Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.828 Operating System Engineering: Fall 2003

Quiz I Solutions

All problems are open-ended questions. In order to receive credit you must answer the question as precisely as possible. You have 80 minutes to answer this quiz.

Write your name on this cover sheet AND at the bottom of each page of this booklet.

Some questions may be much harder than others. Read them all through first and attack them in the order that allows you to make the most progress. If you find a question ambiguous, be sure to write down any assumptions you make. Be neat. If we can't understand your answer, we can't give you credit!

THIS IS AN OPEN BOOK, OPEN NOTES QUIZ.

1 (xx/19)	2 (xx/15)	3 (xx/25)	4 (xx/20)	5 (xx/15)	6 (xx/6)	Total (xx/100)

Name:

I Calling conventions

In lab 2 you extended `printf`. Here is the core of `printf` relevant for this question:

```
void printk(const char *fmt, va_list ap)
{
        register char *p, *q;
        register int ch, n;
        u_quad_t uq;
        int base, lflag, qflag, tmp, width;
        char padc;

        for (;;) {
                padc = ' ';
                width = 0;
                while ((ch = *(u_char *) fmt++) != '%') {
                        if (ch == '\0')
                                return;
                        cons_putc(ch);
                }
                lflag = 0;
                qflag = 0;
        reswitch:
                switch (ch = *(u_char *) fmt++) {
                case 'd':
                        uq = getint(&ap, lflag, qflag);
                        if ((quad_t) uq < 0) {
                                cons_putc('-');
                                uq = -(quad_t) uq;
                        }
                        base = 10;
                        goto number;

                [...  other cases omitted ... not relevant to the question]

                number:
                        p = ksprintn(uq, base, &tmp);
                        if (width && (width -= tmp) > 0)
                                while (width--)
                                        cons_putc(padc);
                        while ((ch = *p--) != '\0')
                                cons_putc(ch);
                        break;
                }
        }
}
```

Name:

```
static u_quad_t
getint(va_list *ap, int lflag, int qflag)
{
        if (lflag)
                return va_arg(*ap, u_long);
        else if (qflag)
                return va_arg(*ap, u_quad_t);
        else
                return va_arg(*ap, u_int);
}


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

        va_start(ap, fmt);
        kprintf(fmt, ap);
        va_end(ap);
        return 0;
}
```

1. [10 points]: What values does gcc on the x86 push on the stack for the call:

`printf(''the class number is %s and used to be %d \n'', ''6828'', 6097)`

1. The number 6097, as a 4-byte integer.
2. The address of the (null-terminated) string "`6828`", as a 4-byte pointer.
3. The address of the string "`the class number ...`", as a 4-byte pointer.

Name:

2. [4 points]: gcc pushes the arguments in a particular order. What is the order and why?

gcc pushes arguments in reverse order, last argument first. Because the stack grows down on the x86 (and PDP-11), this means that the first argument (last one pushed) will have the lowest address in memory, just above the function's return address. This way, the called function can find its first argument without knowing exactly how many and what types of other arguments it was called with. Knowing the location and type of the first argument, the function can then locate the second argument (if there is one), and so on. The `printf()` function depends on this property because it uses its first argument (the format string) to determine how many and what types of additional arguments it was called with.

3. [5 points]: Explain what *va_arg* does briefly.

The `va_arg` macro is invoked with an expression of the form `va_arg(`$vl, type$`)`. The macro assumes that vl is a pointer variable of type `va_list`, which points to the "current" argument in a variable-length list of arguments that was pushed onto the stack as a result of a C function call. The `va_arg` macro further assumes that $type$ is a C-language type name describing the type of this argument. `va_arg` uses the vl pointer to read the current argument from the stack and "return" it as the result of the `va_arg(`$vl, type$`)` expression. After reading the current argument, `va_arg` also increments the vl pointer by the appropriate number of bytes, typically `sizeof(`$type$`)`, to advance vl to point to the next argument on the stack. As a result, the next invocation of the `va_arg` macro on the same vl will read the next argument in the list, and so on.

Name:

II Concurrency

Sheets 86 through 87 show the code for a simple device: the paper tape reader.

4. [15 points]: If you delete *spl4()* on line 8686, can you give a concrete sequence of events that results in deadlock? (Hint: you don't have to understand the device deeply to answer this question; focus on the interaction of *sleep* and *wakeup*.)

1. A user process opens the device, causing a call to `pcopen()` in the kernel.
2. `pcopen()` sets `pc11.pcstate` to `WAITING` (line 8657), then sleeps until an interrupt causes `pc11.pcstate` to change (lines 8658-8661).
3. A device interrupt causes `pcrint()` to set `pc11.pcstate` to `READING` (line 8724).
4. `pcopen()` wakes up and returns to the user process.
5. The user process now invokes the `read()` system call on this device, resulting in a call to `pcread()` in the kernel.
6. `pcread` reaches line 8691 after finding that the input buffer is empty (line 8688) and `pc11.pcstate` is not `EOF` (line 8689).
7. An interrupt occurs before `pcread()` executes line 8691, causing `pcrint()` to be called in the middle of the call to `pcread`.
8. Since `pc11.pcstate` is now `READING`, the `if` block at line 8726 is taken.
9. At line 8727, `pcrint()` discovers that the device has reported an error, so it sets `pc11.pcstate` to `EOF` (line 8728) and then calls `wakeup` (line 8734). This call to `wakeup` does nothing, since the (interrupted) call to `pcread()` has not yet reached its `sleep()` call on line 8693, and no other user process is currently reading the device.
10. `pcrint()` returns, and `pcread()` resumes where it left off, at line 8691.
11. `pcread()` reaches line 8693, where it calls `sleep()` to wait for more characters to arrive from the device. But notice that `pc11.pcstate` is now `EOF`, because `pcrint()` set this condition *after* `pcread()` checked for it in line 8689. Because `pc11.pcstate` is `EOF` and not `READING`, the `wakeup(&pc11.pcin)` call on line 8734 can never be reached even if further device interrupts occur, causing the user process to sleep forever.

Name:

III Virtual memory

Here is the layout of virtual memory that you set up in lab 2.

```
/*
 * Virtual memory map:                                Permissions
 *                                                    kernel/user
 *
 *    4 Gig -------->  +------------------------------+
 *                     |                              | RW/--
 *                     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 *                     :              .               :
 *                     :              .               :
 *                     :              .               :
 *                     |~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~| RW/--
 *                     |                              | RW/--
 *                     |  Physical Memory             | RW/--
 *                     |                              | RW/--
 *    KERNBASE ----->  +------------------------------+
 *                     |  Kernel Virtual Page Table   | RW/--    PDMAP
 *    VPT,KSTACKTOP--> +------------------------------+                 --+
 *                     |        Kernel Stack          | RW/--  KSTKSIZE   |
 *                     | - - - - - - - - - - - - - - -|                 PDMAP
 *                     |       Invalid memory         | --/--             |
 *    ULIM     ------> +------------------------------+                 --+
 *                     |      R/O User VPT            | R-/R-    PDMAP
 *    UVPT      ---->  +------------------------------+
 *                     |        R/O PAGES             | R-/R-    PDMAP
 *    UPAGES    ---->  +------------------------------+
 *                     |        R/O ENVS              | R-/R-    PDMAP
 * UTOP,UENVS -------> +------------------------------+
 * UXSTACKTOP -/       |      user exception stack    | RW/RW   BY2PG
 *                     +------------------------------+
 *                     |       Invalid memory         | --/--   BY2PG
 *    USTACKTOP  ----> +------------------------------+
 *                     |     normal user stack        | RW/RW   BY2PG
 *                     +------------------------------+
 *                     |                              |
 *                     |                              |
 *                     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 *                     .                              .
 *                     .                              .
 *                     .                              .
 *                     |~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|
 *                     |                              |
 *    UTEXT ------->   +------------------------------+
 *                     |                              |  2 * PDMAP
 *    0 ------------>  +------------------------------+
 */
```

Attached to the quiz is the `i386_vm_init` that was provided to you. Assume you completed `i386_vm_init` correctly.

5. [10 points]: What entries (rows) in the page directory have been filled in after `i386_vm_init` has completed? What addresses do they map and where do they point? In other words, fill out this table as much as possible:

Entry	Base Virtual Address	Points to (logically):
1023	0xffc00000	Page table for top 4MB of phys mem
1022	0xff800000	.
.	.	.
.	.	.
.	.	.
.	.	.
960	KERNBASE (0xf0000000)	Page table for low 4MB of phys mem
959	VPT (0xefc00000)	Page directory (kernel-only, R/W)
958	ULIM (0xef800000)	Page table mapping kernel stack
957	UVPT (0xef400000)	Page directory (kernel/user, R-O)
956	UPAGES (0xef000000)	Page table mapping "pages" array
955	UTOP,UENVS (0xeec00000)	Page table mapping "envs" array
954	.	Nothing mapped
.	.	.
.	.	.
.	.	.
.	.	.
.	.	.
.	.	.
.	.	.
.	.	.
.	.	.
.	.	.
.	.	.
.	.	.
2	0x00800000	.
1	0x00400000	.
0	0x00000000	Nothing mapped

Name:

6. [5 points]:

Here is a section of the course staff solution to Lab 2's `i386_vm_init`. This section sets up the UPAGES mapping.

```
//////////////////////////////////////////////////////////////////////
// Make 'pages' point to an array of size 'npage' of 'struct Page'.
// You must allocate this array yourself.
// Map this array read-only by the user at virtual address UPAGES
// (ie. perm = PTE_U | PTE_P)
// Permissions:
//    - pages -- kernel RW, user NONE
//    - the image mapped at UPAGES  -- kernel R, user R
// Your code goes here:
n = npage*sizeof(struct Page);
pages = alloc(n, BY2PG, 1);
boot_map_segment(pgdir, UPAGES, n, PADDR(pages), PTE_U);
```

A common mistake is to add the line:

```
boot_map_segment(pgdir, (u_int)pages, n, PADDR(pages), PTE_W);
```

This line is unnecessary, because the mapping already exists. Why does the mapping already exist? Explain exactly which other code has already provided the mapping. You may find it useful to refer to the `i386_vm_init` attached to this quiz.

> The `alloc()` function, used to allocate the `pages` array, always returns a kernel virtual address in the region from `KERNBASE` to $2^{32}-1$ in which physical memory is directly and contiguously mapped into the kernel's address space. You already established a mapping for this entire region of the kernel virtual address space, in the immediately preceding section of `i386_vm_init()`, under this comment:
>
> ```
> //////////////////////////////////////////////////////////////////////
> // Map all of physical memory at KERNBASE.
> // Ie.  the VA range [KERNBASE, 2^32 - 1] should map to
> //      the PA range [0, 2^32 - 1 - KERNBASE]
> // We might not have that many(ie. 2^32 - 1 - KERNBASE)
> // bytes of physical memory.  But we just set up the mapping anyway.
> // Permissions: kernel RW, user NONE
> ```

Name:

7. [5 points]:

In Lab 3, `env.c` creates the user-level address space for an environment. If the code that created the address space was buggy and did not set up a mapping for the area starting at KERNBASE, when would this bug manifest itself? What specific instruction would cause the bug to "take effect" (triple-fault the processor)? (Note: you can answer this question without having completed lab 3.)

When the kernel attempts to switch to the new environment, `env_run()` will call `lcr3()` to load the new environment's page directory into the processor's page directory base register (PDBR/CR3) and flush the TLB. Since the kernel's code itself resides in the virtual address range from `KERNBASE` to $2^{32} - 1$, when the processor tries to execute the next instruction immediately after the "move to CR3" instruction, it will take a page fault because that virtual address is no longer mapped. In order to handle this fault, the processor attempts to push the old CS, EIP, and EFLAGS on the stack—but since the kernel stack is also in this virtual address region starting at `KERNBASE`, these memory accesses will also fail, causing a triple fault.

8. [5 points]: On the x86, we make the kernel's `Page` structures accessible to the user environments (in the form of the mapping at UPAGES). What specific mechanism (i.e., what register, memory address, or bit thereof) is used to keep the user environments from changing the `Page` structures?

The mapping of the `pages` array at virtual address `UPAGES` includes the permission bit `PTE_U`, allowing code running in user mode access to this mapping. This mapping does *not* include the permission bit `PTE_W`, however, which ensures that this mapping cannot be used to write to the `pages` array (by code running in either user or kernel mode).

Name:

IV System calls

9. [5 points]: Draw the kernel stack after v6's `icode` called its first instruction and the kernel just entered `trap` in trap.c (sheet 26).

Old PSW
Old PC
Old r0
PS in trap
Old r1
User SP
PS & 037 = error code (`dev`)
Address of line 0785

Name:

10. [10 points]: What are the values of the arguments to trap? (Fill out the following table.)

dev	6 - the low 4 bits of the PSW the processor loads from the trap vector defined at line 0518.
sp	0 - the "top" of the user stack, and the top of virtual address space - i.e., 2^16 wrapped to 0.
r1	irrelevant
nps	030346 - PM=user, CM=kernel, spl7, dev=6
r0	irrelevant
pc	2 - just after syscall instruction in icode
ps	170000 - PM=user, CM=user

11. [5 points]: Briefly describe the point of the statement on line 3188.

Sets the saved user-mode PC to zero in order to begin execution of the newly loaded program at virtual address zero.

Name:

V Thread switching

A process in UNIX v6 switches to another process using `retu`, which is called on line 2228.

12. [10 points]: Annotate every line of assembly of `_retu` (reproduced from sheet 07). What does the statement do and why?

```
_retu:
   bis     $340, PS     # Disable interrupts (set SPL=7 in PSW) to
                        # prevent interference while switching stacks.

   mov     (sp)+, r1    # Pop return address of retu's caller
                        # off of old stack, save in r1

   mov     (sp), KISA6  # Switch kernel's u-area (page 6) mapping to
                        # the _new_ process's u-area (including stack)

   mov     $_u, r0      # Load address of bottom of u-area (struct user)
                        # into r0

1:
   mov     (r0)+, sp    # Load new process's saved sp (r6) from u_rsav[0]


   mov     (r0)+, r5    # Load new process's saved r5 from u_rsav[1]


   bic     $340, PS     # Re-enable interrupts


   jmp     (r1)         # Return to the code that called retu,
                        # except executing on the new process's stack!
```

13. [5 points]: What does the stack pointer point to at line 2229 in the first call to swtch, after the kernel booted?

The stack pointer points into process 1's kernel stack at this point, specifically at the stack frame established by `newproc()` and saved by the `savu()` call on line 1889. The call from `main()` to `newproc()` on line 1627 originally set up this stack frame in the context of process 0, but then `newproc()` copied process 0's kernel stack (lines 1897-1916) in order to initialize process 1's kernel stack. It is this snapshot of process 0's kernel stack that the first call to `swtch()` switches to.

Name:

VI Feedback

Since 6.828 is a new subject, we would appreciate receiving some feedback on how we are doing so that we can make corrections. (Any answer, except no answer, will receive full credit!)

14. [2 points]: What is the best aspect of 6.828?

15. [2 points]: What is the worst aspect of 6.828?

16. [2 points]: If there is one thing that you would like to see changed in 6.828, what would it be?

End of Quiz I

Name:

Here is `i386_vm_init` from Lab 2.

```
// Set up a two-level page table:
//    boot_pgdir is its virtual address of the root
//    boot_cr3 is the physical adresss of the root
// Then turn on paging.  Then effectively turn off segmentation.
// (i.e., the segment base addrs are set to zero).
//
// This function only sets up the kernel part of the address space
// (ie. addresses >= UTOP).  The user part of the address space
// will be setup later.
//
// From UTOP to ULIM, the user is allowed to read but not write.
// Above ULIM the user cannot read (or write).
void
i386_vm_init(void)
{
  Pde *pgdir;
  u_int cr0, n;

  panic("i386_vm_init: This function is not finished\n");

  //////////////////////////////////////////////////////////////////////
  // create initial page directory.
  pgdir = alloc(BY2PG, BY2PG, 1);
  boot_pgdir = pgdir;
  boot_cr3 = PADDR(pgdir);

  //////////////////////////////////////////////////////////////////////
  // Recursively insert PD in itself as a page table, to form
  // a virtual page table at virtual address VPT.
  // (For now, you don't have understand the greater purpose of the
  // following two lines.)

  // Permissions: kernel RW, user NONE
  pgdir[PDX(VPT)] = PADDR(pgdir)|PTE_W|PTE_P;

  // same for UVPT
  // Permissions: kernel R, user R
  pgdir[PDX(UVPT)] = PADDR(pgdir)|PTE_U|PTE_P;

  //////////////////////////////////////////////////////////////////////
  // Map the kernel stack (symbol name "bootstack"):
  //   [KSTACKTOP-PDMAP, KSTACKTOP)  -- the complete VA range of the stack
  //     * [KSTACKTOP-KSTKSIZE, KSTACKTOP) -- backed by physical memory
  //     * [KSTACKTOP-PDMAP, KSTACKTOP-KSTKSIZE) -- not backed => faults
  //   Permissions: kernel RW, user NONE
  // Your code goes here:

  //////////////////////////////////////////////////////////////////////
  // Map all of physical memory at KERNBASE.
  // Ie.  the VA range [KERNBASE, 2^32 - 1] should map to
  //      the PA range [0, 2^32 - 1 - KERNBASE]
```

Name:

```
// We might not have that many(ie. 2^32 - 1 - KERNBASE)
// bytes of physical memory.  But we just set up the mapping anyway.
// Permissions: kernel RW, user NONE
// Your code goes here:

////////////////////////////////////////////////////////////////////
// Make 'pages' point to an array of size 'npage' of 'struct Page'.
// You must allocate this array yourself.
// Map this array read-only by the user at virtual address UPAGES
// (ie. perm = PTE_U | PTE_P)
// Permissions:
//    - pages -- kernel RW, user NONE
//    - the image mapped at UPAGES  -- kernel R, user R
// Your code goes here:

////////////////////////////////////////////////////////////////////
// Make 'envs' point to an array of size 'NENV' of 'struct Env'.
// You must allocate this array yourself.
// Map this array read-only by the user at virtual address UENVS
// (ie. perm = PTE_U | PTE_P)
// Permissions:
//    - envs itself -- kernel RW, user NONE
//    - the image of envs mapped at UENVS  -- kernel R, user R
// Your code goes here:

check_boot_pgdir();

////////////////////////////////////////////////////////////////////
// On x86, segmentation maps a VA to a LA (linear addr) and
// paging maps the LA to a PA.  I.e. VA => LA => PA.  If paging is
// turned off the LA is used as the PA.  Note: there is no way to
// turn off segmentation.  The closest thing is to set the base
// address to 0, so the VA => LA mapping is the identity.

// Current mapping: VA KERNBASE+x => PA x.
//     (segmentation base=-KERNBASE and paging is off)

// From here on down we must maintain this VA KERNBASE + x => PA x
// mapping, even though we are turning on paging and reconfiguring
// segmentation.

// Map VA 0:4MB same as VA KERNBASE, i.e. to PA 0:4MB.
// (Limits our kernel to <4MB)
pgdir[0] = pgdir[PDX(KERNBASE)];

// Install page table.
lcr3(boot_cr3);

// Turn on paging.
cr0 = rcr0();
cr0 |= CR0_PE|CR0_PG|CR0_AM|CR0_WP|CR0_NE|CR0_TS|CR0_EM|CR0_MP;
cr0 &= ~(CR0_TS|CR0_EM);
lcr0(cr0);
```

Name:

```
  // Current mapping: KERNBASE+x => x => x.
  // (x < 4MB so uses paging pgdir[0])

  // Reload all segment registers.
  asm volatile("lgdt _gdt_pd+2");
  asm volatile("movw %%ax,%%gs" :: "a" (GD_UD|3));
  asm volatile("movw %%ax,%%fs" :: "a" (GD_UD|3));
  asm volatile("movw %%ax,%%es" :: "a" (GD_KD));
  asm volatile("movw %%ax,%%ds" :: "a" (GD_KD));
  asm volatile("movw %%ax,%%ss" :: "a" (GD_KD));
  asm volatile("ljmp %0,$1f\n 1:\n" :: "i" (GD_KT));  // reload cs
  asm volatile("lldt %0" :: "m" (0));

  // Final mapping: KERNBASE+x => KERNBASE+x => x.

  // This mapping was only used after paging was turned on but
  // before the segment registers were reloaded.
  pgdir[0] = 0;

  // Flush the TLB for good measure, to kill the pgdir[0] mapping.
  lcr3(boot_cr3);
}
```

Name: