CS 240 Quiz Answers 2002 - 2004

Stanford University
Computer Science Department

January 30, 2005

These are all of the answers to the sample questions. However, they are given without a clear mapping from question to answer. A necessary (but far from sufficient) condition for being prepared for the quiz is to be able to connect the questions and answers yourself.

1. *A weakness of the VMS system is that applications only get as many pages as fit in their resident set limit, even if they could profitably use more. Increasing the free list size will let them effectively use more than that since they can fill up the freelist with their pages which are rescued on use. So, one workload that runs better: many small applications and one large on that requires more than its resident set limit (everyone's resident set limit will be smaller, giving the large application a larger free list to rescue pages from.) One workload that runs worse: a single application that needs more than the new resident set limit (it will pay the cost of more rescues).*

2. *Thread on monitor lock queue.*

3. *The global lockset will always be strictly larger than the local one so you will get false negatives, depending on the scheduling. For example, the following contrived execution sequence would prevent eraser from flagging that T2 does an unprotected access to variable x:*

```
T1                              T2
lock(a);
           --->  switch -->
```

```
                                x++;
         <---  switch <---
x++;
unlock(a);
```

4. *If a monitor sleeps holding the lock waiting on a condition it must release it otherwise no other thread can enter the monitor to make the condition true. This is also true in linux with the addition that there is no other monitor, so it must always release the lock since otherwise the OS cannot run at all. So at each sleep point, the kernel releases the BLK and then when the thread wakes up, reaquires it. The problem this causes is that this breaks the critical section, which can cause a race condition.*

5. *Eraser will notice the access to the shared variable without lock and will report an error.*

   There is no race condition in the code because the threads implicitly synchronize using the pointer value stored in `q`.

6. *Monitored record: you would need a thread+event loop for each record. This would be a very large overhead.*

   Message id: tells you which specific message send you are replying to, similar to the return address in a procedural system which tells you which specific callsite for "foo" you are returning to (strictly speaking the return address + stack pointer).

7. *There were a bunch of different possible answers:*

   (a) *lack of mechanism to follow up reports at aecl*

   (b) *assertion that no other accidents had happened*

   (c) *printout feature disabled, so no hard copy of the data*

   (d) *reporting regulations only applied to manufacturers rather than users*

   (e) *error messages*

(f) *aecl could not reproduce accidents*

(g) *letter to users did not warn that patient injury was involved*

(h) *overdoses not acknowleged.*

(i) *audio/video off*

(j) *slow letters*

(k) *lack of information propogation for complaints by users*

8. *The first part is easy: the larger the limit, the more calls you can can do before checking that you've exceeded the stack. The right way to look at the second part is that as you increase the depth of the call chain between checks, the chance that there is a worst case path that will exceed your current stack increases, forcing you to allocate another stack.*

9. *Very bad: if slow moving was low, and an application started using an enormous amount of memory ESX would keep its active set small, causing the application to have way to little memory and do very bad things. Not as probable, but still bad: if the application went through cycles of being busy and then during idle the fast average will go low, dragging the entire average down, causing ESX to take back memory (paging it out, etc) that it will then have to immediately have to give back.*

10. *Wa(t)/Sa = Wb(t)/Sb implies:*

    (a) *Sb Wa = Sa Wb.*

    (b) *t = Wa(t) + Wb(t).*

    (c) *Sum(Si) = Sa + Sb*

    We need to show that given these that the error is zero. I.e.:

```
0 = Wa(t) - t * Sa / Sum(Si)
  = Wa   - (Wa+Wb)*Sa/(Sa+Sb)
  = (Sa+Sb)Wa - (Sa Wa + Sa Wb)
  = Sa Wa + Sb Wa - (Sa Wa - Sa Wb)
  = Sb Wa - Sa Wb
  = Sa Wb - Sa Wb
  = 0.
```

11. *Worse: if you didn't demote and only one base page out of the superpage was being used, you waste memory. Better: if you didn't demote and all pages were being heavily used, you save useless faults and repromotions.*

12. *To block at an arbitrary wait point you would have to manually create an event ("continuation") by manually saving the state of the compuatation at the wait point (i.e., all local variables, parameters, the current message, etc) and place this saved "continuation" on a message queue for later processing.*

13. *Linked list with lock acquired on every element; lock on each record.*

14. *This question is very open-ended. Setting RSS to 0 will probably have the worst overhead. First, since the system will likely break immediately. Second, even if it does not break, every single memory reference will require a list removal and insertion. This would happen every few instructions (if not every instruction given that instructions reside in memory). Assuming 10ms disk access time, then the paper states that a rescue takes 200usec, which means that if we do it more than 50 times we pay more than for a single disk access. The alternative would be setting RSS to the size of memory, degenerating to FIFO, but from the graphs FIFO will most likely not cause us to take a page fault every 50 memory references so would still be better.*

15. *Set-up Test depends on Lmtchk running between the increment of Class3 and the check of f$mal, but there is nothing in the code to enforce this.*

    5 pts: Answer given in solution. 2 pts: Answer talking about overflow on class 3 variable

16. *VMS uses linear page tables with 32-bit entries. Pages are 512 bytes long, which means they can hold 128 PTE entries. So a bad address space layout would be to allocate every 128th page forcing it to use an entire page to hold a single PTE. An even worse one would be to maximize the overhead of the system page table as well, which needs one entry to map a page of a P0 or P1 page table. We can similarly maximize overhead by having it map "empty" PTEs by allocating pages 128*128th page. This is really bad since the system page table resides in physical memory and so cannot be page.*

17. *The figure is right. Looking at the later description of the implementation, any write will take it to shared-modified. Once it is shared it is running the lockset algorithm without giving warnings, which means that the per-variable shadow area contains the lockset pointer, so it can no longer be keeping track of the thread number of the original writer. We can also reason from what it should do. If anyone is writing into a variable that at least one other thread has been reading from, we have a possibility of a race, so we had better we raising alerts if the locking protocol is violated. (a legalistic reading of the text can claim that it is technically accurate; it is true that a write access from a new thread in the Shared state does take it to the Shared-Modified state; they just didn't bother to mention that a write access from the old thread in the Shared state also takes the variable to the Shared-Modified state. Under that interpretation the sin is that the authors forgot to mention one important case.))*

18. *Eraser cannot detect if your critical sections are big enough (ex1), if the compiler performs optimizations that mask reads/writes to memory (ex2), or initization. E.g.,*

```
// ex 1
lock(l);
t = x;
unlock(l);
lock(l);
t++;
unlock(l);
lock(l);
x = t;
unlock(l);

// ex 2
lock(l);
t = x;
unlock(l);
x++;         // if x is in a register, Eraser misses this
lock(l);
u = x;
```

```
unlock(l);

// ex 3
int *g = 0;
void run() {
  int *p = malloc(4);
  *p = 0;
  g = p;
  process();
}

void process() {
  (*p)++;  // race condition
}
```

it will not catch when you make a decision based on information that could be stale (becaues it was not protected by a lock). Possible fixes: For ex 1 let user annotate that two variables should be protected by the same lockset; For ex2, the registers could be checked along with the memory that is loaded to it; For ex3, an annotation could be inserted to indicate when initialization is finished.

19. *The most straightforward: Navarro puts closed files on the inactive list where they can be moved to the cache list under continuity pressure. A simple workload would be (1) a program that runs repeatedly and accesses the same files running concurrently with (2) an application that needs large superpages (e.g., FFTW) and hence steals these pages for its own ends.*

20. *Unfortunately no: while the balloon application could easily "claim" memory from other applications by calling malloc and then frantically touching it, the OS still owns the memory and because the application cannot pin it, the OS can decide to use it for other purposes and page it to disk. The balloon drivers in contrast allocated and pinned memory, making sure that the OS (1) knew it did not own it and (2) would not page it out.*

21. *alloc spikes, which means the application is using more memory. The*

active spike tracks that of alloc closely, which means it is a good estimation of alloc. And the balloon goes down inversely, which means that (1) this application is getting memory (most likely because it has a lower idle count than other ones) and (2) it had a reasonable amont of memory in its balloon and (3) the memory it is using is coming by unballooning. If the system had 2GB then the active line would be the same, the alloc line would be flat (all applications would have their max) and the balloon line would be flat as a result.

22. *Runs when an allococation request fails or under memory pressure. Running it add (1) CPU overhead of moving pages between the inactive to cache list and (2) will increasingly favor continuity over LRU, thereby hurting cache effectiveness (usually).*

23. *Wakeup semantics. No recursive monitors. "Locks with clause" object not protected from modification. Does not worry much about fairness on locks "in a properly designed system should not be many processes waiting for locks."*

24. *If there is enough space: will reserve a 512K reservation either directly or by pre-empting another 512K reservation.*

    If there is not enough space, the continuity deamon will run, trying to move enough memory to the cache and free list to make room. If it finds 512K free, it will make the reservation and immediately go to sleep. If not it will finish and then make the next smallest reservation: either 64K or 8K.

25. *Since there are so few threads involved we know it has nothing to do with the fact that all threads have stacks. The most plasible reason (which we discussed in class) is that (1) since the other systems use preemptive multitasking they protect the shared producer-consumer queue with a lock and (2) the scheduler does not know about this lock and frequently interrupts and blocks threads that hold it. When (2) occurs no other thread can make progress and throughput drops through the floor. In contrast, Capriccio has manually-inserted yield points in the code (this is what "cooperative multitasking" means) and it looks like they carefully avoided putting these in where the shared queue was being manipulated. In my opinion, this is a dishonest experiment since the*

performance win has nothing to do with capriccio itself, just with the ability to say "do not interrupt me here," which is an idea which has been around for awhile.

26. *If a thread T2 with a higher priority is waiting on c, then when c is signaled by T1, the system will switch from the current thread T1 to T2, which will immediately block when attempting to acquire the monitor lock, be put on the lock queue, and then the system will eventually switch back to T1.*

    The compiler can release the monitor lock before the signal if signal is the last instruction in the monitor function.

27. *To detect deadlock, to detect when you should be using a monitor.*

28. *Consider the following contrived program:*

```
T1 T2
lock(a);
                 ---> switch --->
                                         x;
                                         unlock(a);
```

    Here, T2 never acquires the lock ă thus eraser will not flag the access of x̆ since it is protected by ă.

    5 pts: Answer given in the solution (or equivalent scenario). 2 pts: Some mention of missing locks

29. *Uncaught exceptions in forked procedure causes system to go to debugger.*

30. *It's sort of true and sort of false. What they mean by localized reasoning: the waiter checks exactly the condition they were waiting on at the wait-site rather than having to look at all the signal-sites in the monitor code and make sure the invariant it needs is actually gaurenteed there. In contrast, Hoare semantics would require the waiter to look at all the signal-sites to make sure they only signaled when the exact same condition was true. So in this sense it is simpler and more localized. However, it may be less simple in that if the wait condition is false, the waiter will have to issue another signal or do something (use broadcast)*

to make sure that if it cannot make progress it wakes up another waiter who might be able to — the allocation example in the mesa paper gives an example bug of where the waiter does not do this.

31. *You can wrap up system calls in a user-level veneer that retries as long as the system call is interrupted. Sort of a lo-tech, synchronous scheduler activation. This invalidates the example but not necessarily the argument.*

32. *Something about a mix of applications; heavy paging, light paging, varying memory sizes, varying the parameters used to control the lists.*

33. *Input timing; no locks.*

34. *The real definition of a race condition is a concurrent access that causes an application invariant to be violated. There are many ways to concurrently modify shared memory with/without holding a lock that do/do not violate any invariant.*

```
// too strong: unprotected modification of shared statisitics
// variable
count_packets++;


lock(a);
stack[n] = x; // add element to a stack
unlock(a);

// invariant violated at this point: n != number of elements in
// stack.

lock(a);
n++; // increment element count
unlock(a);
```

Lots of other examples are possible.

35. *You have to open a secret channel with ESX. The most straightforward is probably just open a socket connnection. The application would malloc a big region and pin it. It would then unpin it to shrink. Downsides: (1) the application must run before it can pin/unpin memory, which will be much less under ESX's control than a device driver (2) pages still might not be reused quickly by OS when unpinned if they are considered "accessed" (3) the application will have to signal to ESX which pages it pinned — possibly by writing special values into the pages (a little shady), finding a way to walk the page tables yourself to do the virtual to physical translation, or augmenting ESX to do it by inspecting the guest's page table when it received communications over its secret channel.*

36. *The replacement should have no effect for well-formed mesa programs that use a while loop to recheck their wait condition. Such programs will work (albeit possibly more slowly) even in the presence of completely random wakeups.*

37. *There are a few:*

    (a) *If we run one copy of VMS on ESX: VMS could be running many copies of the same program, in which case page sharing could help.*

    (b) *Multiple VMS copies on top of ESX vs one VMS: VMS uses static resident set limits even if different processes are not making good use of their memory. If this was causing problems, ESX could reapportion memory between the different systems.*

38. *This isn't very smart. The touched pages will be used by the guest OS (unlike balloon pages, which are "fake"), thereby increasing rather than reducing its memory pressure. Additionally, keeping them active will displace other more useful pages.*

39. *As with the previous question, it would allow you to change the size of the resident set limit based on how actively the applications were using*

their pages. The most straightforward way would be to recalculate the resident set limit on page fault time based on the taxation calculation.

40. *Main overhead of capriccio: dynamic checks. Main benefit: smaller stack. So we need to find cases where the stack size of static and Capriccio is the same, and Capriccio has to pay the overhead of dynamic checks. First example: a deep sequential callchain where each function only calls the function below it and then stays and does a lot of work at the leaf. Second example: similarly deep callchain where functions call a large function and a small function, but wind up doing all the work in the large function.*

41. *Hoare-wakeup semantic corresponds to adding the port to the list of ports you are waiting on and immediately processes a message on that port if one is available, before processing any other message.*

42. *Without the "slow" average, the system will react too quickly to changes. E.g., assume the guest OS runs two applications, the first memory intensive, the second CPU bound. Each app gets 10ms to run (the scheduling quanta). Then when the CPU bound one is scheduled, it will use little memory, most of the guest's memory will be judged as idle and the entire guest system will be penalized.*