

**CSC 469H1 F / CSC 2208H1F
ADVANCED OPERATING SYSTEMS**

**UNIVERSITY OF TORONTO
Fall 2007**

Term Test #1

NO AIDS ALLOWED

Please **PRINT** in answering the following requests for information:

Family Name: _____

Given Names: _____

Student Number: | _ _ _ | | _ _ _ | | _ _ _ |

Login (@cdf): _____

Notes to students:

1. This test lasts for 110 minutes and consists of 78 marks. Budget your time accordingly.
2. This test has 7 questions and 11 pages (including this one); Check that you have all pages before starting.
3. **Write in pen. No pencils. Really, I mean it.**
4. Write your answers on this “question and answer” paper, in the spaces provided. Be concise. In general, the amount of space provided is an upper bound on the “size” of answer that is expected. If necessary, use space where available and provide explicit pointers.
5. State your assumptions and show your intermediate work, where appropriate.
6. Do **not** go beyond here until instructed to do so. Write your student number at the top of each succeeding page once you get going.

Question	Marks
1	/15
2	/12
3	/12
4	/12
5	/12
6	/6
7	/9
Total	/78

1. [15 marks, 3 each] Definitions

Define (or explain) the following terms, in the context of this course:

a) End-to-end argument

Principle for system design that says you should not implement some feature in low-level software if higher level software must be involved to really make the feature correct. Support for the feature at lower levels should be seen as a performance optimization, rather than a correctness criteria.

b) Hosted virtual machine

Type of virtual machine that provides a complete duplicate of the physical machine by hosting the virtualizing software (i.e. the VMM) as an ordinary process on top of an existing OS installation. An example is VMWare Workstation.

c) Signal delivery

The point at which a process recognizes the arrival of a signal event (i.e. sees that a signal was posted), and the action taken by the process at this point to handle the signal.

d) test-and-test-and-set

A type of spinlock that reduces the excess memory traffic of a test-and-set based lock by repeatedly checking if the lock is free using an ordinary read instruction, and only performing the atomic test-and-set when the lock is observed to be free. This allows the lock value to be read from the cache while a thread is spinning.

e) Backfilling

Enhancement to FCFS parallel job scheduling that allows jobs to move ahead of a blocked job in the scheduler queue to take advantage of CPUs that would otherwise be idle.

2. [12 marks; 6 each] Structure and Performance

Operating system primitives (such as system calls and trap handling) generally require architecture support, however, the performance improvements for these primitives from new hardware rarely match those seen for application code. At the same time, improvements in operating system structure often come at the price of increased overheads for applications with the result that new operating systems are slower than old ones. The data in the two tables below is extracted (and simplified) from a 1991 paper by Anderson et al. that explored these two trends:

Operation	Avg. Speedup
Operating System Primitives	3.9
Application Performance	6.7

Table 1: Average Speedup of CPU2 (newer) over CPU1 (older) on operating system primitives, and on application code.

	Time (sec.)	Address Space switches	System Calls	Emulated Test-and-Set	% Time In OS Primitives
Monolithic	69.3	2336	5513	320	N/A
Microkernel	80.9	16208	16561	213781	5%

Table 2: Performance and number of operating system events for latex formatting a 150 page document on CPU2 with monolithic and microkernel OS. Time in OS primitives **does not** include time executing the system calls themselves, just the overhead of making them. The microkernel adds about 17% to the run time.

a) Suppose I have just replaced my uniprocessor desktop system (running the monolithic OS on CPU1) with a new uniprocessor system running the microkernel OS on CPU2. Suppose also that I spend most of my computing time formatting 150 page latex documents, and am annoyed to discover that my application speedup is only 5.6. Which of the two trends noted above is likely to be the largest contributor to reduced speedup, and why? Base your argument on the data in Tables 1 & 2 and Amdahl's Law.

Amdahl's Law tells us that the amount of speedup we can expect from an optimization is limited by the fraction of the program to which that optimization applies. The same can be said for performance degradations. Even though the speedup of 3.9 for OS primitives is substantially less than 6.7, only 5% of the application run time is spent in OS primitives on the new system, thus they cannot be responsible for much of the observed loss of speedup. We should blame instead the change in OS structure.

b) Explain the differences in (i) numbers of address space switches, (ii) system calls, and (iii) emulated test-and-set instructions shown in Table 2 based on the structural differences between microkernel and monolithic operating systems. (CPU2 did not provide a hardware atomic test-and-set instruction, it had to be emulated in the kernel instead.)

(i) In a microkernel, large parts of the traditional OS function are implemented as user-level servers, running as separate processes. Requesting any OS service requires a context switch to the server process, and then back to the latex process. Also, some services that may have needed one system call in the monolithic case may require multiple requests to different servers, leading to large increases in address space switches.

(ii) Each request for OS service in a microkernel still requires a system call (often two - one for the request and one for the reply) since the microkernel is required to handle the IPC between the latex process and the server processes. Not surprisingly, the number of system calls will be increased.

(iii) This is the trickiest one to explain. Both the monolithic kernel and the server processes of the microkernel contain critical sections. In the monolithic kernel, however, they can be implemented by disabling and restoring interrupts around the critical section code (recall this is a uniprocessor). In the microkernel, however, the user-level servers cannot manipulate interrupts, and instead uses locks based on the atomic test-and-set instruction to implement critical sections. In the absence of this instruction on the target architecture, they are emulated, and we see large increases in the number of these emulated test-and-sets when the more critical sections appear in user-level code.

3.[12 points] Performance**Evaluation & Benchmarking**

a) [6 points] For portability, lmbench uses the `gettimeofday` system call to obtain timing measurements. However, different systems provide different resolutions for `gettimeofday`, from 1 microsecond up to 10 milliseconds, and there is no portable system call to obtain the resolution. Write a function that experimentally determines the resolution of `gettimeofday`, returning the result in microseconds.

```

struct timeval {
    long tv_sec; /* seconds */
    long tv_usec; /* microseconds */
};
/* simplified prototype for gettimeofday */
int gettimeofday(struct timeval *tv);

long get_resolution() {

    struct timeval t1, t2;
    gettimeofday(&t1);
    do {
        gettimeofday(&t2);
    } while (t1.tv_sec == t2.tv_sec && t1.tv_usec == t2.tv_usec);

    /* readings differ, return how much they differ by */
    return ((t2.tv_sec - t1.tv_sec)*1000000 + (t2.tv_usec - t1.tv_usec));

}

```

b) [3 points] Why is it important to know the timer resolution when conducting a performance experiment?

The resolution of the timer determines how much error there may be in our measurements. For a resolution of d , error could be $\pm d$. We can control the error by ensuring our experiment runs long enough, but first we need to know what resolution is in order to determine how long our experiment must run.

c) [3 points] Aside from timer resolution, what else should you be careful of when using `gettimeofday` to measure elapsed time?

`gettimeofday` measures "wall clock" time. It will therefore include time of other activities if a context switch occurred. We should consider whether this is what we want to measure, and if not, how to reduce the likelihood of context switches (using a lightly loaded system) or detect if they have occurred. [In determining the timer resolution, above, we should take the minimum of several readings from the function, since it is possible that a context switch occurred between reading $t1$ & $t2$.]

4. [12 points; 4 each] Interrupts & IPC

a) With respect to BSD Unix (pre FreeBSD 5.2), identify two limitations of executing in "interrupt context", and explain how interrupt handlers deal with these limitations.

In interrupt context we "borrow" the context of the process we interrupted. That means (i) limited stack space and (ii) we are not allowed to block. This is dealt with by splitting interrupt handling into two parts: the part that must happen immediately in interrupt context and the part that can be deferred for later when the code can run in a normal context.

b) Explain how FreeBSD determines to which thread a signal should be posted in a multi-threaded process.

Synchronous signals (those generated by a thread's own actions) are posted to the thread that caused the signal. For asynchronous signals, the list of threads in the process is scanned until one is found that is not blocking the signal, and the signal is posted to it. If all threads are blocking the signal, it is posted to the process overall, and will be picked up by the first thread to unblock the signal.

c) Identify (i) why *select* is not usable for event notification when large numbers of descriptors are involved, (ii) the primary problem with using *poll* instead, and the general solution to this problem as implemented by Linux *epoll*, Solaris */dev/poll*, or FreeBSD *kqueue*.

(i) select uses bitmasks of file descriptors to indicate which fd's it is interested in, and what activity has occurred. This limits the number of fd's to a maximum of 32. (ii) poll can pass an arbitrary number of descriptors, but this means a lot of data is passed between user and kernel, and a lot of work is required to scan the list (up to 3 times) in both the kernel and application. The general solution separates the expression of interest in an event from notification of that event, so that the complete set of descriptors does not need to be passed and scanned each time.

5. [12 points] Read-Copy Update

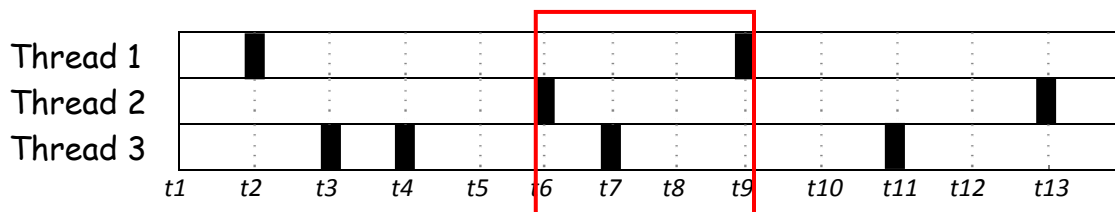
a) [2 points] Suppose we have an implementation of linked lists using read-copy update, in which threads acquire a spinlock when writing, but readers can proceed without acquiring any locks. QSBR ensures that memory is reclaimed safely. Is this implementation non-blocking? Why or why not?

No. A writer that fails while holding the spinlock will prevent other writers from making progress. That is, it can block other writers and is thus not non-blocking.

b) [1 point] Suppose that the spinlock is replaced with a reader-writer lock. Is this new implementation non-blocking? Why or why not?

No. This change is pointless. Readers never required a lock, and we still have the same problem with writer failure while holding the lock (in fact, this makes the problem slightly worse, since a failed reader can now prevent writer progress!)

c) [1 point] Assume in the picture below that prior to time t_1 , no thread has gone through a quiescent state. Black boxes represent quiescent states. Identify as a time interval $[a,b]$ the smallest grace period in the system (assume that quiescent states happen instantaneously at the times marked).



$[t_6, t_9]$

d) [4 points] Below is an incorrect implementation of an insertion function for a doubly-linked list using RCU. The function is given a new node to be linked in between two existing consecutive nodes (prev and next). The caller is responsible for acquiring the necessary writer locks. Explain what is wrong with the existing function, and then write the corrected body of the function. Assume sequential consistency in your answer.

```
void list_add_rcu(
    struct list_head *new,
    struct list_head *prev,
    struct list_head *next)
{
    prev->next = new;
    next->prev = new;
    new->prev = prev;
    new->next = next;
}
```

```
Corrected body: {

    new->next = next;
    new->prev = prev;
    next->prev = new;
    prev->next = new;

}
```

The original function links new into the list (by updating the prev and next nodes) before setting the next and prev fields of the new node. Concurrent readers could thus reach new while traversing the list and either crash when they tried to follow its next/prev pointers (if they contained garbage values) or conclude incorrectly that the list had ended (if they were initialized to NULL prior to the call to list_add_rcu).

e) [4 points] Suppose we have several reader threads that obtain data from a structure containing many elements. Periodically, a single writer will modify the contents of this structure, and we require that readers observe either all the updates, or none of them. To avoid locking we keep two copies of the structure in an array (named *copies*); a variable (named *active*) stores the index of the active copy used by readers, while the writer updates the inactive copy and then updates the *active* variable so that readers will start using the new version of the structure. Code for the writer is shown below:

```
void update_structure(int newval1, int newval2, int newval3)
{
    /* get index of inactive copy */
    int inactive_index = 1 - active;
    /* update fields in inactive copy */
    copies[inactive_index].field1 = newval1;
    copies[inactive_index].field2 = newval2;
    copies[inactive_index].field3 = newval3;
    /* make inactive copy the new active copy */ write_fence();
    active = 1-active;
}
```

If we do not assume sequential consistency, show the fence(s) that are needed on the code above (identify the position and type of fence), and explain why they are needed.

We must guarantee that the updates (writes) to the fields of the structure complete before the update (write) to the active index. Otherwise readers could start using the new active copy but still observe some or all of the old values. We do this by inserting a write fence (or write memory barrier) before the update of active, as shown.

6. [6 points] Transactional Memory

Below is the code for the push and pop operations on a stack that uses transactional memory (TM). Show that the TM stack does not suffer from the ABA problem that was demonstrated for the simple non-blocking stack implementation in the lecture. Assume T1 is doing a pop() and is interrupted just before it commits. Show the contents of the read and write sets for T1, and feasible values for the relevant locations in memory at the time that the interrupted thread tries to commit as part of your answer.

```
typedef struct node_s {
    int val;
    struct node_s *next;
} node_t;

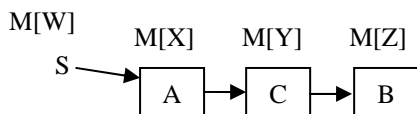
node_t *S; /* the stack */

void push(node_t **stack, node_t *n)
{
    atomic {
        n->next = *stack;
        *stack = n;
    }
}
```

```
node_t * pop(node_t **stack)
{
    node_t *result = NULL;
    atomic {
        result = *stack;
        *stack = *stack->next;
    }
    return result;
}

T1:
...
node_t *my_node = pop(&S);
...
```

Initial stack. The notation M[X] indicates the address of each item. For example, node C in the stack has address Y.



Read set: [(Address M[W], value M[X]) / from read of *stack */
 (Address M[X].next, value M[Y]) /* from read of *stack->next */
]*

Write set: [(Address M[W], value M[Y]) / from write of *stack=... */]*

While interrupted, another another thread successfully pops both A and C off the stack, and then pushes A back onto it.

Memory content:

M[W] (the location where the stack ptr is stored) holds value M[X], since node A is on top of the stack.

M[X].next (the next field of A) stores M[Z] (the address of node B).

On commit, T1 fails on validation of its read set since M[X].next now holds M[Z] and not the value M[Y] that T1 read, even though the first item in the set validates successfully. Thus the ABA problem is avoided, since T1 can compare the values of multiple locations, not just the top of the stack.

7. [9 points; 3 each] Multiprocessor Scheduling

(a) Briefly discuss the tradeoff between load balancing and processor affinity in multiprocessor scheduling.

Good load balancing requires that we move threads to idle, or underloaded processors, but good processor affinity requires that we run threads on the processor where they ran last. It is usually better to run than to not run, so migrating to an idle processor is generally a good idea, but migrating to a less loaded (but not idle) processor may not payoff.

(b) Explain why different scheduling techniques are needed for parallel jobs vs. a standard workload with large numbers of threads.

Because the threads in parallel jobs are not independent. Scheduling can have a severe impact on performance due to dependences such as producer/consumer relationships, execution barriers, or other synchronization.

(c) Describe the EASY scheduling algorithm, and identify one problem with it.

EASY is a variant of FCFS scheduling for a space-shared parallel system which uses backfilling to find jobs that can take advantage of otherwise idle CPUs. In EASY, a reservation is made for the first job that cannot run immediately, and other jobs are allowed to be scheduled ahead of it, provided they do not delay its start time.

Total marks = (78)
End of test