Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.828 Fall 2009

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 finish this quiz.

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

Some questions may be 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 EXAM.

Please do not write in the boxes below.

I (xx/36)	II (xx/20)	III (xx/22)	IV (xx/16)	V (xx/6)	Total (xx/100)

Name:

I Paper questions

1. [6 points]: Many disks have a cache of disk blocks inside the disk and reorder writes to get better performance. Such reordering could cause problems for the version of ext3 as described by Tweedie in "Journaling the Linux ext2fs Filesystem;" give a scenario that results in inconsistent file system metadata structures.

2. [6 points]: How could you extend ext3 to fix this problem without disabling the disk's cache?

3. [6 points]: KeyKOS's use of capabilities is motivated by the "confused deputy" problem. Give an example of the confused deputy problem in Unix. Give an example how a KeyKOS application developer uses capabilities to avoid your example of the confused deputy problem in Unix.

4. [6 points]: To track dependencies introduced by Unix system calls, BackTracker (as described in "Backtracking Intrusions" by King and Chen) maintains objects for processes, files, and filenames. Ben thinks it's redundant to maintain both file objects, which identify files by inode numbers, and filename objects, which identify files by absolute path. He proposes modifying BackTracker to collapse the two into just filename objects, identifying files solely by their absolute path. Alyssa argues that this change will make BackTracker inaccurate (i.e., BackTracker could fail to identify an attack that influences a suspicious file or process). Give an example showing that Alyssa is correct.

5. [6 points]: Louis Reasoner proposes to extend BackTracker to recover from attacks by undoing all the attacker's actions, but not regular user actions. Using BackTracker, he can identify the starting event of an attack. He proposes to first use ReVirt to roll the system back to just before that starting event. For example, for the attack in Figure 1 of the paper, he would roll the system back to just before the "[sh,bash]" process was created. He would then *selectively* roll the system forward using ReVirt, skipping all the events that could have been influenced by the attacker, according to the dependencies that BackTracker has computed. For example, Louis' extension would not replay any operation that depended on the "[sh,bash]" node. Ben points out that this approach may fail to re-execute regular user actions. Give an example showing that Ben is right.

6. [6 points]: Figure 8 in the Klee paper ("Unassisted and automatic generation of high-coverage tests for complex systems programs") shows an example of a bug found by Klee in the `pr` utility. Specially, executing `pr -e t2.txt` will result in a buffer overflow if `t2.txt` has the content:

```
\b\b\b\b\b\b\b\t
```

Explain how Klee is able to construct this input.

Name:

II Lock-free data structures

Joe Buck is increasing the performance of the the Linux physical page allocator using a lock-free LIFO data structure to track free pages. He reasons that recently freed pages will still be in on-chip hardware caches, so allocating those pages first will improve performance. Joe Buck's code for the lock-free LIFO is below.

```
// Compare the value at addr with oldval.  If equal, store newval
// at addr and return 1, otherwise return 0.  Executes atomically.
int cmpxchg(void *addr, uint32_t oldval, uint32_t newval);

struct page {
    struct page *next;
    /* Other page metadata ... */
};
struct page *head;

void push(struct page *page) {
    while (1) {
        page->next = head;
        if (cmpxchg(&head, (uint32_t)page->next, (uint32_t)page))
            break;
    }
}

struct page * pop(void) {
    while (1) {
        struct page *page = head;
        if (page == NULL)
            return NULL;
        struct page *next = page->next;
        if (cmpxchg(&head, (uint32_t)page, (uint32_t)next))
            return page;
    }
}
```

7. [10 points]: Unfortunately Joe Buck's code has a race condition. Describe a sequence of events for two CPUs that could cause the LIFO to be in an inconsistent state (that is, an allocated page appears in the LIFO or an unallocated page does not appear in the LIFO). For your convenience, we provided the first two events of a sequence; your job is to add the others.

1. Initially, *head* → page A → page B → page C → *NULL*.
2. CPU 1 starts to `pop`; it sets `page = A` and `next = B`.

8. [10 points]: Some CPUs provide atomic instructions that make it easier to write lock-free code. For example, PowerPC CPUs provide the Load-Linked and Store-Conditional instructions. LL must be paired with a SC. LL loads a value from memory and the CPU remembers that the address is load-linked. SC stores a value to an address if the address has not been written to by any other CPU since the corresponding LL. Write a correct lock-free LIFO implementation that uses LL and SC.

```
// Return the 32-bit value at addr and remember addr is load-linked.
uint32_t ll(void *addr);
// Store val at addr if *addr has not been written to since the last ll from addr.
// Return 0 on failure, 1 on success.
int sc(uint32_t val, void *addr);

void push(struct page *page) {









}

struct page * pop(void) {









}
```

Name:

III VMM

Many x86 CPUs now have support in hardware for CPU, MMU, and IO port virtualization. The CPU virtualization features provide each guest VM with its own segment/interrupt descriptor tables, control registers, and segment registers. A virtual machine monitor (VMM) can use CPU virtualization to trap a VM when it executes an instruction or causes an event that could affect the VMM or other VMs. IO port virtualization allows VMMs to control VM access to IO ports, trapping into the VMM when the VM accesses certain IO ports.

9. [6 points]: AMD virtualization hardware allows the VMM to intercept certain CPU instructions and events by setting bits in a bitmask. As discussed in Section 4.4 of "A Comparison of Software and Hardware for x86 Virtualization" the performance of a VMM depends on the frequency of exits from the VM. To help avoid frequent exits, AMD hardware copies (or shadows) CPU state into a Virtual Machine Control Block (VMCB). Guest instructions that manipulate CPU state (e.g. `lcr0`) operate on the shadow state instead of the real CPU state. Circle the instructions that the VMM *must* intercept, because it is not possible to shadow the state they manipulate.

- Instructions that read or write control registers (e.g. `cr3`)
- Instructions that cause the computer to shutdown
- Instructions that load descriptor tables (e.g. `lgdt`)
- Instructions that clears cache contents without writing back to memory
- `pushf` and `popf`
- `iret`

10. [6 points]: You are unhappy with the performance of networking in JOS. On x86 hardware, port-based IO is not optimized and JOS's e100 driver is executing many `in` and `out` instructions, which hurt performance. The e100 exposes all of the same control registers using memory-mapped IO, which is much faster than port-based IO. Suppose we're running JOS in a virtual machine, but giving it direct access to our host's e100 card. Describe how you might use dynamic binary translation in the VMM to transparently convert JOS's e100 driver from using `in` and `out` port-based IO to using memory-mapped IO.

11. [10 points]: Modern MMU virtualization hardware works as described in Section 7.4 of "A Comparison of Software and Hardware for x86 Virtualization." As usual, guest VMs create *guest page tables* that translate from guest virtual addresses to guest physical addresses. However, with MMU virtualization, guest physical addresses are then further translated through a *nested page table*, created by the VMM, which maps guest physical addresses to host physical addresses.

When preparing a VM to run, the VMM must allocate physical pages for the guest's memory and initialize this nested page table. Write the code to do this below. Your implementation should allocate `npages` host physical pages and map them in the page table `pgdir` starting at guest physical address 0. Return 0 from the function on success, or return a negative value on an error.

```
/* Helper functions from JOS  */
int     page_insert(pde_t *pgdir, struct Page *pp, void *va, int perm);
int     page_alloc(struct Page **pp_store);
physaddr_t page2pa(struct Page *pp);

/* Guest permission bits */
#define PTE_GUEST       (PTE_P|PTE_W|PTE_U)

int
vm_init_mem(unsigned int npages, pde_t *pgdir)
{









}
```

IV High-performance JOS

12. [10 points]: Ben, a little sleep-deprived from the network lab, dreams of turning JOS into the next big, high-performance web serving platform. He took the easy route in his network lab and made his receive syscall always return immediately, even if no packet is available, which is less than ideal for CPU utilization. He starts by making his receive syscall block, much like `sys_ipc_recv`.

Ben keeps around his old non-blocking receive function for use in the new implementation. Now that he's no longer polling, he configures the e100 to generate an interrupt whenever it puts a packet in the receive ring and installs `e100_rx_intr` as the interrupt handler. Finally, since only one environment can receive at a time, he adds a few global variables to track the currently receiving environment.

```
// Copy the next packet into dst and return the length of the packet.
// If the receive ring is empty, return 0.
int e100_recv_nonblocking(void *dst);

// Environment currently blocked on receive, or NULL if none.
struct Env *e100_receiver;
// Packet buffer of the currently receiving environment.
void       *e100_receiver_dst;
```

Give pseudo-code for the new blocking syscall implementation and for `e100_rx_intr`. You can assume `dst` is a valid pointer to a sufficiently large buffer.

```
int sys_e100_recv(void *dst) {









}

void e100_rx_intr(void) {
    // Acknowledge the interrupt.  Otherwise, the e100 blocks further interrupts.
    e100_ack_rx();









}
```

13. [6 points]: Is your solution susceptible to receive livelock? If so, give a scenario that demonstrates the livelock. If not, explain why not.

V 6.828

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

14. [2 points]: The network lab took more time on average than we had intended. What would have saved you the most time? If it's test cases, which test cases would you add? If it's documentation, what would you like documentation on?

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

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

End of Quiz

Name: