Final Exam

CSC 252

7 May 2011

Directions; PLEASE READ

This exam has 16 questions, many of which have subparts. Each question indicates its point value. The total is 154 points. Questions 16(c) and 16(d) are for extra credit only, and not included in the 154. They won't factor into your exam score, but may help to raise your end-of-semester letter grade.

This is a *closed-book* exam. You must put away all books and notes. Please confine your answers to the space provided.

In the interest of fairness, the proctor has been instructed not to answer questions during the exam. If you are unsure what a question is asking, make a reasonable assumption and state it as part of your answer.

You will have a maximum of three hours to complete the exam. The proctor will collect any remaining exams promptly at 11:30 am. Good luck!

1. (3 points) Put your name on every page (so if I lose a staple I won't lose your answers).

2. (5 points) Calculate `0xfa35` − `0x90d2` as a 16-bit 2's complement difference.

   Answer: `0xfa35` − `0x90d2` = `0xfa35` + (−`0x90d2`) = `0xfa35` + `0x6f2e` = `0x6963` with a discarded carry of 1.

3. (5 points) When writing a floating-point program in C, how does one decide whether to use single-precision (`float`) or double-precision (`double`) values?

   Answer: Single-precision numbers occupy less space. They also support faster arithmetic on many, though not all, machines. They have only 23 bits of significant and 8 bits of exponent, however. Double-precision numbers have 52 bits of significant and 11 bits of exponent, allowing them to represent a much wider range of numbers with much higher precision. So use single precision if it suffices; otherwise, use double precision.

4. (5 points) What is the purpose of dynamic linking?

   Answer: To avoid wasted space in executable files and physical memory, and to allow already-compiled programs to obtain the benefits of library upgrades.

5. The origins of the 32-bit x86 instruction set architecture (ISA) date back to the early 1970s.

(a) (5 points) Modern instruction sets usually use 3-address instructions, in which the source and destination registers of arithmetic operations are independent. Why does the x86 ISA use 2-address instructions, in which the result overwrites one of the operands?

Answer: Two address instructions take less space to encode, and code space was at a premium in the 1970s.

(b) (5 points) Modern 32-bit instruction sets typically have 32 general-purpose integer registers. Why does the x86 have only 8?

Answer: With only 8 registers one can encode a register name in 3 bits. With 32 registers, one needs 5 bits per register. Again, code space was at a premium in the 1970s.

(c) (5 points) The x86 ISA allows one operand of an arithmetic operation to be in memory. Modern instruction sets typically require the operand to be moved into a register first, with a special load instruction. Why?

Answer: If all operands are in registers, then an arithmetic operation has predictable timing, and can easily be pipelined. Memory access introduces the need for an interlock mechanism to stall the pipeline on a cache miss. Limiting memory access to loads and stores avoids the need for interlocks on all the arithmetic operations, simplifying the pipeline.

(d) (5 points) Why do modern compilers refrain from using many of the instructions in the x86 ISA?

Answer: Because they're difficult to pipeline. Modern implementations of the x86 sustain high throughput only when executing a RISC-like subset of the ISA.

Questions 6 and 7 ask you to consider the caching and VM architecture of a hypothetical future machine. The questions build on one another: you should assume all the specifications of the earlier questions when answering the later ones. Show your work if you want to have a chance at partial credit.

6. Suppose our machine has 128-byte cache blocks, that the L1-D cache is 2-way associative and 64KB in size, and that physical addresses are 56 bits long.

(a) (4 points) Within a physical address, how many bits will be used to indicate the offset of a byte within a block?

Answer: $\log_2(128) = 7$.

(b) (4 points) How many lines will the L1-D cache contain?

Answer: 64KB/128 bytes $= 2^{16}/2^7 = 2^9 = 512$.

(c) (4 points) How many sets will it contain?

Answer: $512/2 = 256$.

(d) (4 points) Within a physical address, how many bits will be used to index into the cache?

Answer: $\log_2(256) = 8$.

(e) (4 points) How many bits will be used for the tag in each line?

Answer: $56 - 8 - 7 = 41$.

(f) (4 points) What will be the maximum amount of physical memory the hardware can address?

Answer: 2^{56} bytes = 64 PB.

7. Suppose our machine has 16-kilobyte pages and that a virtual address is a full 64 bits long. Suppose further that we use a 5-level tree-based page table in which each node is guaranteed to be half-page (8KB) aligned and each entry in the node is 8 bytes in size.

(a) (4 points) How many cache blocks will compose each page?

Answer: 16KB/128B = $2^{14}/2^7 = 2^7 = 128$.

(b) (4 points) How many bits will be used to indicate the offset of a byte within a page?

Answer: $\log_2(16\text{KB}) = 14$.

(c) (4 points) How many bits will compose the page number portion of a virtual address?

Answer: $64 - 14 = 50$.

(d) (4 points) How many bits will be required (on average) to index into the node at each level of the page table tree?

Answer: 50/5 = exactly 10.

(e) (4 points) How many potential page table entries (on average) will each node contain?

Answer: $2^{10} = 1024$.

(f) (4 points) How many bits will be required within each page table entry to indicate the (physical) base address of a node at the next lower level of the tree?

Answer: $56 - 14 = 42$.

(g) (4 points) How big will each tree node be?

Answer: $1024 \times 8 = 8192$ bytes = 1/2 page, which is good, because it matches the tree node alignment.

8. (6 points) Consider a machine with three levels of on-chip cache. Suppose that for a certain program the miss rates in the three levels are 3.0%, 2.0%, and 1.33%. What is the average miss rate across all three? Explain your answer. (Remember that the L2 miss rate is the fraction of L1 misses that also miss in the L2, and the L3 miss rate is the fraction of L1 and L2 misses that also miss in the L3.)

Answer: What we're looking for is the rate which, if it were the same at all three levels, would result in the same number of misses going all the way to memory. For this we want the geometric mean:

$$\left(\frac{3}{100} \times \frac{2}{100} \times \frac{1.33}{100}\right)^{(1/3)} = \frac{\sqrt[3]{8}}{100} = 2\%$$

.

9. (6 points) *Copy on Write* is a kernel-supported mechanism that allows separate, logically writable virtual pages (usually in the address spaces of separate processes) to map to the same physical page frame if their content happens to be the same. Both pages are mapped read-only, regardless of the owner processes' logical access rights. If a process tries to write the page, the page fault handler creates a physical copy, patches the page tables to map the virtual pages to the now-separate (and writable) copies, and restarts the faulting instruction.

   Explain the value of copy-on-write for the implementation of `fork()` in Unix.

   Answer: The typical `fork` is followed immediately by some variant of `exec()`. By using copy-on-write for the pages that the parent and child are supposed to have identical copies of, we avoid actually allocating new frames and copying data in pages that aren't modified prior to the `exec`.

10. (8 points) Consider the following program in C:

```
#include <setjmp.h>
#include <stdio.h>

jmp_buf jb;

int f(int x) {
    if (x == 37) longjmp(jb, 3);
    return x;
}

unsigned int best = 0;
int main() {
    unsigned int i;
    if (!setjmp(jb)) {
        for (i = 0; i < 100; i++) {
            int b = f(i);
            if (b > best) best = b;
        }
    } else {
        printf("Exception caught; i == %d, best == %d\n", i, best);
    }
}
```

   When compiled with optimization turned on, this program prints "`Exception caught; i == 0, best == 36`." How can this be? How can `best` be 36 when `i` is 0?

Answer: The compiler can cache `i` in a register, but is likely to leave `best` (a global variable) in main memory across iterations of the loop. When `longjmp()` restores the register values saved at the time of the `setjmp()`, `i` will be overwritten but `best` will remain intact. In general, the programmer cannot make any assumptions about the values of (non-`volatile`) variables in an exception handler.

11. (5 points) In Windows, the foreground process gets to run for a larger share of each second than any background process does. Explain why this policy might make sense.

    Answer: It's likely to provide better performance—and interactive responsiveness—for the application that the human user is paying attention to at the moment.

12. (5 points) Explain the basic tradeoff between hardware and software handling of TLB misses.

    Answer: Hardware handling is somewhat faster, but software handling gives the OS designer complete freedom regarding the format of page tables.

13. (5 points) The Unix "standard I/O" (stream) library is built on top of the system-level I/O library. What is its most important added functionality?

    Answer: Buffering, to reduce the number of individual system calls.

14. (5 points) Why is the open operation considered fundamental to every I/O library package? That is, why not simply specify the file name in each individual call to read and write?

    Answer: Translating a file name to a location on disk requires a very large amount of work, to traverse the file system naming hierarchy and to check permissions at every level of that hierarchy. The open operation allows that work to be amortized over all the reads and writes.

15. (8 points) The semaphore post and wait operations manipulate a data structure that contains both a counter and a list of waiting threads. Because this data structure can potentially be accessed in parallel by threads on separate cores, it is typically protected (inside the semaphore library) by a busy-wait (spin-based) mutual exclusion lock. In other works, semaphores are built *on top of* spin locks. What extra functionality do semaphores provide? Why not just let programs use the underlying spin locks?

    Answer: Because spin locks waste processor cycles when the lock doesn't become available very quickly. Semaphores give the core to a different thread when the current thread must wait. Also, (general) semaphores can be used for purposes other than mutual exclusion.

16. Consider the following x86 subroutine:

```
        .text
.globl _foo
_foo:
        movl    4(%esp), %eax
```

```
        movl    8(%esp), %ecx
        movl    12(%esp), %edx
        jmp     L2
L3:
        movb    (%eax), %dl     ; %dl is the least sig. byte of %edx
        movb    %dl, (%ecx)
        incl    %eax
        incl    %ecx
L2:
        decl    %edx
        cmpl    $-1, %edx
        jne     L3
        ret
```

(a) (6 points) Explain what this subroutine does.

Answer: It takes 3 arguments—call them `p`, `q`, and `n`. It copies an `n`-byte block of data from location `p` to location `q`.

(b) (5 points) Explain why the subroutine doesn't need to save, update, or use `%ebp`.

Answer: Since it doesn't call anything else, it can use the stack pointer to access arguments, and leave the caller's frame pointer intact.

(c) (5 points) Explain why the subroutine doesn't need to save and restore `%eax`, `%ecx`, and `%edx`.

Answer: Those are caller-saves registers. If the caller needs them across the call to `foo`, it (the caller) will save and restore them.

(d) (Extra Credit: up to 10 points) Explain why subroutine `foo` might be considered to have a bug. (Hint: think about overlapping buffers.) Explain how you might modify it to correct the bug. (You don't have to write code; just explain the modification in English.)

Answer: If the "to" buffer has a larger address than the "from" buffer, but the two buffers overlap, then the end of the "from" buffer will be overwritten before it has been read. The fix in this situation is to start at the end of the "from" buffer and work backward.

(e) (Extra Credit: up to 10 points) Explain why subroutine `foo` might be slower than necessary. Explain how you might modify it to make it faster. (Again, just explain the change in English.)

Answer: The code copies a single byte at a time. With a little care we can copy a full word at a time, but the code gets messy. We may need to copy a few bytes individually before or after the main (full-word) loop in order for our main-loop loads and stores to be aligned. Worse, if the two buffers have different alignment, we may need to shift and mask values in registers within the loop (this may still be faster than loading and storing smaller quantities).