Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.828 Fall 2010

Quiz II

All problems are open-ended questions. In order to receive credit you must answer the question as precisely as possible. You have 90 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/30)	**II (xx/46)**	**III (xx/16)**	**IV (xx/8)**	**Total (xx/100)**

Name:

I Short Paper Questions

You are using an xsyncfs file system as described in the paper "Rethink the Sync." You run the following program with its standard output connected to your terminal. Initially, the file xyzzy does not exist.

```
int main(void)
{
  // O_CREAT means create the file if it doesn't already exist.
  // O_WRONLY means open for writing.
  int fd = open("xyzzy", O_CREAT | O_WRONLY);
  write(fd, "one", 4);
  write(fd, "two", 4);
  close(fd);
  printf("done");
  return 0;
}
```

1. [8 points]: Sadly a power failure occurs just after you start the program, and you do not see `done` on the console. You restart the system (including the file system's recovery procedure) and look at file xyzzy. Write yes or no after each line, depending on whether it is possible state for xyzzy.

- xyzzy is empty
- xyzzy contains just `one`
- xyzzy contains just `two`
- xyzzy contains `onetwo`

Suppose now that you delete xyzzy and run the same program with its output redirected to file plugh:

```
$ program > plugh
```

2. [8 points]: Again, the power fails just after you start the program, and you don't see any output on your console. After restart you see that file plugh contains `done`. Which states for file xyzzy are possible now?

- xyzzy is empty
- xyzzy contains just `one`
- xyzzy contains just `two`
- xyzzy contains `onetwo`

The paper “A Comparison of Software and Hardware Techniques for x86 Virtualization” describes techniques by which a software VMM can detect guest kernel modifications to the guest’s page tables. The VMM needs to be aware of such modifications so that the VMM can reflect them in the “shadow” page table that the VMM installs in the real hardware %cr3. Indicate whether each of the following statements are true or false about the the paper’s description of how the VMWare software VMM deals with PTE writes:

3. [6 points]:

True / False : The VMM uses binary translation (BT) to insert instructions before every guest kernel memory-referencing instruction to detect whether the instruction modifies a PTE.

True / False : The VMM write-protects the pages that make up the guest’s page tables.

True / False : Each guest kernel modification of a PTE results in a page fault to the VMM.

True / False : The shadow page table is identical to the guest kernel’s page table, except that some PTE’s present or writable bits differ.

Mark each of the following statements according to whether it accurately reflects the content of the paper “Efficient System-Enforced Deterministic Parallelism.”

4. [8 points]:

True / False : The output of a program on Determinator cannot be influenced by any factors other than the program code itself.

True / False : A given program on Determinator always interleaves the execution of memory read and write instructions from different CPUs in the same order.

True / False : A program run on Determinator cannot have any bugs relating to concurrency.

True / False : The Determinator file system is implemented using Puts and Gets between the file system server and its clients.

True / False : Each Get in a parent must be matched by a Ret in the corresponding child.

True / False : A parent can use Get to observe the contents of a child space’s memory at any time.

II Concurrency and RCU

After taking 6.828, Ben is fascinated with concurrency, and decides to implement concurrent stacks. He starts with the following correct serial implementation of a stack:

```
elem_t *head;    // the top of the stack

void push(int key, int value)
{
  elem_t *e = malloc(sizeof(*e));
  e->next = head;
  e->key = key;
  e->value = value;
  head = e;    // Put it on the stack
}

elem_t *pop(void)
{
  elem_t *e = head;
  if (e) head = e->next;
  return e;
}

elem_t *search(int key)
{
  for (elem_t *e = head; e; e = e->next) {
    if (e->key == key) {
      return e;
    }
  }
  return NULL;
}
```

Ben wants to run this code on his multicore computer. The cores share a cache-coherent memory and Ben places `head`, `freelist`, etc. in shared memory, so that different cores can push and pop from the shared stack.

5. [8 points]: Ben is pretty sure this code cannot run correctly on a multicore computer. That is, if two threads on different cores concurrently invoke `search`, `push`, and `pop`, then there can be races. To refresh Ben's mind, please give an example of a non-benign race in Ben's code.

Ben decides to make the implementation correct using a read-write lock. Here are the locked versions of push and search:

```
struct rwlock rwlock;

void locked_push(int key, int value)
{
  acquire_write(&rwlock);
  push(key, value);
  release_write(&rwlock);
}

elem_t *locked_search(int key)
{
  acquire_read(&rwlock);
  elem_t *e = search(key);
  release_read(&rwlock);
  return e;
}
```

Ben implements read/write locks correctly: either one writer can acquire the lock in write mode, or several readers can acquire the lock in read mode. The acquire parts of his read/write lock are. You don't need to understand them in detail.

```
struct rwlock {
  spinlock_t l;
  volatile unsigned nreader;
};

void acquire_read(struct rwlock *rl)
{
  spin_lock(&rl->l);
  atomic_increment(&rl->nreader);
  spin_unlock(&rl->l);
}

void acquire_write(struct rwlock *rl)
{
   spin_lock(&rl->l);
   while (rl->nreader) /* spin */ ;
}
```

6. [6 points]: If Ben invokes many `locked_search`'s concurrently, why does the version with a read/write lock perform better than if Ben had used a regular spin lock?

7. [6 points]: Ben observes that even with a read/write lock and no concurrent `push`'s or `pop`'s, 48 concurrent `locked_search`'s run slower than 48 concurrent `search`'s. Explain briefly why.

Inspired by the RCU paper ("Read-copy update" by McKenney et al.), Ben decides to create an RCU-like list implementation, but based on time instead of quiescent states. He adds two fields to `struct elem_t` (`time` and `delayed_next`) and introduces the following RCU functions:

```
elem_t *delayed_freelist;
int safe_time[NUMCPU];

void rcu_free(elem_t *e)
{
  e->time = time_since_boot();
  e->delayed_next = delayed_freelist;
  delayed_freelist = e;
  rcu_gc();
}

void rcu_safe(void)
{
  safe_time[mycpu] = time_since_boot();
}

void rcu_gc(void) { /* .. see below .. */ }
```

When a thread wants to free an element, it calls `rcu_free` to inform RCU that it is done with the element. The element cannot be reused yet, because other threads may have a reference to it, so `rcu_free` adds the element to the `delayed_freelist`. When a thread has no references to any element, it informs RCU by calling `rcu_safe`. You can assume that `time_since_boot` is very cheap to execute. The `rcu_gc` function frees elements that are safe to free (i.e., elements that cannot be in use by any thread).

An example of how these functions are intended to be used is as follows:

```
void thread1(void) {
  elem_t *e;
  while (1) {
    acquire_write(&rwlock);
    if ((e = pop())) {
      compute1(e);
      rcu_free(e);
    }
    rcu_safe();
    release_write(&rwlock);
  }
}

void thread2(void) {
 elem_t *e;
 while (1) {
   if ((e = search(random())))
      compute2(e);
   rcu_safe();
  }
}
```

This code assume there is a list with many elements. Thread 1 repeatedly pops an element from this list, computes with it, `rcu_free`'s it, and informs RCU it has no references to any element. Thread 2 searches the list, computes with the found element, and informs RCU it has no references to any element.

8. [8 points]: Why must RCU delay free operations? Give an example scenario and an explanation of what might go wrong if RCU didn't delay free operations.

9. [10 points]: Complete pseudocode for the RCU garbage collector (`rcu_gc`) that frees all elements from the delayed free list that can safely be freed (use `free` to free the individual elements).

```
void rcu_gc(void)
{









}
```

Name:

Consider the following two variations on `thread2` (differences underlined):

```
void thread2_unlocked(void) {
 elem_t *e;
 while (1) {
   int key = random();
   if ((e = search(key))) {
      compute2(e);
   }
   // No rcu_safe();
  }
}

void thread2_locked(void) {
 elem_t *e;
 while (1) {
   int key = random();
   if ((e = locked_search(key))) {
      compute2(e);
   }
  }
}
```

10. [8 points]: Assuming no concurrent `push`'s or `pop`'s, would 48 concurrent `thread2`'s perform as fast as 48 concurrent `thread2_unlocked`'s, as slow as 48 concurrent `thread2_locked`'s, or somewhere between? Explain briefly.

III Network driver

Late one night, Ben Bitdiddle is tooling away at his E100 receive implementation for the network lab. For reference, the receive frame descriptor (RFD) format is

Command word						Status word			
EL	S	000000000	H	SF	000	C	0	OK	Status bits
Link address									
Reserved									
0	0	Size				EOF	F	Actual count	

He implements receive as follows:

```
// The number of RFD's in the receive ring
#define NUM_RFDS 16
// Pointers to the RFD's in the receive ring
static struct rfd *rfd_ring[NUM_RFDS];
// The index of the next RFD to read a packet from
static unsigned int rfd_tail;

// Receive one packet and store it in to buf.
int e100_receive(char *buf)
{
  struct rfd *rfd = rfd_ring[rfd_tail % NUM_RFDS];
  if (!(rfd->status & RFD_STATUS_C))
    // No packets in receive ring
    return 0;
  // Clear completed ("C") bit
  rfd->status = 0;
  // Copy packet data
  int actualCount = rfd->actualCount & 0x3FFF;
  memmove(buf, rfd->data, actualCount);
  // Move to the next entry in the ring
  rfd_tail++;
  return actualCount;
}
```

Name:

Ben tests his implementation using the echo server and finds that it works. Proud of his succinct implementation, he shows it off to Louis Reasoner, who spots a bug.

11. [8 points]: Louis points out to Ben that he needs to manipulate the suspend ("S") bits in the RFD's, so Ben sets the S bit on `rfd_ring[NUM_RFDS-1]` during initialization and adds the following code just after the `memmove` in `e100_receive`:

```
// Set the suspend bit on this RFD
rfd->command = RFD_COMMAND_S;
// Clear the suspend bit on the previous RFD
rfd_ring[(rfd_tail + NUM_RFDS - 1)%NUM_RFDS]->command = 0;
```

Give an example of a problem that Ben's addition prevents.

12. [8 points]: Ben again tests his code using the echo server. Now confident in his implementation, Ben shows it off to Alyssa P. Hacker, who informs Ben that, while the change he just made was important, it missed something: just after clearing the suspend bit on the previous RFD, he needs to issue an "RU resume" command. Give an example of a problem that Alyssa's addition fixes.

IV 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.)

13. [2 points]: In the second half of the term, we read papers instead of xv6 source code. Which papers did you like? (We read ext3 Journaling, Rethink the Sync, Eliminating Receive Livelock, KeyKOS, Singularity, Anderson Locks, RCU, Barrelfish, Determinator, Software vs Hardware Virtualization, SMP-ReVirt, and Ksplice.)

14. [2 points]: With respect to the labs since quiz 1, are those labs too time consuming, too short, or are they about right? (Lab 4 was COW fork, preemptive multitasking, and IPC; lab 5 was the file system; lab 6 was the network; and lab 7 was the shell and project.)

15. [2 points]: Now that you have completed 6.828, what is the best aspect of 6.828?

16. [2 points]: Now that you have completed 6.828, what is the worst aspect of 6.828?

End of Quiz