Signal Implementation in xv6 Kernel

Project Signal - Final Report

Jaouhara Zerhouni Khal
Harrold Tok Kwan Hang
Hok Layheng

June 3, 2025

1 Team Members

1. **Jaouhara Zerhouni Khal** — 12211456
2. **Harrold Tok Kwan Hang** — 12212025
3. **Hok Layheng** — 12210736

2 Executive Summary

This project involved implementing a signal handling mechanism in an XV6-based operating system kernel. The signal system supports asynchronous communication with user processes and mimics POSIX-style signal behavior, enabling processes to send, receive, and handle signals through custom-defined or default behaviors. This report is a contunuation from Checkpoint 1, which now describes the support of new functions such as SIGKILL, signal across fork and exec, and bonus functions (SIGCHLD, SIGINFO and SIGALARM).

3 Project Design and Architecture

Our signal implementation is integrated modularly within the xv6 kernel to ensure clarity and maintainability. The key design components include:

- **Signal Table:** Each process maintains a signal table storing registered handlers, pending signals, and the signal mask.
- **Kernel Modifications:** Core kernel files (`syscall.c` and `trap.c` were modified to manage signal delivery, context saving, and system call support (e.g., `sigaction`, `sigprocmask`, `alarm`).
- **Signal Delivery Logic:** Signals are delivered lazily during user-kernel transitions (e.g., timer interrupt, syscall return) to avoid reentrancy issues. Delivery respects masking and handler type (default, ignore, user-defined).

- **User Context Handling:** User-mode context is saved/restored during signal entry/return, enabling transparent resumption post-handler execution.
- **Extension Support:** Architecture supports additional features like `siginfo_t`, `SIGCHLD`, and timer-based `SIGALRM` via modular extensions and syscall interfaces.

Function Descriptions

`int siginit(struct proc *p)`
Initializes the signal-related structures inside a process control block (PCB). It sets all handlers to default, masks to 0, and clears pending signals.

`int siginit_fork(struct proc *parent, struct proc *child)`
Copies the signal action and mask from the parent process to the child during fork. Clears the child's pending signals. For kernel test mode, it sets SIGUSR2 as pending for testing.

`int siginit_exec(struct proc *p)`
Called during an exec system call to reset signal handlers to default. Retains ignored signals explicitly by re-setting them to SIG_IGN.

`int do_signal(void)`
Called during user-to-kernel transitions. Handles pending, unmasked signals by either performing the default action or invoking a user-defined handler via stack manipulation.

`int sys_sigaction(int signo, const sigaction_t *act, sigaction_t *oldact)`
Implements the `sigaction()` system call. Sets or retrieves the signal handler and associated mask for a particular signal. Prevents overriding SIGKILL.

`int sys_sigreturn(void)`
Called from user space after a signal handler returns. Restores the saved context (registers, PC, and signal mask) from the stack and decreases the signal handling depth.

`int sys_sigprocmask(int how, const sigset_t *set, sigset_t *oldset)`
Modifies or retrieves the signal mask of the current process depending on the `how` argument: `SIG_BLOCK`, `SIG_UNBLOCK`, or `SIG_SETMASK`. SIGKILL cannot be masked.

`int sys_sigpending(sigset_t *set)`
Retrieves the set of pending signals for the current process.

`int sys_sigkill(int pid, int signo, int code)`
Sends a signal to the process with the given PID. Populates the corresponding `siginfo_t` structure and wakes the process if it is sleeping.

`int sys_alarm(unsigned int seconds)`
Sets an alarm for the calling process to go off after `seconds`. When the timer expires, a `SIGALRM` is sent to the process. If a previous alarm was set, the remaining time is returned. Passing `seconds = 0` cancels any pending alarm.

`void check_alarm(void)`
Called periodically by the kernel clock tick handler. Checks if the alarm time for the current process has been reached. If so, it delivers a `SIGALRM` and clears the alarm.

Implementation of Bonus

Bonus Checkpoint 5.3.1 – `SIGALRM`

Goal: Enable support for the `SIGALRM` signal to allow processes to set alarms that deliver a signal after a specified duration.

Implementation Summary:

- Implemented `sys_alarm()` in `ksignal.c`, which schedules or cancels alarms.
- The alarm time is tracked using `p->alarm_ticks` and `p->alarm_interval`, based on the system's tick rate (100Hz).
- Alarms are checked periodically in `check_alarms()`, which queues `SIGALRM` signals and clears expired timers.
- `sys_sleep()` and `check_sleeping_processes()` were implemented to manage process suspension and resumption based on ticks.
- `handle_timer_interrupt()` triggers both alarm and sleep checking during each timer interrupt.

Testing Summary:

- Used `test_alarm`, `test_alarm_remaining` and `test_alarm_cancel` to verify correct SIGALRM behavior.
- Validated:
  - Accurate remaining time when setting new alarms.
  - Correct delivery and cancellation of SIGALRM.
  - Proper handling of sleep and wake-up events.
- Assertions and debug output confirmed tick values, alarm triggering, and cancellation correctness.

Bonus Checkpoint 5.3.2 – `siginfo_t`

Goal: Enable the kernel to construct and deliver a `siginfo_t` structure to user-level signal handlers, as required by the `SA_SIGINFO` convention.

Implementation Summary:

- Extended the signal delivery path to build a `siginfo_t` structure dynamically on the user stack before invoking the handler.
- Populated fields:

  - `si_signo` – the signal number.
  - `si_code` – the reason or metadata for the signal.
  - `si_pid` – the sender's PID or `-1` if from the kernel.

- Updated `sys_sigkill()` to populate `siginfo` metadata used for constructing the structure.
- The main logic was implemented in:
  - `handle_stack()` – builds and pushes `siginfo_t` and `ucontext_t` onto the stack.
  - `do_signal()` – triggers signal delivery using `handle_stack()`.
  - `sys_sigreturn()` – restores the context and signal mask from the user stack.

Testing Summary:

- Verified using the `siginfo_bonus()` test case.
- A child process received:
  - A signal from a user process (`SIGUSR1`).
  - A simulated kernel signal (`SIGUSR2`).
- Inside the handler, assertions verified the accuracy of `si_signo`, `si_pid`, and `si_code`.
- The test passed only when both signals were handled with the correct metadata.

Bonus Checkpoint 5.3.3 – `SIGCHLD`

Goal: Automatically deliver `SIGCHLD` to a parent process when a child exits, including signal metadata.

Implementation Summary:

- Modified `exit()` in `proc.c` to check the parent's signal handler for `SIGCHLD`.
- If set (not `SIG_DFL` or `SIG_IGN`), `SIGCHLD` is delivered with:
  - `si_signo = SIGCHLD`
  - `si_pid = child PID`
  - `si_code = exit code`
- The parent is woken up if it is sleeping, to handle the signal promptly.

Testing Summary:

- The `sigchld_bonus()` test case verifies correct signal handling.
- A parent registers a `SIGCHLD` handler, and a child exits with code `123`.
- Inside the handler:
  - Assertions check `info->si_signo == SIGCHLD`.

  - `wait()` is called to reclaim the child.
  - Verified that `info->si_pid` and `info->si_code` match the expected values.

- The test succeeds only if all validations pass, ensuring correct `SIGCHLD` signal delivery and metadata.

4 Challenges and Solutions

4.1 Challenge 1: Integrating Stack Setup Inline

Problem: We initially factored stack setup into a separate `handle_stack()` helper, but inlining that logic into `do_signal()` risked code duplication and made the signal-delivery loop bulky. **Solution**: We carefully merged the stack-alignment, `ucontext` `siginfo_t` allocation and user-copy calls directly into `do_signal()`, eliminating the extra function call while retaining clear comments and minimizing duplicated lines by reusing a small block that saves registers and copies both structures under the same lock.

4.2 Challenge 2: Mask and Nesting Management

Problem: Ensuring that the original signal mask is preserved across nested handlers and correctly restored by `sys_sigreturn()`, while also preventing the handler itself from being interrupted by the same signal. **Solution**: We embed the old mask in the saved `ucontext.uc_sigmask` and, upon entry, OR in both the handler's `sa_mask` and the signal's own bit to the process `sigmask`. In `sys_sigreturn()`, we restore exactly the saved mask and decrement `handling_depth`, ensuring proper nesting behavior.

4.3 Challenge 3: Atomicity and Lock Ordering

Problem: Avoiding races between `sys_sigkill()` enqueueing new signals and `do_signal()` clearing them, especially when a process is sleeping or executing user-copy operations under the mm lock. **Solution**: In `sys_sigkill()`, we acquire the target's `p->lock` before touching `sigpending` and release only after waking it. In `do_signal()`, we disable interrupts around the pending-check and then acquire `mm->lock` briefly to copy out both `siginfo_t` and `ucontext` in one atomic section, avoiding inversion by always grabbing `p->lock` before `mm->lock`.

4.4 Challenge 4: Default-Action Complexity

Problem: The default action for many signals (kill, stop, ignore, or no-op) has grown into two nearly identical switch statements in `do_signal()`, once for the `SIG_DFL` path and again for handler-failure fallback. **Solution**: We consolidated the default-case logic into a single inline switch block that is invoked both when `sa_sigaction == SIG_DFL` and when our inline setup fails, reducing duplication and ensuring consistent behavior for each signal number.

4.5 Challenge 5: SIGALARM Implementation Complexity

Problem: Many files needed to be modified to support the optional function of an alarm, which added layers of difficulty to a project where most of the code is modified within a single file. **Solution**: Understanding the codebase using LLMs sped up the workflow and understanding of minor files such as usys.pl and proc.h.

5 Testing and Validation

We used the provided `basic.c` test suite—including `basic1--8`, `basic10--11`, and `basic20`—along with our own test cases: `siginfo`, `sigchld`, `alarm1`, and `alarm2`, to validate our implementation from Checkpoint 2 onward.

All tests passed, confirming correct behavior for signal delivery, handler installation, context switching, masking, timer signals, and metadata propagation.

5.1 Test Results

```
sh >> signal
=== TESTSUITE ===
- Project: signal test suite
 Usage: ./signal [testname]
 - [testname] can be one of the following:
  basic1
  basic2
  basic3
  basic4
  basic5
  basic6
  basic7
  basic10
  basic11
  basic20
  siginfo
  sigchld
  alarm1
  alarm2
  alarm3

Running all tests
signaltests starting
test basic1: OK
test basic2: OK
test basic3: OK
handler4 triggered
test basic4: OK
handler5 triggered
handler5 triggered
handler5 triggered
handler5 triggered
handler5 triggered
handler5 triggered
test basic5: OK
handler6 triggered due to 1
handler6_2 triggered due to 2
test basic6: OK
```

Figure 1: Final Test Results: Part1

```
handler7 triggered due to 1
handler7_2 triggered due to 2
test basic7: OK
test basic10: OK
test basic11: OK
test basic20: OK
test siginfo: OK
test sigchld: OK
alarm(2) triggered, 2 second timer starts
Alarm triggered!
test alarm1: OK
Starting test_alarm_remaining
alarm(5) returns remaining = 0
Waiting 2 seconds (busy-wait)...
Finished waiting 2 seconds.
alarm(3) returns remaining = 3
Waiting for alarm to trigger...
Alarm triggered!
alarm(5) returns remaining = 0
Waiting 2 seconds (busy-wait)...
Finished waiting 2 seconds.
alarm(0) returns remaining = 3
Waiting 5 seconds to confirm no alarm...
test_alarm_remaining completed
test alarm2: OK
alarm(10) triggered, 10 second timer starts
Waiting 2 seconds (busy-wait)...
Finished waiting 2 seconds.
Alarm cancelled early, remaining time: 8
test alarm3: OK
sh > child 4 exit with code 0
```

Figure 2: Final Test Results: Part2

Our implementation successfully passes all base checkpoint tests, demonstrating:

- Correct signal handler invocation
- Proper signal masking
- Signal delivery across process boundaries
- Context restoration functionality

6 Conclusion

We successfully completed Checkpoints 2 and beyond of the Signal project, building on the foundation from Checkpoint 1. Our implementation includes:

- **Checkpoint 2 (SIGKILL):** Added support for an uncatchable, unignorable `SIGKILL` that forcefully terminates a process.
- **Checkpoint 3 (fork/exec):** Implemented correct signal behavior across `fork` and `exec`.

- **Optional - SIGALRM:** Designed a user-mode timer syscall to raise `SIGALRM` after a specified duration.
- **Optional - SIGINFO:** Populated `siginfo_t` with detailed metadata passed to user handlers.
- **Optional - SIGCHLD:** Enabled parent processes to receive `SIGCHLD` with exit information upon child termination.

All functionalities were verified with custom test cases, ensuring correct behavior and kernel integration.