HANDOUT - CS202 Review Session 5 for Lab4 (WeensyOS)

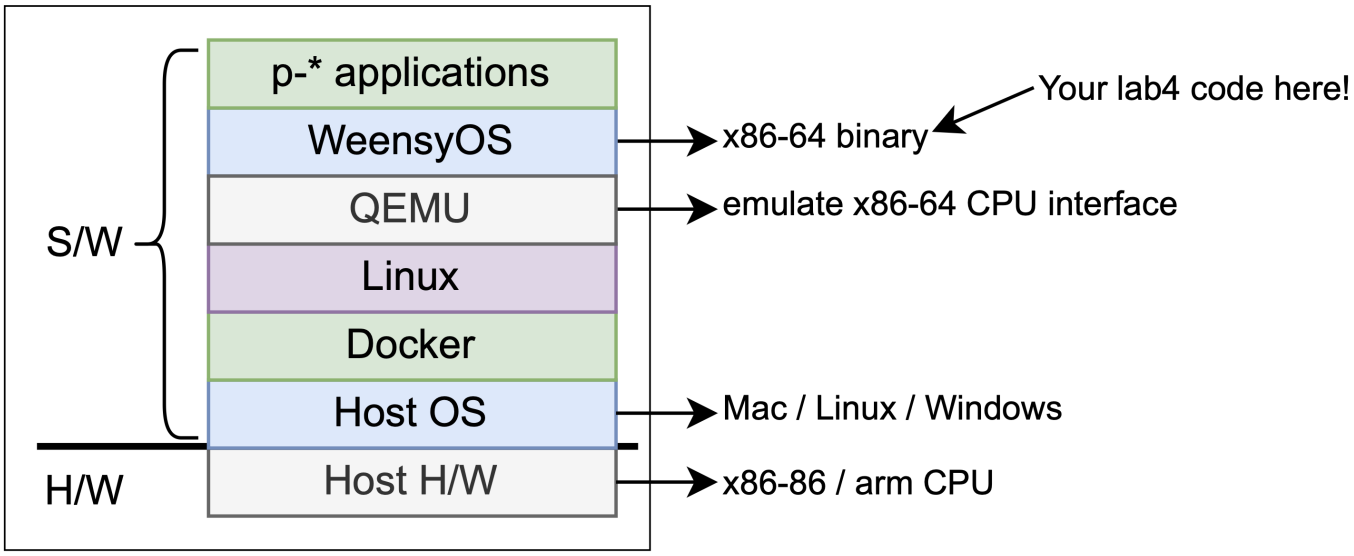
Yuxia Zhan - Fall 2025

Contents of this handout

- S/W stack of lab4
- Address of the first row of the map.
- WeensyOS's user-space applications.
- p-fork under the hood.
- Chain-of-thought and important tools.
- Summary of 5 steps.

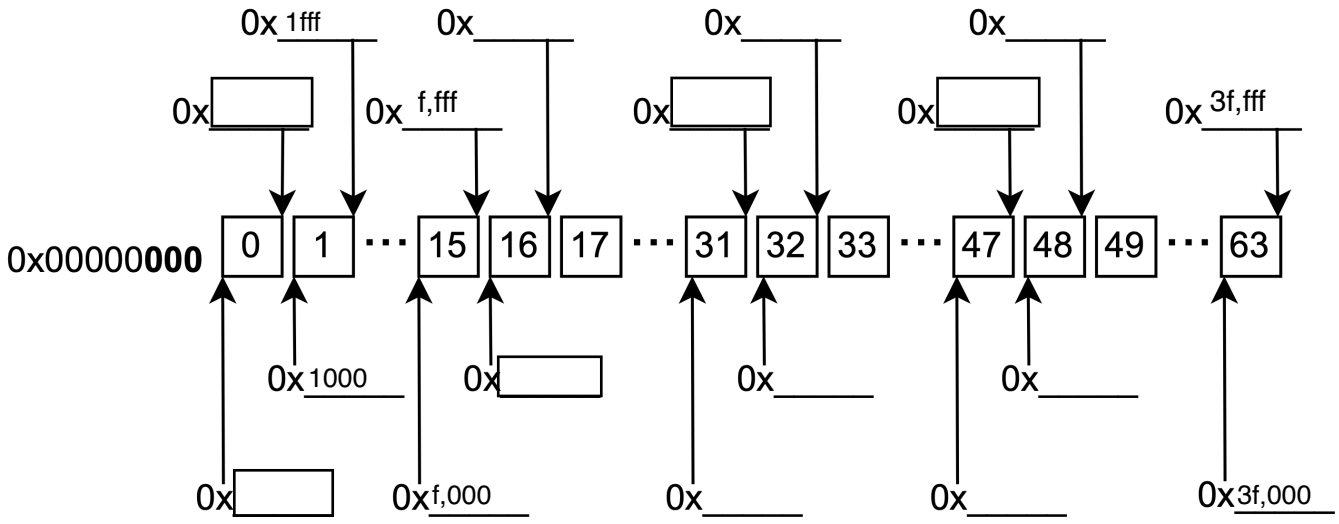
S/W stack

Figure 1: S/W stack of lab4.



Read the "map"

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




```
Largest virtual addr: 0x?????  0x2ff,fff
(translate hex to binary)
=> 0b ???...??                 0x 0010 1111 1111 1 ... 1
(align with virtual addr layout, 9 + 9 + 9 + 9 + 12 = 48 bits)
=> 0b (0_________)(0_________)(0000 0000 1_)(0 1111 1111__)(1111 1111 1111___)
=> 0b (   VPN1   )(   VPN2   )(   VPN3   )(   VPN4   )(   offset    )
```

```
// x86-64.h
// Paged memory constants
#define PAGEOFFBITS     12                  // # bits in page offset
#define PAGESIZE        (1 << PAGEOFFBITS)  // Size of page in bytes
#define PAGEINDEXBITS   9                   // # bits in a page index
level
#define NPAGETABLEENTRIES (1 << PAGEINDEXBITS) // # entries in page table
page
#define PAGENUMBER(ptr) ((int) ((uintptr_t) (ptr) >> PAGEOFFBITS))
#define PAGEADDRESS(pn) ((uintptr_t) (pn) << PAGEOFFBITS)
```

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

// Virtual memory size
#define MEMSIZE_VIRTUAL         0x30000
```

WeensyOS's user-space applications

Figure 3: WeensyOS's user-space applications.

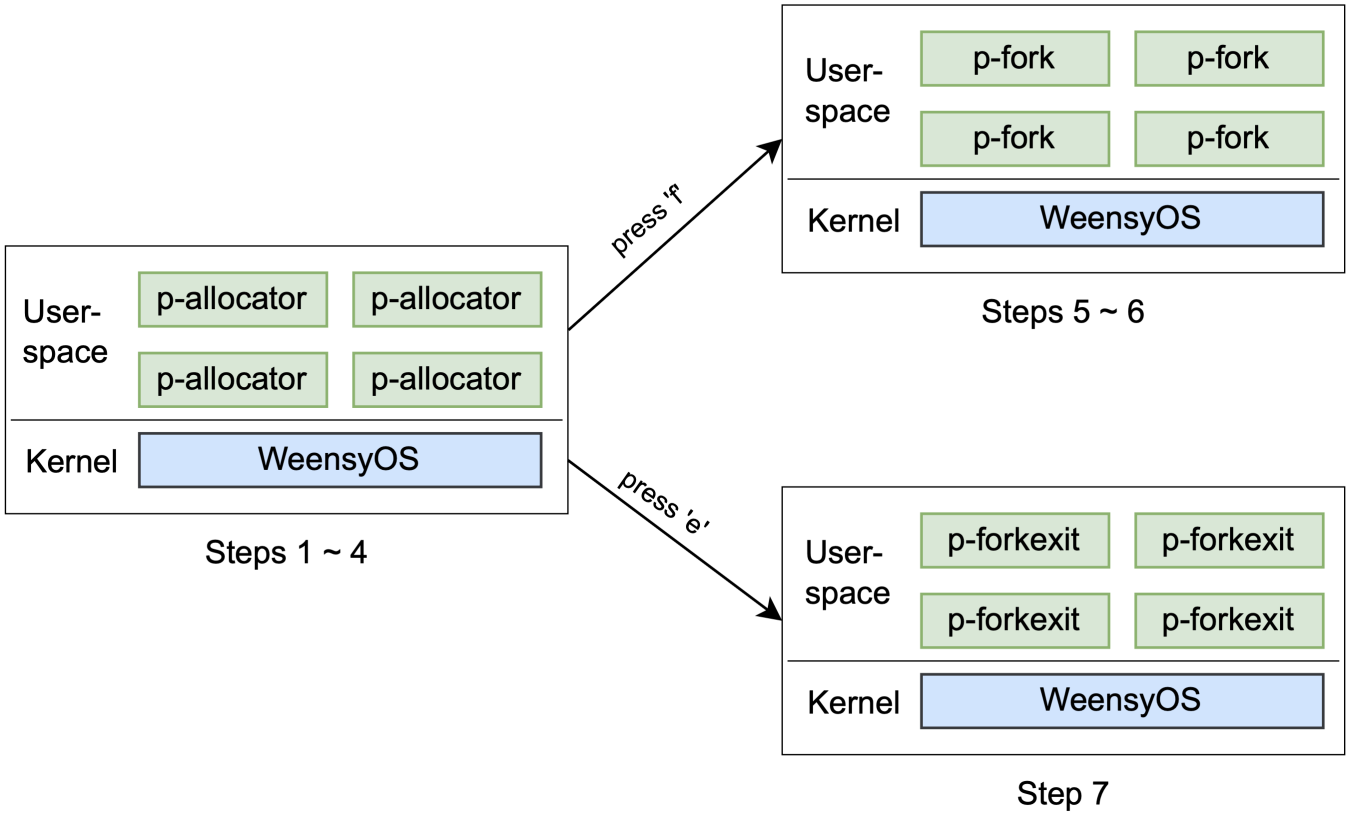
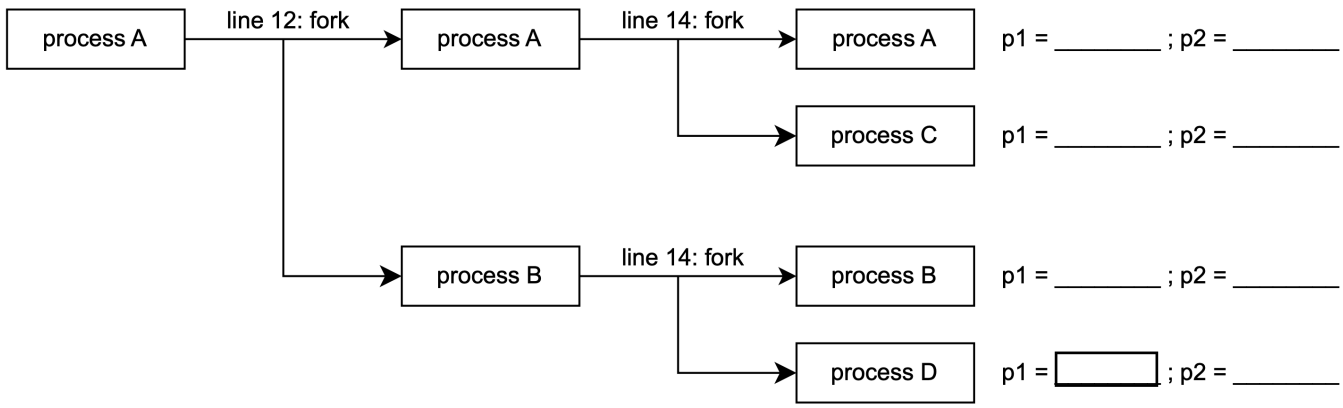


Figure 4: p-fork



Chain-of-thought and important tools.

Key point: **"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

Chain-of-thought: mechanism design => where to put code => related struct and func

About "where to put code" column in Table 1:

- 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`

Important tools:

- `struct proc` and `processes[]`
- `struct physical_pageinfo`, `pageinfo[]`, and `assign_physical_page()`

```
// kernel.c
int assign_physical_page(uintptr_t addr, int8_t owner)
```

- `virtual_memory_map()` and `lookup_l4pagetable()`

```
// k-hardware.c
int virtual_memory_map(x86_64_pagetable* pagetable, uintptr_t va,
                       uintptr_t pa, size_t sz, int perm,
                       x86_64_pagetable* (*allocator)(void))

static x86_64_pagetable* lookup_l4pagetable(
    x86_64_pagetable* pagetable, uintptr_t va, int perm,
    x86_64_pagetable* (*allocator)(void))
```

- `virtual_memory_lookup()` and `struct vamapping`

```
// k-hardware.c
vamapping virtual_memory_lookup(x86_64_pagetable* pagetable, uintptr_t
va)
```

Summary of 5 steps.

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()`