Assignment #2
CS695 Spring 2023-24

Topic: Enter the VM

Statutory note:

- This course is on a ***no-plagiarism*** diet.
- All parties involved in plagiarism harakiri will be penalized to the maximum extent.
- The Moss Detective Agency has agreed to conduct all investigations. https://theory.stanford.edu/~aiken/moss/
- Byomkesh, Sherlock, Phryne, Marple, and Hercule are on standby.
- Hardcoding the runtime output in the code will be heavily penalized.
- Generative AI (ChatGPT, Gemini, etc.) is your friend, but it cannot generate outputs for you to submit. If you want to submit such generated outputs, mark them explicitly in red in your submissions.
- Warning: Submission Guidelines should be strictly followed; otherwise, your submission will not count. (0 Marks)

0. Introduction

Virtualization techniques are essential for Cloud computing and have evolved significantly to address the challenges posed by complex architectures such as x86. Para and Full virtualization techniques you have learned in class do not perform well and do not satisfy the principles of VMM design (Popek & Goldberg [1]). This is because x86 is very difficult to virtualize. Thus, hardware vendors now support x86 hardware virtualization natively.

KVM or Kernel-based Virtual Machine allows you to create hypervisors in Linux, which can be controlled by userspace programs to run guest VMs while handling x86

KVM API allows userspace hypervisors to perform the following operations:

- Creation of new virtual machines
- Allocation of memory to virtual machines.
- Reading and writing virtual CPU registers.
- Injecting and interrupting into a virtual CPU.
- Running a virtual CPU.

References:

- KVM Paper: [link1]
- Linux KVM API: [link2] [link3]

The userspace program relies on the KVM hypervisor for the x86 architecture virtualization. Still, it has to implement its own IO handling and device support — block IO, network IO, console drivers, USB controller, host file system, etc. In practice, QEMU is used as a userspace program alongside KVM, which implements (emulates) all these components and can run any OS as a guest.

Setup Procedure (for Ubuntu): **https://help.ubuntu.com/community/KVM/Installation**
Warning: Running KVM inside a VM requires nested virtualization and is not recommended. Please use your Linux machines to run the assignment.

Download the Assignment code tarball: link

Bonus: You can try out QEMU + KVM to run any x86-64 OS of your choice in Linux KVM https://ubuntu.com/server/docs/virtualization-virt-tools

1a. DIY Hypervisor

In this assignment, you will build your VM using KVM from ~~scratch~~.

a lot of boilerplate code with a simple hypervisor implemented.

a full-fledged OS. Therefore, the guest OS in `guest.c` is a single stream of instructions, with no separation between the guest user space and the guest OS.

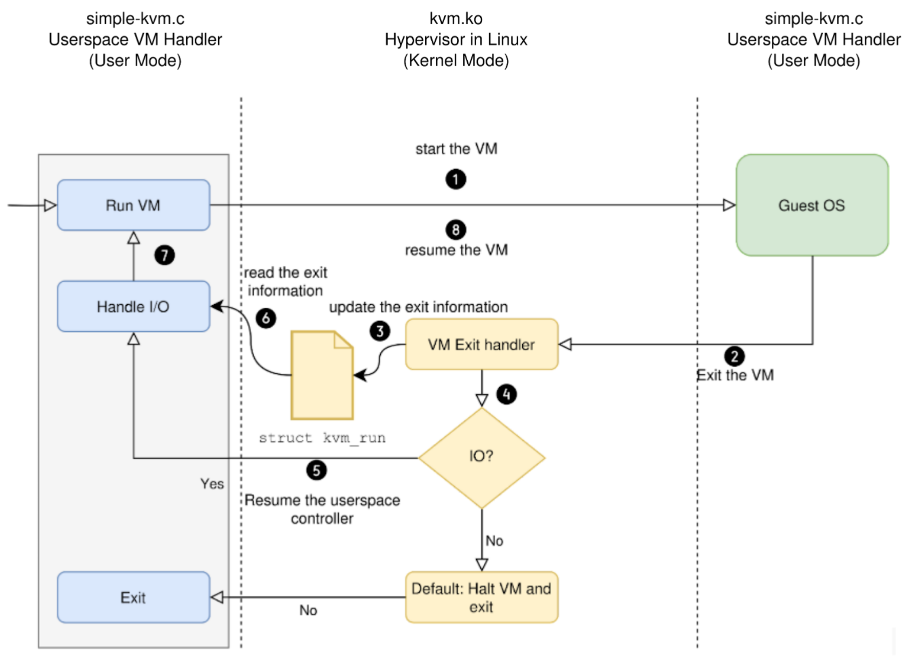

Figure 1: VM Run Loop in `simple-kvm.c`

The userspace hypervisor program `simple-kvm.c` uses the `/dev/kvm` file to create a virtual machine, allocate memory and CPU. The program then starts a KVM run loop that starts a `vCPU` using KVM_RUN and continuously handles the `VM_EXIT` routines when the `vCPU` returns from KVM_RUN, as shown in Figure 1. A simple print character operation by a guest translates to a write on port `0xE9` as a hypercall. The hypercall implementation for this setup uses the write IO operation to trigger a guest OS VM_Exit, and the exit cause is saved. The hypervisor checks that the VM exited trying to do `IO_OUT` operation on port `0xE9`, and then the userspace handler of the hypervisor prints the value written to the port.

The x86 architecture has different operating modes: real mode, modes, and the mode can be selected with a command line argument. While you are encouraged to check the setup of all the modes, we will focus on the long mode for the first part of the assignment.

1. Go through the code and KVM APIs to understand the workings of the hypervisor. With the help of a well-drawn, labeled, and descriptive **flowchart,** explain the logical actions **(at least ten actions)** to setup and execute a VM in the **long mode of operation**.

   Note that this step needs a flowchart diagram; no other output will be considered valid.

2. For each of the logical actions **(at least ten)** mentioned in the above flow chart, describe the KVM APIs associated with the actions and their purpose in your own words.

   e.g.,

   1) Set up the physical memory for the guest OS… To assign the memory to the guest OS, the KVM_YYY_ZZZ ioctl call is used.

   KVM_YYY_ZZZ:
   Brief description of inputs to the ioctl call
   Functionality that the ioctl call provides, etc.
   …

Bonus: If you want to learn more about a full hypervisor implementation, you can refer to kvmtool, a lighter hypervisor than QEMU.
https://github.com/kvmtool/kvmtool

its own, it exits to the hypervisor with the help of KVM. The user space handler can then determine the exit reason and direction of the data and handle it correspondingly.

Note: The boilerplate code already implements an example hypercall `HC_print8bit();` which writes a byte value to the port `0xE9` using assembly instruction `outb`.

Implement the following hypercalls to extend your simple-kvm.c to be not-so-simple.

1. **`void HC_print32bit(uint32_t val)`**
   This hypercall should take a 32-bit value as input, and the hypervisor should handle the hypercall by printing the 32-bit value on the terminal. Make sure to use a single newline `“\n”` or `“endl”` after printing the 32-bit value inside the hypervisor.

2. **`uint32_t HC_numExits()`**
   The hypercall should return the number of times the guest VM has exited to the hypervisor from the beginning of its execution. The hypervisor should maintain the count and return it to the guest. Since the guest can’t print directly on the terminal, please use `HC_print32bit()`, implemented in the previous step, to print the count.

3. **`void HC_printStr(char *str)`**
   The hypervisor should handle the hypercall by printing the string pointed by `str`. But unlike the example

the string from the guest's memory. Use `HC_numExits` to count the instances that exist before and after the hypercall. The count should change by one due to `HC_printStr()`.

Note: You should not add any extra new lines. If the string passed by the guest has a new line, it should be printed.
- The output of the simple-kvm should show the difference of two exits (one for `HC_numExits` and one for `HC_printStr`)

4. **`char * HC_numExitsByType()`**
   The hypervisor should handle the hypercall by returning an address to a string of the following format:
   `IO in: x`
   `IO out: y`
   The values `x` and `y` should be the actual values of VM exits due to IO in and IO out. The `char *` value should then be passed to `HC_printStr` for printing via hypercall.

5. **`uint32_t HC_gvaToHva(uint32_t gva)`**
   The hypercall should return the Host Virtual Address (HVA) corresponding to a given Guest Virtual Address (GVA). If the guest asks to translate an invalid GVA for which no HVA exists, the hypervisor should print "Invalid GVA" on the terminal and return 0. Demonstrate at least one valid and one invalid GVA to HVA conversion. Use `HC_print32bit` to print the HVA value.

- `outt` and `in` assembly instructions should be used for writing and reading 32-bit values from a serial port.

Note: To implement multiple hypercalls, you may use a different serial port for each hypercall or a struct with a field with a hypercall number and all arguments to the hypercall. In the second method, the struct will be known to the guest, and the hypervisor and the guest will pass the address to the struct for every hypercall with the correct hypercall number and the arguments set. The hypervisor can now read the hypercall number and the argument from the guest's memory.
The implementation choice is left to you.

2. Build the matrix cloud

The architect (owner) of the matrix cloud generously offers users powerful bare-metal systems for their use. The architect was buying new machines for new users joining the matrix, and it became clear that many users were not using their powerful machines at maximum capacity. Inspired by insights gained in CS695, the architect decided to create a next-generation matrix cloud by moving users' systems to virtual machines, unlocking various advantages of virtualization.

The architect requires your assistance to create a hypervisor capable of hosting two guest operating systems belonging to Neo and Morpheus within a single machine (with a single CPU). Here's the twist: Neo, under a special agreement (SLA) for high availability, claims 70% of the scheduling time (since he is the one), while Morpheus gets 30%.

The architect was able to create a starter code for the cloud but needs

The starter code has `matrix.c`, which is the custom KVM hypervisor, sets up and runs two guest programs, `guest1.s` and `guest2.s`, inside the respective VMs. Each VM is set up with one vCPU each, and the startup code can configure them to run concurrently (via the Linux scheduler, scheduling each of the vCPU (pthread) threads independently).

Note that **matrix.c** is the control program and cannot proactively pre-empt or de-schedule the VMs. It has to wait for VM_EXITs for control to reach back to it. The guest VMs have the following functionalities —

- `guest1.s`
  Issues a hypercall (using port 0x10) with the value in the **ax** register, and then on return from hypercall increment value stored in the **ax** register, and do this in an infinite loop.

- `guest2.s`
  Issues a hypercall (using port 0x11) with the value in the **ax** register, and then on return from hypercall decrement value stored in the **ax** register, and do this in an infinite loop.

The size of the `guest1.s/guest2.s` is 7 bytes, which will be printed when the guest program is loaded in the user memory. Note that both VMs execute in **real** mode.

Note: The userspace program uses `pthreads` to handle VM exits in separate user threads. These threads are not to be confused with per vCPU threads that Linux uses for scheduling the VMs.

```
Program with size: 7
VMFD: 4 started running
VMFD: 5 started running
VMFD: 4 stopped running - exit reason: 2
VMFD: 4 KVM_EXIT_IO
VMFD: 4 out port: 16, data: 0
VMFD: 5 stopped running - exit reason: 2
VMFD: 5 KVM_EXIT_IO
VMFD: 5 out port: 17, data: 0
VMFD: 4 started running
VMFD: 5 started running
VMFD: 4 stopped running - exit reason: 2
VMFD: 4 KVM_EXIT_IO
VMFD: 5 stopped running - exit reason: 2
VMFD: 5 KVM_EXIT_IO
VMFD: 5 out port: 17, data: 65535
VMFD: 4 out port: 16,
```

controlled with a sleep statement in the user program. Otherwise, the output would have been even more randomized.

[The initial values of VMFD can be different other than 4,5, say x,y, but these x,y will be the same throughout a single run. But will stay consistent in runtime; this is applicable in all examples below]

```
VMFD: 5
stopped
running -
exit reason:
2
VMFD: 5
KVM_EXIT_IO
VMFD: 5 out
port: 17,
data: 65534
VMFD: 4
stopped
running -
exit reason:
2
VMFD: 4
KVM_EXIT_IO
VMFD: 4 out
port: 16,
data: 2
.....
```

2a. One at a time

For your assignment, you must ensure the vCPU threads run on the same core alternatively in a controlled environment. To achieve this, do the following tasks:

Instead of creating two pthreads for vCPUs (and one for the main thread, totaling three. You can observe the same using `htop`), modify the **`kvm_run_vm`** function to not make any pthread calls, but instead use the main thread to alternate between the two VMs on a `KVM_EXIT_IO`. This ensures that you can run two VMs on one physical CPU.

```
$ ./matrix
VMFD: 4,
Loaded
Program with
size: 7
VMFD: 5,
Loaded
Program with
size: 7
VMFD: 4
started
running
```

The single-threaded execution model ensures that the VM runs one after another. The execution model depends on the VM to ensure it returns to the userspace program after

```
VMFD: 4 out
port: 16,
data: 0
VMFD: 5
started
running
VMFD: 5
stopped
running -
exit reason:
2
VMFD: 5
KVM_EXIT_IO
VMFD: 5 out
port: 17,
data: 0
VMFD: 4
started
running
VMFD: 4
stopped
running -
exit reason:
2
VMFD: 4
KVM_EXIT_IO
VMFD: 4 out
port: 16,
data: 1
VMFD: 5
started
running
VMFD: 5
stopped
running -
exit reason:
2
VMFD: 5
KVM_EXIT_IO
VMFD: 5 out
port: 17,
data: 65535
VMFD: 4
started
running
VMFD: 4
stopped
running -
exit reason:
2
VMFD: 4
KVM_EXIT_IO
VMFD: 4 out
port: 16,
```

VMFD values in the corresponding outputs.

```
exit reason:
2
VMFD: 5
KVM_EXIT_IO
VMFD: 5 out
port: 17,
data: 65534
.....
```

Update: If you have implemented part 2a in `matrix.c`, please make a copy of the implementation in a file named `matrix-a.c` and update the makefile to generate `matrix-a` executable accordingly.

Please make sure that `matrix-a.c` should only implement part 2a!

This checkpoint should ensure that you have completed 2a

2b. Vulnerabilities found… Please fix it.

Remove/Comment out the `out %ax, $0x10` and `out %ax, $0x11` in `guest1.s` and `guest2.s`. You can try to run your VMs now but will observe that only `vm1` (or whatever VMs you were running first) will be running. Since no IO operation is happening, no VM exits will happen. Therefore, the previous mechanism will not work.
A sample output is as follows.

```
$ ./matrix
VMFD: 4,
Loaded
Program with
size: 7
VMFD: 5,
Loaded
Program with
size: 7
VMFD: 4
started
running
```

The other VM will starve if the running VM doesn't return to the user space program, which is happening in the example.

therefore, a similar approach can also be used to schedule the VMs. The `timer_create()` subroutine of the time library creates a monotonic per-process interval timer that can be configured to issue a signal to the process after a configurable duration.
For this approach, you must create a timer, select a signal to be delivered, and set and reset the timer interval for the signal to be injected periodically.
Once this is set – the timer ready to fire — attempt to run your VMs. Only one of them should run. On timer expiry, a signal is delivered to the process, which will run the guest VM. On such a signal, the default behavior should result in a VM exit to be handed by the user space program.

This also requires the proper setup of signals within the KVM. You must use the KVM API `KVM_SET_SIGNAL_MASK` to do the same during VM execution. Set the mask to clear the signal that you want to be delivered. Additionally, make sure that the signal is blocked for the control thread.

For sanity check, add the following line after ioctl return from `KVM_RUN`:

```
printf("Time: %f\n",
CURRENT_TIME);
```

If everything is done correctly, the output will be something like that listed below —

```
$ ./matrix
VMFD: 4,
Loaded
Program with
size: 5
VMFD: 5,
Loaded
Program with
size: 5
VMFD: 4
started
```

If signals are injected into the VMs, the execution will be handled by the user space program, which can decide who to run next. In this example, the userspace

```
10
VMFD: 4
KVM_EXIT_INTR
VMFD: 5
started
running
Time:
2.002100
VMFD: 5
stopped
running -
exit reason:
10
VMFD: 5
KVM_EXIT_INTR
VMFD: 4
started
running
Time:
3.002045
VMFD: 4
stopped
running -
exit reason:
10
VMFD: 4
KVM_EXIT_INTR
VMFD: 5
started
running
Time:
4.002001
VMFD: 5
stopped
running -
exit reason:
10
VMFD: 5
KVM_EXIT_INTR
VMFD: 4
started
running
Time:
5.001990
VMFD: 4
stopped
running -
exit reason:
10
VMFD: 4
KVM_EXIT_INTR
VMFD: 5
started
running
```

```
VMFD: 5
KVM_EXIT_INTR
VMFD: 4
started
running
Time:
7.001829
VMFD: 4
stopped
running -
exit reason:
10
VMFD: 4
KVM_EXIT_INTR
VMFD: 5
started
running
Time:
8.001799
VMFD: 5
stopped
running -
exit reason:
10
VMFD: 5
KVM_EXIT_INTR
VMFD: 4
started
running
Time:
9.001752
VMFD: 4
stopped
running -
exit reason:
10
VMFD: 4
KVM_EXIT_INTR
VMFD: 5
started
running
.....
```

Hint:
- `KVM_RUN` returns -1 (`-EINTR`) and exit reason 10 (`KVM_EXIT_INTR`) when an unmasked signal is pending.
- Before re-running a VM on a signal, the timer to fire once again needs careful setup.

b executable accordingly.

Please make sure that `matrix-b.c` should only implement part 2b!

This checkpoint should ensure that you have completed 2b

2c. The final leap

Since you can now control the order of execution of the VMs, implement fractional scheduling for extending solution of 2b, which schedules `vm1` seven times and `vm2` three times out of ten time-quantum. Define the time quantum as a macro (`QUANTUM`) with a value of **1 second** (no matter what unit you are using, the value should equal 1s) and assume there will be no I/O operations in your VMs. Create two more macros, FRAC_A and `FRAC_B,` for scheduling fractions of `vm1` and `vm2` out of ten. By default, it should be seven and three.

References:
- https://man7.org/linux/man-pages/man2/timer_create.2.html
- https://docs.kernel.org/virt/kvm/api.html

Update: No need to make a separate file for this section. `matrix.c` should represent the final solution.

Submission instructions

- Your code should be well-commented and readable.
- You must submit a PDF file for Part 1a [named part1a.pdf] and final code solutions for Parts 1b and 2 according to the directory structure. Make sure that

as `<rollnumber>_assignment2` and the tarball as `<rollnumber>_assignment2.tar.gz` (e.g. `22d0999_assignment2.tar.gz`) - all small letters

Please strictly adhere to this format; otherwise, your submission will not count.

~~`22d0999_assignment2.gz`~~
~~`22d0999_assignment2.zip`~~
~~`22d0999_assignment2.tar.xz`~~

- *You can create the tarball using (copy-paste this command for a happy life)* **`tar -czvf <rollnumber>_assignment2.tar.gz <rollnumber>_assignment2`**
- Before submission, make sure the tarball is not corrupted by opening it.
- Please keep the .git folder in your submission that comes along with the starter code.
- Modify the README.md files to fit your submission.
- The tar should contain the following files in the following directory structure:

```
<rollnumber>_assignment2/
      ├──── .git/
      │         └── . .
. /* all git-related
files */
      │
      ├──── part1b/
      │             ├──
guest.c
      │             ├──
guest.ld
      │             ├──
guest16.s
      │             ├──
Makefile
```

```
      ├──── part2/
      │            ├──
guest1.s
      │            ├──
guest2.s
      │            ├──
Makefile
      │            ├──
matrix-a.c
      │            ├──
matrix-b.c
      │            ├──
matrix.c
      │            └──
README.md
      │
      ├──── part1a.pdf
      ├──── .gitignore
      ├──── Makefile
      └──── README.md
```

Deadline: ~~19th Feb 2024~~, 11:59 PM via Moodle. 21st Feb 2024 (extended)