UCL CS 0019 Brad Karp

Individual Coursework 4: Implementing Virtual Memory in WeensyOS

Due date: 5:05 PM, 8th March 2019

Value: 6% of marks for module

Introduction

In this coursework you will implement process memory isolation, virtual memory, and a system call in a tiny (but very real!) operating system called WeensyOS. The WeensyOS kernel runs on x86-64 CPUs. Because the OS kernel runs on the "bare" hardware, debugging kernel code can be tough: if a bug causes misconfiguration of the hardware, the usual result is a crash of the entire kernel (and all the applications running on top of it). And because the kernel itself provides the most basic system services (e.g., causing the display hardware to display error messages, or causing the disk hardware to log them to disk) deducing what went wrong after a kernel crash can be particularly challenging. In the dark ages[1], the usual way to develop code for an OS (whether as part of a class, in a research lab, or in industry) was to boot it on a physical CPU. The lives of kernel developers have gotten much better since. You will run WeensyOS in QEMU, which is a software-based x86-64 emulator: it "looks" to WeensyOS just like a physical x86-64 CPU, but if your WeensyOS code-in-progress wedges the (virtual) hardware, QEMU itself and the whole OS the *real* hardware is running (i.e., the Linux OS you booted and that QEMU is running on) survive unscathed. So, for example, your last few debugging `printf()`s before a kernel crash will still get logged to disk (by QEMU running on Linux), and "rebooting" the kernel you're developing amounts to re-running the QEMU emulator application.

This coursework must be done on a Linux host. The 0019 virtual machine (VM) provided for use in this class comes with QEMU installed, and is the environment you should use for working on CW4. If you happen to have another Linux or UNIX machine at your disposal, you may be able to install QEMU on it and run WeensyOS there, but the only development environment the 0019 staff can support is the 0019 VM. The UCL CS lab machines and CSRW remote desktop do not (at this writing) have QEMU installed, and the 0019 teaching staff unfortunately aren't able to update the installed software on those machines.

Chapter 9 of CS:APP/3e covers virtual memory, and offers vital background for this coursework. 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, flags in the x86-64 hardware's page table entries that play a central role in this coursework.

Tasks

- Implement complete and correct memory isolation for WeensyOS processes.
- Implement full virtual memory, which will improve utilization.
- Implement the `fork()` system call, used to create new processes at runtime.

[1] i.e., when your instructor was an undergraduate, in a prior century

You will complete the above tasks in five stages described in detail below. Each stage has a test in this coursework's test suite, and is worth an equal share of the total marks for the coursework (i.e., 20% per stage).

We've provided you a lot of support code for this assignment, but the code you will need to write is in fact quite limited in extent. Our complete solution (for all 5 stages) consists of well under 200 lines of code beyond what we initially hand out to you. All the code you write must go in `kernel.c` and `kernel.h`.

This handout provides vital detail on how to interpret the graphical maps of WeensyOS's physical and virtual memory that QEMU will display to you while WeensyOS is running. Studying these graphical memory maps carefully is the best way to determine whether your WeensyOS code for each stage is working correctly. While we do provide tests that you can run to learn what grade your current code would receive, these tests only tell you whether each test has been passed or failed, because the graphical memory maps are already exhaustive in showing how your code behaves. In short, you will definitely want to make sure you understand how to read these maps before you start to hack.

You will also find considerable guidance in how to go about implementing the functionality for each stage of the coursework in this document. *Read this handout in its entirety carefully before you begin!*

As ever, it is important to get started early. Kernel development is less forgiving than developing user-level applications; tiny deviations in the configuration of hardware (e.g., the MMU) by the OS tend to bring the whole (emulated, in this case!) machine to a halt. You will almost certainly need the two weeks allotted to complete CW4.

Getting Started

We will use GitHub for CW4 in much the same manner as for CW2 and CW3. To obtain a copy of the initial code for CW4, please visit the following URL:

```
https://classroom.github.com/a/UIdJnL0b
```

If you'd like a refresher on using git with your own local repository and syncing it to GitHub, please refer to the CW2 handout.

Each time you push updated code to your GitHub repository for CW4, our automatic grading server will pull a copy of your code, run our automated tests on your code, and place a grade report in a file `report.md` in your GitHub repository. Your mark on CW4 will be that produced by the automated tests run by our automatic grading server on the latest commit (update) you make to your GitHub repository before the CW4 deadline. More on these tests below.

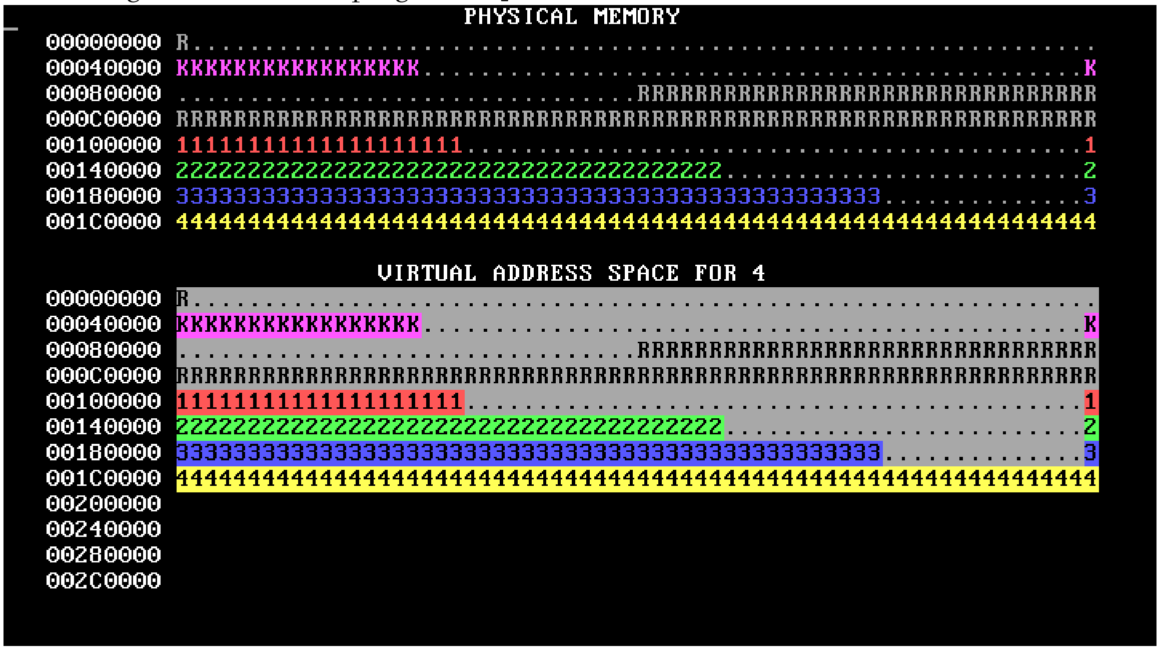
Please note that your code's behavior on the automated tests when run on the 0019 grading server will determine your mark on CW4.[2] If you choose to develop in a different environment, you must ensure that your code passes the automated tests on our automatic grading server. (We push grade reports to your GitHub repo each time you update your code to make it easy for you to check this.) The CS 0019 staff cannot "support" development environments other than the 0019 Linux VM; we cannot diagnose problems you encounter should your code pass the tests in some other environment, but fail them in the 0019 Linux VM.

[2]The *only* exception will be if the instructors determine that a student's submission produces output that matches the expected output without implementing the required functionality; such submissions will receive zero marks.

Getting Familiar with WeensyOS Memory Maps

Once you've cloned your repository from GitHub to your working environment, you can build the initial version of WeensyOS we've given you by issuing the shell command `make run` in your CW4 directory.

You should see something like the below, which shows four processes running in parallel, each running a version of the program in `p-allocator`:



```
                                 PHYSICAL MEMORY
00000000 R...............................................................
00040000 KKKKKKKKKKKKKKKKK..............................................K
00080000 ................................RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
000C0000 RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
00100000 11111111111111111111...........................................1
00140000 22222222222222222222222222222222222222.........................2
00180000 3333333333333333333333333333333333333333333333333..............3
001C0000 4444444444444444444444444444444444444444444444444444444444444444

                           VIRTUAL ADDRESS SPACE FOR 4
00000000 R...............................................................
00040000 KKKKKKKKKKKKKKKKK..............................................K
00080000 ................................RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
000C0000 RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
00100000 11111111111111111111...........................................1
00140000 22222222222222222222222222222222222222.........................2
00180000 3333333333333333333333333333333333333333333333333..............3
001C0000 4444444444444444444444444444444444444444444444444444444444444444
00200000
00240000
00280000
002C0000
```

The image above is in color, and is a single still frame from an animation. You will almost certainly want to view these memory maps in color with the animation. You can find these color animations on the 0019 class web site (along with a duplicate copy of the text in this handout that goes along with the images) at:

`http://www.cs.ucl.ac.uk/staff/B.Karp/0019/s2019/cw/cw4-maps/`

The animated version of the above 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` source 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:

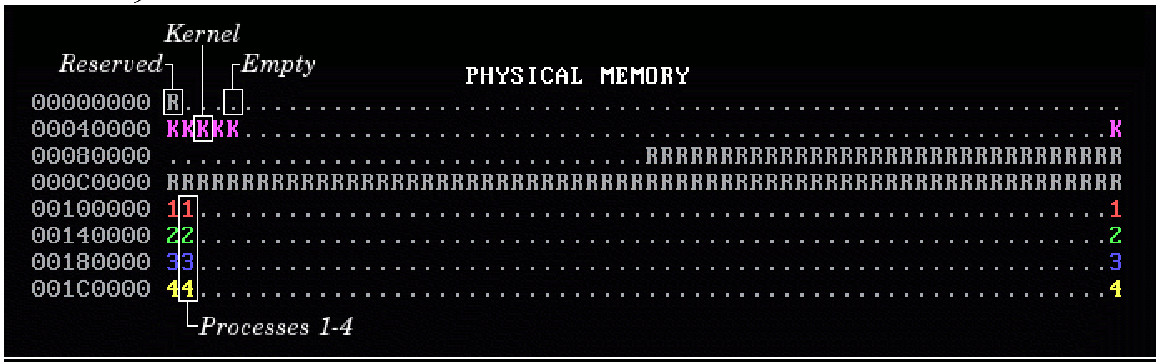
- WeensyOS displays the current state of physical and virtual memory. Each character represents 4 KB of memory: a single page. There are 2 MB of physical memory in total. (Ask yourself: how many pages is this?)

- 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.

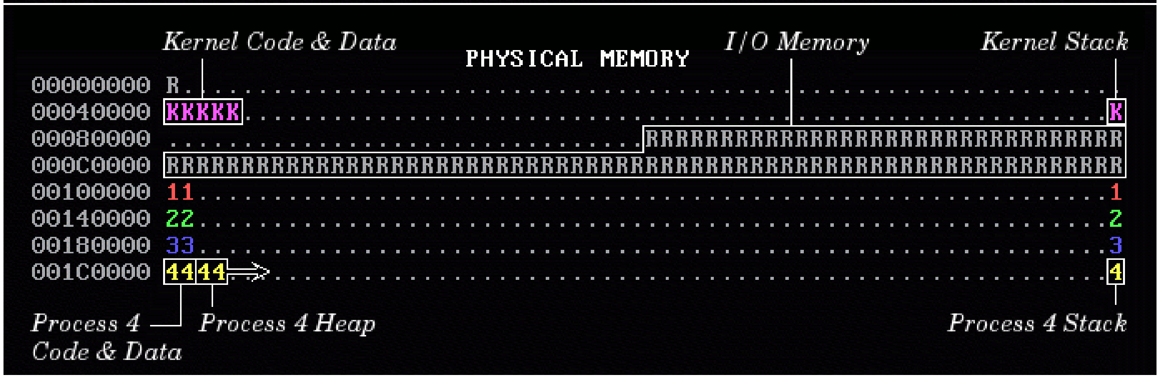
- Each process asks the kernel for more heap memory, one page at a time, until it runs out of room. As usual, each process's heap begins just above its code and global data, and ends just below its stack. The processes allocate heap memory at different rates: compared to Process 1, Process 2 allocates twice as quickly, Process 3 goes three times faster, and Process 4 goes four times faster. (A random number generator is used, so the exact rates may vary.)

The marching rows of numbers (in the animated version of this map on the 0019 web site) show how quickly the heap spaces for processes 1, 2, 3, and 4 are allocated.

Here are two labeled memory maps showing what the characters mean and how memory is arranged. These are best read in color; if you're reading the black-and-white hardcopy handout, visit the 0019 web site at the link above for the color versions!



```
                      PHYSICAL MEMORY
00000000 R...............................................................
00040000 KKKKK..........................................................K
00080000 ................................RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
000C0000 RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
00100000 11.............................................................1
00140000 22.............................................................2
00180000 33.............................................................3
001C0000 44.............................................................4
```



```
                      PHYSICAL MEMORY
00000000 R...............................................................
00040000 KKKKK..........................................................K
00080000 ................................RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
000C0000 RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
00100000 11.............................................................1
00140000 22.............................................................2
00180000 33.............................................................3
001C0000 4444...........................................................4
```

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 generate a page fault hardware exception.
- 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 in 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 `README-OS.md` in your CW4 repository for information on how to run WeensyOS. If QEMU's default display causes accessibility problems, you will want to run `make run-console`. To make `run-console` the default, run `export QEMUCONSOLE=1` in your shell.

There are several ways to debug WeensyOS. We recommend adding `log_printf()` statements to your code. The output of `log_printf()` is written to the file `log.txt` *outside* QEMU, into your CW4 working directory. We also recommend that you use assertions (of which we saw a few in the lecture on Undefined Behavior; try `man assert` at the Linux shell to learn more) to catch problems early. For example, call the helper functions we've provided, `check_page_table_mappings()` and `check_page_table_ownership()`, to test a page table for obvious errors.

Memory System Layout

The WeensyOS memory system layout is defined by several constants:

Constant	Meaning
`KERNEL_START_ADDR`	Start of kernel code
`KERNEL_STACK_TOP`	Top of kernel stack; 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` (2 MB).
`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 construct addresses. They are defined at the top of `x86-64.h`. The most important include:

Macro	Meaning
`PAGESIZE`	Size of a memory page; defined as 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 the level-4 index (bits 12–20).
`PTE_ADDR(pe)`	The physical address contained in page table entry `pe`. Obtained by masking off the flag bits (setting the low-order 12 bits to zero).

Before you begin coding, you should both understand what these macros represent and be able to derive values for them if you were given a different page size.

Kernel and Process Address Spaces

The version of WeensyOS you receive at the start of CW4 places the kernel and all processes in a single, shared address space. This address space is defined by the `kernel_pagetable` page table. `kernel_pagetable` is initialized to the *identity mapping*: virtual address X maps to physical address X.

As you work through CW4's stages, you will shift processes to using their own 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`. That makes a kernel's code more complicated, though, since kernels can't always access all of physical memory directly under that design.

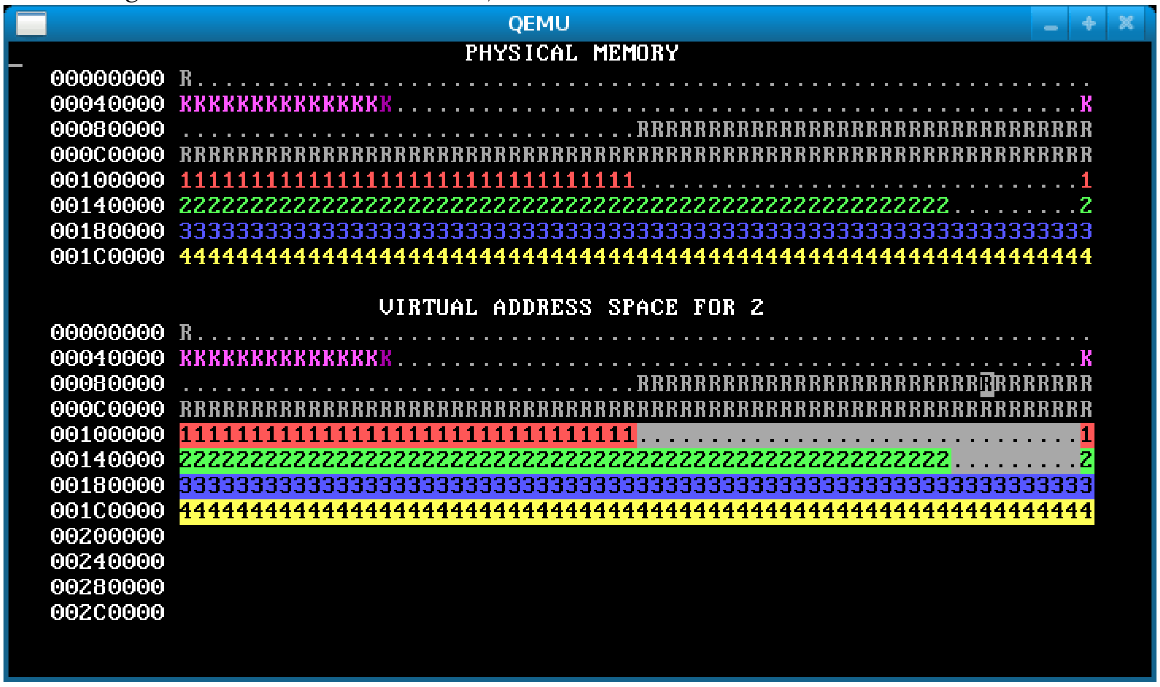
The Five Stages of WeensyOS

We describe below the five implementation stages you must complete in CW4: what you need to implement in each, and hints on how to do so.

Stage 1: Kernel Isolation

In the starting code we've given you, WeensyOS processes could stomp all over the kernel's memory if they wanted to. Better prevent that. Change `kernel()`, the kernel initialization function, so that kernel memory is inaccessible to applications, except for the memory holding the CGA console (the single page at `(uintptr_t) console == 0xB8000`.)[3]

When you are done, WeensyOS should look like the below. In the virtual map, kernel memory is no longer reverse-video, since the user can't access it. Note the lonely CGA console memory block. (As with all these maps, you will want to view the figure below in its online, color version at the URL given earlier in this handout.)



```
                               PHYSICAL MEMORY
00000000 R...............................................................
00040000 KKKKKKKKKKKKKKK................................................K
00080000 ................................RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
000C0000 RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
00100000 11111111111111111111111111111111...............................1
00140000 222222222222222222222222222222222222222222222222222222.........2
00180000 3333333333333333333333333333333333333333333333333333333333333333
001C0000 4444444444444444444444444444444444444444444444444444444444444444

                          VIRTUAL ADDRESS SPACE FOR 2
00000000 R...............................................................
00040000 KKKKKKKKKKKKKKK................................................K
00080000 ................................RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
000C0000 RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
00100000 11111111111111111111111111111111...............................1
00140000 222222222222222222222222222222222222222222222222222222.........2
00180000 3333333333333333333333333333333333333333333333333333333333333333
001C0000 4444444444444444444444444444444444444444444444444444444444444444
00200000
00240000
00280000
002C0000
```

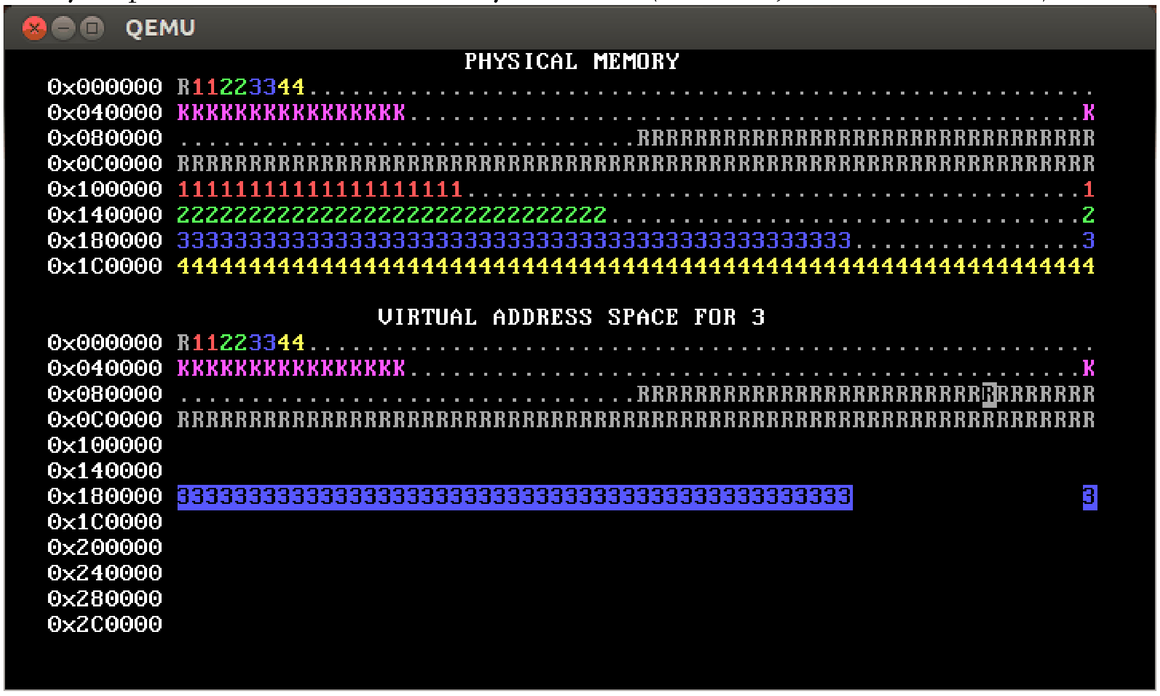
[3] Making the console accessible in this way, by making the range of RAM where the contents of the display are held directly accessible to applications, is a throwback to the days of DOS, whose applications typically generated console output in precisely this way. DOS couldn't run more than one application at once, so there wasn't any risk of multiple concurrent applications clobbering one another's display writes to the same screen locations. We borrow this primitive console design to keep WeensyOS simple and compact.

Hints:

- Use `virtual_memory_map()`. A description of this function is in `kernel.h`. You will benefit from reading all the function descriptions in `kernel.h`. You can supply `NULL` for the `allocator` argument for now.
- If you really want to look at the code for `virtual_memory_map()`, it is in `k-hardware.c`, along with many other hardware-related functions.
- The `perm` argument to `virtual_memory_map()` is a bitwise-or of zero or more `PTE` flags: `PTE_P`, `PTE_W`, and `PTE_U`. `PTE_P` marks Present pages (pages that are mapped). `PTE_W` marks Writable pages. `PTE_U` marks User-accessible pages—pages accessible by applications. You want kernel memory to be mapped with permissions `PTE_P | PTE_W`, which will prevent applications from reading or writing the memory, while allowing the kernel to both read and write.
- Make sure that your `sys_page_alloc()` system call preserves kernel isolation: Applications shouldn't be able to use `sys_page_alloc()` to screw up the kernel.

Stage 2: Isolated Address Spaces for Processes

Implement process isolation by giving each process its own independent page table. Your OS memory map should look like this when you're done (animated, color version online):



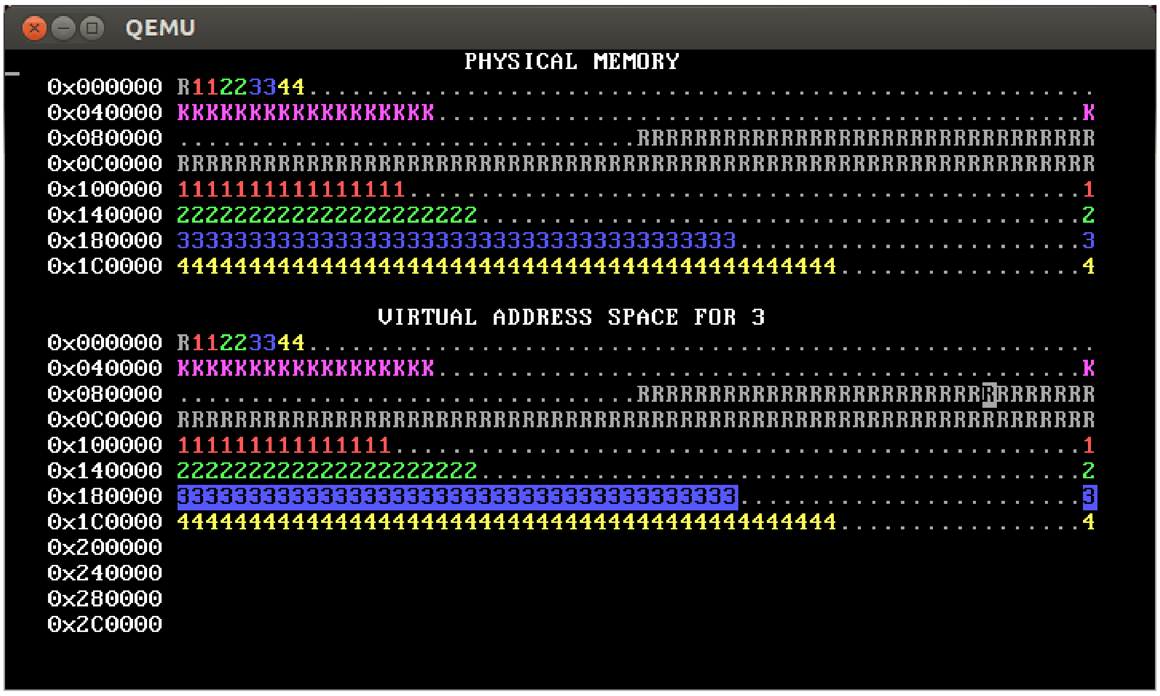
```
                              PHYSICAL MEMORY
0x000000 R11223344.......................................................
0x040000 KKKKKKKKKKKKKKKK...............................................K
0x080000 ................................RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
0x0C0000 RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
0x100000 11111111111111111111...........................................1
0x140000 222222222222222222222222222222.................................2
0x180000 33333333333333333333333333333333333333333333333................3
0x1C0000 4444444444444444444444444444444444444444444444444444444444444444

                        VIRTUAL ADDRESS SPACE FOR 3
0x000000 R11223344.......................................................
0x040000 KKKKKKKKKKKKKKKK...............................................K
0x080000 ................................RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
0x0C0000 RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
0x100000
0x140000
0x180000 33333333333333333333333333333333333333333333333                3
0x1C0000
0x200000
0x240000
0x280000
0x2C0000
```

That is, each process only has permission to access its own pages. You can tell this because only its own pages are shown in reverse video.

What goes in per-process page tables:

- The initial mappings for addresses less than `PROC_START_ADDR` should be copied from those in `kernel_pagetable`. You can use a loop with `virtual_memory_lookup()` and `virtual_memory_map()` to copy them. Alternately, you can copy the mappings from the kernel's page table into the new page tables; this is faster, but make sure you copy the right data!
- The initial mappings for the user area—addresses greater than or equal to `PROC_START_ADDR`—should be inaccessible to user processes (i.e., `PTE_U` should not be set for these PTEs). In

our solution (shown above), these addresses are *totally* inaccessible (so they show as blank), but you can also change this so that the mappings are still there, but accessible only to the kernel, as in this diagram (animated, color version online):



```
                              PHYSICAL MEMORY
0x000000 R11223344.......................................................
0x040000 KKKKKKKKKKKKKKKKKK.............................................K
0x080000 ................................RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
0x0C0000 RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
0x100000 1111111111111111...............................................1
0x140000 222222222222222222222..........................................2
0x180000 333333333333333333333333333333333333333........................3
0x1C0000 4444444444444444444444444444444444444444444444.................4

                        VIRTUAL ADDRESS SPACE FOR 3
0x000000 R11223344.......................................................
0x040000 KKKKKKKKKKKKKKKKKK.............................................K
0x080000 ................................RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
0x0C0000 RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
0x100000 111111111111111................................................1
0x140000 222222222222222222222..........................................2
0x180000 333333333333333333333333333333333333333........................3
0x1C0000 4444444444444444444444444444444444444444444444.................4
0x200000
0x240000
0x280000
0x2C0000
```

The reverse video shows that this OS also implements process isolation correctly.

How to implement per-process page tables:

- Change `process_setup()` to create per-process page tables.
- We suggest you write a `copy_pagetable(x86_64_pagetable *pagetable, int8_t owner)` function that allocates and returns a new page table, initialized as a *full copy* of `pagetable` (including *all* mappings from `pagetable`). This function will be useful in Stage 5. In `process_setup()` you can modify the page table returned by `copy_pagetable()` according to the requirements above. Your function can use `pageinfo[]` to find free pages to use for page tables. Read about `pageinfo[]` at the top of `kernel.c`.
- Remember that the x86-64 architecture uses *four-level* page tables.
- The easiest way to copy page tables involves an *allocator* function suitable for passing to `virtual_memory_map()`.
- You'll need at least to allocate a level-1 page table and initialize it to zero. You can also set up the whole four-level page table skeleton (for addresses `0...MEMSIZE_VIRTUAL - 1`) yourself; then you don't need an allocator function.
- A physical page is free if `pageinfo[PAGENUMBER].refcount == 0`. Look at the other code in `kernel.c` for some hints on how to examine the `pageinfo[]` array.
- All of process `P`'s page table pages must have `pageinfo[...].owner == P` or WeensyOS's consistency-checking functions will fail. This will affect your allocator function. (Hint: Don't forget that global variables are allowed in your code!)

If you create an incorrect page table, WeensyOS might crazily reboot. Don't panic! Add `log_printf()` statements. Another useful technique that may at first seem counterintuitive: *add infinite loops to your kernel* to track down exactly where a fault occurs. (If the OS hangs without crashing once you've added an infinite loop, then the crash you're debugging must occur at a point in the kernel's execution after your infinite loop's place in the code.)

Stage 3: Virtual Page Allocation

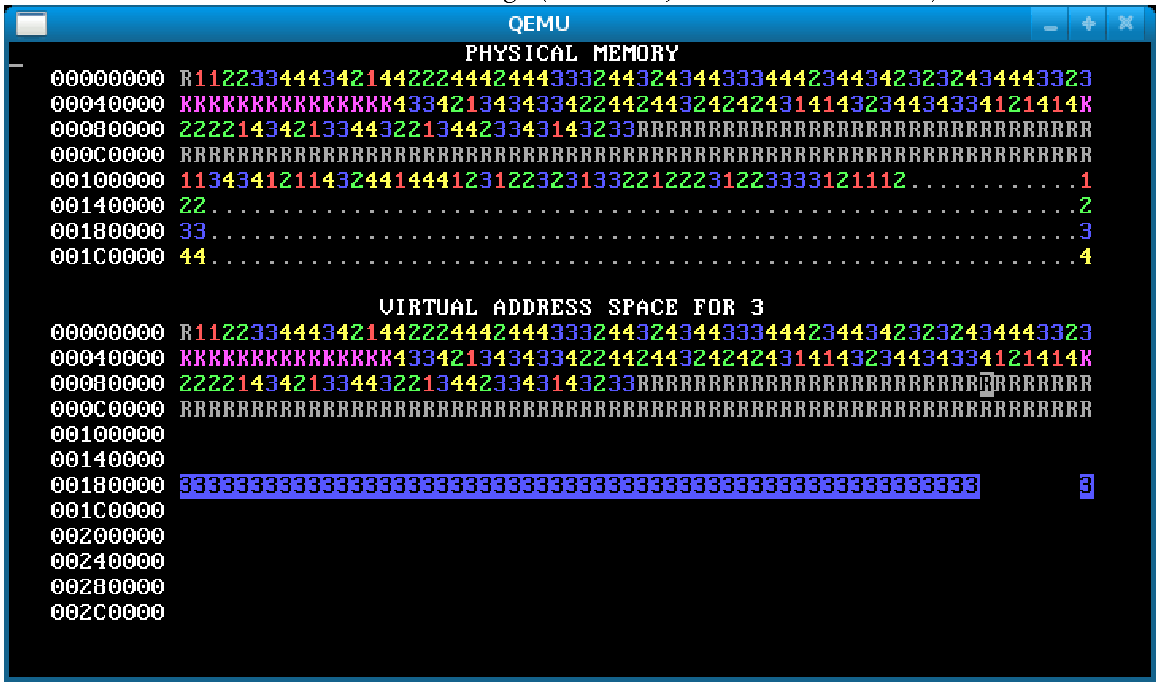
Thus far in CW4, WeensyOS processes have used *physical page allocation*: the page with *physical* address X is used to satisfy the `sys_page_alloc(X)` allocation request for *virtual* address X. This strategy is inflexible and limits utilization. Change the implementation of the `INT_SYS_PAGE_ALLOC` system call so that it can use *any free physical page* to satisfy a `sys_page_alloc(X)` request.

Your new `INT_SYS_PAGE_ALLOC` code must perform the following tasks:

- Find a free physical page using the `pageinfo[]` array. Return `-1` to the application if you can't find one. Use any algorithm you'd like to find a free physical page; in our model solution, we just return the first one we find.
- Record the physical page's allocation in `pageinfo[]`.
- Map that physical page at the requested virtual address.

Don't modify the `physical_page_alloc()` helper function, which is also used by the program loader. You can write a new function if you need to.

Here's how our OS looks after this stage (animated, color version online):

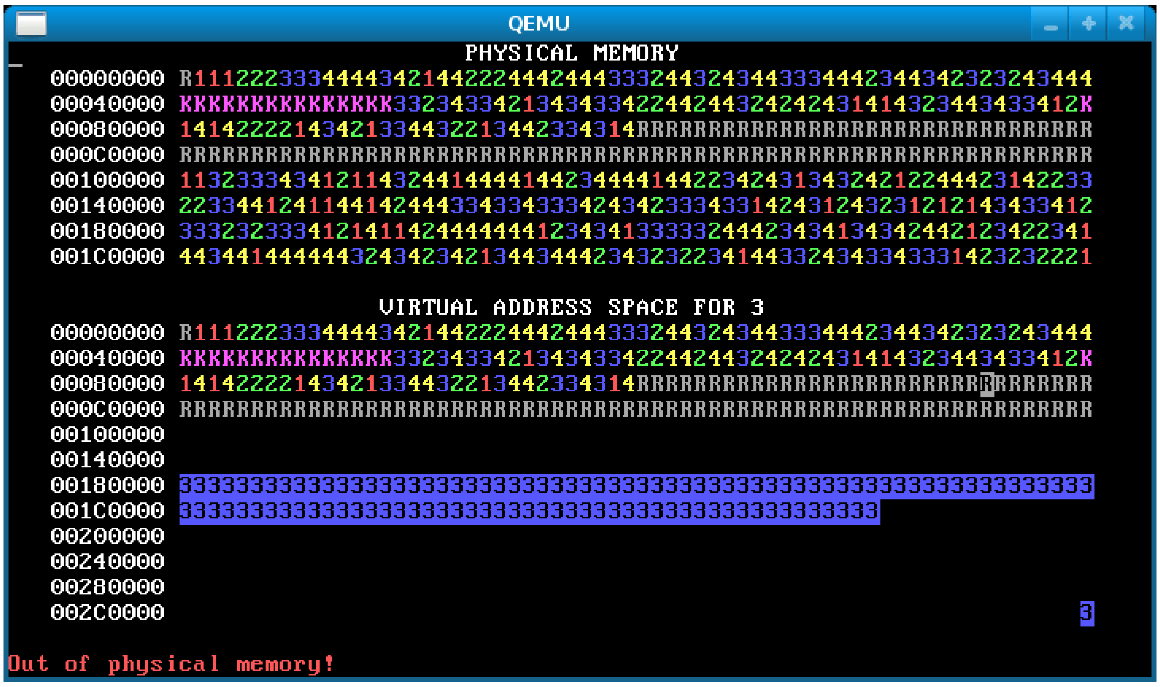

QEMU

```
                              PHYSICAL MEMORY
00000000 R112233444342144222444244433324432434433344423443423232434443323
00040000 KKKKKKKKKKKKKKK433421343433422442443242424314143234434334121414K
00080000 22221434213344322134423343143233RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
000C0000 RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
00100000 113434121143244144412312232313322122231223333121112............1
00140000 22.............................................................2
00180000 33.............................................................3
001C0000 44.............................................................4

                         VIRTUAL ADDRESS SPACE FOR 3
00000000 R112233444342144222444244433324432434433344423443423232434443323
00040000 KKKKKKKKKKKKKKK433421343433422442443242424314143234434334121414K
00080000 22221434213344322134423343143233RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
000C0000 RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
00100000
00140000
00180000 33333333333333333333333333333333333333333333333333333333       3
001C0000
00200000
00240000
00280000
002C0000
```

Stage 4: Overlapping Virtual Address Spaces

Now the processes are isolated, which is excellent. But they're still not taking full advantage of virtual memory. Isolated address spaces can use *the same* virtual addresses for *different* physical memory. There's no need to keep the four processes' address spaces disjoint.

In this stage, change each process's stack to start from address `0x300000 == MEMSIZE_VIRTUAL`. Now the processes have enough heap room to use up all of physical memory! Here's how the memory map will look after you've done it successfully (animated, color version online):



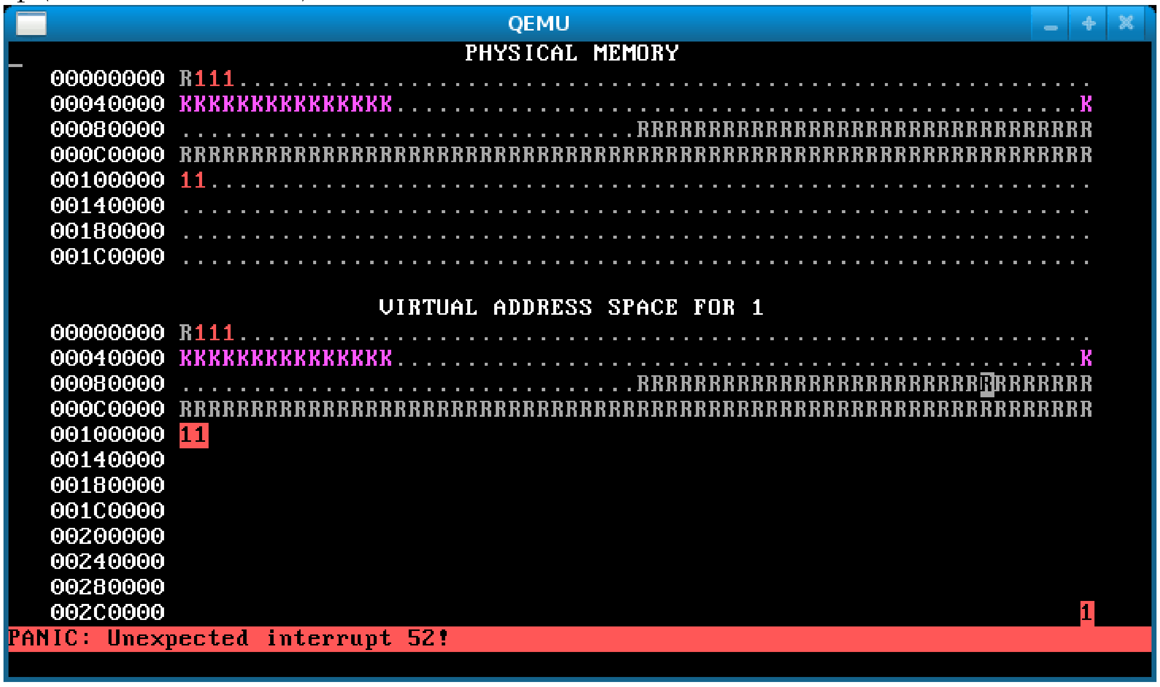
```
QEMU
                               PHYSICAL MEMORY
     00000000 R111222333444434214422244424443332443243443334442344342323243444
     00040000 KKKKKKKKKKKKKKK332343342134343342244244324242431414323443433412K
     00080000 14142222143421334432213442334314RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
     000C0000 RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
     00100000 1132333434121143244144441442344441442234243134324212244423142233
     00140000 2233441241144142444334334333424342333433142431243231212143433412
     00180000 3332323334121411424444444123434133333244423434134342442123422341
     001C0000 4434414444443243423421344344423432322341443324343343331423232221

                         VIRTUAL ADDRESS SPACE FOR 3
     00000000 R111222333444434214422244424443332443243443334442344342323243444
     00040000 KKKKKKKKKKKKKKK332343342134343342244244324242431414323443433412K
     00080000 14142222143421334432213442334314RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
     000C0000 RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
     00100000
     00140000
     00180000 3333333333333333333333333333333333333333333333333333333333333333
     001C0000 3333333333333333333333333333333333333333333333333
     00200000
     00240000
     00280000
     002C0000                                                                3

Out of physical memory!
```

If there's no physical memory available, `sys_page_alloc()` should return an error to the caller (by returning -1). Our model solution additionally prints "`Out of physical memory!`" to the console when this happens; you don't need to.

Stage 5: Fork

The `fork()` system call is one of Unix's great ideas. It starts a new process as a "copy" of an existing one. The `fork()` system call appears to return twice, once to each process. To the child process, it returns 0. To the parent process, it returns the child's process ID.

Run WeensyOS with `make run` or `make run-console`. At any time, press the "`f`" key. This will soft-reboot WeensyOS and cause it to run a single process from the `p-fork` application, rather than the gang of `allocator` processes. You should see something like this in the memory map (color version online):



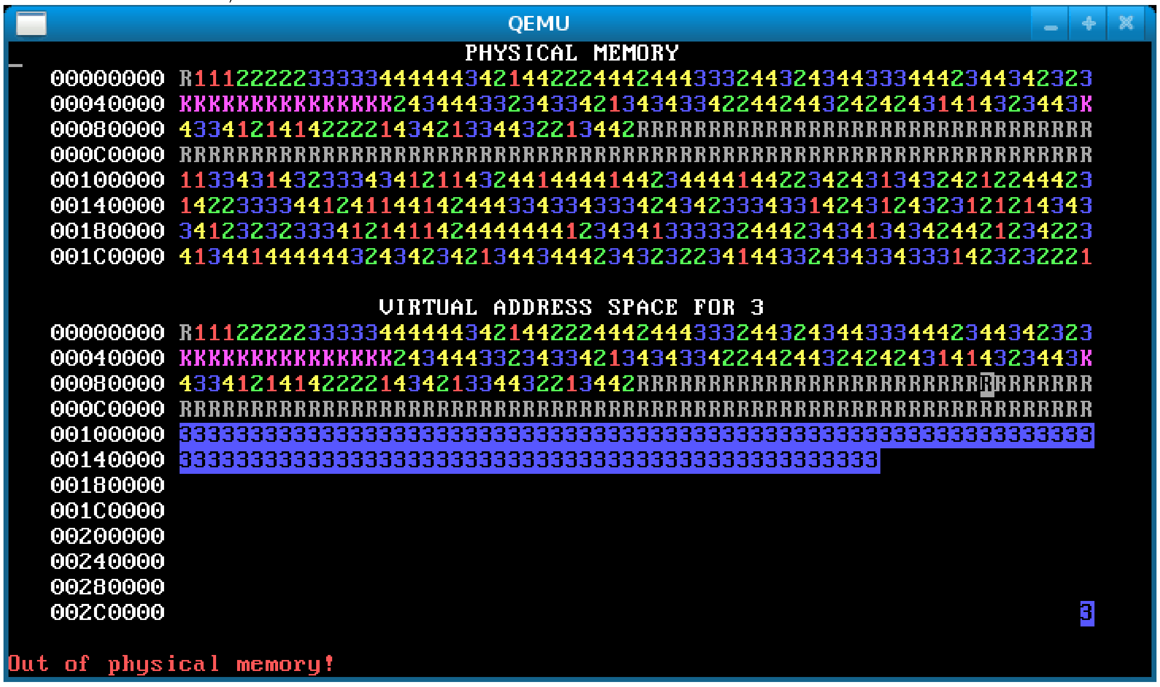
```
QEMU
                               PHYSICAL MEMORY
     00000000 R111............................................................
     00040000 KKKKKKKKKKKKKKK................................................K
     00080000 ................................RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
     000C0000 RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
     00100000 11..............................................................
     00140000 ................................................................
     00180000 ................................................................
     001C0000 ................................................................

                         VIRTUAL ADDRESS SPACE FOR 1
     00000000 R111............................................................
     00040000 KKKKKKKKKKKKKKK................................................K
     00080000 ................................RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
     000C0000 RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
     00100000 11
     00140000
     00180000
     001C0000
     00200000
     00240000
     00280000
     002C0000                                                                1
PANIC: Unexpected interrupt 52!
```

That's because you haven't implemented `fork()` yet.

How to implement `fork()`:

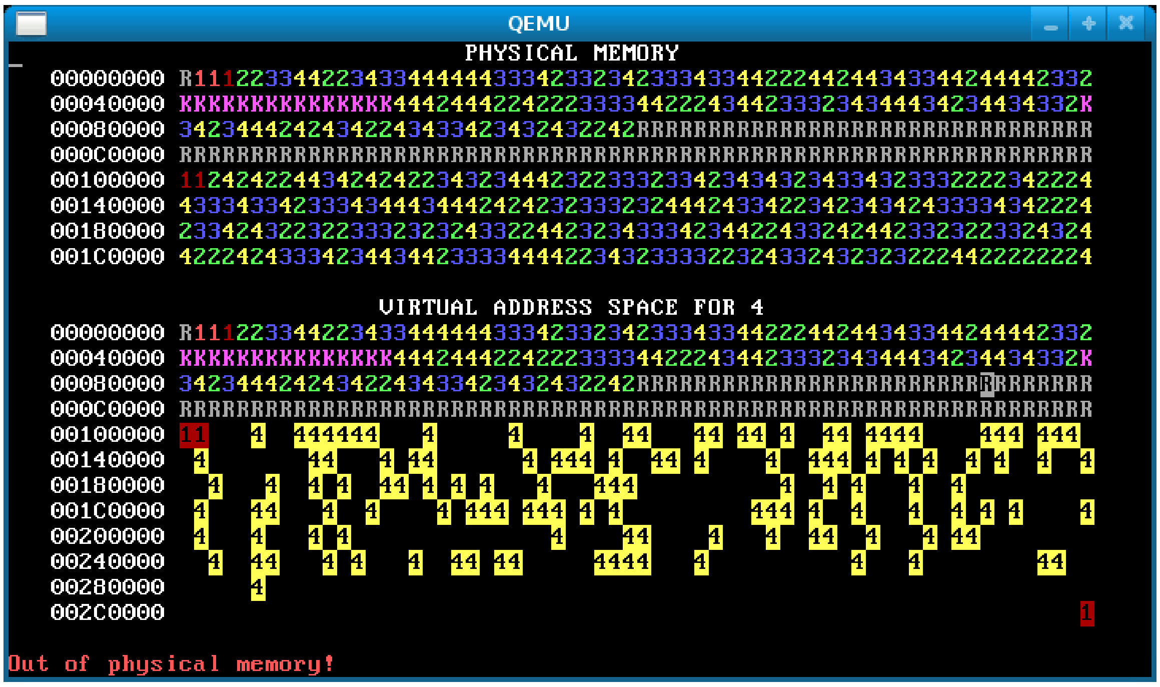
- When a process calls `fork()`, look for a free process slot in the `processes[]` array. Don't use slot 0. If no free slot exists, return `-1` to the caller.
- If a free slot is found, make a copy of `current->p_pagetable`, the forking process's page table, using your function from earlier.
- But you must also copy the *process data* in every application page shared by the two processes. The processes should not share any writable memory except the console (otherwise they wouldn't be isolated). So `fork()` must examine every virtual address in the old page table. Whenever the parent process has an application-writable page at virtual address `V`, then `fork()` must allocate a new physical page `P`; copy the data from the parent's page into `P` using `memcpy()`; and finally map page `P` at address `V` in the child process's page table. (There's a Linux man page for `memcpy()`.)
- The child process's registers are initialized as a copy of the parent process's registers, except for `reg_rax`.
- Use `virtual_memory_lookup()` to query the mapping between virtual and physical addresses in a page table.

When you're done, you should see something like the below after pressing "`f`" (animated, color version online):



```
                              PHYSICAL MEMORY
   00000000 R111222223333344444434214422244424443332443243443334442344342323
   00040000 KKKKKKKKKKKKKKK2434443323433421343433422442443242424314143234 43K
   00080000 433412141422222143421334432213442RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
   000C0000 RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
   00100000 1133431432333434121143244144441442344441442234243134324212244423
   00140000 1422333344124111441424443343343334243423334331424312432312121434 3
   00180000 3412323233341214114244444441234341333332444234341343424421234223
   001C0000 4134414444443243423421344344423432322341443324343343331423232221

                          VIRTUAL ADDRESS SPACE FOR 3
   00000000 R111222223333344444434214422244424443332443243443334442344342323
   00040000 KKKKKKKKKKKKKKK243444332343342134343342244244324242431414323443K
   00080000 433412141422222143421334432213442RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
   000C0000 RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
   00100000 3333333333333333333333333333333333333333333333333333333333333333
   00140000 3333333333333333333333333333333333333333333333333
   00180000
   001C0000
   00200000
   00240000
   00280000
   002C0000                                                                3

Out of physical memory!
```

An image like the below, however, means you forgot to copy the data for some pages, so the processes are actually "sharing" stack and/or data pages when they should not (animated, color version online):



```
                                PHYSICAL MEMORY
   00000000 R111223344223433444444333423323423334334422244244343344244442332
   00040000 KKKKKKKKKKKKKKK444244422422233334422243442333234344434234434332K
   00080000 34234442424342243433423432432242RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
   000C0000 RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
   00100000 1124242244342424223432344423223332334234343234334323332222342224
   00140000 4333433423334344434442424232333232444243342234234342433334342224
   00180000 2334243223223332323243322442323433342344224332424423323223324324
   001C0000 4222424333423443442333344442234323333223243324323232224422222224

                          VIRTUAL ADDRESS SPACE FOR 4
   00000000 R111223344223433444444333423323423334334422244244343344244442332
   00040000 KKKKKKKKKKKKKKK444244422422233334422243442333234344434234434332K
   00080000 34234442424342243433423432432242RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
   000C0000 RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
   00100000 11   4  444444   4     4    4  44   44 44 4  44 4444    444 444
   00140000  4       44   4 44      4 444 4  44 4    4  444 4 4 4  4 4  4  4
   00180000   4   4  4 4  44 4 4 4   4   444          4  4 4   4  4
   001C0000  4   44   4  4    4 444 444 4 4         444 4  4   4  4 4 4    4
   00200000  4   4   4 4              4    44    4   4  44  4   4 44
   00240000   4  44   4 4   4  44 44     4444   4          4   4        44
   00280000      4
   002C0000                                                                1

Out of physical memory!
```

Tips

The kernel defines a constant, HZ, which determines how many times per second the kernel's clock ticks. Don't change this value—there is absolutely no need to do so while solving CW4, and doing so will likely cause your code to fail our tests!

After you run any of our per-stage `make grade-N` tests (see below), if you happen to examine the `log.txt` file, you'll see a vast amount of output therein that we generate for use in the automated tests. You can ignore it (and it will be absent when you run with `make run` while you are developing, so it won't clutter your own debugging `log_printf()`s in those runs).

Running the Tests and Submitting

Unlike for prior courseworks in 0019, the automated tests for CW4 are *not* intended to help you debug! As stated at the start of this handout, the visual memory map displayed by QEMU as your WeensyOS kernel runs is the best way to determine how your code is behaving in all the stages of CW4. The automated tests for CW4 are simply for you to confirm that you've completed a stage (and they are the tests the grading server will use to assign grades). So run with `make run` to visualize how memory is being used while you are coding and validating your design. Then only switch to the automated tests below when you think you're done a stage and want to double-check.

There are five tests, one for each stage. You can run each of them with the shell commands `make grade-one` through `grade-five`. Note that the stage numbers are written out in text and not using digits. Each stage's result is all-or-nothing; pass or fail. The tests report success or failure, and nothing more (see above—the graphical memory map is how you should determine how your code is behaving).

There are three invariants in *all* five stages' tests that your code must uphold; if your code doesn't uphold any invariant, you'll receive an error to that effect and fail the test (regardless of stage). These invariants are:

- The CGA console must be accessible by all processes (this requirement is discussed in the text above on stage 1).

- If we consider process *P*, there should be no virtual page in *P*'s page table at a user-space address (i.e., whose address is above the kernel/user virtual address split point) that is owned by *P* but not accessible by *P*.

- When we run our tests, we configure the WeensyOS kernel to exit after 10 seconds of execution (1000 WeensyOS kernel "ticks").[4] If a bug in your code makes the kernel crash before 1000 WeensyOS kernel ticks, you'll fail the test on which that happens. Alternatively, if your kernel enters an infinite loop, and thus never reaches our exit at 1000 ticks:

  – When you run the tests locally in your VM, the VM will get stuck in the infinite loop and never exit, so the tests will hang and you'll need to terminate your hung kernel. Do so by opening another terminal window and issuing the command `make kill`, which will kill all QEMU processes you have running.
  – When the grading server runs the tests, it will eventually time out and kill the entire VM where the tests of your code are running after 10 minutes (which will cause you to fail all tests).

These invariants are reasonable: regardless of what your memory map display looks like, a good solution should neither crash nor enter an infinite loop.

There is only one `kernel.c` file; you implement the five stages cumulatively in it. As such, when you add further stages' functionality, you should not break that of prior stages. You can confirm this visually when you scrutinize the memory map shown by QEMU. Our tests also verify it: each stage's test runs all prior stages' tests. If any of the prior stages' tests fail, the "current" stage's test is deemed to have failed.

One interesting consequence of the cumulative nature of CW4 is that if you introduce a regression in your changes for some stage after the first stage that causes one or more prior stages' tests to fail, you'll not only lose the marks for the current stage you are working on, but also lose them for any prior stage whose tests your regression causes to fail. If you need to submit (whether at the deadline or late with late days), and find this has happened, do not despair: simply revert your code to the last good version before your regression (using the life-saving history provided by GitHub—so do make sure you commit and push often!), and you'll be back to passing those earlier stages' tests again.

Once again, we urge you to get started early.

Submitting via GitHub

We will deem the timestamp of your CW4 submission to be the timestamp on GitHub's server of the last commit you make before the submission deadline of 5:05 PM GMT on 8th March 2019. Your mark will be the mark you receive on the automated tests for that version of your code. Coursework lateness penalties are described in detail on the 0019 class web site.

If you wish to submit after the deadline, *you must take the following steps for your coursework to be marked:*

1. When you wish to receive a mark for a version of your code that you push to GitHub after the submission deadline, you must begin your commit log message for that commit with the exact string `LATESUBMIT`. *Our grading system will not record a mark for your*

[4] We also disable `ALLOC_SLOWDOWN` in `p-allocator.c` and `p-fork.c` during our tests, so that memory allocation proceeds much more quickly, at machine speed rather than human-vision speed. Thus 1000 ticks are plenty of time for the workload to run and exhibit how your kernel's virtual memory system behaves.

late submission unless you comply with this requirement. We follow this policy so that if a student "accidentally" pushes a further commit after the deadline, they aren't penalized for a late submission.

2. If you wish to claim late days, per the 0019 late days policy (details on the 0019 class web site), you must post a private message on Piazza to the Instructors *within one hour after pushing the version of your code you wish to submit late to GitHub* with a subject line `CW4 LATE DAYS` and a body stating the number of late days you would like to claim. *Please follow these instructions exactly: note the all capitals, and please use this exact subject line string.*

3. You may make only one late submission (i.e., one GitHub commit with the initial string `LATESUBMIT` and one Piazza posting with the subject line `CW4 LATE DAYS`). If you make more than one late submission, we will only mark the first one.

Academic Honesty

This coursework is an *individual coursework*. Every line of code you submit must have been written by you alone, and must not be a reproduction of the work of others—whether from the work of students in this (or any other) class from this year or prior years, from the Internet, or elsewhere (where "elsewhere" includes code written by anyone anywhere).

Students are permitted to discuss with one another the definition of a problem posed in the coursework and the general outline of an approach to a solution, but not the details of or code for a solution. Students are strictly prohibited from showing their solutions to any problem (in code or prose) to a student from this year or in future years. In accordance with academic practice, students must cite all sources used; thus, if you discuss a problem with another student, you must state in your solution that you did so, and what the discussion entailed.

Any use of *any* online question-and-answer forum (other than the CS 0019 Piazza web site) to obtain assistance on this coursework is strictly prohibited, constitutes academic dishonesty, and will be dealt with in the same way as copying of code. The same goes for use of any online material specifically directed toward solving this coursework.

You are free to read other reference materials found on the Internet (and any other reference materials), ***apart from** any source code that implements x86 virtual memory.* You may of course use the code we have given you. *Again, all other code you submit must be written by you alone.*

Copying of code from student to student or from the Internet is a serious infraction; it typically results in automatic awarding of zero marks to all students involved, and is viewed by the UCL administration as cheating under the regulations concerning Examination Irregularities. Penalties imposed can extend all the way to exclusion from all further examinations at UCL. The course staff use extremely accurate plagiarism detection software to compare code submitted by all students this year and in prior years (as well as code found on the Internet) and identify instances of copying of code. This software sees through attempted obfuscations such as renaming of variables and reformatting, and compares the actual parse trees of the code. Rest assured that it is far more work to modify someone else's code to evade the plagiarism detector than to write code for the assignment yourself!

Read the Piazza Web Site

You will find it useful to monitor the 0019 Piazza web site during the period between now and the due date for the coursework. Any announcements (*e.g.,* helpful tips on how to work around unexpected problems encountered by others) will be posted there. And you may ask questions there. *Please remember that if you wish to ask a question that reveals the design of your solution,* *you must mark your post on Piazza as private, so that only the instructors may see it.* Questions about the interpretation of the coursework text, or general questions about C that do not relate to your solution, however, may be asked publicly—and we encourage you to do so, so that the whole class benefits from the discussion.

References

CS:APP/3e Chapter 9, particularly §9.7

Acknowledgement

Eddie Kohler created WeensyOS.