

\$SPAD/src/lib bsdsignal.c

The Axiom Team

July 31, 2014

Abstract

Contents

1	Executive Overview	3
2	Signals	3
3	MAC OSX and BSD platform change	6
4	License	8

1 Executive Overview

2 Signals

The system defines a set of signals that may be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt: the signal is normally blocked from further occurrence, the current process context is saved, and a new one is built. A process may specify a *handler* to which a signal is delivered, or specify that a signal is to be *ignored*. A process may also specify that a default action is to be taken by the system when a signal occurs. A signal may also be *blocked*, in which case its delivery is postponed until it is *unblocked*. The action to be taken on delivery is determined at the time of delivery. Normally, signal handlers execute on the current stack of the process. This may be changed, on a per-handler basis, so that signals are taken on a special *signal stack*.

Signal routines normally execute with the signal that caused their invocation *blocked*, but other signals may yet occur. A global *signal mask* defines the set of signals currently blocked from delivery to a process. The signal mask for a process is initialized from that of its parent (normally empty). It may be changed with a **sigprocmask(2)** call, or when a signal is delivered to the process.

When a signal condition arises for a process, the signal is added to a set of signals pending for the process. If the signal is not currently *blocked* by the process then it is delivered to the process. Signals may be delivered any time a process enters the operating system (e.g., during a system call, page fault or trap, or clock interrupt). If multiple signals are ready to be delivered at the same time, any signals that could be caused by traps are delivered first. Additional signals may be processed at the same time, with each appearing to interrupt the handlers for the previous signals before their first instructions. The set of pending signals is retuned by the **sigpending(2)** system call. When a caught signal is delivered, the current state of the process is saved, a new signal mask is calculated (as described below), and the signal handler is invoked. The call to the handler is arranged so that if the signal handling routine returns normally the process will resume execution in the context from before the signal's delivery. If the process wishes to resume in a different context, then it must arrange to restore the previous context itself.

When a signal is delivered to a process a new signal mask is installed for the duration of the process's signal handler (or until a **sigprocmask(2)** system call is made). This mask is formed by taking the union of the current signal mask set, the signal to be delivered, and the signal mask associated with the handler to be invoked.

The **sigaction()** system call assigns an action for a signal specified by *sig*. If *act* is non-zero, it specifies an action (SIG_DFL, SIG_IGN, or a handler routine) and mask to be used when delivering the specified signal. If *oact* is non-zero, the previous handling information for the signal is returned to the user.

Once a signal handler is installed, it normally remains installed until another **sigaction()** system call is made, or an **execve(2)** is performed. A signal-specific

default action may be reset by setting *sa_handler* to SIG_DFL. The defaults are process termination, possibly with core dump; no action; stopping the process; or continuing the process. See the signal list below for each signal's default action. If *sa_handler* is SIG_DFL, the default action for the signal is to discard the signal, and if a signal is pending, the pending signal is discarded even if the signal is masked. If *sa_handler* is set to SIG_IGN current and pending instances of the signal are ignored and discarded.

Options may be specified by setting *sa_flags*. The meaning of the various bits is as follows:

- **SA_NOCLDSTOP** If this bit is set when installing a catching function for the SIGCHLD signal, the SIGCHLD signal will be generated only when a child process exits, not when a child process stops.
- **SA_NOCLDWAIT** If this bit is set when calling *sigaction()* for the SIGCHLD signal, the system will not create zombie processes when children of the calling process exit. If the calling process subsequently issues a *wait()* (or equivalent), it blocks until all of the calling process's child processes terminate, and then returns a value of -1 with *errno* set to ECHILD.
- **SA_ONSTACK** If this bit is set, the system will deliver the signal to the process on a *signal stack*, specified with **sigaltstack(2)**.
- **SA_NODEFER** If this bit is set, further occurrences of the delivered signal are not masked during the execution of the handler.
- **SA_RESETHAND** If this bit is set, the handler is reset to SIG_DFL at the moment the signal is delivered.
- **SA_RESTART** See the paragraph below
- **SA_SIGINFO** If this bit is set, the handler function is assumed to be pointed to by the *sa_sigaction* member of struct *sigaction* and should match the prototype shown above or as below in EXAMPLES. This bit should not be set when assigning SIG_DFL or SIG_IGN

If a signal is caught during the system calls listed below, the call may be forced to terminate with the error EINTR, the call may return with a data transfer shorter than requested, or the call may be restarted. Restart of pending calls is requested by setting the SA_RESTART bit in *sa_flags*. The affected system calls include **open(2)**, **read(2)**, **write(2)**, **sendto(2)**, **recvfrom(2)**, **sendmsg(2)** and **recvmsg(2)** on a communications channel or a slow device (such as a terminal, but not a regular file) and during a **wait(2)** or **ioctl(2)**. However, calls that have already committed are not restarted, but instead return a partial success (for example, a short read count).

After a **fork(2)** or **vfork(2)** all signals, the signal mask, the signal stack, and the restart/interrupt flags are inherited by the child.

The **execve(2)** system call reinstates the default action for all signals which were caught and resets all signals to be caught on the user stack. Ignored signals

remain ignored; the signal mask remains the same; signals that restart pending system calls continue to do so.

The following is a list of all signals with names in the include file `<signal.h>`:

NAME	Default Action	Description
SIGHUP	terminate process	terminal line hangup
SIGINT	terminate process	interrupt program
SIGQUIT	create core image	quit program
SIGILL	create core image	illegal instruction
SIGTRAP	create core image	trace trap
SIGABRT	create core image	abort(3) call (formerly SIGIOT)
SIGEMT	create core image	emulate instruction executed
SIGFPE	create core image	floating-point exception
SIGKILL	terminate process	kill program
SIGBUS	create core image	bus error
SIGSEGV	create core image	segmentation violation
SIGSYS	create core image	non-existent system call invoked
SIGPIPE	terminate process	write on a pipe with no reader
SIGALRM	terminate process	real-time timer expired
SIGTERM	terminate process	software termination signal
SIGURG	discard signal	urgent condition present on socket
SIGSTOP	stop process	stop (cannot be caught or ignored)
SIGSTP	stop process	keyboard generated stop signal
SIGCONT	discard signal	continue after stop
SIGCHLD	discard signal	child status has changed
SIGTTIN	stop process	background read attempted from control terminal
SIGTTOU	stop process	background write attempted from control terminal
SIGIO	discard signal	I/O possible on descriptor <code>fcntl(2)</code>
SIGXCPU	terminate process	cpu limit exceeded <code>setrlimit(2)</code>
SIGXFSZ	terminate process	filesize exceeded <code>setrlimit(2)</code>
SIGVTALRM	terminate process	virtual time alarm <code>setitimer(2)</code>
SIGPROF	terminate process	profiling timer alarm <code>setitimer(2)</code>
SIGWINCH	discard signal	Window size change
SIGINFO	discard signal	status request from keyboard
SIGUSR1	terminate process	User defined signal 1
SIGUSR2	terminate process	User defined signal 2

The `sigaction()` function returns the value 0 if successful; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

Signal handlers should have either the ANSI C prototype:

```
void handler(int);
```

or the POSIX SA_SIGINFO prototype:

```
void handler(int, siginfo_t *info, ucontext_t *uap);
```

The handler function should match the SA_SIGINFO prototype when the SA_SIGINFO bit is set in flags. It then should be pointed to by the sa_sigaction member of struct sigaction. Note that you should not assign SIG_DFL or SIG_IGN this way.

If the SA_SIGINFO flag is not set, the handler function should match either the ANSI C or traditional BSD prototype and be pointed to by the sa_handler member of struct sigaction. In practice, FreeBSD always sends the three arguments of the latter and since the ANSI C prototype is a subset, both will work. The sa_handler member declaration in FreeBSD include files is that of ANSI C (as required by POSIX), so a function pointer of a BSD-style function needs to be casted to compile without warning. The traditional BSD style is not portable and since its capabilities are a full subset of a SA_SIGINFO handler its use is deprecated.

The *sig* argument is the signal number, one of the SIG... values from *signal.h*.

The *code* argument of the BSD-style handler and the si_code member of the info argument to a SA_SIGINFO handler contain a numeric code explaining the cause of the signal, usually one of the SI... values from *sys/signal.h* or codes specific to a signal, i.e. one of the FPE... values for SIGFPE.

The *uap* argument to a POSIX SA_SIGINFO handler points to an instance of ucontext_t.

The **sigaction()** system call will fail and no new signal handler will be installed if one of the following occurs:

- **[EFAULT]** Either *act* or *oact* points to memory that is not a valid part of the process address space
- **[EINVAL]** The *sig* argument is not a valid signal number
- **[EINVAL]** An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP

3 MAC OSX and BSD platform change

— * —

```
#include "bsdsignal.h"
```

—————

The MACOSX platform is broken because no matter what you do it seems to include files from `[[/usr/include/sys]]` ahead of `[[/usr/include]]`. On linux systems these files include themselves which causes an infinite regression of includes that fails. GCC gracefully steps over that problem but the build fails anyway. On MACOSX the `[[/usr/include/sys]]` versions of files are badly broken with respect to the `[[/usr/include]]` versions.

```

    __ * __

#if defined(MACOSXplatform)
#include "/usr/include/signal.h"
#else
#include <signal.h>
#endif

#include "bsdsignal.h1"

SignalHandlerFunc
bsdSignal(int sig,SignalHandlerFunc action,int restartSystemCall)
{
#ifdef MSYSplatform

    struct sigaction in,out;
    in.sa_handler = action;
    /* handler is reinstalled - calls are restarted if restartSystemCall */

```

 We needed to change `[SIGCLD]` to `[SIGCHLD]` for the `[MAC OSX]` platform and we need to create a new platform variable. This change is made to propagate that platform variable.

```

    __ * __

#if defined(LINUXplatform)
    if(restartSystemCall) in.sa_flags = SA_RESTART;
    else in.sa_flags = 0;
#elif defined (ALPHAplatform)
    if(restartSystemCall) in.sa_flags = SA_RESTART;
    else in.sa_flags = 0;
#elif defined(RIOSplatform)
    if(restartSystemCall) in.sa_flags = SA_RESTART;
    else in.sa_flags = 0;
#elif defined(SUN4OS5platform)
    if(restartSystemCall) in.sa_flags = SA_RESTART;
    else in.sa_flags = 0;
#elif defined(SGIplatform)
    if(restartSystemCall) in.sa_flags = SA_RESTART;
    else in.sa_flags = 0;
#elif defined(HP10platform)
    if(restartSystemCall) in.sa_flags = SA_RESTART;
    else in.sa_flags = 0;
#elif defined(MACOSXplatform)
    if(restartSystemCall) in.sa_flags = SA_RESTART;
    else in.sa_flags = 0;
#elif defined(BSDplatform)

```

```

    if(restartSystemCall) in.sa_flags = SA_RESTART;
    else in.sa_flags = 0;
#elif defined(SUNplatform)
    if (restartSystemCall) in.sa_flags = 0;
    else in.sa_flags = SA_INTERRUPT;
#elif defined(HP9platform)
    in.sa_flags = 0;
#else
    in.sa_flags = 0;
#endif

    return (sigaction(sig, &in, &out) ? (SignalHandlerFunc) -1 :
        (SignalHandlerFunc) out.sa_handler);
#else /* MSYSplatform */
    return (SignalHandlerFunc) -1;
#endif /* MSYSplatform */
}

```

4 License

```

/*
Copyright (c) 1991-2002, The Numerical ALgorithms Group Ltd.
All rights reserved.

```

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of The Numerical ALgorithms Group Ltd. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,

EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*/

References

- [1] nothing