UCLA Computer Science 33 (Spring 2015)
Final exam
180 minutes total, open book, open notes

Name:_____ Student ID:_____

1	2	3	4	5	6	7	total

1 (10 minutes). Why do CPU caches interact so poorly with GNU/Linux exception handlers? Give a brief example.

2. The book says that the 'cltq' instruction is just a shorthand for 'movslq %eax,%rax'. And yet if you put the following assembly language program:

```
                .globl  funa
        funa:
                pushq   %rbx
                movq    %rsi, %rbx
                call    g
                cltq
                addq    %rbx, %rax
                popq    %rbx
                ret

                .globl  funb
        funb:
                pushq   %rbx
                movq    %rsi, %rbx
                call    g
                movslq  %eax, %rax
                addq    %rbx, %rax
                popq    %rbx
                ret
```

[continued in next column]

into the file fun.s and run these shell commands:

```
        gcc -O2 -c fun.s
        objdump -d fun.o
```

the resulting output will contain something like this:

```
  0000000000000000 <funa>:
     0: 53                   push   %rbx
     1: 48 89 f3             mov    %rsi,%rbx
     4: e8 00 00 00 00       callq  9 <funa+0x9>
     9: 48 98                cltq
     b: 48 01 d8             add    %rbx,%rax
     e: 5b                   pop    %rbx
     f: c3                   retq

  0000000000000010 <funb>:
    10: 53                   push   %rbx
    11: 48 89 f3             mov    %rsi,%rbx
    14: e8 00 00 00 00       callq  19 <funb+0x9>
    19: 48 63 c0             movslq %eax,%rax
    1c: 48 01 d8             add    %rbx,%rax
    1f: 5b                   pop    %rbx
    20: c3                   retq
```

so the two instructions do differ.

2a (10 minutes). Explain the seeming disagreement between the book and the objdump output.

2b (10 minutes). Write C source code that corresponds to the above assembly-language program. Your program should define functions and/or data that have the same behavior as those of the given program. Do not use 'asm' directives.

2c (10 minutes). What symbol table and relocation entries should appear in fun.o? Briefly explain.

3 (15 minutes). Where are PTEs most commonly used by the Nehalem microarchitecture implementation of the Core i7? Where else can PTEs appear, other than their most commonly-used locations? For each place in the hardware where a PTE can appear, explain how and why it would appear there.

4. Suppose we change the Core i7's virtual addresses to use "huge" pages, i.e., pages of size 1 GiB. We alter the rest of the implementation as little as possible: for example, we don't change the page table size.

4a (10 minutes). What should virtual addresses look like with under this regime? In other words, what components would virtual addresses be broken up into, compared to the components normally used in the Core i7?

4b (10 minutes). Give two performance advantages that come from having huge pages, as opposed to the usual 4 KiB pages. One should be a time advantage, the other a space advantage. Briefly explain.

4c (10 minutes). Give two performance disadvantages, again one in time and one in space. Briefly explain.

4d (10 minutes). Given the above, give an example practical application that would benefit from huge pages. Give an example that would suffer from huge pages. Briefly explain.

5 (10 minutes). If you compile the following GNU/Linux x86 assembly-language program:

```
     1          .globl  main
     2  main:
     3          movl    $amusing, p
     4          call    *p
     5  amusing:
     6          .byte   15
     7          .byte   -57
     8          .byte   -16
     9          .bss
    10  p:
    11          .zero   4
```

with 'gcc -m32' and run it on the SEASnet GNU/Linux servers, the program will behave this way:

```
  $ ./a.out
  Illegal instruction (core dumped)
```

This program has received signal 4 (SIGILL, Illegal instruction) and has dumped core.

Suppose we remove line 3 from the program. How will it behave instead? Give the sequence of instructions that it will execute, and the behavior the invoker will observe.

6. Consider the following GNU/Linux x86 assembly-language application:

```
 1  h:
 2          movl    4(%esp), %eax
 3          movl    $1, (%eax)
 4          ret
 5
 6          .globl  main
 7  main:
 8          leal    4(%esp), %ecx
 9          andl    $-16, %esp
10          pushl   -4(%ecx)
11          pushl   %ebp
12          movl    %esp, %ebp
13          pushl   %ecx
14          subl    $28, %esp
15          pushl   $h
16          pushl   $4
17          call    signal
18          addl    $16, %esp
19          movl    $1, %edx
20  .L3:
21          rdrand  %eax
22          movl    %eax, -12(%ebp)
23          cmovb   %edx, %eax
24          testl   %eax, %eax
25          je      .L3
26          movl    -4(%ebp), %ecx
27          movl    -12(%ebp), %eax
28          leave
29          leal    -4(%ecx), %esp
30          ret
```

[continued in next column]

Unlike the other instructions in this listing, the rdrand instruction in this listing is not in the book. rdrand is available on Ivy Bridge and many later Intel CPUs. On these processors, 'rdrand %eax' either sets %eax to a random bit pattern and sets the carry flag, or it clears both %eax and the carry flag. On processors where rdrand is not available, it is an illegal instruction.

6a (5 minutes). Suppose we are running on an Ivy Bridge CPU. This application has a loop with suboptimal code. Optimize the loop, and briefly explain why your improvement works and should have better performance.

6b (10 minutes). Again, suppose we are running on an Ivy Bridge CPU. List the instructions that the original program executes by line number, briefly explain any tricky parts, and describe the externally-visible behavior of this program. Assume that rdrand always sets the carry flag in your run.

6c (10 minutes). Now, suppose we are running on the SEASnet GNU/Linux servers. They use Xeon E5630 Westmere-EP processors, which do not support rdrand. Repeat part (b) under this new assumption. You need not list the part of the execution history that merely duplicates part (b)'s; just indicate which part of the execution history is the same and list the part that differs.

7. Valgrind is a program for debugging and profiling executables running under GNU/Linux. The shell command 'valgrind foo bar' acts like 'foo bar' except that Valgrind will report some (but not all) memory errors in 'foo', e.g., subscript errors, memory leaks, etc. Valgrind operates not by executing your program directly, but by copying a few instructions from your program to a buffer it allocates (instructions that it has analyzed and has checked to be safe to execute), and then by using a jmp instruction to jump into the buffer. Valgrind arranges for the last instruction of the buffer to return to Valgrind. By caching commonly-used safe instruction sequences of this sort, Valgrind can run your program nearly as fast as it would run natively, while still carefully checking problematic instructions (e.g., array-subscript instructions) for validity at runtime.

Valgrind's documentation says:

> Valgrind is compiled into a Linux shared object, 'valgrind.so'.... The 'valgrind' shell script adds 'valgrind.so' to the 'LD_PRELOAD' list of extra libraries to be loaded with any dynamically linked library. This is a standard trick, one which I assume the 'LD_PRELOAD' mechanism was developed to support.
>
> 'valgrind.so' is linked with the '-z initfirst' flag, which requests that its initialisation code is run before that of any other object in the executable image. When this happens, Valgrind gains control. The real CPU becomes "trapped" in 'valgrind.so' and the translations it generates. The synthetic CPU provided by Valgrind does, however, return from this initialisation function. So the normal startup actions, orchestrated by the dynamic linker 'ld.so', continue as usual, except on theg synthetic CPU, not the real one. Eventually main is run and returns, and then the finalisation code of the shared objects is run, presumably in inverse order to which they were initialised. Remember, this is still all happening on the simulated CPU. Eventually valgrind.so's own finalisation code is called. It spots this event, shuts down the simulated CPU, prints any error summaries and/or does leak detection, and returns from the initialisation code on the real CPU. At this point, in effect the real and synthetic CPUs have merged back into one, Valgrind has lost control of the program, and the program finally 'exit()'s back to the kernel in the usual way.

7a (10 minutes). Does valgrind run under control of the dynamic linker, or vice versa? Briefly explain.

7b (10 minutes). Suppose valgrind.so were not linked with the '-z initfirst' flag. What would go wrong?

7c (20 minutes). Explain what is meant in the above passage about "real" and "synthetic" CPUs. What's the relationship between these two CPUs and virtual memory? Between these two CPUs and the Linux kernel? Between these two CPUs and the physical CPU or CPUs that are running your program?

7d (10 minutes). The documentation quoted above is obsolete. As of Valgrind 3.0, Valgrind is an executable program in its own right, instead of being a shared object that is linked into your program. Describe generally how Valgrind 3.0 and later can still do its job and check a program while executing it reasonably efficiently, without ceding control to the program, even though Valgrind 3.0 is no longer a shared library.