achieve this is to wrap the code containing the fork in a code block using curly brackets (`{ ... }`). This can be a top-level code block, or can be within a loop or function call.

To illustrate, this code is a familiar C idiom for forking will NOT be interpreted properly:

```
pid = fork(slave=NULL)

if(pid==0) {

    cat("Hi from the child process"); exit()

} else {

    cat("Hi from the parent process");

}
```

On the other hand, wrapping this code with curly brackets ensures it IS interpreted properly:

```
{
pid = fork(slave=NULL)
if(pid==0) {
    cat("Hi from the child process\n"); exit()
} else {
    cat("Hi from the parent process\n");
}
}
```

Value

This function returns the process ID of the child process to the parent process. If `slave` is `NULL` the function returns 0 to the child process.

Author(s)

Gregory R. Warnes <greg@warnes.net>

References

'fork' man page

See Also

[getpid](#), [exit](#), [wait](#), [kill](#), [killall](#)

Examples

```
###
# Count from 1 to 10 in a separate process
###

# define the function to do the work
testfun <- function()
{
  cat("Counting in process", getpid(), "\n")
  for(i in 1:10)
  {
    i <<- i+1 # assign into Global environment
    cat("i=",i,"\n")
  }
  cat("Done counting in process", getpid(), "\n")
}

# run normally, the function will change our value of i
i <- 0
testfun()
i

# Run in a separate process, our value of i remains unchanged
i <- 0
{
  pid <- fork(testfun)
  wait(pid) # wait until the child finishes, then display its exit status
}

###
# Use a socket to communicate between two processes. Information
# typed on the console, which is read by the initial process, will be send
# to the child process for display.
###
## Not run:
send <- function()
{
  pid <- getpid()
  con1 <- socketConnection(Sys.info()["nodename"], port = 6011)
  i <- 1
  while(TRUE)
  {
    cat("[",pid,"] ",i,": ",sep="")
    data <- readLines(stdin(), n=1)
    writeLines(data, con=con1)
    if( length(grep("quit", data))>0 )
  }
}
```

```

        break;
        i <- i+1
    }
    close(con1)
}

recieve <- function()
{
    pid <- getpid()
    con2 <- socketConnection(port = 6011, block=TRUE, server=TRUE)
    i <- 1
    while(TRUE)
    {
        data <- readLines(con2, n=1)
        cat("[",pid,"] ",i," ",sep="")
        writeLines(data, stdout())
        if( length(grep("quit", data))>0 )
            break;
        i <- i+1
    }
    close(con2)
}

## Important: if we aren't going to explicitly wait() for the child
## process to exit, we need to set up a dummy signal handler to collect
## (and then throw away) the exit status so that child processes will not
## become zombies when they exit
handleSIGCLD()

# fork off the process
pid <- fork(recieve)

# start sending...
send()

## End(Not run)

## restore SIGCLD signal handling to the previous state
restoreSIGCLD()

```

getpid

Obtain the process id for the current process.

Description

Obtain the process number for the current process.

Usage

```
getpid()
```

Details

This function is a simple wrapper around the Unix "getpid" function call.

Value

Integer process id.

Author(s)

Gregory R. Warnes <greg@warnes.net>

References

Unix "getpid" man page

See Also

[fork](#), [exit](#), [wait](#), [kill](#), [killall](#)

Examples

```
getpid()

## Not run:
for(i in 1:10)
  fork( function() { cat("PID:", getpid(), "\n"); exit()} )

## End(Not run)
```

handleSIGCLD

Establish or remove a dummy handler for SIGCLD signals

Description

Establish handleSIGCLD or remove ignoreSIGCLD a dummy handler for SIGCLD signals

Usage

```
handleSIGCLD()
restoreSIGCLD()
```

Details

The handleSIGCLD function establishes a 'dummy' handler for SIGCLD signals. It accepts signals from exiting child processes created by fork and ignores them. This prevents child processes from becoming 'zombies', which would occur when the parent process does not handle this signal.

The restoreSIGCLD function restores the previous (lack of) signal handler.

Value

No return value.

Note

The SIGCLD handling mechanism implemented by `handleSIGCLD` should prevent zombie process creation on systems derived from both SYSV and BSD UNIX, including Linux, Mac OSX, NetBSD, and Solaris.

Author(s)

Gregory R. Warnes <greg@warnes.net>, with financial support from Metrum Research Group, LLC <http://www.metrumrg.com>.

References

W.R. Stevens and S.A. Rago, *Advanced Programming in the UNIX environment*, 2nd ed. (c) 2005, Pearson Education, pp 308-310.

See Also

[fork](#), [wait](#), [signal](#)

Examples

```
## set up the dummy signal handler
handleSIGCLD()

## Generate 100 child processes
for(i in 1:100)
{
  pid = fork(slave=NULL)
  if(pid==0)
  {
    ## don't do anything useful
    exit()
  }
}

## remove the dummy signal handler
restoreSIGCLD()
```

kill	<i>Send a signal to one or more processes.</i>
------	--

Description

kill sends a signal to a process. killall sends a signal to all processes forked during the current session.

Usage

```
kill(pid, signal = 15)
killall(signal = 15)
```

Arguments

pid	Process ID for the target process
signal	Signal number to send. Defaults to 9 (SIGKILL)

Details

The kill function provides a thin wrapper around the Unix "kill" system call, which sends a signal to the specified process. The killall function sends a signal to all processes which have been forked during the current session.

Refer to the local Unix man pages for details.

Value

kill returns 0 on successful completion, -1 on errors. killall does not return a value.

Author(s)

Gregory R. Warnes <greg@warnes.net>

References

"kill" and "waitpid" man pages

See Also

[getpid](#), [exit](#), [wait](#), [kill](#), [killall](#)

Examples

```
# start a process that just sleeps for 10 seconds
sleepy <- function()
{
  cat("Going to sleep..")
  Sys.sleep(10)
  cat("Woke up!")
}
pid <- fork( sleepy )

# kill the sleeping process
kill(pid)
```

signal

Enable or disable handling of signals

Description

This function allows enabling or disabling handling of the specified signal.

Usage

```
signal(signal, action = c("ignore", "default"))
```

Arguments

signal	Signal handler to manipulate, either as a numeric id or character mnemonic
action	Either "ignore" or "default"

Details

It is occasionally necessary to instruct the R process to ignore certain signals. This function allows changing the status of a signal to either ignore the signal (`SIG\IGN="ignore"`) or to the OS's default handler (`SIG\DFL="default"`)

Value

Nothing of interest.

Note

Be very careful with this function. It can be used to totally confuse the R process.

On systems derived from BSD UNIX, setting the SIGCHLD signal to 'ignore' will allow child processes to exit cleanly without becoming zombies. This also prevents the parent process from checking the exit status of children, since this information is no longer available once child process disappears. On systems derived from SysV UNIX, setting SIGCHLD to 'ignore' has no effect on the status of child processes that exit, and they will still become zombies unless the exit status is checked. See the `handleSIGCLD` function for a mechanism that will prevent child processes from becoming zombies on both SYSV and (hopefully) BSD-derived systems.

Author(s)

Gregory R. Warnes <greg@warnes.net>, with financial support from Metrum Research Group, LLC <http://www.metrumrg.com>.

References

See the unix man page for "signal" for more information

See Also

[signal](#), [fork](#), [handleSIGCLD](#)

Examples

```
## Not run:
## Ignore child termination signals for forked processes NOTE: This
## mechanism only works on UNIX SYSV-derived systems. See Note above.
signal("SIGCHLD","ignore")

# Fork off a child process to say "Hi!".
{
  pid = fork(slave=NULL)
  if(pid==0) {
    # only runs in the child process
    cat("Hi from the child process!\n");
    exit() # force process to die
  }
}

## End(Not run)
```

signame

Obtain information about signals

Description

`signame` looks up signal information by the symbolic signal name. `signal` looks up signal information by the numeric value. `siglist` returns a table of signal information.

Usage

```
signame(val)
sigval(name)
siglist()
```

Arguments

val	Integer signal value.
name	Symbolic signal name (with or without the SIG prefix).

Details

These functions return information stored in a table constructed at compile time.

Value

A list (signame, sigval) or data frame (siglist) containing the components:

name	Symbolic name
val	Integer value
desc	Description

Author(s)

Gregory R. Warnes <greg@warnes.net>

References

Unix "signal" man page.

See Also

[getpid](#), [exit](#), [wait](#), [kill](#), [killall](#)

Examples

```
# look up the numeric value for SIGABT:
sigval("SIGKILL")

# the 'SIG' component can be omitted:
sigval("HUP")

# now look up based on value
signame(9)

# and get a complete table of signal information
siglist()
```

wait	<i>Wait for child process(es) to stop or terminate.</i>
------	---

Description

Wait for child process(es) created using 'fork' command to stop or terminate.

Usage

```
wait(pid, nohang=FALSE, untraced=FALSE)
```

Arguments

pid	integer indicating the set of child processes for which status is requested. If missing or NULL, wait for all child processes.
nohang	Use the WNOHANG flag.
untraced	Use the WUNTRACED flag.

Details

This function provides a thin wrapper around the Unix "wait" and "waitpid" system calls. If pid is missing or NULL call, "wait" is called, otherwise "waitpid" is called.

Refer to the local Unix man pages for details on these system calls and the meanings of the various flags.

Value

A vector of length 2 containing

pid	Process id of a child process
status	Status of the child process.

Refer to the local Unix man pages for details on these system calls and the meanings of the status indicator.

Author(s)

Gregory R. Warnes <greg@warnes.net>

References

"wait" and "waitpid" man pages

See Also

[fork](#), [exit](#), [getpid](#), [kill](#), [killall](#)

Examples

```
waittest <- function()
{
  pid = fork(NULL)
  if(pid==0)
  {
    cat("Hi, I'm the child process and I need to explicitly call exit().")
    cat("\n\n")
    exit()
  }
  else
  {
    wait(pid)
    cat("Hi, I'm the main process, and I wait()ed until the child process\n")
    cat("finished before introducing myself.\n")
  }
}

waittest()
```

Index

*Topic **programming**

- exit, [2](#)
- fork, [3](#)
- getpid, [6](#)
- handleSIGCLD, [7](#)
- kill, [9](#)
- signal, [10](#)
- signame, [11](#)
- wait, [13](#)

exit, [2](#), [5](#), [7](#), [9](#), [12](#), [13](#)

fork, [2](#), [3](#), [7](#), [8](#), [11](#), [13](#)

getpid, [2](#), [5](#), [6](#), [9](#), [12](#), [13](#)

handleSIGCLD, [7](#), [11](#)

kill, [2](#), [5](#), [7](#), [9](#), [9](#), [12](#), [13](#)

killall, [2](#), [5](#), [7](#), [9](#), [12](#), [13](#)

killall (kill), [9](#)

restoreSIGCLD (handleSIGCLD), [7](#)

siglist (signame), [11](#)

signal, [8](#), [10](#)

signame, [11](#)

sigval, [11](#)

sigval (signame), [11](#)

wait, [2](#), [5](#), [7-9](#), [12](#), [13](#)