Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

System Construction Course 2019,

Assignment 6

Felix Friedrich, ETH Zürich

Lessons to Learn

- Learn how to debug a kernel using virtual environments.
- Understand call chains and stack traces.
- Understand synchronization semantics of Active Oberon.

1 Debugging a Kernel

Preparation

1. Install a recent version of the Bochs emulator, .e.g from `http://bochs.sourceforge.net/`. Ubuntu linux users install it using `apt-get install bochs bochs-x`.
2. Open a console in directory assignments/assignment06

The first part of this lab is about debugging a kernel using the A2-built in tracing features and a hardware emulating tool (Bochs).

Find the Bugs!

For this lab we have prepared an implementation of the A2 kernel together with build scripts to set it up and run it in a virtual machine. The system reports a successful boot with the following output

```
A2 Test System
Successfully booted
```

You will *not* see this report in the first place because we have injected bugs into the kernel that prevent it from booting successfully. Find and correct the bugs!

Guidelines:

1. Use the script in file makeA2 in order to to compile and link the boot-file and to inject the files into a bootable HDD image by calling `oberon execute makeA2` or, equivalently, by executing `System.DoFile makeA2` within the Oberon shell.

   The linker log file `linker.log` can contain valuable information about the arrangement. Have a look at it!
2. Use the hardware emulator Bochs (2.6.8) for starting and debugging the kernel.
   (a) Windows users start the system by clicking a2.bxrc. If you right-click this file, using the context menu you can start the debugging mode of Bochs. Use the command "help" to find out about facilities of the debugger.
   (b) Linux users start the system by executing `bochs -f linuxBochsSettings.txt`. The Linux version starts in debugging mode. In order to start the emulation, enter `c` (for **c**ontinue).
3. The log of A2 will be written to the serial port. Bochs redirects it to the file `a2.log`. Hint: Use the log file for tracing the kind of errors that provide a trap stack trace-back report. Stack trace-backs are described in the next section of this document.

- Further hints: when in debugging mode, you can interrupt a running system with Ctrl-C (typed at the debugging console of Bochs). Make use of time-breakpoints in Bochs, when you cannot locate the exact location of a problem. Use the linker-log to find out where you are with respect to the source code.

Understanding a Trap Traceback

When you (later on) run code in $\mathcal{A}_2$, it can happen that you see a red window popping up. Such a red window indicates that something went wrong. Usually it happens as a result of an unhandled runtime exception that needs intervention, such an array index out of bounds, nil pointer access, programmed halt, assert failed etc. During startup of a kernel the information is displayed on the text console and / or written to other debug channels such as a serial port.

The following module, for example, will produce a trap when `TrapExample.Test` is executed.

```
MODULE TrapExample;

VAR a: ARRAY 2 OF CHAR;

  PROCEDURE Test2(x: SIZE);
  BEGIN
    a[x] := "A"; (* if x exceeds the length of a, this will lead to a trap *)
  END Test2;

  PROCEDURE Test*;
  BEGIN
    Test2(10);
  END Test;

END TrapExample.
```

The trap output can be used to diagnose the history of a trap. Inspection of the trap is also referred to as *Post Mortem Debugging*. A trap starts with information on the trap number and reason (here: index out of range). Then there is more general information on the system release followed by the state of the registers and flags. After that we see the process ID and the active object that is associated with the process (here: `Commands.Runner`).

```
LinuxA2 Gen. 64-bit, Oct 26 2019 2019/10/26 15:34

Trap 5.7   (index out of range)

SP = 00007F79FA0A8430 FP = 00007F79FA0A8448 PC = 00007F79770C38DD

RAX = 0000000000000000 RBX = 00007F797704B9C0 RCX = 00007F79FFC3E320 RDX = 00007F79FA0A9A20
RSI = 00007F797704B9C0 RDI = 000000000000000A R8 = 0000000000000000 R9 = 00007F79640075E0
R10 = 0000000000000000 R11 = 0000000000200246 R12 = 00007F79FA0A8FC0 R13 = 0000000000000000
R14 = 000000000807FC60 R15 = 00007F79FA0664E8

Process:   85 run 0 000007F7977049890:Commands.Runner NIL {0}
```

After this prolog starts the stack trace. The runtime builds this information by traversing the stack frames from top to bottom. Read from bottom to top (lines 7,5,4,2), it shows how procedures were called. In our example it starts with `Objects.Wrapper`. `Objects.BodyStarter` executed the body of object `Commands.Runner` that called `TrapExample.Test` which itself called

`TrapExample.Test2`. This is where the trap occured. More specifically, at instruction offset 44 (bytes) relative to the start of `TrapExample.Test2`. The offset can be utilized to determine the exact location of a trap both in binary code but also, using the compiler, in source code. In a kernel output, the `pc=number [hex number]` shows the location of the program counter as absolute value and therefore allows also to examine where the trap happened by comparison with the linker script.

Between procedure and module names we see other names followed by an equal sign. They denote the variables and parameters of the respective procedures. For example, in procedure `TrapExample.Test2`, variable `x` had a value of 10, ultimately causing the index out of bound trap.

```
StackTraceBack:
TrapExample.Test2:44 pc=00007F79770C38DD fp=00007F79FA0A8448 crc=D3BC8C17
  x= [@16] 10
TrapExample.Test:39 pc=00007F79770C3919 fp=00007F79FA0A8470 crc=D3BC8C17
Commands.Runner.@Body:790 pc=00000000080A3A85 fp=00007F79FA0A8490 crc=6208ACC3
  @Self= [@16] 00007F7977049890 (Commands.Runner)
Objects.BodyStarter:714 pc=000000000807FF2A fp=00007F79FA0A84D8 crc=97D54EE8
  p= [@−16] 00007F797704B100 (Objects.Process)
  res= [@−20] 0
  sp= [@−32] 00007F79FA0A84A8
```

2 A Recursive Lock

This second part of the exercise does only work in case you have solved part 1.

This second part of this lab is to learn how to use the language constructs of Active Oberon for process synchronisation in A2.

Critical sections can be executed by at most one process at a time. In A2, only non recursive locks are implemented, which means that a process can only enter a critical section once, even if it holds the lock. Implement a recursive lock to allow a process to re-enter a critical section recursively.

The intended usage is like this:

```
VAR lock: RecursiveLock;

NEW(lock);
...
lock.Acquire();
... (* critical section (without AWAIT) *)
lock.Release()
```

We provide some initial context with module Locks.Mod

Module TestLocks.Mod serves as a testing program.

Once, successfully implemented, the kernel log should end with lines like the following:

```
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20
```

The following command can be used in order to build and run the test example (in Linux):

```
./oberon execute makeTestLocks && bochs −f linuxBochsSettings.txt
```

Hints:

- An `AWAIT` statement must always be placed in an `EXCLUSIVE` section.
- Condition evaluation of all waiting conditions takes place when a process exits the monitor. This implies that statements that change the state of a condition of an `AWAIT` statement should be put into an `EXCLUSIVE` section.
- Condition evaluation potentially takes place in the context of a different thread.
- A pointer to the currently running process can be acquired by the procedure `Objects.ActiveObject()` You may assign it to a variable of type `ANY`, for example.

Documents

- System Construction Lecture 6 slides from the course-homepage
  `http://lec.inf.ethz.ch/syscon`
- $\mathcal{A}_2$ Programming Quickstart Guide. File A2QuickStartGuide.pdf in folder documents/oberon