F28HS Hardware-Software Interface: Systems Programming

Hans-Wolfgang Loidl

School of Mathematical and Computer Sciences,
Heriot-Watt University, Edinburgh

Semester 2 — 2025/26

[0]No proprietary software has been used in producing these slides

Outline

HERIOT WATT UNIVERSITY

Lecture 5. Exceptional Control Flow and Signals

HERIOT WATT UNIVERSITY

What are interrupts and why do we need them?

- In order to deal with internal or external events, **abrupt** changes in control flow are needed.
- Such abrupt changes are also called **exceptional control flow (ECF)**.
- Informally, these are known as **hardware- and software-interrupts**.
- The system needs to take special action in these cases (call interrupt handlers, use non-local jumps)

[0] Lecture based on Bryant and O'Hallaron, Ch 8

ECF on different levels

ECF occurs at different levels:

- **hardware level:** e.g. arithmetic overflow events detected by the hardware trigger abrupt control transfers to **exception handlers**
- **operating system:** e.g. the kernel transfers control from one user process to another via **context switches**.
- **application level:** a process can send a signal to another process that abruptly transfers control to a signal handler in the recipient.

In this class we will cover **an overview of ECF with examples from the operating system level.**

HERIOT WATT UNIVERSITY

Handling ECF on different levels

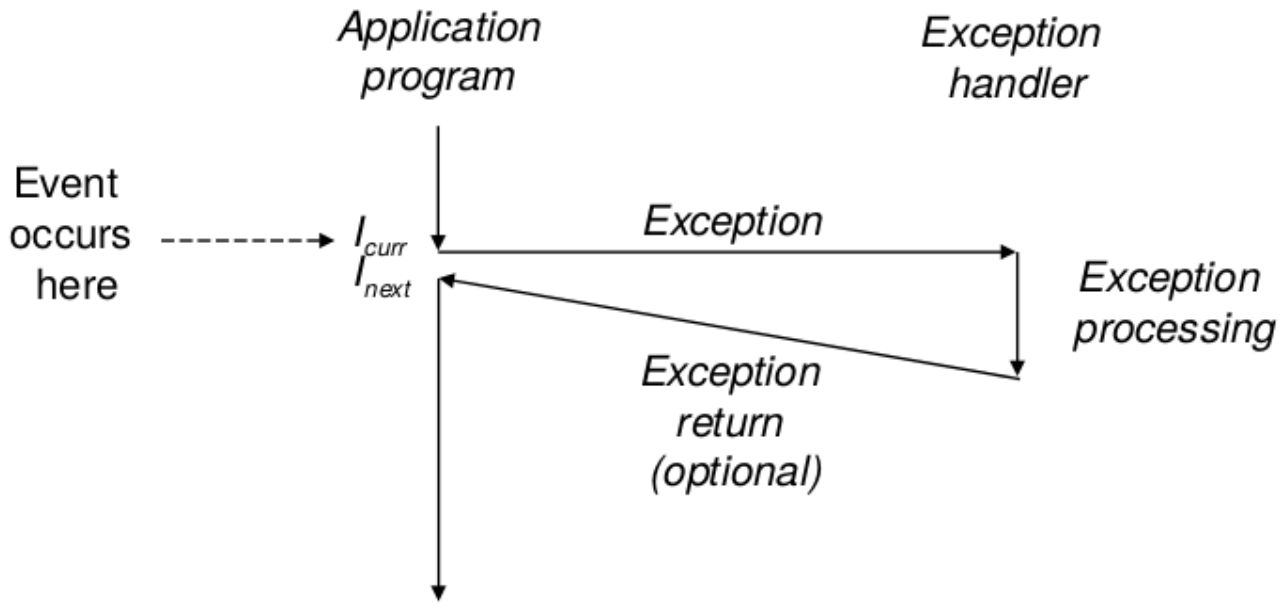
ECF is dealt with in different ways:

- **hardware level:** call an **interrupt** routine, typ. in Assembler
- **operating system:** call a **signal** handler, typ. in C
- **application level:** call an **exception** handler, e.g. in a Java `catch` block

Exceptions

Definition

An exception is an abrupt change in the control flow in response to some change in the processor's state.



A change in the processor's state (event) triggers an abrupt control transfer (an **exception**) from the application program to an exception handler. After it finishes processing, the handler either returns control to the interrupted program or aborts.

Exceptions (cont'd)

When the processor detects that the event has occurred, it makes an indirect procedure call (the **exception**), through a jump table called an **exception table**, to an operating system subroutine (the **exception handler**) that is specifically designed to process this particular kind of event.

When the exception handler **finishes** processing, one of three things happens, depending on the type of event that caused the exception:

- The handler **returns control to the current instruction**, i.e. the instruction that was executing when the event occurred.
- The handler **returns control to the instruction that would have executed next** had the exception not occurred.
- The handler **aborts** the interrupted program.

HERIOT WATT UNIVERSITY

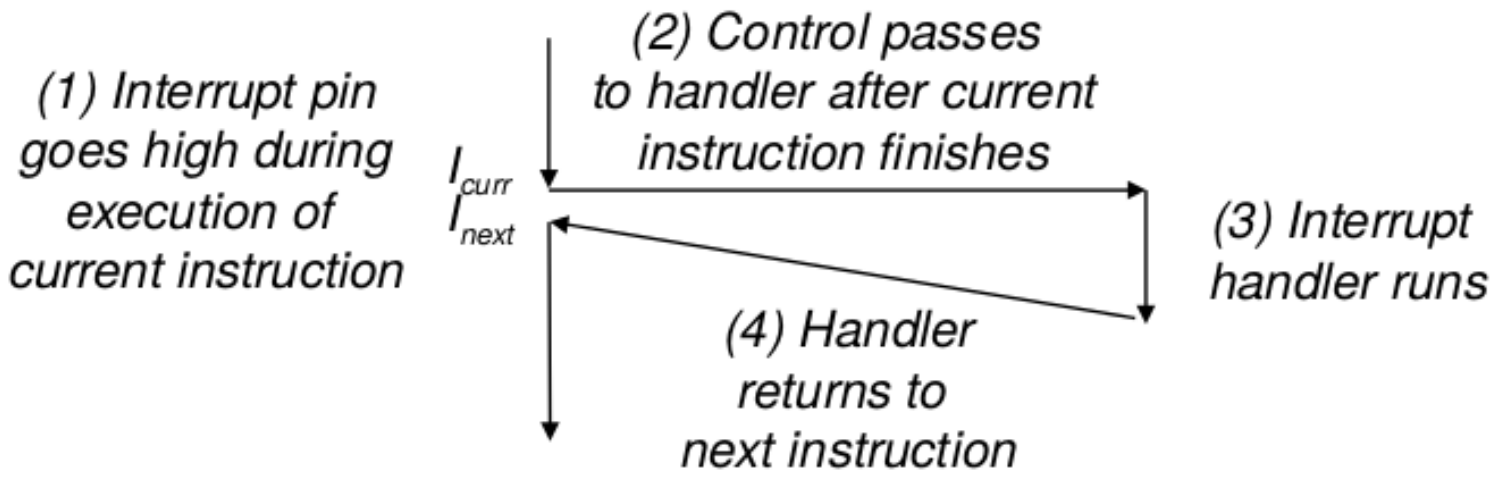
Exceptions (cont'd)

When the processor detects that the event has occurred, it makes an indirect procedure call (the **exception**), through a jump table called an **exception table**, to an operating system subroutine (the **exception handler**) that is specifically designed to process this particular kind of event.
When the exception handler **finishes** processing, one of three things happens, depending on the type of event that caused the exception:

- The handler **returns control to the current instruction**, i.e. the instruction that was executing when the event occurred.
- The handler **returns control to the instruction that would have executed next** had the exception not occurred.
- The handler **aborts** the interrupted program.

HERIOT WATT UNIVERSITY

Interrupt handling



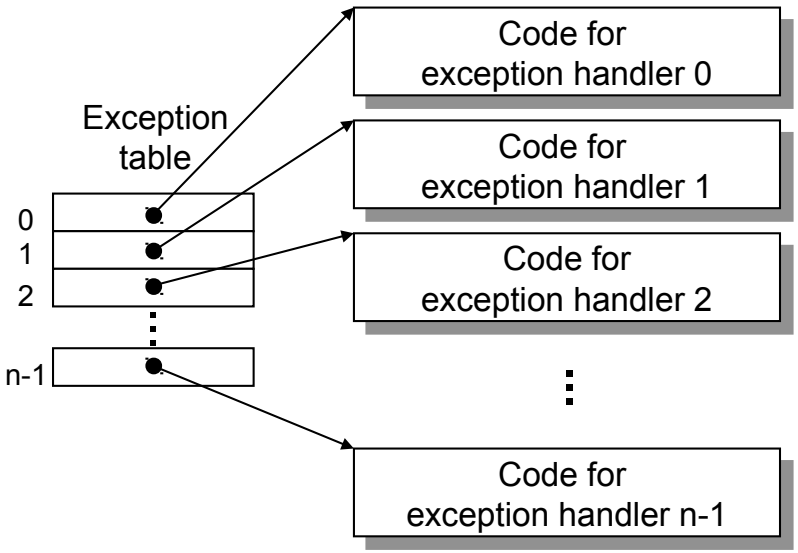
The interrupt handler returns control to the next instruction in the application program's control flow.

Exception Handling

Exception Handling requires close cooperation between software and hardware.

- Each type of possible exception in a system is assigned a unique nonnegative integer **exception number.**
- Some of these numbers are assigned by the designers of the processor. Other numbers are assigned by the designers of the operating system kernel.
- At system **boot time** (when the computer is reset or powered on), the operating system allocates and initializes a jump table called an **exception table**, so that entry k contains the address of the handler for exception k.
- At **run time** (when the system is executing some program), the processor detects that an event has occurred and determines the corresponding exception number k. The processor then triggers the exception by making an **indirect procedure call**, through entry k of the exception table, to the corresponding handler.

Exception table



The exception table is a jump table where entry *k* contains the address of the handler code for exception *k*.

Differences between exception handlers and procedure calls

Calling an exception handler is similar to calling a procedure/method, but there are some important differences:

- Depending on the class of exception, the return address is either the current instruction or the next instruction.
- The processor also pushes some additional processor state onto the stack that will be necessary to restart the interrupted program when the handler returns.
- If control is being transferred from a user program to the kernel, all of these items are pushed onto the kernel's stack rather than onto the user's stack.
- Exception handlers run in kernel mode, which means they have complete access to all system resources.

HERIOT WATT UNIVERSITY

Classes of exceptions

Exceptions can be divided into four classes: interrupts, traps, faults, and aborts:

Class	Cause	(A)Sync	Return behavior
Interrupt	Signal from I/O device	Async	Always returns to next instr
Trap	Intentional exception	Sync	Always returns to next instr
Fault	Potent. recoverable error	Sync	Might return to current instr
Abort	Nonrecoverable error	Sync	Never returns

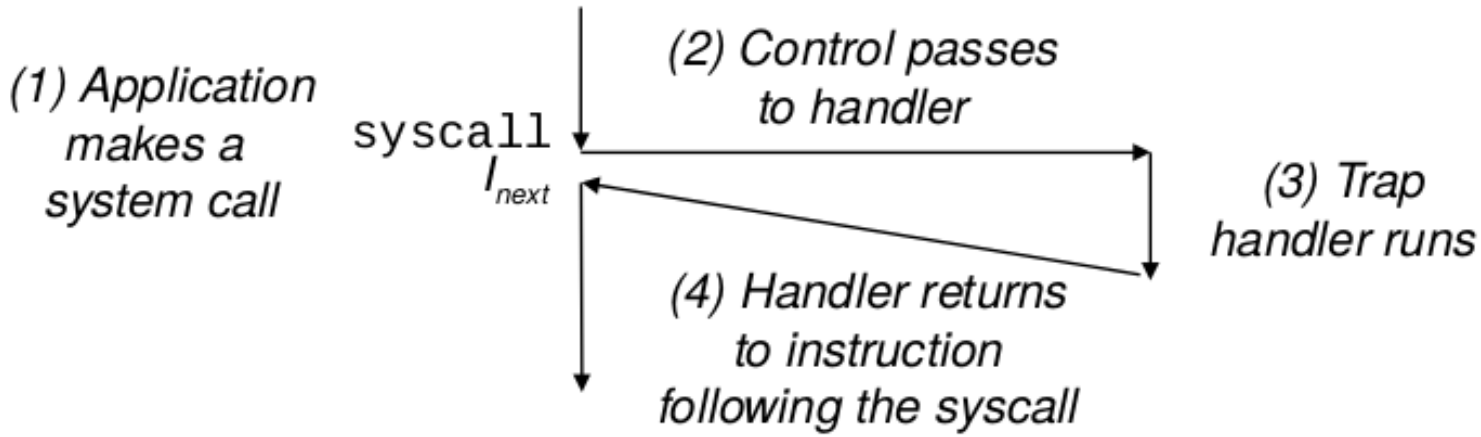
It is useful to distinguish 2 reasons for an exceptional control flow:

- an **exception** is **any** unexpected change in control flow;
  e.g. arithmetic overflow, using an undefined instruction, hardware timer
- an **interrupt** is an unexpected change in control flow triggered by an **external event**;
  e.g. I/O device request, hardware malfunction

Traps and System Calls

- **Traps** are **intentional exceptions** that occur as a result of executing an instruction.
- Traps are often used as an interface between application program and OS kernel.
- **Examples**: reading a file (`read`), creating a new process (`fork`), loading a new program (`execve`), or terminating the current process (`exit`).
- Processors provide a special *"syscall n"* instruction.
- This is exactly the `SWI` instruction on the ARM processor.

Trap Handling



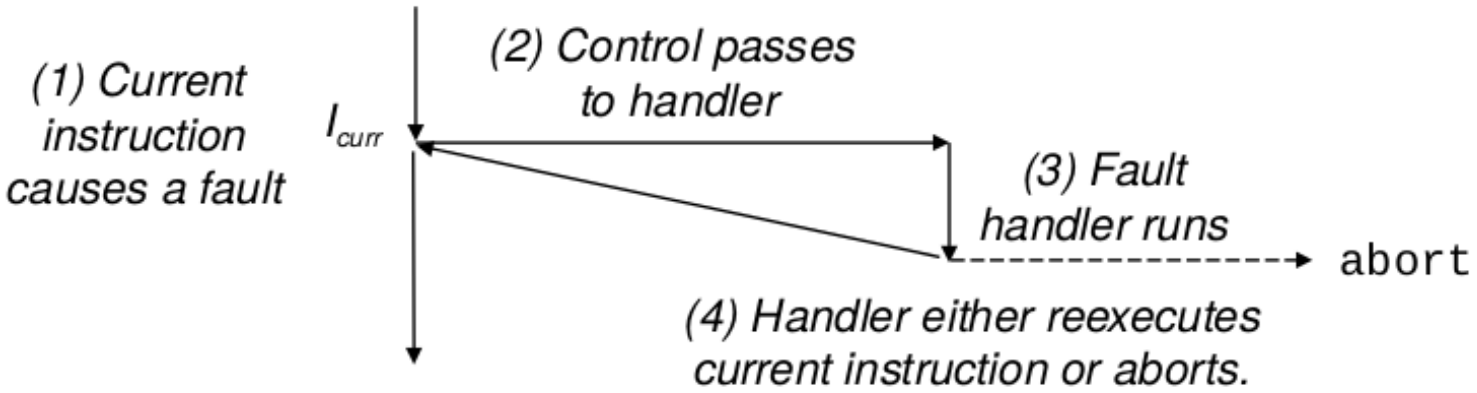
The **trap** handler returns control to the next instruction in the application program's control flow.

Faults

- **Faults** result from **error conditions** that a handler might be able to correct.
- Note that after fault handling, the processor typically reexecutes the same instruction.
- **Example**: page fault exception.
  - Assume an instruction references a virtual address whose corresponding physical page is not in memory.
  - In this case **page fault** is triggered.
  - The fault handler loads the required page into main memory.
  - After that **the same instruction** needs to be executed again.

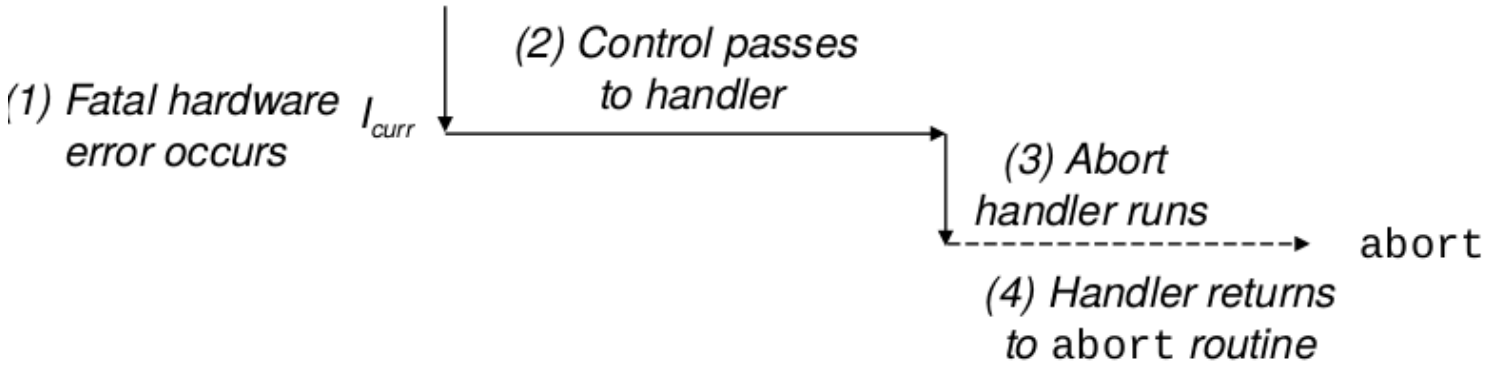
HERIOT WATT UNIVERSITY

Fault handling



Depending on whether the fault can be repaired or not, the fault handler either reexecutes the faulting instruction or aborts.

Aborts



Aborts result from unrecoverable fatal errors, typically hardware errors such as parity errors that occur when DRAM or SRAM bits are corrupted. Abort handlers never return control to the application program.

Common system calls

Number	Name	Description
1	exit	Terminate process
2	fork	Create new process
3	read	Read file
4	write	Write file
5	open	Open file
6	close	Close file
7	waitpi	Wait for child to terminate
11	execve	Load and run program
19	lseek	Go to file offset
20	getpid	Get process ID

[0] For a more complete list see Smith, Appendix B "Raspbian System Calls"

Common system calls

Number	Name	Description
27	alarm	Set signal delivery alarm clock
29	pause	Suspend process until signal arrives
37	kill	Send signal to another process
48	signal	Install signal handler
63	dup2	Copy file descriptor
64	getppid	Get parent's process ID
65	getpgrp	Get process group
67	sigaction	Install portable signal handler
90	mmap	Map memory page to file
106	stat	Get information about file

[0] For the truly complete list see `/usr/include/sys/syscall.h`

Signal handlers in C

UNIX **signals** are a higher-level software form of exceptional control flow, that allows processes and the kernel to interrupt other processes.

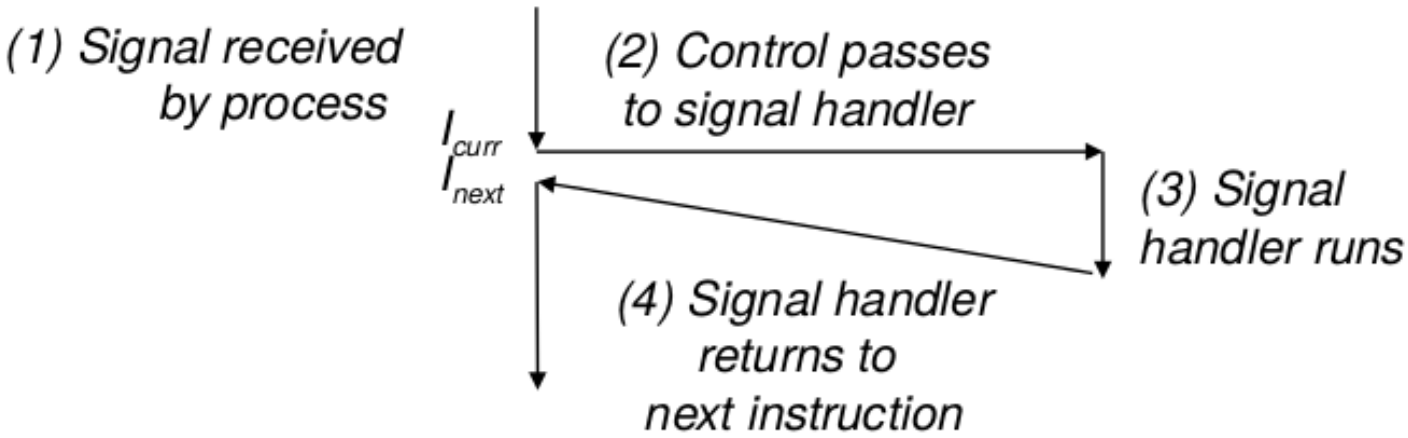
- Signals provide a mechanism for exposing the occurrence of such exceptions to user processes.
- For example, if a process attempts to divide by zero, then the kernel sends it a SIGFPE signal (number 8).
- Other signals correspond to higher-level software events in the kernel or in other user processes.

[0] From Bryant and O'Hallaron, Sec 8.5

Signal handlers in C (cont'd)

- For example, if you type a `ctrl-c` (i.e. press the ctrl key and the c key at the same time) while a process is running in the foreground, then the kernel sends a `SIGINT` (number 2) to the foreground process.
- A process can forcibly terminate another process by sending it a `SIGKILL` signal (number 9).
- When a child process terminates or stops, the kernel sends a `SIGCHLD` signal (number 17) to the parent.

Signal handling



Receipt of a signal triggers a control transfer to a signal handler. After it finishes processing, the handler returns control to the interrupted program.

Example: handling `ctrl-c`

```
// header files
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void ctrlc_handler(int sig) {
  fprintf(stderr, "Received␣signum␣%d;␣thank␣you␣for␣pressing␣
      CTRL-C\n", sig);
  exit(1);
}

int main() {
  signal(SIGINT, ctrlc_handler);  // install the signal handler
  while (1) { } ; // infinite loop
}
```

Example: handling `ctrl-c` in more detail

See `signal2.c`

Example: sending SIGALARM by the kernel

```
/* signal handler, i.e. the fct called when a signal is
    received */
void handler(int sig)
{
    static int beeps = 0;

    printf("BEEP␣%d\n", beeps+1);
    if (++beeps < 5)
        alarm(1); /* Next SIGALRM will be delivered in 1 second
            */
    else {
        printf("BOOM!\n");
        exit(1);
    }
}

int main() {
    signal(SIGALRM, handler); /* install SIGALRM handler; see:
        man 2 signal */
    alarm(1); /* Next SIGALRM will be delivered in 1s; see: man
```

Timers

- We now want to use timers, i.e. setting up an interrupt in regular intervals.
- The BCM2835 chip as an on-board timer for time-sensitive operations.
- We will explore three ways of achieving this:
  - using C library calls (on top of Raspbian)
  - using assembler-level system calls (to the kernel running inside Raspbian)
  - by directly probing the on-chip timer available on the RPi2
- In this section we will cover **how to use the on-chip timer to implement a simple timeout function in C**

HERIOT WATT UNIVERSITY

Overview

Features of the different approaches:

- C library calls (on top of Raspbian)
  - are **portable** across hardware and OS
  - require a (system) library for handling the timer
- assembler-level system calls (to the kernel running inside Raspbian)
  - **depend** on the OS, but are **portable** across hardware
  - require a support for software-interrupts in the OS kernel
- directly probing the on-chip timer available on the RPi2
  - **depend** on both hardware and OS
  - the instructions for probing a hardware timer are specific to the hardware

Example: C library functions for controlling timers

getitimer, setitimer - get or set value of an interval timer

```
#include <sys/time.h>

int getitimer(int which, struct itimerval *curr_value);
int setitimer(int which, const struct itimerval *
    new_value,
                  struct itimerval *old_value);
```

setitimer sets up an interval timer that issues a signal in an interval specified by the *new_value* argument, with this structure:

```
struct itimerval {
    struct timeval it_interval; /* next value */
    struct timeval it_value;    /* current value */
  };
  struct timeval {
    time_t      tv_sec;         /* seconds */
    suseconds_t tv_usec;        /* microseconds */
  };
```

C library functions for controlling timers

There are three kinds of timers, specified by the *which* argument:

- `ITIMER_REAL` decrements in real time, and delivers `SIGALRM` upon expiration.
- `ITIMER_VIRTUAL` decrements only when the process is executing, and delivers `SIGVTALRM` upon expiration.
- `ITIMER_PROF` decrements both when the process executes and when the system is executing on behalf of the process. Coupled with `ITIMER_VIRTUAL`, this timer is usually used to profile the time spent by the application in user and kernel space. `SIGPROF` is delivered upon expiration.

[0] See: `man getitimer`

Programming a C-level signal handler

Signals (or software interrupts) can be programmed on C level by associating a C function with a signal sent by the kernel.
sigaction - examine and change a signal action

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

The sigaction structure is defined as something like:

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

NB: the `sa_handler` or `sa_sigaction` fields define the action to be performed when the signal with the id `signum` is sent.

[0] See `man sigaction`

Programming Timers using C library calls

We need the following headers:

```
#include <signal.h>
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <sys/time.h>

// in micro-sec
#define DELAY 250000
```

[0] Sample source in itimer11.c

Programming Timers using C library calls

```
int main ()
{
 struct sigaction sa;
 struct itimerval timer;

 fprintf(stderr, "configuring␣a␣timer␣with␣a␣delay␣of␣%d␣
    micro-seconds␣...\n", DELAY);

 /* Install timer_handler as the signal handler for
    SIGALRM. */
 memset (&sa, 0, sizeof (sa));
 sa.sa_handler = &timer_handler;
 sigaction (SIGALRM, &sa, NULL);
```

Calling `sigaction` like this, causes the function `timer_handler` to be called whenever signal `SIGALRM` arrives.

Programming Timers using C library calls

Now, we need to set-up a timer to send `SIGALRM` every `DELAY` micro-seconds:

```
 /* Configure the timer to expire after 250 msec... */
 timer.it_value.tv_sec = 0;
 timer.it_value.tv_usec = DELAY;
 /* ... and every 250 msec after that. */
 timer.it_interval.tv_sec = 0;
 timer.it_interval.tv_usec = DELAY;
 /* Start a real timer. It counts down whenever this
    process is executing. */
 setitimer (ITIMER_REAL, &timer, NULL);

 /* A busy loop, doing nothing but accepting signals */
 while (1) {} ;
}
```

[0] Sample source in itimer11.c

Further Reading & Hacking

- Randal E. Bryant, David R. O'Hallaron *"Computer Systems: A Programmers Perspective"*,
  3rd edition, Pearson, 7 Oct 2015. ISBN-13: 978-1292101767.
  Chapter 8: Exceptional Control Flow
- David A. Patterson, John L. Hennessy. *"Computer Organization and Design: The Hardware/Software Interface"*,
  ARM edition, Morgan Kaufmann, Apr 2016. ISBN-13: 978-0128017333.
  Section 4.9: Exceptions
- Stewart Weiss. *"UNIX Lecture Notes"*
  Chapter 5: Interactive Programs and Signals
  Department of Computer Science, Hunter College, 2011

HERIOT WATT UNIVERSITY

Summary

- **Interrupts trigger an exceptional control flow**, to deal with special situations.
- Interrupts can occur at several levels:
  - hardware level, e.g. to report hardware faults
  - OS level, e.g. to switch control between processes
  - application level, e.g. to send signals within or between processes
- The **concept** is the same on all levels: execute a short sequence of code, to deal with the special situation.
- Depending on the source of the interrupt, execution will continue with the same, the next instruction or will be aborted.
- The **mechanisms** how to implement this behaviour are different: in software on application level, in hardware with jumps to entries in the interrupt vector table on hardware level

Gitlab repo Signals and Timers