Assignment #2

CSE 511 Fall 2022
Assigned: Thursday, September 29, 11:59 PM
Due: Thursday, October 20, 11:59 PM

Abstract

Assignment 2 extends the kernel and the boot loader that you have already built for Assignment 1. Please contact the instructor if you have not completed all questions for Assignment 1. Please also see Assignment 1's system installation and other instructions.

1 Introduction

1.1 Policy

You will work as individual to solve the problems for this assignment. Academic integrity is crucial. **Please read the syllabus regarding the academic integrity policy, which contains a list of things that are not permitted!** Note that academic integrity issues may arise when helping other students or looking for an answer online. All online sources have to be cited properly unless the piece of information comes directly from the official UEFI, Intel, or AMD manuals/specifications. OSDev is a good source for OS-related questions, but typically it is allowed only for certain corner cases which will be specifically mentioned by the instructor. For example, you cannot copy and paste code from OSDev or any other online source, or post any homework questions on forums. When in doubt, please ask the instructor or the TA first.

1.2 Required Changes

This assignment builds upon Assignment 1. (Please contact the instructor if you have not completed *all* questions for Assignment 1.)

You need to integrate your boot loader code `boot/boot.c` (replace this file) and the OS kernel code `kernel/kernel.c` (replace this file). Please do not replace `kernel/kernel_entry.S` or `Makefile` as they have been substantially modified in Assignment 2. Inside `kernel/kernel.c`, you need to remove the code which draws a picture since you will use a framebuffer console instead. (You still need to have the page table code though.)

Additional files are provided with the Assignment 2 code. Please explore new directories and files. Note that all kernel-related headers reside in `kernel/include`. Likewise, all user-program headers reside in `user/include`.

In this assignment, you need to additionally load a user program (its main source file is `user/user.c`). The executable file for the user program is located at `EFI\\BOOT\\USER`. Assignment 1's boot loader loads the kernel executable only. Please extend your boot loader to additionally read this user program into a dedicated *page-aligned* buffer (allocated in the boot loader), which you can pass as an argument to `kernel_start`. The size of this buffer should be a *multiple of a page size.*

Attention: Please also *page align* the kernel buffer in your boot loader implementation if you have not done that already. This is required to avoid any potential code misalignment issues. Since the compiled kernel binary will be larger than a single page, please make sure your boot loader can really load the *entire* kernel binary successfully.

For both the user program and the kernel buffers, you can simply call `AllocatePages()` in UEFI, which already returns page-aligned addresses. You can assume that both the user program and the kernel never exceed 128 KB.

In addition, you now *must* allocate at least one page for the kernel stack, and one page for the user program stack. Due to additional initialization code in `kernel/kernel_entry.S`, you *must* pass the kernel stack buffer pointer as the **very first argument** to `kernel_start`. Please modify your implementation accordingly. You can pass other parameters as you wish but please do not pass more than six arguments. If you need to pass more arguments, please allocate a separate structure that contains all additional arguments that you wish to pass and then pass a pointer to that structure as one argument.

1.3 Provided Code

Please get the code and create your own repository similar to the "Setup Assignment" by following this link: `https://classroom.github.com/a/aCb1GsGr`

`kernel/kernel_entry.S` is one of the files that was substantially changed since Assignment 1. Previously, `_start` would simply jump to `kernel_start`, where the actual kernel code is located. The new code also eventually jumps to `kernel_start`, but it first reloads GDT (Global Descriptor Table), initializes segment registers, and sets up a new kernel stack pointer using the first passed argument. It will also store this pointer in the `kernel_stack` variable. (Note that you still have to initialize the user space stack pointer, `user_stack`, on your own.)

This file also defines the GDT itself. If you need to add new GDT entries, you can modify this file. However, please avoid adding any other code that is unrelated to initialization, and also avoid moving the `_start` definition.

For the remaining assembly code, `kernel/kernel_asm.S` is provided. This file already includes a preliminary implementation of the system call entry point, `syscall_entry`. Additionally, it includes `user_jump`, a special assembly function which can be used for the *initial* kernel-to-user-space transition. System calls are *partially* implemented in `kernel/kernel_syscall.c`. `kernel/kernel_asm.S` also includes templates for an exception handler and an APIC timer interrupt handler that you will use as discussed below.

The user-space program code must be placed in `user/user.c`.

Finally, kernel-related headers can be found in `kernel/include`, and user-related headers can be found in `user/include`. For example, `kernel/include/kernel_syscall.h` provides declarations for `kernel/kernel_asm.S` and `kernel/kernel_syscall.c`, whereas `user/include/syscall.h` provides system call invocation functions that can be used from user-space applications.

To initialize system calls in the kernel, you have to call `syscall_init`. Note that `syscall_init` is incomplete and you must extend it prior to using system calls.

1.4 Compilation and Execution

Please compile the code by running "make" in your repository. This command will also create a bootable disk image ("boot.img"). To run the compiled code in qemu, you can simply run "make test", which will internally execute the following command:

```
qemu-system-x86_64 -bios /usr/share/qemu/OVMF.fd -m 1024
  -drive format=raw,file=<path_to_img_file>
```

Note that the path to the UEFI BIOS is provided for Ubuntu. Please locate OVMF.fd if your system is different. If this file is missing, you can download and unpack "bios.zip" from Canvas.

When building "boot.img", the boot loader is placed into

```
EFI\\BOOT\\BOOTX64.EFI,
```

the kernel is placed into

```
EFI\\BOOT\\KERNEL,
```

and the user program is placed into

```
EFI\\BOOT\\USER.
```

1.5 Submission

You will receive ***zero*** points if:

- you break any of the assignment rules specified in the document
- your code does not compile/build
- your code is buggy and crashes qemu or system
- the GIT history (in your github repo) is not present or simply shows one or multiple large commits; it must contain the *entire* history which shows the actual *incremental* work.

Please commit your changes relatively frequently so that we can see your work clearly. **Do not forget to push changes to the remote repository in github!** You can get as low as ***zero*** points if we do not see any confirmation of your work. **Do not attempt to falsify time stamps, this may be viewed as an academic integrity violation!**

Your submission can use the late policy, and we will calculate and use the better of the normal score and the late score with the late penalty.

Finally, include "report.pdf" in your github repository with **qemu screenshots** (several questions request separate screenshots, see below in the text) and a brief description of what was implemented for each of the questions below. Please be specific when describing what was (and was not) implemented but do not be too verbose.

In your Canvas submission, please indicate the GIT commit number corresponding to the final submission. You should also specify your github username. The submission format in plain text

```
<your_github_username>:GIT commit
```

For example, runikola:936c332e7eb7feb5cc751d5966a1f67e8089d331

1.6 Disassembler

If you need to disassemble the kernel binary, you need to specify corresponding options to objdump (since it is a "raw binary"):

```
objdump -D -Mx86-64 -b binary -m i386 kernel.x86_64
```

or this if you prefer Intel/Microsoft syntax:

```
objdump -D -Mintel,x86-64 -b binary -m i386 kernel.x86_64
```

1.7 Framebuffer Console

The provided code extends your kernel to support console output using a printf-like function (implemented in `kernel/printf.c`). For that purpose, it implements a simple framebuffer console driver (`kernel/fb.c`), which renders characters using a bitmap (raster) 8x16 ASCII font (`kernel/ascii_font.c`).

The provided `printf` function implements many (though not all) features of LibC's printf. This will help you with debugging since this function can format numbers, strings, pointers, etc. Unlike in UEFI, you will use ordinary UTF-8 strings (but only the ASCII subset is supported currently), i.e., use ordinary strings "..." rather than L"...". In addition, this version of printf for the kernel will help to implement system calls for the user-space application as described below.

In Assignment 1, you were asked to draw some figure as the very final step in your kernel. Please remove that code and replace it with the framebuffer console initialization:

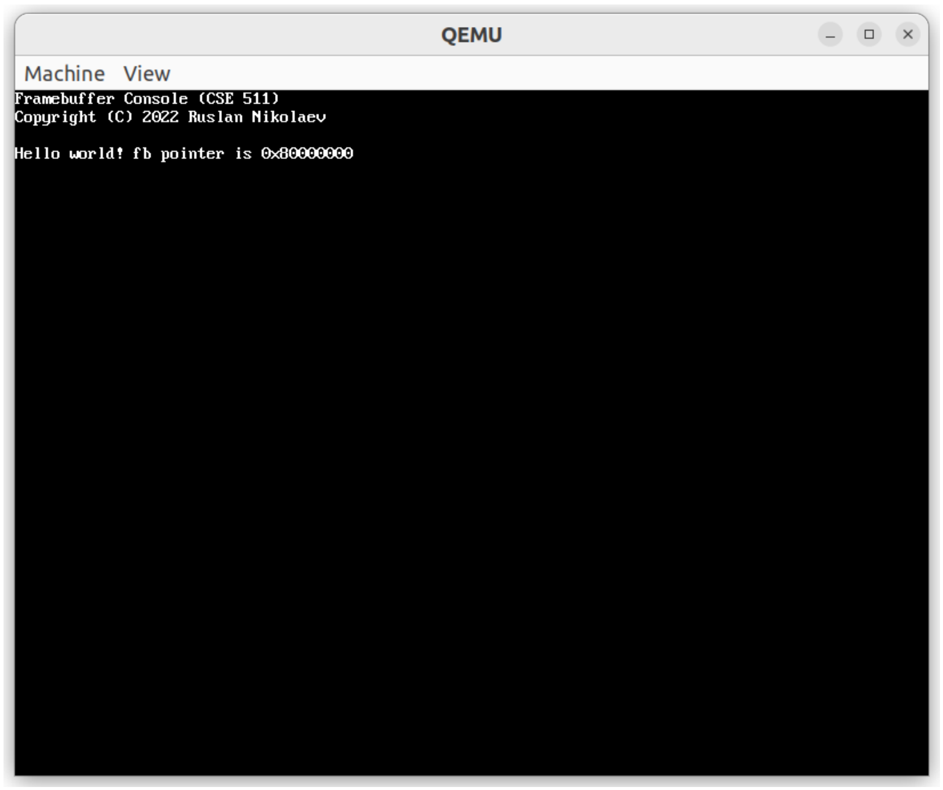
```
#include <fb.h>
...
void kernel_start(...)
{
...
fb_init(framebuffer, width, height);

/* Never exit! */
while (1) {};
}
```

After calling `fb_init`, you can use `printf` in your *kernel* as in the following example:

```
#include <printf.h>
...
printf("Hello world! fb pointer is %p\n", framebuffer);
```

You should expect the output similar to Figure 1. As you can notice, the framebuffer base is at 0x80000000 (i.e., it corresponds to 3 GB). This is why you were previously asked to map 4 GB rather than just 1 GB (the amount of RAM). For this assignment, your qemu instance should still be using no more than 1 GB of RAM.



```
Framebuffer Console (CSE 511)
Copyright (C) 2022 Ruslan Nikolaev

Hello world! fb pointer is 0x80000000
```

Figure 1: An example output.

2 Assignment Questions

2.1 Basics [45 pts]

Please integrate the frame buffer console into your kernel. You also need to load the user application into a memory buffer directly from the boot loader. Please allocate the kernel and user stacks and set them up accordingly. (See above.)

You must create a page table for the application so that you can also execute application code from user space. This page table should point to the kernel portion (i.e., the first entry of Level 4's page table can simply refer to your existing kernel's Level 3). However, this page table will also refer to the pages that are assigned to the user application itself (the page-aligned buffer that you allocated as well as the stack page). Note that the kernel portion should *only* be accessible in kernel space, whereas the application portion is accessible in both user and kernel space. Also note that `user_stack` must point to the user-space (rather than kernel-space) virtual address.

You can assign any virtual addresses from the bottom 1 GB for the application. As shown in Figure 2, that would correspond to the last entry of Level 4 and to the last entry of Level 3. Please make sure to set up correct permissions for these new mappings (i.e., that the user-space portion can be accessed in user/unprivileged mode). The kernel-space portion can only be accessed in kernel/privileged mode.

Finally, you need to jump to the user application code by switching to unprivileged mode. To do that, you need to load the corresponding page table and then use the provided code for the initial kernel-to-user-space transition.

Your application should not do anything yet, but to complete this part you have to complete the above-mentioned steps.

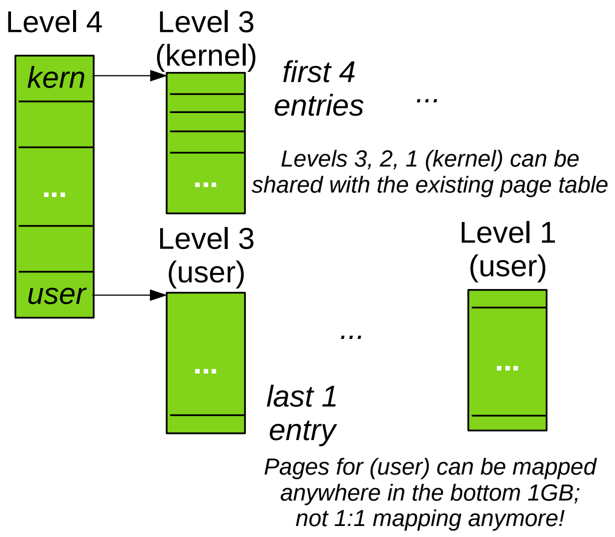

Level 4
Level 3 (kernel)
kern
first 4 entries
...
Levels 3, 2, 1 (kernel) can be shared with the existing page table
Level 3 (user)
Level 1 (user)
user
last 1 entry
Pages for (user) can be mapped anywhere in the bottom 1GB; not 1:1 mapping anymore!

Figure 2: A user-space page table

2.2 System calls [25 pts]

Please extend your application and kernel to support system calls, i.e., making user-to-kernel and kernel-to-user transitions. Your application should call the same system call at least *twice*. This system call will print a message using the kernel framebuffer console. You must pass a message string *from user space*. When calling the same system call multiple times, you should pass different strings, as shown in Figure 3. You must use the SYSCALL/SYSRET mechanism. (Already *partially* implemented in the provided code. Hint: please check what MSR_LSTAR is and initialize it appropriately.)

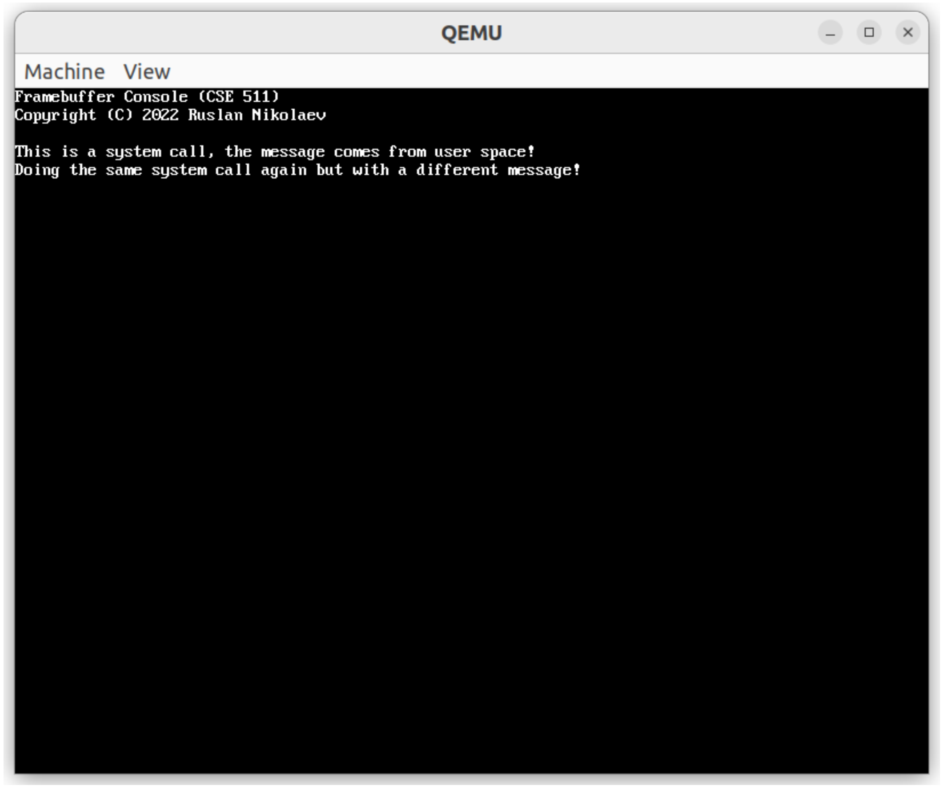
The message itself will be printed in the kernel using kernel's `printf`. For simplicity, you can assume that memory locations where message strings are located are always valid. (Though OS kernels do not typically trust any arguments passed by user applications in general.) You must still check that the system call number is valid and handle error cases appropriately.

2.3 Task State Segment (TSS) [15 pts]

Remember that for each CPU you need to define a separate TSS segment. We are using just one CPU, therefore we will need one TSS segment.

Please add a TSS descriptor in GDT (`kernel/kernel_entry.S`). (Remember that you actually need two entries for a TSS descriptor per Figure 4-22 of the AMD64 Manual, Volume 2.) These entries will be initialized in your C code, so just reserve zero-valued entries.

`interrupts.h` provides `load_tss_segment`. The purpose of this function is two-fold: (1). Initialize the reserved (zero-valued) TSS descriptor entries in GDT; (2). Load the TSS descriptor



```
Framebuffer Console (CSE 511)
Copyright (C) 2022 Ruslan Nikolaev

This is a system call, the message comes from user space!
Doing the same system call again but with a different message!
```

Figure 3: An example system call output (must be called from user space).

from GDT to the task register (the *ltr* instruction). The code already sets all flags, you need to specify the GDT selector for TSS (the byte offset in GDT) and the address of the TSS segment. (Remember that kernel-virtual and physical addresses are identical in our case.)

The TSS segment must be allocated and initialized appropriately prior to `load_tss_segment`. You must page-align the TSS segment (allocating a separate page is recommended)! If the TSS segment is not page aligned and spans two different pages, your CPU may load it incorrectly.

Remember that the only value that we need to worry about in the TSS segment is the stack pointer for ring 0, which corresponds to rsp[0]. TSS additionally supports IST stack values, but we will not use them. Also `iobp_base` should be initialized to the size (**not limit!**) of the TSS segment to avoid using the I/O bitmap (`iobp_base` specifies an offset which exceeds the TSS segment limit.) To initialize unused values, you can use `__builtin_memset()` provided by GCC.

Please allocate a separate page for TSS's rsp[0]. This stack value will be automatically loaded when an interrupt or exception occurs, which is critical since we cannot rely on user-space stacks.

For this task, do not consult the rumprun code as rumprun does not properly load TSS.

Please provide a screenshot in your report.

2.4 Interrupt Descriptor Table (IDT) and Exceptions [25 pts]

Please define and load a 256-entry IDT table to handle all 256 interrupts and exceptions. The IDT table should be 16-byte aligned. Please see `interrupts.h` for comments and helper functions. You will need to define an 80-bit IDT table "pointer" (similar to the GDT table pointer), then you need to use this “pointer” in `load_idt` (see the *lidt* instruction description).

Please initialize the IDT table prior to loading it. The first 32 entries correspond to CPU exceptions and must be initialized to the default CPU exception handler. The corresponding IDT entries should be initialized appropriately (you can check the rumprun code) to the default handler that you need to add to `kernel/kernel_asm.S`. This file already includes an unfinished template of this handler. Your handler must print a message that an exception occurred and halt the system.

Please note that the compiler has a bug/issue when compiling a “raw binary” **with** position-independent code. It manifests when you directly calculate addresses of functions (i.e., when you need a function pointer rather than just call a function). To work around this problem, we provide the `syscall_entry_ptr` variable, which is set in `kernel/kernel_entry.S`. You need to create a similar variable for the exception handler function (define this variable in the C file where you access it). Please initialize it also in `kernel_entry.S` similar to the existent variable:

```
leaq syscall_entry(%rip), %rax /* syscall_entry_ptr -> syscall_entry() */
movq %rax, syscall_entry_ptr(%rip)
```

In the handler message, **you must print** the `%rsp` value. (Please also print `kernel_stack` and `user_stack` in the beginning of your kernel code after they are initialized.) When an exception happens, you have to confirm that `%rsp` truly resides in a different page, i.e., the corresponding *page* not equals to that of `user_stack-1` (`user_stack` itself actually points to the next page).

In your user program, please deliberately access a non-existent address, e.g.,

```
*((char *)0xFFFFFFFFFFFFFFFFULL) = 0;
```

That assumes that the last page does not have any mapping. Alternatively, you can use any other address that is not mapped in user space within the bottom 1 GB. That will trigger an exception which will print a message and terminate the system.

Please provide a screenshot in your report.

2.5 Lazy Allocation [15 pts]

Note that the actual exception that happens in the previous question is the page-fault exception (number 14). Please provide a customized exception handler for page-fault exceptions.

Please preallocate a page that you will map in the exception handler (do it initially in the boot loader similar to `user_stack`). In the previous question, you were asked to access a non-existent page. For convenience, the address can be anywhere in the bottom 1 GB, not necessarily in the last page, assuming the page was not initially mapped.

Your exception handler should call a function, which will modify the user page table and reload `%cr3`. For simplicity, you can just prepare global variables with the page that you need to map as well as the entry that you need to modify. After modifying the page table, the address that you were trying to access in the previous question should point to the existent page. When an exception handler completes, the same instruction will be tried again and should now succeed. Please make a system call with a message after that to confirm that the program is still alive.

2.6 APIC Timer [25 pts]

Implement an APIC-based periodic timer. First, you need to initialize the local APIC controller by including `apic.h` and calling `x86_lapic_enable` from your kernel. This will automatically detect whether your system supports x2APIC or legacy (x)APIC. The corresponding function will also initialize certain APIC registers. Specifically, it will initialize the spurious-interrupt vector register, for which vector 0xFF is chosen. Make sure you do not use this vector for anything else.

You need to initialize the APIC timer such that it will work in *periodic mode*. The corresponding interrupt handler simply needs to print a message **and** acknowledge the interrupt to APIC. To that end, a template in `kernel_asm.S` is provided. You can simply call your C function from that handler. To read or write APIC registers, you can use `x86_lapic_read` and `x86_lapic_write` functions (from `apic.c`) correspondingly. They will transparently handle xAPIC and x2APIC modes. You need to specify offsets as they are defined by the x2APIC (not xAPIC!) specification. *Remember that to reference an interrupt handler function when initializing IDT, you need to use a workaround similar to syscall_entry_ptr.*

You need to program the APIC timer appropriately (see the provided references and our lectures). As part of initialization, you need to enable the APIC timer and specify an interrupt vector associated with the APIC timer handler. You can choose any vector (except 0..31 and 255 which are already used for other purposes). As part of initialization, you need to specify the timer counter and configure the divide configuration register. Although, no calibration is required for this assignment, you need to choose parameters such that they work well on your system (i.e., triggering an interrupt roughly every 0.5-2 seconds, not too fast and not too slow). As a starting point, you can try 4194304 as the counter and 128 as the divisor.

You should expect an output similar to Figure 4. **Please provide a screenshot in your report.**

3 References

Intel 64 documentation `https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html`

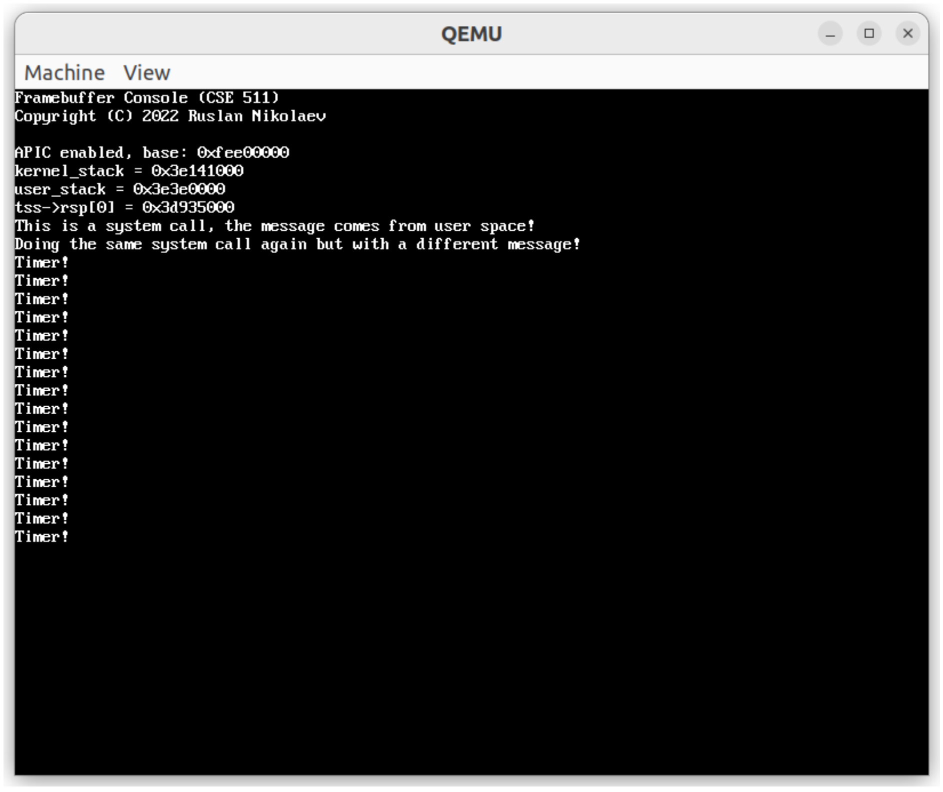
AMD64 documentation `https://www.amd.com/system/files/TechDocs/40332.pdf`

OSDev Paging and System Calls Examples `https://wiki.osdev.org/Paging`, `https://wiki.osdev.org/Page_Tables`, `https://wiki.osdev.org/System_Calls`, and others. Note that we use 64-bit (long) mode.

APIC (general), Chapter 10, Volume 3A `https://www.intel.com/content/www/us/en/architecture-an`
`64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html`

x2APIC changes `https://courses.cs.washington.edu/courses/cse451/21wi/readings/x2apic.pdf`

OSDev APIC Timer (an extra reference just in case, prefer the Intel manual) `https://wiki.osdev.org/APIC_timer`



```
Framebuffer Console (CSE 511)
Copyright (C) 2022 Ruslan Nikolaev

APIC enabled, base: 0xfee00000
kernel_stack = 0x3e141000
user_stack = 0x3e3e0000
tss->rsp[0] = 0x3d935000
This is a system call, the message comes from user space!
Doing the same system call again but with a different message!
Timer!
Timer!
Timer!
Timer!
Timer!
Timer!
Timer!
Timer!
Timer!
Timer!
Timer!
Timer!
Timer!
Timer!
Timer!
Timer!
```

Figure 4: An example APIC timer output.