Norwegian University of Science and Technology

Operating Systems

Example solutions for Theoretical Exercise 8

Michael Engel

8.1 Emulation and JIT compilation

Options for binary compatibility (run x86-64 programs on Aarch64):

- Recompile all programs for the Aarch64 target instruction set
- Translate compiled binaries from x86-64 machine code to Aarch64

Find out why, in the common case, it is not possible to statically translate a binary executable from one instruction set to the other.

A static binary translation would work like a translation of a book, e.g. from bokmål to German. Such a translation can never be perfect and tends to lose semantic details or introduce different possible meaning or interpretation in the translated text.

The semantics of computer code is in general defined much more precisely, but there can be a number of problems when trying to translate (i.e. *compile*) a program statically.

8.1 Emulation and JIT compilation

1. **Code discovery problem**

Depending on the architecture (especially for variable-length machine instructions such as x86-64), it is difficult to *predecode* (disassemble) the whole program in advance

Consider this x86 code sequence:

```
                  | mov  %ch,0 ??
31 c0 | 8b | b5 00 00 03 08 8b bd 00 00 03 00
      |      movl  %esi, 0x08030000(%ebp)  ??
```

If 8b is decoded as the first byte of an instruction, it's a `movl`, if b5 is the first most, it's a different `mov` instruction

2. **Code location problem**

Mapping of the source program counter to the destination PC for indirect jumps – indirect jump addresses in the translated code still refer to source addresses for indirect jumps

x86 source code

```
movl  %eax, 4(%esp)  ;load jump address from memory
jmp   %eax           ;jump indirect through %eax
```

PowerPC target code

```
addi  r16,r11,4      ;compute x86 address
lwzx  r4,r2,r16      ;get x86 jump address
                     ;  from x86 memory image
mtctr r4             ;move to count register
bctr                 ;jump indirect through ctr
```

More information in case you are interested:

- An article about Apple's Rosetta2 binary translation
  https://ffri.github.io/ProjectChampollion/
  https://github.com/FFRI/ProjectChampollion
- An Emulator Writer's HOWTO for Static Binary Translation – http://www.gtoal.com/sbt/

8.2 Virtual memory ballooning

Assuming no swap space is configured, explain why the memory ballooning approach can work.

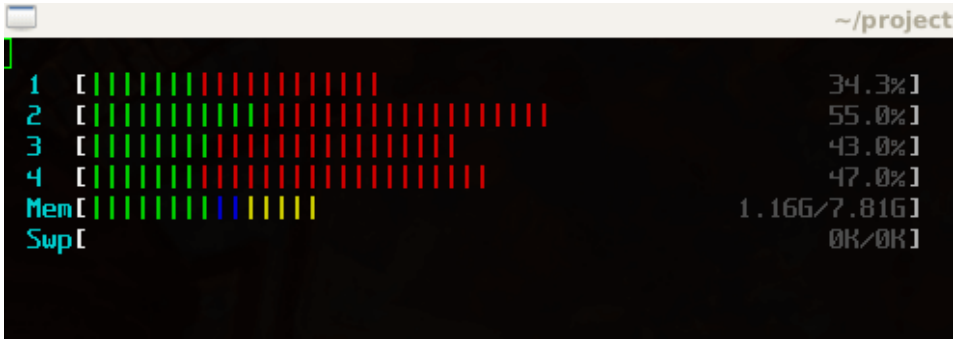
Memory ballooning is simply an overallocation of memory. If you have 16 GB of RAM and three VMs allocating 6 GB each (so 18 GB overall), it is very unlikely that all three VMs use the maximum amount of memory at the same time.

The virtual machine's kernel implements a "balloon driver" which allocates unused memory within the VM's address space into a reserved memory pool (the "balloon") so that it is unavailable to other processes on the VM.

This works since in many operating systems, a significant amount of physical memory is free in regular operation.

In htop, the "Mem" entry shows the amount of memory allocated for different functionality (here shown for Linux): Green = Used memory, Blue = Buffers, Yellow/Orange = Cache

MacOS allocates all free memory for buffers/caches, so the memory use in htop always appears "full".



8.3 Containers

Give three examples of missing functionality in chroot to implement container functionality as provided by Docker and explain why the functionality is required

(we give four here, there may be even more…)

1. **chroot does not hide other processes in the system**

   A process running inside the chroot environment can see activities outside the chroot

2. **chroot does not prohibit the execution of specific syscalls**

   Often, you want to limit process functionality, e.g. to accept network connections. This is not provided by chroot

3. **Escaping chroot "jails" is simple**

   If you manage to obtain root permissions inside a chroot, you can execute chroot again to break out

4. **Insufficient isolation against other users**

   Chroot does not isolate resources (memory, CPU time, size of files…), so a user inside of a chroot environment can e.g. start a denial-of-service (DoS) attack

In general, chroot is *not* a security feature (but might be used as a part of if)!
See also: Container from scratch: Using chroot to isolate the filesystem

https://kevinboone.me/containerfromscratch_chroot.html

8.4 Popek and Goldberg Criteria

Find out why the Motorola 68000 `MOVE from SR` instruction is problematic and how a virtual machine could exploit this instruction.

This instruction is sensitive because it allows access to the entire status register, which includes not only the condition codes but also the user/supervisor bit, interrupt level, and trace control.

Thus, an OS or other code running as guest OS under a hypervisor could find out it is not running natively and behave differently. E.g., malware could only become active when running on real hardware.

In most later 68000 family members, starting with the MC68010, the MOVE from SR instruction was made privileged, and a new MOVE from CCR instruction was provided to allow access to the condition code register only

8.5 Privilege Modes

Find out why more than two privilege levels could be useful and in which ways they were used in the Windows NT 3.5 and VMS operating system.

These additional levels (called "rings" on x86) are used to isolate components of the OS that need access to the hardware from the kernel itself. This is usually done for device drivers, which are often provided by third parties and not completely trustworthy

Running kernel code in an additional level has a performance penalty, since mode switches are now required between driver and OS kernel.

Windows NT up to version 3.51 ran drivers outside of the most privileged "ring 0", which resulted in a severe performance problem. In NT 4, especially graphics drivers were moved back to ring 0.

The VMS OS on DEC VAX hardware supports four "access modes", used as follows:

1. kernel (scheduling, I/O operations, memory management)
2. executive (file subsystem)
3. supervisor (shell)
4. user (user programs, compilers, editors, etc.)

See also https://people.cs.clemson.edu/~mark/syscall/vax.html and http://www.bitsavers.org/pdf/dec/vax/vms/4.0/AA-Z102A-TE_VAX_VMS_4.0_Glossary_198409.pdf

8.6 Shadow Page Tables

Describe what happens in case of a page fault in a virtual machine on a system that employs a hypervisor using shadow page tables. Which lookups are performed, which parts of which of the page tables are accessed?

Because most operating systems use paged virtual memory, granting the guest OS direct access to the MMU would mean loss of control by the virtualization manager.

Thus, some of the work of the (e.g. x86) MMU needs to be duplicated in software for the guest OS using shadow page tables. This involves denying the guest OS any access to the actual page table entries by trapping access attempts and emulating them instead in software.

See also https://www.vmware.com/pdf/asplos235_adams.pdf

8.6 Shadow Page Tables

Describe what happens in case of a page fault in a virtual machine on a system that employs a hypervisor using shadow page tables.

Which lookups are performed, which parts of which of the page tables are accessed?

Rather than describing here shortly what happens, please take a look at https://courses.engr.illinois.edu/cs423/fa2011/lectures/lect35-virt2.pdf