Operating Systems

Signals

Giorgio Grisetti
grisetti@diag.uniroma1.it

Department of Computer Control and Management Engineering
Sapienza University of Rome

These slides are heavily based from material of Prof. Francesco Quaglia (thanks Francesco)

Basic Concepts

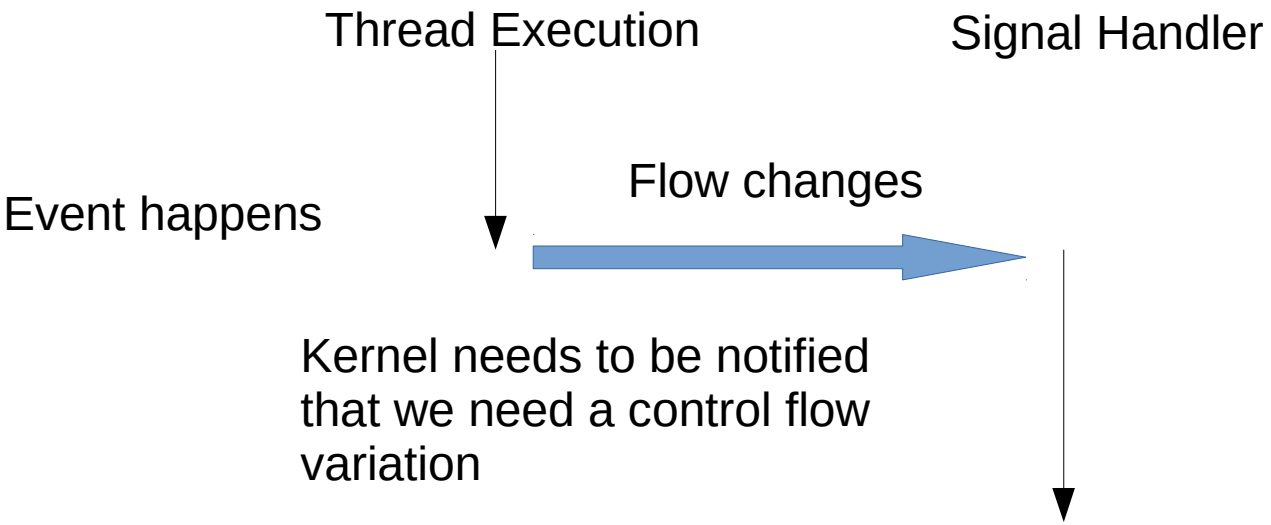
Signals in UNIX are an basic mechanism through which events are notified to a process

- When a process receives such an event, he can perform specific actions to handle the event

Signals are NOT messages since

- Signals are sent occasionally by another process, very often by the OS as a result of a system event (`SIGSEGV/SIGINT/SIGFPE`)
- A signal might not carry information about the sender

Signals in UNIX



There is a predefined set of events.

For a process, each event might be either

- Ignored implicitly
- Ignored explicitly
- Captured

Such a policy is changed through syscall and can vary during the process execution

The OS sends a signal anyway to a process, if a process chooses of not explicitly ignore or not capture some events, it might be terminated when receiving them.

Common UNIX signals

- **`SIGHUP`**: received when the terminal to which it was associated has been closed, or the connection is interrupted
- **`SIGINT`**: received when the user presses key combination (typically `ctrl+c`)
- **`SIGQUIT`**: same as `SIGINT`, but the OS generates a “core-dump”
- **`SIGILL`**: sent by the OS when a process attempt to execute an illegal instruction
- **`SIGKILL`**: cannot be captured and brutally terminates the receiving process
- **`SIGSEGV`**: sent by the OS when an attempt to preform an operation on a memory address would cause a violation
- **`SIGTERM`**: something in between `SIGINT` and `SIGKILL`. It can be captured
- **`SIGALRM`**: sent by the process alarm (if set) when a certain time interval elapsed
- **`SIGUSR1/2`**: user defined - you can do what you like with these
- **`SIGCHLD`**: sent to a process when one of its children terminates

Sending Signals

```
int kill(int pid, int signal)// sends a signal to a pid

int raise(int signal) // sends a signal to self

uint alarm(uint time) // sends sigalarm after time seconds
```

- All signals but `SIGKILL` are implicitly ignored
- `SIGCHLD`, albeit implicitly ignored by default, does not terminate the process
- The alarm settings are not completely preserved through a fork

Capturing Signals

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

- `signum`: the signal we want to capture
- `handler`: the function pointer
- Two specific values are defined for the handler
  - `SIG_DFL`: default behavior
  - `SIG_IGN`: explicitly ignore the signal

Inheritance behavior of signal handlers

- Signal handlers are inherited through `fork()`
- `exec*(...)` only preserves `SIG_IGN/SIG_DFL` (why?)

How kernel manages signals

Overall:

- A signal is generated by some means (another process, the OS, kill, raise or alarm). Receiving a signal alters a bit in the **signal mask** within the PCB
- When this happens the receiving process is moved in ready (even if it was not in ready)
- If a bit was set in te signal mask, when the CPU is assigned to the process, the context is the one of the signal handles

Caveats:

- Multiple deliveries of the same signal might be lost
- If the computation was interrupted at a generic instruction, the flow continues from that point when the handler is done
- If a process was in a syscall, two things can happen:
  - blocking syscalls (e.g. a read from disk), are aborted and **`errno`** is set to **`EINTR`**. Such a syscalls are not resumed
  - Non blocking syscalls are not interrupted by any signal
  - **`errno`** is now thread safe, and is a macro

Safe bloking syscall

Bad

```
if (syscall()==-1) {
    Do stuff..
}
```

Good

```
while (syscall()==-1) {
  if (errno!=EINTR){
    Do stuff..
  }
}
```

Waiting for Signals

```
#include <unistd.h>

int pause(void);
```

- blocks a process until any signal arrives
- does not know who sent the signal or which signal unlocked the process just from such a syscall
  - How do we tackle this problem?
    - Easy answer:
      - Capture the signal
      - In the implementation one should:
        - Save the signal number
        - Roll-back to the original handler (**if any**).

Two simple examples

```
#include <stdio.h>
#include <fcntl.h>
#include <signal.h>
char c;

void sh() {
   printf("I'm alive!\n");
    //reinstall the handler
    signal(SIGALRM, sh);
    alarm(5)
}


int main(int argc, char *argv[]) {
   alarm(5);
   signal(SIGALRM,sh);
   while(1) read(0, &c, 1);
}
```

```
#include <stdio.h>
#include <fcntl.h>
#include <signal.h>
int x,y,i;

void sh() {
   //race conditon x and y might be
   //different
   printf("I'm alive! %d-%d-\n",
            x,y);
   signal(SIGALRM, sh);
   alarm(5)
}


int main(int argc, char *argv[]) {
   alarm(5);
   signal(SIGALRM,sh);
   while(1)    x = y = i++ % 1000;
}
```

Unreliability of the Signals

```
char c;

void sh() {
   printf("caught sigint!\n");
   // receiving another sigint
   // in this time terminates
   // the process
   signal(SIGINT, sh)
}


int main(int argc, char *argv[]) {
   signal(SIGINT,sh);
   while(1) read(0, &c, 1);
}
```

Signal Sets

`sigset_t:` represents a set of signals

Functions to manage a `sigset`

```
int sigemptyset(sigset_t *set);

int sigfillset(sigset_t *set);

int sigaddset(sigset_t *set, int signum);

int sigdelset(sigset_t *set, int signum);

int sigismember(const sigset_t *set, int signum);
```

Masking Signals

```
int sigprocmask(int how,
                const sigset_t *set,
                sigset_t *oldset);
```

Examines and changes blocked signals, setting the management of the signal mask. Arguments:

- **how**: **SIG_BLOCK**, **SIG_UNBLOCK**, **SIG_SETMASK**
- **set**: the set in the pool is the one to which the change will be applied
- **oldset**: if not null, stores a backup prior the operation

```
int sigpending(const sigset_t *set);
```

Returns the signals that were masked while blocked

Sigaction: safe signals

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
```

Allows to inspect or modify the action performed when a signal is received

signum: the signal

act: the new struct encoding all fields for the handler

oldact: the old struct returned as backup

```
struct sigaction {
  // old style handler
  void     (*sa_handler)(int);
  // new style handler // use either one or the other. Not both!!!
  void     (*sa_sigaction)(int, siginfo_t *, void *);

  sigset_t   sa_mask;   //signals blocked while handling this one
  int        sa_flags;   //behavior (ignore, reinstall..etc)
  void     (*sa_restorer)(void); // obsolete
};
```

Siginfo: gets even more info

```
siginfo_t {
            int      si_signo;     /* Signal number */
            int      si_errno;     /* An errno value */
            int      si_code;      /* Signal code */
            int      si_trapno;    /* Trap number that caused
                                      hardware-generated signal
                                      (unused on most architectures) */
            pid_t    si_pid;       /* Sending process ID */
            uid_t    si_uid;       /* Real user ID of sending process */
            int      si_status;    /* Exit value or signal */
            clock_t  si_utime;     /* User time consumed */
            clock_t  si_stime;     /* System time consumed */
            sigval_t si_value;     /* Signal value */
            int      si_int;       /* POSIX.1b signal */
            void    *si_ptr;       /* POSIX.1b signal */
            int      si_overrun;   /* Timer overrun count;
                                      POSIX.1b timers */
            int      si_timerid;   /* Timer ID; POSIX.1b timers */
            void    *si_addr;      /* Memory location which caused fault
*/
            …
        }
```