Process Control

Introduction

Process Control support in PHP implements the Unix style of process creation, program execution, signal handling and process termination. Process Control should not be enabled within a web server environment and unexpected results may happen if any Process Control functions are used within a web server environment.

This documentation is intended to explain the general usage of each of the Process Control functions. For detailed information about Unix process control you are encouraged to consult your systems documentation including fork(2), waitpid(2) and signal(2) or a comprehensive reference such as Advanced Programming in the UNIX Environment by W. Richard Stevens (Addison-Wesley).

PCNTL now uses ticks as the signal handle callback mechanism, which is much faster than the previous mechanism. This change follows the same semantics as using "user ticks". You use the **declare()** statement to specify the locations in your program where callbacks are allowed to occur. This allows you to minimize the overhead of handling asynchronous events. In the past, compiling PHP with pcntl enabled would always incur this overhead, whether or not your script actually used pcntl.

There is one adjustment that all pcntl scripts prior to PHP 4.3.0 must make for them to work which is to either to use **declare()** on a section where you wish to allow callbacks or to just enable it across the entire script using the new global syntax of **declare()**.

Note
This extension is not available on Windows platforms.

Installing/Configuring

Requirements

No external libraries are needed to build this extension.

Installation

Process Control support in PHP is not enabled by default. You have to compile the CGI or CLI version of PHP with *--enable-pcntl* configuration option when compiling PHP to enable Process Control support.

Note
Currently, this module will not function on non-Unix platforms (Windows).

Runtime Configuration

This extension has no configuration directives defined in *php.ini*.

Resource Types

This extension has no resource types defined.

Predefined Constants

The following list of signals are supported by the Process Control functions. Please see your systems signal(7) man page for details of the default behavior of these signals.

WNOHANG (integer)

WUNTRACED (integer)

SIG_IGN (integer)

SIG_DFL (integer)

SIG_ERR (integer)

SIGHUP (integer)

SIGINT (integer)

SIGQUIT (integer)

SIGILL (integer)

SIGTRAP (integer)

SIGABRT (integer)

SIGIOT (integer)

SIGBUS (integer)

SIGFPE (integer)

SIGKILL (integer)

SIGUSR1 (integer)

SIGSEGV (integer)

SIGUSR2 (integer)

SIGPIPE (integer)

SIGALRM (integer)

SIGTERM (integer)

SIGSTKFLT (integer)

SIGCLD (integer)

SIGCHLD (integer)

SIGCONT (integer)

SIGSTOP (integer)

SIGTSTP (integer)

SIGTTIN (integer)

SIGTTOU (integer)

SIGURG (integer)

SIGXCPU (integer)

SIGXFSZ (integer)

SIGVTALRM (integer)

SIGPROF (integer)

SIGWINCH (integer)

SIGPOLL (integer)

SIGIO (integer)

SIGPWR (integer)

SIGSYS (integer)

SIGBABY (integer)

Examples

This example forks off a daemon process with a signal handler.

Example #1 - Process Control Example

```
<?php
declare(ticks=1);

$pid = pcntl_fork();
if ($pid == -1) {
    die("could not fork");
} else if ($pid) {
    exit(); // we are the parent
} else {
    // we are the child
}

// detatch from the controlling terminal
if (posix_setsid() == -1) {
   die("could not detach from terminal");
}

// setup signal handlers
pcntl_signal(SIGTERM, "sig_handler");
pcntl_signal(SIGHUP, "sig_handler");

// loop forever performing tasks
while (1) {

   // do something interesting here

}

function sig_handler($signo)
{

    switch ($signo) {
        case SIGTERM:
            // handle shutdown tasks
            exit;
            break;
        case SIGHUP:
            // handle restart tasks
            break;
        default:
            // handle all other signals
    }

}

?>
```

PCNTL Functions

See Also

A look at the section about POSIX functions may be useful.

pcntl_alarm

pcntl_alarm -- Set an alarm clock for delivery of a signal

Description

int **pcntl_alarm** (int `$seconds`)

Creates a timer that will send a **SIGALRM** signal to the process after the given number of seconds. Any call to pcntl_alarm() will cancel any previously set alarm.

Parameters

seconds
: The number of seconds to wait. If *seconds* is zero, no new alarm is created.

Return Values

Returns the time in seconds that any previously scheduled alarm had remaining before it was to be delivered, or *0* if there was no previously scheduled alarm.

pcntl_exec

pcntl_exec -- Executes specified program in current process space

Description

void **pcntl_exec** (string `$path` [, array `$args` [, array `$envs`]])

Executes the program with the given arguments.

Parameters

`path`
: *`path`* must be the path to a binary executable or a script with a valid path pointing to an executable in the shebang (#!/usr/local/bin/perl for example) as the first line. See your system's man execve(2) page for additional information.

`args`
: *`args`* is an array of argument strings passed to the program.

`envs`
: *`envs`* is an array of strings which are passed as environment to the program. The array is in the format of name => value, the key being the name of the environmental variable and the value being the value of that variable.

Return Values

Returns **FALSE** on error and does not return on success.

pcntl_fork

pcntl_fork -- Forks the currently running process

Description

int **pcntl_fork** (void)

The pcntl_fork() function creates a child process that differs from the parent process only in its PID and PPID. Please see your system's fork(2) man page for specific details as to how fork works on your system.

Return Values

On success, the PID of the child process is returned in the parent's thread of execution, and a 0 is returned in the child's thread of execution. On failure, a -1 will be returned in the parent's context, no child process will be created, and a PHP error is raised.

Examples

Example #2 - pcntl_fork() example

```
<?php

$pid = pcntl_fork();
if ($pid == -1) {
    die('could not fork');
} else if ($pid) {
    // we are the parent
    pcntl_wait($status); //Protect against Zombie children
} else {
    // we are the child
}

?>
```

See Also

- pcntl_waitpid()
- pcntl_signal()

pcntl_getpriority

pcntl_getpriority -- Get the priority of any process

Description

int **pcntl_getpriority** ([int `$pid` [, int `$process_identifier`]])

pcntl_getpriority() gets the priority of *`pid`*. Because priority levels can differ between system types and kernel versions, please see your system's getpriority(2) man page for specific details.

Parameters

`pid`
: If not specified, the pid of the current process is used.

`process_identifier`
: One of **PRIO_PGRP**, **PRIO_USER** or **PRIO_PROCESS**.

Return Values

pcntl_getpriority() returns the priority of the process or **FALSE** on error. A lower numerical value causes more favorable scheduling.

Warning
This function may return Boolean **FALSE**, but may also return a non-Boolean value which evaluates to **FALSE**, such as *0* or "". Please read the section on Booleans for more information. Use the === operator for testing the return value of this function.

See Also

- pcntl_setpriority()

pcntl_setpriority

pcntl_setpriority -- Change the priority of any process

Description

bool **pcntl_setpriority** (int `$priority` [, int `$pid` [, int `$process_identifier`]])

pcntl_setpriority() sets the priority of *`pid`*.

Parameters

`priority`
: *`priority`* is generally a value in the range *-20* to *20*. The default priority is *0* while a lower numerical value causes more favorable scheduling. Because priority levels can differ between system types and kernel versions, please see your system's setpriority(2) man page for specific details.

`pid`
: If not specified, the pid of the current process is used.

`process_identifier`
: One of **PRIO_PGRP**, **PRIO_USER** or **PRIO_PROCESS**.

Return Values

Returns **TRUE** on success or **FALSE** on failure.

See Also

- pcntl_getpriority()

pcntl_signal

pcntl_signal -- Installs a signal handler

Description

bool **pcntl_signal** (int `$signo`, callback `$handler` [, bool `$restart_syscalls`])

The pcntl_signal() function installs a new signal handler for the signal indicated by *`signo`*.

Parameters

`signo`
The signal number.

`handler`
The signal handler which may be the name of a user created function, or method, or either of the two global constants **SIG_IGN** or **SIG_DFL**.

Note
Note that when you set a handler to an object method, that object's reference count is increased which makes it persist until you either change the handler to something else, or your script ends.

`restart_syscalls`
Specifies whether system call restarting should be used when this signal arrives and defaults to **TRUE**.

Return Values

Returns **TRUE** on success or **FALSE** on failure.

ChangeLog

Version	Description
4.3.0	The *`restart_syscalls`* parameter was added.
4.3.0	The ability to use an object method as a callback became available.

4.3.0	As of PHP 4.3.0 PCNTL uses ticks as the signal handle callback mechanism, which is much faster than the previous mechanism. This change follows the same semantics as using " user ticks ". You must use the declare() statement to specify the locations in your program where callbacks are allowed to occur for the signal handler to function properly (as used in the above example).

Examples

Example #3 - pcntl_signal() example

```
<?php
// tick use required as of PHP 4.3.0
declare(ticks = 1);

// signal handler function
function sig_handler($signo)
{

    switch ($signo) {
        case SIGTERM:
            // handle shutdown tasks
            exit;
            break;
        case SIGHUP:
            // handle restart tasks
            break;
        case SIGUSR1:
            echo "Caught SIGUSR1...\n";
            break;
        default:
            // handle all other signals
    }

}

echo "Installing signal handler...\n";

// setup signal handlers
pcntl_signal(SIGTERM, "sig_handler");
pcntl_signal(SIGHUP,  "sig_handler");
pcntl_signal(SIGUSR1, "sig_handler");

// or use an object, available as of PHP 4.3.0
// pcntl_signal(SIGUSR1, array($obj, "do_something");

echo"Generating signal SIGTERM to self...\n";

// send SIGUSR1 to current process id
```

```
posix_kill(posix_getpid(), SIGUSR1);

echo "Done\n"

?>
```

See Also

- pcntl_fork()
- pcntl_waitpid()

pcntl_wait

pcntl_wait -- Waits on or returns the status of a forked child

Description

int **pcntl_wait** (int `&$status` [, int `$options`])

The wait function suspends execution of the current process until a child has exited, or until a signal is delivered whose action is to terminate the current process or to call a signal handling function. If a child has already exited by the time of the call (a so-called "zombie" process), the function returns immediately. Any system resources used by the child are freed. Please see your system's wait(2) man page for specific details as to how wait works on your system.

Note
This function is equivalent to calling pcntl_waitpid() with a *-1* `pid` and no `options`.

Parameters

`status`
: pcntl_wait() will store status information in the `status` parameter which can be evaluated using the following functions: pcntl_wifexited(), pcntl_wifstopped(), pcntl_wifsignaled(), pcntl_wexitstatus(), pcntl_wtermsig() and pcntl_wstopsig().

`options`
: If wait3 is available on your system (mostly BSD-style systems), you can provide the optional `options` parameter. If this parameter is not provided, wait will be used for the system call. If wait3 is not available, providing a value for `options` will have no effect. The value of `options` is the value of zero or more of the following two constants *OR* 'ed together:

Possible values for `options`

WNOHANG	Return immediately if no child has exited.
WUNTRACED	Return for children which are stopped, and whose status has not been reported.

Return Values

pcntl_wait() returns the process ID of the child which exited, -1 on error or zero if WNOHANG

was provided as an option (on wait3-available systems) and no child was available.

See Also

- pcntl_fork()
- pcntl_signal()
- pcntl_wifexited()
- pcntl_wifstopped()
- pcntl_wifsignaled()
- pcntl_wexitstatus()
- pcntl_wtermsig()
- pcntl_wstopsig()
- pcntl_waitpid()

pcntl_waitpid

pcntl_waitpid -- Waits on or returns the status of a forked child

Description

int **pcntl_waitpid** (int `$pid`, int `&$status` [, int `$options`])

Suspends execution of the current process until a child as specified by the *`pid`* argument has exited, or until a signal is delivered whose action is to terminate the current process or to call a signal handling function.

If a child as requested by *`pid`* has already exited by the time of the call (a so-called "zombie" process), the function returns immediately. Any system resources used by the child are freed. Please see your system's waitpid(2) man page for specific details as to how waitpid works on your system.

Parameters

`pid`

The value of *`pid`* can be one of the following:

possible values for *`pid`*

< -1	wait for any child process whose process group ID is equal to the absolute value of *`pid`*.
-1	wait for any child process; this is the same behaviour that the wait function exhibits.
0	wait for any child process whose process group ID is equal to that of the calling process.
> 0	wait for the child whose process ID is equal to the value of *`pid`*.

Note
Specifying *-1* as the *`pid`* is equivalent to the functionality pcntl_wait() provides (minus *`options`*).

`status`

pcntl_waitpid() will store status information in the *status* parameter which can be evaluated using the following functions: pcntl_wifexited(), pcntl_wifstopped(), pcntl_wifsignaled(), pcntl_wexitstatus(), pcntl_wtermsig() and pcntl_wstopsig().

options

The value of *options* is the value of zero or more of the following two global constants *OR* 'ed together:

possible values for *options*

WNOHANG	return immediately if no child has exited.
WUNTRACED	return for children which are stopped, and whose status has not been reported.

Return Values

pcntl_waitpid() returns the process ID of the child which exited, -1 on error or zero if WNOHANG was used and no child was available

See Also

- pcntl_fork()
- pcntl_signal()
- pcntl_wifexited()
- pcntl_wifstopped()
- pcntl_wifsignaled()
- pcntl_wexitstatus()
- pcntl_wtermsig()
- pcntl_wstopsig()

pcntl_wexitstatus

pcntl_wexitstatus -- Returns the return code of a terminated child

Description

int **pcntl_wexitstatus** (int `$status`)

Returns the return code of a terminated child. This function is only useful if pcntl_wifexited() returned **TRUE**.

Parameters

`status`
The *`status`* parameter is the status parameter supplied to a successful call to pcntl_waitpid().

Return Values

Returns the return code, as an integer.

See Also

- pcntl_waitpid()
- pcntl_wifexited()

pcntl_wifexited

pcntl_wifexited -- Checks if status code represents a normal exit

Description

bool **pcntl_wifexited** (int `$status`)

Checks whether the child status code represents a normal exit.

Parameters

`status`
: The *`status`* parameter is the status parameter supplied to a successful call to pcntl_waitpid().

Return Values

Returns **TRUE** if the child status code represents a normal exit, **FALSE** otherwise.

See Also

- pcntl_waitpid()
- pcntl_wexitstatus()

pcntl_wifsignaled

pcntl_wifsignaled -- Checks whether the status code represents a termination due to a signal

Description

bool **pcntl_wifsignaled** (int `$status`)

Checks whether the child process exited because of a signal which was not caught.

Parameters

`status`
: The *`status`* parameter is the status parameter supplied to a successful call to pcntl_waitpid().

Return Values

Returns **TRUE** if the child process exited because of a signal which was not caught, **FALSE** otherwise.

See Also

- pcntl_waitpid()
- pcntl_signal()

pcntl_wifstopped

pcntl_wifstopped -- Checks whether the child process is currently stopped

Description

bool **pcntl_wifstopped** (int `$status`)

Checks whether the child process which caused the return is currently stopped; this is only possible if the call to pcntl_waitpid() was done using the option *WUNTRACED*.

Parameters

`status`
The *`status`* parameter is the status parameter supplied to a successful call to pcntl_waitpid().

Return Values

Returns **TRUE** if the child process which caused the return is currently stopped, **FALSE** otherwise.

See Also

- pcntl_waitpid()

pcntl_wstopsig

pcntl_wstopsig -- Returns the signal which caused the child to stop

Description

int **pcntl_wstopsig** (int `$status`)

Returns the number of the signal which caused the child to stop. This function is only useful if pcntl_wifstopped() returned **TRUE**.

Parameters

`status`
The *`status`* parameter is the status parameter supplied to a successful call to pcntl_waitpid().

Return Values

Returns the signal number.

See Also

- pcntl_waitpid()
- pcntl_wifstopped()

pcntl_wtermsig

pcntl_wtermsig -- Returns the signal which caused the child to terminate

Description

int **pcntl_wtermsig** (int `$status`)

Returns the number of the signal that caused the child process to terminate. This function is only useful if pcntl_wifsignaled() returned **TRUE**.

Parameters

`status`
The *`status`* parameter is the status parameter supplied to a successful call to pcntl_waitpid().

Return Values

Returns the signal number, as an integer.

See Also

- pcntl_waitpid()
- pcntl_signal()
- pcntl_wifsignaled()