Linux Signals and Daemons

Alessandro Barenghi

Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano

alessandro.barenghi - at - polimi.it

May 20, 2014

Lesson contents

Overview

- Asynchronous interruption mechanisms
- Signal issuing and handling
- Signal masking

Signals

Overview

- Signals are practically implemented as software triggered interrupts
- We have already seen the system utility employed to raise signals (`kill`)
- We will now understand how to manage signals from within a program
- We will also deal on how the delivery of signals in order to obtain interrupt free sections

Interrupts

Software what?

- Interrupts are events altering the regular execution of a program by a processor
- Can be caused by :
  - Illegal operations (e.g. divide by zero)
  - Invalid opcodes in the binary
  - Page faults
  - Software triggers (debuggers)
- The interruption of the program is instantaneous, thus process context saving issues occur

Interrupts

Interrupt table

- Interrupt occours → interrupt handler is called and:
  - Immediately executed in case of hardware interrupts or...
  - Scheduled for execution immediate execution when the process resumes
- Pointer to interrupt handlers are stored in an interrupt table
- x86(_64) has 256 of them, the first one is reserved for hardware interrupts
- Linux employs the 128th to store the signal handlers for the process (`int 0x80`)

Interrupts

Interrupt Handling

- In order to handle an interrupt, the control unit of the CPU:
  - Finds the correct interrupt vector and determines which entry has been triggered
  - Checks if the interrupts handler requires a change in the privilege level
  - Saves the process context (registers content and program status word)
  - Loads the interrupt handler entry point and sets correctly segment selector and offsets
  - The handler is finally run :)
- Interrupts can be blocked through setting a proper flag in the control unit, creating interruption safe regions

Signals

Signal Handling

- Signal handling mimicks interrupt handling :
  - Every process has an associated region of the process descriptor to track the signals sent to him
  - Upon sending a signal, the kernel updates the process descriptor of the destination process
  - The signal is received as soon as the process is selected for execution
  - Before the process is run, the kernel checks if there are any pending signal to be run
  - If the signals are not blocked, the process execution resumes from the handler instead of the previous state
  - This is repeated until all the pending signals have been dealt with

Signals

Differences

- Signal handlers are userspace code, interrupts are not
- Signal handlers may invoke system calls
- Signal handlers are not dealt with in the same instant a signal is risen
- Multiple signals of the same type may be issued before the first is dealt with
- The behaviour for multiple issues of the same signal to the same process is not defined
- Since signal handling in Linux employs the realtime signal architecture, only one signal is received

Signals

Signal Handling

The actions to be performed upon receiving a signal are specified in a `sigaction` structure :

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

Signals

Signal Handling

The signal handler function `void (sa_handler)(int)`

- Receives as the only parameter, the number of the signal
- Does not return anything (as there is no one to return the value to)
- Can be interrupted as its running time is considered user code
- Should be kept as small as possible to minimize interruptions

Signals

Handler installation

A signal handler is installed via the `sigaction` primitive taking as parameters :

- The signal number `signum`
- A sigaction struct `act` containing the new handler
- A sigaction struct `oldact` where the old handler is saved
- The `SIG_DFL` macro specifies the default signal handler
- The signal handler is installed until a new one is set

Signals

Signal Masking

- It is possible to block some signals from being delivered
- A blocked signal will be delivered as soon as the block is removed[a]
- The set of signal to be temporarily blocked can be specified in a `sigset_t` structure
- The `sigemptyset` function initializes an empty signal set, while the `sigfillset` initializes a full signal set
- The `signaladdset` and `signaldelset` respectively add and remove a signal from the set

[a]this is different from the interrupt, which, if blocked, will be ignored

Signals

Signal Masking

- Once a signal set has been built , it can be used either as a block or unblock mask
- The `sigprocmask` primitive adds/removes to/from the blocked signal set of the process
- The action is specified via the first parameter which can be either `SIG_BLOCK, SIG_UNBLOCK` or `SIG_SET`
- The function saves the previous signal block mask for convenience in restoring

Signals

Peculiarities

- Two signals cannot be blocked: `KILL` and `STOP`
- Every child inherits a copy of its parent signal mask upon the call of the `fork` primitive
- The signal mask is also preserved across the `execve` primitive
- If a signal is raised as a consequence of a hardware interrupt (e.g. `SIGSEGV` or `SIGFPE`) the kernel will take drastic actions even if the signal is masked
- It is possible to meaningfully recover from a `SIGSEGV` through keeping an alternate stack and restoring the correct program flow in the SIGSEGV handler

Continuous running

Daemons

- Running a process in background is commonly called transforming it into a daemon
- A daemon is a process which runs for an undefinite amount of time (usually, until killed or the machine bursts in flames)
- By default, the daemon only communicates via logfiles as no terminal is expected to be running it
- Usually, the working directory of a daemon is the root directory (i.e. `/`)
- For the sake of clarity, the daemon processes have a filename ending in `d` (e.g. `/usr/sbin/sshd`)

Continuous running

the `daemon` primitive

- A convenient method to transform a process into a daemon is the `daemon` primitive
- This function accepts two integer parameters and performs the following actions :
  - Forks the running process
  - Makes the parent program call an `_exit()`, thus reparenting the program to `init`
  - If the `nochdir` parameter is zero, changes the working directory to `/`
  - If the `noclose` parameter is zero, closes standard input, output and error descriptors

Continuous running

Logging

- Since the daemons run in background and have no associated terminal, some way for them to communicate errors should be devised
- The most common way is to employ a log file
- In order to ease the output on the log file, usually it replaces either standard output or standard error (or both)
- This can be accomplished via the `dup2` primitive
- `int dup2(int oldfd, int newfd)` duplicates `oldfd` into `newfd`: passing either 1 or 2 as `newfd` effectively replaces `stdout` and `stderr`