CS372H: Spring 2009 – Midterm 1

Instructions

- This midterm is closed book and notes with one exception: you may bring and refer to a 1-sided 8.5x11-inch piece of paper printed with a 10-point or larger font. If you hand-write your review sheet, the text density should not be greater than a 10-point font would afford. For reference, a typical page in 10-point font has about 55-60 lines of text.
- If a question is unclear, write down the point you find ambiguous, make a reasonable interpretation, write down that interpretation, and proceed.
- State your assumptions and show your work. **Write brief, precise, and legible answers.** Rambling brain-dumps are unlikely to be effective. **Think before you start writing** so that you can crisply describe a simple approach rather than muddle your way through a complex description that "works around" each issue as you come to it. **Perhaps jot down an outline** to organize your thoughts. And remember, a **picture can be worth 1000 words.**
- For full credit, show your work and explain your reasoning.
- There are two problems on this exam
- Problem 1 has three parts. Answer any **two** of the **three** parts of problem 1.
- Write your name on this exam

1 Project (50)

This problem has three parts numbered 1, 2, and 3 and each taking a page. Answer any **two** of the **three** parts of this problem.

1. At the end of this exam is the the bootloaders boot/main.c, the kernel's kern/entry.S file and a snippet from obj/kernel/kernel.asm

   Suppose that during lab 1 (before you set up paging in lab 2) you wanted to use bochs to put a breakpoint at the start of function cons_getc(void).

   - Suppose you want to use the bochs vb command to set a breakpoint at a virtual address, what would you type?
   - Suppose you want to use the bochs lb command to set a breakpoint at a linear address, what would you type?
   - Suppose that you want to use the bochs pb command to set a breakpoint at a physical address, what would you type?
   - Suppose that after lab 2 (after you set up paging) you want to set a breakpoint at cons_getc(void). Asumming cons_getc(void) ends up at the same offset within the kernel code, which of the above bochs commands will still set a breakpoint where you want it ? **For full credit, explain your answer.**

2. Consider the following code:

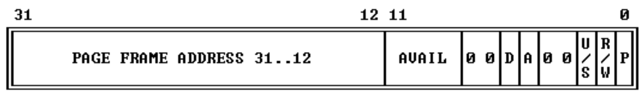
```
int main(int argc, char **argv){
  int x = 1, y = 3, z = 4;
  cprintf("x %d, y %x, z %d\n", x, y, z);
}

int
cprintf(const char *fmt, ...)
{
        va_list ap;
        int cnt;

        va_start(ap, fmt);
        cnt = vcprintf(fmt, ap);
        va_end(ap);

        return cnt;
}
```

Draw a picture of the stack just before execution of the first instruction of cprintf. This picture should show what word the stack pointer points to and show where each of the four arguments to cprintf are on the stack relative to the stack pionter `esp` and frame pointer `ebp`.

3. Suppose I have a readable/writable kernel data structure `Foo *foo` of size `FOO_SIZE` $= 2^{20}$ bytes and I wish to map that data structure read-only to user-level environments from addresses `FOO_MAP` to `FOO_MAP + FOO_SIZE`. What should be stored in the page directory and page tables for both the kernel and the user mappings? **A picture can be worth a Kword.**

   For reference, the following figures are reprinted from the I386 architecture manual

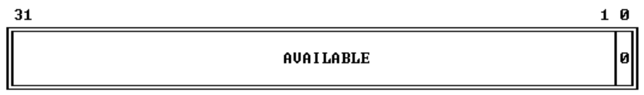
Figure 5-10. Format of a Page Table Entry



P	- PRESENT
R/W	- READ/WRITE
U/S	- USER/SUPERVISOR
D	- DIRTY
AVAIL	- AVAILABLE FOR SYSTEMS PROGRAMMER USE

NOTE: 0 INDICATES INTEL RESERVED. DO NOT DEFINE.

Figure 5-11. Invalid Page Table Entry



2 Multi-threaded programming (50)

Implement a *Rendezvous* object that allows allocation of a group of 10 resources each of which has 10 instances (e.g., 10 instances of resource 1, 10 of resource 2, etc.). `Redezvous→put(int resources[])` makes resources available. `Rendezvous→get(int resources[])` blocks until the requested resources are available and them claims them. `resources[i]` is the count of how many instances of resource `i` are made available by the `put()` call or claimed by the `get()` call.

You will implement two versions of `Rendezvous`.

The first, `SimpleRendezvous`, should be as simple as possible while still ensuring the following *no unnecessary waiting* property: if sufficient resources are available to satisfy any waiting `get()` request, then one or more `get()` requests must return.

The second, `FairRendezvous` should still ensure the *no unnecessary waiting* property, but it should also ensure the *fair waiting* property: if at any time two waiting `get()`s can be satisfied by the currently available resources, the one that started waiting later should not return until the one that started waiting earlier is allowed to return.

Put your solution on the pages that follow.

1. List the state and synchronization variables for the `SimpleRendezvous` class and state their initial values

Type	name	initial value

2. Implement `SImpleRendezvous::put(int resources[])`

3. Implement `SImpleRendezvous::get(int resources[])`

4. Suppose your workload comprises a set of 100 threads, each of which does alternating `get()` and `put()` calls, is the system always guaranteed to be deadlock free? Why or why not?

5. List the state and synchronization variables for the `FairRendezvous` class and state their initial values

Type	name	initial value

6. Implement `FairRendezvous::put(int resources[])`

7. Implement `FairRendezvous::get(int resources[])`

boot/main.c

```
#include <inc/x86.h>
#include <inc/elf.h>

/**********************************************************************
 * This a dirt simple boot loader, whose sole job is to boot
 * an ELF kernel image from the first IDE hard disk.
 *
 * DISK LAYOUT
 *  * This program(boot.S and main.c) is the bootloader.  It should
 *    be stored in the first sector of the disk.
 *
 *  * The 2nd sector onward holds the kernel image.
 *
 *  * The kernel image must be in ELF format.
 *
 * BOOT UP STEPS
 *  * when the CPU boots it loads the BIOS into memory and executes it
 *
 *  * the BIOS intializes devices, sets of the interrupt routines, and
 *    reads the first sector of the boot device(e.g., hard-drive)
 *    into memory and jumps to it.
 *
 *  * Assuming this boot loader is stored in the first sector of the
 *    hard-drive, this code takes over...
 *
 *  * control starts in bootloader.S -- which sets up protected mode,
 *    and a stack so C code then run, then calls bootmain()
 *
 *  * bootmain() in this file takes over, reads in the kernel and jumps to it.
 **********************************************************************/

#define SECTSIZE 512
#define ELFHDR ((struct Elf *) 0x10000) // scratch space

void readsect(void*, uint32_t);
void readseg(uint32_t, uint32_t, uint32_t);

void
bootmain(void)
{
struct Proghdr *ph, *eph;

// read 1st page off disk
readseg((uint32_t) ELFHDR, SECTSIZE*8, 0);

// is this a valid ELF?
if (ELFHDR->e_magic != ELF_MAGIC)
goto bad;

// load each program segment (ignores ph flags)
ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
eph = ph + ELFHDR->e_phnum;
```

```
for (; ph < eph; ph++)
readseg(ph->p_va, ph->p_memsz, ph->p_offset);

// call the entry point from the ELF header
// note: does not return!
((void (*)(void)) (ELFHDR->e_entry & 0xFFFFFF))();

bad:
outw(0x8A00, 0x8A00);
outw(0x8A00, 0x8E00);
while (1)
/* do nothing */;
}

// Read 'count' bytes at 'offset' from kernel into virtual address 'va'.
// Might copy more than asked
void
readseg(uint32_t va, uint32_t count, uint32_t offset)
{
uint32_t end_va;

va &= 0xFFFFFF;
end_va = va + count;

// round down to sector boundary
va &= ~(SECTSIZE - 1);

// translate from bytes to sectors, and kernel starts at sector 1
offset = (offset / SECTSIZE) + 1;

// If this is too slow, we could read lots of sectors at a time.
// We'd write more to memory than asked, but it doesn't matter --
// we load in increasing order.
while (va < end_va) {
readsect((uint8_t*) va, offset);
va += SECTSIZE;
offset++;
}
}

void
waitdisk(void)
{
// wait for disk reaady
while ((inb(0x1F7) & 0xC0) != 0x40)
/* do nothing */;
}

void
readsect(void *dst, uint32_t offset)
{
// wait for disk to be ready
waitdisk();
```

```
outb(0x1F2, 1); // count = 1
outb(0x1F3, offset);
outb(0x1F4, offset >> 8);
outb(0x1F5, offset >> 16);
outb(0x1F6, (offset >> 24) | 0xE0);
outb(0x1F7, 0x20); // cmd 0x20 - read sectors

// wait for disk to be ready
waitdisk();

// read a sector
insl(0x1F0, dst, SECTSIZE/4);
}
```

```
/* See COPYRIGHT for copyright information. */

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

# Shift Right Logical
#define SRL(val, shamt) (((val) >> (shamt)) & ~(-1 << (32 - (shamt))))


###################################################################
# The kernel (this code) is linked at address ~(KERNBASE + 1 Meg),
# but the bootloader loads it at address ~1 Meg.
#
# RELOC(x) maps a symbol x from its link address to its actual
# location in physical memory (its load address).
###################################################################

#define RELOC(x) ((x) - KERNBASE)


.set CODE_SEL,0x8 # index of code seg within mygdt
.set DATA_SEL,0x10 # index of data seg within mygdt

#define MULTIBOOT_PAGE_ALIGN  (1<<0)
#define MULTIBOOT_MEMORY_INFO (1<<1)
#define MULTIBOOT_HEADER_MAGIC (0x1BADB002)
#define MULTIBOOT_HEADER_FLAGS (MULTIBOOT_MEMORY_INFO | MULTIBOOT_PAGE_ALIGN)
#define CHECKSUM (-(MULTIBOOT_HEADER_MAGIC + MULTIBOOT_HEADER_FLAGS))

###################################################################
# entry point
###################################################################

.text

# The Multiboot header
.align 4
.long MULTIBOOT_HEADER_MAGIC
.long MULTIBOOT_HEADER_FLAGS
.long CHECKSUM

.globl _start
_start:
movw $0x1234,0x472 # warm boot

# Establish our own GDT in place of the boot loader's temporary GDT.
lgdt RELOC(mygdtdesc) # load descriptor table

# Immediately reload all segment registers (including CS!)
# with segment selectors from the new GDT.
movl $DATA_SEL, %eax # Data segment selector
movw %ax,%ds # -> DS: Data Segment
movw %ax,%es # -> ES: Extra Segment
movw %ax,%ss # -> SS: Stack Segment
ljmp $CODE_SEL,$relocated # reload CS by jumping
relocated:

# Clear the frame pointer register (EBP)
# so that once we get into debugging C code,
# stack backtraces will be terminated properly.
movl $0x0,%ebp # nuke frame pointer

        # Set the stack pointer
movl $(bootstacktop),%esp

# now to C code
call i386_init

# Should never get here, but in case we do, just spin.
spin: jmp spin


###################################################################
# See <inc/memlayout.h> for a complete description of these two symbols.
###################################################################
.data
.globl vpt
.set vpt, VPT
.globl vpd
.set vpd, (VPT + SRL(VPT, 10))


###################################################################
# boot stack
###################################################################
.p2align PGSHIFT # force page alignment
.globl bootstack
bootstack:
.space KSTKSIZE
.globl bootstacktop
bootstacktop:

###################################################################
# setup the GDT
###################################################################
.p2align 2 # force 4 byte alignment
mygdt:
SEG_NULL # null seg
SEG(STA_X|STA_R, -KERNBASE, 0xffffffff) # code seg
SEG(STA_W, -KERNBASE, 0xffffffff) # data seg
mygdtdesc:
.word 0x17 # sizeof(mygdt) - 1
.long RELOC(mygdt) # address mygdt
```

Snippit from kernel.asm

```
...

void
serial_intr(void)
{
f01002f9: 55                   push   %ebp
f01002fa: 89 e5                mov    %esp,%ebp
f01002fc: 83 ec 08             sub    $0x8,%esp
if (serial_exists)
f01002ff: 83 3d 44 03 11 f0 00  cmpl   $0x0,0xf0110344
f0100306: 74 0c                je     f0100314 <serial_intr+0x1b>
cons_intr(serial_proc_data);
f0100308: c7 04 24 93 05 10 f0  movl   $0xf0100593,(%esp)
f010030f: e8 8e ff ff ff       call   f01002a2 <cons_intr>
}
f0100314: c9                   leave
f0100315: c3                   ret

f0100316 <cons_getc>:
}

// return the next input character from the console, or 0 if none waiting
int
cons_getc(void)
{
f0100316: 55                   push   %ebp
f0100317: 89 e5                mov    %esp,%ebp
f0100319: 83 ec 08             sub    $0x8,%esp
int c;

// poll for any pending input characters,
// so that this function works even when interrupts are disabled
// (e.g., when called from the kernel monitor).
serial_intr();
f010031c: e8 d8 ff ff ff       call   f01002f9 <serial_intr>
kbd_intr();
f0100321: e8 bf ff ff ff       call   f01002e5 <kbd_intr>

// grab the next character from the input buffer.
if (cons.rpos != cons.wpos) {
f0100326: a1 60 05 11 f0       mov    0xf0110560,%eax
f010032b: b9 00 00 00 00       mov    $0x0,%ecx
f0100330: 3b 05 64 05 11 f0    cmp    0xf0110564,%eax
f0100336: 74 21                je     f0100359 <cons_getc+0x43>
c = cons.buf[cons.rpos++];
f0100338: 0f b6 88 60 03 11 f0  movzbl -0xfeefca0(%eax),%ecx
f010033f: 8d 50 01             lea    0x1(%eax),%edx
if (cons.rpos == CONSBUFSIZE)
cons.rpos = 0;
f0100342: 81 fa 00 02 00 00    cmp    $0x200,%edx
f0100348: 0f 94 c0             sete   %al
f010034b: 0f b6 c0              movzbl %al,%eax
f010034e: 83 e8 01              sub    $0x1,%eax
f0100351: 21 c2                 and    %eax,%edx
f0100353: 89 15 60 05 11 f0     mov    %edx,0xf0110560
return c;
}
return 0;
}
f0100359: 89 c8                 mov    %ecx,%eax
f010035b: c9                    leave
f010035c: c3                    ret

f010035d <getchar>:
cons_putc(c);
}

...
```