# CS 444/544 OS II
# Lab Tutorial #7

Multiprocessor Support and Cooperative Multitasking

(Lab4 – Part A)

Acknowledgement: Slides drawn heavily from Yeongjin Jiang

# Getting Started

- Checkout to lab4 branch
- Merge lab3 into lab4
- Solve merge conflicts, if there is any

# Multiprocessor Support

- Symmetric Multiprocessing Environment (SMP)
  - All CPUs are equivalent
  - Core 0, 1, 2, 3, …, all can access hardware resource, memory, etc.
- In boot
  - We boot with one CPU – Bootstrapping processor (BSP), which is core 0
  - BSP will initialize lab1 (monitor), lab2 (vm), lab3 (env), and then
  - JOS will wake up other processes (in lab4)
- APIC (Advanced Programmable Interrupt Controller)
  - Local APIC – used for getting information about how many cores available, what is current CPU, and control CPUs to run code per each CPU, etc.
  - kern/lapic.c

# Exercise 1: mmio_map_region(pa, size)

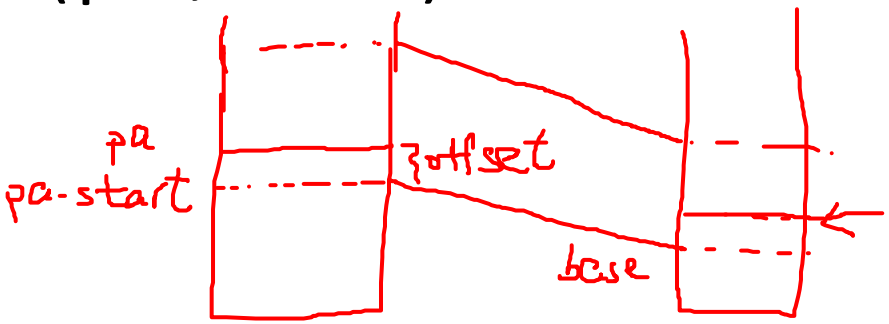
**Note**

**Exercise 1.** Implement `mmio_map_region()` in `kern/pmap.c`. To see how this is used, look at the beginning of `lapic_init()` in `kern/lapic.c`. You'll have to do the next exercise, too, before the tests for `mmio_map_region()` will run.

- Map memory for Memory-Mapped IO (MMIO)
  - Similar to `boot_map_region / region_alloc`
- Map `[pa, pa+size) to [va, va + size)`
  - Where va starts with MMIOBASE

```
static uintptr_t base = MMIOBASE;
```

# Exercise 1: mmio_map_region(pa, size)




- Corner cases
- `pa/pa+len` are not aligned with page size
  - `pa_start = ROUNDDOWN(pa, PGSIZE)`
  - `pa_end = ROUNDUP(pa+len, PGSIZE)`
  - `pa_offset = pa & 0xfff` // gets the last 12-bit of the address
- Map `[pa_start, pa_end)` to
  - `va_start = base`
  - `new_base = va_start + pa_end - pa_start`
- Return?
  - `return old_base + pa_off;`
  - Why add `pa_off`?
    - `pa` might not be page aligned, but it must return the corresponding virtual address to that physical address (with offset)

# Exercise 1: mmio_map_region(pa, size)

- What is PTE_PCD, PTE_PWT?
- PTE_PCD
  - Page Cache Disable
  - If this bit is set, CPU ignores caching memory blocks from this page
  - Required for MMIO (caching will prevent direct access to hardware!)
- PTE_PWT
  - Page Write Through
  - If this bit is set, CPU immediately forward write to memory/hardware
  - Required for MMIO (non-write through would delay hardware access!)

# Exercise 2: Waking up Application Processors

**Exercise 2.** Read `boot_aps()` and `mp_main()` in `kern/init.c`, and the assembly code in `kern/mpentry.S`. Make sure you understand the control flow transfer during the bootstrap of APs. Then modify your implementation of `page_init()` in `kern/pmap.c` to avoid adding the page at `MPENTRY_PADDR` to the free list, so that we can safely copy and run AP bootstrap code at that physical address. Your code should pass the updated `check_page_free_list()` test (but might fail the updated `check_kern_pgdir()` test, which we will fix soon).

# Exercise 2: Waking up Application Processors

- What it does?
  - Copy code for waking up APs to MPENTRY_PADDR
- Algorithm
  - code = vaddr(0x7000)
  - Move code at mpentry_start to 0x7000
- Run as the bootloader at 0x7c00

```
// Start the non-boot (AP) processors.
static void
boot_aps(void)
{
    extern unsigned char mpentry_start[], mpentry_end[];
    void *code;
    struct CpuInfo *c;

    // Write entry code to unused memory at MPENTRY_PADDR
    code = KADDR(MPENTRY_PADDR);
    memmove(code, mpentry_start, mpentry_end - mpentry_start);

    // Boot each AP one at a time
    for (c = cpus; c < cpus + ncpu; c++) {
        if (c == cpus + cpunum())  // We've started already.
            continue;

        // Tell mpentry.S what stack to use
        mpentry_kstack = percpu_kstacks[c - cpus] + KSTKSIZE;
        // Start the CPU at mpentry_start
        lapic_startap(c->cpu_id, PADDR(code));
        // Wait for the CPU to finish some basic setup in mp_main()
        while(c->cpu_status != CPU_STARTED)
            ;
    }
}
```

# Exercise 2: Waking up Application Processors

- What's at 0x7000?
  - Looks very similar to bootloader
- Why?
  - CPU 0 enabled protected mode
  - CPU 0 enabled paging
  - What about others??
    - CPU 1, 2, 3, …?
  - Are in the real mode!

```
mpentry_start:
    cli

    xorw    %ax, %ax
    movw    %ax, %ds
    movw    %ax, %es
    movw    %ax, %ss

    lgdt    MPBOOTPHYS(gdtdesc)
    movl    %cr0, %eax
    orl     $CR0_PE, %eax
    movl    %eax, %cr0

    ljmpl   $(PROT_MODE_CSEG), $(MPBOOTPHYS(start32))

.code32
start32:
    movw    $(PROT_MODE_DSEG), %ax
    movw    %ax, %ds
    movw    %ax, %es
    movw    %ax, %ss
    movw    $0, %ax
    movw    %ax, %fs
    movw    %ax, %gs

    # Set up initial page table. We cannot use kern_pgdir yet because
    # we are still running at a low EIP.
    movl    $(RELOC(entry_pgdir)), %eax
    movl    %eax, %cr3
    # Turn on paging.
    movl    %cr0, %eax
    orl     $(CR0_PE|CR0_PG|CR0_WP), %eax
    movl    %eax, %cr0

    # Switch to the per-cpu stack allocated in boot_aps()
    movl    mpentry_kstack, %esp
    movl    $0x0, %ebp       # nuke frame pointer

    # Call mp_main().  (Exercise for the reader: why the indirect call?)
    movl    $mp_main, %eax
    call    *%eax
```

# Exercise 2: Waking up Application Processors

- So we need to enable
  - Protected mode

Make sure to mark 0x7000 is in-use in page_init

  - Enable paging

  - Set stack (mpentry_kstack)
  - Call `mp_main()`

```
mpentry_start:
    cli

    xorw    %ax, %ax
    movw    %ax, %ds
    movw    %ax, %es
    movw    %ax, %ss

    lgdt    MPBOOTPHYS(gdtdesc)
    movl    %cr0, %eax
    orl     $CR0_PE, %eax
    movl    %eax, %cr0

    ljmpl   $(PROT_MODE_CSEG), $(MPBOOTPHYS(start32))

.code32
start32:
    movw    $(PROT_MODE_DSEG), %ax
    movw    %ax, %ds
    movw    %ax, %es
    movw    %ax, %ss
    movw    $0, %ax
    movw    %ax, %fs
    movw    %ax, %gs

    # Set up initial page table. We cannot use kern_pgdir yet because
    # we are still running at a low EIP.
    movl    $(RELOC(entry_pgdir)), %eax
    movl    %eax, %cr3
    # Turn on paging.
    movl    %cr0, %eax
    orl     $(CR0_PE|CR0_PG|CR0_WP), %eax
    movl    %eax, %cr0

    # Switch to the per-cpu stack allocated in boot_aps()
    movl    mpentry_kstack, %esp
    movl    $0x0, %ebp       # nuke frame pointer

    # Call mp_main().  (Exercise for the reader: why the indirect call?)
    movl    $mp_main, %eax
    call    *%eax
```

# Exercise 2: Waking up Application Processors

- Algorithm
  - code = vaddr(0x7000)
  - Move code at mpentry_start to 0x7000
  - For each core in CPU:
    - Set separated kernel stack
    - Let lapic to run 0x7000 on the current core
    - Wait until it changes cpu_status to started
      - In `mp_main()`

**So it initialize one core, wait, one core, wait, …**

```
// Start the non-boot (AP) processors.
static void
boot_aps(void)
{
    extern unsigned char mpentry_start[], mpentry_end[];
    void *code;
    struct CpuInfo *c;

    // Write entry code to unused memory at MPENTRY_PADDR
    code = KADDR(MPENTRY_PADDR);
    memmove(code, mpentry_start, mpentry_end - mpentry_start);

    // Boot each AP one at a time
    for (c = cpus; c < cpus + ncpu; c++) {
        if (c == cpus + cpunum())  // We've started already.
            continue;

        // Tell mpentry.S what stack to use
        mpentry_kstack = percpu_kstacks[c - cpus] + KSTKSIZE;
        // Start the CPU at mpentry_start
        lapic_startap(c->cpu_id, PADDR(code));
        // Wait for the CPU to finish some basic setup in mp_main()
        while(c->cpu_status != CPU_STARTED)
            ;
    }
}
```

# Exercise 3&4: Per-CPU Init

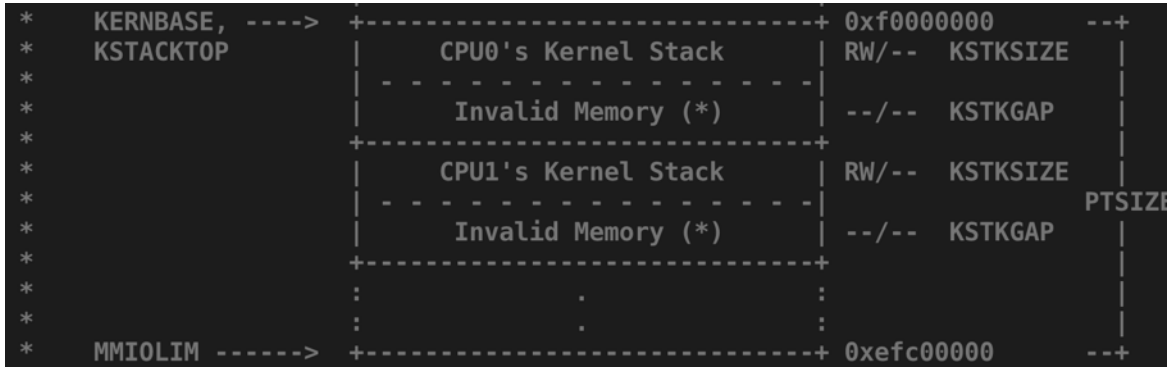
- Kernel Stack
  - Use percpu_kstacks
- TSS
  - Update GDT
- curenv
  - thiscpu->cpu_env
- Set cr3, tr, gdt, idt..

- **Per-CPU kernel stack.** Because multiple CPUs can trap into the kernel simultaneously, we need a separate kernel stack for each processor to prevent them from interfering with each other's execution. The array `percpu_kstacks[NCPU][KSTKSIZE]` reserves space for NCPU's worth of kernel stacks.

  In Lab 2, you mapped the physical memory that `bootstack` refers to as the BSP's kernel stack just below `KSTACKTOP`. Similarly, in this lab, you will map each CPU's kernel stack into this region with guard pages acting as a buffer between them. CPU 0's stack will still grow down from `KSTACKTOP`; CPU 1's stack will start `KSTKGAP` bytes below the bottom of CPU 0's stack, and so on. `inc/memlayout.h` shows the mapping layout.
- **Per-CPU TSS and TSS descriptor.** A per-CPU task state segment (TSS) is also needed in order to specify where each CPU's kernel stack lives. The TSS for CPU *i* is stored in `cpus[i].cpu_ts`, and the corresponding TSS descriptor is defined in the GDT entry `gdt[(GD_TSS0 >> 3) + i]`. The global `ts` variable defined in `kern/trap.c` will no longer be useful.
- **Per-CPU current environment pointer.** Since each CPU can run different user process simultaneously, we redefined the symbol `curenv` to refer to `cpus[cpunum()].cpu_env` (or `thiscpu->cpu_env`), which points to the environment *currently* executing on the *current* CPU (the CPU on which the code is running).
- **Per-CPU system registers.** All registers, including system registers, are private to a CPU. Therefore, instructions that initialize these registers, such as `lcr3()`, `ltr()`, `lgdt()`, `lidt()`, etc., must be executed once on each CPU. Functions `env_init_percpu()` and `trap_init_percpu()` are defined for this purpose.

# Exercise 3: Per-CPU Init, Set Kernel Stack

- inc/memlayout.h
- i-th CPU's stack
- VA: KSTACKTOP – i * (KSTKSIZE+KSTKGAP)
- PA: PADDR(&percpu_kstacks[i])




- Use boot_map_region to map those region

# Exercise 4: Per-CPU Init, trap_init_percpu

- Lab 3
  - Set values of TSS
  - Set TSS to GDT
  - Load TSS
  - Load IDT

```
// Initialize and load the per-CPU TSS and IDT
void
trap_init_percpu(void)
{
    // Setup a TSS so that we get the right stack
    // when we trap to the kernel.
    ts.ts_esp0 = KSTACKTOP;
    ts.ts_ss0 = GD_KD;
    ts.ts_iomb = sizeof(struct Taskstate);

    // Initialize the TSS slot of the gdt.
    gdt[GD_TSS0 >> 3] = SEG16(STS_T32A, (uint32_t) (&ts),
                              sizeof(struct Taskstate) - 1, 0);
    gdt[GD_TSS0 >> 3].sd_s = 0;

    // Load the TSS selector (like other segment selectors, the
    // bottom three bits are special; we leave them 0)
    ltr(GD_TSS0);

    // Load the IDT
    lidt(&idt_pd);
}
```

We need to change this to initialize each processor!!

# Exercise 4: Per-CPU Init, trap_init_percpu

- Lab 4
  - Replace ts with
    - thiscpu->cpu_ts
  - Access gdt via
    - gdt[(GD_TSS0 >> 3) + cpunum()]
  - Load TR per each CPU
    - ltr(GD_TSS0 + (cpunum() << 3))
  - Load IDT (we use the same IDT)
    - lidt(&idt_pd)

```
// Initialize and load the per-CPU TSS and IDT
void
trap_init_percpu(void)
{
    // Setup a TSS so that we get the right stack
    // when we trap to the kernel.
    ts.ts_esp0 = KSTACKTOP;
    ts.ts_ss0 = GD_KD;
    ts.ts_iomb = sizeof(struct Taskstate);

    // Initialize the TSS slot of the gdt.
    gdt[GD_TSS0 >> 3] = SEG16(STS_T32A, (uint32_t) (&ts),
                    sizeof(struct Taskstate) - 1, 0);
    gdt[GD_TSS0 >> 3].sd_s = 0;

    // Load the TSS selector (like other segment selectors, the
    // bottom three bits are special; we leave them 0)
    ltr(GD_TSS0);

    // Load the IDT
    lidt(&idt_pd);
}
```

# Exercise 5: Lock

- We will have concurrent kernel execution, and to avoid interference of having multiple execution in kernel, we use mutex (Lock/Unlock) to define the entire kernel execution as a critical section
- Add lock_kernel and unlock_kernel in these locations…

- In `i386_init()`, acquire the lock before the BSP wakes up the other CPUs.
- In `mp_main()`, acquire the lock after initializing the AP, and then call `sched_yield()` to start running environments on this AP.
- In `trap()`, acquire the lock when trapped from user mode. To determine whether a trap happened in user mode or in kernel mode, check the low bits of the `tf_cs`.
- In `env_run()`, release the lock *right before* switching to user mode. Do not do that too early or too late, otherwise you will experience races or deadlocks.

**Note**

**Exercise 5.** Apply the big kernel lock as described above, by calling `lock_kernel()` and `unlock_kernel()` at the proper locations.

# Exercise 5: Lock

- In i386_init (kern/init.c)

```
// Acquire the big kernel lock before waking up APs
// Your code here:
lock_kernel();
```

- In `i386_init()`, acquire the lock before the BSP wakes up the other CPUs.
- In `mp_main()`, acquire the lock after initializing the AP, and then call `sched_yield()` to start running environments on this AP.
- In `trap()`, acquire the lock when trapped from user mode. To determine whether a trap happened in user mode or in kernel mode, check the low bits of the `tf_cs`.
- In `env_run()`, release the lock *right before* switching to user mode. Do not do that too early or too late, otherwise you will experience races or deadlocks.

**Note**

**Exercise 5.** Apply the big kernel lock as described above, by calling `lock_kernel()` and `unlock_kernel()` at the proper locations.

# Exercise 5: Lock

- In mp_main (kern/init.c)

```
// Now that we have finished some basic setup, call sched_yield()
// to start running processes on this CPU.  But make sure that
// only one CPU can enter the scheduler at a time!
//
// Your code here:
lock_kernel();
sched_yield();
```

- In `mp_main()`, acquire the lock after initializing the AP, and then call `sched_yield()` to start running environments on this AP.
- In `trap()`, acquire the lock when trapped from user mode. To determine whether a trap happened in user mode or in kernel mode, check the low bits of the `tf_cs`.
- In `env_run()`, release the lock *right before* switching to user mode. Do not do that too early or too late, otherwise you will experience races or deadlocks.

**Note**

**Exercise 5.** Apply the big kernel lock as described above, by calling `lock_kernel()` and `unlock_kernel()` at the proper locations.

# Exercise 5: Lock

- In trap (kern/trap.c)

```
if ((tf->tf_cs & 3) == 3) {
    // Trapped from user mode.
    // Acquire the big kernel lock before doing any
    // serious kernel work.
    // LAB 4: Your code here.
    lock_kernel();
    assert(curenv);
```

- In `i386_init()`, acquire the lock before the BSP wakes up the other CPUs.
- In `mp_main()`, acquire the lock after initializing the AP, and then call `sched_yield()` to start running environments on this AP.
- In `trap()`, acquire the lock when trapped from user mode. To determine whether a trap happened in user mode or in kernel mode, check the low bits of the `tf_cs`.
- In `env_run()`, release the lock *right before* switching to user mode. Do not do that too early or too late, otherwise you will experience races or deadlocks.

**Note**

**Exercise 5.** Apply the big kernel lock as described above, by calling `lock_kernel()` and `unlock_kernel()` at the proper locations.

# Exercise 5: Lock

- In env_run (kern/env.c)
  - Insert unlock_kernel before env_pop_tf

```
// Step 2
unlock_kernel();
env_pop_tf(&e->env_tf);
```

- In `i386_init()`, acquire the lock before the BSP wakes up the other CPUs.
- In `mp_main()`, acquire the lock after initializing the AP, and then call `sched_yield()` to start running environments on this AP.
- In `trap()`, acquire the lock when trapped from user mode. To determine whether a trap happened in user mode or in kernel mode, check the low bits of the `tf_cs`.
- In `env_run()`, release the lock *right before* switching to user mode. Do not do that too early or too late, otherwise you will experience races or deadlocks.

**Note**

**Exercise 5.** Apply the big kernel lock as described above, by calling `lock_kernel()` and `unlock_kernel()` at the proper locations.

# Exercise 6: Round-Robin Scheduling

- We will use sched_yield() to enable cooperative multitasking
  - Remember, cooperative multitasking works with voluntary yield from processes…

- When yield happens,
  - We will run the next available environment (this is round-robin!)

# Exercise 6: Round-Robin Scheduling

- We will use sched_yield() to enable cooperative multitasking
  - Remember, cooperative multitasking works with voluntary yield from processes…

- When yield happens,
  - We will run the next available environment (this is round-robin!)

| | | | | | |
|---|---|---|---|---|---|
| x | x | RUNNING | x | RUNNABLE | x |

# Exercise 6: Round-Robin Scheduling

- Tips
- How to get the current envid?
  - curenv->env_id
- How to loop over the envs array?
  - envs[ENVX(curenv->env_id)]  // this is current env
  - envs[(ENVX(curenv->env_id) + i) % NENV]  // this will run a circular loop
- What if curenv == NULL?
  - Set env_id as 0…
- Call env_run if you found a runnable environment

# Exercise 6: Round-Robin Scheduling

- Also edit syscall.c (kern/syscall.c) to dispatch sched_yield()

```
case SYS_yield:
{
    sys_yield();
    return 0;
}
```

# Exercise 7: Implement syscalls For New Environment Creation

**Note**

**Exercise 7.** Implement the system calls described above in `kern/syscall.c`. You will need to use various functions in `kern/pmap.c` and `kern/env.c`, particularly `envid2env()`. For now, whenever you call `envid2env()`, pass 1 in the `checkperm` parameter. Be sure you check for any invalid system call arguments, returning `-E_INVAL` in that case. Test your JOS kernel with `user/dumbfork` and make sure it works before proceeding.

Implement 5 system calls to support new environment creation!

```
sys_exofork
sys_env_set_status
sys_page_alloc
sys_page_map
sys_page_unmap
```

# Exercise 7: Implement syscalls For New Environment Creation

- sys_exofork()
  - System call for support fork (in user space)
- What it should do?

- `sys_exofork`:

  This system call creates a new environment with an almost blank slate: nothing is mapped in the user portion of its address space, and it is not runnable. The new environment will have the same register state as the parent environment at the time of the `sys_exofork` call. In the parent, `sys_exofork` will return the `envid_t` of the newly created environment (or a negative error code if the environment allocation failed). In the child, however, it will return 0. (Since the child starts out marked as not runnable, `sys_exofork` will not actually return in the child until the parent has explicitly allowed this by marking the child runnable using....)

# Exercise 7: Implement syscalls For New Environment Creation

- Allocating a new environment
  - `Struct Env *e;`
  - `env_alloc(&e, curenv ? curenv->env_id : 0)`
    - We can set the parent env_id 0 if there is no curenv…

```
// Allocate a new environment.
// Returns envid of new environment, or < 0 on error.  Errors are:
//   -E_NO_FREE_ENV if no free environment is available.
//   -E_NO_MEM on memory exhaustion.
static envid_t
sys_exofork(void)
{
    // Create the new environment with env_alloc(), from kern/env.c.
    // It should be left as env_alloc created it, except that
    // status is set to ENV_NOT_RUNNABLE, and the register set is copied
    // from the current environment -- but tweaked so sys_exofork
    // will appear to return 0.
```

# Exercise 7: Implement syscalls For New Environment Creation

- Set new env status as ENV_NOT_RUNNABLE
  - `e->env_status = ENV_NOT_RUNNABLE`
- Copy registers
  - `e->env_tf = curenv->env_tf` // trapframe stores registers
- Set child return value as 0?
  - `e->env_tf.tf_regs.reg_eax = 0`

```
// Allocate a new environment.
// Returns envid of new environment, or < 0 on error.  Errors are:
//  -E_NO_FREE_ENV if no free environment is available.
//  -E_NO_MEM on memory exhaustion.
static envid_t
sys_exofork(void)
{
    // Create the new environment with env_alloc(), from kern/env.c.
    // It should be left as env_alloc created it, except that
    // status is set to ENV_NOT_RUNNABLE, and the register set is copied
    // from the current environment -- but tweaked so sys_exofork
    // will appear to return 0.
```

# Exercise 7: Implement syscalls For New Environment Creation

- sys_env_set_status()
  - If status is not either of ENV_RUNNABLE or ENV_NOT_RUNNABLE
    - Return –E_INVAL

```
// Set envid's env_status to status, which must be ENV_RUNNABLE
// or ENV_NOT_RUNNABLE.
//
// Returns 0 on success, < 0 on error.  Errors are:
//  -E_BAD_ENV if environment envid doesn't currently exist,
//      or the caller doesn't have permission to change envid.
//  -E_INVAL if status is not a valid status for an environment.
static int
sys_env_set_status(envid_t envid, int status)
{
    // Hint: Use the 'envid2env' function from kern/env.c to translate an
    // envid to a struct Env.
    // You should set envid2env's third argument to 1, which will
    // check whether the current environment has permission to set
    // envid's status.
```

# Exercise 7: Implement syscalls For New Environment Creation

- sys_env_set_status()
  - You may use envid2env(envid, struct Env **store, checkperm) to check if envid is valid or not
  - Read the function envid2env; this checks if:

```
e = &envs[ENVX(envid)];
if (e->env_status == ENV_FREE || e->env_id != envid) {
    *env_store = 0;
    return -E_BAD_ENV;
}

// Check that the calling environment has legitimate permission
// to manipulate the specified environment.
// If checkperm is set, the specified environment
// must be either the current environment
// or an immediate child of the current environment.
if (checkperm && e != curenv && e->env_parent_id != curenv->env_id) {
    *env_store = 0;
    return -E_BAD_ENV;
}
```

# Exercise 7: Implement syscalls For New Environment Creation

- sys_page_alloc(envid, void *va, int perm)
  - Map a page at the virtual address `va` with permission `perm`
  - Use `page_alloc(ALLOC_ZERO)` to get a free physical page as
    - `struct PageInfo *pp`
  - Use `page_insert(e->env_pgdir, pp, va, perm)` to map it!

```
static int
sys_page_alloc(envid_t envid, void *va, int perm)
{
    // Hint: This function is a wrapper around page_alloc() and
    //   page_insert() from kern/pmap.c.
    //   Most of the new code you write should be to check the
    //   parameters for correctness.
    //   If page_insert() fails, remember to free the page you
    //   allocated!
```

# Exercise 7: Implement syscalls For New Environment Creation

- sys_page_alloc(envid, void *va, int perm)
- Tips
  - How to check perm only includes bits in PTE_SYSCALL?

```
if ((perm & 0xfff) & (~PTE_SYSCALL))
    return -E_INVAL;
```

  - Why?
    - perm&0xfff = get lower 12 bit permission flag
    - ~PTE_SYSCALL = negation of PTE_SYSCALL; bits are 1 except for allowed flags
    - If any of bit is 1, then perm has a flag other than sets in PTE_SYSCALL

# Exercise 7: Implement syscalls For New Environment Creation

- sys_page_alloc(envid, void *va, int perm)
- Tips
  - How to check va is page aligned?

```
if ((uintptr_t)va & 0xfff)
    return -E_INVAL;
```

  - Why?
    - va&0xfff = offset
    - It checks if offset == 0 or not…

# Exercise 7: Implement syscalls For New Environment Creation

- sys_page_alloc(envid, void *va, int perm)
- Tips
  - How to check envid is valid?
  - Use envid2env() with checkperm = 1

# Exercise 7: Implement syscalls For New Environment Creation

- sys_page_map(srcenvid, srcva, dstenvid, dstva, perm)
  - Map the physical page that backs `srcva` to `dstva`

- You have to implement many checks, use the tricks with &0xfff and ~
  - Also use `envid2env` to check `srcenvid` and `dstenvid`

- How to map a page from src to dst?
  - Use `page_lookup` to get the `struct PageInfo *pp` of `srcva`
    - This is a physical page
  - Use `page_insert` to map that physical page to `dstva`

Also check the permission in the pte of srcva if it is allowed for write or not (PTE_W)

# Exercise 7: Implement syscalls For New Environment Creation

- sys_page_unmap(envid, va)
  - This unmaps the virtual address at `va`

- Check va and envid using similar tricks that we've done for other syscalls

- Use `page_remove` to unmap the mapping

# Part-A Result

```
dumbfork: OK (1.0s)
Part A score: 5/5
```

- If you are getting any assert error from pmap.c
  - Please check if your implementation for mmio_map_region and page_init for excluding MPENTRY_PADDR
- If you are getting any panic from user/kernel
  - Check if you edited kern/syscall.c to dispatch each system calls in syscall()
    - Fill the switch-case statement!
  - Check if you returned correct values to user environment
    - Check syscall return
    - Check if you set the trapframe’s eax to 0 for sys_exofork