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.

Exceptions are... exceptional (rare), so data and instructions from exception handlers are likely not cached. In fact, since exceptions are run in kernel mode, the data accessed will be things from the kernel's stack/heap/data and will likely require code that could not ever be accessed in the user space (except in the case where the same exception has previously occurred).

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
```

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.

ctlq is not a true shorthand for movslq %eax, %rax, since that implies that ctlq instructions will always translate to movslq %eax, %rax. As seen in the code, both instructions are unique, and ctlq is simply a shorter instruction with the same effect.

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.

```
long f(arg1, long arg2, ...) {
      return g(...) + arg2;
}
```

...where arg1 can be any integer style data type (ie. not a double), and f() and g() can take any number of arguments. Due to the fact that after calling g(), the value in %eax is sign extended into %rax, it is inferred that g() returns an int. As a result, the function body may also be some variation of the following form:

```
int ret = g(...);
long lret = (long) ret;
return lret + arg2;
```

Due to C’s typecasting rules however (the operands in an expression are converted to the type of the highest ranking type in the expression), having “return g(...) + arg2” will implicitly convert the return type of g(...) into a long.

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

There are three globally available names that are known to this snippet of code:

g(), funa(), funb()

As a result, the symbol table will contain entries for g, funa, and funb. However, in this code snippet, the only references to these globally available symbols are when funa and funb call function g. As a result, there will be two relocation entries, one for where funa calls g and one for where funb calls g.

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.

Although page table entries logically belong to the page table, which is a data structure stored in RAM, in the ideal case, various forms of caching are used to try to reduce the amount of accesses to physical memory. As a result, the page table entries most commonly appear in the following:

- Translation Lookaside Buffer (TLB): This if the first place that the MMU looks for a PTE. If it misses, it must seek the PTE in the caches/RAM, but when it finds the PTE it was looking for, the very last thing it does is store that PTE in the TLB for use next time.
- L1/L2/L3 Cache: If the page table entry is not in the TLB, the series of caches are consulted. Because the page table is in physical memory just like any other data, when the MMU has to go to RAM to get the page table entry, the memory block that it sits in can be pulled into cache using normal caching operations.
- RAM/Physical Memory: If all else fails and the PTE is not in the TLB or the caches, the PTE is expected to be in memory since Page Tables are generally considered to be “memory resident” which means that unlike the virtual memory of normal programs which may be in RAM but may exist only on disk, the page tables are assumed to be in RAM. Note: According to the book, the Core i7 may swap page tables in or out of physical memory.

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?

Page Size = 1 GiB bytes/page = 2^30 bytes/page.

As a result, in order to index into a page of this size, we need 30 bits for page offset. The 2nd ed. book will tell you that the typical Nehalem implementation will have a 48-bit Virtual Addresses which means the PPN will be the remaining 18 bits, which mean a total of 2^18 pages in the virtual address space.

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.

Space advantage: Smaller page tables. Despite the larger page size, the entry of each PTE may actually decrease since if we assume the VA and PA sizes remains the same, the increase in the number of VPO/PPO bits implies that the PPN that you'd get from a PTE would be decreased from the normal case. Now, each page table entry governs a larger range of memory, which results in far fewer page table entries because there are far fewer pages.

Time advantage: Applications that access large sequential data structures will benefit from not needing to constantly swap in new pages for that data structure. If a single array spans 1 GiB, to iterate through this using the normal Nehalem 4 KiB page size (2^12 bytes per page), you'd need to page fault 2^18 times to pull it all into memory. With the 1 GiB huge page, it would take only page fault. Given that accessing the disk is millions of times slower than accessing the L1 cache, and thousands of times slower than accessing RAM, the considerable reduction of disk accesses could be a big win if you're accessing large, highly local data.

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

Space disadvantage: If you have a program that has small bits of data scattered around the virtual memory space (ie a little in the text segment, a little in the heap, a little in the stack), it is a waste of memory to have a 1 GiB page loaded for each small piece of data. If we have 4 MiB of data that we're interested in and they are spread across 4 pages, to pull this into RAM, we need to use of 4GiB of physical memory, even though we really only need 4 MiB of data.

Time disadvantage: Because each page is huge, actually copying a single page from disk to RAM will take much longer than in the case of a conservatively sized page.

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.

As mentioned above, if our program deals with a huge amount of data with high locality, (ex. data applications that work on large arrays, scientific computing, large matrices), the huge pages may provide better performance

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.

“p” and “amusing” are defined by the compiler as labels, which essentially make them pointers to memory in which data is initialized. The value of pointer “p” is initialized to 4 bytes of zero (as defined by the assembler directive “.zero 4”) and due to the “.bss” directive at line 9, it is placed in the bss section. The value of pointer “amusing” is initialized to three arbitrary bytes (as defined by the assembler directive “.byte XX”) and it is implicitly placed in the text section since the current section is not changed between the definition of main and amusing. The assumption is that these arbitrary bytes don’t just happen to be valid instructions.

In the unmodified case, the data at “p” is replaced with the address of “amusing” by line 3. This is because in a mov instruction, the ‘$’ has the same effect on labels as it does immediates; without the ‘$’, you are dereferencing the pointer. However, the call instruction treats labels a little differently. If you have “call <LABEL>”, you are calling a function whose address begins at <LABEL>. If you have “call *<LABEL>”, you are calling a function whose address is specified by the data at <LABEL>” Therefore, “call *p” will set %rip to be the data at p, which was previously set as the pointer to amusing. This means we are effectively calling “amusing” as a function and the next instruction to be executed is the byte configuration pointed to by “amusing”. Because this byte configuration was arbitrary and was not expected to be a real instruction, attempting to execute it causes a SIGILL, or illegal instruction.

In the modified case, you remove instruction 3 and therefore never replace the contents of “p” with the address of “amusing”. As a result, the instruction “call *p”, will find the value at “p” which is four bytes of 0x00 and then try to call the instruction whose address starts at 0x00000000. Because this address is out of bounds, you instead get a SIGSEGV, or a segmentation fault for attempting to access bad memory.

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
```

```
 subl   $28, %esp
 pushl  $h
 pushl  $4
 call   signal
 addl   $16, %esp
 movl   $1, %edx
.L3:
 rdrand %eax
 movl   %eax, -12(%ebp)
 cmovb  %edx, %eax
 testl  %eax, %eax
 je     .L3
 movl   -4(%ebp), %ecx
 movl   -12(%ebp), %eax
 leave
 leal   -4(%ecx), %esp
 ret
```

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.

No need to constantly mov %eax to -12(%ebp) inside the loop. Before the loop, the following instruction:

```
movl $1, %edx
```

... places 1 into %edx, which is subsequently used an “always 1” control variable.

The body of the loop does the following:

```
.L3:
  rdrand %eax            #Attempt to generate a random int into %eax. If
success %eax = rand and CF = 1. Else %eax = 0, CF = 0
  movl %eax, -12(%ebp)   #Back up the generated random variable into memory
  cmovb %edx, %eax       #If CF = 1 (ie. if the rdrand succeeded), mov 1
into %eax instead.
  testl %eax, %eax       #If rdrand failed, %eax would be 0. If it
succeeded, the cmov would have moved 1 into %eax
  je .L3                 #Jump if %eax was 0.
```

Since the rdrand sets the CF, the loop body really only needs to be:

```
.L3:
  rdrand %eax    # Attempt to generate a random int into %eax
  jae .L3        # If CF = 0 (generate random failed), jump back to .L3
```

Note: if this is what the loop is changed to be, the code immediately after the loop would need to be modified. Specifically, line 27 would need to be removed.

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.

Tricky parts:

This part aligns the stack and moves the return address to the fixed location:

```
      leal 4(%esp), %ecx
      andl $-16, %esp
      pushl -4(%ecx)
```

This part sets the signal handler to h() for signal 4:

```
      pushl $h
      pushl $4
      call signal
```

The body of the loop will attempt to generate a random number. If the loop random number is failed to be generated, the %rip will jump back to rdrand and attempt to generate the random number again. If it succeeds, the loop is exited.

The signal handler never gets called on the Ivy Bridge.

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.

Same as above, but the signal handler should it called for the illegal instruction and essentially does this, which will seg fault:

```
      h(int sig) { *(int *)sig = 1; }
```

The C signal function allows the user to register a function as the signal handler for a particular signal. It has the signature:

void (*signal(int *sig*, void (**func*)(int)))(int);

For non-insane people, this means you invoke signal() as follows:

```
signal(sig, func)
```

...where "sig" is the signal number (an int identifier for the type of signal) and func is a pointer to the function you want to be the signal handler. In this code, the signal is 4 (SIGILL) and the function to be run if a SIGILL is received is h(). Additionally, the function handler must take an int as an argument and when the function handler is called, the signal number is passed as that argument. In this case, when a signal arrives, h(sig) will be called. Since sig is 4, this code attempts to treat "4" as a pointer and tries to write to that location. Because the first million-ish or so addresses are forbidden, this will be a memory violation.

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.

In some ways, the answer is both. Valgrind is compiled into a .so file but when invoking “valgrind”, you’re actually calling a script which adds the the .so file to LD_PRELOAD. In this way, when you run the program, the Valgrind library is linked by the dynamic linker. In this way, the valgrind is under control of the linker. However, when the valgrind.so is initialized, it takes over and runs the program under it’s watchful eye. This means that valgrind runs the target program in a sort of valgrind-provided jail. The target program may include dynamic linker. In this sense, valgrind is in control of the linker

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

valgrind’s initialization (ie. the point at which valgrind takes over) could occur after other libraries are initialized. In other words, other libraries can be initialized without being checked by valgrind. If the other libraries have memory errors, valgrind would be unaware of them.

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?

“real CPU” and “synthetic CPU” are just terms concocted by the valgrind people to refer to the CPU when used to execute the code of valgrind (ie. the entity in control of monitoring a target program) versus the CPU when used to execute the code of the target program. The code of valgrind is run by the “real CPU”. Conversely, synthetic CPUs are used by valgrind to control the execution of the target program and the linker. As a result, the differentiation is irrelevant from the perspective of the Linux kernel or the memory system; it’s just the normal physical CPU, renamed based on how it is used.

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.

Possible ideas:

- Valgrind can load the target program into some location in memory and redirect all of its memory accesses properly, essentially performing the role of the linker. Thus, valgrind runs as the main code while allowing the loaded target program to run when necessary. This suggests that wherever the library is loaded must have write and execute permissions.
- Same as above, except valgrind relocates its own code.