CS240
Advanced Topics in Operating Systems
Midterm – May 3, 2007

OPEN BOOK, OPEN NOTES

!! You should skip 10 points worth of questions !!

Your name: ________________________________

Stanford ID: ________________________________

In accordance with both the letter and the spirit of the Stanford Honor Code, I did not cheat on this exam. Furthermore, I did not and will not assist anyone else in cheating on this exam.

Signature: ________________________________

The quiz has 15 questions. You have 75 minutes to complete the remainder. Some questions may be much harder than others.

I Speculative Execution

1. [5 points]: What optimizations is xsyncfs doing in Figure 1? What is the difference between eager and output triggered commits in xsyncfs? How is eager commit xsyncfs different from just making ext3 synchronous?

In figure 1, xsyncfs is:

- *Reporting i/o completion to the app before it hits the disk*
- *Accumulating multiple pending i/o's and then issuing them in bulk*
- *Even reporting completion of system calls that notify the user or other apps of i/o completion before the i/o hits the disk*
- *Buffering those system calls though and NOT allowing the user or any other app to see their effect until the i/o does hit the disk*

Eager commits launch the underlying i/o as soon as the application issues it whereas output triggered waits until a dependent (that is, externally visible) operation happens. Output triggering allows batching and even discarding of i/o that is never observed, but can have much higher latency between the application issuing an i/o and the filesystem actually performing and committing it.

Eager commits differ from sync ext3 still report success to the application before issuing underlying i/o and still allow the application to continue executing until / unless it an external observation occurs (and attempts to buffer external observations).

2. [5 points]: What different things does xsyncfs do when `exit` is called? Why? Give two example uses that would cause it to switch between these behaviors.

II Boehm tricks

3. [5 points]: Consider the code:

```
initialization:
    int y = 0;
    int x = 0;

  thread  1   |  thread 2
  -----------------------
  1: r1 = x;  | 3: r2 = y;
  2: y =  1;  | 4: x = 2;
  -------------------------
```

Give all values you could see for `r1` and `r2` after running this code, explaining anything surprising.
Obvious: r1=0,r2=0, but also r1=0,r2=0, r1=2,r2=0,r1=0,r2=1 if the compiler re-orders things or perhaps you observe these r1 and r2 from another processor on a machine without sequential consistency.

III Eraser

4. [10 points]: You have a structure:

```
struct foo {
        // a and b are of size= 1 bit.
        unsigned a:1,
                 b:1;
        lock a_lock;   // protect's field a.
        lock b_lock;   // protect's field b.
};
```

And your program declares a global `f` of type `struct foo`. It then runs this code in thread T1:

```
    lock(f.a_lock);
    f.a = 1;
    unlock(f.a_lock);
```

And this code in thread T2:

```
    lock(f.b_lock);
    f.b = 1;
    unlock(f.b_lock);
```

What error(s) will Eraser flag? Are these error(s) real or false positives? What if you change `a` and `b` to be of type `char`?
It will complain that the memory word holding a and b is not consistently protectd. In this case these are real errors, assuming the machine does not do atomic bit-sized writes. Changing to char on the alpha will not fix (since it does not store bytes atomically) but on many other machines will.

IV Mesa

5. [5 points]: Your code has two monitors, M1:

```
entry foo1() { bar1(); }
entry foo2() { bar2(); }
```

And M2:

```
condition c;
entry bar1() { wait(c); }
entry bar2() { signal(c); } }
```

Your program calls into M1. What can happen? Why does MESA work this way compared to the alternative?
It can deadlock: M1.foo1(), then m1.foo2(); The alternative is for wait to release all containing locks. They chose to not do this b/c it would make modularity hard to preserve: before any call out of the monitor you'd have to figure out if it could block and would have to guarantee the monitor invariant.

6. [5 points]: Eraser claims that: "Monitors are an effective way to avoid data races if all shared variables are static globals, but they don't protect against data races in programs with dynamically allocated shared variables..."

Is this claim correct?

No. You can do monitored records, which associates a monitor instance with each allocated instance of a structure.

V The world according to Burrows

7. [10 points]: Which statement is not true and why?

A. Event-driven code *always* rips apart functions, spreading one logical task over a number of functions.

B. Event-driven code typically cannot take advantage of multiprocessors.

C. Event-driven programs block on a page fault.

D. They don't use Eraser at Google because it had too many false positives.

Event-driven code does not always rip apart functions, only when they could call a function that could block.

VI VMS

8. [5 points]: In terms of virtual memory stuff: what do you have to do on context-switch?
Switch the page table registers for the P1 and P2 page tables, but not the system page table. Also you'd have to probably flush the TLB since it's almost certainly not tagged.

9. [5 points]: What happens if the RSS limit is too high? Too low? What happens if the difference between the high and low watermark on the modified list is too small? Too large?
Too high: no LRU. Too low: no performance isolation, lots of rescues. Difference too small: not enough batching. Too large: really tie up disk.

VII Appel and Li

10. [**5 points**]**:** Which is more expensive, `prot1` or `unprot`? Why? If `protN` is useful, why not also provide `unprotN`?
prot1 is more expensive since you have to reduce protections and thus must kill TLB entries, potentially on other CPUs. Unprot can be done lazily at fault time. unprotN is not that useful since all of their tricks just unprotect a single page.

11. [**5 points**]**:** Assume your OS does not provide `DIRTY`: how to emulate it at user level?
Make the page read only. When a write happens there will be a trap. Can mark page as dirty in user-level structure and then unprotect it.

VIII Superpages

12. [10 points]:

In the paper "Practical, transparent operating system support for superpages," by Navarro et al., the authors propose some virtual memory implementation changes that generally benefit application performance. However, there is always the danger that an OS change that benefits one application may harm another. Which of the following properties of the proposed scheme could potentially hurt performance?

Circle either True or False next to each statement. (2 points each)

A. **True False** The author's implementation uses an expensive SHA-1 hash function to detect which subpages of a superpage have been modified. Computing SHA-1 consumes CPU time that would otherwise be available to user applications.

B. **True False** Paging superpages to disk must cause fragmentation in the swap partition, thereby increasing seek times during page faults.

C. **True False** Superpage support can cause more cache misses in a direct-mapped, physically-addressed cache if many processes use the same virtual address for different physical memory.

D. **True False** Under memory pressure, the superpage scheme no longer evicts pages in (approximate) least-recently-used order, which can cause an increase in paging activity.

E. **True False** The proposed scheme uses extra pages of physical memory for reservations, which can result in applications being paged out even when there are a fair number of unused pages of physical memory.

A. *False: they don't do this, it was only proposed.*

B. *False: it doesn't have too, you could each individual page in the superpage to different disk locations. This won't increase seek times more than non-superpages since it's isomorphic.*

C. *False: which processes use virtual address doesn't matter for physical.*

D. *True: they favor continuity slightly, which can cause problems (though experiments suggest does not).*

E. *False: they will preempt these reservations.*

IX VMMs

For each of the following workloads, explain whether you would expect better performance with a software or hardware VMM similar to the ones described in the Adams and Agesen ASPLOS paper. Justify your answers with concrete data from the paper.

13. [10 points]:

- A userlevel program the iteratively computes the first 100000 Fibonacci numbers.

  Fibonacci is pure userlevel direct execution so performance should be the same (and near native speed) for both VMMs.

- 100 invocations of a C program that reads /etc/passwd a line at a time and printf()'s each one with any uppercase letters converted to lowercase and vice versa.

  This rapidly becomes a buffer cache / system call benchmark since very little disk i/o, address space, or privileged operations are involved. The hardware VMM will likely be faster.

- A network daemon that each time it gets a connection fork()'s a child process that reads in a paragraph of text from the network a line a time, printf()'s each one with any uppercase letters converted to lowercase and vice versa, and then exit()'s.

  Process creation, process destruction, and copy-on-write are all expensive MMU operations to virtualize. Additionally network i/o is real i/o, not cached. Those operations all favour the software VMM.

X ESX

14. [5 points]: One problem with a hypervisor implementing transparent paging is the "double paging." Imagine an OS that implements transparent page sharing (just internally, for its kernel and its applications). Is there a similar "double sharing" problem? Why or why not?

There is not much destructive interference. Double-sharing will limit the intra-VM sharing opportunities for the VMM (which Carl estimated during class as a few percent of total memory). However, even if the VMM first shares two pages and then the OS does, the only "extra" work the VMM has to do is decrement the reference count in the bookkeeping for its shared frame. There is no extra i/o writing the page the disk and retrieving it. And, inter-VM sharing opportunities, the brunt of the win for page sharing, still remain for the VMM.

Note: Another contrast is that while guest paging policy is opaque to the VMM, the guest "sharing policy" (which is just: share all pages that have identical contents) is known to the VMM.

15. [5 points]: Most OSes have some sort of `mpin(void *va, unsigned len)` system call, which will pin a range of addresses in physical memory. Explain how to use this system call to replace the need to write a balloon driver. What is the advantage/disadvantage of this approach?
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.