CS202 Review Session 5 Based on previous notes of cs202 review session 5 and 6.

Yuxia Zhan - Fall 2025

Contents

1. Overview of Lab4

As you learned from the class, kernel gives process the **illusion that the process can use all the virtual memory space**, even when the physical memory space is smaller than the virtual memory space. So when you see virtual mem is 3M while physical mem only has 2M, you shouldn't feel surprised.

The key point here is, **how to allocate physical pages, and how to create and maintain page table for each process**. From my perspective, this lab will be much easier if you think all the steps from this perspective. That is, all the steps are essentially asking you the following question:

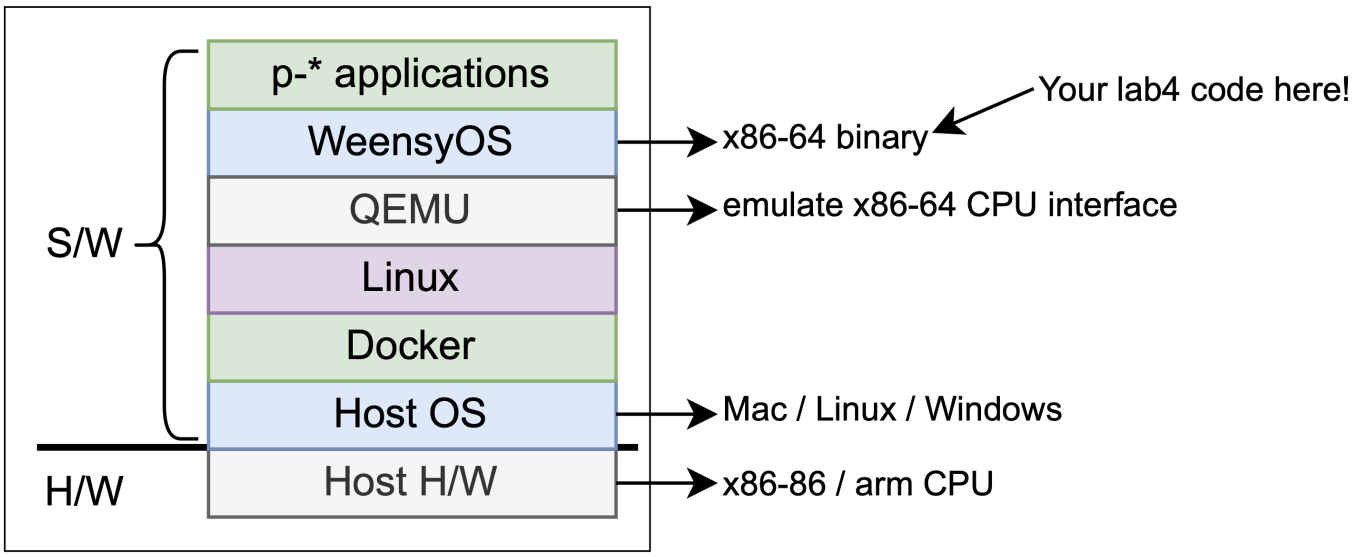
"How should you (on behalf of kernel) allocate physical pages or maintain the page table such that you can balabala".

And you can replace this "balabala" with:

- (step 1) isolate the kernel's memory from processes'
- (step 2) isolate the processes' memory from each other
- (step 3) achieve a non identity-mapping between VPN and PPN
- (step 4) make different processes have overlapping addr space
- (step 5) support fork() syscall
- (step 6) [extra] let processes to share some pages
- (step 7) [extra] delete process's memory pages if it exits

Your job in this lab is to implement all these to shift from a "naive version of virtual memory mechanism" to the version you learned in the class.

Figure 1: S/W stack of lab4.



Before we going into more details of Lab4, it would be helpful to review this s/w stack, you might see a more detailed version in this Monday's class. After lab 2 and 3, you should be familiar with the stack below "Linux", and what's new in this lab are QEMU and WeensyOS. QEMU is an application running on Linux, and it emulates x86-64 interface for anything that runs on top of it. WeensyOS is an x86-64 binary, running on top of QEMU. And the code you will write for this lab will be on WeensyOS level.

2. How to read the "map"

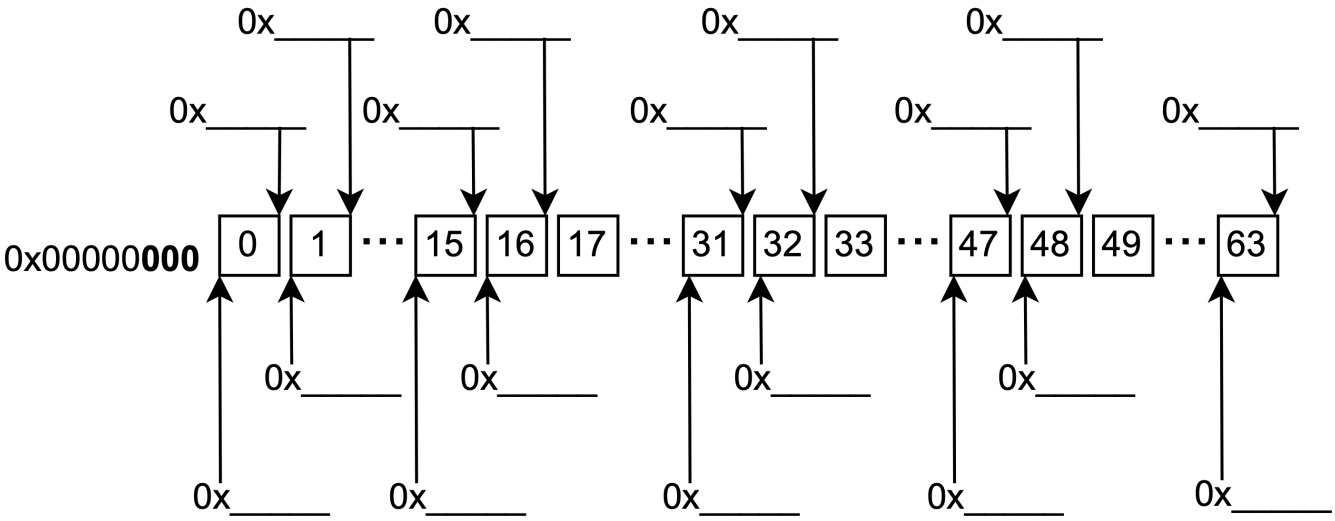
You will find there are two parts in this map, the upper part is showing the physical memory layout, while the lower part shows the virtual memory layout for different processes. Let's focus on the upper part for now.

- Each of this character is a page.
- What is the page size? 4KB.

```
// x86-64.h
#define PAGEOFFBITS     12                  // # bits in page offset
#define PAGESIZE        (1 << PAGEOFFBITS)  // Size of page in bytes
```

- How many pages are there in the first row? 64.
- By counting or by calculating. (0x40,000 => 0x40 pages => 64 pages)

Figure 2: Address of the first row of the map.



- What is the address of the first byte of the 0-th page? 0x000.
- What is the address of the last byte of the 0-th page? 0xfff.
- What is the address of the first byte of the 63-th page? 0x3f000.
- What is the address of the last byte of the 63-th page? 0x3ffff. (That's why you will see the first byte's address of second row is 0x40000).
- What's the page number corresponding to addr 0x4f123? 0x4f.

```
// x86-64.h
#define PAGENUMBER(ptr) ((int) ((uintptr_t) (ptr) >> PAGEOFFBITS))
```

- What is the addr of the first byte of the 65-th page? 0x41,000

```
// x86-64.h
#define PAGEADDRESS(pn) ((uintptr_t) (pn) << PAGEOFFBITS)
```

- How many pages are there in total in physical memory space? 512 pages (0x200)

```
// kernel.h
// Physical memory size
#define MEMSIZE_PHYSICAL        0x200000
// Number of physical pages
#define NPAGES                  (MEMSIZE_PHYSICAL / PAGESIZE)
```

- How large is the virtual memory space? 768 pages (0x300)

```
// kernel.h
// Virtual memory size
#define MEMSIZE_VIRTUAL         0x300000
```

Calculate the VPN1 ~ 4.

```
Largest virtual addr: 0x2fffff
(translate hex to binary)
=> 0b 0010 1111 1111 1111 1111 1111
(align with virtual addr layout 9 + 9 + 9 + 9 + 12 = 48, so append 24 0s.)
=> 0b (0000 0000 0)(000 0000 00)(00 0000 001)(0 1111 1111)(1111 1111 1111)
=> 0b (   VPN1    )(   VPN2    )(   VPN3    )(   VPN4    )(    offset    )
```

- How many L-1 page directory do you need to handle the mapping here? 1 (with only 1 PTE set)
- L-2. 1 (with only 1 PTE set)
- L-3. 1 (with 2 PTEs set)
- L-4. 2.

The char represents the owner of the page. K for kernel, R for reserved, 1 ~ 4 for different processes. What does '.' and ' ' stand for? what's the difference? Which flag is relevant in PTE?

- '.' for empty but mapped
- ' ' for unmapped
- bit-0, "P" flag.

You might notice there are two types of format to show a page, first is with black background, while the second is with non-black background color. This is called "reverse video". What does reverse video mean?

- process is allowed to access the corresponding address

In process 1's virtual memory space, you see a page with char 2,and with reverse video color, what does that mean?

- process 2 owns the page, and process 1 is allowed to access the page. Is it good or no?

What's wrong with this "naive version of virtual memory"?

- no isolation
  - process can see kernel's pages
  - process 1 can see process 2's pages
- not an efficient usage of physical pages
  - identity-mapping from (process-1's VPN 256, PPN 256), (process-1's VPN 257, PPN 257) will shift later in step 3 to (process1-'s VPN 256, PPN 28), (process-1's VPN 257, PPN 48)
  - not using the whole virtual address space

So these exactly are the problems you are going to solve following the lab's steps !

3. Dive into WeensyOS

To figure out why after `make run`, the four processes are magically running, we should dive into the `kernel()` func first.

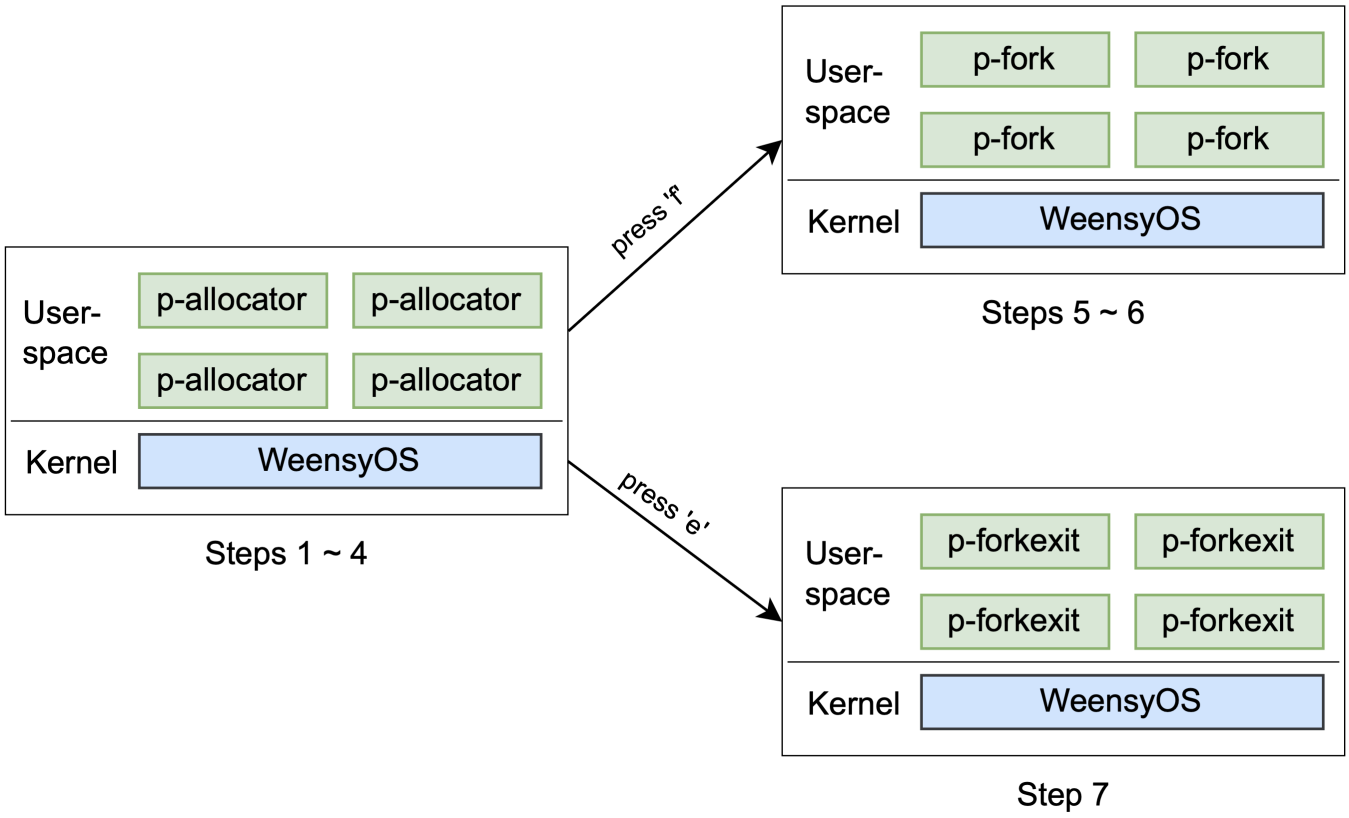
[Walkthrough] briefly walk through the `kernel()` func and `process_setup()`.

- `processes[]` and `struct proc`: pid and P_FREE
- `process_setup(i, i - 1)`: init program data as pid-i
    - set L1 page table
    - assign a new page for stack
    - map the pat for stack (naive version: identity-mapping)
    - P_RUNNABLE
    - `run(&process[1])`: only run one process

Actually, if you are interested, you can find the loaded programs (`ramimage` in `k-loader.c`, as well as `.gdbinit`), each of them runs the same source file.

Remember the s/w stack we see in Figure 1? We can actually zoom in to the WeensyOS layer and get to this **[Figure 3]**. And through out the lab4, you only needs to add code in `kernel.c` (except for part of step 6). Besides, there will be 3 different user-space applications, we only cover `p-allocator` and `p-fork` here, p-forkexit is only needed in Step 7 (extra credits).

Figure 3: WeensyOS's user-space applications.



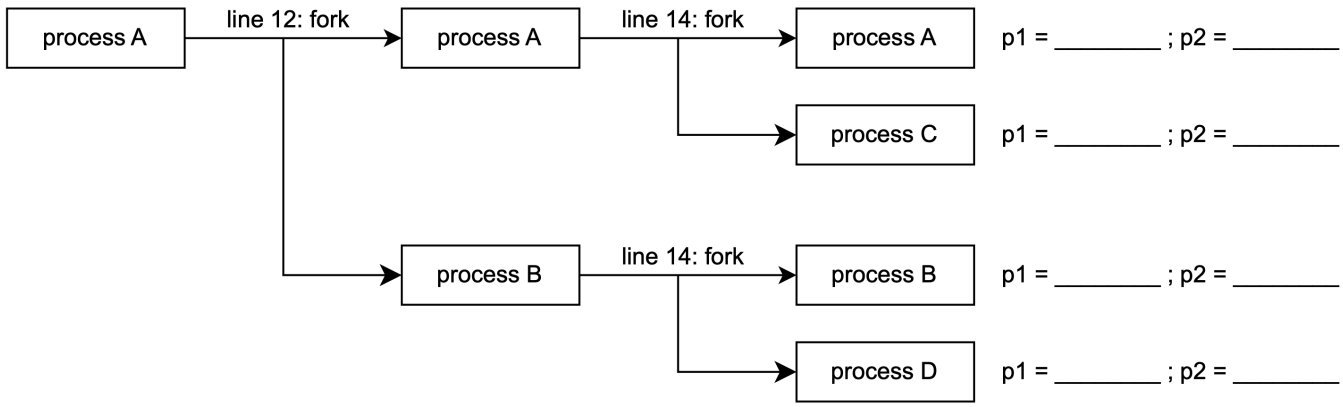
[Walkthrough] `p-allocator.c`

- keeps asking for kernel to allocate pages
- different allocate rate (controlled by `p`, which is pid)
- `yield()` (this explains why kernel only has to run one process in the beginning `run(&process[1])`)

[Walkthrough] `p-fork.c`

- `process_setup(1, 4)`: initially, kernel only setups one process

Figure 4: p-fork



Fork appears twice in the source code, what happen under the hood?

- How many times does fork() get executed? 3.
- How many processes in total? 4.

4. Chain-of-thought and important tools

Just as mentioned earlier, all these steps are guiding you to do some improvement to the naive version of virtual memory. So, the first step is to figure out what the improvement is, and what's the mechanism design behind it. Usually, the lab's webpage tells you the design, either explicitly, or by showing you some hints.

Next is to find the right place to put your code. You will need to figure out the phase the mechanism design plan to do some modification - is it during kernel initialization, or during process initialization, or during kernel handling the syscalls?

- related to "initializing the kernel" => `kernel()`
- related to "initializing a new process" => `process_setup()`
- related to "handling syscalls (page_alloc / fork / exit)" => `exceptions()`'s `case INT_SYS_xxx`

Finally, you need to get yourself familiar with related tools that will be needed during coding. These include data structures, helper functions (and their parameters) and some macros. Of course you can choose to write your own implementations of these tools, but that's not recommended. You will find it much much faster if you can first take some time fully understand them and then directly use them.

Important data structures

[walkthrough] `struct proc` and `processes[]`

[walkthrough] `struct physical_pageinfo` and `pageinfo[]`

Recall that it is a bookkeeping for physical pages' state, not the pages themselves, and not the page tables.

Important helper functions

You don't need to modify them, but you do need to understand them if you want to use them correctly.

[walkthrough] `assign_physical_page()`

- allocate a physical page to a process, by maintaining the bookkeeping `pageinfo[]`.

[walkthrough] `virtual_memory_map()`

- read the comments
    - add or modify a mapping in page table
    - set the permission flags in the page table
    - by reading it, you will know how your allocator function should look like, i.e., the signature `x86_64_pagetable* (*allocator)(void)`
    - actually, you can use to create a 4-level page tables.
- the `pagetable` passed in is the L1 page table (or called L1 page directory)
- `cur_index123` is concatenating: VPN1 || VPN 2 || VPN3.
- `if (cur_index123 != last_index123)`: if any of VPN1, VPN 2, VPN3 changes, will call `lookup_l4pagetable`
    - `lookup_l4pagetable` func: pass in L1 page table , and return the corresponding L4 page table for virt-addr
        - `for (int i = 0; i <= 2; ++i)` is to traverse L1~L3 page directory, if during traverse, no page table is found for next level (i.e., L1's `PTE_P` will check if L2 page table is present), it will allocate a new physical page to for a new page table.
        - the allocator is actually used here! as in `x86_64_pagetable* new_pt = allocator()`
- after getting the L4 page table, it will set the PTE with corresponding perm.

[walkthrough] `virtual_memory_lookup()`

- traverse the page table, to fetch the `struct vamapping` of a virt-addr

Other useful macros

We covered some of them so far, you can find the others either in the lab's website or in the header files: `lib.h` (memset and memcpy), `kernel.h` and part of `x86-64.h`.

5. More details of steps 1 ~ 5.

Table 1: Summary of 5 steps.

	Mechanism design	Where to put code	Related variable / func
Step 1	• prohibit a process from accessing kernel's pages • by setting permission flags in PTE, and owner field of pageinfo[]	• during kernel initialization; • while handling page_alloc syscall (think about a malicious user process asking for a page that belongs to kernel)	• `virtual_memory_map()`
Step 2	• each process should have distinct 4-levels of page tables	• during process initialization • particularly, replace the line `processes[pid].p_pagetable = kernel_pagetable;`	• `virtual_memory_map()` ◦ though you can write your own code to create 4-levels of page table, but it's recommended to use this helper function to do so. • `virtual_memory_lookup()` • `assign_physical_page()`
Step 3	• whenever need to allocate a page for a process, find the next free and available physical page for it	• while handling page_alloc syscall (still, remember the malicious user process case)	• `pageinfo[]`
Step 4	• set process's stack_bottom to be the top of the virtual memory space • print "out of memory" when needed.	• during process initialization • while handling page_alloc syscall (for "out of memory" print)	• console_printf (for "out of memory" print, learn to use it from other references in `kernel.c`) • still interested? see `memshow_` functions in `kernel.c` and the source code of `console_` in `lib.c`
Step 5	• create a new process, with the same copy of data as the parent process • "data" includes: all physical pages (including the pages for 4 level page tables), registers (except for rax)	• while handling fork syscall	• `processes[]` • `virtual_memory_lookup()`

6. Debugging for Lab4

Usually, to see if you implement these steps correct is by visually compare the map to the reference images you see in the website. You can also use `log_printf` and you will find the output of these print statements in `/tmp/log.txt`. For those errors that hard to debug, you can use `gdb`.

- tmux (split windows)
- to run QEMU (with W-OS booted) + gdb at the same time