Project - Signal

April 30, 2025

1 Background

In Linux, a **Signal** is an inter-process communication (IPC) mechanism that allows the operating system or other processes to **send asynchronous notifications** to a process, informing it of certain events. Signals are typically used to notify a process to handle events such as interrupts, termination, or errors.

This project requires implementing the Signal mechanism in the xv6 kernel, referencing the POSIX.1-2008 standard and parts of the Linux implementation. Some framework code and header files are provided.

Basic Concepts of Signals

- **Signals are asynchronous**, meaning they can arrive at a process at any time without the process actively checking for them.
- Each signal has a default behavior, typically terminating the process, pausing the process, or ignoring the signal.
- Processes can customize signal handling, ignore signals, or block their delivery.

2 Specification

2.1 POSIX.1

Signal Generation and Delivery A signal is said to be ”**generated**” for (or **sent** to) a process when the event that causes the signal first occurs. Examples of such events include detection of hardware faults, timer expiration, signals generated via terminal activity, as well as invocations of the `kill()` and `sigqueue()` functions.

Each process has an action to be taken in response to each signal defined by the system (see Signal Actions). A signal is said to be ”**delivered**” to a process when the appropriate action for the process and signal is taken.

During the time between the generation of a signal and its delivery or acceptance, the signal is said to be ”**pending**”. Ordinarily, this interval cannot be detected by an application. However, a signal can be ”**blocked**” from delivery to a process.

Build commit hash:

If the action associated with a blocked signal is anything other than to ignore the signal, the signal shall remain pending until it is **unblocked**, or the action associated with it is set to ignore the signal.

Each process has a ”signal mask” that defines the set of signals currently blocked from delivery to it. The signal mask for a process shall be initialized from that of its parent.

The determination of which action is taken in response to a signal is made at the time the signal is delivered, allowing for any changes since the time of generation. This determination is independent of the means by which the signal was originally generated. If a subsequent occurrence of a pending signal is generated, it is implementation-defined as to whether the signal is delivered more than once in circumstances other than those in which queuing is required.

Signal Actions There are three types of action that can be associated with a signal: `SIG_DFL`, `SIG_IGN`, or a pointer to a function. Initially, all signals shall be set to `SIG_DFL` or `SIG_IGN` prior to entry of the main() routine. The actions prescribed by these values are as follows.

SIG_DFL Signal-specific default action.

If the default action is to terminate the process abnormally, the process is terminated as if by a call to `exit()`, except that the status made available to `wait()` indicates abnormal termination by the signal.

If the default action is to ignore the signal, delivery of the signal shall have no effect on the process.

Setting a signal action to `SIG_DFL` for a signal that is pending, and whose default action is to ignore the signal (for example, SIGCHLD), shall cause the pending signal to be discarded, whether or not it is blocked.

SIG_IGN Ignore signal.

Delivery of the signal shall have no effect on the process. The behavior of a process is undefined after it ignores a SIGFPE, SIGILL, SIGSEGV, or SIGBUS signal that was not generated by kill(), sigqueue(), or raise().

The system shall not allow the action for the signals SIGKILL or SIGSTOP to be set to `SIG_IGN`.

Setting a signal action to `SIG_IGN` for a signal that is pending shall cause the pending signal to be discarded, whether or not it is blocked.

If a process sets the action for the SIGCHLD signal to `SIG_IGN`, the behavior is unspecified.

Pointer to a Function Catch signal.

On delivery of the signal, the receiving process is to execute the signal-catching function at the specified address. After returning from the signal-catching function, the receiving process shall resume execution at the point at which it was interrupted.

Note: *If you lookat the original POSIX or Linux documentation, you may noticed that this specification is not exactly the same as POSIX. However our project will always imply the SA_SIGINFO flag.*

The signal-catching function shall be entered as a C-language function call as follows:

```
void func(int signo, siginfo_t *info, void *context);
```

where `func` is the specified signal-catching function, signo is the signal number of the signal being delivered, and info is a pointer to a `siginfo_t` structure defined in <*signal.h*> containing at least the following members:

Member Type	Member Name	Description
int	*si_signo*	Signal number.
int	*si_code*	Cause of the signal.
int	*si_pid*	Sending process ID.
void *	*si_addr*	Faulting Address.
int	*si_status*	Exit value or signal

The system shall not allow a process to catch the signals SIGKILL and SIGSTOP.

Signal dispositions Each signal has a current disposition, which determines how the process behaves when it is delivered the signal.

A process can change the disposition of a signal using sigaction(2) or signal(2). Using this system calls, a process can elect one of the following behaviors to occur on delivery of the signal: perform the default action; ignore the signal; or catch the signal with a signal handler, a programmer-defined function that is automatically invoked when the signal is delivered.

By default, a signal handler is invoked on the normal process stack.

A child created via fork(2) inherits a copy of its parent's signal dispositions. During an execve(2), the dispositions of handled signals are reset to the default; the dispositions of ignored signals are left unchanged.

Sending a signal The following system calls and library functions allow the caller to send a signal:

- kill(2)

  Sends a signal to a specified process. **Note: In our project, this syscall is named sigkill.**

- sigqueue(3)

  Sends a real-time signal with accompanying data to a specified process.

Signal mask and pending signals A signal **may be blocked**, which means that it will not be delivered until it is later unblocked. Between the time when it is generated and when it is delivered a signal is said to be pending.

In a single-threaded application, sigprocmask(2) can be used to manipulate the signal mask.

A child created via fork(2) inherits a copy of its parent's signal mask; the signal mask is preserved across execve(2).

A child created via fork(2) initially has an empty pending signal set; the pending signal set is preserved across an execve(2).

Execution of signal handlers Whenever there is **a transition from kernel-mode to user-mode execution** (e.g., on return from a system call or scheduling of a thread onto the CPU), the kernel checks whether there is a **pending unblocked signal** for which the process has established a signal handler. If there is such a pending signal, the following steps occur:

(1) The kernel performs the necessary preparatory steps for execution of the signal handler:

(1.1) The signal is removed from the set of pending signals.

(1.3) Various pieces of signal-related context are saved into a special frame that is created on the **user-stack**. The saved information includes:

• the program counter register (i.e., the address of the next instruction in the main program that should be executed when the signal handler returns);

• architecture-specific register state required for resuming the interrupted program; For RISC-V architecture, they are general purpose registers x1-x31.

• the thread's current signal mask;

The above information is accessible via the ucontext_t object that is pointed to by the third argument of the signal handler. This object reflects the state at which the signal is delivered, rather than in the handler; for example, the mask of blocked signals stored in this object will not contain the mask of new signals blocked through sigaction(2).

(1.4) Any signals specified in act->sa_mask when registering the handler with sigaction(2) are added to the thread's signal mask. **The signal being delivered is also added to the signal mask.** These signals are thus blocked while the handler executes.

(2) The kernel constructs a frame for the signal handler on the user-stack. The kernel sets the program counter for the thread to point to the first instruction of the signal handler function, and configures the return address for that function to point to a piece of user-space code known as the signal trampoline (described in sigreturn(2)).

(3) The kernel passes control back to user-space, where execution commences at the start of the signal handler function.

(4) When the signal handler returns, control passes to the signal trampoline code.

(5) The signal trampoline calls sigreturn(2), a system call that uses the information in the stack frame created in step 1 to restore the thread to its state before the signal handler was called. The thread's signal mask and alternate signal stack settings are restored as part of this procedure. Upon completion of the call to sigreturn(2), the kernel transfers control back to user space, and the thread recommences execution at the point where it was interrupted by the signal handler.

From the kernel's point of view, execution of the signal handler code is exactly the same as the execution of any other user-space code. **That is to say, the kernel does not record any special state information indicating that the thread is currently executing inside a signal handler.** All necessary state information is maintained in user-space registers and the user-space stack. The depth to which nested signal handlers may be invoked is thus limited only by the user-space stack (and sensible software design!).

3 Project Specification

3.1 Basic

Codebase The code for this project is located at `https://github.com/yuk1i/SUSTechOS/tree/signals-codebase`.

You should clone the repository to a local project directory, manually modify the `git remote` to a private repository of your own, and share it with your team members. You can create a repository on GitHub or the university's GitLab, **ensuring your repository remains private**.

You should create a new branch `signals-1234` from the `signals-codebase` branch, where 1234 can be any name you prefer, and develop on this branch. Then, add your private repository to `git remote`, e.g., `myrepo`, and push the `signals-1234` branch to `myrepo`. Note that the `signals-codebase` branch may receive updates, so do not modify the codebase branch directly.

Below are reference commands:

```
$ git clone https://github.com/yuk1i/SUSTechOS -b signals-codebase project
$ cd project
(signals-codebase) $ git remote add myrepo git@github.com:your/reponame.git
(signals-codebase) $ git branch -C signals-1234
(signals-codebase) $ git checkout signals-1234
Switched to branch 'signals-1234'
Your branch is up to date with 'origin/signals-codebase'.
(signals-1234) $ git push --set-upstream myrepo
Enumerating objects: 886, done.
Counting objects: 100% (886/886), done.
Delta compression using up to 16 threads
Compressing objects: 100% (286/286), done.
Writing objects: 100% (886/886), 424.14 KiB | 11.78 MiB/s, done.
Total 886 (delta 591), reused 871 (delta 586), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (591/591), done.
To github.com:yuk1i/test-project-2025S.git
* [new branch]      signals-1234 -> signals-1234
branch 'signals-1234' set up to track 'myrepo/signals-1234'.
```

Codebase Explanation In this project, we provide header files related to signals: *os/signal/signal.h* and *os/signal/ksignal.h*. The former is shared between user programs and the xv6 kernel, so it should not include other kernel header files except *#include "types.h"*. The latter describes the implementation details of signals in the kernel and should not be referenced by user programs.

We also provide the *os/signal/ksignal.c* file, which contains the framework for signal implementation. You need to implement system calls such as `sigaction`, `sigreturn`, and `sigkill` in this file.

Usertest This project **provides usertest only for Base Checkpoints**. After starting the kernel with `make run`, you can execute `signal` in the `sh >>` prompt.

We may update the usertest, so please stay informed of notifications.

Documentation This document will be updated continuously. Please pay attention to Blackboard and email notifications. Each update will be noted at the end of the document with changes highlighted in the main text. The first page of each document includes the build date and build hash (located in the bottom left corner). Please ensure you are using the latest version of the document.

3.2 Defined Signals

This project defines 10 signals, each with a numeric identifier. Their properties are shown in the table below.

[t]

Signal Name	Number	Description	Default Action
SIGUSR0	1	User-Defined	Term
SIGUSR1	2	User-Defined	Term
SIGUSR2	3	User-Defined	Term
SIGKILL	4	Kill Process	Term
SIGTERM	5	Terminate Process	Term
SIGCHLD	6	Child Process stops or terminates	Ign
SIGSTOP	7	Stop Process	Stop
SIGCONT	8	Continue Process	Continue
SIGSEGV	9	Segmentation Fault	Term
SIGINT	10	Interrupt Process	Term

Default Action: *Term* means terminates the process, *Ign* means ignores the signal. *Stop* means stops the process and *Continue* means continues the process.

The signals `SIGKILL` and `SIGSTOP` **cannot be blocked or ignored**.

When multiple signals can be delivered, they are processed in the order of their numbers, with lower numbers having higher priority.

3.3 sigset_t

`sigset_t` represents a bit-mask set of signals. It is `typedef`'d as a `uint64` type. Each signal occupies a specific binary bit. The `sigmask` macro represents the binary bit of a signal, i.e., `1 << (signal_number)`.

For example, `SIGUSR2` is signal number `3`, so `sigmask(SIGUSR2)` has the value `1 << 3 = 0x8`.

We can use functions like `sigemptyset`, `sigaddset`, `sigdelset`, and `sigismember` to manipulate this set.

3.4 sigaction

The `sigaction_t` structure represents a signal handling function. Its definition is as follows:

```
struct sigaction {
    void (*sa_sigaction)(int, siginfo_t*, void *);
    sigset_t sa_mask;
    void (*sa_restorer)(void);
};
```

Here, `sa_sigaction` is a pointer to the signal handling function, `sa_mask` is a `sigset_t` type signal set indicating signals to be blocked during the execution of the signal handling function, and `sa_restorer` is a pointer to the `sigreturn` system call.

User programs set the signal handling function via the `sigaction` system call. The prototypes in the kernel and user space are as follows:

```
// os/signal/ksignal.h
int sys_sigaction(int signo, const sigaction_t __user *act, sigaction_t __user *oldact);

// user/lib/syscall.h
int sigaction(int signo, sigaction_t *act, sigaction_t *oldact);
```

User programs pass two user pointers through the system call: `act` and `oldact`. These indicate that the kernel should set the handling method for the specified `signo` to the method described in the `sigaction_t` structure pointed to by `act`, and the kernel should write the previous handling method back to the `sigaction_t` structure pointed to by `oldact`.

Taking the `basic6` and `basic7` test cases as examples, the following code registers the handling method for `SIGUSR0` as the `sa` structure, specifying the handling function as `handler7`, the signal return function as `sigreturn` (a stub for the system call), and blocking the `SIGUSR1` signal during the execution of `handler7`. In the `basic6` example, the `SIGUSR1` signal is not blocked.

```
void basic7(char* s) {
    sigaction_t sa = {
        .sa_sigaction = handler7,
        .sa_restorer  = sigreturn,
    };
    sigemptyset(&sa.sa_mask);
    sigaddset(&sa.sa_mask, SIGUSR1); // block SIGUSR1 when handling SIGUSR0

    sigaction(SIGUSR0, &sa, NULL);
    // ...
}
```

Thus, the execution order for the `basic6` example is: the parent process sends `SIGUSR0`, the child process enters `handler6` and enters sleep; then the parent process sends `SIGUSR1`, and upon exiting the sleep system call, the child process finds `SIGUSR1` pending, enters `handler6_2`, exits, executes the `sigreturn` system call, resumes the user state, continues executing the remaining code of `handler6`, and then exits with `sigreturn`.

In contrast, the execution order for `basic7` is: the parent process sends `SIGUSR0`, the child process enters `handler7` and enters sleep; then the parent process sends `SIGUSR1`, and upon exiting the sleep system call, the child process finds `SIGUSR1` pending but masked, so it continues executing the remaining code of `handler7` and exits with `sigreturn`. After rolling back the process's mask, `SIGUSR1` becomes pending and unmasked, so it enters `handler7_2` and finally exits with `sigreturn`.

3.5 siginfo_t & ucontext

The `siginfo_t` structure represents additional information about a signal. For the Base-Checkpoint phase, we only need to fill it entirely with zeros.

The `ucontext` structure represents the context of a user process before entering the signal handling function, including the PC pointer, all GPR values (i.e., the `mcontext` structure), and the original signal mask.

3.6 sigreturn

The `sigreturn` system call is a function that must be called at the end of every signal handling function. Its purpose is to allow the kernel to restore the user-space state to the state before the signal handling function was called.

We do not explicitly call it in the signal handler but implement it by setting the `ra` register in the trapframe in the kernel. After the user program finishes executing the signal handler, it jumps to the `ra` register, which contains the address of the `sigreturn` system call.

3.7 sigkill

The `sigkill` system call is used to send a signal to a specified process. Its prototype is as follows:

```
int sigkill(int pid, int signo, int code);
```

`pid` is the PID of the target process, `signo` is the number of the signal to send, and `code` is additional information about the signal.

4 Project Implementation Hints

4.1 How to Kill a Process

In kernel mode, if we need to terminate a process, we should use the `setkilled` function, defined as follows. `p` indicates the process to terminate, and `reason` indicates the reason for termination, which must be negative.

```
void setkilled(struct proc *p, int reason);
```

For example, when a page fault occurs in user mode, we use `setkilled(p, -2)` to terminate process `p`.

```
static void handle_pgfault(void) {
    infof("page fault in application ...");
    setkilled(p, -2);
}
```

Before returning to user mode (*usertrap*), we use `iskilled` to check if the current process has been terminated. If so, the `exit` function is called to terminate the process.

```c
void usertrap() {
    // are we still alive?
    if ((killed = iskilled(p)) != 0)
        exit(killed);

    usertrapret();
}
```

When handling signals, we require using the `setkilled` function to terminate a process, passing `-10 - signo` in `reason`. This allows us to retrieve the process's exit reason in `wait` and identify which signal caused the termination.

4.2 When to Check Pending Signals

We require checking pending signals **every time the kernel returns to user mode**. This occurs in the user trap handling function *usertrap* in *trap.c*, before calling *usertrapret*. Note that handling signals may cause the process to terminate, so we need to check if the process has been terminated after returning.

```c
void usertrap() {
    // prepare for return to user mode
    assert(!intr_get());

    do_signal();
    // are we still alive? pending signals may kill us.
    if ((killed = iskilled(p)) != 0)
        exit(killed);

    usertrapret();
}
```

4.3 Signal Handler is a Function

In the test code, we define a signal handler as if it were a function, for example:

```c
void handler6(int signo, siginfo_t *info, void *context) {
    // ...
}
```

For user programs, a signal is like a kernel interrupt, which can interrupt the execution of the user program at any time, transfer control to the signal handler function, and return to the original control flow after completion. User programs can specify a handler function for each signal, use the default handler function, or ignore it; they can also use `sigprocmask` to block certain signals.

Thus, when the kernel is about to trigger a signal handler, we need to ”call” the signal handler in user mode as if calling a function, following the Calling Convention. For example, function arguments are passed via registers a0–a7, a valid stack must be set up before entering the function, and the return address is stored in `ra`.

5 Project Checkpoints

Checkpoint Description You need to first implement the 3 Base Checkpoints. Subsequently, choose additional Checkpoints to implement. Refer to the grading section for details on scoring and team arrangements.

5.1 Base Checkpoint 1

Implement the `sigaction` system call, allowing a process to specify the handling method for a signal as `SIG_DFL`, `SIG_IGN`, or a specified `sa_sigaction`. Implement the `sigkill` system call, enabling a process to **send** a signal to a specified process.

Refer to POSIX.1 specifications in 2.1Execution of signal handlers and 2.1Signal Actions. When returning to user space, check for pending signals. If the user has specified a handling function, jump to the signal handling function in user mode and restore the user-mode state before the signal handler upon its return.

1 Check pending signals every time the kernel returns to user mode.

2 If the signal is not blocked or cannot be blocked, execute (`SIG_DFL`, `SIG_IGN`) or prepare to jump to the user-mode handling function specified in the *sigaction* system call.

3 Save the state of the user program before entering the signal handler on the user stack, i.e., the `ucontext` structure, containing the PC pointer, GPRs (architecture-specific register state), and signal-related information (i.e., signal mask). The contents of architecture-specific registers can be obtained from the Trapframe. When the CPU enters kernel mode from user mode, all architecture-related information is saved in the Trapframe.

4 Construct `siginfo_t` on the user stack (temporarily all zeros). This pointer is required as the second argument to the signal handler.

5 Refer to 2.1Execution of signal handlers, and modify the current process’s signal mask based on the specified handling method `sigaction_t`.

6 Set the Trapframe so that after `sret`, **the user mode begins executing the signal handling function** `sa_sigaction`. You need to set the three parameters of the signal handler: `signo`, the user pointer `siginfo_t`, and the user pointer `ucontext`. Additionally, after the signal handler exits, execute the `sa_restorer` function (i.e., initiate the `sigreturn` system call from user mode).

7 When the user exits the signal handler, it calls the `sigreturn` system call to restore the previously saved user-mode state from the user stack, allowing the user program to continue executing the code before entering the signal handler.

To complete the above steps, you need to implement system calls such as `sigaction`, `sigreturn`, and `sigpending`.

5.2 Base Checkpoint 2 - SIGKILL

Implement a special signal `SIGKILL` with the following behavior:

1 This signal cannot be blocked or ignored.

2 This signal cannot be handled.

3 This signal terminates the process with an exit code of `-10 - SIGKILL`.

5.3 Base Checkpoint 3 - Signal across fork and exec

Implement the behavior of signals across fork and exec.

During a fork, the child process inherits the parent's signal handling methods (sigaction) and signal mask but clears all pending signals.

During an exec, the child process resets all signal handling methods to their default values but retains those manually specified as ignored in sigaction. The signal mask and pending signals remain unchanged.

5.4 Optional Checkpoints

The remaining Checkpoints (optional) will be released later. Please stay informed via Blackboard and email notifications.

6 Grading

This project is worth 100 points and will be factored into the final grade proportionally. Teams are allowed, with a maximum of 3 members.

The 3 Base Checkpoints are worth 50 points, 10 points, and 10 points, respectively, and passing the `usertest` earns the points. Each optional Checkpoint is worth 10 points (for 3-member teams) or 15 points (for solo or 2-member teams). Requirements for the optional part will be specified in subsequent documentation.

6.1 Schedule

This project has two submissions and one defense.

- The first submission deadline is Week 14 of the academic calendar: May 20 (Tuesday) at 14:00.

  In the first submission, you must complete Base Checkpoint 1 and pass its `usertest`.

- The second submission deadline is Week 16 of the academic calendar: June 3 (Tuesday) at 14:00.

  This is the final submission, and we will grade based on this version.

During Week 14 (May 20–22) lab sessions, we will conduct a mid-term check to verify the completion of Base Checkpoint 1. Please ensure you complete this part before the lab session. If you fail to pass Base Checkpoint 1 in the first submission, the maximum score for this part will be reduced by 10 points (i.e., a maximum of 40 points for this part, and a total maximum of 90 points), with other parts unaffected.

During Week 16 (June 3–5) lab sessions, we will hold the project defense. You need to present your implementation ideas and details to us, and we may ask questions about specific parts of your code.

6.2 Submission

For each submission, you need to include:

- Code and git commit history
- A PDF report including your team member list, project design architecture, implementation details, and screenshots of passing usertest.

Regarding the report, you do not need to restate the project requirements. Focus on your implementation ideas, details, and challenges encountered. Consider the report as an introduction to your code for someone unfamiliar with it.

6.3 Plagiarism

Plagiarism is strictly prohibited. Any plagiarism will be dealt with severely.

7 History

The revision history of this document and the codebase is as follows:

- April 28 (2c8fba1): First release. Codebase commit bb9be7a.