

**LAB 4**

**PREEMPTIVE MULTITASKING**

**PART A AND PART B**

**AMAL JOY**

**CS11B006**

# Lab 4: Preemptive Multitasking

*cooperative* round-robin scheduling, allows the kernel to switch from one environment to another when the current environment voluntarily relinquishes the CPU (or exits). *preemptive* scheduling, which allows the kernel to re-take control of the CPU from an environment after a certain time has passed even if the environment does not cooperate.

"symmetric multiprocessing" (SMP) is a multiprocessor model in which all CPUs have equivalent access to system resources such as memory and I/O buses. While all CPUs are functionally identical in SMP, during the boot process they can be classified into two types: the bootstrap processor (BSP) is responsible for initializing the system and for booting the operating system; and the application processors (APs) are activated by the BSP only after the operating system is up and running.

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

A processor accesses its LAPIC using memory-mapped I/O (MMIO). In MMIO, a portion of *physical* memory is hardwired to the registers of some I/O devices, so the same load/store instructions typically used to access memory can be used to access device registers.

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.

```
// Reserve size bytes in the MMIO region and map [pa,pa+size) at this
// location. Return the base of the reserved region. size does *not*
// have to be multiple of PGSIZE.
//
void *
mmio_map_region(physaddr_t pa, size_t size)
{
    // Where to start the next region. Initially, this is the
    // beginning of the MMIO region. Because this is static, its
    // value will be preserved between calls to mmio_map_region
    // (just like nextfree in boot_alloc).
    static uintptr_t base = MMIOBASE;

    // Reserve size bytes of vvirtual memory starting at base and
    // map physical pages [pa,pa+size) to virtual addresses
    // [base,base+size). Since this is device memory and not
    // regular DRAM, you'll have to tell the CPU that it isn't
    // safe to cache access to this memory. Luckily, the page
    // tables provide bits for this purpose; simply create the
```

```

// mapping with PTE_PCD|PTE_PWT (cache-disable and
// write-through) in addition to PTE_W. (If you're interested
// in more details on this, see section 10.5 of IA32 volume
// 3A.)
//
// Be sure to round size up to a multiple of PGSIZE and to
// handle if this reservation would overflow MMIOLIM (it's
// okay to simply panic if this happens).
//
// Hint: The staff solution uses boot_map_region.
//
// Your code here:
void * ret = (void*)base;
size = ROUNDUP(size, PGSIZE);
if (base + size > MMIOLIM || base + size < base )
    panic("mmio_map_region : reservation overflow.\n");
boot_map_region(kern_pgdir , base, size, pa, PTE_PCD | PTE_PWT | PTE_W);
base += size;
return ret;
}

```

## Application Processor Bootstrap

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.

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).

```

void
page_init(void)
{
    // LAB 4:
    // Change your code to mark the physical page at MPENTRY_PADDR
    // as in use

    // The example code here marks all physical pages as free.
    // However this is not truly the case. What memory is free?
    // 1) Mark physical page 0 as in use.
    // This way we preserve the real-mode IDT and BIOS structures

```

```

// in case we ever need them. (Currently we don't, but...)
// 2) The rest of base memory, [PGSIZE, npages_basemem * PGSIZE)
// is free.
// 3) Then comes the IO hole [IOPHYSMEM, EXTPHYSMEM), which must
// never be allocated.
// 4) Then extended memory [EXTPHYSMEM, ...).
// Some of it is in use, some is free. Where is the kernel
// in physical memory? Which pages are already in use for
// page tables and other data structures?
//
// Change the code to reflect this.
// NB: DO NOT actually touch the physical memory corresponding to
// free pages!
size_t i;
//start i from 1 as 0 in use
//Sets [PGSIZE, npages_basemem * PGSIZE) as free.

size_t i_IOPHYSMEM_begin = ROUNDDOWN(IOPHYSMEM, PGSIZE) / PGSIZE;
size_t i_EXTPHYSMEM_end = PADDR(boot_alloc(0)) / PGSIZE;
size_t i_MPENTRY_PADDR = MPENTRY_PADDR / PGSIZE;
page_free_list = NULL;
for (i = 0; i < npages; i++) {
    pages[i].pp_ref = 0;
    if(i == 0) continue;
    if(i_IOPHYSMEM_begin <= i && i < i_EXTPHYSMEM_end) continue;
    if(i == i_MPENTRY_PADDR) continue;
    pages[i].pp_link = page_free_list;
    page_free_list = &pages[i];
}

```

#### 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.

ANS:

#define MPBOOTPHYS(s) ((s) - mpentry\_start + MPENTRY\_PADDR))).MPBOOTPHYS is to calculate symbol address relative to MPENTRY\_PADDR. The ASM is executed in the load address above KERNBASE, but JOS need to run mp\_main at 0x7000 address! Of course 0x7000's page is reserved at pmap.c.

#### Per-CPU State and Initialization

When writing a multiprocessor OS, it is important to distinguish between per-CPU state that is private to each processor, and global state that the whole system shares. g

struct CpuInfo, which stores per-CPU variables. cpunum() always returns the ID of the CPU that calls it, which can be used as an index into arrays like cpus.

- **Per-CPU kernel stack.**

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

- **Per-CPU TSS and TSS descriptor.**

A per-CPU task state segment (TSS) is also needed in order to specify where each CPU's kernel stack lives. The TSS for CPU *i* is stored in cpus[i].cpu\_ts, and the corresponding TSS descriptor is defined in the GDT entry gdt[(GD\_TSS0 >> 3) + i].

- **Per-CPU current environment pointer.**

Since each CPU can run different user process simultaneously, we redefined the symbol curenv to refer to cpus[cpunum()].cpu\_env (or thiscpu->cpu\_env), which points to the environment *currently* executing on the *current* CPU

- **Per-CPU system registers.**

All registers, including system registers, are private to a CPU. Therefore, instructions that initialize these registers, such as lcr3(), ltr(), lgdt(), lidt(), etc., must be executed once on each CPU.

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().

```
// 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)
{
    // 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:

int i = 0;
uint32_t kstacktop_i;
for (; i < NCPU; i++){
    kstacktop_i = KSTACKTOP - i * (KSTKSIZE + KSTKGAP);
    boot_map_region(kern_pgdir, kstacktop_i - KSTKSIZE, KSTKSIZE,
                    PADDR(&percpu_kstacks[i]), PTE_W);
}

```

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.)

```

// 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()
    //
    // 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:

thiscpu->cpu_ts.ts_esp0 = KSTACKTOP - (thiscpu->cpu_id) * (KSTKSIZE + KSTKGAP);
thiscpu->cpu_ts.ts_ss0 = GD_KD;

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

// Load the TSS selector (like other segment selectors, the
// bottom three bits are special; we leave them 0)
ltr(GD_TSS0 + (thiscpu->cpu_id << 3));

// Load the IDT
lidt(&idt_pd);

}

```

## Locking

Before letting the AP get any further, we need to first 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 global lock that is held whenever an environment enters kernel mode, and is released when the environment returns to user mode.

kern/spinlock.h declares the big kernel lock, namely `kernel_lock`. It also provides `lock_kernel()` and `unlock_kernel()`, shortcuts to acquire and release the lock.

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

```

void
env_run(struct Env *e)
{
    if(curenv!=NULL)
    {
        if(curenv->env_status==ENV_RUNNING)
            curenv->env_status=ENV_RUNNABLE;
    }
}

```

```

    }
    curenv=e;
    e->env_status=ENV_RUNNING;
    e->env_runs++;

    lcr3(PADDR(e->env_pgdir));
unlock_kernel();

    env_pop_tf(&e->env_tf);
}

```

```

void
i386_init(void)
{
    extern char edata[], end[];

    // Before doing anything else, complete the ELF loading process.
    // Clear the uninitialized global data (BSS) section of our program.
    // This ensures that all static/global variables start out zero.
    memset(edata, 0, end - edata);

    // 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();

    int i;
    for (i = 0; i < NCPU; i++)
        ENV_CREATE(user_idle, ENV_TYPE_USER);

    #if defined(TEST)
        // Don't touch -- used by grading script!
    #endif
}

```



```

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();
}

```

## Question

2. It seems that using the big kernel lock guarantees that only one CPU can run the kernel code at a time. Why do we still need separate kernel stacks for each CPU? Describe a scenario in which using a shared kernel stack will go wrong, even with the protection of the big kernel lock.

If it leaves future-needed data in the single kernel stack, It will have the re-entrant problem.

Round-robin scheduling in JOS works as follows:

- The function `sched_yield()` in the new `kern/sched.c` is responsible for selecting a new environment to run. It searches sequentially through the `envs[]` array in circular fashion, starting just after the previously running environment (or at the beginning of the array if there was no previously running environment), picks the first environment it finds with a status of `ENV_RUNNABLE` (see `inc/env.h`), and calls `env_run()` to jump into that environment.
- `sched_yield()` must never run the same environment on two CPUs at the same time. It can tell that an environment is currently running on some CPU (possibly the current CPU) because that environment's status will be `ENV_RUNNING`.

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

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

```

```

size_t i = 0;
size_t nxt = 0;
idle = NULL;

if (curenv)
    nxt = ENVX(ENVX(curenv->env_id));

for ( ; i < NENV; i++) {
    nxt = (nxt + 1) & (NENV-1);
    if (envs[nxt].env_status == ENV_RUNNABLE) {
        idle = &envs[nxt];
        break;
    }
}

if (idle)
    env_run(idle);
else if (curenv && curenv->env_status == ENV_RUNNING)
    env_run(curenv);

// sched_halt never returns
sched_halt();
}

```

### Question

3. In your implementation of `env_run()` you should have called `lcr3()`. Before and after the call to `lcr3()`, your code makes references (at least it should) to the variable `e`, the argument to `env_run`. Upon loading the `%cr3` register, the addressing context used by the MMU is instantly changed. But a virtual address (namely `e`) has meaning relative to a given address context--the address context specifies the physical address to which the virtual address maps. Why can the pointer `e` be dereferenced both before and after the addressing switch?

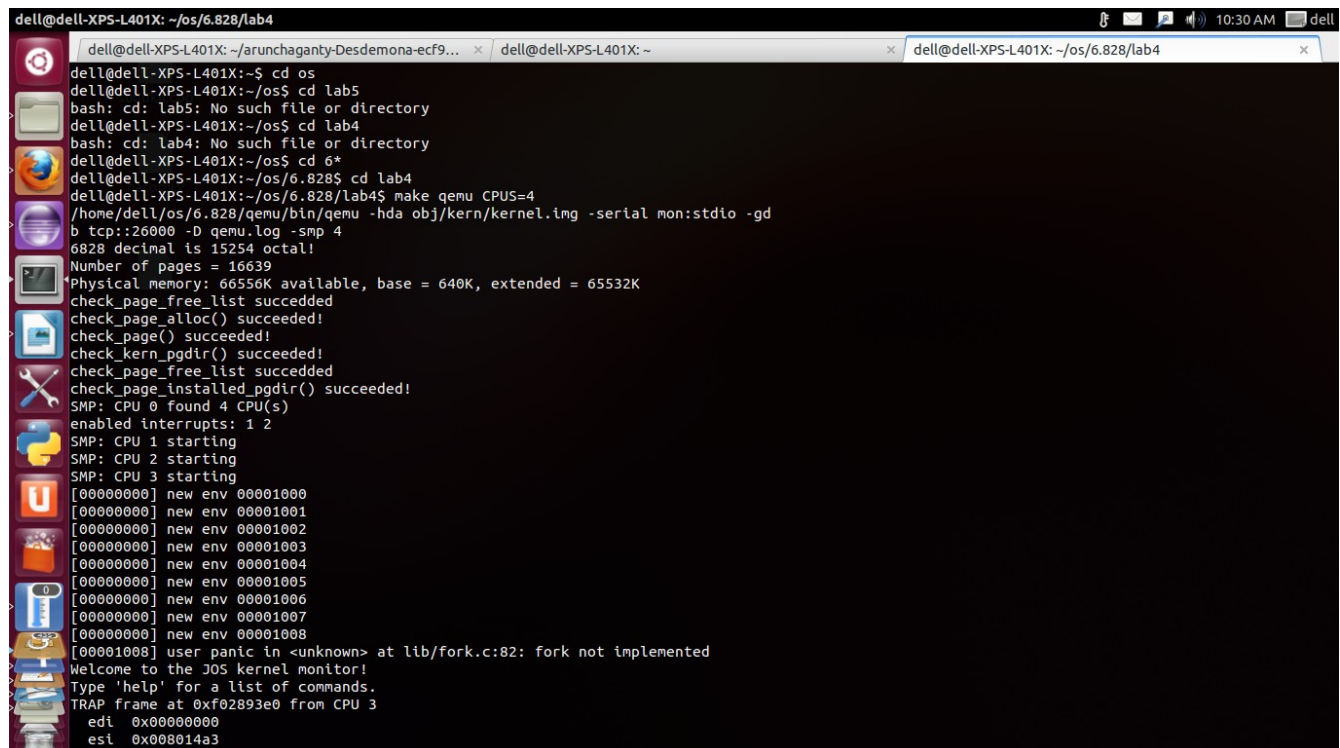
Code Segment is in the Kernel Privilege, it still can access the kern\_pgdir's part belonged to the kernel like environment e.

Unix provides the fork() system call as its process creation primitive. Unix fork() copies the entire address space of calling process (the parent) to create a new process (the child). The only differences between the two observable from user space are their process IDs and parent process IDs (as returned by getpid and getppid). In the parent, fork() returns the child's process ID, while in the child, fork() returns 0. By default, each process gets its own private address space, and neither process's modifications to memory are visible to the other.

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

Code is in attached file.

AFTER EXERCISE 4



```
dell@dell-XPS-L401X: ~/os/6.828/lab4
dell@dell-XPS-L401X: ~$ cd os
dell@dell-XPS-L401X: ~/os$ cd lab5
bash: cd: lab5: No such file or directory
dell@dell-XPS-L401X: ~/os$ cd lab4
bash: cd: lab4: No such file or directory
dell@dell-XPS-L401X: ~/os$ cd 6*
dell@dell-XPS-L401X: ~/os/6.828$ cd lab4
dell@dell-XPS-L401X: ~/os/6.828/lab4$ make qemu CPUS=4
/home/dell/os/6.828/qemu/bin/qemu -hda obj/kern/kernel.img -serial mon:stdio -gd
b tcp::26000 -D qemu.log -smp 4
6828 decimal is 15254 octal!
Number of pages = 16639
Physical memory: 66556K available, base = 640K, extended = 65532K
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 4 CPU(s)
enabled interrupts: 1 2
SMP: CPU 1 starting
SMP: CPU 2 starting
SMP: CPU 3 starting
[00000000] new env 00001000
[00000000] new env 00001001
[00000000] new env 00001002
[00000000] new env 00001003
[00000000] new env 00001004
[00000000] new env 00001005
[00000000] new env 00001006
[00000000] new env 00001007
[00000000] new env 00001008
[00001008] user panic in <unknown> at lib/fork.c:82: fork not implemented
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf02893e0 from CPU 3
edi 0x00000000
esi 0x000014a3
```

## PART B

Unix provides the `fork()` system call as its primary process creation primitive. The `fork()` system call copies the address space of the calling process (the parent) to create a new process (the child).

later versions of Unix took advantage of virtual memory hardware to allow the parent and child to *share* the memory mapped into their respective address spaces until one of the processes actually modifies it. This technique is known as copy-on-write.

To do this, on `fork()` the kernel would copy the address space mappings from the parent to the child instead of the contents of the mapped pages, and at the same time mark the now-shared pages read-only. When one of the two processes tries to write to one of these shared pages, the process takes a page fault. At this point, the Unix kernel realizes that the page was really a "virtual" or "copy-on-write" copy, and so it makes a new, private, writable copy of the page for the faulting process. In this way, the contents of individual pages aren't actually copied until they are actually written to. The child will probably only need to copy one page (the current page of its stack) before it calls `exec()`.

A typical Unix kernel must keep track of what action to take when a page fault occurs in each region of a process's space. For example, a fault in the stack region will typically allocate and map new page of physical memory. A fault in the program's BSS region will typically allocate a new page, fill it with zeroes, and map it. In systems with demand-paged executables, a fault in the text region will read the corresponding page of the binary off of disk and then map it.

**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**

```
// 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 *e;
    if (envid2env(envid, &e, 1) < 0)
        return -E_BAD_ENV;
    e->env_pgfault_upcall = func;
    return 0;
}
```

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

We will call the state of the user environment at the time of the fault the trap-time state. If there is no page fault handler registered, the JOS kernel destroys the user environment with a message as before. Otherwise, the kernel sets up a trap frame on the exception stack that looks like a struct Utrapframe

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. We push an empty 32-bit word, then a struct Utrapframe.

**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?)**

```
// 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.
// The trap handler needs one word of scratch space at the top of the
// trap-time stack in order to return. 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.
// To change what the user environment runs, modify 'curenv->env_tf'
// (the 'tf' variable points at 'curenv->env_tf').

// LAB 4: Your code here.

assert(curenv);
struct UTrapframe uframe;
uintptr_t ufp;
bool bad_case = false;

if (!curenv->env_pgfault_upcall){
    bad_case = true;
    cprintf("[%08x] user fault va %08x ip %08x\n",
```

```

        curenv->env_id, fault_va, tf->tf_eip);

    print_trapframe(tf);
    env_destroy(curenv);
}

if (tf->tf_esp < UXSTACKTOP && tf->tf_esp >= UXSTACKTOP - PGSIZE)
    //recursive page fault, push empty 32-bit word, leaving for ret addr
    ufp = tf->tf_esp - sizeof(struct UTrapframe) -4;
else
    ufp = UXSTACKTOP - sizeof(struct UTrapframe);
user_mem_assert(curenv, (void *)ufp, sizeof(struct UTrapframe), PTE_W);

uframe.utf_eflags = tf->tf_eflags;
uframe.utf_eip = tf->tf_eip;
uframe.utf_err = tf->tf_err;
uframe.utf_esp = tf->tf_esp;
uframe.utf_regs = tf->tf_regs;
uframe.utf_fault_va = fault_va;

*((struct UTrapframe *)ufp) = uframe;
tf->tf_eip = (uintptr_t) curenv->env_pgfault_upcall;
tf->tf_esp = ufp;
env_run(curenv);

```

**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.**

```

// 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.
// 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.

// 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.
addl $8, %esp // pop fault_va, error code
movl 0x28(%esp), %eax // 0x28(%esp)trap-time stack
subl $4, %eax
movl %eax, 0x28(%esp)
movl 0x20(%esp), %ebx // 0x20(%esp)trap-time eip
movl %ebx, (%eax)

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

```

## Exercise 11. Finish set\_pgfault\_handler() in lib/pgfault.c.

```

// 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.
        if ((r = sys_page_alloc(0, (void *) (UXSTACKTOP - PGSIZE),
                                PTE_U|PTE_P|PTE_W)) < 0)
            panic("sys_page_alloc: %e", r);
        sys_env_set_pgfault_upcall(0, _pgfault_upcall);
    }

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

```



```
dell@dell-XPS-L401X: ~/os/6.828/lab4
+ ld obj/user/forktree
+ cc[USER] user/spin.c
+ ld obj/user/spin
+ cc[USER] user/fairness.c
+ ld obj/user/fairness
+ cc[USER] user/pingpong.c
+ ld obj/user/pingpong
+ cc[USER] user/pingpongs.c
+ ld obj/user/pingpongs
+ cc[USER] user/primes.c
+ ld obj/user/primes
+ ld obj/kern/kernel
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 384 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]: Leaving directory `/home/dell/os/6.828/lab4'
dumbfork: OK (1.2s)
Part A score: 5/5

faultread: OK (1.0s)
faultwrite: OK (1.0s)
faultdie: OK (1.0s)
(Old jos.out.faultdie failure log removed)
faultregs: OK (1.0s)
(Old jos.out.faultregs failure log removed)
faultalloc: OK (1.0s)
(Old jos.out.faultalloc failure log removed)
faultallocbad: FAIL (0.9s)
QEMU: Terminated via GDBstub
MISSING '.00001000. user_mem_check assertion failure for va deadbeef'
MISSING '.00001000. free env 00001000'
QEMU output saved to jos.out.faultallocbad
```

now on running make grade faultread,faultdie,faultalloc all are working

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()` 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.

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.

`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

## 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'
```

## FORK

```
// 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.
// 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.
    set_pgfault_handler(pgfault);
    envid_t envid;
    int r;

    if ((envid = sys_exofork()) < 0)
        panic("error in fork(), sys_exofork\n");
    if (envid == 0){
        thisenv = &envs[ENVX(sys_getenvid())];
        return 0;
    }
}
```

```

    }
    uint32_t pn = 0;
    for ( ; pn < PGNUM(UTOP); pn++){

        if (!(uvpd[PDX(pn<<PGSHIFT)] & PTE_P)) continue;
        if (!(uvpt[pn] & PTE_P)) continue;
        if ((pn << PGSHIFT) != UXSTACKTOP - PGSIZE){
            duppage(envid, pn);
        }
    }
}

if ((r = sys_page_alloc(envid, (void *) (UXSTACKTOP - PGSIZE),
                        PTE_U|PTE_P|PTE_W)) < 0)
    panic("error in fork(), sys_page_alloc: %e\n", r);
if ((r = sys_env_set_pgfault_upcall(envid,
                                    thisenv->env_pgfault_upcall)) < 0)
    panic("error in fork(), sys_env_set_pgfault_upcall: %e\n", r);

if ((r = sys_env_set_status(envid, ENV_RUNNABLE)) < 0)
    panic("error in fork(), sys_env_set_status: %e\n", r);

return environ;

```

## DUPPAGE

```

// Map our virtual page pn (address pn*PGSIZE) into the target environ
// 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(environ_t environ, unsigned pn)
{
    int r;

    // LAB 4: Your code here.
    int perm = PTE_P|PTE_U;
    void *va = (void *) (pn << PGSHIFT);
    if ((uvpt[pn] & PTE_W) || (uvpt[pn] & PTE_COW))
        perm |= PTE_COW;
    if ((r = sys_page_map(0, va, environ, va, perm)) < 0)
        panic("error in duppage(), sys_page_map: %e\n", r);
    if ((r = sys_page_map(0, va, 0, va, perm)) < 0)
        panic("error in duppage(), sys_page_map: %e\n", r);
    return 0;
}

```

## pgfault

```
// 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.
    uint32_t pte = uvpt[PGNUM(addr)];
    if (!(pte & (PTE_W|PTE_COW)))
        panic("bad permission in COW pgfault handler\n");

    // 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.
    // No need to explicitly delete the old page's mapping.

    // LAB 4: Your code here.
    addr = ROUNDDOWN(addr, PGSIZE);
    if ((r = sys_page_alloc(0, PFTEMP, PTE_W|PTE_U|PTE_P)) < 0)
        panic("error in pgfault(), sys_page_alloc: %e", r);
    memmove((void *)PFTEMP, addr, PGSIZE);
    if ((r = sys_page_map(0, (void *)PFTEMP, 0, addr, PTE_W|PTE_U|PTE_P)) < 0)
        panic("error in pgfault(), sys_page_map: %e", r);
    if ((r = sys_page_unmap(0, (void *)PFTEMP)) < 0)
        panic("error in pgfault(), sys_page_unmap: %e", r);
}
```

```
dell@dell-XPS-L401X: ~/os/6.828/lab4
Hello, I am environment 00001001.
Hello, I am environment 00001002.
[00001003] new env 00001004
0: I am the parent!
0: I am the child!
Back in environment 00001000, iteration 0.
Back in environment 00001001, iteration 0.
Back in environment 00001002, iteration 0.
1: I am the parent!
1: I am the child!
Back in environment 00001000, iteration 1.
Back in environment 00001001, iteration 1.
Back in environment 00001002, iteration 1.
2: I am the parent!
2: I am the child!
Back in environment 00001000, iteration 2.
Back in environment 00001001, iteration 2.
Back in environment 00001002, iteration 2.
3: I am the parent!
3: I am the child!
Back in environment 00001000, iteration 3.
Back in environment 00001001, iteration 3.
```

## MAKE QEMU

```
dell@dell-XPS-L401X: ~/os/6.828/6.828-lab-master
+ ld obj/kern/kernel
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 384 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]: Leaving directory `/home/dell/os/6.828/6.828-lab-master'
dumbfork: OK (1.5s)
Part A score: 5/5
faultread: OK (1.0s)
faultwrite: OK (1.0s)
faultdie: OK (0.9s)
faultregs: OK (1.1s)
faultalloc: OK (1.0s)
faultallocbad: OK (0.9s)
faultnostack: OK (1.8s)
faultbadhandler: OK (1.2s)
faultevilhandler: OK (1.8s)
forktree: OK (2.4s)
Part B score: 50/50
```

## MAKE GRADE