Final Exam

CSC 252
12 May 2021
Computer Science Department
University of Rochester

Instructor: Yuhao Zhu
TAs: Rongcui Dong, Elana Elman, Kalen Frieberg, Sudhanshu Gupta, Yiyao (Jack) Yu, Vladimir Maksimovski, Nathan Reed, Rafaello Sanna

Name: _____

Problem 0 (3 points):	_____
Problem 1 (14 points):	_____
Problem 2 (15 points):	_____
Problem 3 (16 points):	_____
Problem 4 (22 points):	_____
Problem 5 (30 points)	_____
Total (100 points):	_____
Extra Credit (15 points)	_____

Remember "**I don't know**" is given 15% partial credit, but you must erase everything else. This does not apply to extra credit questions.

Your answers to all questions must be contained in the given boxes. Use spare space to show all supporting work to earn partial credit.

You have 3 hours to work.

Please sign the following. I have not given nor received any unauthorized help on this exam.

Signature:_____

GOOD LUCK!!!

Problem 0: Warm-up (3 Points)
What's the most surprising thing about computers you learned from 252?

Problem 1: Miscellaneous (14 points + 12 points extra credit)
Part a) (3 points) Convert the decimal number 334 to hexadecimal.

0x14E

Part b) (3 points) Why wouldn't a user program that contains an infinite loop freeze the entire computer even if the computer has only one processor?

OS preempts processes periodically to context-switch and execute other processes.

To get full points, the answer must show:
1. Operating System
2. Context switch
3. Preemption

Part c) (8 points) Consider a shell in Linux, `sh`, that supports basic foreground/background job control, just like the one you implemented in Assignment 4. Initially, `myprogA` is running in the foreground, `myprogB` is running in the background as job 1, and `myprogC` is stopped in the background as job 2. **Assume this initial state for each question below.**

(4 points) The user presses `Ctrl-C` and kills the foreground process. List all signals that are generated and the receivers of these signals.

SIGINT to myprogA; SIGCHLD to sh.

Alternative answer: SIGINT to sh, which forwards to myprogA; SIGCHLD to sh

(4 points) `myprogA` finishes, then the user presses `Ctrl-Z` at the shell's command prompt. List all the signals that are generated and the receivers of these signals.

SIGCHLD to sh; SIGTSTP to sh

Part d) (this entire part is extra credit; 12 points) You are designing a new ISA called URISA, which supports the following instructions.

Instruction	Semantics
`ADDI rd, rs1, imm`	Sum immediate `imm` and register `rs1`, store the sum in register `rd`
`ADD rd, rs1, rs2`	Sum two registers `rs1` and `rs2`, store the result in register `rd`
`LW rd, delta(rs1)`	Load 4 bytes to register `rd` from memory location specified by `rs1 + delta`
`SW rd, delta(rt)`	Store 4 bytes from register `rd` to memory location specified by `rt + delta`
`BLTU rs, rt, label`	Jump to `label` if register `rs < rt`; both `rs` and `rt` are interpreted as unsigned numbers.

In addition, these registers are supported in URISA:

Register(s)	Description	X86 Equivalent
`zero`	Always reads zero; write ignored	-
`ra`	Return address of a function	On stack
`a1`	Return value of a function	`%eax`
`sp`	Stack pointer	`%esp`
`t0-6`	Temporary registers	Any free register, e.g., `%ecx`, `%edx`

Translate each of the following x86 instructions to the equivalent URISA instructions using only the instructions listed above. You may use any register above, but no new memory location other

than what is used in the original x86 instruction. Note that the x86 instructions use the AT&T/GAS syntax, i.e., **`opcode src, dst`**.

For example, the x86 instruction **`pop (%eax)`** would be equivalent to:

```
LW t0, 0(sp)
SW t0, 0(a1)
ADDI sp, sp, 4
```

(3 points extra credit) **`add $0x8, %esp`**

ADDI sp, sp, 0x8

(3 points extra credit) **`add %eax, 8(%esp)`**

LW to, 8(sp)
ADD to, a1, to
SW to, 8(sp)
// Or equivalent

(3 points extra credit) **`push $0x252`**

ADDI to, zero, 0x252
ADDI sp, sp, -4
SW to, 0(sp)
// Or equivalent

(3 points extra credit) This processor does not have an Overflow flag. Fill in the following assembly so that if the ADD instruction causes an unsigned overflow, the program jumps to the `overflow` label.

```
ADD t3, t1, t2
_____BLTU t3, t1, overflow ; or BLTU t3, t2, overflow
;… code omitted
overflow:
;… code omitted
```

Problem 2: Floating-Point Arithmetics (15 points)

Part a) (3 points) Put $6\frac{5}{16}$ into the binary normalized form.

1.100101×2^2

Part b) (12 points) The IEEE has decided to introduce a floating-point standard of some unknown size, whose main characteristics are consistent with existing floating-point number representations that we discussed in the class.

The following bit sequence contains the exact encoding of $6\frac{5}{16}$ in this new standard, but is padded with arbitrary bits at the beginning and in the end: `10010100110010110`.

(3 points) How many bits are used for the fraction in this new standard?

6

(3 points) How many bits are used for the exponent in this new standard?

4

(3 points) What is the bias in this new standard?

7

(3 points) What is the smallest positive value that can be precisely expressed using this new floating-point format? You can write your answer either in the binary-normalized form or as a bit sequence.

0|0000|000001 or 2^(-12)

Problem 3: Assembly Programming (16 points)

Conventions:

1. For this section, the assembly shown uses the AT&T/GAS syntax **`opcode src, dst`** for instructions with two arguments where **`src`** is the source argument and **`dst`** is the destination argument. For example, this means that **`mov a, b`** moves the value **`a`** into **`b`** and **`cmp a, b`** then **`jge c`** would compare **`b`** to **`a`** then jump to **`c`** if **`b ≥ a.`**
2. All C code is compiled on a 64-bit machine, where arrays grow toward higher addresses.
3. For functions that take two arguments, the first argument is stored in **`%edi`** and the second is stored in **`%esi`** at the time the function is called. The return value of this function is stored in **`%eax`** at the time the function returns.

Part a) (9 points) The following is assembly code for a C function **`find_next()`**:

```
0000000000001169 <find_next>:
     1169:    mov    %edi,%eax
     116b:    inc    %eax
     116d:    mov    $0x2,%edx
     1172:    cmp    %eax,%edx
     1174:    jge    1184 <find_next+0x1b>
     1176:    mov    %eax,%ecx
     1178:    sub    %edx,%ecx
     117a:    test   %ecx,%ecx
     117c:    jg     1178 <find_next+0xf>
     117e:    je     116b <find_next+0x2>
     1180:    inc    %edx
     1182:    jmp    1172 <find_next+0x9>
     1184:    ret
```

(3 points) What is the return value of **`find_next(2)`**?

3

(3 points) What is returned when a non-positive **`x`** is provided to **`find_next(x)`**? Show your answer in terms of **`x`**.

x+1

(3 points) For positive values of **`x`** as input, what does **`find_next(x)`** do?

Finds the smallest prime larger than x

Part b) (7 points) Some students wrote a CSC252 project in assembly. They discover right before submission that the machine used for grading has a faulty **`call`** instruction. They decide to rewrite the project without using **`call`**. The behavior of a **`call`** can be reproduced with **exactly two** other instructions.

(3 points) On a **correct 64-bit machine**, which register(s) does **`call 1169`** update? Explain how the register value(s) change.

%rsp = %rsp - 8
%rip = 1169

(4 points) Which instructions would they use to replace the faulty **`call`**? Provide **just the names** of the instructions in the order they are used.

push, jmp
(You can not mov a value into %rip)

Problem 4: Cache (22 points)

You are designing a cache for an 8-bit machine that has a byte-addressable memory. Remember that an 8-bit machine means that the length of a memory address is 8 bits. The cache line size is 2 bytes. As a first attempt, you design a 4-way set associative cache with 2 sets and a random replacement policy. A random replacement policy chooses a line to replace at random within the set. To reduce memory traffic, you design it to be a write-back cache.

Assuming that the machine runs only one program, which will generate the following memory access sequence.

Access	Address
1	00110000
2	10100010
3	10011001
4	01011101
5	00110111
6	10011010
7	01011110
8	01101100
9	10100010

(6 points) How many bits do you need for the tag, the set index, and the cache line offset?

6 bits tag, 1 bit set, 1 bit offset
(Of the 8 bits in an address, 1 bit decides which byte of the cache line it is, 1 bit decides which set it's in, and the remaining 6 bits must be the tag.)

(3 points) How many overhead bits does this cache have? Recall that overhead includes the tag bits, the dirty bits (if needed), and the replacement bits (if needed). Show you work to earn full credit.

64 bits.

(No bits for random replacement, so overhead = (# lines)*(1 valid bit + 1 dirty bit + 6 tag bits) = 8*8 = 64 bits. Accepted 56 bits with a point taken off if the valid bit is not included.)

(3 points) Assuming the cache is initially empty. How many cache misses will the program generate when it's run for the first time?

8 (all but the last access miss)

(3 points) How many cache misses will the program generate when it's run for the second time?

0 (All addresses were loaded into the cache on the first run, and none were replaced.)

(4 points) Looking carefully at the access pattern, you realize that it is possible to design a cache with a smaller overhead that achieves a 100% hit rate for the program after the initial run.

Without changing the cache line size, explain the new design of your cache, i.e., its associativity and the number of sets.

4 sets, 2-associative

(An 8 set direct-mapped cache causes some misses, and a fully associative cache does not reduce overhead because set index bits are not part of the overhead.)

(3 points) What is the overhead of your new cache design?

56 bits

(A 2-way associative 4 set cache needs 1 less tag bit per line, so 8 bits total less than the second answer. 1 point taken off if valid bit is not included.)

Problem 5: Virtual Memory (30 points + 3 points extra credit)

You are building a machine with virtual memory support. The virtual address space is 256 KB (2^18 bytes). The physical memory size is 32KB. The system uses a one-level page table.

Part a) Basic organization (13 points + 3 points extra credit)
(4 points) Assume the page size is 1KB. How many bits are used for the physical page number (PPN), and how many for the virtual page number (VPN)?

PPN: 5
VPN: 8

(3 points) What is the size of one PTE, assuming that the only overhead in the PTE is one valid bit and that the disk address is 2 bits longer than the PPN.

Disk address + 1 = 8

(3 points) You decide to experiment with changing the page size to 4KB. What is the size of one PTE now? Continue to assume that the only overhead in the PTE is one valid bit and that the disk address is 2 bits longer than the PPN.

Disk address + 1 = 6

(3 points) The system has a TLB. What is likely to happen to the TLB hit rate when we increase the page size to 4KB? Will it increase or decrease? Explain your answer to receive ANY credit.

It will increase.
1. Since PTEs are smaller, more of them can be stored in a cache of the same size.
2. Since there are fewer pages, the TLB will store a higher proportion of PTEs.
3. Spatial locality properties make it likely that TLB hit rate will increase.

(3 points extra credit) If the machine were to use a two-level page table with a 4KB page size, how many PTE entries are there in the level 1 page table?

Just 1 PTE.

Part b) Protection (8 points)

The major customer for your system is the US Government. They want a secure system that is safer from buffer overflow attacks. The memory protection system should help protect against arbitrary code execution in the event that someone manages to exploit a buffer overflow.

(4 points) Briefly explain the mechanism by which an attacker exploits a buffer overflow. Your answer must cover 1) where in memory the attacker can place the code and 2) how the injected code can be executed.

1. Attacker uses buffer that is too short/input is not bounds checked to store encoded machine instructions.
2. Attacker input overwrites return address. When vulnerable function returns, it will return to the attacker-controlled return address, which will point to the start of the attacker's code.

(4 points) Now you want to augment the PTE structure with permission information to defend against buffer overflow attacks. In order to keep the overhead low, you decide that you will use a maximum of 2 permission bits, which will indicate whether a page can be written to, whether a page can be read from, and whether a page's contents can be executed as code. Assuming defending buffer overflow attacks is your sole goal in this design process, how will you use the 2 permission bits? Explain the semantics of each bit to receive full credit.

Use 1 bit for read permission (1 = can read, 0 = no read)
Use 1 bit for write/execute (1 = write, no execute, 0 = execute, no write)

The key insight is that we don't want user-writable memory to also be executable, so we can use one bit to handle both write and execute permission, since if it's writable, it can't be executable, and if it's executable, it can't be writable.

Part c) Performance (9 points)

Management decided to go with the 1KB page size for this machine and removed the TLB to cut costs. Consider the C code below. Assume:

1. A 1-level page table translation scheme with the page table starting empty, and
2. `arr` is aligned such that it starts 8 bytes before a page boundary.

```
void func() {
      int arr[2048];
```

```
    for(int i = 0; i < 8; i++) {

        if(rand_0_1() == 1) {
            printf("%d\n", arr[(i << 8) + 1]);
            printf("%d\n", arr[(i << 8) + 2]);
        } else {
            printf("%d\n", arr[i]);
            printf("%d\n", arr[i + 2]);
        }
    }
}
```

`rand_0_1()` is a function that returns either 0 or 1, at random.

For the following 3 questions, only count page faults generated by accessing elements of `arr`.

(3 points) What is the maximum number of page faults that this code could possibly generate? Explain.

9. 2 on the first iteration. Then if the function returns 1 every time, it will cause 1 additional page fault, because the first access for each iteration will be on the page that was loaded for the second access of the last iteration. The second access on each iteration will cause a fault.

(3 points) What is the minimum number of page faults that this code could possibly generate? Explain.

2, if the function returns 0 every time. The 2 are from the first iteration, everything after that is on the same page.

(3 points) Suppose that the `rand_0_1()` function is faulty and it returns 0 40% of the time and 1 60% of the time. How many page faults, on average, will this code generate? Explain.

6.2. Two initially plus an expected value of 4.2 for the other 7 iterations (0.6 chance for 1 page fault per iteration, 0.4 chance of 0, 0.6*7 = 4.2)