Sample Midterm (Fall 2009)

Solutions are shown in this style. This exam was given in Fall 2009.

1. Executing Programs on IA32 (20 pts)

The following questions relate to how programs are compiled for IA32.

a) (8 pts) In lecture, we had discussed how each function obtains its own activation record, or stack frame, every time it is called. The stack frame is used for several purposes, including to hold the values of arguments passed to a function or to hold the values of local variables that cannot be kept in registers. Typically, accesses to these arguments and variables involve loads or stores that use relative addressing using the %ebp register as a base.
Recent versions of gcc support an optimization option '-fomit-frame-pointer' that organizes accesses to local variables differently. Instead of using the base/frame pointer register %ebp, the stack pointer register %esp is used to access local variables and arguments passed to a function. As a result, %ebp is available for other uses.

i. (6 pts) Explain why and how this would work!
Why is the base pointer, apparently, redundant?

The base pointer is always at known offset from the stack pointer, so any accesses that use addressing relative to $ebp can be replaced with accesses that use addressing relative to $esp.

ii. (2 pts) Consider the example of accessing the first argument, which is traditionally accessed using `8(%ebp)`.
How would code compiled with –fomit-frame-pointer access this argument?

Let SFSIZE = |$ebp-$esp| be the current stack frame size, then any access to disp($ebp) can be replaced with disp+SFSIZE($esp). For example, 8($ebp) would become 8+SFSIZE($esp). For example:

```
int sum(int x, int y)
{
    char localarray[16];
    return x + y;
}
```

When compiled with –fomit-frame-pointer (but without other optimizations), the code shows:

```
sum:
    subl    $16, %esp
    movl    24(%esp), %eax
    addl    20(%esp), %eax
    addl    $16, %esp
    ret
```

b) (12 pts) Consider the following assembly code, which was produced by gcc for a function 'g()'. The left column shows the result when compiling at the first level of optimization (-O1), the right column shows the result of compiling at the second optimization level.

IA 32 Code,compiled with –O1	IA 32 Code, compiled with –O2

g:
    pushl   %ebp
    movl   %esp, %ebp
    subl $8, %esp
    movl   8(%ebp), %eax
    movl   12(%ebp), %edx
    cmpl   %edx, %eax
    je .L2
    cmpl $1, %eax
    je .L6
    cmpl $1, %edx
    jne .L4
.L6:
    movl $1, %eax
    jmp .L2
.L4:
    cmpl   %edx, %eax
    jge .L7
    movl   %eax, 4(%esp)
    subl   %eax, %edx
    movl   %edx, (%esp)
    call   g
    jmp .L2
.L7:
    movl   %edx, 4(%esp)
    subl   %edx, %eax
    movl   %eax, (%esp)
    call   g
.L2:
    leave
    ret
	g:
	    pushl %ebp
	    movl %esp, %ebp
	    movl 8(%ebp), %edx
	    movl 12(%ebp), %ecx
	    cmpl %ecx, %edx
	    je .L3
	.L15:
	    cmpl $1, %edx
	    je .L5
	    cmpl $1, %ecx
	    je .L5
	    cmpl %ecx, %edx
	    jge .L9
	    movl %ecx, %eax
	    movl %edx, %ecx
	    subl %edx, %eax
	    movl %eax, %edx
	.L7:
	    cmpl %edx, %ecx
	    jne .L15
	.L3:
	    popl %ebp
	    movl %ecx, %eax
	    ret
	.L5:
	    popl %ebp
	    movl $1, %eax
	    ret
	.L9:
	    subl %ecx, %edx
	    jmp .L7

i. (9 pts) Provide a C version of function g()!
Hint: 'g' implements a well-known, classic mathematical algorithm!

'g' implements Euclid's algorithm for finding the greatest common divisor:

```
int g(int m, int n)
{
```

```
    if (m == n)
        return m;
    if (m == 1 || n == 1)
        return 1;
    return (m < n) ? g(n - m, m) : g(m - n, n);
}
```

ii. (3 pts) Which optimization did the compiler apply in the –O2 column that is not applied in the –O1 column?

The compiler applied recursion removal – the recursive calls were transformed into a loop.

2. Linking (22 pts)

The following questions center on linking and memory layout.

a) (12 pts) Assume you have three files a.c, b.c, and main.c with the following structure:

shared.h		
`int global_shared_variable = -1;`		
a.c	b.c	main.c
`#include "shared.h"`	`#include "shared.h"`	`int main() { }`

When you attempt to compile and link these files, you obtain:

```
$ gcc -c a.c b.c main.c
$ gcc a.o b.o main.o
b.o:(.data+0x0): multiple definition of `global_shared_variable'
a.o:(.data+0x0): first defined here
collect2: ld returned 1 exit status
```

1. (2 pts) Why does the error message say that a.o defines 'global_shared_variable' when in fact 'global_shared_variable' is defined in shared.h?

shared.h is not a compilation unit – the C preprocessor folds it into both a.c and b.c, respectively. The linker processes a.o, b.o, and main.o only.

2. You attempt to change shared.h to remove the initialization, i.e.:

```
int global_shared_variable;
```

You repeat the compilation and, in fact, the error goes away:

```
$ gcc -c a.c b.c main.c
$ gcc a.o b.o main.o
$
```

(2 pts) Why did the linker not report an error this time?

As an uninitialized global variable, global_shared_variable becomes a weak symbol, hence the linker will not report an error for multiple definitions.

3. Your teammate proposes to fix the error in a different way, by making the variable static, i.e., by changing shared.h to read:

   ```
   static int global_shared_variable = -1;
   ```

   You repeat the compilation and, in fact, the error is gone:

```
$ gcc -c a.c b.c main.c
$ gcc a.o b.o main.o
$
```

   (2 pts) Why did the linker not report an error this time?

global_shared_variable has become 2 distinct local symbols in a.o and b.o that happen to have the same name, hence there is no conflict for the linker to report.

   (2 pts) Explain why this solution is not a good one!

It would create 2 copies of this variable with distinct memory locations holding potentially different values, updates to one would not affect the other. This is likely not what the programmer intended when placing the definition into shared.h.

4. (4 pts) Complete the following table to show the correct way to address the issue in a way that avoids linker errors and allows the variable to be initialized!

shared.h		
`extern int global_shared_variable;`		
a.c	b.c	main.c
`#include “shared.h”` **`int global_shared_variable = -1;`**	`#include “shared.h”`	`int main() { }`

Alternatively, the definition could be contained in b.c or main.c
It's also possible to omit the 'extern', in which case the linker rule applies that a single strong definition in a.o overrides the weak definition in b.o. However, this is not good practice (-Wl,--warn-common would flag it). Some suggested placing 'extern int global_shared_variable' in b.c – this would compile and link, but is generally not considered sound programming practice.

b) (4 pts) A "fence" is a technique that is sometimes used to detect out-of-bounds memory accesses. The idea is to place some 'fence' values that rarely occur during normal execution before and after each array. Then, out-of-bounds accesses can be detected by checking whether the fence values were changed.
Complete the program below to implement this idea to protect array 'a' which is passed to a buggy update routine that contains out-of-bounds accesses.

```
void
buggy_update_array(int *array, int n, int delta)
{
    int i;
    for (i = 0; i <= n; i++) {
        array[i-1] = array[i] + delta;
    }
}

int a[10];

int main()
{
    buggy_update_array(a, 10, 1);
}
```

Knowing that the linker will allocate variables of the same storage class consecutively in memory, the program can be completed as follows:

```
void
buggy_update_array(int *array, int n, int delta)
{
    int i;
    for (i = 0; i <= n; i++) {
        array[i-1] = array[i] + delta;
    }
}

int leftfence;
int a[10];
int rightfence;

int main()
{
#define MAGIC 0xdeadbeef;
    leftfence = rightfence = MAGIC;
```

```
    buggy_update_array(a, 10, 1);
    assert (leftfence == MAGIC && rightfence == MAGIC);
}
```

c) (6 pts) During his recent distinguished lecture at Virginia Tech, Dr. Eugene Spafford pointed out that the vast majority of current security exploits involve dynamically linked libraries. He proposed to eliminate shared libraries. Discuss the merits of this idea! Provide at least 2 distinct arguments for or against it; keep your points brief!

Arguments in favor of Spafford's idea:

- *Eliminating shared libraries would remove the potential for attacks in which a program's accesses to OS functionality are intercepted and redirected when resolving a program's dynamic references. For instance, on Linux, the linker will check /etc/ld.so.preload for a list of libraries to be loaded first when linking a dynamically linked executable. Plus, an attacker could simply replace system libraries with their own version, affecting all applications.*

The counterarguments are the advantages of dynamic linking/shared libraries discussed in lecture, including

- *Shared libraries allow independent updates of the library code without requiring that all binaries referring to it be re-linked*
- *Shared libraries save physical memory since library code used by multiple processes can be shared*
- *Dynamic linking enables runtime extensibility, e.g., plug-ins.*

3. Locality (20 pts)

Consider the following naïve implementation of matrix transpose for large, dense matrices:

```
void inplace_transpose(int matrix[N][N])
{
    int i, j;
    for (i = 0; i < N - 1; i++) {
        for (j = i + 1; j < N; j++) {
            int tmp = matrix[i][j];
            matrix[i][j] = matrix[j][i];
            matrix[j][i] = tmp;
        }
    }
}
```

a) (5 pts) Does this algorithm exhibit temporal locality? Briefly say why or why not!

i. (2 pts) With respect to code

Yes, it uses loops whose instructions are executed many times.

ii. (3 pts) With respect to data

No, each matrix element is accessed once and only once. (Though less relevant, I also accepted 'yes' if you pointed out that there is reuse of 'i' and 'j' – but not 'tmp')

b) (5 pts) Does this algorithm exhibit spatial locality? Briefly say why or why not!

i. (2 pts) With respect to code

Yes – the executed code is contained in a contiguous section of instructions. (The fact that each backward branch in the loop causes a non-contiguous control transfer notwithstanding.)

ii. (3 pts) With respect to data

If the matrix is stored in row-major order, as in C, the accesses to matrix[i][j] exhibit spatial locality, but the accesses to matrix[j][i] do not.

c) (5 pts) Assume a memory hierarchy with just one level of caching and a cache line size of 64 bytes, which can hold 16 ints. How many cache misses would you expect per inner loop iteration?

Each loop iteration accesses both matrix[i][j] and matrix[j][i]. If the matrix is large enough (so that the distance between &matrix[k][x] and &matrix[k+1][x] is large), matrix[j][i] would miss every time, and matrix[i][j] every 16th time – once per cache line - thus we would expect 1+1/16=1.0625 cache misses per iteration.

d) (5 pts) In lecture we had discussed blocking as a method to speed up dense matrix multiplication. Could blocking be applied to speed up in-place matrix transposition? Briefly justify your answer!

Yes. Divide the matrix into small squares that fit in the cache, transpose the elements in each square block using a temporary buffer. If the temporary buffer, the source block, and the destination block fit into the cache, there will be no penalty for the lack of spatial locality because the cache block fetched when accessing the b[k][*] will still be in the cache when b[k+1][*] is accessed. This description is simplified: in practice, one needs to worry about conflict misses as well. This blocking avoids the cache misses due to lack of spatial locality when accessing neighboring columns; it does not introduce temporal locality.

4. Optimizations (18 pts)

a) (4 pts) Consider the following C code

```
void matrix_vector_multiply(int * y, int M[2][2], int * x)
{
    y[0] =  M[0][0] * x[0] + M[0][1] * x[1];
    y[1] =  M[1][0] * x[0] + M[1][1] * x[1];
}
```

Suppose you have an infinitely sophisticated compiler and you are using a machine with plenty of registers such as x86_64. How many memory load instructions and how many memory store instructions would the body of this function contain? (Not counting any accesses needed for stack frame management or saving callee-saved registers.)

Because x and y could refer to the same vector, we need 8 loads and 2 stores. Loads are for M[0][0], M[0][1], x[0], x[1], M[1][0], M[1][1], x[0], and x[1], stores for y[0] and y[1]. For example, here is the x86_64 code:

```
matrix_vector_multiply:
    movl    4(%rdx), %ecx      # load x[1]
    movl    (%rdx), %eax       # load x[0]
    imull   4(%rsi), %ecx      # load M[0][1]
    imull   (%rsi), %eax       # load M[0][0]
    addl    %eax, %ecx
    movl    %ecx, (%rdi)       # store y[0]
    movl    4(%rdx), %ecx      # load x[1]
    movl    (%rdx), %eax       # load x[0]
    imull   12(%rsi), %ecx     # load M[1][1]
    imull   8(%rsi), %eax      # load M[1][0]
    addl    %eax, %ecx
    movl    %ecx, 4(%rdi)      # store y[1]
    ret
```

b) (4 pts) Now consider this C function, which is almost identical to the one above, except that the matrix M is no longer a nested array:

```
void matrix_vector_multiply2(int * y, int *M[], int * x)
{
    y[0] =  M[0][0] * x[0] + M[0][1] * x[1];
    y[1] =  M[1][0] * x[0] + M[1][1] * x[1];
}
```

How many memory load and store instructions would a compiler emit for this function?

Here, the address of the first element of an array of pointers is passed to the function, which must be dereferenced to obtain M[0] and M[1]. This adds two loads, hence 10 loads and 2 stores.

```
matrix_vector_multiply2:
    movq    (%rsi), %r8         # load M[0]
    movl    4(%rdx), %ecx       # load x[1]
    movl    (%rdx), %eax        # load x[0]
    imull   4(%r8), %ecx        # load M[0][0]
    imull   (%r8), %eax         # load M[0][1]
    addl    %eax, %ecx
    movq    8(%rsi), %rax       # load M[1]
    movl    %ecx, (%rdi)        # store y[0]
    movl    4(%rax), %ecx       # load M[1][1]
    movl    (%rax), %eax        # load M[1][0]
    imull   4(%rdx), %ecx       # load x[1]
    imull   (%rdx), %eax        # load x[0]
    addl    %eax, %ecx
    movl    %ecx, 4(%rdi)       # store y[1]
    ret
```

c) (4 pts) The ISO-C99 standard added a keyword 'restrict' to the language which can be applied to pointer variables. 'restrict' says that a given pointer is the only means of accessing the data to which it points within the scope in which it is declared; no other pointer will refer to the same data. If these assumptions are violated, undefined behavior will result. Below, the 'restrict' keyword is applied to x and y.

```
void matrix_vector_multiply_r(int * restrict y, int M[2][2],
                              int * restrict x)
{
    y[0] =  M[0][0] * x[0] + M[0][1] * x[1];
    y[1] =  M[1][0] * x[0] + M[1][1] * x[1];
}
```

How many memory load and store instructions would a sophisticated compiler emit for this function? Note that M is again a nested array as in part a)!

If x is assumed to not refer to the same location as y – which is what 'restrict' implies – then we only need 6 loads and 2 stores since x[1] and x[0] do not be read from memory again after y[0] is written.

```
matrix_vector_multiply_r:
    movl    4(%rdx), %r8d       # load x[1]
    movl    (%rdx), %ecx        # load x[0]
    movl    %r8d, %eax
    movl    %ecx, %edx
```

```
imull   4(%rsi), %eax     # load M[0][1]
imull   (%rsi), %edx      # load M[0][0]
addl    %edx, %eax
movl    %eax, (%rdi)      # store y[0]
imull   12(%rsi), %r8d    # load M[1][1]
imull   8(%rsi), %ecx     # load M[1][0]
addl    %ecx, %r8d
movl    %r8d, 4(%rdi)     # store y[1]
ret
```

d) (6 pts) Assuming an optimizing compiler, is there always a performance cost for declaring many local variables within one function? If yes, say why. If not, explain precisely when there is a cost and when there isn't!

No, not always. Optimizing compilers perform register allocation. The number of declared local variables does not matter unless the lifetime of these local variables overlaps. Each register can hold only one local variable at a time, if there are more local variables alive at any point in a function than there are registers, spilling occurs and a performance penalty is paid.

5. Unix Process Management (20pts)

a) (14 pts) Consider the following example programs. List all legal outputs this program may produce when executed on a Unix system. The output consists of strings made up of multiple letters.

<table>
<tr><td>

```
// included in both programs
#include <unistd.h>
#include <sys/wait.h>
// W(A) means write(1, "A", sizeof "A")
#define W(x) write(1, #x, sizeof #x)
```

</td><td>Possible Outputs:</td></tr>
<tr><td>

```
int main()
{
    W(A);
    fork();
    W(B);
    fork();
    W(C);
}
```

</td><td>i) 6 pts

Possible outputs are:
ABBCCCC
ABCBCCC
ABCCBCC</td></tr>
<tr><td>

```
int main()
{
    W(A);
    int child = fork();
    W(B);
    if (child)
        wait(NULL);
```

</td><td>ii) 8 pts

Possible outputs are:
ABCBC
ABBCC</td></tr>
</table>

```
    W(C);
}
```

There is a bug in the program: it should be write(1, #x, sizeof #x – 1). The program as is outputs a '\0' character, which however does not appear on the terminal.

b) (6 pts) Consider the following two programs. Below each program is shown the output sent to the terminal when the program is run:

```
int main() {
    if (fork())
        *(int *)0 = 42;
}
```

Output:

```
$ ./crash
Segmentation fault
$
```

```
int main() {
    if (!fork())
        *(int *)0 = 42;
}
```

Output:

```
$ ./crash
$
```

Why is the message "Segmentation fault" displayed for the program on the left, but not for the program on the right?

The segmentation fault message is displayed by the shell if a child process is terminated with signal 11, SIGSEGV. On the left, where fork() returns not zero, the shell's child is terminated. On the right, the process that is terminated is the child process, which is a grandchild of the shell. 'wait()' does not allow the shell to wait for grandchildren, hence the shell cannot learn that the process terminated with a fault, hence no message.
Note that this behavior occurs independent of whether the scheduler runs the parent or the child first after the fork (on a single processor system).