

**Assignment 6**Felix Friedrich, ETH Zürich, 20.10.2015

---

# 1 Introduction to the synchronisation concept of $\mathcal{A}_2$

**Lessons to Learn**

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

## Preparation

1. Update your repository. Copy new and modified files / directories from the `Work` folder located in [\(repo\)/a2/](#) to your work directory.
2. Recompile the compiler: Within `A2`, open file `Fox/Fox.Tool` and execute the contained compile command.
3. Install a recent version of the Bochs IA 32 emulator, .e.g from <https://sourceforge.net/projects/bochs/files/bochs/2.6.8/>.

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

## Recursive Locks

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 `Assignment4/RecursiveLocks.Mod`.

Module `Assignment4/RecursiveLocksTest.Mod` serves as a testing program.

The tool file `Assignment4/Assignment4a.Tool` contains the most frequently used commands.

Hints:

- Module unloading is done with the command `SystemTools.Free <moduleNames> ~`.
- 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 `OBJECT`.
- Should you see a red trap window popping up, please read the section below about trap trace backs in order to understand what happened.

## 2 Debugging a Kernel

The second part of the 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 hardware emulator Bochs (2.6.8) for starting and debugging the kernel.
2. Use the script contained `Assignment6/AssignmentB.Tool` in order to compile and link the boot-file and to inject the files into a bootable HDD image.
  - Source files are contained in `src/`
  - The linker log file `linker.log` contains valuable information about the arrangement. Have a look at it!
3. Start the system by executing `a2.bxrc` in folder `Assignment6`.
  - 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.
4. The system starts with a boot manager. Press key "1" to start the system.
  - If you enter "b" before, the next time Bochs starts it will not show the boot-manager again. The latter may be important to get a consistent timing behaviour at the beginning, in particular if you want to make use of time breakpoints.
5. 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 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;

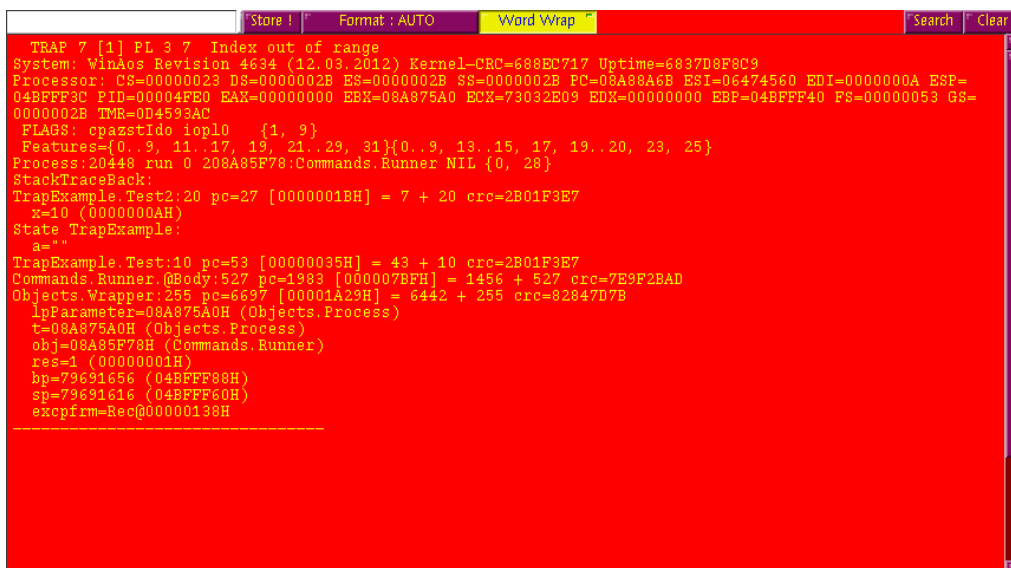
VAR a: ARRAY 2 OF CHAR;

PROCEDURE Test2(x: LONGINT);
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.
```

In  $\mathcal{A}_2$  the result is a red window popping up that looks like this:



```
TRAP 7 [1] PL 3 7 Index out of range
System: WinAos Revision 4634 (12.03.2012) Kernel-CRC=688EC717 Uptime=6837D8F8C9
Processor: CS=00000023 DS=0000002B ES=0000002B SS=0000002B PC=08A88A6B ESI=06474560 EDI=0000000A ESP=
04BFFF3C PID=00004FE0 EAX=00000000 EBX=08A875A0 ECX=73032E09 EDX=00000000 EBP=04BFFF40 FS=00000053 GS=
0000002B TMR=0D4593AC
FLAGS: cpazstIdo iopl0 {1, 9}
Features={0..9, 11..17, 19, 21..29, 31}{0..9, 13..15, 17, 19..20, 23, 25}
Process:20448 run 0 208A85F78:Commands.Runner NIL {0, 28}
StackTraceBack:
TrapExample.Test2:20 pc=27 [0000001BH] = 7 + 20 crc=2B01F3E7
x=10 (0000000AH)
State TrapExample:
a=""
TrapExample.Test:10 pc=53 [00000035H] = 43 + 10 crc=2B01F3E7
Commands.Runner.@Body:527 pc=1993 [000007BFH] = 1456 + 527 crc=7E9F2BAD
Objects.Wrapper:255 pc=6697 [00001A29H] = 6442 + 255 crc=82847D7B
lpParameter=08A875A0H (Objects.Process)
t=08A875A0H (Objects.Process)
obj=08A85F78H (Commands.Runner)
res=1 (00000001H)
bp=79691656 (04BFFF88H)
sp=79691616 (04BFFF60H)
excpfrm=Rec@00000138H
```

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`).

```

1 TRAP 7 [1] PL 3 7 Index out of range
2 System: WinAos Revision 4634 (12.03.2012) Kernel_CRC=688EC717
   Uptime=C17E0CB3B7
3 Processor: CS=00000023 DS=0000002B ES=0000002B SS=0000002B PC=08C810CB ...
4 FLAGS: cpazstIdo iopl0 {1, 9}
5 Features={0..9, 11..17, 19, 21..29, 31}{0..9, 13..15, 17, 19..20, 23, 25}
6 Process:20956 run 0 208C7D2F8:Commands.Runner NIL {0, 28}

```

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 8,7,6,2), it shows how procedures were called. In our example it starts with `Objects Wrapper`. `Objects Wrapper` executed the body of object `Commands.Runner` that called `TrapExample.Test` which itself called `TrapExample.Test2`. This is where the trap occurred. More specifically, at offset 20 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. At line 4 we see state information about the module involved in the trap. 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.

```

1 StackTraceBack:
2 TrapExample.Test2:20 pc=27 [0000001BH] = 7 + 20 crc=2B01F3E7
3   x=10 (0000000AH)
4 State TrapExample:
5   a=""
6 TrapExample.Test:10 pc=53 [00000035H] = 43 + 10 crc=2B01F3E7
7 Commands.Runner.@Body:527 pc=1983 [000007BFH] = 1456 + 527 crc=7E9F2BAD
8 Objects.Wrapper:255 pc=6697 [00001A29H] = 6442 + 255 crc=82847D7B
9   lpParameter=08C7E920H (Objects.Process)
10  t=08C7E920H (Objects.Process)
11  obj=08C7D2F8H (Commands.Runner)
12  res=1 (00000001H)
13  bp=76087176 (0488FF88H)
14  sp=76087136 (0488FF60H)
15  excpfrm=Rec@00000138H
16 -----

```

## Documents

- [System Construction Lecture 6](http://lec.inf.ethz.ch/syscon) slides from the course-homepage <http://lec.inf.ethz.ch/syscon>
- The Programming Language Oberon. File [OberonReport.pdf](#) in folder `documents/oberon`
- $\mathcal{A}_2$  Programming Quickstart Guide. File [A2QuickStartGuide.pdf](#) in folder `documents/oberon`
- Active Oberon Language Report. File [ActiveReport.pdf](#) in folder `documents/oberon`