CPSC 313, 04w Term 2— Final Exam — Solutions

Date: April 21, 2005; Instructor: Mike Feeley

1. **(20 marks)** Short answers.

1a. Explain the following parts of a UNIX process' address space, by giving an example of one thing that is stored in each of these parts:

.text: code

.data: initialized global variables

.bss: uninitialized global variables

heap: dynamically allocated variables

stack: procedure local variables and arguments

1b. What does a `call` instruction do that a `jmp` instruction does not?

It saves the return address on the stack.

1c. What is a *jump table* and what C-language control structure is typically implemented by one?

It is a list of code addresses use to transfer control via an indirect jump. Switch statements are often implemented using a jump table.

1d. A pipelined CPU can deal with hazards by introducing pipeline bubbles. List one technique that eliminates bubbles for some *data hazards* and one for some *control hazards*.

Data hazards: data forwarding. Control hazards: branch prediction.

1e. Why are the sizes of cache blocks and sets powers of two?

So that block address and offset can be computed from a data address by using some address bits for the block address and the rest for the block offset. Otherwise, division would be required.

1f. Briefly, how do caches seek to exploit *spatial locality* to improve performance?

By using block sizes larger than a single word.

1g. What is the main difference between *interrupts* and other types of exceptions such as traps and faults?

Interrupts are caused by events external to the executing program when other exceptions are caused by the program itself.

1h. In terms of their implementation, what is the main difference between a procedure call and a switch between two user-mode threads (i.e., implemented by `setjmp` and `longjmp`).

NOT COVERED in Fall 2005.

1i. Virtual memory replacement is based on LRU, but set-associative cache replacement is random. List two differences between these two problems that justify the different solutions.

(1) The ratio of cost of page fault to a non-faulting memory access is much larger than the ratio of cost of cache miss to a cache hit, so it is more important to make a smart replacement decision with VM than with caches. (2) Virtual memory replacement is computed in software and does not impact the cost of each memory access, but cache replacement is implemented in hardware and a complex cache-replacement scheme would likely increase the cost of a cache hit.

1j. Give one benefit and one drawback of inverted page tables.

Benefit: size of page table is function of size of physical memory not the size of the virtual address space; this is particularly important for 64-bit address spaces, which are HUGE. Drawback: an inverted page table is a hash table and thus it is more complex to implement and more costly to access.

2. **(3 marks)** The following assembly language was generated by the x86 C compiler from a simple control structure and a few additional statements.

```
        testl   %edx, %edx
        je      .L7
.L5:    addl    %eax, %eax
        decl    %edx
        jne     .L5
.L7:
```

Give the simplest C program that could have produced this assembly language.

```
while (i) {
    j += j;
    i -= 1;
}
```

3. **(7 marks)** Compile this C procedure into x86 (or y86) assembly language. Give both the `.text` and `.data` sections. You can use "`.skip <number-of-bytes>, 0`" to allocate space for variables. Be sure to include the prologue and epilogue. **Comment your code.**

```
int g[4][3];

int foo (int i, int j)
{
  int a;

  g[i][j] = &a;
  return g[i][j];
}
```

```
.text
    foo:    # prologue
            pushl   %ebp                # save base pointer
            movl    %esp, %ebp          # create new frame
            subl    $4,%esp             # allocate "a"

            # body
            movl    8(%ebp), %ecx       # ecx = i
            movl    12(%ebp), %eax      # eax = j
            leal    (%ecx,%ecx,2), %ecx # ecx = i*3
            addl    %eax, %ecx          # ecx = j+i*3
            leal    -4(%ebp), %eax      # eax = &a
            movl    %eax, g(,%ecx,4)    # g[i][j] = &a

            # epilogue
            movl    %ebp, %esp          # esp = reg save area
            popl    %ebp                # restore caller frame
            ret                         # return to caller
.data
    g:      .skip 4*3*4, 0              # int g[4][3]
```

[NOTE: .data above should actually be .bss, because g is not initalized, but I wasn't that picky.]

4. **(8 marks)** Consider the following C-language procedure.

```
void foo (int a1, int *a2)
{
    *a2=bar (a1);
}
```

These three statements could be implemented by the following assembly language code:

```
    # prologue omitted
    movl    8(%ebp), %ebx
    pushl   %ebx
    call    bar
    movl    12(%ebp), %esi
    movl    %eax, (%esi)
    # epilogue omitted
    ret
```

Now, someone has decided to modify the x86 stack discipline to eliminate the frame pointer, freeing `%ebp` for general-purpose use (i.e., `%ebp` is not the frame pointer anymore).

4a. Explain how this decision complicates the compiler's code generation for procedures.

Normally, local variables and arguments are static offsets from the base pointer. If there is no base pointer, then they must be located using offsets from the stack pointer, however, the stack pointer changes during the execution of a procedure (e.g., when procedure calls another procedure) and so the compiler must track the current stack pointer value for each line of code to generate appropriate offsets. The result, for example, is that a variable/argument will be accessed using different offsets in different parts of the procedure.

4b. Illustrate this complexity by modifying the two instructions above that use the frame pointer; make no changes to this code other than removing the two instructions and adding other instructions in the same place. Carefully comment your code and state any assumptions you make. (You can make this change by modifying the original code above or by writing the code below.)

```
    ...
    movl    8(%esp), %ebx       # assume no locals or saved regs
    ...
    movl    16(%esp), %esi      # esp moved by pushl %ebx
    ...
```

4c. Does this change complicate the restoration of the calling stack frame in the epilogue? If so, carefully explain the problem and a possible solution.

Yes. The stack pointer must be set to point to the beginning of the register save area, before saved registers and the saved frame pointer can be restored by `pop`ing. Previously, the save area was a static offset from the base pointer. If there is no base pointer, then the compiler must again track the value of the stack pointer to compute the appropriate adjustment.

5. (6 marks) This problem is a bit more challenging. Consider the following assembly-language code. Start by commenting every line of code. If you aren't sure what this code does, big partial credit for clearly commenting the individual lines of code. (Remember that `%dl` is the low-order byte of `%edx`.)

```
foo:    # prologue omitted
        movl    8(%ebp), %eax
        xorl    %edx, %edx
        subb    12(%ebp), %dl
        jne     .L1
        movl    $1, %eax
        jmp     .L3
.L1:    leal    .L2(%edx, %edx, 2), %edx
        jmp     *%edx
.L2:    imull   %eax, %eax      # this instruction is 3 bytes long
        imull   %eax, %eax
        ... repeat for a total of 255 imull lines
.L3:    # epilogue omitted
        ret
```

What function does foo(a,b) implement?

foo(i,j) = i^{2^j} as long as j $\leq$ 255 and 1 otherwise

What is foo(2,3)? 256

6. (6 marks) Consider a three-stage pipeline where the gate delay of the stages are 13ns, 18ns and 8ns. The delay of the registers between stages is 2 ns. A ns is 10^{-9} seconds. Answer the following questions; you can skip the addition and multiplication by just giving me a formula that I can plug into a calculator to get the answer. Show your work.

6a. What is the shortest possible clock period (in units of ns)?

20 ns

6b. What is the maximum throughput of the processor (in units of instructions per second)?

$\frac{1\ cycle}{20\ ns} = \frac{10^9\ cycle}{20\ s}$

6c. Suggest a single architectural change that might improve throughput (I can think of two). Say **why it might** and **might why it might not** improve throughput.

Split middle stage into two stages. Reduces clock period to 15 ns and thus might improve throughput to $\frac{10^9\ cycle}{15\ s}$. Might not work if splitting results in additional pipeline stalls/bubbles.

6d. A bit more challenging. If 10% of instructions introduce a single pipeline bubble, what is the throughput of the processor (in units of instructions per second or cycles per instruction; be sure to label your answer with the units you use)?

$$\frac{.9\ instructions}{cycle} \times \frac{10^9\ cycle}{20\ s} = \frac{.9 \times 10^9\ instructions}{20\ s}$$

7. **(5 marks)** For each of the following y86 code snippets, indicate three things. **First**, say whether a data dependency exists and if so show where it is, indicate its type, and explain. **Second**, say whether there is a data or control hazard for the *y86 PIPE* implementation and if so, show where it is and explain. **Third**, say whether the *y86 PIPE* would insert any pipeline bubbles, indicate where, say how many bubbles are inserted, and explain. You can combine your answers if, for example, the same pair of instructions are dependent, cause a hazard and insert a pipeline bubble.

7a.

```
addl    %eax, %ebx
addl    %ebx, %exc
```

(1) Yes; causal dependency with `%ebx`. (2) Yes; hazard exists, because register-file write occurs three cycles after register-file read. (3) No bubbles, because y86 resolves the hazard using data forwarding.

7b.

```
movl    (%eax), %ebx
addl    %ebx, %exc
```

(1) Yes; causal dependency with `%ebx`. (2) Yes; hazard exists, because register-file write occurs three cycles after register-file read. (3) Yes, one bubble, because this is a *load-use* hazard, which y86 resolves by stalling the *use* one cycle and then forwarding the value the *load* reads from memory to the decode stage.

7c.

```
addl    %ebx, %exc
addl    %eax, %ebx
```

(1) Yes; anti dependency with `%ebx`. (2) No hazard. (3) No bubbles.

7d.

```
subl    %eax, %eax
jne     foo
...
```

(1) Yes; control dependency. (2) Yes; hazard exists, because the determination of whether conditional branch is taken occurs in the execute phase and so the two "instruction-issue slots" following the conditional branch must be filled without knowing whether branch is taken. (3) Yes, two bubbles are inserted if branch is not taken, because y86 predicts the branch will be taken.

8. **(7 marks)** Consider a 512-byte, 4-way, set-associative cache with 16-byte blocks. Draw a picture and use pseudo code to carefully explain how the cache handles a long-word (i.e., 4-byte) read request. **First show how the cache determines whether the access is a hit or a miss and then show how it would satisfy the read if it is a hit.** Show how every bit of the data's physical address is used. Your pseudo code should represent the cache using a combination of arrays and structs (e.g., something like $C[i].foo[j].bar$).

$S = \frac{512\ B}{cache} \times \frac{block}{16\ B} \times \frac{set}{4\ block} = \frac{8\ set}{cache}$

```
phys_addr = [   tag    | set index | block offset ]
            [ 25 bits |   3 bits  |    4 bits    ]

let p be physical address
    p.tag be tag part
    p.si  be set-index part
    p.bo  be block-offset part
let C be cache
    C[i] be set i
        C[i][j] be line j of set i
            C[i][j].v       be valid flag
            C[i][j].tag     be the tag
            C[i][j].block   be the block

hit = (C[p.si][0].v && C[p.si][0].tag==p.tag) ||
      (C[p.si][1].v && C[p.si][1].tag==p.tag) ||
      (C[p.si][2].v && C[p.si][2].tag==p.tag) ||
      (C[p.si][3].v && C[p.si][3].tag==p.tag)

let C[i][j] be selected cache line

data = C[i][j].block[p.bo]
```

9. **(6 marks)** You work at Intel again and are allowed to make one single change to an existing cache design. For each change listed below list one potential **advantage** of making that change. In each case, the total size of the cache (i.e., the number of data bytes is stores) remains the same.

9a. Increase the block size?

Improve ability of cache to capture spatial locality.

9b. Decrease the block size?

Improve ability of cache to capture temporal locality.

9c. Increase the set associativity (e.g., from 4-way to 8-way)?

Reduce the number of conflict misses.

9d. Decrease the set associativity?

Reduce the cache hit time (i.e., # of cycles for cache hit).

9e. Change from write through to write back?

Reduce cost of a write hits (i.e., # of cycles a memory-write instruction stalls when the target memory location is currently stored in the cache).

10. **(6 marks)** Shared Libraries

10a. Give one benefit of dynamically-linked shared libraries, compared to static linking.

> The read-only part of the library (e.g., code and read-only globals) can be shared by all processes that use the library, thus dramatically reducing the total amount of physical memory required to store a large number of programs that use common libraries.

10b. Why can't shared libraries be statically linked? Explain briefly.

> Because they store virtual addresses for accessing global variables and for transferring control (e.g., procedure calls) and the actual virtual address assigned to a shared library is different in every program that links it.

10c. Why must dynamically-linked shared libraries use *position independent code*?

> So that they can be linked at any virtual address. To do otherwise, would require that all programs in the universe that share a library coordinate with each other to determine what virtual address to assign to the library.

10d. Outline the key idea that allows an instruction in shared library X to read a global variable allocated in shared library X in a position-independent fashion. (No need for assembly code; just describe the idea.)

> While the compiler (and linker) do not know the virtual address of either the instruction that reads the variable nor the variable itself, the both know the number of bytes between them. The compiler thus generates code that adds this offset value to the program counter of the accessing instruction, which at runtime computes the virtual address of the target variable. It then uses this virtual address, in the normal way, to complete the read.

11. **(6 marks)** Answer these questions about a system call trap from an application (i.e., user-mode) process into the operating system. Consider only the interval that starts with the trap instruction (i.e., `int $80`) and ends with the execution of the first instruction of the particular system call specified by the application (i.e. in `%eax`).

11a. Is this trap a *protected call*, a *context switch*, *both*, or *either*? Explain briefly.

> It is a protected call, because it involves into a different protection domain through a controlled entry point (i.e., the callee limits the addresses to which callers can jmp/call), but it is not a context switch, because there is no change of address space (the operating system is mapped into the address space of every application).

11b. What hardware registers are involved? Carefully explain how the hardware uses them.

> Registers involved: protection mode, exception mask, exception table base, stack pointer, and program counter. Exception mask is checked to determine whether exception is currently allowed; exception delivered only when it is allowed. Protection mode changes from 3 (user) to 0 (system/kernel) when delivering the exception. Exception table base register is used to locate the exception table, a list of code addresses in the operating system, indexed by exception type, to which the exception transfers control. Stack pointer is changed so that the current program counter can be saved on the stack. Program counter is changed to the selected code address in the exception table.

11c. What additional steps are handled by the operating system? Give a very brief outline (no need

for details).

> The operating system exception handler saves all of the processes registers on its stack and then switches to a protected system stack. In the case of a system call, the OS's handler then reads from `%eax` the system call number and uses this to index into a table of code addresses for each system call. It transfers control to one of these. Upon return, the process is reversed, the kernel switches back to the user stack, restores the program's registers and then issues the *return-from-interrupt* instruction to switch the protection mode back to 3 (user) and return control to the program at the saved PC address, the instruction immediately following the `int $80` instruction that initiated the exception.

12. (6 marks) For each of the following virtual-memory mangement operations indicate whether, for the architectures discussed in class, it is handled by *hardware*, *software*, *both* or *it depends on the architecture*. **Give a very brief explanation for each answer.**

12a. Translation of a virtual address that maps to a physical-memory-resident page who's page-table entry (i.e., PTE) is stored in the translation look-aside buffer (i.e., TLB)?

> Hardware. Every memory address in instructions processed by the CPU is a virtual address. The CPU hardware checks the on-chip TLB (translation cache) for every address. If a mapping for the virtual address is found in the TLB, the hardware checks the protection flags, updates the accessed/dirty flags and then uses the physical-frame number to construct a physical address that it then hands off to the memory hierarchy (i.e., caches and DRAM).

12b. Handling of a TLB miss for a virtual address that maps to a physical-memory resident page who's PTE is stored in the current process's page table?

> Depends on architecture. x86 has a hardware-loaded TLB. On an TLB miss, the CPU hardware reads the current process's page table for the target address. If a valid mapping exists (i.e., the target page is resident in physical memory), the hardware loads its's PTE into the TLB and restarts the TLB lookup. Some other architectures use software-loaded TLB's in which the CPU traps to the operating system software on a TLB miss and leaves it to the operating system to load a valid translation for the address into the TLB or to abort the program.

12c. Handling of a TLB miss for a virtual address that maps to a page that is not resident in physical memory?

> Handled in software. Only the operating system is able to fetch non-resident pages from disk. This process is called demand paging.

12d. Handling of a TLB miss for a virtual address that maps to a page who's PTE is not stored in the current process's page table?

> Handled in software.

13. (8 marks) You are responsible the page-table implementation for a new, hypothetical, Pentium 4 processor that uses 42-bit virtual addresses instead of the current 32-bit addresses. Answer the following questions.

13a. Do you work at Intel or Microsoft? Explain briefly.

> Intel. The x86 page table format is determined by the hardware implementation, because the CPU hardware must be able to read the page table to handle a TLB miss.

13b. If you stick with the current two-level page-table design, where each chunk of the level-two page table is exactly one 4096-byte page in size (and page table entries are 4 bytes), how many bytes are required to store the level-one page table? Explain briefly.

$4096 = 2^{12}$. So, the VPN is 30 bits; the remaining 12 bits are the page offset. Each page-table-entry (PTE) is 4 bytes, so a page can store $1024 = 2^{10}$ PTEs. Thus, the index to the level-one page table is 20 bits long with the remaining 10 VPN bits indexing a level-two entry. So, the level-one page table stores 2^{20} page directory entries (PDE). Each PDE is 4 bytes long, so the level-one page table is $2^{20} \times 4\ B = 2^{22} B = \frac{2^{32}\ B}{\frac{2^{20}\ B}{MB}} = 4\ MB$. *When answering questions like this it is fine to just give the equation (i.e., the part up to the first equal sign). There is no need to do the complete calculation.*

13c. You decide to change to a three-level page-table design. What improvement are you expecting and why?

The total amount of space occupied by the page table will be substantially less. The reason is that large parts of the old level-one page table map nothing and in the revised table, this structure is replaced by a new level-two table where any page of the original table that was completely filled with empty PDE's is left out.

13d. Your three-level page table is comprised of chucks each of which is a single page in size; that is, no two page-table pages need be contiguous (i.e., next to each other) in physical memory. **Draw a picture of this design showing how an virtual address is translated into a physical address** (ignore the TLB and data caches; just show the page-table lookup). **Indicate how each bit of the virtual address is used. Give a pseudo-code expression for this computation.** The pseudo-code should use $M[i]$ to indicate reading the value of memory at physical address i.

I can't draw the picture. Let va be the virtual address, ptbr a register that stores the physical address of the base of level-one of the current page table (i.e., its the page-table base register) and PDE() and PTE() be type casts that take an integer value and casts it to a page-director or page-table entry. Omitted are protection checks that would normally occur at each stage of the page-table lookup when checking for PDE/PTE residency (i.e., *.p==0).

```
va = [ 10 bits | 10 bits | 10 bits | 12 bits ]
     [   i1    |   i2    |    i3   |    po   ]

l1-pde = PDE( M[ ptbr + va.i1*4 ] );
if (l1-pde.p==0)
   page-fault-to-os
else {
   l2-pde = PDE( M[ l1-pde.pfn * 4096 + va.l2*4 ] )
   if (l2.pde.p==0)
      page-fault-to-os
   else {
      l3.pte = PTE( M[ l2-pde.pfn * 4096 + va.l3*4 ] )
      if (l3.pte.p==0)
          page-fault-to-os
      else {
          phys_addr = l3.pte.pfn * 4096 + va.po
      }
   }
}
```

14. **(6 marks)** The Fifo-with-Second-Chance page replacement algorithm organizes pages on several fifos, one of which is called the *inactive list*. For the following two architectural variations briefly explain how the inactive list gives pages a second chance to be accessed before they are replaced.

14a. Architectures that have an *accessed* bit in the PTE.

When a page is moved from the bottom of the active fifo to the top of the active fifo, the "accessed" flag in its PTE is cleared. This flag is set by the hardware on every access. When pages reach the bottom of the inactive list, the system, checks their "accessed" flag is frees them if and only if their "accessed" flag is still clear. Thus the inactive list gives a page a second-chance to be accessed before it is freed. Any page accessed while it is on the inactive list returns to the active list.

14b. Architectures that do not have an *accessed* bit.

NOT COVERED IN AUTUMN 2005.