

Signals, Part 2: Pending Signals and Signal Masks

dimyr7 edited this page on Dec 17, 2014 · 10 revisions

Where is Part 1?

There is no official "Part 1" page. However we introduced a simple `signal()` callback at the beginning of the course (e.g. [Forking](#), [Part 2: Fork, Exec, Wait Kill](#))

How can I learn more about signals?

The linux man pages discusses signal system calls in section 2. There is also a longer article in section 7 (though not in OSX/BSD):

```
man -s7 signal
```

What is a process's signal disposition?

For each process, each signal has a disposition which means what action will occur when a signal is delivered to the process. For example, the default disposition `SIGINT` is to terminate it. However this disposition can be changed by calling `signal()` (or as we will learn later) `sigaction()` to install a signal handler for a particular signal. You can imagine the processes' disposition to all possible signals as a table of function pointers entries (one for each possible signal).

The default disposition for signals can be to ignore the signal, stop the process, continue a stopped process, terminate the process, or terminate the process and also dump a 'core' file. Note a core file is a representation of the processes' memory state that can be inspected using a debugger.

Can multiple signals be queued?

No - however it is possible to have signals that are in a pending state. If a signal is pending it means it has not yet been delivered to the process. The most common reason for a signal to be pending is that the process (or thread) has currently blocked that particular signal.

If a particular signal, e.g. `SIGINT`, is pending then it is not possible to queue up the same signal again.

It *is* possible to have more than one signal of a different type in a pending state. For example `SIGINT` and `SIGTERM` signals may be pending (i.e. not yet delivered to the target process)

Edit

New Page

▼ Pages 51

[Home](#)

[#Example Markdown](#)

[#Informal Glossary](#)

[#Piazza: When And How to Ask For Help](#)

[C Programming, Part 1: Introduction](#)

[C Programming, Part 2: Text Input And Output](#)

[C Programming, Part 3: Common Gotchas](#)

[C Programming, Part 4: Debugging](#)

[Deadlock, Part 1: Resource Allocation Graph](#)

[Deadlock, Part 2: Deadlock Conditions](#)

[File System, Part 1: Introduction](#)

[File System, Part 2: Files are inodes \(everything else is just data...\)](#)


[File System, Part 3: Permissions](#)

[File System, Part 4: Working with directories](#)


[File System, Part 5: Virtual file systems](#)


Show 36 more pages...


Clone this wiki locally


 Clone in Desktop

<>









How do I block signals?

Signals can be blocked (meaning they will stay in the pending state) by setting the process signal mask or, when you are writing a multi-threaded program, the thread signal mask.

What happens when creating a new thread?

The new thread inherits a copy of the calling thread's mask

```
pthread_sigmask( ... ); // set my mask to block delivery of some signals
pthread_create( ... ); // new thread will start with a copy of the same mask
```

What happens when forking?

The child process inherits a copy of the parent's signal dispositions. In other words, if you have installed a SIGINT handler before forking, then the child process will also call the handler if a SIGINT is delivered to the child.

Note pending signals for the child are *not* inherited during forking.

What is signal disposition ?

The signal disposition of a process is a table of actions. It defines what will happen when a particular signal is delivered to a process. For example, the default disposition of SIG-INT is to terminate the process. The signal disposition is per process not per thread. The signal disposition can be changed by calling `signal()` (which is simple but not portable as there are subtle variations in its implementation on different POSIX architectures and also not recommended for multi-threaded programs) or `sigaction` (discussed later)

What happens during exec ?

Remember that `exec` replaces the current image with a new program image. In addition the signal disposition is reset. Any pending signals are cleared.

What happens during fork ?

The child process inherits a copy of the parent process's signal disposition and a copy of the parent's signal mask.

For example if `SIGINT` is blocked in the parent it will be blocked in the child too. For example if the parent installed a handler (call-back function) for SIG-INT then the child will also perform the same behavior.

Pending signals however are not inherited by the child.

How do I block signals in a single-threaded

program?

Use `sigprocmask` ! With `sigprocmask` you can set the new mask, add new signals to be blocked to the process mask, and unblock currently blocked signals. You can also determine the existing mask (and use it for later) by passing in a non-null value for `oldset`.

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);`
```

From the Linux man page of `sigprocmask`,

```
SIG_BLOCK: The set of blocked signals is the union of the current set and the set
SIG_UNBLOCK: The signals in set are removed from the current set of blocked signa
SIG_SETMASK: The set of blocked signals is set to the argument set.
```

The `sigset` type behaves as a bitmap, except functions are used rather than explicitly setting and unsetting bits using `&` and `|`.

It is a common error to forget to initialize the signal set before modifying one bit. For example,

```
sigset_t set, oldset;
sigaddset(&set, SIGINT); // Oops!
sigprocmask(SIG_SETMASK, &set, &oldset)
```

Correct code initializes the set to be all on or all off. For example,

```
sigfillset(&set); // all signals
sigprocmask(SIG_SETMASK, &set, NULL); // Block all the signals!
// (Actually SIGKILL or SIGSTOP cannot be blocked...)

sigemptyset(&set); // no signals
sigprocmask(SIG_SETMASK, &set, NULL); // set the mask to be empty again
```

How do I block signals in a multi-threaded program?

Blocking signals is similar in multi-threaded programs to single-threaded programs:

- Use `pthread_sigmask` instead of `sigprocmask`
- Block a signal in all threads to prevent its asynchronous delivery

The easiest method to ensure a signal is blocked in all threads is to set the signal mask in the main thread before new threads are created

```
sigemptyset(&set);
sigaddset(&set, SIGQUIT);
sigaddset(&set, SIGINT);
pthread_sigmask(SIG_BLOCK, &set, NULL);

// this thread and the new thread will block SIGQUIT and SIGINT
pthread_create(&thread_id, NULL, myfunc, funcparam);
```

Just as we saw with sigprocmask, pthread_sigmask includes a 'how' parameter that defines how the signal set is to be used:

```
pthread_sigmask(SIG_SETMASK, &set, NULL) - replace the thread's mask with given s
pthread_sigmask(SIG_BLOCK, &set, NULL) - add the signal set to the thread's mask
pthread_sigmask(SIG_UNBLOCK, &set, NULL) - remove the signal set from the thread'
```

How are pending signals delivered in a multi-threaded program?

A signal is delivered to any signal thread that is not blocking that signal.

If the two or more threads can receive the signal then which thread will be interrupted is arbitrary!

Legal and Licensing information: Unless otherwise specified, submitted content to the wiki must be original work (including text, java code, and media) and you provide this material under a [Creative Commons License](#). If you are not the copyright holder, please give proper attribution and credit to existing content and ensure that you have license to include the materials.

