Mentoring **O**perating **S**ystem (**MentOS**)

Signals

Created by
Enrico Fraccaroli
enrico.fraccaroli@univr.it

Table of Contents

Signals

Signals

Signals transmission

Signals transmission

The kernel distinguishes two different phases related to signal transmission:

- **Signal generation** The kernel updates a data structure of the destination process to represent that a new signal has been sent.
- **Signal delivery** The kernel forces the destination process to react to the signal by changing its execution state, by starting the execution of a specified signal handler, or both.

Signals that have been generated but not yet delivered are called *pending signals*. At any time, only one pending signal of a given type may exist for a process; additional pending signals of the same type to the same process are not queued but simply discarded.

Signals transmission

Although the notion of signals is intuitive, the kernel implementation is rather complex. The kernel must:

- Remember which signals are blocked by each process.
- When switching from Kernel Mode to User Mode, check whether a signal for a process has arrived. This happens at almost every timer interrupt (roughly every millisecond).
- Determine whether the signal can be ignored. This happens when all of the following conditions are fulfilled:
  - The signal is not blocked by the destination process.
  - The signal is being ignored by the destination process (either because the process explicitly ignored it or because the process did not change the default action of the signal and that action is "ignore").
- Handle the signal, which may require switching the process to a handler function at any point during its execution and restoring the original execution context after the function returns.

Signals

Actions Performed upon Delivering a Signal

Actions Performed upon Delivering a Signal

There are three ways in which a process can respond to a signal:

- Explicitly ignore the signal.
- Execute the default action associated with the signal. This action, which is predefined by the kernel, depends on the signal type and may be any one of the following:
  - **Terminate** The process is terminated (killed).
  - **Dump** The process is terminated (killed) and a core file containing its execution context is created, if possible; this file may be used for debug purposes.
  - **Ignore** The signal is ignored.
  - **Stop** The process is stopped—i.e., put in the TASK_STOPPED state
  - **Continue** If the process was stopped (TASK_STOPPED), it is put into the TASK_RUNNING state
- Catch the signal by invoking a corresponding signal-handler function.

Signals

Data Structures Associated with Signals

Data Structures Associated with Signals

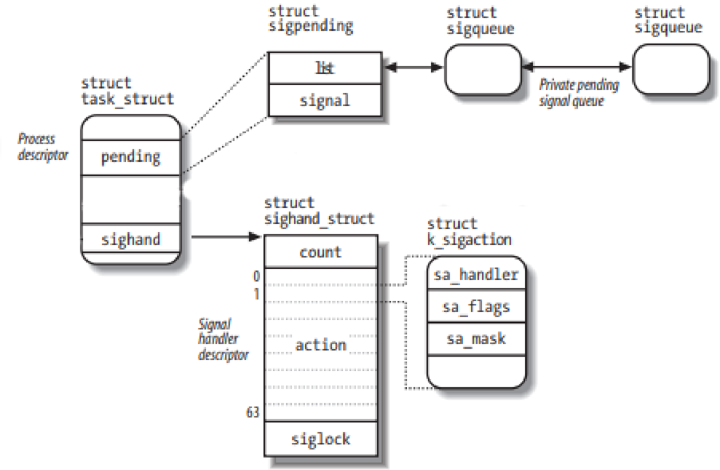

Figure: The most significant data structures related to signal handling

Data Structures Associated with Signals

The struct `sigpending` *`pending`* contains a list of `sigqueue_t` structs rappresenting all the *private pending signals* of the process and the bitfield `signal` that indicates which signals types are currently inside the list.

The struct `sighand_t` *`sighand`* contains all the process's signal handler descriptor, one for each type of signal.

The `sigaction_t` struct describes how each signal must be handled, and it contains the following fields:

- **sa_handler** This field specifies the type of action to be performed; its value can be a pointer to the signal handler, SIG_DFL (that is, the value 0) to specify that the default action is performed, or SIG_IGN (that is, the value 1) to specify that the signal is ignored
- **sa_flags** This set of flags specifies how the signal must be handled
- **sa_mask** This sigset_t variable specifies the signals to be masked when running the signal handler

Data Structures Associated with Signals

The `sigqueue_t` struct rappresents an entry of the signal queue and it contains the following fields:

- **flags** Flags of the sigqueue data structure
- **info** Describes the event that raised the signal using the `siginfo_t` struct, 128-byte data structure that stores information about an occurrence of a specific signal. It contains the following important fields:
    - **si_signo** The signal number
    - **si_errno** The error code of the instruction that caused the signal to be raised, or 0 if there was no error
    - **si_code** A code identifying who raised the signal
    - **sigval_t** A union storing information depending on the type of signal.

Signals

Generating a Signal

Generating a Signal

The most important syscall for generating signals is `kill`, it ends up invoking the `__send_signal` function which inserts a new item in the private pending signal queue of the desired process descriptor. This function does not directly perform the second phase of delivering the signal.

```
static int __send_signal(int sig, siginfo_t* info, struct task_struct* t);
```

Before adding the signal the `handle_stop_signal` function is invoked to check for some types of signals that might nullify other pending signal in the process. For example if the signal is a stop signal then the `SIGCONT` signal is removed if present.

```
if (sig == SIGSTOP || sig == SIGTSTP || sig == SIGTTIN || sig == SIGTTOU) {

    sigset_t mask;
    sigemptyset(&mask);
    sigaddset(&mask, SIGCONT);

    __rm_from_queue(&mask, &p->pending);
}
```

Generating a Signal

In the same way if the signal is SIGCONT the SIGSTOP, SIGTSTP, SIGTTIN, and SIGTTOU signals are removed from the private pending queue and the process is immediately awakened.

```
if (sig == SIGCONT) {

    sigset_t mask;
    sigemptyset(&mask);

    sigaddset(&mask, SIGSTOP);
    sigaddset(&mask, SIGTSTP);
    sigaddset(&mask, SIGTTIN);
    sigaddset(&mask, SIGTTOU);

    __rm_from_queue(&mask, &p->pending);

    struct list_head *it, *tmp;
    list_for_each_safe (it, tmp, &stopped_queue.task_list) {
        // Awakens targeted process...
    }
}
```

Generating a Signal

At the beginning the `__send_signal` function checks if the signal is not ignored by the process, that is the sighandler for this specific signal is not set to `SIG_IGN`

```
if (__sig_is_ignored(t, sig)) {
    pr_debug("Trying to send signal (%2d)‘%s‘ to task (%2d)‘%s‘: ignored.\n",
            sig, strsignal(sig), t->pid, t->name);
    __unlock_task_sighand(t);
    return 0;
}
```

Then we checks if the process is in a valid state

```
if ((t->state == EXIT_ZOMBIE) || (t->state == EXIT_DEAD)) {
    pr_debug("Trying to send signal (%2d)‘%s‘ to task (%2d)‘%s‘: zombie or dead.\n",
            sig, strsignal(sig), t->pid, t->name);
    __unlock_task_sighand(t);
    return -EINVAL;
}
```

Generating a Signal

In the end a new `sigqueue_t` struct is allocated and appended to the private pending queue, the signal bitfield is also updated with the newly inserted signal type using the `sigaddset` function.

```
sigqueue_t *q = __sigqueue_alloc(t, sig, GFP_KERNEL);
if (q == NULL) {
    __unlock_task_sighand(t);
    return -EAGAIN;
}

list_head_add_tail(&q->list, &t->pending.list);
if (info != SEND_SIG_NOINFO)
    memcpy(&q->info, info, sizeof(siginfo_t));

// Set that there is a signal pending.
sigaddset(&t->pending.signal, sig);
pr_debug("Added pending signal (%2d)`%s` to task (%2d)`%s`, pending `%d, %d`.\n",
        sig, strsignal(sig), t->pid, t->name, t->pending.signal.sig[0], t->pending.signal.sig[1]);

__unlock_task_sighand(t);
return 0;
```

Signals

Delivering a Signal

Delivering a Signal

To handle the nonblocked pending signals, the kernel invokes the `do_signal` function.

```
int do_signal(struct pt_regs *f);
```

The heart of this function consists of a loop that repeatedly invokes the `__dequeue_signal` function until no nonblocked pending signals are left in the private pending signal queues.

```
while (!list_head_empty(&current->pending.list)) {
    signr = exit_code = __dequeue_signal(&current->pending, &current->blocked, &info);

    ...
}
```

The `__dequeue_signal` considers all signals in the private pending signal queue, starting from the lowest-numbered signal. It updates the data structures to indicate that the signal is no longer pending and returns its number.

Delivering a Signal

Inside the dequeue loop we obtain the correspondent signal action and if it is ignored by the process we continue with a new loop execution and a new signal.

```
sigaction_t *ka = &current->sighand.action[signr - 1];

if (ka->sa_handler == SIG_IGN) {
    continue;
}
```

The only exception comes when the receiving process is init, in which case the signal is discarded.

```
if (current->pid == 1)
    continue;
```

Delivering a Signal

If the `ka->sa_handler` is equal to `SIG_DFL` then `do_signal` must perform the default action of the signal.

```
if (ka->sa_handler == SIG_DFL) {
    switch (signr) {
        ...
    }
}
```

The default action depends on the signal number and is hardcoded. Signals like `SIGCONT`, `SIGCHLD`, `SIGWINCH` or `SIGURG` are simply ignored.

```
case SIGCONT:
case SIGCHLD:
case SIGURG:
case SIGWINCH:
    continue;
```

Delivering a Signal

The signals whose default action is **dump** may create a core file in the process working directory; this file lists the complete contents of the process's address space and CPU registers.

```
case SIGFPE:
case SIGSEGV:
case SIGBUS:
case SIGSYS:
case SIGXCPU:
case SIGXFSZ:
    if (do_coredump(signr, f))
        exit_code |= 0x80;
```

The signals whose default action is **stop** may stop the current process. To do this, `do_signal` sets the state of current to `TASK_STOPPED` and then invokes the `schedule` function.

```
case SIGTSTP:
case SIGTTIN:
case SIGTTOU:
    if (is_orphaned_pgrp(current->gid))
        continue;

case SIGSTOP:
    __do_signal_stop(current, f, signr);
```

Delivering a Signal

The difference between `SIGSTOP` and the other signals is subtle: `SIGSTOP` always stops the process, while the other signals stop the process only if it is not in an **orphaned process group**. The POSIX standard specifies that a process group is not orphaned as long as there is a process in the group that has a parent in a different process group but in the same session.

The default action of the remaining signals is **terminate** which consists of simply killing the process.

```
case SIGQUIT:
case SIGILL:
case SIGTRAP:
case SIGABRT:
    sys_exit(3);
```

Delivering a Signal

If a handler has been established for the signal, the `do_signal` function must enforce its execution. It does this by invoking `handle_signal`.

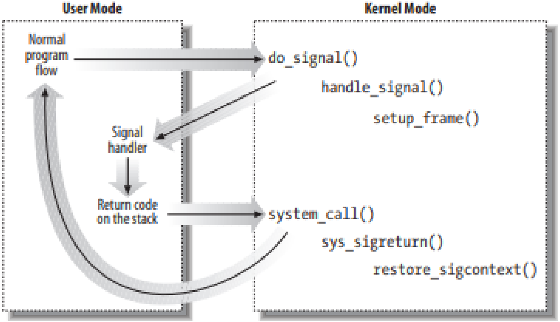

Figure: Flow of code in delivering a signal to a process

Delivering a Signal

Executing a signal handler is a rather complex task because of the need to juggle stacks carefully while switching between `User Mode` and `Kernel Mode`.

Signal handlers are functions defined by User Mode processes and included in the User Mode code segment. The `handle_signal` function runs in Kernel Mode while signal handlers run in User Mode; this means that the current process must first execute the signal handler in User Mode before being allowed to resume its "normal" execution.

Moreover, when the kernel attempts to resume the normal execution of the process, the Kernel Mode stack no longer contains the hardware context of the interrupted program, because the Kernel Mode stack is emptied at every transition from User Mode to Kernel Mode.

Delivering a Signal

At the beginning the `handle_signal` function saves the previous signal mask, adds the signal to the list of blocked signals and stores the process' hardware state.

```
memcpy(&current->saved_sigmask, &current->blocked, sizeof(sigset_t));
sigaddset(&current->blocked, signr);

current->thread.signal_regs = *regs;
```

Then it sets the instruction pointer of the current process to the specified signal handler.

```
regs->eip = (uintptr_t)ka->sa_handler;
```

Delivering a Signal

Then it sets up the User Mode stack. When the process switches again to User Mode, it starts executing the signal handler, because the handler's starting address was forced into the program counter.

```
// Push on the stack the signal number, first and only argument of the handler.
PUSH_ARG(regs->useresp, int, signr);

// Push on the stack the function required to handle the signal return.
PUSH_ARG(regs->useresp, uint32_t, current->sigreturn_eip);
```

When that function terminates, the return code placed on the User Mode stack is executed. This code invokes the `sigreturn` system call, which restores the registers before the signal handling, allowing the process to resume execution from where the signal handler was called.

```
*f = current->thread.signal_regs;

// Restore the previous signal mask.
memcpy(&current->blocked, &current->saved_sigmask, sizeof(sigset_t));
```

Delivering a Signal

TODO: immagine user stack