Signals in Unix Systems

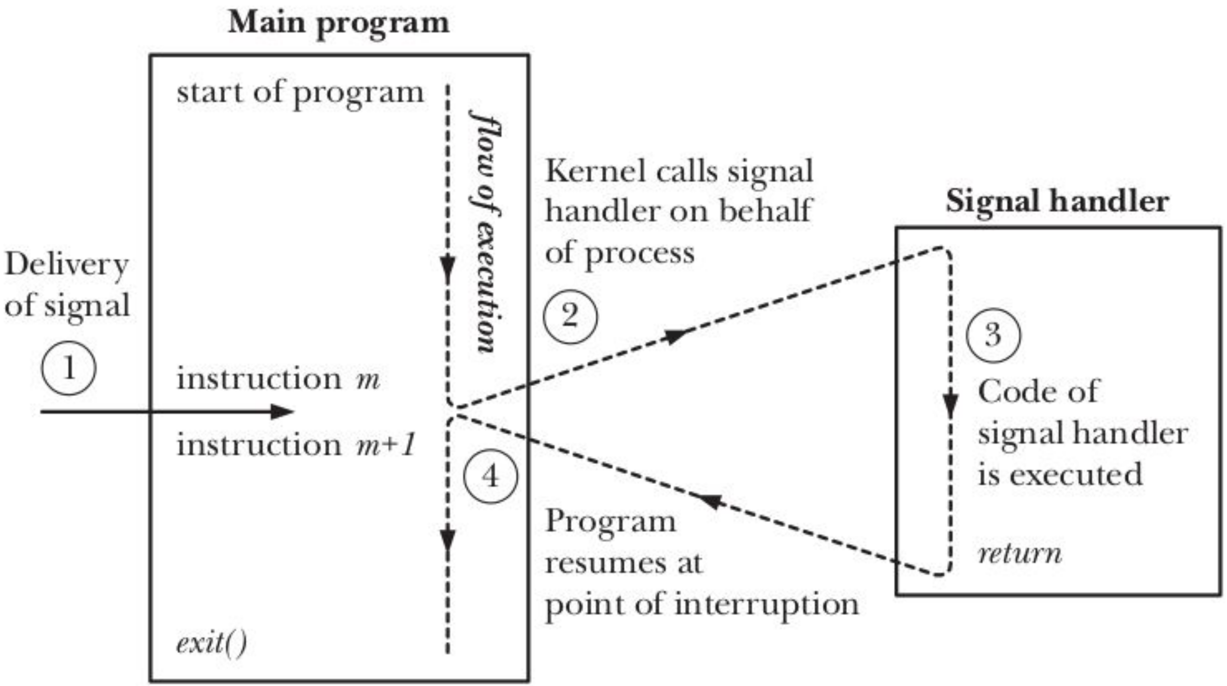
An overview of signals processing

Rodrigo Gonzalez, PhD

Introduction to Signals

- Definition: Signals as notifications to a process that an event has occurred.
- Analogy: Signals as software interrupts, disrupting the normal program flow.

Introduction to Signals



Sources of Signals

- Process-to-process communication with appropriate permissions.
- Self-generated signals within a process.
- Kernel-generated signals in response to events.

Causes of Kernel-Generated Signals

- Hardware exceptions: Invalid instructions, divide-by-zero, inaccessible memory access.
- Terminal signals: Special characters like Control-C (interrupt) and Control-Z (suspend).
- Software events: Input availability, window resizing, timer expirations, resource limits exceeded, child process termination.

Identifying Signals

- Signal numbers and symbolic names (e.g., SIGINT for interrupt signal).
- Use of `<signal.h>` for portable symbolic names across implementations.

Identifying Signals

Table 20-1: Linux signals

Name	Signal number	Description	SUSv3	Default
SIGABRT	6	Abort process	•	core
SIGALRM	14	Real-time timer expired	•	term
SIGBUS	7 (SAMP=10)	Memory access error	•	core
SIGCHLD	17 (SA=20, MP=18)	Child terminated or stopped	•	ignore
SIGCONT	18 (SA=19, M=25, P=26)	Continue if stopped	•	cont
SIGEMT	undef (SAMP=7)	Hardware fault		term
SIGFPE	8	Arithmetic exception	•	core
SIGHUP	1	Hangup	•	term
SIGILL	4	Illegal instruction	•	core
SIGINT	2	Terminal interrupt	•	term
SIGIO / SIGPOLL	29 (SA=23, MP=22)	I/O possible	•	term
SIGKILL	9	Sure kill	•	term
SIGPIPE	13	Broken pipe	•	term

Standard Signals

- Explanation of standard (traditional) signals.
- Range: Numbered from 1 to 31 on Linux.
- Role in notifying processes of system or external events.

Signal Identification

- Signals defined by unique small integers, starting from 1.
- Symbolic names (e.g., SIGINT, SIGKILL) used for portability and readability.
- Importance of using symbolic names in programming.

Commonly Used Signals

- SIGINT: Interrupt from keyboard (Ctrl-C).
- SIGTERM: Termination request.
- SIGKILL: Immediate program termination.
- SIGSEGV: Invalid memory access.
- SIGCHLD: Child process stopped or terminated.

Default Actions for Signals

- Ignore: SIGCHLD (sometimes).
- Terminate: SIGTERM.
- Core Dump: SIGSEGV.
- Stop: SIGSTOP.
- Continue: SIGCONT.

Modifying Default Actions

- Mechanisms to change signal responses: Signal handlers and signal masking.
- Discussion on the importance of custom handling for application-specific behavior.

Handling of SIGKILL and SIGSTOP

- Special status: Cannot be caught, blocked, or ignored.
- Purpose: Ensure the ability to terminate or stop processes in any state.

Handling Signals

- Default actions: Ignore, terminate, generate core dump and terminate, stop (suspend), resume.
- Overriding default actions with signal handlers for custom behavior.

Signal Handlers

- Definition and purpose of a signal handler.
- Custom functions executed in response to signals.
- The role of signal handlers in managing application-specific signal responses.

Signal Handlers

```
void
handler(int sig)
{
    /* Code for the handler */
}
```

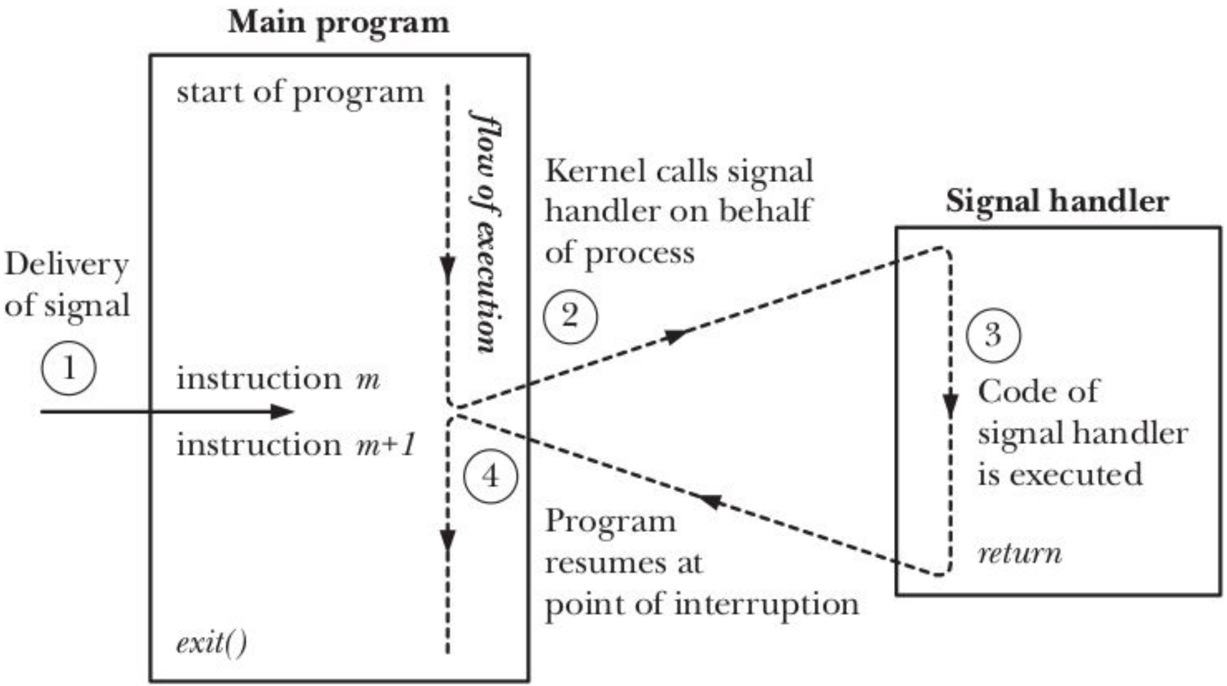
```
#include <signal.h>

void ( *signal(int sig, void (*handler)(int)) ) (int);
```

Returns previous signal disposition on success, or SIG_ERR on error

Signal Handler Execution

- Description of what happens when a signal handler is invoked:
- The normal flow of program execution is interrupted.
- The specified handler function is executed.
- Program execution resumes from the point of interruption after the handler returns.

Signal Handlers



Design Considerations for Signal Handlers

- Emphasis on simplicity due to the asynchronous execution of handlers.
- Highlight restrictions on safe operations within a signal handler.

Best Practices for Signal Handlers

- Recommendations for writing effective and safe signal handlers.
- Avoiding complex operations and unsafe functions within handlers.

Use Case: Handling Interrupt Signals

- Practical example of using a signal handler to catch interrupt signals (e.g., Control-C).
- Discussion on how to gracefully handle such interrupts in user applications.

Introduction to Sending Signals

- Brief overview of what signals are in Unix/Linux systems.
- The importance of sending signals for process control and communication.

The `kill()` System Call

- Definition and primary use of the `kill()` function to send signals.
- Basic syntax: `int kill(pid_t pid, int sig);`.

Parameters of `kill()`

- Explanation of `pid` parameter: Process ID or group to which the signal is sent.
- Explanation of `sig` parameter: The specific signal to be sent.

Usage Scenarios for `kill()`

- Single process signaling.
- Signaling processes within a group.
- Special case: Sending signals to all processes accessible by a user.

Permission Requirements

- Description of permission rules for sending signals.
- The role of process ownership and effective user IDs in signal permissions.

Special Signals

- Highlighting `SIGKILL` and `SIGSTOP` as signals that cannot be ignored or caught.
- Discussion on the implications of these signals being unblockable.

Error Handling in `kill()`

- Common error scenarios (e.g., `EPERM`, `ESRCH`) and their meanings.
- Importance of error checking in signal-sending operations.

Practical Examples

- Demonstrating how to use `kill()` in real-world programming scenarios.
- Code snippets illustrating the sending of signals to manage process behavior.