CS202: Lab 4: WeensyOS

Introduction

In this lab, you will implement process memory isolation, virtual memory, and a system call (`fork()`) in a tiny (but real!) operating system, called WeensyOS.

This will introduce you to virtual memory and reinforce some of the concepts that we have covered this semester.

The WeensyOS kernel runs on x86-64 CPUs. Because the OS kernel runs on the “bare” hardware, debugging ... figuration of the hardware, the usual re ... plications running on top of it). And beca ... em services (for example, causing the ... ucing what led to a kernel crash can be ... l way to develop code for an OS (wheth ... stry) was to boot it on a physical CPU. Th ... etter since. You will run WeensyOS in QE...

QEMU is a software-b ... st like a physical x86-64 CPU, but if you ... al) hardware, QEMU itself and the w ... hat is, the Linux OS you booted and th ... is in quotation marks because your L ... dware). So, for example, your last few ... still get logged to disk (by QEMU runnin ... veloping amounts to re-running the QEMU emulator application.

Heads up. As always, it’s important to start *on time*. In this case, *on time* means 2-3 weeks before the assignment is due, as you will almost certainly need all of the allotted time to complete the lab. Kernel development is less forgiving than developing user-level applications; tiny deviations in the configuration of hardware (such as the MMU) by the OS tend to bring the whole (emulated) machine to a halt.

To save yourself headaches later, read this lab writeup in its entirety before you begin.

Resources.

- You may want to look at Chapter 9 of CS:APP3e (from which our x86-64 virtual memory handout is borrowed). The book is on reserve at the Courant library. Section 9.7 in particular describes the 64-bit virtual memory architecture of the x86-64 CPU. Figure 9.23 and Section 9.7.1 show and discuss the `PTE_P`, `PTE_W`, and `PTE_U` bits; these are flags in the x86-64 hardware's page table entries that play a central role in this lab.

- You may find yourself during the lab wanting to understand particular assembly instructions. Here are two guides to x86-64 instructions, from Brown and CMU. The former is more digestible; the latter is more comprehensive. The supplied code also uses certain assembly instructions like `iret`; see here for a reference.

Getting Started

Obtain the lab files as follows. We assume that you have run the commands in the "Getting Started" section of lab3. To check issue the following command:

```
$ git remote -v
origin  git@github.c                                        git (fetch)
origin  git@github.c                                        git (push)
upstream    https://
upstream    https://
```

The `upstream` should   ends in `labs.git`, then follow the instruc

Once `$ git remote`   g:

```
$ cd ~/cs202
$ git fetch upstream
$ git merge upstream
```

This lab's files are loc

If you have any "confli   rther. Run `git push` to save your wor

Another heads up. Given the complexity of this lab, and the possibility of breaking the functionality of the kernel if you code in some errors, make sure to commit and push your code often! It's very important that your commits have working versions of the code, so if something goes wrong, you can always go back to a previous commit and get back a working copy! At the very least, for this lab, you should be committing once per step (and probably more often), so you can go back to the last step if necessary.

Goal

You will implement complete and correct memory isolation for WeensyOS processes. Then you'll implement full virtual memory, which will improve utilization. You'll implement `fork()` (creating new processes at runtime) and for extra credit, you'll implement `exit()`

(destroying processes at runtime).

We've provided you with a lot of support code for this assignment; the code you will need to write is in fact limited in extent. Our complete solution (for all 5 stages) consists of well under 300 lines of code beyond what we initially hand out to you. All the code you write will go in `kernel.c` (except for part of step 6).

Testing, checking, and validation

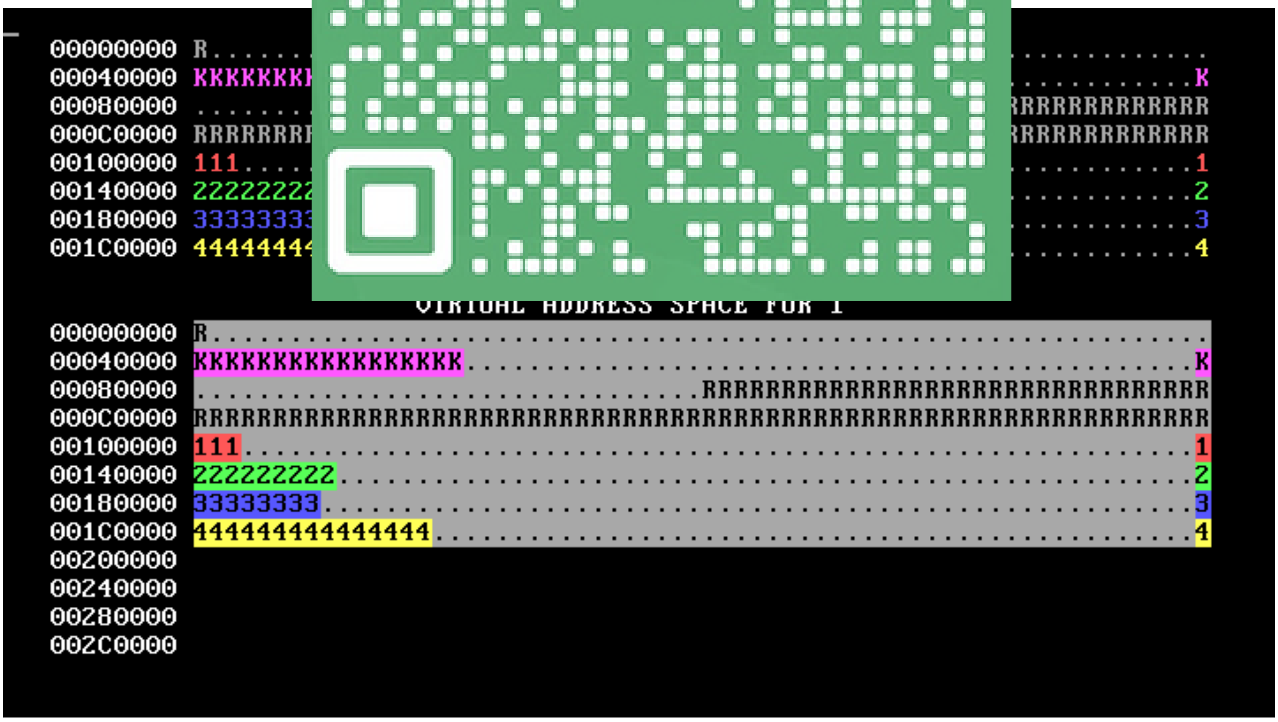
For this assignment, your primary checking method will be to run your instance of Weensy OS and visually compare it to the images you see below in the assignment.

Studying these *graphical memory maps* carefully is the best way to determine whether your WeensyOS code for each stage is working correctly. Therefore, you will definitely want to **make sure you understand how to read these maps before you start to code**.

We supply some grad… e will not be your principal source of fee… ner a given step is passing or failing; look…

Initial state

Run `make run` in your … below, which shows four processes … program in `p-allocator`:



This image loops forever; in an actual run, the bars will move to the right and stay there. Don't worry if your image has different numbers of K's or otherwise has different details.

If your bars run painfully slowly, edit the `p-allocator.c` file and reduce the `ALLOC_SLOWDOWN` constant.

Stop now to read and understand `p-allocator.c`.

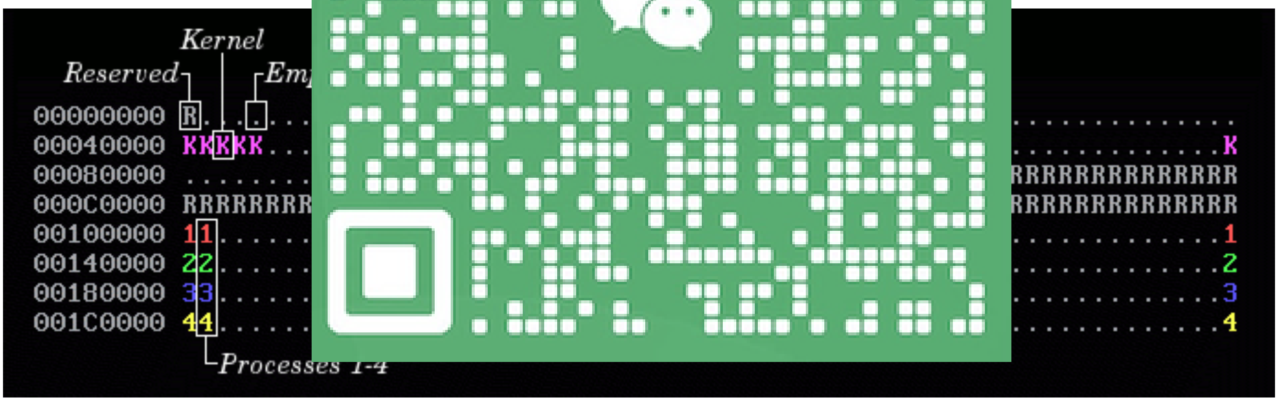
Here's how to interpret the memory map display:

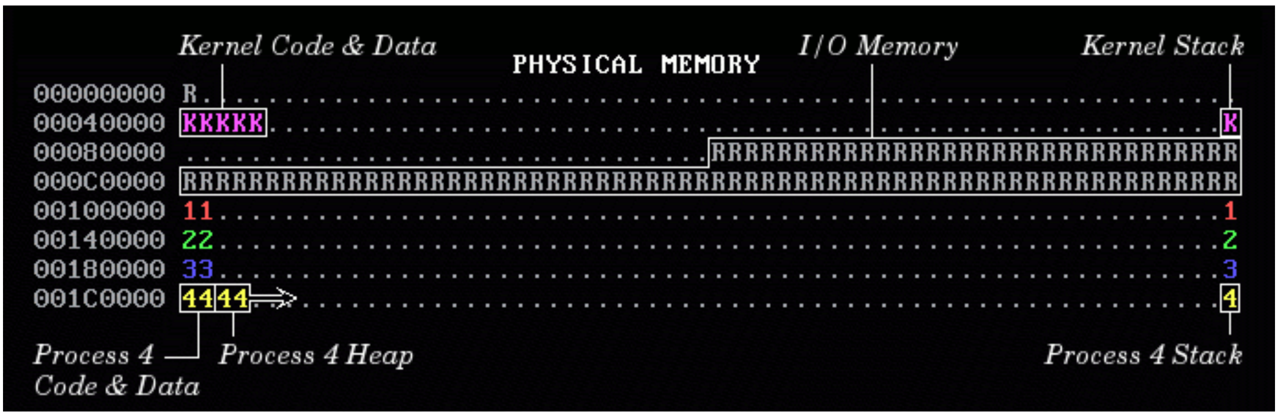
- WeensyOS displays the current state of physical and virtual memory. Each character represents 4 KB of memory: a single page. There are 2 MB of physical memory in total. (Ask yourself: how many pages is this?)

- WeensyOS runs four processes, 1 through 4. Each process is compiled from the same source code (`p-allocator.c`), but linked to use a different region of memory.

- Each process asks the kernel for more heap memory, one page at a time, until it runs out of room. As usual, each process's heap begins just above its code and global data, and ends just below its stack. The processes allocate heap memory at different rates: compared to Process 1, Process 2 allocates twice as quickly, Process 3 goes [illegible] faster. (A random number generat[illegible]ching rows of numbers show h[illegible], and 4 are allocated.

Here are two labeled [illegible]ean and how memory is arranged.





The virtual memory display is similar.

- The virtual memory display cycles successively among the four processes’ address spaces. In the base version of the WeensyOS code we give you to start from, all four processes’ address spaces are the same (your job will be to change that!).
- Blank spaces in the virtual memory display correspond to unmapped addresses. If a process (or the kernel) tries to access such an address, the processor will page fault.
- The character shown at address X in the virtual memory display identifies the owner of the corresponding physical page.
- In the virtual memory display, a character is **reverse video** if an application process is allowed to access the corresponding address. Initially, *any* process can modify *all* of physical memory, including the kernel. Memory is not properly isolated.

Running WeensyOS

Read the `README.md` fi… …EMU’s default display causes access… …nsole`. To make `run-console` the defa…

There are several way… …og_printf` statements to your co… …log.txt` *outside* QEMU, into your `lab4`… …use assertions (of which we saw a few ir… …the helper functions we’ve provid… `check_page_table_ow`…

Memory system la…

The WeensyOS mem…

Constant	
`KERNEL_START_ADDR`	Start of kernel code.
`KERNEL_STACK_TOP`	Top of kernel stack. The kernel stack is one page long.
`console`	Address of CGA console memory.
`PROC_START_ADDR`	Start of application code. Applications should not be able to access memory below this address, except for the single page at `console`.
`MEMSIZE_PHYSICAL`	Size of physical memory in bytes. WeensyOS does not support physical addresses ≥ this value. Defined as `0x200000` (2MB).
`MEMSIZE_VIRTUAL`	Size of virtual memory. WeensyOS does not support virtual addresses ≥ this value. Defined as `0x300000` (3MB).

Writing expressions for addresses

WeensyOS uses several C macros to handle addresses. They are defined at the top of `x86-64.h`. The most important include:

Macro	Meaning
`PAGESIZE`	Size of a memory page. Equals 4096 (or, equivalently, `1 << 12`).
`PAGENUMBER(addr)`	Page number for the page containing `addr`. Expands to an expression analogous to `addr / PAGESIZE`.
`PAGEADDRESS(pn)`	The initial address (zeroth byte) in page number `pn`. Expands to an expression analogous to `pn * PAGESIZE`.
`PAGEINDEX(addr, level)`	The index in the `level`th page table for `addr`. `level` must be between 0 and 3; 0 returns the level-1 page table index (address bits 39–47), 1 returns the level-2 index (bits 30–38), 2 returns the level-3 index (bits 21–29), and 3 returns
`PTE_ADDR(pe)`	able entry `pe`. ing the low-order

Before you begin codi … s represent and be able to derive valu … .

Kernel and proces

The version of Weens … rnel and all processes in a single, … fined by the `kernel_pagetable` pa … *entity mapping*: virtual address X map

As you work through t … independent address spaces, where each process can access only a subset of physical memory.

The kernel, though, must remain able to access *any* location in physical memory. Therefore, all kernel functions run using the `kernel_pagetable` page table. Thus, in kernel functions, each virtual address maps to the physical address with the same number. The `exception()` function explicitly installs `kernel_pagetable` when it begins.

WeensyOS system calls are more expensive than they need to be, since every system call switches address spaces twice (once to `kernel_pagetable` and once back to the process's page table). Real-world operating systems avoid this overhead. To do so, real-world kernels access memory using *process* page tables, rather than a kernel-specific `kernel_pagetable`. This makes a kernel's code more complicated, since kernels can't always access all of physical memory directly under that design.