

Systems Comprehensive Exam, Fall 2007

August 13, 2007

1 Instructions

This is a closed-book, closed-notes exam with a total of 100 points. You may not use any external source for answering these questions, including but not limited to the Internet, books, notes, or other people. Please direct any questions about this exam to Professor Bridges. Professor Bridges may be reached either in person in his office in 301B Farris, by phone at 277-3032 or 363-8798, or by email at `bridges@cs.unm.edu`. Turn your exam in to Professor Bridges or the front office by 5:00 PM MDT on Wednesday, August 13, 2007. Exams *will not* be accepted after this time except by prior arrangement with Professor Bridges.

Type or write your answers to the stated number of questions in each of the following three sections. Make any *reasonable* assumptions necessary to answer the question, but be sure to state any assumptions that you make.

2 Short Answer - Answer 3 of 4 (30 points)

Briefly answer 3 of the following 4 questions. Your answer should be no longer than *one* paragraph.

1. One feature of modern architectures and compilers is that they can *reorder* memory writes from a single thread to improve pipeline and memory system (e.g. caching) performance. Unfortunately, this reordering can make it very difficult to write concurrent programs. Most architectures provide an (expensive) `write_barrier()` operation across which memory reads and writes cannot be reordered, which allows programmers to address this issue.

The code in figure 1 implements a simple lock-free single-reader/single-writer FIFO ring buffer. Indicate where *if anywhere* write barriers must be added to guarantee correctness of this code on a modern multiprocessor system, and what specific error each write barrier you add is needed to guard against. Because write barriers are expensive on multiprocessor systems, it is important that you not add any unnecessarily!

2. Describe briefly what service Lamport's clock algorithm provides in a distributed system, and provide a brief (e.g. 3 sentence) overview of how the algorithm works.
3. How does the IP protocol implement datagram fragmentation and reassembly? In particular, what elements of the network fragment, which elements reassemble, and what data is stored in each fragment to facilitate reassembly?
4. Briefly contrast sequential and causal consistency in a group RPC system.

```

1     typedef struct ringbuffer {
2         /* Buffer is empty if rb->head == rb->tail. Buffer is full if
3          * next(rb->tail) == rb->head */
4         void **head, /* Points to first occupied element in buffer */
5         **tail, /* Points to first free element in buffer */
6         void **buf, /* Points to the start of the allocated ring */
7         **end; /* Points one element *past* the allocated ring */
8     } ringbuffer_t;
9
10    void **RingbufferNext(ringbuffer_t *rb, void **item)
11    {
12        void **next = item + 1;
13        if (next >= rb->end)
14            return rb->buf;
15        else
16            return next;
17    }
18
19    int RingbufferAppendSingle(ringbuffer_t *rb, void *value)
20    {
21        void **next = RingbufferNext(rb, rb->tail);
22        if (next == rb->head)
23            return -1;
24        *rb->tail = value;
25        rb->tail = next;
26        return 0;
27    }
28
29    int RingbufferRemoveSingle(ringbuffer_t *rb, void **value)
30    {
31        if (rb->head == rb->tail)
32            return -1;
33        *value = *rb->head;
34        rb->head = RingbufferNext(rb, rb->head);
35        return 0;
36    }

```

Figure 1: A simple single-reader, single writer FIFO ring buffer.

3 Medium Answer - Answer 2 of 3 (40 points)

Provide detailed answers to two of the following three questions. Be sure to state any assumptions you make and to fully justify your answers. Limit your answers to approximately one to two pages in length.

1. **Swapping.** Swapping is a particularly tricky issue in UNIX-like operating systems because, as a “slow” operation that requires I/O, it may put a process to sleep in the kernel, potentially allowing multiple processes to be swapping in or out the same page simultaneously. Linux addresses this issue by introducing a “swap cache”, with the key rule being that no process can start a swap-in or swap-out without checking whether the swap cache already includes the affected page.

Put them in the order that they must occur in to avoid synchronization issues:

- A. Read the page from the swap drive
- B. Restore the page table entry for any process that has faulted on the read of the page
- C. Free the physical page to the system
- D. For all processes, change all page table entries from a page table entry to a swap cache entry
- E. Put the page in a swap cache
- F. Write the physical page to the swap drive
- G. Remove the page from the swap cache when no processes have a swap cache entry for it any longer
- H. Obtain a physical memory page

Note: only processes with page table entries can access the physical page, and processes with swap cache entries will fault to the kernel.

2. **Synchronization.** Referring again to the code in figure 1 which supports only a single producer and a single consumer, not multiple concurrent producers, or consumers:

- A. How could this code be modified with locks (either semaphores or mutexes/condition variables) to support multiple concurrent producers and/or consumers? Be specific, being sure to describe exactly what variables would need be added and where those locks would be acquired and released.
- B. Assuming the existence of an *atomic* “compare-and-swap” instruction defined as:

```
boolean compare_and_swap(void **dest, void *orig_val, void *new_val) {  
    if (*dest == orig_val) { *dest = new_val; return true; }  
    else return false;  
}
```

describe how the code in figure 1 could be modified to support multiple concurrent producers and consumers *without* the use of locks.

- C. Describe in detail when each implementation (lock-based and lock-free) would be desirable from a performance standpoint being sure to describe the advantages and disadvantages of each version with specific reference to, for example, the locks, loops, or compare-and-swap instructions in the code you described in parts (A) and (B) above.

(Note: Assume for simplicity’s sake that you need not worry about write barriers in this problem.)

3. **Distributed Systems.** Ring algorithms are frequently used in distributed systems to implement a broad range of distributed services. Describe in detail one distributed algorithm based on a ring topology and contrast its tradeoffs versus another algorithm for implementing the same service in terms of performance and fault tolerance.

4 Long Answer - Answer 1 of 2 (30 points)

Provide a *full* and *detailed* answer to one of the following two questions. Be sure to state any assumptions you make and to fully justify your answer.

4.1 Distributed Sensor Systems

Sensor networks are being deployed in a wide range of situations, and supplemented with large-scale back-end cluster systems for real-time analysis of data acquired by sensors. However, large-scale disruptions, for example natural disasters or adversarial attacks, are a potential difficulty in these systems. In particular, such disruptions may cause sensors or back-end analysis systems to become unavailable. How would you design system software support for such a sensor/analysis system so that online users of the sensor network could still have real-time access to (potentially degraded) sensor network data and analysis information in the face of network disruption? Be specific in terms of the system services that you would provide, the challenges these services would have to deal with, and how you would address these challenges.

4.2 Security and Memory Systems

For security reasons researchers have proposed randomizing the address space layout of processes in the system. Assume this is possible at the page granularity, so that every virtual memory page of a process will now have a random location in the virtual memory address space (but the total amount of virtual memory consumed remains unchanged).

- A. How might this improve system security?
- B. How could this be implemented on modern hardware and operating systems, for example in dynamically linked Linux programs running on a machine with 32-bit addresses and a 2-level page table (Assume a 10-10-12 bit virtual address like the Pentium)?
- C. How would different operating system, application, and hardware configurations effect the implementation and performance of this approach? Consider, for example:
 - 64-bit virtual address spaces with a 3-level hierarchical page tables
 - 64-bit virtual address spaces with inverted page tables
 - Static program linking
 - OS-level virtualization a la Xen