MIT 6.828: Operating System Engineering

Lab Report
Lab 4: Preemptive Multitasking

CSNLP
csnlp16@126.com
`https://csnlp.github.io/`

Contents

```
1     struct list {
2       int data;
3       struct list *next;
4     };
5
6     struct list *list = 0;
7
8     void
9     insert(int data)
10    {
11      struct list *l;
12
13      l = malloc(sizeof *l);
14      l->data = data;
15      l->next = list;
16      list = l;
17    }
```

Figure 1: An example of race condition

1 Introduction

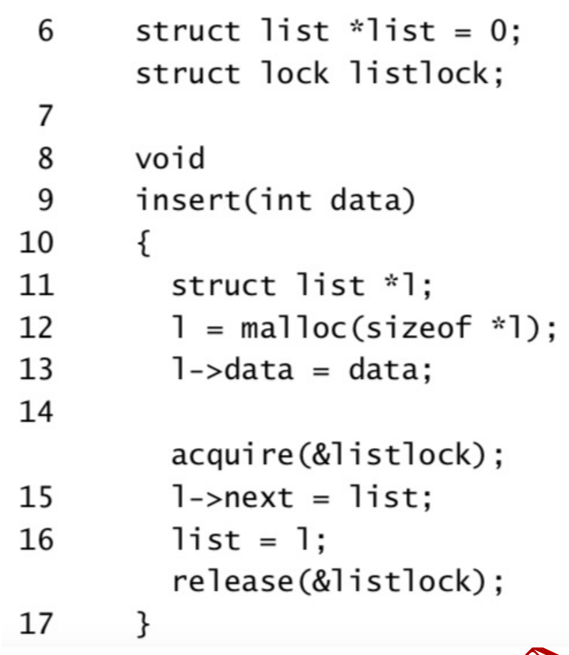
1.1 Locking

A lock provides mutual exclusion, ensuring that only one CPU at a time can hold the lock. If a lock is associated with each shared data item. When using a given item, the code always holds the associated lock. In this way, we can be sure that the item is used from only one CPU at a time.

Give an example to show a *race condition*:

There will be a race between two CPU in line 16 since they both use the data item list.

To avoid race, the common way is to use a lock.



```
struct list *list = 0;
struct lock listlock;

void
insert(int data)
{
  struct list *l;
  l = malloc(sizeof *l);
  l->data = data;

  acquire(&listlock);
  l->next = list;
  list = l;
  release(&listlock);
}
```

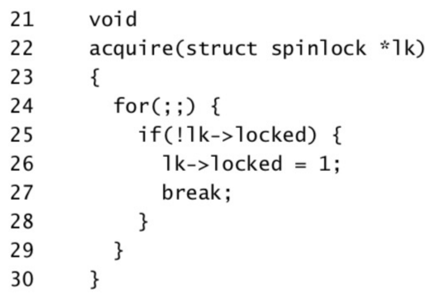
Figure 2: Avoid race condition with a lock

1.2 The Lock Implementation in xv6

There are two types of locks in xv6: spin-lock and sleep-lock.

1.2.1 Spin-lock in xv6

The important field in spin-lock is locked, a word that is zero when the lock is available and non-zero when it is held. To acquire a lock, xv6 execute the following code:



```
21    void
22    acquire(struct spinlock *lk)
23    {
24      for(;;) {
25        if(!lk->locked) {
26          lk->locked = 1;
27          break;
28        }
29      }
30    }
```

Figure 3: Spin-lock: race in its code

This code cannot ensure mutual exclusion. When two CPU simultaneously reach line 25, they both grab the lock by executing line 26. In other words, they both acquire the lock. So, this implementation of acquire has its own race condition. To solve this problem, **line 25 and line 26 must execute in one *atomic* step.**

1.3 How To Use Locks

A hard part about using locks is deciding how many locks to use and which data and invariants each lock protects. There are a few basic principles:

1. Any time a variable can be **written** by one CPU at the same time that another CPU can **read or write** to it, a lock should be introduced to keep the two operations from overlapping.

2. Keep in mind that locks are used to protect **invariants**: if an invariant involves multiple memory locations, typically all of them need to be protected by a single lock to ensure the invariant is maintained.

1.4 Deadlock and Lock Ordering

1.4.1 How Deadlock Happens?

Let's say two code paths need lock A and lock B.

1. The first code path acquires locks in the order {A then B}.

2. The second code path acquires locks in the order {B then A}.

3. If thread T1 executes code path 1 while T2 executes code path 2 concurrently. T1 has acquired lock A while T2 has acquired lock B. They both face the situation that the other thread holds the needed lock and won't release it until its acquire returns.

The above situation will cause deadlock.

1.5 Sleep Locks

1.6 Limitation of Locks

1.7 New source files in LAB 4

1. kern/cpu.h: Kernel-private definitions for multiprocessor support.

2. kern/mpconfig.c: Code to read the multiprocessor configuration.

3. kern/lapic: Kernel code driving the local APIC unit in each processor.

4. kern/mpentry.S: Assembly-language entry code for non-boot CPUs.

5. kern/spinlock.h: Kernel-private definitions for spin locks, including the big kernel lock.

6. kern/spinlock.c: Kernel code implementing spin locks.

7. kern/sched.c: Code skeleton of the scheduler that you are about to implement.

2 Part A: Multiprocessor Support and Cooperative Multitasking

2.1 Multiprocessor Support

In this part, we expect that JOS can support *symmetric multiprocessing* (SMP), a multiprocessor model in which all CPUs have equivalent access to system resources such as memory and I/O buses.

During the boot process, these functionally identical CPUs can be classified into two types:

1. Bootstrap processor (BSP): responsible for initializing the system and for booting the OS.

2. Application processors (APs): be activated by the BSP only after the OS is up and running.

In an SMP system, each CPU has an accompanying local APIC (LAPIC) unit. The LAPIC units are responsible for:

1. Delivering interrupts throughout the system.

2. Providing its connected CPU with a unique identifier.

Which basic functionality of the LAPIC unit we will use:

1. APIC ID: to tell which CPU on which our code is current running, see in **cpunum()**.

2. Sending the STARTUP interprocessor interrupt(IPI) from the BSP to the APs to bring up other CPUs, see in **lapic_startap**

3. The LAPIC's built-in timer, see in **apic_init()**.

2.1.1 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.*

Let's firstly check **lapic_init()** in **kern/lapic.c**

```
void
lapic_init(void)
{
  if (!lapicaddr)
    return;

  // lapicaddr is the physical address of the LAPIC's 4K MMIO
  // region.  Map it in to virtual memory so we can access it.
  lapic = mmio_map_region(lapicaddr, 4096);

  // Enable local APIC; set spurious interrupt vector.
  lapicw(SVR, ENABLE | (IRQ_OFFSET + IRQ_SPURIOUS));

  // The timer repeatedly counts down at bus frequency
  // from lapic[TICR] and then issues an interrupt.
  // If we cared more about precise timekeeping,
  // TICR would be calibrated using an external time source.
  lapicw(TDCR, X1);
  lapicw(TIMER, PERIODIC | (IRQ_OFFSET + IRQ_TIMER));
  lapicw(TICR, 10000000);

  // Leave LINT0 of the BSP enabled so that it can get
  // interrupts from the 8259A chip.
  //
  // According to Intel MP Specification, the BIOS should initialize
  // BSP's local APIC in Virtual Wire Mode, in which 8259A's
  // INTR is virtually connected to BSP's LINTIN0. In this mode,
  // we do not need to program the IOAPIC.
  if (thiscpu != bootcpu)
    lapicw(LINT0, MASKED);

  // Disable NMI (LINT1) on all CPUs
  lapicw(LINT1, MASKED);

  // Disable performance counter overflow interrupts
  // on machines that provide that interrupt entry.
  if (((lapic[VER]>>16) & 0xFF) >= 4)
    lapicw(PCINT, MASKED);

  // Map error interrupt to IRQ_ERROR.
  lapicw(ERROR, IRQ_OFFSET + IRQ_ERROR);

  // Clear error status register (requires back-to-back writes).
  lapicw(ESR, 0);
  lapicw(ESR, 0);

  // Ack any outstanding interrupts.
  lapicw(EOI, 0);

  // Send an Init Level De-Assert to synchronize arbitration ID's.
  lapicw(ICRHI, 0);
  lapicw(ICRLO, BCAST | INIT | LEVEL);
  while(lapic[ICRLO] & DELIVS)
    ;

  // Enable interrupts on the APIC (but not on the processor).
  lapicw(TPR, 0);
}
```

It calls mmio_map_region()

```
        // lapicaddr is the physical address of the LAPIC's 4K MMIO
        // region.  Map it in to virtual memory so we can access it.
        lapic = mmio_map_region(lapicaddr, 4096);
```

Let's see the implementation of mmio_map_region:

```
void *
mmio_map_region(physaddr_t pa, size_t size)
{
        static uintptr_t base = MMIOBASE;
        void *ret = (void *)base;
        size = ROUNDUP(size, PGSIZE);
        if(base + size > MMIOLIM || base + size < base)
                panic("mmio_map_region fail: overflow");
        boot_map_region(kern_pgdir, base, size, pa, PTE_W | PTE_PCD | PTE_PWT);
        base += size;
        return ret;
}
```

2.1.2 Application Processor Bootstrap

1. **Information Collection**: Before booting up APs, the BSP should first collect information about the multiprocessor system, such as the total number of CPUs, their APIC IDs and the MMIO address of the LAPIC unit. The mp_init() function in kern/mpconfig.c retrieves this information by reading the MP configuration table that resides in the BIOS's region of memory. Let's see the mp_init() in kern/mpconfig.c

```
void
mp_init(void)
{
  struct mp *mp;
  struct mpconf *conf;
  struct mpproc *proc;
  uint8_t *p;
  unsigned int i;

  bootcpu = &cpus[0];
  if ((conf = mpconfig(&mp)) == 0)
    return;
  ismp = 1;
  lapicaddr = conf->lapicaddr;

  for (p = conf->entries, i = 0; i < conf->entry; i++) {
    switch (*p) {
    case MPPROC:
      proc = (struct mpproc *)p;
      if (proc->flags & MPPROC_BOOT)
        bootcpu = &cpus[ncpu];
      if (ncpu < NCPU) {
        cpus[ncpu].cpu_id = ncpu;
        ncpu++;
      } else {
        cprintf("SMP: too many CPUs, CPU %d disabled\n",
          proc->apicid);
      }
      p += sizeof(struct mpproc);
      continue;
    case MPBUS:
    case MPIOAPIC:
    case MPIOINTR:
    case MPLINTR:
      p += 8;
      continue;
    default:
      cprintf("mpinit: unknown config type %x\n", *p);
      ismp = 0;
      i = conf->entry;
    }
  }

  bootcpu->cpu_status = CPU_STARTED;
  if (!ismp) {
    // Didn't like what we found; fall back to no MP.
    ncpu = 1;
    lapicaddr = 0;
    cprintf("SMP: configuration not found, SMP disabled\n");
    return;
  }
  cprintf("SMP: CPU %d found %d CPU(s)\n", bootcpu->cpu_id,  ncpu);

  if (mp->imcrp) {
    // [MP 3.2.6.1] If the hardware implements PIC mode,
    // switch to getting interrupts from the LAPIC.
    cprintf("SMP: Setting IMCR to switch from PIC mode to symmetric I/O mode\n")
    ↪ ;
    outb(0x22, 0x70);   // Select IMCR
    outb(0x23, inb(0x23) | 1);  // Mask external interrupts.
  }
}
```

2. **AP bootstrap process**: The boot_aps() function (in kern/init.c) drives the AP bootstrap process. APs start in real mode, much like how the bootloader started in boot/boot.S, so boot_aps() copies the AP entry code (kern/mpentry.S) to a memory location that is addressable in the real mode. Unlike with the bootloader, we have some control over where the AP will start executing code; we copy the entry code to 0x7000 (MPENTRY_PADDR), but any unused, page-aligned physical address below 640KB would work.

3. **AP's activation**: boot_aps() activates APs one after another, by sending STARTUP IPIs to the LAPIC unit of the corresponding AP, along with an initial CS:IP address at which the AP should start running its entry code (MPENTRY_PADDR in our case). The entry code in kern/mpentry.S is quite similar to that of boot/-boot.S. After some brief setup, it puts the AP into protected mode with paging enabled, and then calls the C setup routine mp_main() (also in kern/init.c). boot_aps() waits for the AP to signal a CPU_STARTED flag in cpu_status field of its struct CpuInfo before going on to wake up the next one.

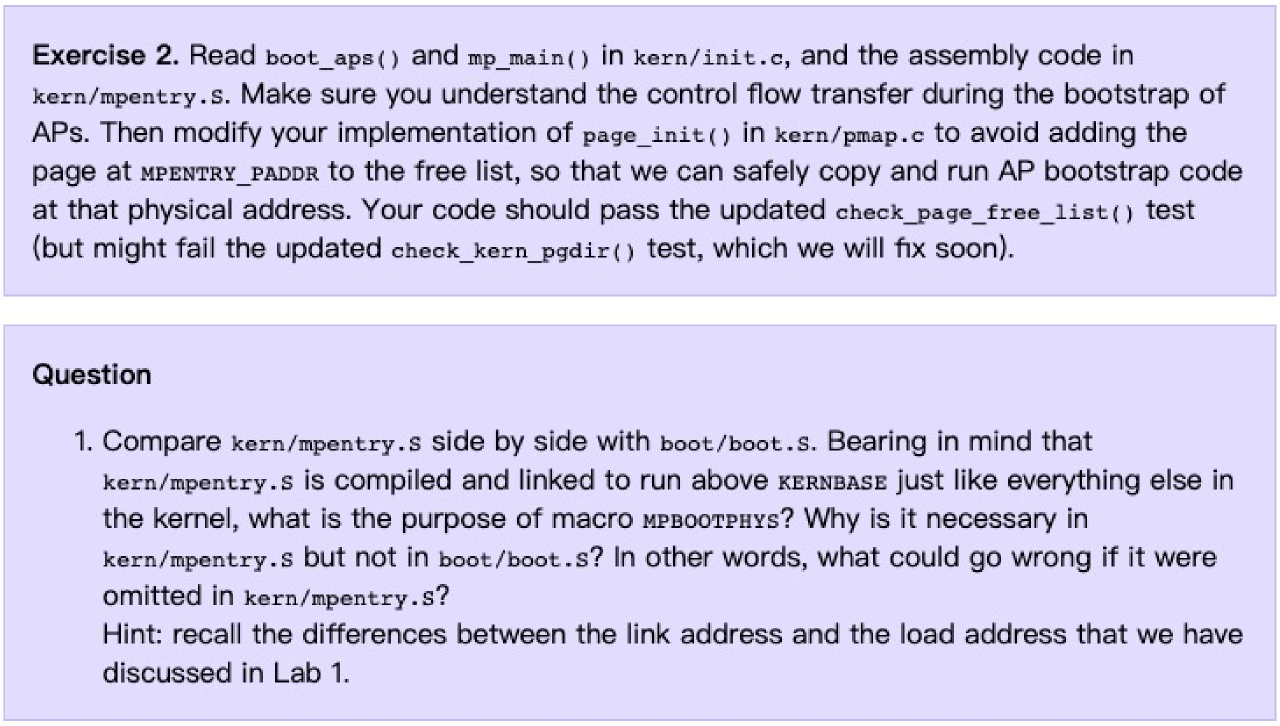
```
// Start the non-boot (AP) processors.
static void
boot_aps(void)
{
        extern unsigned char mpentry_start[], mpentry_end[];
        void *code;
        struct CpuInfo *c;

        // 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)
                        ;
        }
}
```

The boot_aps() is about:

- mpentry_start and mpentry_end is the start and end of the virtual address of the assembly code: kern/mpentry.S in memory.
- memmove is to move the assembly code kern/mpentry.S to physical address: MPENTRY_PADDR.
- mpentry_kstack is stack of the boot loader of the selected AP.
- lapic_startap: start the CPU at the MPENTRY_PADDR, notice that at this time MPENTRY_PADDR is already the start address of kern/mpentry.S.

2.1.3 Exercise 2



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).

Question

1. Compare `kern/mpentry.S` side by side with `boot/boot.S`. Bearing in mind that `kern/mpentry.S` is compiled and linked to run above `KERNBASE` just like everything else in the kernel, what is the purpose of macro `MPBOOTPHYS`? Why is it necessary in `kern/mpentry.S` but not in `boot/boot.S`? In other words, what could go wrong if it were omitted in `kern/mpentry.S`?
   Hint: recall the differences between the link address and the load address that we have discussed in Lab 1.

Figure 4: Exercise 2

```c
// Setup code for APs
void
mp_main(void)
{
	// We are in high EIP now, safe to switch to kern_pgdir
	lcr3(PADDR(kern_pgdir));
	cprintf("SMP: CPU %d starting\n", cpunum());

	lapic_init();
	env_init_percpu();
	trap_init_percpu();
	xchg(&thiscpu->cpu_status, CPU_STARTED); // tell boot_aps() we're up

	// 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:

	// Remove this after you finish Exercise 6
	for (;;);
}
```

Listing 1: map_main in kern/init.c

2.1.3.1 The Control Flow

The control flow transfer during the bootstrap of APs can be summarized as follows:

1. **mem_init()**: Lab 2 memory management initialization.
2. **env_init()** and **trap_init()**: Lab 3 user environment initialization.
3. **mp_init()** and **lapic_init()**: Lab 4 multiprocessor initialization.
4. **pic_init()**: multitasking initialization.
5. **lock_kernel()**: acquire the kernel before waking up APs.
6. **boot_aps()**: starting non-boot CPUs.

2.1.4 Per-CPU State and Initialization

When writing a multiprocessor OS, it's important to distinguish between:

- per-CPU state that is private to each processor.
- Global state that the whole system shares.

Here is the per-CPU state that we should be aware of:

- **per-CPU kernel stack:** Since 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. See the actual structure in **inc/memlayout.h**

```c
#ifndef JOS_INC_MEMLAYOUT_H
#define JOS_INC_MEMLAYOUT_H

#ifndef __ASSEMBLER__
#include <inc/types.h>
#include <inc/mmu.h>
#endif /* not __ASSEMBLER__ */

/*
 * This file contains definitions for memory management in our OS,
 * which are relevant to both the kernel and user-mode software.
 */

// Global descriptor numbers
#define GD_KT     0x08     // kernel text
#define GD_KD     0x10     // kernel data
#define GD_UT     0x18     // user text
#define GD_UD     0x20     // user data
#define GD_TSS0   0x28     // Task segment selector for CPU 0

/*
 * Virtual memory map:                                Permissions
 *                                                    kernel/user
 *
 *    4 Gig -------->  +------------------------------+
 *                     |                              | RW/--
 *                     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 *                     :              .               :
 *                     :              .               :
 *                     :              .               :
 *                     |~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~| RW/--
 *                     |                              | RW/--
 *                     |   Remapped Physical Memory   | RW/--
 *                     |                              | RW/--
 *    KERNBASE, ---->  +------------------------------+ 0xf0000000      --+
 *    KSTACKTOP        |     CPU0's Kernel Stack      | RW/--  KSTKSIZE   |
 *                     | - - - - - - - - - - - - - - -|                   |
 *                     |      Invalid Memory (*)      | --/--  KSTKGAP    |
 *                     +------------------------------+                   |
 *                     |     CPU1's Kernel Stack      | RW/--  KSTKSIZE   |
 *                     | - - - - - - - - - - - - - - -|                 PTSIZE
 *                     |      Invalid Memory (*)      | --/--  KSTKGAP    |
 *                     +------------------------------+                   |
 *                     :              .               :                   |
 *                     :              .               :                   |
 *    MMIOLIM ------>  +------------------------------+ 0xefc00000      --+
 *                     |       Memory-mapped I/O      | RW/--  PTSIZE
 * ULIM, MMIOBASE -->  +------------------------------+ 0xef800000
 *                     |  Cur. Page Table (User R-)   | R-/R-  PTSIZE
 *    UVPT      ---->  +------------------------------+ 0xef400000
 *                     |          RO PAGES            | R-/R-  PTSIZE
 *    UPAGES    ---->  +------------------------------+ 0xef000000
 *                     |           RO ENVS            | R-/R-  PTSIZE
 * UTOP,UENVS ------>  +------------------------------+ 0xeec00000
 * UXSTACKTOP -/       |     User Exception Stack     | RW/RW  PGSIZE
 *                     +------------------------------+ 0xeebff000
 *                     |       Empty Memory (*)       | --/--  PGSIZE
 *    USTACKTOP  --->  +------------------------------+ 0xeebfe000
```

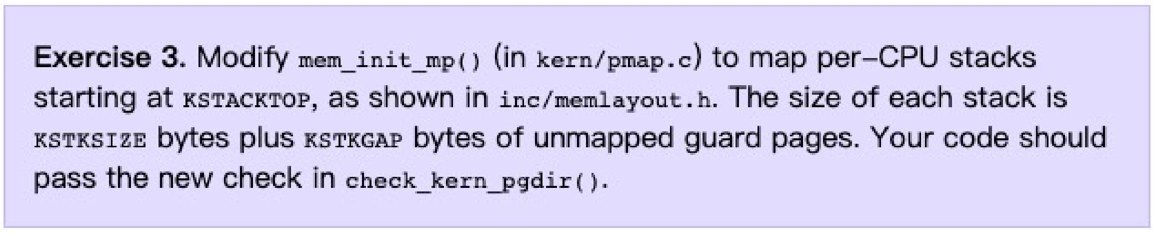
```
 *                     |      Normal User Stack       | RW/RW  PGSIZE
 *                     +------------------------------+ 0xeebfd000
 *                     |                              |
 *                     |                              |
 *                     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 *                     .                              .
 *                     .                              .
 *                     .                              .
 *                     |~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|
 *                     |     Program Data & Heap      |
 *    UTEXT -------->  +------------------------------+ 0x00800000
 *    PFTEMP ------->  |       Empty Memory (*)       |        PTSIZE
 *                     |                              |
 *    UTEMP -------->  +------------------------------+ 0x00400000      --+
 *                     |       Empty Memory (*)       |                   |
 *                     | - - - - - - - - - - - - - - -|                   |
 *                     |  User STAB Data (optional)   |                 PTSIZE
 *    USTABDATA ---->  +------------------------------+ 0x00200000        |
 *                     |       Empty Memory (*)       |                   |
 *    0 ------------>  +------------------------------+                 --+
 *
 * (*) Note: The kernel ensures that "Invalid Memory" is *never* mapped.
 *     "Empty Memory" is normally unmapped, but user programs may map pages
 *     there if desired.  JOS user programs map pages temporarily at UTEMP.
 */


// All physical memory mapped at this address
#define KERNBASE  0xF0000000

// At IOPHYSMEM (640K) there is a 384K hole for I/O.  From the kernel,
// IOPHYSMEM can be addressed at KERNBASE + IOPHYSMEM.  The hole ends
// at physical address EXTPHYSMEM.
#define IOPHYSMEM 0x0A0000
#define EXTPHYSMEM  0x100000

// Kernel stack.
#define KSTACKTOP KERNBASE
#define KSTKSIZE  (8*PGSIZE)      // size of a kernel stack
#define KSTKGAP   (8*PGSIZE)      // size of a kernel stack guard

// Memory-mapped IO.
#define MMIOLIM   (KSTACKTOP - PTSIZE)
#define MMIOBASE  (MMIOLIM - PTSIZE)

#define ULIM    (MMIOBASE)

/*
 * User read-only mappings! Anything below here til UTOP are readonly to user.
 * They are global pages mapped in at env allocation time.
 */

// User read-only virtual page table (see 'uvpt' below)
#define UVPT    (ULIM - PTSIZE)
// Read-only copies of the Page structures
#define UPAGES    (UVPT - PTSIZE)
// Read-only copies of the global env structures
#define UENVS   (UPAGES - PTSIZE)
```

```c

/*
 * Top of user VM. User can manipulate VA from UTOP-1 and down!
 */

// Top of user-accessible VM
#define UTOP    UENVS
// Top of one-page user exception stack
#define UXSTACKTOP  UTOP
// Next page left invalid to guard against exception stack overflow; then:
// Top of normal user stack
#define USTACKTOP (UTOP - 2*PGSIZE)

// Where user programs generally begin
#define UTEXT   (2*PTSIZE)

// Used for temporary page mappings.  Typed 'void*' for convenience
#define UTEMP   ((void*) PTSIZE)
// Used for temporary page mappings for the user page-fault handler
// (should not conflict with other temporary page mappings)
#define PFTEMP    (UTEMP + PTSIZE - PGSIZE)
// The location of the user-level STABS data structure
#define USTABDATA (PTSIZE / 2)

// Physical address of startup code for non-boot CPUs (APs)
#define MPENTRY_PADDR 0x7000

#ifndef __ASSEMBLER__

typedef uint32_t pte_t;
typedef uint32_t pde_t;

#if JOS_USER
/*
 * The page directory entry corresponding to the virtual address range
 * [UVPT, UVPT + PTSIZE) points to the page directory itself.  Thus, the page
 * directory is treated as a page table as well as a page directory.
 *
 * One result of treating the page directory as a page table is that all PTEs
 * can be accessed through a "virtual page table" at virtual address UVPT (to
 * which uvpt is set in lib/entry.S).  The PTE for page number N is stored in
 * uvpt[N].  (It's worth drawing a diagram of this!)
 *
 * A second consequence is that the contents of the current page directory
 * will always be available at virtual address (UVPT + (UVPT >> PGSHIFT)), to
 * which uvpd is set in lib/entry.S.
 */
extern volatile pte_t uvpt[];     // VA of "virtual page table"
extern volatile pde_t uvpd[];     // VA of current page directory
#endif

/*
 * Page descriptor structures, mapped at UPAGES.
 * Read/write to the kernel, read-only to user programs.
 *
 * Each struct PageInfo stores metadata for one physical page.
 * Is it NOT the physical page itself, but there is a one-to-one
 * correspondence between physical pages and struct PageInfo's.
 * You can map a struct PageInfo * to the corresponding physical address
 * with page2pa() in kern/pmap.h.
 */
struct PageInfo {
  // Next page on the free list.
  struct PageInfo *pp_link;

  // pp_ref is the count of pointers (usually in page table entries)
  // to this page, for pages allocated using page_alloc.
  // Pages allocated at boot time using pmap.c's
  // boot_alloc do not have valid reference count fields.

  uint16_t pp_ref;
};

#endif /* !__ASSEMBLER__ */
#endif /* !JOS_INC_MEMLAYOUT_H */
```

Listing 2: inc/memlayout.h

- **Per-CPU TSS and TSS descriptor:** A per-CPU task state segment (TSS) is needed to specify each CPU's kernel lives. The TSS for CPU i is stored in cpu[i].cpu_ts, and the corresponding TSS descriptor is defined in the GDT entry gdt[(GD_TSS0 ¿¿ 3) + i].
- **Per-CPU current environment pointer:** Because each CPU can run different user process simultaneously, we redefine they symbol **curenv** to refer to cpus[cpunum()].cpu_env (or thiscpu-¿cpu_env), which points to the environment currently executing on the current CPU. **Per-CPU system registers:** All registers are private to CPU. Therefore, instructions initializing these registers (such as lcr3(), ltr(), lgdt(), lidt()) must be executed once on each CPU. Functions env_init_percpu() and trap_init_percpu() are defined for this purpose.

2.1.5 Exercise 3



Exercise 3. Modify `mem_init_mp()` (in `kern/pmap.c`) to map per-CPU stacks starting at `KSTACKTOP`, as shown in `inc/memlayout.h`. The size of each stack is `KSTKSIZE` bytes plus `KSTKGAP` bytes of unmapped guard pages. Your code should pass the new check in `check_kern_pgdir()`.

Figure 5: Exercise 3

Let's firstly see hints in mem_init_mp():

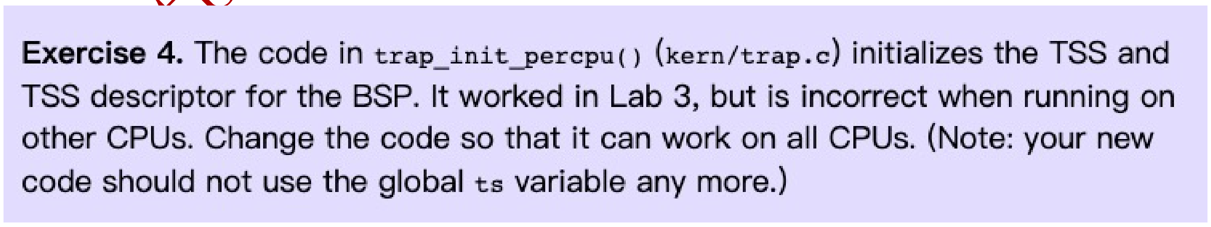
```
// Modify mappings in kern_pgdir to support SMP
//   - Map the per-CPU stacks in the region [KSTACKTOP-PTSIZE, KSTACKTOP)
//
static void
mem_init_mp(void)
```

```c
{
        // Map per-CPU stacks starting at KSTACKTOP, for up to 'NCPU' CPUs.
        //
        // For CPU i, use the physical memory that 'percpu_kstacks[i]' refers
        // to as its kernel stack. CPU i's kernel stack grows down from virtual
        // address kstacktop_i = KSTACKTOP - i * (KSTKSIZE + KSTKGAP), and is
        // divided into two pieces, just like the single stack you set up in
        // mem_init:
        //     * [kstacktop_i - KSTKSIZE, kstacktop_i)
        //          -- backed by physical memory
        //     * [kstacktop_i - (KSTKSIZE + KSTKGAP), kstacktop_i - KSTKSIZE)
        //          -- not backed; so if the kernel overflows its stack,
        //             it will fault rather than overwrite another CPU's stack.
        //             Known as a "guard page".
        //     Permissions: kernel RW, user NONE
        //
        // LAB 4: Your code here:

}
```

See its implementation:

```c
static void
mem_init_mp(void)
{
        uintptr_t kstacktop_i;
        int i;
        for(i=0; i < NCPU; i++) {
                kstacktop_i = KSTACKTOP - i * (KSTKSIZE + KSTKGAP);
                boot_map_region(kern_pgdir,
                                kstacktop_i - KSTKSIZE,
                                ROUNDUP(KSTKSIZE, PGSIZE),
                                PADDR(&percpu_kstacks[i]),
                                PTE_W | PTE_P);
        }

}
```

2.1.6 Exercise 4



Exercise 4. The code in `trap_init_percpu()` (`kern/trap.c`) initializes the TSS and TSS descriptor for the BSP. It worked in Lab 3, but is incorrect when running on other CPUs. Change the code so that it can work on all CPUs. (Note: your new code should not use the global `ts` variable any more.)

Figure 6: Exercise 4

Firstly, see its hints:

```c
// Initialize and load the per-CPU TSS and IDT
void
trap_init_percpu(void)
```

```
{
  // The example code here sets up the Task State Segment (TSS) and
  // the TSS descriptor for CPU 0. But it is incorrect if we are
  // running on other CPUs because each CPU has its own kernel stack.
  // Fix the code so that it works for all CPUs.
  //
  // Hints:
  //   - The macro "thiscpu" always refers to the current CPU's
  //     struct CpuInfo;
  //   - The ID of the current CPU is given by cpunum() or
  //     thiscpu->cpu_id;
  //   - Use "thiscpu->cpu_ts" as the TSS for the current CPU,
  //     rather than the global "ts" variable;
  //   - Use gdt[(GD_TSS0 >> 3) + i] for CPU i's TSS descriptor;
  //   - You mapped the per-CPU kernel stacks in mem_init_mp()
  //   - Initialize cpu_ts.ts_iomb to prevent unauthorized environments
  //     from doing IO (0 is not the correct value!)
  //
  // ltr sets a 'busy' flag in the TSS selector, so if you
  // accidentally load the same TSS on more than one CPU, you'll
  // get a triple fault.  If you set up an individual CPU's TSS
  // wrong, you may not get a fault until you try to return from
  // user space on that CPU.
  //
  // LAB 4: Your code here:

  // 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);
}
```

See the SMP version:

```
void
trap_init_percpu(void)
{
        thiscpu->cpu_ts.ts_esp0 = KSTACKTOP - cpunum() * (KSTKGAP + KSTKSIZE);
        thiscpu->cpu_ts.ts_ss0 = GD_KD;

        // Initialize the TSS slot of the gdt.
        gdt[(GD_TSS0 >> 3) + cpunum()] = SEG16(STS_T32A,
                                        (uint32_t)(&thiscpu->cpu_ts),
  ↪                              sizeof(struct Taskstate)-1,
                                        0);
        gdt[(GD_TSS0 >> 3) + cpunum()].sd_s = 0;
        // Load the TSS selector (like other segment selectors, the
        // bottom three bits are special; we leave them 0)
        ltr(GD_TSS0 + sizeof(struct Segdesc) * cpunum());

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

It JUST RUN AGAIN AND AGAIN!!!!!

2.1.7 Locking

Before letting the AP get any further, we need to firstly address race conditions when multiple CPUs run kernel code simultaneously. The simplest way to achieve this is to use a *big kernel lock*. The big kernel lock is a single lock that is held whenever an environment enters **kernel mode**, and released when the environment returns to **user mode**. In this model:

1. Environments in user mode can run concurrently on any available CPUs.
2. No more that one environment can run in kernel mode. Any other environments trying to enter kernel mode are forced to wait.

This kernel lock is declared in **kern/spinlock.h**, namely **kernel_lock**. It also provides **lock_kernel()** and **unlock_kernel()** to acquire and release the lock.

This kernel lock can be applied in the following locations:

1. **i386_init()**: acquire the lock before the BSP wakes up the other CPUs.
2. **mp_main()**: acquire the lock after initializing the AP. Then, call **sched_yield()** to start running environments on this AP.
3. **trap()**: acquire the lock when trapped from the user mode. To determine whether the trap is from the user mode or kernel mode, check the low bits of the **tf_cs**.
4. **env_run()**: release the lock right before switching to user mode.

2.1.8 Exercise 5

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

1. **i386_init()** in **kern/init.c**

```

void
i386_init(void)
{
  // Initialize the console.
  // Can't call cprintf until after we do this!
  cons_init();

  cprintf("6828 decimal is %o octal!\n", 6828);

  // Lab 2 memory management initialization functions
  mem_init();

  // Lab 3 user environment initialization functions
  env_init();
  trap_init();

  // Lab 4 multiprocessor initialization functions
  mp_init();
  lapic_init();

  // Lab 4 multitasking initialization functions
  pic_init();

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

  // Starting non-boot CPUs
  boot_aps();

#if defined(TEST)
  // Don't touch -- used by grading script!
  ENV_CREATE(TEST, ENV_TYPE_USER);
#else
  // Touch all you want.
  ENV_CREATE(user_primes, ENV_TYPE_USER);
#endif // TEST*

  // Schedule and run the first user environment!
  sched_yield();
}
```

2. mp_main() in kern/init.c

```
// Setup code for APs
void
mp_main(void)
{
        // We are in high EIP now, safe to switch to kern_pgdir
        lcr3(PADDR(kern_pgdir));
        cprintf("SMP: CPU %d starting\n", cpunum());

        lapic_init();
        env_init_percpu();
        trap_init_percpu();
        xchg(&thiscpu->cpu_status, CPU_STARTED); // tell boot_aps() we're up

        // 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();
        // Remove this after you finish Exercise 6
        for (;;);
}
```

3. trap() in kern/trap.c

```c
void
trap(struct Trapframe *tf)
{
  // The environment may have set DF and some versions
  // of GCC rely on DF being clear
  asm volatile("cld" ::: "cc");

  // Halt the CPU if some other CPU has called panic()
  extern char *panicstr;
  if (panicstr)
    asm volatile("hlt");

  // Re-acqurie the big kernel lock if we were halted in
  // sched_yield()
  if (xchg(&thiscpu->cpu_status, CPU_STARTED) == CPU_HALTED)
    lock_kernel();
  // Check that interrupts are disabled.  If this assertion
  // fails, DO NOT be tempted to fix it by inserting a "cli" in
  // the interrupt path.
  assert(!(read_eflags() & FL_IF));

  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);

    // Garbage collect if current enviroment is a zombie
    if (curenv->env_status == ENV_DYING) {
      env_free(curenv);
      curenv = NULL;
      sched_yield();
    }

    // Copy trap frame (which is currently on the stack)
    // into 'curenv->env_tf', so that running the environment
    // will restart at the trap point.
    curenv->env_tf = *tf;
    // The trapframe on the stack should be ignored from here on.
    tf = &curenv->env_tf;
  }

  // Record that tf is the last real trapframe so
  // print_trapframe can print some additional information.
  last_tf = tf;

  // Dispatch based on what type of trap occurred
  trap_dispatch(tf);

  // If we made it to this point, then no other environment was
  // scheduled, so we should return to the current environment
  // if doing so makes sense.
  if (curenv && curenv->env_status == ENV_RUNNING)
    env_run(curenv);
  else
    sched_yield();
}
```

4. env_run() in kern/env.c

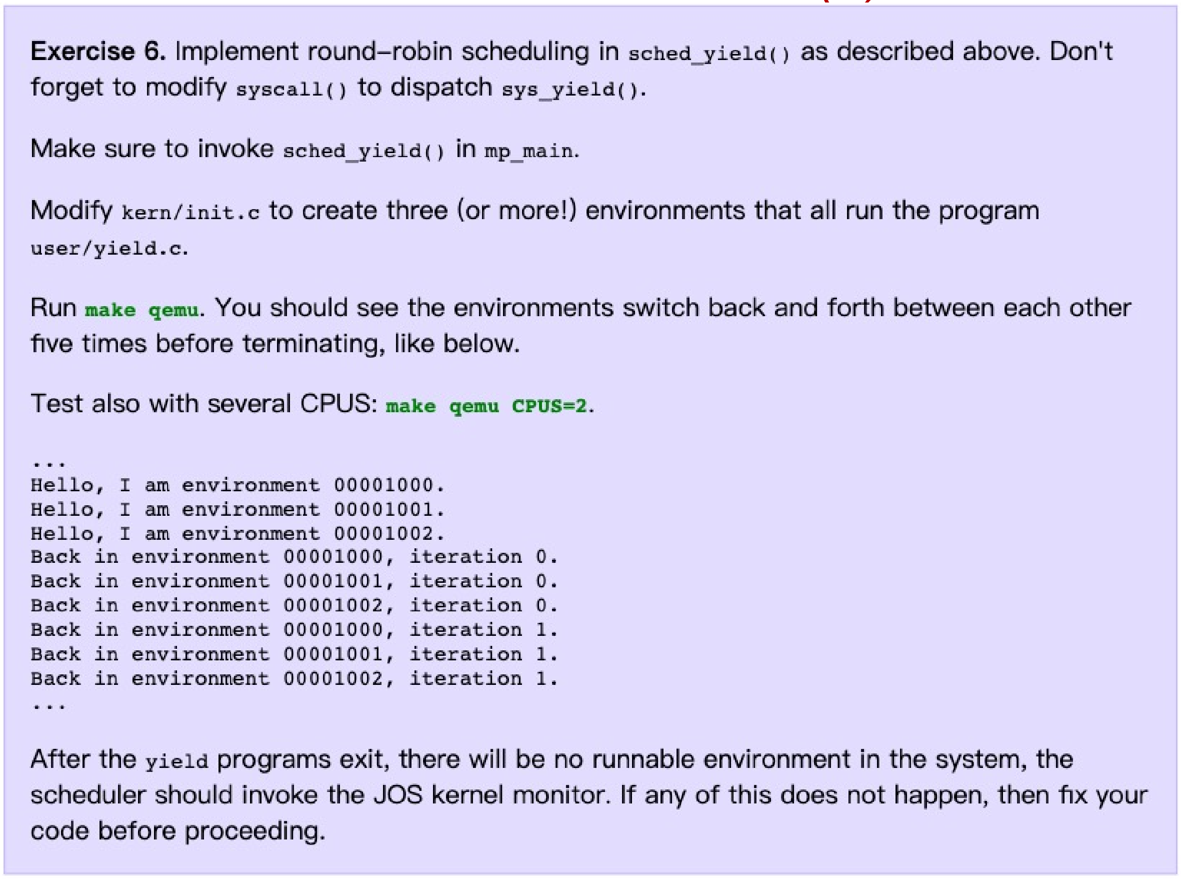
```c
// Context switch from curenv to env e.
// Note: if this is the first call to env_run, curenv is NULL.
//
// This function does not return.
//
void
env_run(struct Env *e)
{
  // Step 1: If this is a context switch (a new environment is running):
  //     1. Set the current environment (if any) back to
  //        ENV_RUNNABLE if it is ENV_RUNNING (think about
  //        what other states it can be in),
  //     2. Set 'curenv' to the new environment,
  //     3. Set its status to ENV_RUNNING,
  //     4. Update its 'env_runs' counter,
  //     5. Use lcr3() to switch to its address space.
  // Step 2: Use env_pop_tf() to restore the environment's
  //     registers and drop into user mode in the
  //     environment.

  // Hint: This function loads the new environment's state from
  //  e->env_tf.  Go back through the code you wrote above
  //  and make sure you have set the relevant parts of
  //  e->env_tf to sensible values.

  // LAB 3: Your code here.
  // Step 1: change the curenv's env_status from ING to ABLE.
  if(curenv != NULL && curenv->env_status == ENV_RUNNING)
    curenv->env_status = ENV_RUNNABLE;

  // Step 2: switch to the new env
  curenv = e;
  // Step 3: set status
  curenv->env_status = ENV_RUNNING;
  // Step 4: update env_runs counter
  curenv->env_runs++;
  // Step 5: use lcr3() to switch to its address space
  lcr3(PADDR(curenv->env_pgdir));

  // LAB 4: unlock_kernel()
  unlock_kernel();
  // Step 6: use env_pop_tf() to restore the env's register
  //     and drop into user mode in the environment.
  env_pop_tf(&curenv->env_tf);


  panic("env_run not yet implemented");
}
```

2.2 Round-Robin Scheduling

Change the JOS kernel so that it can alternate between multiple environments in *round-robin* [1] fashion:

- sched_yield() in kern/sched.c: responsible for selecting a new environment to run. It searches sequentially through the envs[] array in circular fashion, picks the first environment it finds with a status of ENV_RUNNABLE (see in inc/env.h) and calls env_run() to jump to the selected environment. Notice that sched_yield() **must NEVER** run the same environment on two CPUs at the same time.
- Implement a new system call sys_yield(): by which user environment can call to invoke kernel's sched_yield() function and thereby **voluntarily give up** the CPU to a different environment.

2.2.1 Exercise 6



Exercise 6. Implement round-robin scheduling in `sched_yield()` as described above. Don't forget to modify `syscall()` to dispatch `sys_yield()`.

Make sure to invoke `sched_yield()` in `mp_main`.

Modify `kern/init.c` to create three (or more!) environments that all run the program `user/yield.c`.

Run `make qemu`. You should see the environments switch back and forth between each other five times before terminating, like below.

Test also with several CPUS: `make qemu CPUS=2`.

```
...
Hello, I am environment 00001000.
Hello, I am environment 00001001.
Hello, I am environment 00001002.
Back in environment 00001000, iteration 0.
Back in environment 00001001, iteration 0.
Back in environment 00001002, iteration 0.
Back in environment 00001000, iteration 1.
Back in environment 00001001, iteration 1.
Back in environment 00001002, iteration 1.
...
```

After the `yield` programs exit, there will be no runnable environment in the system, the scheduler should invoke the JOS kernel monitor. If any of this does not happen, then fix your code before proceeding.

Figure 7: Exercise 6

[1]Round-robin is one of the algorithms employed by process and network schedulers in computing. As the term is generally used, time slices are assigned to each process in equal portions and in circular order, handling all processes without priority.

2.2.1.1 sched_yield()

Firstly, see its hints:

```c
// Choose a user environment to run and run it.
void
sched_yield(void)
{
        struct Env *idle;

        // Implement simple round-robin scheduling.
        //
        // Search through 'envs' for an ENV_RUNNABLE environment in
        // circular fashion starting just after the env this CPU was
        // last running.  Switch to the first such environment found.
        //
        // If no envs are runnable, but the environment previously
        // running on this CPU is still ENV_RUNNING, it's okay to
        // choose that environment.
        //
        // Never choose an environment that's currently running on
        // another CPU (env_status == ENV_RUNNING). If there are
        // no runnable environments, simply drop through to the code
        // below to halt the cpu.

        // LAB 4: Your code here.

        // sched_halt never returns
        sched_halt();
}
```

The implementation:

```c
// Choose a user environment to run and run it.
void
sched_yield(void)
{
  struct Env *idle;

  // LAB 4: Your code here.
  uint32_t i, j, start;
  struct Env *foundenv;
  foundenv = NULL;
  // get the current env running on this CPU.
  struct Env *cpu_curenv = thiscpu->cpu_env;
  start = (cpu_curenv != NULL) ? ENVX(cpu_curenv->env_id) : 0;
  // get the env with the least priority.
  for(i=0; i < NENV; i++) {
    j = (start + i) % NENV;
      if (envs[j].env_status == ENV_RUNNABLE) {
        if (foundenv == NULL)
          foundenv = &envs[j];
          break;
      }
  }
  // check if the kernel has found available CPU if not use the curenv
  if ((cpu_curenv && cpu_curenv->env_status == ENV_RUNNING) && (foundenv == NULL)) {
    env_run(cpu_curenv);
    return;
  }
  if(foundenv) {
    env_run(foundenv);
    return;
  }


  // sched_halt never returns
  sched_halt();
}
```

2.2.1.2 Modify syscall() to Dispatch sys_yield()

Add a case SYS_yield().

```
// Dispatches to the correct kernel function, passing the arguments.
int32_t
syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4, uint32_t a5)
{
        // Call the function corresponding to the 'syscallno' parameter.
        // Return any appropriate return value.
        // LAB 3: Your code here.


        //panic("syscall not implemented");

        switch (syscallno) {
        case SYS_cputs:
                sys_cputs((char *)a1, a2);
                return 0;
        case SYS_cgetc:
                return sys_cgetc();
        case SYS_getenvid:
                return sys_getenvid();
        case SYS_env_destroy:
                return sys_env_destroy(a1);
        case SYS_yield:
                sys_yield();
                return 0;
        default:
                return -E_INVAL;
        }
}
```

2.2.1.3 Modify i386_init() in kern/init.c to Add More Environments

```
#if defined(TEST)
        // Don't touch -- used by grading script!
        ENV_CREATE(TEST, ENV_TYPE_USER);
#else
        // Touch all you want.
        //ENV_CREATE(user_primes, ENV_TYPE_USER);
        ENV_CREATE(user_yield, ENV_TYPE_USER);
        ENV_CREATE(user_yield, ENV_TYPE_USER);
        ENV_CREATE(user_yield, ENV_TYPE_USER);
```

```
#endif // TEST*

        // Schedule and run the first user environment!
        sched_yield();
```

The result is as follows:

```
+ cc kern/init.c
+ ld obj/kern/kernel
+ mk obj/kern/kernel.img
sed "s/localhost:1234/localhost:26000/" < .gdbinit.tmpl > .gdbinit
qemu-system-i386 -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -serial
    ↪ mon:stdio -gdb tcp::26000 -D qemu.log -smp 1
6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2
[00000000] new env 00001000
[00000000] new env 00001001
[00000000] new env 00001002
Hello, I am environment 00001000.
Hello, I am environment 00001001.
Hello, I am environment 00001002.
Back in environment 00001000, iteration 0.
Back in environment 00001001, iteration 0.
Back in environment 00001002, iteration 0.
Back in environment 00001000, iteration 1.
Back in environment 00001001, iteration 1.
Back in environment 00001002, iteration 1.
Back in environment 00001000, iteration 2.
Back in environment 00001001, iteration 2.
Back in environment 00001002, iteration 2.
Back in environment 00001000, iteration 3.
Back in environment 00001001, iteration 3.
Back in environment 00001002, iteration 3.
Back in environment 00001000, iteration 4.
All done in environment 00001000.
[00001000] exiting gracefully
[00001000] free env 00001000
Back in environment 00001001, iteration 4.
All done in environment 00001001.
[00001001] exiting gracefully
[00001001] free env 00001001
Back in environment 00001002, iteration 4.
All done in environment 00001002.
[00001002] exiting gracefully
[00001002] free env 00001002
No runnable environments in the system!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
```

2.3 System Calls for Environment Creation

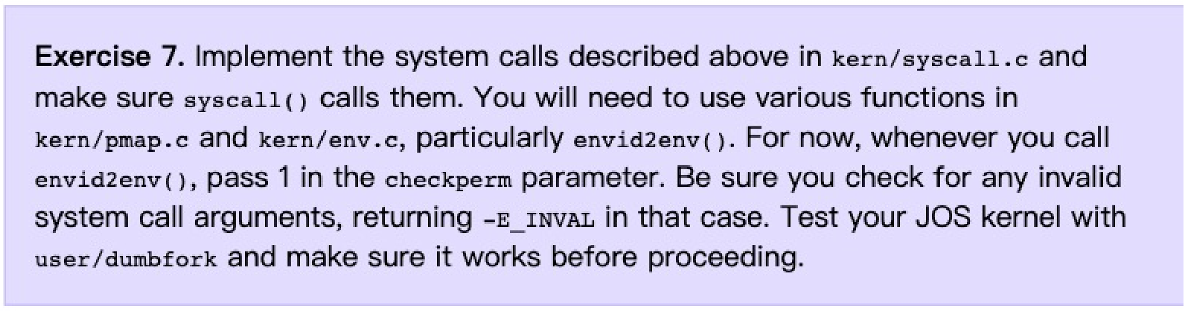
Now implement the necessary JOS system calls to allow user environments to create and start other new user environments. Unix provides the fork() system call as its process creation primitive.

- fork() copies the entire address space of the calling process (the parent) to create a new process (the child).
- In the parent, the fork() returns the child's process IDs.
- In the child, the fork() returns 0.
- Both the parent and child have its private address space.

Implement a more primitive system call to create a new environment.

- sys_exofork(): Create a new environment.
- sys_env_set_status(): Set status.
- sys_page_alloc: Allocate page.
- sys_page_map: Copy a page mapping.
- sys_page_unmap: Unmap a page.

2.3.1 Exercise 7



Exercise 7. Implement the system calls described above in `kern/syscall.c` and make sure `syscall()` calls them. 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.

Figure 8: Exercise 7

2.3.2 The Implementation of sys_exofork()

Let's see the hints:

```c
// 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.

        // LAB 4: Your code here.
        panic("sys_exofork not implemented");
}
```

See the implementation:

```
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.

        // LAB 4: Your code here.
        struct Env *newenv;
        int32_t ret = 0;
        if ((ret = env_alloc(&newenv, curenv->env_id)) < 0) {
                return ret;
        }
        newenv->env_status = ENV_NOT_RUNNABLE;
        newenv->env_tf = curenv->env_tf;
        newenv->env_tf.tf_regs.reg_eax = 0;
        return newenv->env_id;
        //panic("sys_exofork not implemented");
}
```

2.3.3 The Implementation of sys_env_set_status()

Sets the status of a specified environment to ENV_RUNNABLE or ENV_NOT_RUNNABLE. This system call is typically used to mark a new environment ready to run, once its address space and register state has been fully initialized.

2.3.3.1 See The Initial Hints:

```
// 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.
```

```c

        // LAB 4: Your code here.
        panic("sys_env_set_status not implemented");
}
```

2.3.3.2 See the implementation

```c
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.

        // LAB 4: Your code here.
        struct Env *env;
        // Check whether the current environment is a valid status
        if(status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE)
                return -E_INVAL;
        // Check whether the current environment has permission to set
        // envid's status
        if (envid2env(envid, &env, 1) < 0)
                return -E_BAD_ENV;
        env->env_status = status;
        return 0;
        //panic("sys_env_set_status not implemented");
}
```

2.3.3.3 The envid2env() Function in kern/env.c

```c
//
// Converts an envid to an env pointer.
// If checkperm is set, the specified environment must be either the
// current environment or an immediate child of the current environment.
//
// RETURNS
//   0 on success, -E_BAD_ENV on error.
//   On success, sets *env_store to the environment.
//   On error, sets *env_store to NULL.
//
int
envid2env(envid_t envid, struct Env **env_store, bool checkperm)
{
  struct Env *e;

  // If envid is zero, return the current environment.
  if (envid == 0) {
    *env_store = curenv;
    return 0;
  }

  // Look up the Env structure via the index part of the envid,
  // then check the env_id field in that struct Env
  // to ensure that the envid is not stale
  // (i.e., does not refer to a _previous_ environment
  // that used the same slot in the envs[] array).
  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;
  }

  *env_store = e;
  return 0;
}
```

2.3.4 The Implementation of sys_page_alloc()

Allocate a page of physical memory and maps it at a given virtual address in a given environment's address space.

2.3.4.1 The hints

```
// Allocate a page of memory and map it at 'va' with permission
// 'perm' in the address space of 'envid'.
// The page's contents are set to 0.
// If a page is already mapped at 'va', that page is unmapped as a
// side effect.
//
// perm -- PTE_U | PTE_P must be set, PTE_AVAIL | PTE_W may or may not be set,
//         but no other bits may be set.  See PTE_SYSCALL in inc/mmu.h.
//
// Return 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 va >= UTOP, or va is not page-aligned.
//      -E_INVAL if perm is inappropriate (see above).
//      -E_NO_MEM if there's no memory to allocate the new page,
//              or to allocate any necessary page tables.
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!

        // LAB 4: Your code here.
        panic("sys_page_alloc not implemented");
}
```

2.3.4.2 See the implementation

```c
static int
sys_page_alloc(envid_t envid, void *va, int perm)
{
        // LAB 4: Your code here.
        struct Env *env;
        // check whether environment envid exists or not.
        if (envid2env(envid, &env, 1) < 0)
                return -E_BAD_ENV;
        // Check if va >= UTOP or va is not page-aligned.
        if((uintptr_t) va >= UTOP || PGOFF(va))
                return -E_INVAL;
        // Check if perm is inappropriate
        if((perm & PTE_SYSCALL) == 0)
                return -E_INVAL;
        if((perm & ~PTE_SYSCALL))
                return -E_INVAL;

        struct PageInfo *pp = page_alloc(ALLOC_ZERO);
        if(!pp)
                return -E_NO_MEM;
        if (page_insert(env->env_pgdir, pp, va, perm) < 0){
                // page_insert fails: remember free the page
                page_free(pp);
                return -E_NO_MEM;
        }

        // Success, return 0
        return 0;
}
```

2.3.5 The Implementation of sys_page_map()

sys_page_map() is used to copy a page mapping(**NOT the contents of a page**) from one environment's address space to another, leaving a memory sharing arrangement in place so that the new and the old mappings both refer to the same page of physical memory.

2.3.5.1 The Hints

2.3.5.2 Some hints from page_lookup()

```c
//
// Return the page mapped at virtual address 'va'.
// If pte_store is not zero, then we store in it the address
// of the pte for this page.  This is used by page_remove and
// can be used to verify page permissions for syscall arguments,
// but should not be used by most callers.
//
// Return NULL if there is no page mapped at va.
//
// Hint: the TA solution uses pgdir_walk and pa2page.
//
struct PageInfo *
```

```c
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
}
```

2.3.5.3 Some hints from page_insert()

```c
// Map the physical page 'pp' at virtual address 'va'.
// The permissions (the low 12 bits) of the page table entry
// should be set to 'perm|PTE_P'.
//
// Requirements
//   - If there is already a page mapped at 'va', it should be page_remove()d.
//   - If necessary, on demand, a page table should be allocated and inserted
//     into 'pgdir'.
//   - pp->pp_ref should be incremented if the insertion succeeds.
//   - The TLB must be invalidated if a page was formerly present at 'va'.
//
// Corner-case hint: Make sure to consider what happens when the same
// pp is re-inserted at the same virtual address in the same pgdir.
// However, try not to distinguish this case in your code, as this
// frequently leads to subtle bugs; there's an elegant way to handle
// everything in one code path.
//
// RETURNS:
//   0 on success
//   -E_NO_MEM, if page table couldn't be allocated
//
// Hint: The TA solution is implemented using pgdir_walk, page_remove,
// and page2pa.
//
int
page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm)
{
}
```

2.3.5.4 The Implementation of sys_page_map()

```c
static int
sys_page_map(envid_t srcenvid, void *srcva,
             envid_t dstenvid, void *dstva, int perm)
{
        // LAB 4: Your code here.
        struct Env *srcenv, *dstenv;
        struct PageInfo *pp;
        pte_t *pte;
        // srcenvid or/and dstenvid doesn't exist.
        if ((envid2env(srcenvid, &srcenv, 1) < 0) || (envid2env(dstenvid, &dstenv, 1) < 0)) {
                return -E_BAD_ENV;
        }
        // Check srcva>= UTOP or srcva is not page-aligned
        if ((uintptr_t)srcva >= UTOP || PGOFF(srcva))
                return -E_INVAL;
        // Check dstva>= UTOP or dstva is not page-aligned
        if ((uintptr_t)dstva >= UTOP || PGOFF(dstva))
                return -E_INVAL;
        // Check the perm is inappropriate
        if (((perm & PTE_SYSCALL) == 0)|| (perm & ~PTE_SYSCALL))
```

```
                return -E_INVAL;
        // Look up the page
        pp = page_lookup(srcenv->env_pgdir, srcva, &pte);
        if (pp == NULL)
                return -E_INVAL;

        // Check if (perm & PTE_W), but srcva is read-only in
        // srcenvid's address space.
        if ((perm & PTE_W) && ((*pte & PTE_W) == 0))
                return -E_INVAL;
        // Check if there's no memory in dstenv to allocate
        // any necessary page tables.
        if(page_insert(dstenv->env_pgdir, pp, dstva, perm) < 0)
                return -E_NO_MEM;
        return 0;
        //panic("sys_page_map not implemented");
}
```

2.3.5.5 Summary

Its overall steps can be summarized as follow:

1. Check the parameters.
2. Use page_lookup() to find the physical page pp mapped by srcva in srcenv.
3. Map the pp to dstva in dstenv.

2.3.6 The Implementation of sys_page_unmap()

Unmap a page mapped at a given virtual address in a given environment.
Let's firstly check the hints:

2.3.6.1 The Hints of sys_page_unmap()

```
// Unmap the page of memory at 'va' in the address space of 'envid'.
// If no page is mapped, the function silently succeeds.
//
// Return 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 va >= UTOP, or va is not page-aligned.
static int
sys_page_unmap(envid_t envid, void *va)
{
        // Hint: This function is a wrapper around page_remove().

        // LAB 4: Your code here.
        panic("sys_page_unmap not implemented");
}
```

The implementation is quite easy.

2.3.6.2 The Implementation

```c
static int
sys_page_unmap(envid_t envid, void *va)
{
	// Hint: This function is a wrapper around page_remove().

	// LAB 4: Your code here.
	struct Env *env;
	if (envid2env(envid, &env, 1) < 0)
		return -E_BAD_ENV;
	if((uintptr_T)va >= UTOP || PGOFF(va))
		return -E_INVAL;
	page_remove(env->env_pgdir, va);
	return 0;
	//panic("sys_page_unmap not implemented");
}
```

2.3.7 Current syscall() in syscall.c

```c
int32_t
syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4, uint32_t a5)
{
	// Call the function corresponding to the 'syscallno' parameter.
	// Return any appropriate return value.
	// LAB 3: Your code here.


	//panic("syscall not implemented");

	switch (syscallno) {
	case SYS_cputs:
		sys_cputs((char *)a1, a2);
		return 0;
	case SYS_cgetc:
		return sys_cgetc();
	case SYS_getenvid:
		return sys_getenvid();
	case SYS_env_destroy:
		return sys_env_destroy(a1);
	case SYS_yield:
		sys_yield();
		return 0;
	case SYS_page_alloc:
		return sys_page_alloc((envid_t)a1, (void *)a2, (int)a3);
	case SYS_page_map:
		return sys_page_map((envid_t)a1, (void *)a2, (envid_t)a3, (void *)a4, (int)a5);
	case SYS_page_unmap:
		return sys_page_unmap((envid_t)a1, (void *)a2);
	case SYS_exofork:
		return sys_exofork();
	case SYS_env_set_status:
		return sys_env_set_status((envid_t)a1, (int)a2);
	default:
		return -E_INVAL;
	}
}
```

3 Part B: Copy-on-Write Fork

3.1 The Motivation of Copy-on-Write Technique

The copying of the parent's space into the child is the most expensive part of the fork() operation. However, a call to fork() is frequently followed by a call to exec() in the child process, which replace the child's memory with a new program. Under this condition, the time spent copying the parent's address space is largely wasted because the child process will use very little of its memory before exec()
Copy-on-Write Technique: For this reason, later verion Unix took advantage of virtual memory hardware to allow the parent and child to **share** the memory mapped into the respective address space until one of the processes actually modifies it. This technique is know as *copy-on-write* technique. To do this,

1. On fork(), the kernel would copy the address space mappings from the parent to the child instead of the content of the mapped pages. At the same time, the kernel should mark the now-shared pages read-only.

2. When one of the two processes tries to write to one of these shared pages, the process take a page fault. At this point, the kernel realizes that the page was really a *virtual* or *copy-on-write* copy, and so the kernel makes a **new, private writable** copy of the page for the faulting process.

In this way, the contents of individual page aren't actually copied until they are actually written to. This optimization makes a fork() followed by an exec() in the child much cheaper.

3.2 User-level Page Fault Handling

3.2.1 Setting the Page Fault Handler

In order to handle its own page fault, a user environment will need to register a **page fault handler entrypoint** with the JOS kernel. This registration is done via sys_env_set_pgfault_upcall() system call.

3.2.1.1 Exercise 8

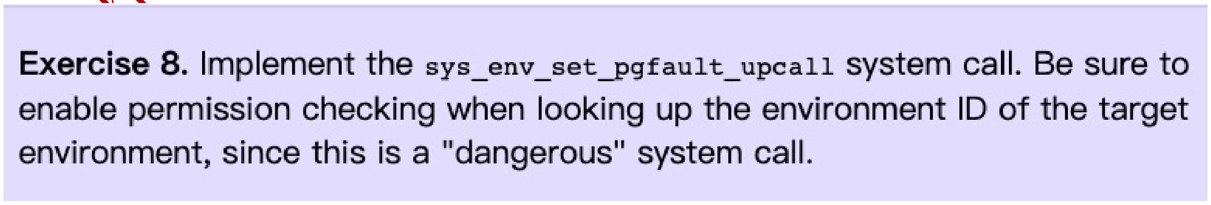

Exercise 8. Implement the `sys_env_set_pgfault_upcall` system call. Be sure to enable permission checking when looking up the environment ID of the target environment, since this is a "dangerous" system call.

Figure 9: Exercise 8

3.2.1.2 The hints in sys_env_set_pgfault_upcall()

```
// Set the page fault upcall for 'envid' by modifying the corresponding struct
// Env's 'env_pgfault_upcall' field.  When 'envid' causes a page fault, the
// kernel will push a fault record onto the exception stack, then branch to
// 'func'.
//
// 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.
static int
sys_env_set_pgfault_upcall(envid_t envid, void *func)
{
        // LAB 4: Your code here.
        panic("sys_env_set_pgfault_upcall not implemented");
}
```

3.2.1.3 See The Implementation of sys_env_set_pgfault_upcall()

```
static int
sys_env_set_pgfault_upcall(envid_t envid, void *func)
{
        // LAB 4: Your code here.
        struct Env *env;
        // the environment envid doesn't currently exist
        // or the caller has no permission to change envid
        if(envid2env(envid, &env, 1) < 0)
                return -E_BAD_ENV;
        // set the env_pgfault_upcall
        env->env_pgfault_upcall = func;
        return 0;

        //panic("sys_env_set_pgfault_upcall not implemented");
}
```

3.2.2 Normal and Exception Stacks in User Environments

3.2.2.1 Normal User Stack and User Exception Stack

1. **Normal User Stack:** During normal execution, a user environment in JOS will run on the normal user stack: its ESP register starts out pointing at USTACK-TOP, and the stack data it pushes resides on the page between USTACKTOP-PGSIZE and USTACKTOP-1 inclusive.

2. **User Exception Stack:** When a page fault occurs in user mode, however, the kernel will restart the user environment running a designated user-level page fault handler on a different stack, namely the **user exception stack**.

3.2.2.2 About JOS User Exception Stack

JOS user exception stack is also one page in size.

The JOS user exception stack is also one page in size, and its top is defined to be at virtual address UXSTACKTOP, so the valid bytes of the user exception stack are from UXSTACKTOP-PGSIZE through UXSTACKTOP-1 inclusive. While running on this exception stack, the user-level page fault handler can use JOS's regular system calls to map new pages or adjust mappings so as to fix whatever problem originally caused the page fault. Then the user-level page fault handler returns, via an assembly language stub, to the faulting code on the original stack.

Each user environment that wants to support user-level page fault handling will need to allocate memory for its own exception stack, using the sys_page_alloc() system call introduced in part A.

3.2.3 Invoking the User Page Fault Handler

We will now need to change the page fault handling coder in **kern/trap.c** to handle page faults from user mode as follows.

1. If there is no page fault handler registered, the JOS kernel destroys the user environment with a message as before.
2. Otherwise
   - the kernel sets up a trap frame on the exception stack that looks like a **struct UTrapframe** from **inc/trap.h**:

```
                    <-- UXSTACKTOP
trap-time esp
trap-time eflags
trap-time eip
trap-time eax       start of struct PushRegs
trap-time ecx
trap-time edx
trap-time ebx
trap-time esp
trap-time ebp
trap-time esi
trap-time edi       end of struct PushRegs
tf_err (error code)
fault_va            <-- %esp when handler is run
```

Figure 10: Struct UTrapframe

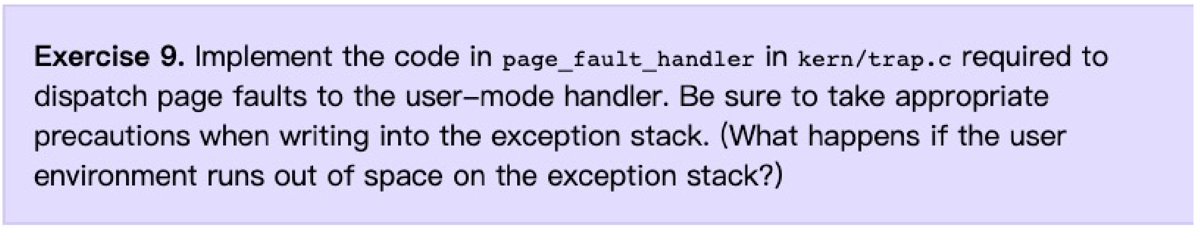
   - The kernel then arranges for the user environment to resume execution with the page fault handler running on the exception stack with this stack frame; you must figure out how to make this happen. The fault_va is the virtual address that caused the page fault.
   - If the user environment is already running on the user exception stack when an exception occurs, then the page fault handler itself has faulted. In this case, you should start the new stack frame just under the current tf-¿tf_esp rather than at UXSTACKTOP. You should first push an empty 32-bit word, then a struct UTrapframe.

Stack Type	Address
User Normal Stack	[UTEXT, USTACKTOP]
User Exception Stack	[UXSTACKTOP - PGSIZE, UXSTACKTOP]
Kernel Stack	[KSTACK-KSTKSIZE, KSTACKTOP]

- To test whether tf-¿tf_esp is already on the user exception stack, check whether it is in the range between UXSTACKTOP-PGSIZE and UXSTACKTOP-1, inclusive.

NOW, the involved stack are:

3.2.4 Exercise 9



Exercise 9. Implement the code in `page_fault_handler` in `kern/trap.c` required to dispatch page faults to the user-mode handler. Be sure to take appropriate precautions when writing into the exception stack. (What happens if the user environment runs out of space on the exception stack?)

Figure 11: Exercise 9

Firstly, see its hints:

```
void
page_fault_handler(struct Trapframe *tf)
{
        uint32_t fault_va;

        // Read processor's CR2 register to find the faulting address
        fault_va = rcr2();

        // Handle kernel-mode page faults.

        // LAB 3: Your code here.
        if(tf->tf_cs && 0x01 == 0) {
                panic("page fault in kernel mode, fault address %d\n", fault_va);
        }

        // We've already handled kernel-mode exceptions, so if we get here,
        // the page fault happened in user mode.

        // Call the environment's page fault upcall, if one exists.  Set up a
        // page fault stack frame on the user exception stack (below
        // UXSTACKTOP), then branch to curenv->env_pgfault_upcall.
        //
        // The page fault upcall might cause another page fault, in which case
        // we branch to the page fault upcall recursively, pushing another
        // page fault stack frame on top of the user exception stack.
        //
        // It is convenient for our code which returns from a page fault
        // (lib/pfentry.S) to have one word of scratch space at the top of the
        // trap-time stack; it allows us to more easily restore the eip/esp. In
        // the non-recursive case, we don't have to worry about this because
        // the top of the regular user stack is free.  In the recursive case,
        // this means we have to leave an extra word between the current top of
        // the exception stack and the new stack frame because the exception
        // stack _is_ the trap-time stack.
        //
        // If there's no page fault upcall, the environment didn't allocate a
        // page for its exception stack or can't write to it, or the exception
        // stack overflows, then destroy the environment that caused the fault.
        // Note that the grade script assumes you will first check for the page
        // fault upcall and print the "user fault va" message below if there is
        // none.  The remaining three checks can be combined into a single test.
        //
        // Hints:
        //   user_mem_assert() and env_run() are useful here.
        //   To change what the user environment runs, modify 'curenv->env_tf'
        //   (the 'tf' variable points at 'curenv->env_tf').

        // LAB 4: Your code here.

        // Destroy the environment that caused the fault.
        cprintf("[%08x] user fault va %08x ip %08x\n",
                curenv->env_id, fault_va, tf->tf_eip);
        print_trapframe(tf);
        env_destroy(curenv);
}
```

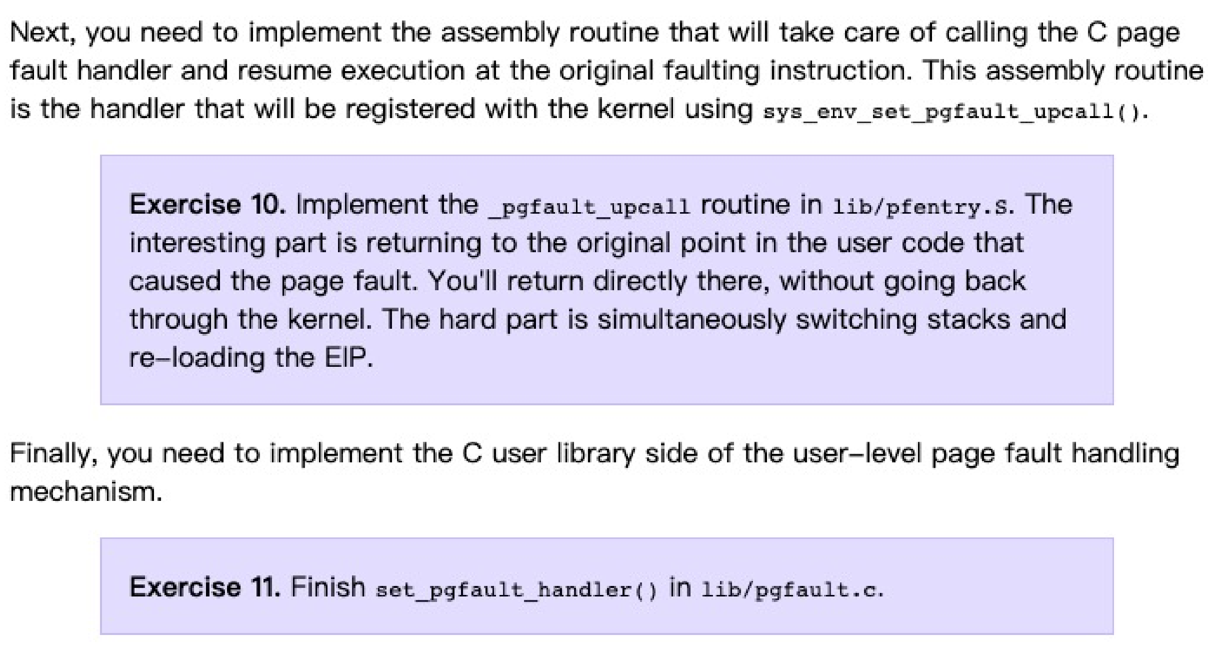
3.2.4.1 Trapframe and UTrapframe

```
struct Trapframe {
        struct PushRegs tf_regs;
        uint16_t tf_es;
        uint16_t tf_padding1;
        uint16_t tf_ds;
        uint16_t tf_padding2;
        uint32_t tf_trapno;
        /* below here defined by x86 hardware */
        uint32_t tf_err;
        uintptr_t tf_eip;
        uint16_t tf_cs;
        uint16_t tf_padding3;
        uint32_t tf_eflags;
        /* below here only when crossing rings, such as from user to kernel */
        uintptr_t tf_esp;
        uint16_t tf_ss;
        uint16_t tf_padding4;
} __attribute__((packed));

struct UTrapframe {
        /* information about the fault */
        uint32_t utf_fault_va;  /* va for T_PGFLT, 0 otherwise */
        uint32_t utf_err;
        /* trap-time return state */
        struct PushRegs utf_regs;
        uintptr_t utf_eip;
        uint32_t utf_eflags;
        /* the trap-time stack to return to */
        uintptr_t utf_esp;
} __attribute__((packed));
```

3.2.4.2 The Implementation of Exercise 9

```
void
page_fault_handler(struct Trapframe *tf)
{
  uint32_t fault_va;

  // Read processor's CR2 register to find the faulting address
  fault_va = rcr2();

  // Handle kernel-mode page faults.

  // LAB 3: Your code here.
  if(tf->tf_cs && 0x01 == 0) {
    panic("page fault in kernel mode, fault address %d\n", fault_va);
  }


  // LAB 4: Your code here.

  struct UTrapframe *utf;
  if(curenv->env_pgfault_upcall) {
    // tf->tf_esp in [UXSTACKTOP - PGSIZE, UXSTACKTOP-1]
    if(UXSTACKTOP - PGSIZE <= tf->tf_esp && tf->tf_esp <= UXSTACKTOP - 1)
      utf = (struct UTrapframe *)(tf->tf_esp - sizeof(struct UTrapframe) - 4);
    else
      utf = (struct UTrapframe *)(UXSTACKTOP - sizeof(struct UTrapframe));
    // mem_assert
    user_mem_assert(curenv, (void *)utf, sizeof(struct UTrapframe), PTE_U | PTE_W);
    // set utf
    utf->utf_fault_va = fault_va;
    utf->utf_err = tf->tf_trapno;
    utf->utf_eip = tf->tf_eip;
    utf->utf_eflags = tf->tf_eflags;
    utf->utf_esp = tf->tf_esp;
    utf->utf_regs = tf->tf_regs;

    // set tf
    tf->tf_eip = (uint32_t)curenv->env_pgfault_upcall;
    tf->tf_esp = (uint32_t)utf;
    // run current env
    env_run(curenv);
  }else {

  // Destroy the environment that caused the fault.
  cprintf("[%08x] user fault va %08x ip %08x\n",
    curenv->env_id, fault_va, tf->tf_eip);
  print_trapframe(tf);
  env_destroy(curenv);
  }
}
```

3.2.5 User-mode Page Fault Entrypoint: Exercise 10 and Exercise 11



Next, you need to implement the assembly routine that will take care of calling the C page fault handler and resume execution at the original faulting instruction. This assembly routine is the handler that will be registered with the kernel using `sys_env_set_pgfault_upcall()`.

Exercise 10. Implement the `_pgfault_upcall` routine in `lib/pfentry.S`. The interesting part is returning to the original point in the user code that caused the page fault. You'll return directly there, without going back through the kernel. The hard part is simultaneously switching stacks and re-loading the EIP.

Finally, you need to implement the C user library side of the user-level page fault handling mechanism.

Exercise 11. Finish `set_pgfault_handler()` in `lib/pgfault.c`.

Figure 12: Exercise 10 and Exercise 11

3.2.5.1 See the hints in lib/pfentry.S

```
#include <inc/mmu.h>
#include <inc/memlayout.h>

// Page fault upcall entrypoint.

// This is where we ask the kernel to redirect us to whenever we cause
// a page fault in user space (see the call to sys_set_pgfault_handler
// in pgfault.c).
//
// When a page fault actually occurs, the kernel switches our ESP to
// point to the user exception stack if we're not already on the user
// exception stack, and then it pushes a UTrapframe onto our user
// exception stack:
//
//      trap-time esp
//      trap-time eflags
//      trap-time eip
//      utf_regs.reg_eax
//      ...
//      utf_regs.reg_esi
//      utf_regs.reg_edi
//      utf_err (error code)
//      utf_fault_va            <-- %esp
//
// If this is a recursive fault, the kernel will reserve for us a
// blank word above the trap-time esp for scratch work when we unwind
// the recursive call.
//
```

```
// We then have call up to the appropriate page fault handler in C
// code, pointed to by the global variable '_pgfault_handler'.
.text
.globl _pgfault_upcall
_pgfault_upcall:
        // Call the C page fault handler.
        pushl %esp                      // function argument: pointer to UTF
        movl _pgfault_handler, %eax
        call *%eax
        addl $4, %esp                   // pop function argument

        // Now the C page fault handler has returned and you must return
        // to the trap time state.
        // Push trap-time %eip onto the trap-time stack.
        //
        // Explanation:
        //   We must prepare the trap-time stack for our eventual return to
        //   re-execute the instruction that faulted.
        //   Unfortunately, we can't return directly from the exception stack:
        //   We can't call 'jmp', since that requires that we load the address
        //   into a register, and all registers must have their trap-time
        //   values after the return.
        //   We can't call 'ret' from the exception stack either, since if we
        //   did, %esp would have the wrong value.
        //   So instead, we push the trap-time %eip onto the *trap-time* stack!
        //   Below we'll switch to that stack and call 'ret', which will
        //   restore %eip to its pre-fault value.
        //
        //   In the case of a recursive fault on the exception stack,
        //   note that the word we're pushing now will fit in the
        //   blank word that the kernel reserved for us.
        //
        // Throughout the remaining code, think carefully about what
        // registers are available for intermediate calculations.  You
        // may find that you have to rearrange your code in non-obvious
        // ways as registers become unavailable as scratch space.
       // LAB 4: Your code here.

        // Restore the trap-time registers.  After you do this, you
        // can no longer modify any general-purpose registers.
        // LAB 4: Your code here.

        // Restore eflags from the stack.  After you do this, you can
        // no longer use arithmetic operations or anything else that
        // modifies eflags.
        // LAB 4: Your code here.

        // Switch back to the adjusted trap-time stack.
        // LAB 4: Your code here.

        // Return to re-execute the instruction that faulted.
        // LAB 4: Your code here.
```

3.2.5.2 Implementation of the _pgfault_upcall routine in lib/pfentry.S

```
.text
.globl _pgfault_upcall
_pgfault_upcall:
        // Call the C page fault handler.
        pushl %esp                      // function argument: pointer to UTF
        movl _pgfault_handler, %eax
        call *%eax
        addl $4, %esp                   // pop function argument
        // LAB 4: Your code here.

        // 48(%esp) = 48(%esp) - 4
        movl 48(%esp), %ebp
        subl $4, %ebp
        movl %ebp, 48(%esp)

        // %ebp = 40(%esp)
        movl 40(%esp), %eax
        movl %eax, (%ebp)

        // Restore the trap-time registers.  After you do this, you
        // can no longer modify any general-purpose registers.
        // LAB 4: Your code here.
        addl $8, %esp
        popal

        // Restore eflags from the stack.  After you do this, you can
        // no longer use arithmetic operations or anything else that
        // modifies eflags.
        // LAB 4: Your code here.
        addl $4, %esp
        popfl

        // Switch back to the adjusted trap-time stack.
        // LAB 4: Your code here.
        popl %esp

        // Return to re-execute the instruction that faulted.
        // LAB 4: Your code here.
        ret
```

3.2.5.3 Exercise 11: Finish set_pgfault_handler() in lib/pgfault.c.

Firstly, see its hints:

```c
// User-level page fault handler support.
// Rather than register the C page fault handler directly with the
// kernel as the page fault handler, we register the assembly language
// wrapper in pfentry.S, which in turns calls the registered C
// function.

#include <inc/lib.h>


// Assembly language pgfault entrypoint defined in lib/pfentry.S.
extern void _pgfault_upcall(void);

// Pointer to currently installed C-language pgfault handler.
void (*_pgfault_handler)(struct UTrapframe *utf);

//
// Set the page fault handler function.
// If there isn't one yet, _pgfault_handler will be 0.
// The first time we register a handler, we need to
// allocate an exception stack (one page of memory with its top
// at UXSTACKTOP), and tell the kernel to call the assembly-language
// _pgfault_upcall routine when a page fault occurs.
//
void
set_pgfault_handler(void (*handler)(struct UTrapframe *utf))
{
        int r;

        if (_pgfault_handler == 0) {
                // First time through!
                // LAB 4: Your code here.
                panic("set_pgfault_handler not implemented");
        }

        // Save handler pointer for assembly to call.
        _pgfault_handler = handler;
}
```

See the implementation:

```c
void
set_pgfault_handler(void (*handler)(struct UTrapframe *utf))
{
        int r;

        if (_pgfault_handler == 0) {
                // First time through!
                // LAB 4: Your code here.
                if((r = sys_page_alloc(thisenv->env_id, (void *)(UXSTACKTOP - PGSIZE), PTE_P | PTE_W | PTE_U)) < 0)
                        panic("set_pgfault_handler failed: %e", r);
                sys_env_set_pgfault_upcall(thisenv->env_id, _pgfault_upcall);
                //panic("set_pgfault_handler not implemented");
        }

        // Save handler pointer for assembly to call.
        _pgfault_handler = handler;
}
```

Remind register the sys_env_set_pgfault_upcall() in kern/syscall.c:

```c
// Set the page fault upcall for 'envid' by modifying the corresponding struct
// Env's 'env_pgfault_upcall' field.  When 'envid' causes a page fault, the
// kernel will push a fault record onto the exception stack, then branch to
// 'func'.
//
// 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.
static int
sys_env_set_pgfault_upcall(envid_t envid, void *func)
{
        // LAB 4: Your code here.
        struct Env *env;
        // the environment envid doesn't currently exist
        // or the caller has no permission to change envid
        if(envid2env(envid, &env, 1) < 0)
```

```
                return -E_BAD_ENV;
        // set the env_pgfault_upcall
        env->env_pgfault_upcall = func;
        return 0;

        //panic("sys_env_set_pgfault_upcall not implemented");
}
```

3.2.6 Remember to Modify Your syscall() in kern/syscall.c

Add a case in syscall() in kern/syscall.c:

```
        case SYS_env_set_pgfault_upcall:
                return sys_env_set_pgfault_upcall((envid_t)a1, (void *)a2);
```

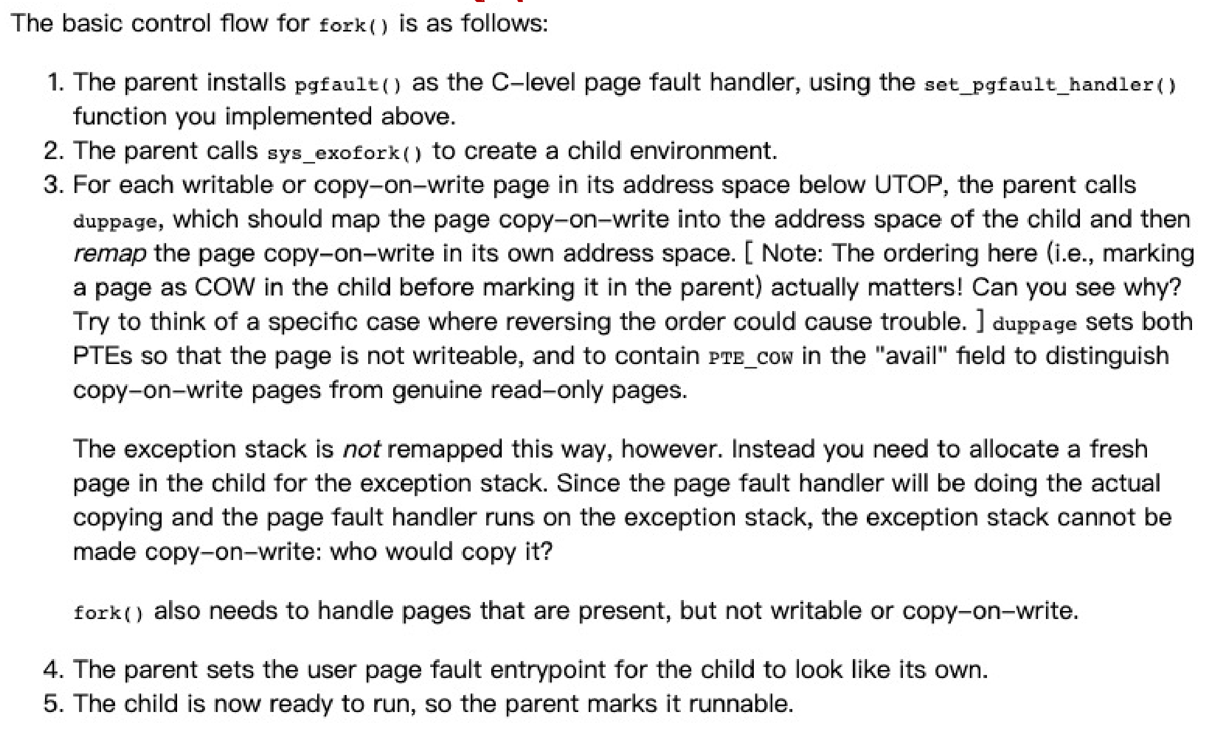
3.3 Implementing Copy-on-Write Fork

Like dumbfork(), fork() should:

- Create a new environment.
- Then scan through the parent environment's entire address space and set up corresponding page mappings in the child.

The key difference is that, while dumbfork() copied pages, fork() will initially only copy page mappings. fork() will copy each page only when one of the environments tries to write it.

The basic control flow for fork():



The basic control flow for `fork()` is as follows:

1. The parent installs `pgfault()` as the C-level page fault handler, using the `set_pgfault_handler()` function you implemented above.
2. The parent calls `sys_exofork()` to create a child environment.
3. For each writable or copy-on-write page in its address space below UTOP, the parent calls `duppage`, which should map the page copy-on-write into the address space of the child and then *remap* the page copy-on-write in its own address space. [Note: The ordering here (i.e., marking a page as COW in the child before marking it in the parent) actually matters! Can you see why? Try to think of a specific case where reversing the order could cause trouble.] `duppage` sets both PTEs so that the page is not writeable, and to contain `PTE_COW` in the "avail" field to distinguish copy-on-write pages from genuine read-only pages.

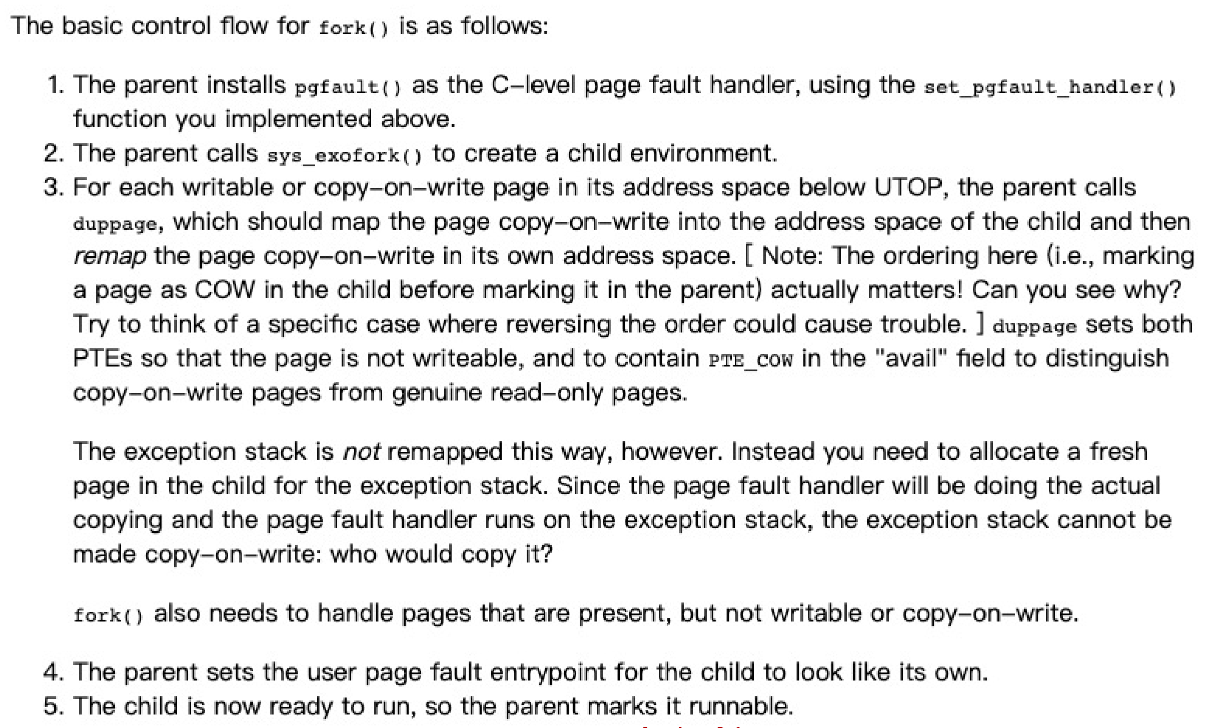
   The exception stack is *not* remapped this way, however. Instead you need to allocate a fresh page in the child for the exception stack. Since the page fault handler will be doing the actual copying and the page fault handler runs on the exception stack, the exception stack cannot be made copy-on-write: who would copy it?

   `fork()` also needs to handle pages that are present, but not writable or copy-on-write.

4. The parent sets the user page fault entrypoint for the child to look like its own.
5. The child is now ready to run, so the parent marks it runnable.

Figure 13: Basic control flow of fork()

Each time one of the environments writes a copy-on-write page that it hasn't yet written, it will take a page fault. Here's the control flow for the user page fault handler:



The basic control flow for `fork()` is as follows:

1. The parent installs `pgfault()` as the C-level page fault handler, using the `set_pgfault_handler()` function you implemented above.
2. The parent calls `sys_exofork()` to create a child environment.
3. For each writable or copy-on-write page in its address space below UTOP, the parent calls `duppage`, which should map the page copy-on-write into the address space of the child and then *remap* the page copy-on-write in its own address space. [Note: The ordering here (i.e., marking a page as COW in the child before marking it in the parent) actually matters! Can you see why? Try to think of a specific case where reversing the order could cause trouble.] `duppage` sets both PTEs so that the page is not writeable, and to contain `PTE_COW` in the "avail" field to distinguish copy-on-write pages from genuine read-only pages.

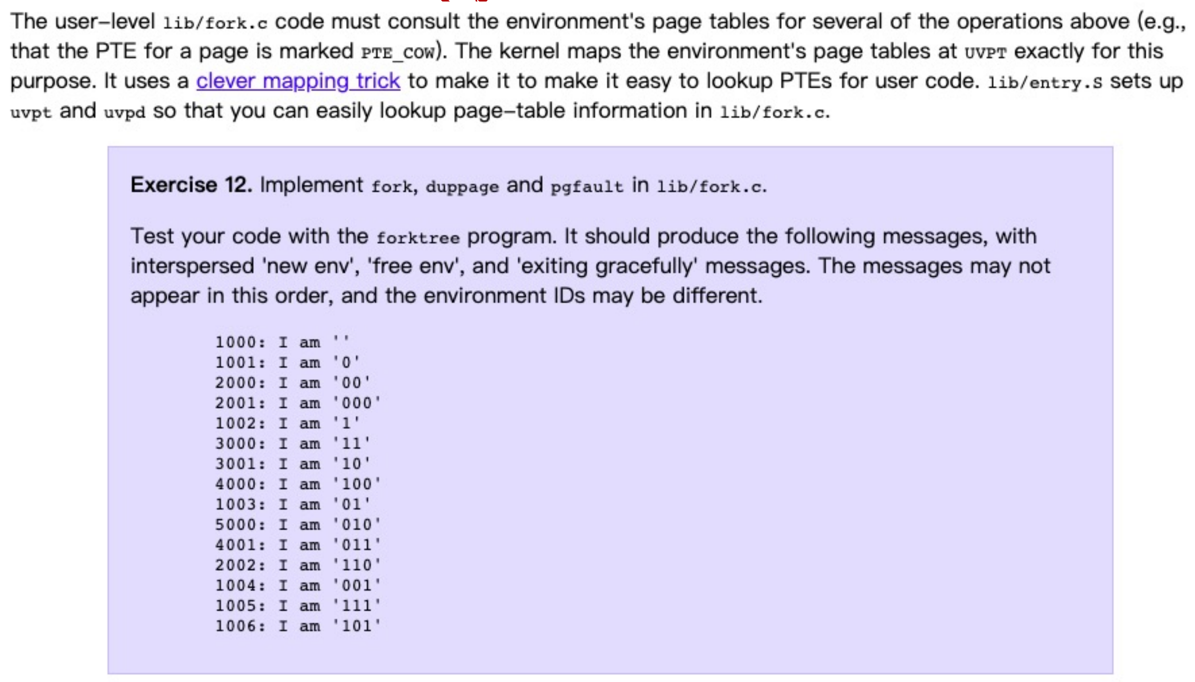
   The exception stack is *not* remapped this way, however. Instead you need to allocate a fresh page in the child for the exception stack. Since the page fault handler will be doing the actual copying and the page fault handler runs on the exception stack, the exception stack cannot be made copy-on-write: who would copy it?

   `fork()` also needs to handle pages that are present, but not writable or copy-on-write.

4. The parent sets the user page fault entrypoint for the child to look like its own.
5. The child is now ready to run, so the parent marks it runnable.

Figure 14: Control flow of user page fault handler

3.3.1 Exercise 12



The user-level `lib/fork.c` code must consult the environment's page tables for several of the operations above (e.g., that the PTE for a page is marked `PTE_COW`). The kernel maps the environment's page tables at `UVPT` exactly for this purpose. It uses a clever mapping trick to make it to make it easy to lookup PTEs for user code. `lib/entry.S` sets up `uvpt` and `uvpd` so that you can easily lookup page-table information in `lib/fork.c`.

Exercise 12. Implement `fork`, `duppage` and `pgfault` in `lib/fork.c`.

Test your code with the `forktree` program. It should produce the following messages, with interspersed 'new env', 'free env', and 'exiting gracefully' messages. The messages may not appear in this order, and the environment IDs may be different.

```
1000: I am ''
1001: I am '0'
2000: I am '00'
2001: I am '000'
1002: I am '1'
3000: I am '11'
3001: I am '10'
4000: I am '100'
1003: I am '01'
5000: I am '010'
4001: I am '011'
2002: I am '110'
1004: I am '001'
1005: I am '111'
1006: I am '101'
```

Figure 15: Exercise 12

3.3.2 Clever

3.3.3 The Implementation of pgfault()

3.3.3.1 Its Hints

```c
// Custom page fault handler - if faulting page is copy-on-write,
// map in our own private writable copy.
//
static void
pgfault(struct UTrapframe *utf)
{
        void *addr = (void *) utf->utf_fault_va;
        uint32_t err = utf->utf_err;
        int r;

        // Check that the faulting access was (1) a write, and (2) to a
        // copy-on-write page.  If not, panic.
        // Hint:
        //   Use the read-only page table mappings at uvpt
        //   (see <inc/memlayout.h>).

        // LAB 4: Your code here.

        // Allocate a new page, map it at a temporary location (PFTEMP),
        // copy the data from the old page to the new page, then move the new
        // page to the old page's address.
        // Hint:
        //   You should make three system calls.

        // LAB 4: Your code here.

        panic("pgfault not implemented");
}
```

3.3.3.2 Its Implementation

```c
static void
pgfault(struct UTrapframe *utf)
{
        void *addr = (void *) utf->utf_fault_va;
        uint32_t err = utf->utf_err;
        int r;
        // LAB 4: Your code here.
        // Check that the faulting access was
        // (1). a write
        // (2). to a copy-on-write page.
        if((err & FEC_WR) == 0 || (uvpt[PGNUM(addr)] & PTE_COW) == 0)
                panic("pgfault failed: this page is not writable or attempt to access a non-cow page!");
        envid_t envid = sys_getenvid();
        if((r = sys_page_alloc(envid, (void *)PFTEMP, PTE_P|PTE_W|PTE_U)) < 0)
                panic("pgfault failed: page allocation failed!");

        addr = ROUNDDOWN(addr, PGSIZE);
        memmove(PFTEMP, addr, PGSIZE);
        if ((r = sys_page_unmap(envid, addr)) < 0)
                panic("pgfault failed: page unmap failed: %e", r);
        if ((r = sys_page_map(envid, PFTEMP, envid, addr, PTE_P | PTE_W | PTE_U)) < 0)
                panic("pgfault failed: page map failed: %e", r);
        if ((r = sys_page_unmap(envid, PFTEMP)) < 0)
                panic("pgfault failed: page map failed: %e", r);
        //panic("pgfault not implemented");

}
```

3.3.4 The Implementation of duppage()

3.3.4.1 See its hints

```
// Map our virtual page pn (address pn*PGSIZE) into the target envid
// at the same virtual address.  If the page is writable or copy-on-write,
// the new mapping must be created copy-on-write, and then our mapping must be
// marked copy-on-write as well.  (Exercise: Why do we need to mark ours
// copy-on-write again if it was already copy-on-write at the beginning of
// this function?)
//
// Returns: 0 on success, < 0 on error.
// It is also OK to panic on error.
//
static int
duppage(envid_t envid, unsigned pn)
{
        int r;

        // LAB 4: Your code here.
        panic("duppage not implemented");
        return 0;
}
```

3.3.4.2 See its Implementation

```
static int
duppage(envid_t envid, unsigned pn)
{
        int r;

        // LAB 4: Your code here.
        void *addr;
        pte_t pte;
        int perm;

        addr = (void *)((uint32_t)pn * PGSIZE);
        pte = uvpt[pn];
        perm = PTE_P | PTE_U;
        if ((pte & PTE_W) || (pte & PTE_COW))
                perm |= PTE_COW;
        if ((r = sys_page_map(thisenv->env_id, addr, envid, addr, perm)) < 0) {
                panic("duppage failed: page remapping failed: %e", r);
                return r;
        }
        if (perm & PTE_COW) {
                if ((r = sys_page_map(thisenv->env_id, addr, thisenv->env_id, addr,
      ↪ perm)) < 0) {
                        panic("duppage failed: page remapping failed: %e", r);
                        return r;
                }
        }
        //panic("duppage not implemented");
        return 0;
}
```

3.3.5 User-Level Fork with Copy-on-Write Implementation

3.3.5.1 See its Hints

```
//
// User-level fork with copy-on-write.
// Set up our page fault handler appropriately.
// Create a child.
// Copy our address space and page fault handler setup to the child.
// Then mark the child as runnable and return.
//
// Returns: child's envid to the parent, 0 to the child, < 0 on error.
// It is also OK to panic on error.
//
// Hint:
//   Use uvpd, uvpt, and duppage.
//   Remember to fix "thisenv" in the child process.
//   Neither user exception stack should ever be marked copy-on-write,
//   so you must allocate a new page for the child's user exception stack.
//
envid_t
fork(void)
{
        // LAB 4: Your code here.
        panic("fork not implemented");
}
```

3.3.5.2 See its Implementation

```
envid_t
fork(void)
{
        // LAB 4: Your code here.
        envid_t childenvid;
        uint32_t addr;
        int i, j, pn, r;
        extern void _pgfault_upcall(void);

        // (1). Install pgfault() as the C-level page fault handler.
        set_pgfault_handler(pgfault);

        // (2). The parent calls sys_exofork() to create a child environment
        if ((childenvid = sys_exofork()) < 0) {
                panic("sys_exofrok failed: %e", childenvid);
                return childenvid;
        }
        if(childenvid == 0) {
                // child environment
                thisenv = &envs[ENVX(sys_getenvid())];
                return 0;
        }
        // (3). For each writable or copy-on-write page in its address space before
        ↪ UTOP, the parent calls duppage, which should
        //      (3.1) map the page copy-on-write into the address space of the child.
        //      (3.2) then, remap the page copy-on-write into its own address space.
        ↪ Mind the order of (3.1) and (3.2)
        // Following is parent environment operation.

        //      (3.1) Copy address space and page fault handler setup
        //            to the child.
        for (pn = PGNUM(UTEXT); pn < PGNUM(USTACKTOP); pn++) {
                if((uvpd[pn >> 10] & PTE_P) && (uvpt[pn] & PTE_P)) {
                        if ((r = duppage(childenvid, pn)) < 0)
                                return r;
                }
        }

        // allocate a page for child exception stack
        if ((r = sys_page_alloc(childenvid, (void *)(UXSTACKTOP-PGSIZE), PTE_U | PTE_P
        ↪ | PTE_W)) < 0)
                return r;
        // (4) The parent sets the user page fault entrypoint for
        //      the child to look like its own.
        if ((r = sys_env_set_pgfault_upcall(childenvid, _pgfault_upcall)) < 0)
                return r;
        // (5). The child is now ready to run, so the parent marks it runnable
        if ((r = sys_env_set_status(childenvid, ENV_RUNNABLE)) < 0)
                panic("fork failed: sys_env_set_status failed: %e", r);
        return childenvid;

        //panic("fork not implemented");
}
```

3.4 Grade

Now, we have grades:
Part A: 5/5. Part B: 50. Part C: 0/25.

4 Preemptive Multitasking and Inter-Process communication (IPC)

In this part, two objectives:

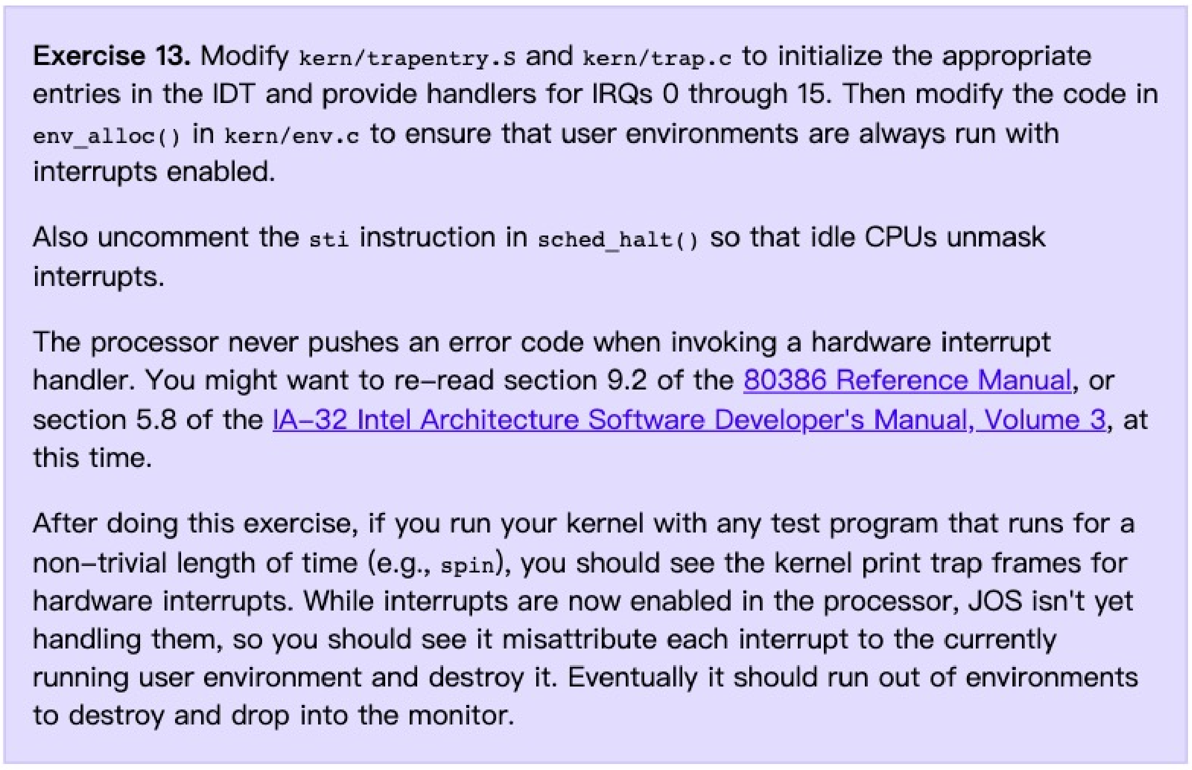
- Modify the kernel to preempt uncooperative environments.
- Allow environments to pass messages to each other explicitly.

4.1 Clock Interrupts and Preemption

In order to allow the kernel to **preempt** a running environment, forcefully retaking control of the CPU from this running environment, we must extend the JOS kernel to support external hardware interrupts from the clock hardware.

4.1.1 Interrupt Discipline

4.1.2 Exercise 13



Exercise 13. Modify `kern/trapentry.S` and `kern/trap.c` to initialize the appropriate entries in the IDT and provide handlers for IRQs 0 through 15. Then modify the code in `env_alloc()` in `kern/env.c` to ensure that user environments are always run with interrupts enabled.

Also uncomment the `sti` instruction in `sched_halt()` so that idle CPUs unmask interrupts.

The processor never pushes an error code when invoking a hardware interrupt handler. You might want to re-read section 9.2 of the 80386 Reference Manual, or section 5.8 of the IA-32 Intel Architecture Software Developer's Manual, Volume 3, at this time.

After doing this exercise, if you run your kernel with any test program that runs for a non-trivial length of time (e.g., `spin`), you should see the kernel print trap frames for hardware interrupts. While interrupts are now enabled in the processor, JOS isn't yet handling them, so you should see it misattribute each interrupt to the currently running user environment and destroy it. Eventually it should run out of environments to destroy and drop into the monitor.

Figure 16: Exercise 13

4.1.2.1 Declaration and set SETGATE in trap_init() in kern/trap.c

```
        // IRQs for LAB 4
        void IRQ0();
        SETGATE(idt[IRQ_OFFSET + 0], 0, GD_KT, IRQ0, 0);
        void IRQ1();
        SETGATE(idt[IRQ_OFFSET + 1], 0, GD_KT, IRQ1, 0);
        void IRQ2();
        SETGATE(idt[IRQ_OFFSET + 2], 0, GD_KT, IRQ2, 0);
        void IRQ3();
        SETGATE(idt[IRQ_OFFSET + 3], 0, GD_KT, IRQ3, 0);
        void IRQ4();
        SETGATE(idt[IRQ_OFFSET + 4], 0, GD_KT, IRQ4, 0);
        void IRQ5();
        SETGATE(idt[IRQ_OFFSET + 5], 0, GD_KT, IRQ5, 0);
        void IRQ6();
        SETGATE(idt[IRQ_OFFSET + 6], 0, GD_KT, IRQ6, 0);
        void IRQ7();
        SETGATE(idt[IRQ_OFFSET + 7], 0, GD_KT, IRQ7, 0);
        void IRQ8();
        SETGATE(idt[IRQ_OFFSET + 8], 0, GD_KT, IRQ8, 0);
        void IRQ9();
        SETGATE(idt[IRQ_OFFSET + 9], 0, GD_KT, IRQ9, 0);
        void IRQ10();
        SETGATE(idt[IRQ_OFFSET + 10], 0, GD_KT, IRQ10, 0);
        void IRQ11();
        SETGATE(idt[IRQ_OFFSET + 11], 0, GD_KT, IRQ11, 0);
        void IRQ12();
        SETGATE(idt[IRQ_OFFSET + 12], 0, GD_KT, IRQ12, 0);
        void IRQ13();
        SETGATE(idt[IRQ_OFFSET + 13], 0, GD_KT, IRQ13, 0);
        void IRQ14();
        SETGATE(idt[IRQ_OFFSET + 14], 0, GD_KT, IRQ14, 0);
        void IRQ15();
        SETGATE(idt[IRQ_OFFSET + 15], 0, GD_KT, IRQ15, 0);
```

4.1.2.2 Generating entry points for IRQs in kern/trapentry.S

```
// LAB 4
// IRQs
// LAB 4
// IRQs
TRAPHANDLER_NOEC(IRQ0, IRQ_OFFSET+0);
TRAPHANDLER_NOEC(IRQ1, IRQ_OFFSET+1);
TRAPHANDLER_NOEC(IRQ2, IRQ_OFFSET+2);
TRAPHANDLER_NOEC(IRQ3, IRQ_OFFSET+3);
TRAPHANDLER_NOEC(IRQ4, IRQ_OFFSET+4);
TRAPHANDLER_NOEC(IRQ5, IRQ_OFFSET+5);
TRAPHANDLER_NOEC(IRQ6, IRQ_OFFSET+6);
TRAPHANDLER_NOEC(IRQ7, IRQ_OFFSET+7);
TRAPHANDLER_NOEC(IRQ8, IRQ_OFFSET+8);
TRAPHANDLER_NOEC(IRQ9, IRQ_OFFSET+9);
TRAPHANDLER_NOEC(IRQ10, IRQ_OFFSET+10);
TRAPHANDLER_NOEC(IRQ11, IRQ_OFFSET+11);
TRAPHANDLER_NOEC(IRQ12, IRQ_OFFSET+12);
TRAPHANDLER_NOEC(IRQ13, IRQ_OFFSET+13);
TRAPHANDLER_NOEC(IRQ14, IRQ_OFFSET+14);
TRAPHANDLER_NOEC(IRQ15, IRQ_OFFSET+15);
```

4.1.2.3 Modify env_alloc() in kern/env.c to ensure that user environment are always run with interrupts enabled.

```
// Allocates and initializes a new environment.
// On success, the new environment is stored in *newenv_store.
//
// Returns 0 on success, < 0 on failure.  Errors include:
//      -E_NO_FREE_ENV if all NENV environments are allocated
//      -E_NO_MEM on memory exhaustion
//
int
env_alloc(struct Env **newenv_store, envid_t parent_id)
{
        int32_t generation;
        int r;
        struct Env *e;

        if (!(e = env_free_list))
                return -E_NO_FREE_ENV;

        // Allocate and set up the page directory for this environment.
        if ((r = env_setup_vm(e)) < 0)
                return r;

        // Generate an env_id for this environment.
        generation = (e->env_id + (1 << ENVGENSHIFT)) & ~(NENV - 1);
        if (generation <= 0)    // Don't create a negative env_id.
                generation = 1 << ENVGENSHIFT;
        e->env_id = generation | (e - envs);

        // Set the basic status variables.
        e->env_parent_id = parent_id;
        e->env_type = ENV_TYPE_USER;
        e->env_status = ENV_RUNNABLE;
        e->env_runs = 0;
        // Clear out all the saved register state,
        // to prevent the register values
        // of a prior environment inhabiting this Env structure
        // from "leaking" into our new environment.
        memset(&e->env_tf, 0, sizeof(e->env_tf));

        // Set up appropriate initial values for the segment registers.
        // GD_UD is the user data segment selector in the GDT, and
        // GD_UT is the user text segment selector (see inc/memlayout.h).
        // The low 2 bits of each segment register contains the
        // Requestor Privilege Level (RPL); 3 means user mode.  When
        // we switch privilege levels, the hardware does various
        // checks involving the RPL and the Descriptor Privilege Level
        // (DPL) stored in the descriptors themselves.
        e->env_tf.tf_ds = GD_UD | 3;
        e->env_tf.tf_es = GD_UD | 3;
        e->env_tf.tf_ss = GD_UD | 3;
        e->env_tf.tf_esp = USTACKTOP;
        e->env_tf.tf_cs = GD_UT | 3;
        // You will set e->env_tf.tf_eip later.

        // Enable interrupts while in user mode.
        // LAB 4: Your code here.
        e->env_tf.tf_eflags |= FL_IF;

        // Clear the page fault handler until user installs one.
        e->env_pgfault_upcall = 0;

        // Also clear the IPC receiving flag.
        e->env_ipc_recving = 0;

        // commit the allocation
        env_free_list = e->env_link;
        *newenv_store = e;

        cprintf("[%08x] new env %08x\n", curenv ? curenv->env_id : 0, e->env_id);
        return 0;
}
```

Just add one line:

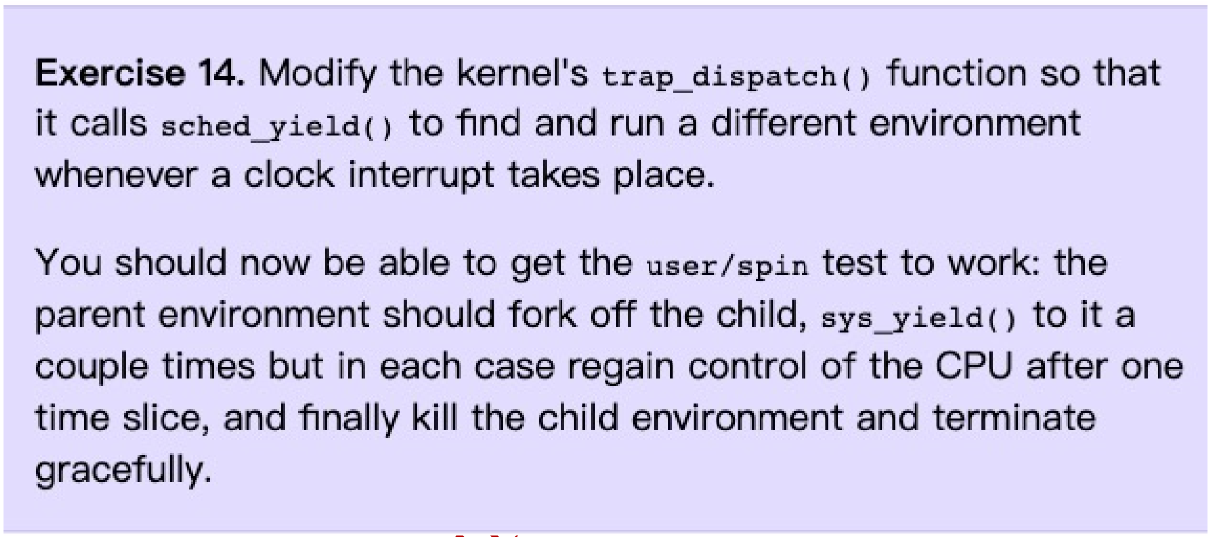
```
        // Enable interrupts while in user mode.
        // LAB 4: Your code here.
        e->env_tf.tf_eflags |= FL_IF;
```

4.1.2.4 DO NOT FORGET COMMENT the sti in sched_halt() in kern/sched.c

```
// Halt this CPU when there is nothing to do. Wait until the
// timer interrupt wakes it up. This function never returns.
//
void
sched_halt(void)
{
        int i;

        // For debugging and testing purposes, if there are no runnable
        // environments in the system, then drop into the kernel monitor.
        for (i = 0; i < NENV; i++) {
                if ((envs[i].env_status == ENV_RUNNABLE ||
                     envs[i].env_status == ENV_RUNNING ||
                     envs[i].env_status == ENV_DYING))
                        break;
        }
        if (i == NENV) {
                cprintf("No runnable environments in the system!\n");
                while (1)
                        monitor(NULL);
        }

        // Mark that no environment is running on this CPU
        curenv = NULL;
        lcr3(PADDR(kern_pgdir));

        // Mark that this CPU is in the HALT state, so that when
        // timer interrupts come in, we know we should re-acquire the
        // big kernel lock
        // big kernel lock
        xchg(&thiscpu->cpu_status, CPU_HALTED);

        // Release the big kernel lock as if we were "leaving" the kernel
        unlock_kernel();

        // Reset stack pointer, enable interrupts and then halt.
        asm volatile (
                "movl $0, %%ebp\n"
                "movl %0, %%esp\n"
                "pushl $0\n"
                "pushl $0\n"
                // Uncomment the following line after completing exercise 13
                //"sti\n"
                "1:\n"
                "hlt\n"
                "jmp 1b\n"
        : : "a" (thiscpu->cpu_ts.ts_esp0));
}
```

4.1.3 Handling Clock Interrupts

4.1.4 Exercise 14



Exercise 14. Modify the kernel's `trap_dispatch()` function so that it calls `sched_yield()` to find and run a different environment whenever a clock interrupt takes place.

You should now be able to get the `user/spin` test to work: the parent environment should fork off the child, `sys_yield()` to it a couple times but in each case regain control of the CPU after one time slice, and finally kill the child environment and terminate gracefully.

Figure 17: Exercise 14

```
static void
trap_dispatch(struct Trapframe *tf)
{
        // Handle spurious interrupts
        // The hardware sometimes raises these because of noise on the
        // IRQ line or other reasons. We don't care.
        if (tf->tf_trapno == IRQ_OFFSET + IRQ_SPURIOUS) {
                cprintf("Spurious interrupt on irq 7\n");
                print_trapframe(tf);
                return;
        }
        // Handle processor exceptions.
        // LAB 3: Your code here.
        switch(tf->tf_trapno) {
                case T_PGFLT:
                        page_fault_handler(tf);
                        return;
                case T_BRKPT:
                        monitor(tf);
                        return;
                case T_SYSCALL:
                        tf->tf_regs.reg_eax = syscall(tf->tf_regs.reg_eax,
                                tf->tf_regs.reg_edx,
                                tf->tf_regs.reg_ecx, tf->tf_regs.reg_ebx,
                                tf->tf_regs.reg_edi, tf->tf_regs.reg_esi);
                        return;
                // Handle clock interrupts. Don't forget to acknowledge the
                // interrupt using lapic_eoi() before calling the scheduler!
                // LAB 4: Your code here.
                case IRQ_OFFSET + IRQ_TIMER:
                        lapic_eoi();
                        sched_yield();
                        return;
                default:
                        // Unexpected trap: The user process or the kernel has a bug.
                        print_trapframe(tf);
                        if (tf->tf_cs == GD_KT)
                                panic("unhandled trap in kernel");
                        else {
                                env_destroy(curenv);
                                return;
                        }
                        break;
        }
}
```

4.1.5 Present Score

Now:

Part	Score
Part A	5/5
Part B	50/50
Part C	10/25
Final	65/80

4.2 Inter-Process communication (IPC)

4.2.1 Background Knowledge about IPC

- We've been focusing on the isolation aspects of the operating system, the ways it provides the illusion that each program has a machine all to itself;
- Another important service of an operating system is to allow programs to communicate with each other when they want to.

The Unix pipe model is the canonical example.

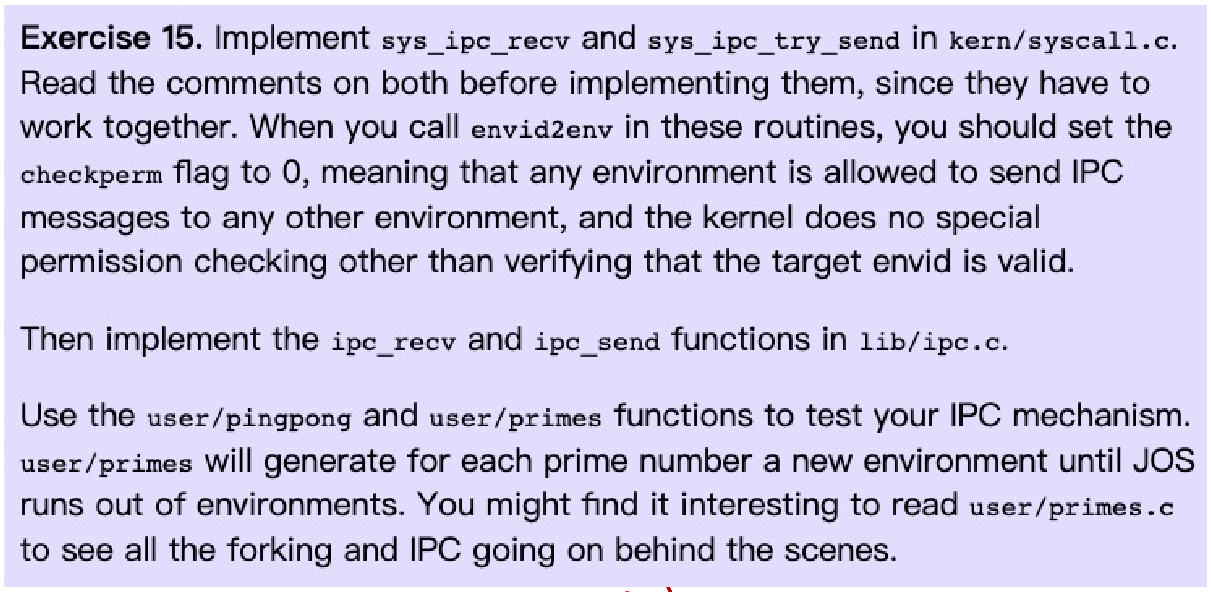
You will implement two system calls, sys_ipc_recv and sys_ipc_try_send. Then you will implement two library wrappers ipc_recv and ipc_send.

The *message* that user environments can send consists of two components:

- A single 32-bit value.

- Optionally a single page mapping. Allowing environments to pass page mappings in messages provides an efficient way to transfer more data than will fit into a single 32-bit integer, and also allows environments to set up shared memory arrangements easily.

4.2.2 Exercise 15



Exercise 15. Implement `sys_ipc_recv` and `sys_ipc_try_send` in `kern/syscall.c`. Read the comments on both before implementing them, since they have to work together. When you call `envid2env` in these routines, you should set the `checkperm` flag to 0, meaning that any environment is allowed to send IPC messages to any other environment, and the kernel does no special permission checking other than verifying that the target envid is valid.

Then implement the `ipc_recv` and `ipc_send` functions in `lib/ipc.c`.

Use the `user/pingpong` and `user/primes` functions to test your IPC mechanism. `user/primes` will generate for each prime number a new environment until JOS runs out of environments. You might find it interesting to read `user/primes.c` to see all the forking and IPC going on behind the scenes.

Figure 18: Exercise 15

4.2.3 The Hints in kern/syscall.c

```
// Try to send 'value' to the target env 'envid'.
// If srcva < UTOP, then also send page currently mapped at 'srcva',
// so that receiver gets a duplicate mapping of the same page.
//
// The send fails with a return value of -E_IPC_NOT_RECV if the
// target is not blocked, waiting for an IPC.
//
// The send also can fail for the other reasons listed below.
//
// Otherwise, the send succeeds, and the target's ipc fields are
// updated as follows:
//    env_ipc_recving is set to 0 to block future sends;
//    env_ipc_from is set to the sending envid;
//    env_ipc_value is set to the 'value' parameter;
//    env_ipc_perm is set to 'perm' if a page was transferred, 0 otherwise.
// The target environment is marked runnable again, returning 0
// from the paused sys_ipc_recv system call.  (Hint: does the
// sys_ipc_recv function ever actually return?)
//
// If the sender wants to send a page but the receiver isn't asking for one,
// then no page mapping is transferred, but no error occurs.
// The ipc only happens when no errors occur.
//
// Returns 0 on success, < 0 on error.
```

```c
// Errors are:
//      -E_BAD_ENV if environment envid doesn't currently exist.
//              (No need to check permissions.)
//      -E_IPC_NOT_RECV if envid is not currently blocked in sys_ipc_recv,
//              or another environment managed to send first.
//      -E_INVAL if srcva < UTOP but srcva is not page-aligned.
//      -E_INVAL if srcva < UTOP and perm is inappropriate
//              (see sys_page_alloc).
//      -E_INVAL if srcva < UTOP but srcva is not mapped in the caller's
//              address space.
//      -E_INVAL if (perm & PTE_W), but srcva is read-only in the
//              current environment's address space.
//      -E_NO_MEM if there's not enough memory to map srcva in envid's
//              address space.
static int
sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm)
{
        // LAB 4: Your code here.
        panic("sys_ipc_try_send not implemented");
}

// Block until a value is ready.  Record that you want to receive
// using the env_ipc_recving and env_ipc_dstva fields of struct Env,
// mark yourself not runnable, and then give up the CPU.
//
// If 'dstva' is < UTOP, then you are willing to receive a page of data.
// 'dstva' is the virtual address at which the sent page should be mapped.
//
// This function only returns on error, but the system call will eventually
// return 0 on success.
// Return < 0 on error.  Errors are:
//      -E_INVAL if dstva < UTOP but dstva is not page-aligned.
static int
sys_ipc_recv(void *dstva)
{
        // LAB 4: Your code here.
        panic("sys_ipc_recv not implemented");
        return 0;
}
```

4.2.4 Sending and Receiving Message

4.2.4.1 To receive a message

To receive a message, an environment calls sys_ipc_recv. This system call **de-schedules** the current environment and does not run it again until a message has been received.

Here is a safe technique for **malfunction**:
When an environment is waiting to receive a message, **any other** environment can send it a message. In this *safe* approach, an environment cannot cause another environment to malfunction simply by sending a message.

4.2.4.2 To try to send a value

To try to send a value, an environment calls sys_ipc_try_send with the {the receiver environment id, and the value to be sent}. Then they are two cases in this situation:

- If the named environment is actually receiving, then the sender delivers the message and return 0.
- Otherwise, the send returns -E_IPC_NOT_RECV to indicate the target environment is not currently expecting to receive a value.

4.2.5 Transferring Pages

4.2.5.1 Receive a page mapping

When an environment calls sys_ipc_recv with a valid dstva(below UTOP) parameter, the environment is stating that it is willing to receive a page mapping. If the sender sends a page, then that page should be mapped at the dstva in the receiver's address space. If the receiver already had a page mapped at dstva, then that previous page is unmapped.

4.2.5.2 Send a page mapping

When an environment calls sys_ipc_try_send with a valid srcva, it means that the sender wants to send the page currently mapped at srcva to the receiver, with permission perm. After a successful IPC:

- The sender keeps its original mapping for the page at srcva in its address space.
- The receiver also obtains a mapping for this same physical page at the dstva specified by the receiver in the receiver's address space.
- This physical page is shared between the sender and the receiver.

If either the sender or the receiver does not indicate that a page should be transferred, then no page is transferred. After any IPC the kernel sets the new field env_ipc_perm in the receiver's Env structure to the permissions of the page received or 0 if no page was received.

4.2.6 The Implementation of sys_ipc_try_send() in kern/syscall.c

4.2.6.1 Let's see its hints

```
// Try to send 'value' to the target env 'envid'.
// If srcva < UTOP, then also send page currently mapped at 'srcva',
// so that receiver gets a duplicate mapping of the same page.
//
// The send fails with a return value of -E_IPC_NOT_RECV if the
// target is not blocked, waiting for an IPC.
//
// The send also can fail for the other reasons listed below.
//
// Otherwise, the send succeeds, and the target's ipc fields are
// updated as follows:
//    env_ipc_recving is set to 0 to block future sends;
//    env_ipc_from is set to the sending envid;
//    env_ipc_value is set to the 'value' parameter;
//    env_ipc_perm is set to 'perm' if a page was transferred, 0 otherwise.
// The target environment is marked runnable again, returning 0
// from the paused sys_ipc_recv system call.  (Hint: does the
```

```c
// sys_ipc_recv function ever actually return?)
//
// If the sender wants to send a page but the receiver isn't asking for one,
// then no page mapping is transferred, but no error occurs.
// The ipc only happens when no errors occur.
//
// Returns 0 on success, < 0 on error.
// Errors are:
//      -E_BAD_ENV if environment envid doesn't currently exist.
//              (No need to check permissions.)
//      -E_IPC_NOT_RECV if envid is not currently blocked in sys_ipc_recv,
//              or another environment managed to send first.
//      -E_INVAL if srcva < UTOP but srcva is not page-aligned.
//      -E_INVAL if srcva < UTOP and perm is inappropriate
//              (see sys_page_alloc).
//      -E_INVAL if srcva < UTOP but srcva is not mapped in the caller's
//              address space.
//      -E_INVAL if (perm & PTE_W), but srcva is read-only in the
//              current environment's address space.
//      -E_NO_MEM if there's not enough memory to map srcva in envid's
//              address space.
static int
sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm)
{
        // LAB 4: Your code here.
        panic("sys_ipc_try_send not implemented");
}
```

4.2.6.2 See the Implementation

```c
static int
sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm)
{
        // LAB 4: Your code here.
        struct Env *dstenv;
        pte_t *pte;
        struct PageInfo *pp;
        int r;
        // environment envid doesn't currently exist.
        if ((r = envid2env(envid, &dstenv, 0)) < 0)
                return -E_BAD_ENV;
        //if envid is not currently blocked in sys_ipc_recv
        if((dstenv->env_ipc_recving != true) || (dstenv->env_ipc_from != 0))
                return -E_IPC_NOT_RECV;
        // if srcva < UTOP but srcva is not page-aligned
        if ((srcva < (void *)UTOP) && (PGOFF(srcva)))
                return -E_INVAL;
        // if srcva < UTOP but perm is inappropriate
        if (srcva < (void *)UTOP) {
                // perm inappropriate
                if(((perm & PTE_P) == 0) || ((perm & PTE_U) == 0))
                        return -E_INVAL;
                if ((perm & ~(PTE_P | PTE_U | PTE_W | PTE_AVAIL)) != 0)
                        return -E_INVAL;
                // no page mapped at srcva
                if ((pp = page_lookup(curenv->env_pgdir, srcva, &pte)) == NULL)
                        return -E_INVAL;
                // (perm & PTE_W), but srcva is read-only
                if(((perm & PTE_W) != 0) && ((*pte & PTE_W) == 0))
                        return -E_INVAL;
                if (dstenv->env_ipc_dstva != 0) {
                        // there is no enough memory to map srcva
                        // in envid's address space.
                        if ((r = page_insert(dstenv->env_pgdir, pp, dstenv->
    ↪ env_ipc_dstva, perm)) < 0)
                                return -E_NO_MEM;
                        dstenv->env_ipc_perm = perm;
                }

        }
        dstenv->env_ipc_from = curenv->env_id;
        dstenv->env_ipc_recving = false;
        dstenv->env_ipc_value = value;
        dstenv->env_status = ENV_RUNNABLE;
        dstenv->env_tf.tf_regs.reg_eax = 0;
        return 0;

        //panic("sys_ipc_try_send not implemented");
}
```

4.2.7 The Implementation of sys_ipc_recv() in kern/syscall.c

4.2.7.1 See its hints

```c
// Block until a value is ready.  Record that you want to receive
// using the env_ipc_recving and env_ipc_dstva fields of struct Env,
// mark yourself not runnable, and then give up the CPU.
//
// If 'dstva' is < UTOP, then you are willing to receive a page of data.
// 'dstva' is the virtual address at which the sent page should be mapped.
//
// This function only returns on error, but the system call will eventually
// return 0 on success.
// Return < 0 on error.  Errors are:
//      -E_INVAL if dstva < UTOP but dstva is not page-aligned.
static int
sys_ipc_recv(void *dstva)
{
        // LAB 4: Your code here.
        panic("sys_ipc_recv not implemented");
        return 0;
}
```

4.2.7.2 See the implementation

```c
static int
sys_ipc_recv(void *dstva)
{
        // LAB 4: Your code here.
        if (dstva < (void *)UTOP && PGOFF(dstva))
                return -E_INVAL;
        // Record that you want to receive using env_ipc_recving
        // and env_ipc_dstva fields of struct Env
        curenv->env_ipc_recving = true;
        curenv->env_ipc_dstva = dstva;
        // mark yourself not runnable.
        curenv->env_status = ENV_NOT_RUNNABLE;
        curenv->env_ipc_from = 0;
        // Give up CPU
        sched_yield();
        return 0;
        //panic("sys_ipc_recv not implemented");
}
```

4.2.8 DO NOT FORGET REGISTER IN syscall() in kern/syscall.c

```
        // LAB 4: sys_ipc_recv and sys_ipc_try_send
        case SYS_ipc_recv:
                return sys_ipc_recv((void *)a1);
        case SYS_ipc_try_send:
                return sys_ipc_try_send((envid_t)a1, (uint32_t)a2, (void *)a3, (
    ↪ unsigned)a4);
```

4.2.9 The Implementation of ipc_recv() in kern/ipc.c

4.2.9.1 Firstly, see its hints

```
// Receive a value via IPC and return it.
// If 'pg' is nonnull, then any page sent by the sender will be mapped at
//      that address.
// If 'from_env_store' is nonnull, then store the IPC sender's envid in
//      *from_env_store.
// If 'perm_store' is nonnull, then store the IPC sender's page permission
//      in *perm_store (this is nonzero iff a page was successfully
//      transferred to 'pg').
// If the system call fails, then store 0 in *fromenv and *perm (if
//      they're nonnull) and return the error.
// Otherwise, return the value sent by the sender
//
// Hint:
//   Use 'thisenv' to discover the value and who sent it.
//   If 'pg' is null, pass sys_ipc_recv a value that it will understand
//   as meaning "no page".  (Zero is not the right value, since that's
//   a perfectly valid place to map a page.)
int32_t
ipc_recv(envid_t *from_env_store, void *pg, int *perm_store)
{
        // LAB 4: Your code here.
        panic("ipc_recv not implemented");
        return 0;
}
```

4.2.9.2 See its implementation

```
int32_t
ipc_recv(envid_t *from_env_store, void *pg, int *perm_store)
{
        // LAB 4: Your code here.
        int r;
        if (pg == NULL)
                pg = (void *)UTOP;
        r = sys_ipc_recv(pg);
        // If 'from_env_store' is nonnull, then store the IPC
        // sender's envid in *from_env_store
        if (from_env_store != NULL)
                *from_env_store = r < 0 ? 0 : thisenv->env_ipc_
from;
        // If 'perm store' is nonnull, then store the IPC sende
r's
        // page permission in *perm_store
        if (perm_store != NULL)
                *perm_store = r < 0 ? 0 : thisenv->env_ipc_perm
;

        // If the system call fails, store 0 in *fromenv and
        // *perm if they're nonnull and return the error.
        // Otherwise, return the value sent by the sender
        if (r < 0)
                return r;
        else
                return thisenv->env_ipc_value;
        //panic("ipc_recv not implemented");
        //return 0;
}
```

4.2.10 The Implementation of ipc_send() in kern/ipc.c

4.2.10.1 Hints of ipc_send()

```
// Send 'val' (and 'pg' with 'perm', if 'pg' is nonnull) to 'toenv'.
// This function keeps trying until it succeeds.
// It should panic() on any error other than -E_IPC_NOT_RECV.
//
// Hint:
//   Use sys_yield() to be CPU-friendly.
//   If 'pg' is null, pass sys_ipc_try_send a value that it will understand
//   as meaning "no page".  (Zero is not the right value.)
void
ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)
{
        // LAB 4: Your code here.
        panic("ipc_send not implemented");
}
```

4.2.10.2 See its implementation

```
void
ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)
{
        // LAB 4: Your code here.
        int r;
        void *dstpg;

        dstpg = (pg == NULL ? (void *)UTOP : pg);
        // Keep trying unitl it succeeds.
        while ((r = sys_ipc_try_send(to_env, val, dstpg, perm)) < 0) {
                if (r != -E_IPC_NOT_RECV)
                        panic("ipc_send failed: error other than -E_IPC_NOT_RECV
    ↪ happened: %e", r);
                // Use sys_yield() to be CPU-friendly
                sys_yield();
        }
        //panic("ipc_send not implemented");
}
```

4.3 Run user/pingpong and user/primes to Test IPC

This result is useful for check the merge result of LAB 4 and LAB 5. Just see the result:

4.3.1 Run user/pingpong

4.3.1.1 See the code in pingpong.c

```c
// Ping-pong a counter between two processes.
// Only need to start one of these -- splits into two with fork.

#include <inc/lib.h>

void
umain(int argc, char **argv)
{
        envid_t who;

        if ((who = fork()) != 0) {
                // get the ball rolling
                cprintf("send 0 from %x to %x\n", sys_getenvid(), who);
                ipc_send(who, 0, 0, 0);
        }

        while (1) {
                uint32_t i = ipc_recv(&who, 0, 0);
                cprintf("%x got %d from %x\n", sys_getenvid(), i, who);
                if (i == 10)
                        return;
                i++;
                ipc_send(who, i, 0, 0);
                if (i == 10)
                        return;
        }
}
```

4.3.1.2 See the results

```
make[1]: Entering directory '/home/cui/mit6828/lab'
+ cc kern/init.c
+ cc[USER] user/pingpong.c
+ ld obj/user/pingpong
+ ld obj/kern/kernel
+ mk obj/kern/kernel.img
make[1]: Leaving directory '/home/cui/mit6828/lab'
qemu-system-i386 -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -serial
    ↪ mon:stdio -gdb tcp::26000 -D qemu.log -smp 1
6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
```

```
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2
[00000000] new env 00001000
[00001000] new env 00001001
send 0 from 1000 to 1001
1001 got 0 from 1000
1000 got 1 from 1001
1001 got 2 from 1000
1000 got 3 from 1001
1001 got 4 from 1000
1000 got 5 from 1001
1001 got 6 from 1000
1000 got 7 from 1001
1001 got 8 from 1000
1000 got 9 from 1001
[00001000] exiting gracefully
[00001000] free env 00001000
1001 got 10 from 1000
[00001001] exiting gracefully
[00001001] free env 00001001
No runnable environments in the system!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

4.3.2 Run user/primes

```
// Concurrent version of prime sieve of Eratosthenes.
// Invented by Doug McIlroy, inventor of Unix pipes.
// See http://swtch.com/~rsc/thread/.
// The picture halfway down the page and the text surrounding it
// explain what's going on here.
//
// Since NENV is 1024, we can print 1022 primes before running out.
// The remaining two environments are the integer generator at the bottom
// of main and user/idle.
#include <inc/lib.h>

unsigned
primeproc(void)
{
        int i, id, p;
        envid_t envid;

        // fetch a prime from our left neighbor
top:
        p = ipc_recv(&envid, 0, 0);
        cprintf("CPU %d: %d ", thisenv->env_cpunum, p);

        // fork a right neighbor to continue the chain
        if ((id = fork()) < 0)
                panic("fork: %e", id);
        if (id == 0)
                goto top;

        // filter out multiples of our prime
        while (1) {
                i = ipc_recv(&envid, 0, 0);
                if (i % p)
                        ipc_send(id, i, 0, 0);
        }
}

void
umain(int argc, char **argv)
{
        int i, id;

        // fork the first prime process in the chain
        if ((id = fork()) < 0)
                panic("fork: %e", id);
        if (id == 0)
                primeproc();

        // feed all the integers through
        for (i = 2; ; i++)
                ipc_send(id, i, 0, 0);
}
```

4.3.2.1 The results

```
make[1]: Entering directory '/home/cui/mit6828/lab'
make[1]: Leaving directory '/home/cui/mit6828/lab'
qemu-system-i386 -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -serial
    ↪ mon:stdio -gdb tcp::26000 -D qemu.log -smp 1
6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2
[00000000] new env 00001000
[00001000] new env 00001001
CPU 0: 2 [00001001] new env 00001002
CPU 0: 3 [00001002] new env 00001003
CPU 0: 5 [00001003] new env 00001004
CPU 0: 7 [00001004] new env 00001005
CPU 0: 11 [00001005] new env 00001006
CPU 0: 13 [00001006] new env 00001007
CPU 0: 17 [00001007] new env 00001008
CPU 0: 19 [00001008] new env 00001009
CPU 0: 23 [00001009] new env 0000100a
CPU 0: 29 [0000100a] new env 0000100b
CPU 0: 31 [0000100b] new env 0000100c
CPU 0: 37 [0000100c] new env 0000100d
CPU 0: 41 [0000100d] new env 0000100e
CPU 0: 43 [0000100e] new env 0000100f
CPU 0: 47 [0000100f] new env 00001010
CPU 0: 53 [00001010] new env 00001011
CPU 0: 59 [00001011] new env 00001012
CPU 0: 61 [00001012] new env 00001013
```

```
.................................................................
.................................................................
CPU 0: 8093 [000013fa] new env 000013fb
CPU 0: 8101 [000013fb] new env 000013fc
CPU 0: 8111 [000013fc] new env 000013fd
CPU 0: 8117 [000013fd] new env 000013fe
CPU 0: 8123 [000013fe] new env 000013ff
CPU 0: 8147 [000013ff] user panic in <unknown> at lib/fork.c:127: sys_exofrok failed:
    ↪ out of environments
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf02d3f84 from CPU 0
  edi  0x00000000
  esi  0x008016a3
  ebp  0xeebfdf30
  oesp 0xefffffdc
  ebx  0xeebfdf44
  edx  0xeebfddd8
  ecx  0x00000001
  eax  0x00000001
  es   0x----0023
  ds   0x----0023
  trap 0x00000003 Breakpoint
  err  0x00000000
  eip  0x008001f8
  cs   0x----001b
  flag 0x00000292
  esp  0xeebfdf08
  ss   0x----0023
K>
```

4.4 Final Score

Part	Score
Part A	5/5
Part B	50/50
Part C	25/25
Final	80/80

5 Reference

1. bysui's github and blog
   - github: https://github.com/bysui/mit6.828
   - blog: https://blog.csdn.net/bysui
2. SmallPond's github and blog
   - github: https://github.com/SmallPond/MIT6.828_OS
   - blog: https://me.csdn.net/Small_Pond
3. SimpCosm's github
   - github: https://github.com/SimpCosm/6.828
4. fatsheep9146's blog
   - blog: https://www.cnblogs.com/fatsheep9146/category/769143.html