Project - signal

2025 年 4 月 28 日

1 Background

在 Linux 中，Signal（信号）是一种进程间通信（IPC）机制，允许操作系统或其他进程向一个进程**发送异步的通知**，通知其发生了某些事件。信号通常用于通知进程处理某些事件，比如中断、终止或者错误等。

本次项目要求在 xv6 内核上实现 Signal 机制，参考 POSIX.1-2008 标准以及部分 Linux 实现。部分框架代码和头文件已经提供。

信号的基本概念

- **信号是异步的**，这意味着信号可以在任何时刻到达进程，而不需要进程主动检查。
- 每个信号都有一个默认的行为，通常是终止进程、暂停进程或忽略信号。
- 进程可以选择自定义信号的处理方式，也可以选择忽略它，甚至可以阻塞信号的传递。

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.

Build commit hash: 2c8fba1

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. 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

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

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:

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 本次 Project 的代码位于 `https://github.com/yuk1i/SUSTechOS/tree/signals-codebase`.

你应该将代码仓库 clone 到本地 project 目录下，并手动修改 `git remote` 为一个自己的私有仓库，并与小组成员共享。你可以在 GitHub 或校内 GitLab 上创建仓库，**请保持你的仓库为私有**。

你应该从 `signals-codebaes` 上创建一个新的分支 `signals-1234`，1234 可以换成任何你喜欢的名字，并在该分支上进行开发。随后，添加你创建的私有仓库到 `git remote` 中，例如 `myrepo`，并将 `signals-1234` 分支推送到 `myrepo` 上。请注意，`signals-codebase` 分支可能会更新，不要在 codebase 分支上修改内容。

以下是参考命令：

```
$ 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 Explaination 本次 Project 中，我们给定了关于 signal 的头文件：*os/signal/signal.h* 与 *os/signal/ksignal.h*。前者为用户程序和 xv6 内核共享的头文件，所以除了 *#include "types.h"* 不要引用其他内核头文件。后者描述了内核中 signal 的实现细节，用户程序不应该引用该头文件。

我们也给定了 *os/signal/ksignal.c* 文件，其中包含了 signal 的实现框架。你需要在该文件中实现 `sigaction`、`sigreturn`、`sigkill` 等系统调用。

Usertest 本次 Project **仅对 Base Checkpoints 提供了 usertest**，你在使用 `make run` 启动内核后，在 `sh >>` 中执行 `signal` 即可。

我们可能会更新 usertest，请留意通知。

Documentation 本文档会滚动更新，请留意 Blackboard 和邮件通知。每次更新都会在文档的结尾注明修改内容，并且在正文中高亮。每一份文档的第一页中包含该文档的构建日期和构建 Hash（位于左下角），请确保你使用的文档是最新的。

3.2 Defined Signals

该 Project 一共定义了 10 种 signal，每一种 signal 均有一个数字编号，它们的属性如下表所示。

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

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

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

当多个信号可以被 delivered 时，按照上述编号的顺序进行处理，编号越小优先级越高。

3.3 sigset_t

`sigset_t` 表示一个 signal 的 bit-mask 集合。它被 `typedef` 为一个 `uint64` 类型。每一种 signal 占据其中某个二进制位。`sigmask` 宏表示了一个 signal 的二进制位，即 `1 << (signal_number)`。

例如，`SIGUSR2` 为 3 号 signal，那么 `sigmask(SIGUSR2)` 的值为 `1 << 3 = 0x8`。

我们可以使用 `sigemptyset`、`sigaddset`、`sigdelset`、`sigismember` 等函数来操作这个集合。

3.4 sigaction

`sigaction_t` 结构体表示一个 signal 的处理函数。它的定义如下：

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

其中 `sa_sigaction` 是 signal 处理函数的指针，`sa_mask` 是一个 `sigset_t` 类型的信号集，表示在执行 signal 处理函数时需要屏蔽的信号。`sa_restorer` 是一个指向 `sigreturn` 系统调用的指针。

用户程序通过 sigaction 系统调用来设置 signal 的处理函数。该系统调用在内核和用户态的原型如下：

```
// 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);
```

用户程序通过系统调用传入两个用户指针：act 和 oldact，表示内核应该将该 signo 新的处理方式设置为 act 指针所指向的 sigaction_t 结构体中描述的方式，以及内核应该将原来的处理方式重新传回给用户程序（即写入到用户程序传入的 oldact sigaction_t 结构体指针中）。

以 basic6 与 basic7 测试点为例，以下代码将 SIGUSR0 的处理方式注册为 sa 结构体，其中指定了处理函数为 handler7，信号返回函数为 sigreturn（这是一个系统调用的 Stub），以及在执行 handler7 时屏蔽 SIGUSR1 信号。而 basic6 样例中并没有屏蔽 SIGUSR1 信号。

```
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);
    // ...
}
```

所以，basic6 样例的执行顺序为：父进程发送 SIGUSR0，子进程进入 handler6，陷入 sleep；然后父进程再发送 SIGUSR1，子进程在退出 sleep 系统调用时发现 SIGUSR1 为 pending，进入 handler6_2，其退出后执行 sigreturn 系统调用，恢复用户态状态，继续执行 handler6 的后续代码，然后再使用 sigreturn 退出。

而 basic7 的执行顺序为：父进程发送 SIGUSR0，子进程进入 handler7，陷入 sleep；然后父进程再发送 SIGUSR1，子进程在退出 sleep 系统调用时发现 SIGUSR1 为 pending 但是被 mask，所以继续执行 handler7 的后续代码并使用 sigreturn 退出，在回滚进程的 mask 后，SIGUSR1 变得 pending 但未 mask，所以进入 handler7_2，最后使用 sigreturn 退出。

3.5 siginfo_t & ucontext

siginfo_t 结构体表示一个 signal 的附加信息。在 Base-Checkpoint 阶段，我们只需要将其全部填充为 0 即可。

ucontext 结构体表示一个用户进程在进入 signal 处理函数前的上下文，包括 PC 指针以及所有 GPR 的值（即 mcontext 结构体），以及原先的 signal mask。

3.6 sigreturn

sigreturn 系统调用是每个 signal 处理函数结束后都需要调用的函数，它的作用是令内核还原用户态的状态，回到 signal 处理函数被调用前的状态。

我们并没有在 signal handler 中显式地调用它，而是通过在内核设置 trapframe 中的 `ra` 的方式来实现的。在用户程序执行完 signal handler 后，它会跳转到 `ra` 寄存器，而其中存放的恰好是 `sigreturn` 系统调用的地址。

3.7 sigkill

`sigkill` 系统调用用于向指定进程发送 signal。它的原型如下：

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

`pid` 是目标进程的 pid，`signo` 是要发送的 signal 的编号，`code` 是信号的附加信息。

4 Project Implementation Hints

4.1 How to kill a process

在内核态，如果我们需要结束一个进程，我们应该使用 `setkilled` 函数，该函数定义如下。`p` 表示要结束的进程，`reason` 表示结束的原因，必须为负数。

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

例如，在用户态发生 Page Fault 时，我们使用 `setkilled(p, -2)` 来结束进程 `p`。

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

在返回到用户模式前 (*usertrap*)，我们会使用 `iskilled` 检查当前进程是否被结束，如果是，则会调用 `exit` 函数结束进程。

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

    usertrapret();
}
```

在处理 signal 时，我们要求使用 `setkilled` 函数来结束进程，并在 `reason` 中传入 `-10 - signo`，这样我们就可以在 `wait` 时取回进程退出原因，并识别出是哪个 signal 结束了进程。

4.2 When to check pending signals

我们要求在**每次从内核回退到用户态时**检查 Pending Signals。即位于 *trap.c* 用户 Trap 处理函数 *usertrap* 中，调用 *usertrapret* 前。注意到处理信号可能会导致该进程被结束，所以我们在其返回后需要检查进程是否被结束。

```
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

在测试代码中，我们像定义一个函数一样定义了一个 signal handler，例如：

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

对于用户程序而言，signal 就好像内核的中断，它会在任何时刻打断用户程序的执行，转而执行 signal handler 函数；并在结束后返回原来的控制流。用户程序可以为每个 signal 指定一个 handler 函数、使用默认的 handler 函数、或者忽略它；也可以使用 `sigprocmask` 来屏蔽某些 signal。

那么，在内核中即将触发 signal handler 时，我们就需要像调用一个函数一样在用户态“调用” signal handler，而这是我们要遵循的即是 Calling Convention。例如，函数的参数由 a0 - a7 寄存器传递，进入函数前要有一个合法的栈，返回地址保存在 `ra` 中。

5 Project Checkpoints

Checkpoint 说明 你需要先实现 3 个 Base Checkpoint。随后，在后续的 Checkpoint 中挑选你想做的功能。关于分数、小组安排，请参照后续的评分章节。

5.1 Base Checkpoint 1

实现 `sigaction` 系统调用，进程可以指定某个 signal 的处理方式为 `SIG_DFL`、`SIG_IGN` 或指定的 `sa_sigaction`。实现 `sigkill` 系统调用，进程能向某个指定进程 **Send** signal。

参照 POSIX.1 规范中的 2.1 Execution of signal handlers 以及 2.1 Signal Actions，在回退到用户空间时检查 Pending 的 signal，如果用户指定了处理函数，则在用户态跳转到 signal 处理函数，并在其返回后恢复执行 signal handler 前的用户态状态。

1 在每次从内核回退到用户态时检查 Pending Signals。

2 如果该 Signal 没有被 Blocked 或者不能被 Blocked，执行 (`SIG_DFL`、`SIG_IGN`) 或准备跳转到以前在 *sigaction* 系统调用里指定的用户态处理函数。

3 在用户栈上保存用户程序进入 signal handler 前的状态，即 `ucontext` 结构体，包含 PC 指针、GPRs（architecture-specific register state）、以及 signal 相关的信息（即 signal mask）。架构相关寄存器的内容可以从 Trapframe 中获得。在 CPU 从用户态进入内核态时，所有架构相关的信息都被保存在了 Trapframe 上。

4 在用户栈上构造 `siginfo_t` （暂时全为 0 即可）。该指针要求传入 signal handler 的第二个参数。

5 参照2.1 Execution of signal handlers，根据指定的处理方式 `sigaction_t`，修改当前进程的 signal mask。

6 设置 Trapframe，使得当 `sret` 后，**用户态开始执行 signal 处理函数** `sa_sigaction`。你需要设置 signal handler 的三个参数：`signo`、用户指针 `siginfo_t` 以及用户指针 `ucontext`。并且，在 signal handler 退出后执行 `sa_restorer` 函数（即从用户态发起 `sigreturn` 系统调用）。

7 用户从 Signal Handler 中退出时，会调用 `sigreturn` 系统调用后，从用户栈上恢复以前保存的用户态状态，使用户程序继续执行进入 signal handler 前的代码。

在完成上述步骤时，你需要实现 `sigaction`、`sigreturn`、`sigpending` 等系统调用。

5.2 Base Checkpoint 2 - SIGKILL

实现一个特殊的 signal `SIGKILL`，它的行为如下：

1 该 signal 无法被 Blocked 或者 Ignored。

2 该 signal 无法被处理。

3 该 signal 会终止进程，退出代码为 `-10 - SIGKILL`。

5.3 Base Checkpoint 3 - Signal across fork and exec

实现 signal 在 fork 和 exec 之间的行为。

在 fork 时，子进程继承父进程的 signal 处理方式 (sigaction)、signal mask，但是清空所有 pending signal。

在 exec 时，子进程重置所有的 signal 处理方式为默认值，但是保留被手动指定为 ignore 的那些 sigaction，signal mask 以及 pending signal 不变。

5.4 自选 Checkpoint

剩余 Checkpoint (自选部分) 将在后续发布，请留意 Blackboard 和邮件通知。

6 Grading

本次 Project 总分 100 分，将按照比例计入总评。允许组队，上限为 3 人。

3 个 Base Checkpoint 分别为 50 分、10 分、10 分，通过 `usertest` 即可获得分数。每个自选 Checkpoint 分数为 10 分（3 人组）或 15 分（单人组或双人组），自选部分的要求将在后续发布的文档中额外说明。

6.1 Schedule

本次 Project 共有两次提交、一次答辩。

- 第一次提交时间为校历第 14 周：5 月 20 日（星期二）14:00 前。

  在第一次提交中，你必须完成 Base Checkpoint 1 并且通过它的 `usertest` 测试。

- 第二次提交时间为校历第 16 周：6 月 3 日（星期二）14:00 前。

  这是最后一次提交，我们将依此版本进行评分。

在第 14 周（5 月 20 日 - 5 月 22 日）的实验课中，我们将进行一次中期检查，检查 Base Checkpoint 1 的完成情况。请务必在实验课前完成该部分的实现。如果你未能在第一次提交中通过 Base Checkpoint 1，该部分的分值上限将被扣除 10 分（即该部分最高为 40 分，总分最高为 90 分），其余部分不受影响。

在第 16 周（6 月 3 日 - 6 月 5 日）的实验课中，我们将进行 Project 答辩。你需要向我们展示你在 Project 中的实现思路、实现细节，我们可能会对你代码中的某些部分进行提问。

6.2 Submission

在每次提交中，你需要包含：

- 代码以及 git 提交记录
- 一份 PDF 格式的报告，包含你的小组成员列表，Project 设计架构、实现细节，以及通过 usertest 的截图。

关于报告，你不需要在报告中重新阐述我们的题目，请集中在你的实现思路、实现细节、以及遇到的问题。你可以将报告视为你正在向一个不熟悉你代码的人介绍你的代码。

6.3 Plagiarism

严禁抄袭。任何抄袭行为都将被严肃处理。

7 History

本文档和 Codebase 的修订历史如下：

- 4 月 28 日 (2c8fba1)：第一次发布。codebase commit bb9be7a。