Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.828 Operating System Engineering: Fall 2004

Quiz 2 Solutions

(Thanks to Eddie Kohler and David Mazières for contributing many of the questions on this quiz.)

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.

1-4 (xx/35)	5-6 (xx/15)	7-8 (xx/20)	9-11 (xx/15)	12 (xx/10)	13-14 (xx/5)	Total (xx/100)

Name:

I Labs

Ursula Unsafe is working on Lab 4, System Calls for Environment Creation. Here is her first version of the `sys_mem_map` system call:

```
//
// Map the page of memory at 'srcva' in srcenvid's address space
// at 'dstva' in dstenvid's address space with permission 'perm'.
// Perm has the same restrictions as in sys_mem_alloc, except
// that it also must not grant write access to a read-only
// page.
//
// Return 0 on success, < 0 on error.  Errors are:
// -E_BAD_ENV if srcenvid and/or dstenvid doesn't currently exist,
// or the caller doesn't have permission to change one of them.
// -E_INVAL if srcva >= UTOP or srcva is not page-aligned,
// or dstva >= UTOP or dstva is not page-aligned.
// -E_INVAL is srcva is not mapped in srcenvid's address space.
// -E_INVAL if perm is inappropriate (see sys_page_alloc).
// -E_INVAL if (perm & PTE_W), but srcva is read-only in srcenvid's
// address space.
// -E_NO_MEM if there's no memory to allocate the new page,
// or to allocate any necessary page tables.
static int
sys_mem_map(uint_t srcenvid, u_int srcva,
             uint_t dstenvid, u_int dstva,
             u_int perm)
{
        struct Env* srcenv, *dstenv;
        struct Page* pg;
        Pte* pte;
        int retval;

        if ((retval = envid2env(srcenvid, &srcenv, 1)) < 0
            || (retval = envid2env(dstenvid, &dstenv, 1)) < 0)
                return retval;
        if ((srcva & (BY2PG - 1)) || (dstva & (BY2PG - 1)))
                return -E_INVAL;
        if (!(pg = page_lookup(srcenv->env_pgdir, srcva, &pte)))
                return -E_INVAL;
        return page_insert(dstenv->env_pgdir, pg, dstva, perm);
}
```

1. [10 points]: Describe two different ways that Ursula's `sys_mem_map` would let a user environment inject arbitrary code into the kernel. In particular, give two code fragments, including calls to `sys_mem_map`, after which the statement "`strcpy((char*) UTEMP, "Ha ha!");`" would write the string "`Ha ha!`" at kernel virtual address 0xF0003000.

Since the above implementation of `sys_mem_map` *does not check either* $srcva$ *or* $dstva$ *against* `UTOP`*, some of the possible ways are:*

- *Copy the kernel's mapping for the page at virtual address 0xF0003000 into a writeable, user-accessible mapping at* `UTEMP`*:*

  `sys_mem_map(0, 0xF0003000, 0, UTEMP, PTE_U|PTE_W|PTE_P);`
- *Allocate a new page at* `UTEMP` *and map this page into the kernel at virtual address 0xF0003000:*

  ```
  sys_mem_alloc(0, UTEMP, PTE_U|PTE_W|PTE_P);
  sys_mem_map(0, UTEMP, 0, 0xF0003000, PTE_U|PTE_W|PTE_P);
  ```
- *Use* `sys_mem_map` *to add* `PTE_U` *permission to the KVPD mapping (the current env's page directory), or similarly add* `PTE_W` *permission to the UVPD mapping, allowing the user env free write access to all of its own page tables. The user env can then directly map the kernel's page 0xF0003000 into its own space at* `UTEMP` *with simple memory writes.*

2. [10 points]: Describe one way that Ursula's `sys_mem_map` would let a user environment change arbitrary pages in *other* user environments. In particular, sketch out code steps (possibly using comments instead of actual C), including a call to `sys_mem_map`, after which the statement "`strcpy((char*) UTEMP, "Ha ha!");`" would write the string "`Ha ha!`" at environment `e`'s virtual address 0x80000. (Hint: Consider using `envs[e].env_cr3`.)

A few of the many possible ways:

- *Since the kernel maps all (usable) physical memory at* `KERNBASE`*, the user can use* `sys_mem_map` *to obtain access to its own page directory by copying the kernel's mapping for address* `KERNBASE+envs[e].env_cr3`*. It can then similarly find and map the page table for address 0x80000 in* e*, then finally find and map the page for address 0x80000 in* e *itself at* `UTEMP` *writeable into its own space.*
- *A quicker but more subtle approach: Just use* `sys_mem_map` *to make the* `envs[]` *table page containing* e*'s* `Env` *structure writeable in our address space, then change* `envs[e].env_parent_id` *to our own envid. We can then use* `sys_mem_map` *to transfer arbitrary mappings between our space and* e*'s because* e *is now our "child" and so the* `env2envid` *parent/child permission check will succeed.*
- *A more practically complex but totally general approach: patch some code into the kernel itself, e.g., replacing a frequently-called kernel routine such as* `env_run`*, with code that does whatever we want to do from within privileged mode. This uploaded "kernel" code could then just* `lcr3()` e*'s page directory and write into* e*'s address space using ordinary memory writes, for example.*

Here's a version of the user-level page fault handler you wrote in Lab 4.

```
// Page fault upcall entrypoint.
// This is where we ask the kernel
// to redirect us to whenever we cause a page fault in user space
// (see the call to sys_set_pgfault_handler in pgfault.c).
//
// When a page fault actually occurs,
// the kernel switches our ESP to point to the user exception stack
// if we're not already on the user exception stack,
// and then it pushes the following minimal trap frame
// onto our user exception stack:
//
// [ 5 spare words ]
// trap-time eip
// trap-time eflags
// trap-time esp
// tf_err (error code)
// fault_va <-- %esp
//
// We then have to save additional caller-saved registers
// and call up to the appropriate page fault handler in C code,
// pointed to by the global variable '_pgfault_handler' declared above.

.text
.globl _pgfault_upcall
_pgfault_upcall:
        // Save the caller-saved registers.
        movl %eax, 28(%esp)
        movl %ecx, 24(%esp)
        movl %edx, 20(%esp)

        // Call the C page fault handler.
        movl _pgfault_handler, %eax
        call *%eax

        // Push trap-time %eip and %eflags onto the trap-time stack.
        //
        // Explanation:
        //   We must prepare the trap-time stack for our eventual return to
        //   re-execute the instruction that faulted.
        //   Unfortunately, we can't return directly from this stack
        //   (the exception stack).  Why not?
        //   We can't call 'jmp', since that requires that we load the address
        //   into a register, and all registers must have their trap-time
        //   values after the return.
        //   We can't call 'ret' from the exception stack either, since if we
        //   did, %esp would have the wrong value.
        //   So instead, we push the trap-time %eip onto the *trap-time* stack!
        //   Below we'll switch to that stack and call 'ret', which will
        //   restore %esp to its pre-fault value.
        //   We'll also restore %eflags from the trap-time stack, in case an
        //   intervening instruction changes the flags.  ('ret' does not.)
        //
        //   In the case of a recursive fault on the exception stack,
        //   note that the two words we're pushing now will overlap with
        //   the current exception frame!
        //
        movl 8(%esp), %eax      // trap-time esp in eax
        subl $8, %eax           // add space for eip and eflags
        movl %eax, 8(%esp)

        movl 16(%esp), %ecx     // eip
        movl %ecx, 4(%eax)                        <---- CIRCLE HERE
        movl 12(%esp), %ecx     // eflags
        movl %ecx, 0(%eax)                        <---- CIRCLE HERE
```

Name:

```
// Restore the caller-saved registers.
movl 20(%esp), %edx
movl 24(%esp), %ecx
movl 28(%esp), %eax

// Switch back to the adjusted trap-time stack.
movl 8(%esp), %esp

// Restore eflags from the stack.
popf                                    <---- SQUARE HERE

// Return to re-execute the instruction that faulted.
ret                                     <---- SQUARE HERE
```

3. [5 points]: Fill in the blanks in the following exception stack layout, referring to the code above. The top two words are used only in case of a recursive fault (i.e., a fault in the page fault handler itself); additionally circle the two instructions above that will set these two words, and draw a box around the two instructions that will use them.

```
+------------------------+
|                        |
|     trap-time _eip_    |
+------------------------+   <-- 36(%esp)
|                        |
|     trap-time _eflags_ |
+------------------------+   <-- 32(%esp)
|                        |
|     trap-time _eax_    |
+------------------------+   <-- 28(%esp)
|                        |
|     trap-time _ecx_    |
+------------------------+   <-- 24(%esp)
|                        |
|     trap-time _edx_    |
+------------------------+   <-- 20(%esp)
|     trap-time eip      |
+------------------------+   <-- 16(%esp)
|     trap-time eflags   |
+------------------------+   <-- 12(%esp)
|     trap-time esp      |
+------------------------+   <-- 8(%esp)
|   tf_err (error code)  |
+------------------------+   <-- 4(%esp)
|        fault_va        |
+------------------------+   <-- %esp when handler is run
```

Name:

4. [10 points]: Fill in the following function, which implements simple shared memory.

```
// Allocate a page of memory at 'va' that is shared with environment 'e'.
// Do not return until environment 'e' has also called shmget() with
// similar arguments.
// Don't worry about race conditions.
// Assume that 'e' calls sys_ipc_recv only from shmget.
void shmget(void* va, envid_t e, int perm)
{
        assert(!((u_int) va & (BY2PG - 1)));

        // Is 'e' waiting in ipc_recv?
        if (envs[e].env_ipc_recving) {


                // allocate a page to share.
                sys_mem_alloc(0, va, perm);

                // Send a mapping of this page to env e.
                ipc_send(e, va, va, perm);


        } else {
                // Receive a page from 'e' mapped at 'va'.
                u_int va_from_e = ipc_recv(0, va, 0);
                assert(va_from_e == (u_int) va);
        }
}
```

Here are some header comments for reference.

```
// Allocate a page of memory and map it at 'va' with permission
// 'perm' in the address space of 'envid'.
// The page's contents are set to 0.
// If a page is already mapped at 'va', that page is unmapped as a
// side effect.
//
// perm -- PTE_U | PTE_P must be set, PTE_AVAIL | PTE_W may or may not be set,
//         but no other bits may be set.
//
// Return 0 on success, < 0 on error.  ...
int sys_page_alloc(u_int envid, u_int va, int perm);

// Try to send 'value' to the target env 'envid'.
// If va != 0, then also send page currently mapped at va,
// so that receiver gets a duplicate mapping of the same page.
//
// The send fails with a return value of -E_IPC_NOT_RECV if the
// target has not requested IPC with sys_ipc_recv.
//
// Otherwise, the send succeeds, and the target's ipc fields are
// updated as follows:
//    env_ipc_recving is set to 0 to block future sends
//    env_ipc_from is set to the sending envid
//    env_ipc_value is set to the 'value' parameter
// The target environment is marked runnable again.
//
// Return 0 on success, < 0 on error.
```

```
//
// If the sender sends a page but the receiver isn't asking for one,
// then no page mapping is transferred but no error occurs.
//
// srcva and perm should have the same restrictions as they had
// in sys_mem_map.
//
// Hint: you will find envid2env() useful.
int sys_ipc_can_send(u_int envid, u_int value,
                     u_int srcva, unsigned perm);

// Block until a value is ready.  Record that you want to receive,
// mark yourself not runnable, and then give up the CPU.
//
// Again, dstva should have the same restrictions as it had in
// sys_mem_map.  If it violates these restrictions, assume that it is
// zero.
static int sys_ipc_recv(u_int dstva)
```

II Brief paper questions

5. [10 points]: shell Answer the following question with respect to "*Es*: A shell with higher-order functions," by Haahr and Rakitzis. What will the following code print in the *es* shell?

```
fn xxx first rest {
    if {~ $#rest 0} {
        return $first
    }
    return <>{ xxx $rest } $first
}

echo <>{ xxx a b c }
```

Answer: `c b a`

6. [5 points]: Synchronization primitives Professor Dumbo wants to demonstrate a variant of the "ticket lock" to his class. Here's what he comes up with:

```
int nwaiting;

void acquire_lock() {
        atomic_inc(&nwaiting);
        while (nwaiting != 1)
                // wait for the other (nwaiting - 1) processes to finish
                pause(nwaiting - 1);
}

void release_lock() {
        atomic_dec(&nwaiting);
}
```

Give a specific scenario demonstrating that this implementaiton of the ticket lock is wrong.

There is more than one way this code can go wrong, but the simplest is probably:

1 `nwaiting` *starts at 0 (lock is not held).*
2 *Process 1 atomically increments* `nwaiting` *to 1.*
3 *Process 2 atomically increments* `nwaiting` *to 2.*
4 *Process 1 tests* `nwaiting`*, and goes into a loop because* `nwaiting = 2`.
5 *Process 2 tests* `nwaiting` *and similarly goes into a loop waiting for the lock.*
6 *The two processes are now deadlocked: neither of them can acquire the lock because each is waiting for the other to release it.*

III Microkernels

The paper "Improving IPC by Kernel Design," by Jochen Liedtke, describes a number of optimizations the author made to achieve good IPC performance in the L3 kernel. One such optimization involved the addition of new system calls. L3 contained two traditional calls for implementing IPC:

- **send** – send a message (asynchronously) to another process
- **receive** – wait for and return a message sent to a process

In addition, Liedtke added two more calls each of which combines the functionality of **send** and **receive**:

- **call**
- **reply & receive next**

7. [10 points]: Which of the following statements is true for the new **call/reply & receive next** interface, compared to equivalent code that issues a **receive** immediately following a **send** system call?

(Circle all that apply; there may be more than one answer.)

A. The new interface reduces the number of user-kernel crossings required for an IPC.

Yes: the total number of system calls for a typical client/server interaction is reduced from four to two.

B. The new interface reduces the number of TLB flushes required during an IPC.

No: Either way, the TLB must be flushed once when switching from client to server, then again on the switch from the server to client. With the old interface, a TLB flush isn't required on `send` *(which doesn't block or transfer control), only on* `receive` *(which does).*

C. The new interface lets L3 reduce the number of scheduler queue manipulations required during an IPC.

Yes: with the old interface, on a `send` *the kernel would have to add the receiving thread to the scheduler queue, and then on the sender's subsequent* `receive` *call the kernel would have to go back and find the receiving thread on the scheduler queue (which it put there on the previous system call), remove it, and transfer control to the receiving thread. With the new interface, the kernel can simply transfer control atomically from the sender to the receiver during the single* **call** *or* **reply & receive next** *system call, and avoid touching the scheduler queue at all.*

D. When no processes are swapped out to disk, the new interface reduces the number of times the kernel must copy IPC arguments and results that don't fit in registers.

No: Either way the kernel copies messages directly from the sender's address space to the receiver's, resulting in the same total amount of data copying. The receiver window trick makes this direct cross-space data copying possible, but that's a separate optimization not directly related to the combined control transfer interface for IPC.

E. The new interface makes the scheduler less fair, and could even lead to starvation were it not for the "long time wakeup list."

No: The new interface doesn't change the scheduler's normal behavior in any way. The optimization merely avoids the need to frequently put a thread onto the scheduling queue (during a `send`*) only to take it off again immediately (during a subsequent* `receive`*), effectively reducing the total number of scheduling operations that need to be done in the first place.*

To optimize data transfers during IPC, the kernel copies straight from the sender's address space into the receiver's, as follows. The kernel reserves a portion of virtual memory known as the communication window. The kernel also restricts each IPC argument to a maximum of 4 Megabytes. To copy into the recipient's address space, the kernel takes two page directory entries from the recipient's page directory and places them into the sender's at the virtual address of the communication window. (The kernel also clears the `PTE_U` bit, so that user code in the sender cannot access the memory mapped in the communications window.)

One concern is that, on the processor used by L3, it is expensive to flush TLB entries for a large virtual address range. (You can only flush one page translation at a time, or else the whole TLB.) Thus, Liedtke introduces the term *window clean*, to mean the TLB is guaranteed not to be caching any mappings for virtual addresses in the communication window.

8. [10 points]: Describe a scenario in which, when the kernel must copy data between address spaces, the TLB is *not* window clean. (Hint: feel free to use the old or the new system calls in your scenario.)

There are at least two such possible scenarios, and probably others:

- *Suppose that a thread performs several non-blocking* `send`*s in succession (perhaps to several different receivers) without intervening* `receive` *calls. The first* `send` *will use the window trick to copy the message into the receiver's address space, leaving the sender's TLB "dirty" with the temporary page mappings used to perform this copy. However, since* `send` *does not block the sending thread and change address spaces, it does not automatically flush the TLB. Thus the thread will* not *be window clean at the start of the next* `send` *operation, and the kernel will have to flush the TLB explicitly before re-using the window to copy the next message.*
- *Suppose that one thread sends a message and then blocks waiting for a reply. The TLB is now "dirty" because the kernel mapped in memory for the send. If another thread in the same address space is scheduled next, the TLB will not be flushed because the kernel does not need to change address spaces. Now if this second thread calls* `send`*, the kernel will need to manually flush the window to clean the TLB.*

IV Disco

Answer questions in this section with respect to the paper "Disco: Running Commodity Operating Systems on Scalable Multiprocessors" by Bugnion, Devine, and Rosenblum.

9. [5 points]: On page 11, the authors write, "The execution of the operating system in general and critical section in particular is slower on disco, which increases the contention for semaphores and spinlocks." What is the intuition for why contention increases, and what experiment demonstrates it?

Executing operating system code is much slower under Disco than without because of the virtualization overhead. In particular, the slowdown of OS code due to virtualization is disproportionate to the slowdown of application code because OS code tends to execute many privileged instructions and perform many low-level operations that require special handling by the VMM, which is where the main performance penalty lies. For this reason, when one processor holds a semaphore or spinlock while within the OS kernel, it will tend to take much longer to release it under Disco, so other processors will be more likely to come along during that time and create contention for the same semaphore or spinlock.

10. [5 points]: In Figure 5, why does "kernel time" decrease when running on Disco?

The Disco VMM performs some important tasks that the "stand-alone" IRIX kernel would normally have to do itself. For example, Disco allocates and initializes machine pages, thereby absorbing much of the memory access overhead that the IRIX kernel would normally incur performing these operations.

11. [5 points]: In Figure 6, why does Irix data grow? Why does buffer cache stay the same size for the M bar?

IRIX data grows because the kernel datastructures are now replicated in each instance of the kernel. The buffer cache is shared among all instances and therefore stays the same size.

V File systems

12. [5 points]: After finishing a Window systems for JOS, Austin Ports is evaluating whether to use soft-updates to improve the performance of the JOS file system. He keeps the layout and the basic file structures the same, but he would like to arrange that the JOS file server writes as many disk blocks asynchronously to the disk as possible, while maintaining file system consistency. Is there a scenario in which the file server must undo file system modifications when writing blocks out to disk to ensure file system consistency? If so, draw a picture with a dependency. (The next page shows the JOS file system structures.)

Yes. For example, suppose we have a file `foo` *in some directory, and* `foo` *has one data block attached to it. We delete* `foo`*, causing its one data block to be returned to the free block bitmap. We then create a new file* `bar` *in the same block of the same directory, then write a few bytes to it, causing the data block previously attached to* `foo` *to be re-allocated from the free block bitmap and attached to the new file* `bar`*. We now have a dependency chain as follows:*

Write directory block to remove `foo`
⇓
Write bitmap block to mark `foo`*'s old data block free*
⇓
Write directory block to create `bar`
⇓
Write bitmap block to mark `bar`*'s new data block in-use*
⇓
Write directory block to attach `bar`*'s new data block to* `bar`

Since the above logical writes alternate between two physical disk blocks, this sequence creates circular dependencies requiring the later logical writes to be rolled back in order to commit the earlier logical writes to disk. For example, the file system could roll back all but the first two writes, commit those two in order, then roll forward the next two writes and commit those in order, and finally roll forward the final write and commit that one.

13. [5 points]: What file system calls would add more dependencies? (Explain briefly why.)

The Unix `rename` *operation, for example, removes a file from one directory block and inserts it in another while ensuring that the file can never be "lost" if the system crashes in the middle of the operation. Ensuring this property creates a dependency between the insertion of the file in the new directory (which must happen first) and the removal of the file from the old directory (which must happen afterwards).*

```
// File nodes (both in-memory and on-disk)

// Bytes per file system block - same as page size
#define BY2BLK          BY2PG
#define BIT2BLK         (BY2BLK*8)


// Maximum size of a filename (a single path component), including null
#define MAXNAMELEN      128

// Maximum size of a complete pathname, including null
#define MAXPATHLEN      1024

// Number of (direct) block pointers in a File descriptor
#define NDIRECT         10
#define NINDIRECT       (BY2BLK/4)

#define MAXFILESIZE     (NINDIRECT*BY2BLK)

#define BY2FILE         256
struct File
  union
    struct
        uint8_t f_name[MAXNAMELEN];     // filename
        uint32_t f_size;                // file size in bytes
        uint32_t f_type;                // file type
        uint32_t f_direct[NDIRECT];
        uint32_t f_indirect;

        struct File *f_dir;             // valid only in memory
    ;
    uint8_t f_pad[BY2FILE];             // make sizeof(struct File) == BY2FILE
  ;
;

#define FILE2BLK        (BY2BLK/sizeof(struct File))

// File types
#define FTYPE_REG               0       // Regular file
#define FTYPE_DIR               1       // Directory


// File system super-block (both in-memory and on-disk)

#define FS_MAGIC        0x68286097      // Everyone's favorite OS class

struct Super
        uint32_t s_magic;       // Magic number: FS_MAGIC
        uint32_t s_nblocks;     // Total number of blocks on disk
        struct File s_root;     // Root directory node
;
```

```
/* from fs/fs.h */
int file_open(char *path, struct File **pfile);
int file_get_block(struct File *f, u_int blockno, void **pblk);
int file_set_size(struct File *f, u_int newsize);
void file_close(struct File *f);
int file_remove(char *path);
void fs_init(void);
int file_dirty(struct File *f, u_int offset);
void fs_sync(void);
void file_flush(struct File*);

/* from inc/fd.h */
struct Dev

        int dev_id;
        char *dev_name;
        int (*dev_read)(struct Fd*, void*, u_int, u_int);
        int (*dev_write)(struct Fd*, const void*, u_int, u_int);
        int (*dev_close)(struct Fd*);
        int (*dev_stat)(struct Fd*, struct Stat*);
        int (*dev_seek)(struct Fd*, u_int);
        int (*dev_trunc)(struct Fd*, u_int);
;

struct Fd

        u_int fd_dev_id;
        u_int fd_offset;
        u_int fd_omode;
;

struct Stat

        char st_name[MAXNAMELEN];
        u_int st_size;
        u_int st_isdir;
        struct Dev *st_dev;
;

struct Filefd

        struct Fd f_fd;
        u_int f_fileid;
        struct File f_file;
;
```

VI 6.828

We love to have your suggestions for improving 6.828. Please, answer the following question. (Any answer, except no answer, will receive full credit!)

14. [2 points]: If you could change one aspect of 6.828, what would it be?

15. [3 points]: Are there any topics you'd like to see added to the class, or any topics you'd like to see removed?

End of Quiz 2

Name: