*Department of Electrical Engineering and Computer Science*

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

**6.828 Operating System Engineering: Fall 2004**

# 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/40) | 2 (xx/25) | 3 (xx/25) | 4 (xx/10) | Total (xx/100) |
|---|---|---|---|---|
| | | | | |

**Name:**

# I Virtual memory

**1. [5 points]:** The PDP-11 uses two bits in the Processor Status Word (PSW) to determine whether it is currently running in kernel or user mode. Which bits in which register on the x86 determine whether the processor is running in kernel or user mode, and what x86 instruction does the JOS kernel use to enter user mode? (Explain briefly)

> **The low two bits of the Code Segment (CS) register** *represent the x86 processor's Current Privilege Level (CPL): 0 is the most privileged level ("kernel mode"), 3 is the least privileged level ("user mode"), and the other two values represent intermediate privilege levels that most operating systems do not use.*
>
> *The JOS kernel uses* **the IRET instruction** *to enter (or "return to") user mode. IRET loads the EIP, CS, EFLAGS, ESP, and SS registers from the kernel stack, in effect changing to the user privilege level, switching to the user's stack, and jumping to the appropriate user code all in one atomic processor operation.*

**2. [5 points]:** The v6 kernel must copy values from user space with the special instruction `mfpi` (push onto the current stack the value of the designated word in the "previous" address space). What instructions does the JOS kernel use to access user-level values? (Explain briefly)

> **No special instructions are required** *for the JOS kernel to access the user address space, because JOS maps the kernel's private virtual address region identically into every user environment's virtual address space, protecting the kernel-private mappings from user-mode access via the PTE_U bit in the appropriate page directory and page table entries. Whenever the kernel is executing in the context of (on behalf of) a particular user environment whose address space is active (loaded into the processor's CR3 register), as long as the kernel takes appropriate precautions to protect itself from invalid pointers passed by the user,* **the kernel can access the user's address space with ordinary instructions such as MOV**, *using the same user virtual addresses that the user environment itself would use to access its own 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 ------------>  +------------------------------+
 */
```

**Name:**

Also, here is the layout of physical memory in a PC:

```
+------------------+  <- 0xFFFFFFFF (4GB)
|      32-bit      |
|  memory mapped   |
|     devices      |
|                  |
/\/\/\/\/\/\/\/\/\/\

/\/\/\/\/\/\/\/\/\/\
|                  |
|      Unused      |
|                  |
+------------------+  <- depends on amount of RAM
|                  |
|                  |
| Extended Memory  |
|                  |
|                  |
+------------------+  <- 0x00100000 (1MB)
|     BIOS ROM     |
+------------------+  <- 0x000F0000 (960KB)
|  16-bit devices, |
|  expansion ROMs  |
+------------------+  <- 0x000C0000 (768KB)
|   VGA Display    |
+------------------+  <- 0x000A0000 (640KB)
|                  |
|    Low Memory    |
|                  |
+------------------+  <- 0x00000000
```

**3. [5 points]:** Why does the JOS kernel arrange that the virtual addresses from `KERNBASE` and higher have read-write access in kernel mode and no access in user mode? (Explain briefly.)

*The JOS kernel needs to reserve part of the virtual address space for its own use because privilege level changes on the x86 (i.e., transitions in the CPL field of the CS register) do not cause the processor to make an automatic address space transition, as the PDP-11 does when switching privilege levels. On the x86, the kernel therefore needs "a place to stand" in the user's address space during transitions between kernel and user mode, and simply mapping all of the physical memory that the kernel makes use of into every user environment's address space is the easiest way to solve this problem.*

**In order to protect the kernel from buggy or malicious user environments, and to protect different user environments from each other**, *the kernel must ensure that this special kernel-private portion of each environment's address space remains completely inaccessible to user environments while they are running in user mode. Allowing user environments to write into the virtual address region above KERNBASE would enable users to modify arbitrary kernel data structures and thereby compromise the integrity of the entire system, while allowing user environments to read in this region would allow any user environment to "spy on" all other running environments and find sensitive information such as passwords or cryptographic keys.*

**Name:**

You completed the following function in lab 2:

```
//
// Initialize page structure and memory free list.
//
page_init(void)
{
        // The exaple code here marks all pages as free.
        // However this is not truly the case.  What memory is free?
        //  1) Mark page 0 as in use(for good luck)
        //  2) Mark the rest of base memory as free.
        //  3) Then comes the IO hole [IOPHYSMEM, EXTPHYSMEM) => mark it as in u
se
        //     So that it can never be allocated.
        //  4) Then extended memory(ie. >= EXTPHYSMEM):
        //     ==> some of it's in use some is free. Where is the kernel?
        //     Which pages are used for page tables and other data structures?

        //
        // Change the code to reflect this.
        int i;
        LIST_INIT (&page_free_list);
        for (i = 0; i < npage; i++) {
                pages[i].pp_ref = 0;
                LIST_INSERT_HEAD(&page_free_list, &pages[i], pp_link);
        }
}
```

**4. [10 points]:** Which pages in extended memory should `page_init` mark as in use? (Explain briefly and feel free to refer to the virtual memory map.)

*The kernel's* `page_init` *function must mark all of the extended memory* **from 1MB to the final value of the** `freemem` **variable** *as in use. The kernel's code, data, and bss itself is located in extended memory starting at physical address 0x100000 (1MB), between the magic linker-defined symbols* `start` *and* `end`*. Immediately after the kernel executable is the physical memory the kernel has already allocated before* `page_init` *time, via the boot-time* `alloc` *function, for various critical kernel data structures such as the kernel's page tables, the* `pages` *array, and the* `envs` *array. The* `page_init` *function must make sure that the physical pages used to hold the kernel's program itself and these critical boot-time-allocated structures do not get reused for other purposes during subsequent memory allocations.*

**5. [5 points]:** Why should `page_init` mark page 0 as in use for good luck? (Explain briefly.)

**The real-mode IDT and certain important BIOS data structures are historically located in physical page 0.** *To be specific, the real-mode IDT is from 0x000 through 0x3ff, and the BIOS data area is from 0x400 through 0x4ff. If JOS ever were need to switch back to real mode and make calls back to the system's built-in BIOS, for example to support Advanced Power Management (APM) or to detect physical memory beyond the 64MB that can be registered in the system's CMOS RAM, then these data structures would need to be preserved. JOS currently does not do any BIOS callbacks, however, which is why preserving page 0 is at the moment merely "for good luck."*

*We included this admittedly obscure question not because it has any particularly critical importance to OS design or implementation principles, but as a reward for those students who have been faithfully coming and paying attention to lectures. :)*

In lab 3 you wrote the following code to setup the IDT for system calls in `idt_init`:

```
SETGATE(idt[T_SYSCALL], 1, GD_KT, handler_SYSCALL, 3);
```

**6. [5 points]:** Why must the descriptor protection level be set to 3 instead of 0 (as with most of the other entries in the IDT)? (Explain briefly.)

*The processor checks the descriptor privilege level (DPL) in the IDT gate descriptor against the current privilege level (CPL) in the CS register to determine whether to allow a software interrupt (*`int` *instruction) to execute using that interrupt vector. Therefore,* **the DPL for the system call gate must be set to 3 in order for the processor to permit system calls from user mode via an** `int $0x30` **instruction.** *(*`T_SYSCALL` *is 0x30.)*

**7. [5 points]:** If the descriptor protection level is set to 0, what would happen if the processor executes an "int 30" instruction? (Explain briefly.)

*The processor takes a* **General Protection Fault (GPF)** *instead of using the interrupt vector requested by the* `int` *instruction.*

**Name:**

# II Proccesses

**8. [5 points]:** In which structure does v6 allocate the kernel stack? (Explain briefly.)

*The v6 kernel stack for a given process is located* **in the u-area** *for that process. A process's u-area consists of the kernel stack and the* `user` *structure.*

**9. [5 points]:** How many of these structures does v6 allocate? (Explain briefly.)

*The v6 kernel allocates* **one kernel stack per process**. *The v6 kernel is designed this way so that when kernel code running on behalf of a given process needs block waiting for some event, the kernel can retain state on the blocked process's kernel stack even as other processes are allowed to continue running independently. Retaining the state of blocked processes on separate kernel stacks in this way makes it easier to implement complex, high-level functionality such as file systems within the kernel.*

**10. [5 points]:** The structure (and thus the kernel stack) is stored at the *virtual* address 140000 and up. What must happen when the kernel switches to a different process? (Explain briefly.)

*Whenever the v6 kernel switches from one process to another,* **it must change the value in KPAR6, the Kernel Page Address Register for this "magic" region of the kernel's virtual address space, to refer to the new process's u-area.** *In this way, regardless of which process is currently running, the current process's u-area always appears at virtual address 140000 in the kernel's address space.*

**11. [10 points]:** In v6, the `init` process, pid 1, inherits processes whose parent have exited. Sketch out the required changes to the kernel so that children that have no parent anymore clean themselves up on `exit` (sheet 32).

*The trickiest part of modifying the v6 kernel is to decide how to deal with deallocating the orphan process's stack. Two possible approaches are: either modifying* `swtch` *to free memory; or by adding a reaper process to the kernel (although, depending on how it is implemented, the later is probably not too much different from the way the* `init` *process is used to clean up orphans right now). We cannot simply free the u-area in* `exit` *since, the u-area contains the specific kernel stack being used to execute the* `exit` *code. To free all the memory being used by a process, we need to switch to a different kernel stack.*

*Some other points: processes need a way to tell if they are orphans (e.g. storing* `-1` *in* `p_ppid`*); zombie processes and the cleanup code in* `wait` *will still be needed for the case of non-orphan processes;* `exit` *may need to be modified to clean up zombie children that haven't been* `wait`*'d on when an orphan* `exit`*'s;* `exit` *needs to make orphans of children who aren't zombies instead of making them children of the init process; some of the code from* `wait` *to dealloc/reinit memory may be added to* `exit` *(e.g. the* `mfree` *and the code to reset the proc table entry).*

*Answers mentioning some of the above points, within a coherent kernel modification plan, were given credit. Students had to realize the complication with the kernel stack to score full points on this quesiton.*

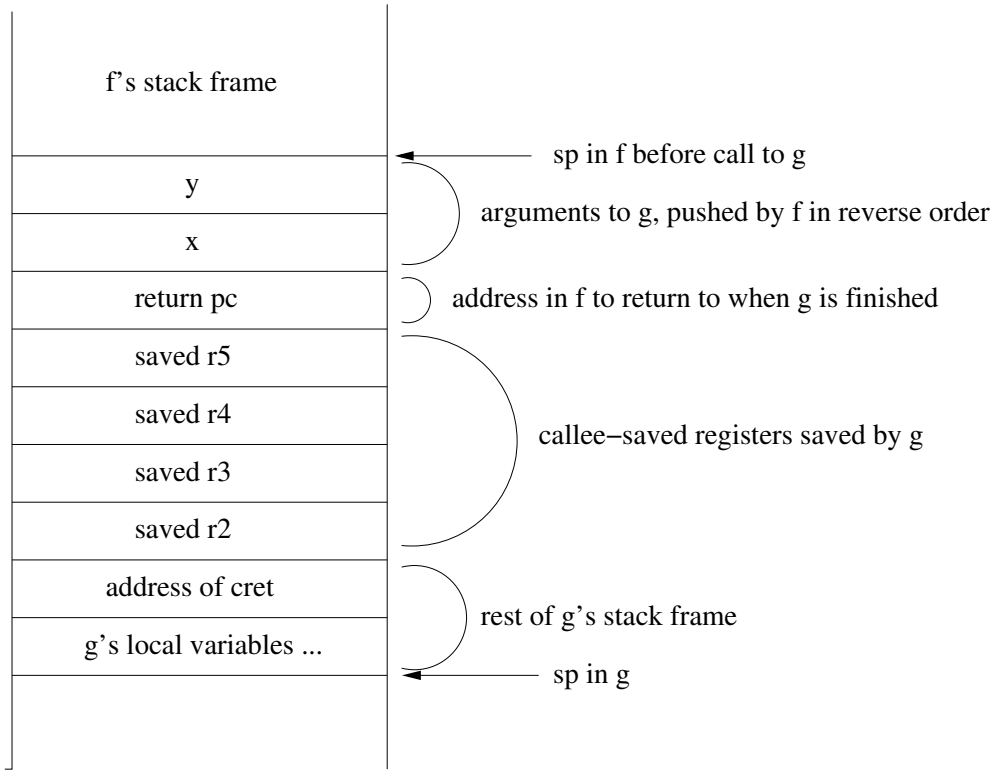
*Students who answered this question assuming that we were asking you to modify the kernel so that a parent would kill off all of its children when* `exit` *was called misinterpreted the question. Such answers scored low because they showed a lack of understanding of the unix process model and the purpose of having zombie processes.*

# III Calling conventions

**12. [5 points]:** Assume a function `f` calls a function `g` using the C calling conventions of v6 on the PDP-11. The callee function `g` takes two arguments, as follows:

```
void g(int x, int y) {
...
}
```

Below is the stack at the point that `f` is about to call `g`. So far `f`'s own local variables and other private state are already on the stack, but nothing related to its call to `g`. Draw the stack at the point just before the processor executes the first instruction of `g` that does "real" work, after running `g`'s function prologue. (The information on page 10-3 of the commentary might be helpful.)




*We gave full credit to any solution that mentions and lists in the correct order: the arguments y and x, the return address, and callee-saved registers.*

**13. [5 points]:** Annotate your picture by providing for each value a brief explanation why the value is pushed on the stack.

**Name:**

The following PDP-11 assembly function appears (with a different name) in the v6 C library:

```
_mystery:
        mov     r5,-(sp)
        mov     2(r5),r1                / pc of caller of caller
        mov     sp,r5
        clr     r0
        cmp     -4(r1),jsrsd
        bne     8f
        mov     $2,r0
8:
        cmp     (r1),tsti
        bne     1f
        add     $2,r0
        br      2f
1:
        cmp     (r1),cmpi
        bne     1f
        add     $4,r0
        br      2f
1:
        cmp     (r1),addi
        bne     1f
        add     2(r1),r0
        br      2f
1:
        cmp     (r1),jmpi
        bne     1f
        add     2(r1),r1
        add     $4,r1
        br      8b
1:
        cmpb    1(r1),bri+1
        bne     2f
        mov     r0,-(sp)
        mov     (r1),r0
        swab    r0
        ash     $-7,r0
        add     r0,r1
        add     $2,r1
        mov     (sp)+,r0
        br      8b
2:
        asr     r0
        mov     (sp)+,r5
        rts     pc

.data
jsrsd:  jsr     pc,*$0
tsti:   tst     (sp)+
cmpi:   cmp     (sp)+,(sp)+
addi:   add     $0,sp
jmpi:   jmp     0
bri:    br      .
```

**Name:**

For your convenience, Russ Cox has translated it into some equivalent pseudocode:

```
mystery:
        cpc = *(r5+2)   /* cpc = pc of caller of caller */
        n = 0
        if inst[cpc-4] == "jsr pc, *$x"
                n = 2
again:
        if inst[cpc] == "tst (sp)+"
                n += 2
        else if inst[cpc] == "cmp (sp)+,(sp)+"
                n += 4
        else if inst[cpc] == "add $x, sp"
                n += x
        else if inst[cpc] == "jmp x"
                cpc += x
                goto again
        else if inst[cpc] == "br .+x"
                cpc += x
                goto again
        return n/2
```

**14. [15 points]:** What the does the `mystery` function compute? (Please, give a brief explanation.)

*This mystery function determines* **the number of 2-byte arguments** *that were passed to the function that called it. For example, suppose we have a function* `f` *that takes a variable number of arguments, like* `printf`*, and it is called from another function* `g`*:*

```
void f(int x, ...) {
  printf("I was called with %d arguments!\n", mystery());
}

void g() {
  f(1, 2, 3, 4);
}
```

*Then in this case* `f` *will report that it was called with 4 arguments. The "mystery function" determines this number by scanning the code in* `g` *that will be run immediately after* `f` *returns, and counting the number of 2-byte words that* `g` *pops off the stack immediately after calling* `f`*. Since the return address to* `g` *is popped off by* `f` *itself while returning, and* `g`*'s local variables and callee-saved registers will be popped off later by* `g`*'s epilogue code, the stack pop instructions that the mystery function counts consists of exactly the argument words pushed onto the stack by* `g` *for* `f`*.*

*The correct functioning of this mystery function of course depends on the authors of the function knowing exactly which instructions the compiler might use to pop off arguments after calling a function. Kernighan and Ritchie, having written the C compiler along with the operating system, could be much more confident in this knowledge than most OS designers today would be.*

**Name:**

## IV 6.828

**15. [6 points]:** Describe the most memorable error you have made so far in one of the labs. (Provide enough detail that we can understand your answer.)

We love to know what your experience is in 6.828. Please, answers the following two questions. (Any answer, except no answer, will receive full credit!)

**16. [2 points]:** What is the best aspect of 6.828?

**17. [2 points]:** What is the worst aspect of 6.828?

# End of Quiz I

**Name:**