Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.828 Operating System Engineering: Fall 2006

Quiz II

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.

In addition to this test booklet, you should have received a copy of selected files from the JOS Lab 6 distribution, which you may use for reference while answering the quiz questions. Please do not write any answers in those pages; that packet is yours to keep.

1 (xx/10)	2 (xx/10)	3 (xx/20)	4 (xx/20)	5 (xx/20)	6 (xx/15)	7 (xx/5)	Total (xx/100)

Name: Solution Set

Quiz score statistics: median 81, mean 80, maximum 98.

I Papers

1. [10 points]: True or false?

A. T / F : According to Liedtke in "Improving IPC performance by kernel design" a designer should exploit processor-specific features to get better IPC performance, sacrificing portability.

B. T / F : Disco (as described in "Disco: Running Commodity Operating Systems on Scalable Multiprocessors") pays a high overhead for TLB faults when running IRIX because the IRIX kernel occupies more TLB entries when running on Disco instead of on the physical hardware.

C. T / F : On a symmetric multiprocessor with a small number of processors and low contention, the time for acquiring an Mellor-Crummey & Scott (MCS) lock (as described in the "Algorithms for scalable synchronization on shared-memory multiprocessors") will be lower than a test-and-set lock with exponential back-off.

D. T / F : The methods described in "Bugs as Deviant Behavior: a general approach to inferring errors in systems code" can reliably determine which line of code contains an error.

E. T / F : Adding IPv6 support to Plan 9 (as described in "Plan 9 from Bell Labs") would require recompiling all the network client programs (telnet, ssh, etc.) to be IPv6-aware.

A. True. Liedtke strongly advocates to specialize the kernel implementation to exploit features of the underling hardware, in this case the x86. One x86 feature used is the page directory for fast temporary mappings.

B. True. Irix on Disco will incur more TLB faults than Irix on the physical hardware, because under Disco kernel addresses must be translated while on the physical hardware the kernel is in an untranslated part of the memory.

C. False. The test-and-set lock takes fewer instructions on acquire, and as can been seen in the performance figures it incurs a lower delay than MCS lock under low contention.

D. False. For MAYBE errors the approach cannot identify a specific line for an error. The approach reports only that there are inconsistencies.

E. False. Because Plan 9 treats network addresses as uninterpreted strings and relies on the connection server to translate names to connection instructions, network clients do not need to be rebuilt to add a new protocol.

II Memory-mapped files

Many operating systems support memory-mapped files. This feature allows a process to map a file into its address space and read and write the file using ordinary memory operations (e.g., `mov` instructions) instead of special `read` and `write` system calls.

The prototype for `mmap` and `munmap` from BSD Unix is:

```
mmap(void *addr, size_t len, int prot, int flags, int fd, off_t off);
munmap(void *addr, size_t len);
```

`Mmap` maps the *len* bytes of pages starting at *addr* so that they correspond to the *len* bytes starting at offset *off* in the file specified by the descriptor *fd*. The additional arguments *prot* and *flags* specify permissions and additional behaviors beyond the scope of this question. For this question, you may assume that *addr*, *len*, and *off* are multiples of the hardware page size and that *prot* and *flags* are set to create a read-write mapping where changes made to the mapped memory are eventually written back to the underlying file.

`Munmap` is the inverse of `mmap`: it flushes any modified memory pages back to the underlying file and then removes the mapping from the specified address range. Subsequent references to that address range will fault.

2. [5 points]: Describe how you would implement memory-mapped files in JOS. Describe the changes, if any, that would be necessary in the library operating system, in the buffer cache, and in the file server. Sketch pseudocode for the implementation of the `mmap` and `munmap` functions. (Hint: the implementations are very simple.)

Since JOS already maps its files into memory, the changes can be implemented entirely in the library operating system by copying the existing file mapping to the desired address.

```
mmap(addr, len, prot, flags, fd, off):
        for(i=0; i<len; i+=4096)
                sys_page_map(0, INDEX2DATA(fd)+off+i, 0, addr+i,
                             PTE_P|PTE_U|PTE_W)

munmap(addr, len)
        for(a=addr; a<addr+len; a+=4096)
                if(page at a is dirty)
                        dirty the underlying file page
                sys_page_unmap(0, a)
```

Name:

3. [5 points]: Can Multics applications read and write files using memory operations? If so, how is that implemented? (Answer this question at the high level; you don't have to go into great detail why not or how.)

Multics's segments, which are its equivalent of files, are only ever accessed by using memory operations. The operating system pages in sections of disk on demand.

III IPC and scheduling (thanks to Eddie Kohler)

4. [10 points]: Ben Bitdiddle is building a multi-environment application on JOS. Environments E1–En mostly send messages to environment E0. E0 generally processes these messages right away, but it must occasionally perform a lot of computation on a message, which delays its receiving the next messages. Ben expects E0 to take the whole CPU during this computation—after all, E1–En can't do anything useful during that time, since E0 isn't blocked in `sys_ipc_recv`. But Ben notices that E0 doesn't get as much CPU as it should. Instead, E1–En keep running, trying to send messages to E0. "This reminds me of receive livelock!" muses Ben: "I need a better IPC implementation."

Ben adds a new flag to the `sys_ipc_try_send` system call. If the flag is set, and the receiving process isn't ready, the sending process will block until the message is sent.

```
struct Env { ...
        bool env_ipc_recving;           // env is blocked receiving
        envid_t env_ipc_sending_to;     // env is blocked sending to this env
        ...
};
#define ENV_IPC_TRY_SEND_BLOCKED 3

static int
sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm,
                 int block)
{
        int r;    struct Env *e;   ...

        if ((r = envid2env(envid, &e, 0)) < 0)
                return r;
        if (!e->env_ipc_recving) {
                if (block) {
                        curenv->env_ipc_sending_to = e->env_id;
                        curenv->env_status = ENV_IPC_TRY_SEND_BLOCKED;
                        // just in case someone wakes us up early
                        // with sys_env_set_status():
                        curenv->env_tf.tf_regs.reg_eax = -E_IPC_NOT_RECV;
                        sched_yield();  // does not return
                }
                return -E_IPC_NOT_RECV;
        }

        ... rest of code as before ...
}
```

Name:

```
static int
sys_ipc_recv(void *dstva)
{
        ... ensure that !curenv->env_ipc_recving, as before ...
        ... check dstva, as before ...

        curenv->env_ipc_recving = 1;
        curenv->env_ipc_dstva = dstva;

        // look for an environment blocked
        for (i = 0; i < NENV; i++)

                if (envs[i].env_status == ENV_IPC_TRY_SEND_BLOCKED &&
                    envs[i].env_ipc_sending_to == curenv->env_id
                    _____________________________________________) {
                        // envs[i] has something to send to us!
                        // Mark it runnable, make it send to us,
                        // and if the send succeeds,
                        // return 0 from sys_recv.
                        struct Env *save_curenv = curenv;
                        curenv = &envs[i];

                        struct trapframe *tf = &curenv->env_tf;
                        int r = sys_ipc_try_send(tf->tf_edx, tf->tf_ecx,
                                                 tf->tf_ebx, tf->tf_edi,
                                                 tf->tf_ebx);
                        tf->tf_eax = r;
                        curenv->env_status = ENV_RUNNABLE;
                        if (r < 0) {
                          curenv = save_curenv;
                          continue;
                        }

                        ____________________________________________
                        curenv = save_curenv;
                        return 0;
                }

        // Otherwise, block as before
        ... rest of code as before ...
}
```

Fill in the blanks in Ben's `sys_ipc_recv` function. *Hint:* How will you make `envs[i]` send to `curenv`? Can you reuse `sys_ipc_try_send`? Remember that the five arguments to a system call are passed in registers `%edx`, `%ecx`, `%ebx`, `%edi`, and `%esi`, respectively. (Register `%eax` is used for the system call number.)

(The blanks are filled in in italics.)

Name:

Ben tries his system and observes that his fix causes `sys_ipc_recv`'s performance to slow down linearly with the *maximum* number of environments (`NENV`). Worse, Ben's system still suffers from *starvation*: environment E1 tries to send to environment E2, which continually tries to receive, yet E1 blocks forever because E0 always sends first. (The original system also suffered from the same starvation scenario.)

5. [10 points]: Sketch an implementation in precise English that would fix these problems, replacing the bold code in the `sys_ipc` system calls above. (A full solution would need to change `sys_env_set_status` and `env_free` too, but don't worry about those.)

Maintain in each `struct Env` a linked list of environments waiting to send to that one.
When `sys_ipc_try_send` blocks, it adds the current `Env` to the end of the target `Env`'s list.
When `sys_ipc_recv` checks for waiting sends, it consults the beginning of the list.

Name:

IV Unix Security and OKWS: Chroot

The `chroot` system call on Unix allows programmers and administrators to isolate programs in the file system, denying these programs access to sensitive data such as password files, private keys, or `setuid` executables. In a Unix process, `chroot("/tmp/siberia")` will set the directory `/tmp/siberia` as the new root directory: if the chrooted process then tries to access the file `/a/b`, the kernel will rewrite the access as though the process were accessing `/tmp/siberia/a/b`. This limits the process's file system access to the files in `/tmp/siberia` and its subdirectories, making inaccessible sensitive files like `/etc/passwd` and `/bin/su`.

The prototype for `chroot` from BSD Unix is:

```
int chroot(const char *newroot);
```

Where `newroot` is a directory on the file system. It returns `0` on success and `-1` on failure. Only root can call `chroot` successfully. Consider the implementation of `chroot` for JOS.

6. [5 points]: Where should most of `chroot` be implemented? Give short arguments for or against putting it in the JOS kernel (`kern/*`), the library operating system (`lib/*`), and the file server (`fs/*`).

It would be awkward to put in the kernel, because the kernel doesn't know about file systems.

It can't go in the library operating system, because then a malicious program could just not use it.

The file server must implement the actual rewriting of paths, since it is the final authority on file system operations.

7. [5 points]: Using the best of the three choices you just analyzed, describe an implementation of `chroot` for JOS. Be sure to explain how `chroot` affects later file system interactions by the same environment.

The file server can maintain for each `env_id` a file system root path. Chroot is then an IPC call that informs the file server of the new root path (interpreted relative to the current root path).

8. [5 points]: A sophisticated attacker might try to break out of a `chroot` by calling `fork` or `spawn`, hoping that the newly created environment would not be chrooted. Describe how you would need to change your implementation to prevent this attack. (If your implementation needs no changes, explain why not.)

One solution is to define that by default, a newly created environment has *no* file system access unless explicitly granted by its parent environment. Then in `fork` and `spawn` it will be up to the library implementation to grant access to the current root. If this granting is bypassed, the new environment will be *more restricted* than usual, not less.

(An equally correct, yet less exokerneley implementation of `chroot` would be to store the file system root as a variable in the `struct Env`, maintained by the kernel, and have a `sys_chroot` system call that appends to the current root path. The file server could read the root path from `UENVS` and `sys_exofork` could simply copy the root when creating the new environment.)

9. [5 points]: Chuck Phil O'Bugz recently wrote an application on BSD Unix that runs malware executable (`0wnU`) in a controlled sandbox. Chuck plans to observe the malware as it runs but wants to protect his private data from theft and wants to protect other running processes on the system from `0wnU`'s interference.

The problem is, Chuck mistakenly ran his code through `sort` and now it doesn't work anymore. Here is the code he has, with line numbers added:

```
 1 int launch_0wnU ()
 2 {
 3   int fd;
 4   char buf[20];
 5   int uid = 2001; // unprivileged UID
 6   int gid = 2001; // unprivileged GID

 7   chroot("/tmp/siberia");
 8   copy_program("/home/chuck/0wnU", "/tmp/siberia");
 9   execl("./0wnU", "-f", buf);
10   fd = open("/var/log/malware.log");
11   mkdir("/tmp/siberia", 0777);
12   setgid(gid);
13   setuid(uid);
14   sprintf(buf, "%d", fd);

15 }
```

Lines 7 through 14 are out of order. What is the correct order for those lines? Write your answer in the form "7,8,9,10,11,12,13,14".

One correct solution is 11,8,10,7,12,13,14,9.

The important constraints are:

11 before 8
8 and 10 before 7
10 before 14
7 and 12 before 13
13 and 14 before 9

Name:

V Buffer overruns

In the Lab 6 source packet you received with this test, you will find a copy of the JOS `user/sh.c`.

For the purposes of this question, assume that any code implemented outside `user/sh.c` and any code that you filled in (the `Your code here` sections) is free of buffer overruns.

You'll note that the top of the file contains the suggestive comment:

```
#define BUFSIZ 1024 /* Find the buffer overrun bug! */
```

But is there really a buffer overrun bug?

10. [5 points]: Which buffer is the comment talking about? Identify the name of the buffer and the line number where it is declared.

`argv0buf`, line 24

11. [5 points]: How can an attacker cause that buffer to overflow?

Pass a command with a very long program name that does not start with a slash. The `strcpy` on line 132 will overflow the buffer.

Name:

12. [10 points]: If the buffer does overflow, can an attacker use that overflow to run arbitrary code? If so, explain exactly how the attack works: draw a stack diagram showing which values are overwritten and explain how the control flow changes as a result. If the attacker cannot run such an attack against `user/sh`, explain why not.

The attacker *cannot* run arbitrary code: the buffer overflow can overwrite `runcmd`'s return address, but `runcmd` never returns after executing line 132.

(The buffer overflow might also overwrite any of the other local stack values, but changing those cannot cause any damage either.)

VI File descriptors

13. [5 points]: In Unix when a parent process forks a child process, the parent and process share open file descriptors. In particular, they share the offset. An alternative plan is for the parent and child not to share the offset but each have its own instance. Explain why sharing is a more desirable plan. Use a specific shell command to illustrate your explanation.

`(date; ls) >out` works better if the offset is shared. Otherwise `ls` overwrites the output of `date` when writing to out.

14. [5 points]: In JOS, to spawn a child process, the parent process must open and read the file that contains the child's program. Explain how the implementation of `spawn` might accidentally leave this file descriptor open in the child. What is the fix?

If the file descriptor is open when `spawn` copies the file table, then the child will inherit the open file descriptor.

The simplest fix is to close the file descriptor before copying the file table.

15. [5 points]: Besides not passing the grading script, what harm is there in leaving the file descriptor open in the child?

Eventually the environments will run out of file descriptors.

VII 6.828

We would like to hear your opinions about 6.828, so please answer the following two questions. (Any answer, except no answer, will receive full credit!)

16. [1 points]: On a scale of 1 to 10, please rate how much you learned from 6.828 (1 is low, 10 is high).

17. [2 points]: If you could change one aspect of 6.828, what would it be?

18. [2 points]: Are there any topics you would like to see added to or removed from the class?

End of Quiz II. Thanks for your participation and enjoy a well-deserved break!

Name: