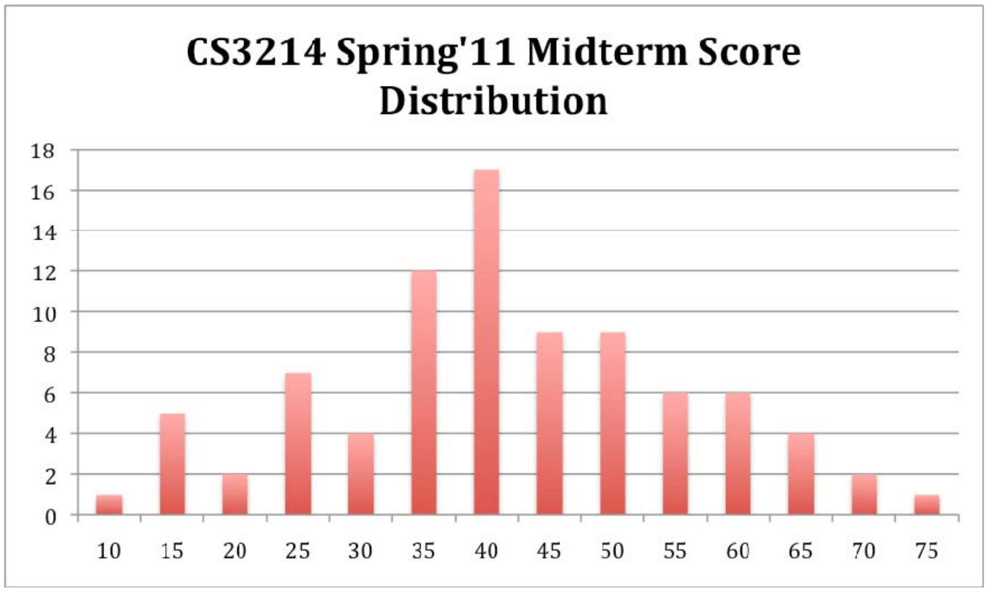
CS 3214 Midterm



#	Problem	Points	Min	Max	Average	Median	SD	Grader
1	Memory Layout and Locality	25	2	25	14.2	14	5.7	Bill
2	Stack	25	3	22	12.6	13	4.2	Peter
3	Compilation and Linking	25	0	19	7.6	6	4.7	Maggie
4	Execution and Optimization	25	0	25	9.3	8	6.2	Ali
	Total	100	14	76	43.6	43	14.0	

1. Memory Layout and Locality (25 pts)

Consider the following C implementation of a function to sum the columns of a two-dimensional matrix; a precondition of the function is that the array `Sum[]` has been initialized to hold zeros.

```
void sumCols(int M, int N, int Sum[M], int A[N][M]) {

   for (int j = 0; j < M; j++) {
      for (int i = 0; i < N; i++) {
         Sum[i] = Sum[i] + A[i][j];
      }
   }
}
```

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

i. (2 pts) with respect to code?

Yes. The recomputation of Sum[i] is executed N times in succession during each pass through the inner loop.

Note: the question is about locality wrt code accesses, not data accesses. Answers that referred to variables, like i or Sum[i] are talking about data accesses. The question relates to the way in which the machine language translation of this code would be managed at run-time (although you did not need to take the effects of the translation into account).

ii. (4 pts) with respect to data?

Very little. No elements of Sum[] or A[][] are accessed more than once during a pass through the inner loop; there is some slight locality in that the loop counters i and j are accessed repeatedly.

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

i. (2 pts) with respect to code?

Yes. The executed code would be stored in a relatively small, contiguous section of instruction memory, and the instructions would be executed repeatedly (due to the loop).

ii. (4 pts) with respect to data?

Yes. The inner loop traverses Sum[] with stride 1, indicating good spatial locality wrt Sum[]. Since A[][] is stored in row-major order, and the inner loop drives the row counter i, not the column counter j, successive accesses to A[][] are with stride N, which indicates poor spatial locality wrt A[][].

c) (7 pts) Assume a memory hierarchy with just one level of caching, and a cache line size of 48 bytes (12 ints). Assume that the dimensions of the arrays are large in relation to the size of the cache. How many cache misses would you expect to occur per iteration of the inner loop?

Note: A cache line is one storage unit of the cache, not the entire cache, which would consist of many lines. When data is fetched into the cache, a whole line is populated (from RAM) at once.

S[] is traversed with stride 1, so a line of cache would store 12 successive, relevant elements of S[]; we should expect a pattern of 1 miss followed by 11 hits for S[].

Since accesses in A[][] are via stride N, and we are assuming the array dimensions are large in relation to the cache size, we would expect that each fetch into a cache line would usually fetch only one value from the current row of A[][]. Therefore, almost all accesses to A[][] would result in a cache miss.

So, on a single pass, we'd expect 1 + 1/12 = 13/12 or about 1.0833 cache misses per pass.

Now consider the following alternative implementation:

```
void sumCols(int M, int N, int Sum[M], int A[N][M]) {

   for (int i = 0; i < N; i++) {
      for (int j = 0; j < M; j++) {
         Sum[i] = Sum[i] + A[i][j];
      }
   }
}
```

d) (3 pts) Does the alternative implementation exhibit temporal locality with respect to data? Briefly say why or why not!

Yes. Now the same element of Sum[] is accessed on every pass through the inner loop; as before, there is some slight locality in that the loop counters i and j are accessed repeatedly.

However, it's still true that no element of A[][] is used more than once.

e) (3 pts) Does the alternative implementation exhibit spatial locality with respect to data? Briefly say why or why not!

Yes. The accesses to Sum[] (driven by the outer loop) are still stride 1, indicating good spatial locality for Sum[]. However, now the accesses to A[][] are also via stride 1 since the inner loop walks across row i of A. So we would now see good spatial locality with respect to A as well.

2. Stack (25 pts)

Consider the following low-quality C implementation of the `getline` function:

```
 1:  char *getline() {
 2:     char buf[8];
 3:     char *result;
 4:     gets(buf);
 5:     result = malloc(strlen(buf));
 6:     strcpy(result, buf);
 7:     return result;
 8:  }
```

We obtain the following disassembly of `getline`, up to the call to `gets` in line 4:

```
 1:  080485c0 <getline>:
 2:  80485c0:  55                 push   %ebp
 3:  80485c1:  89 e5              mov    %esp, %ebp
 4:  80485c3:  83 ec 28           sub    $0x28, %esp
 5:  80485c6:  89 5d f4           mov    %ebx, -0xc(%ebp)
 6:  80485c9:  89 75 f8           mov    %esi, -0x8(%ebp)
 7:  80485cc:  89 7d fc           mov    %edi, -0x4(%ebp)
```
Part a) refers to this point in the code
```
 8:  80485cf:  8d 75 ec           lea    -0x14(%ebp), %esi
 9:  80485d2:  89 34 24           mov    %esi, (%esp)
10:  80485d5:  e8 a3 ff ff ff     call   804857d <gets>
```
Part b) refers to this point in the code

Suppose that `getline` is called with the return address equal to `0x8048643`, register `%ebp` equal to `0x2`, and register `%esi` equal to `0x3`.

Note: this question was intended to be practice problem 3.43 from the text, but part of the preceding paragraph was inadvertently omitted.

You type in the following string: `012345678901234567890123`

FYI: digits are assigned consecutive ASCII codes, and '0' is represented by 0x30.

The program terminates with a segmentation fault, and when you run `GDB` you determine that the error occurs during the execution of the `ret` instruction in `getline`.

a) (9 pts) Fill in the diagram below, showing as much detail as you can determine about the state of the stack immediately after the execution of line 7 in the disassembly. Label the quantities stored on the stack (e.g., "Return address") on the right, and show the hexadecimal values (if known) within the table. Each cell of the table represents 4 bytes. Indicate the position of `%ebp`.

`08 04 86 43`	Return address **written to the stack before getline() is called**
`00 00 00 02`	**Saved `%ebp` – pushed by instruction 3**
`?? ?? ?? ??`	**Saved `%edi` – pushed by instruction 7; note the offsets that are used relative to %ebp**
`00 00 00 03`	**Saved `%esi` – pushed by instruction 6**
`?? ?? ?? ??`	**Saved `%ebx` – pushed by instruction 5**
	`buf[4-7]` – note the stack pointer is moved 0x28 (40) bytes by instruction 4; 16 bytes are used above
	`buf[0-3]` – that leaves a residue of 24 bytes, which provides space for buf[] (and more)

b) (6 pts) Redraw your diagram to show the effect (on the stack) of the call to `gets` in line 10 of the disassembly.

`08 04 86 00`	Return address
`33 32 31 30`	`Saved %ebp`
`39 38 37 36`	`Saved %edi`
`35 34 33 32`	`Saved %esi`
`31 30 39 38`	`Saved %ebx`
`37 36 35 34`	`buf[7-4]`
`33 32 31 30`	`buf[3-0]`

The call to gets() will 24 bytes of data (corresponding to the ASCII codes for the given string) into memory, starting at the address passed to gets(), which points to buf[0], and follow that with a zero byte to terminate the string.

So, the first 8 bytes will fill buff[], and the next 16 will overwrite the next 16 bytes (destroying the register backups). The zero byte will then overwrite the last stored byte of the return address.

c) (2 pts) To what address does the program attempt to return?

`08 04 86 00 :` low-order byte was overwritten by the string terminator

d) (4 pts) What register(s) have corrupted value(s) when `getline` returns?

The saved values of the following registers were altered:

%ebp
%edi
%esi
%ebx

e) (4 pts) Aside from the potential for buffer overflow, what other things are wrong with the given C code for `getline`?

The parameter in the call to malloc is incorrect because it does not allow for the necessary terminating byte; it should be strlen(buf) + 1.

The implementation also fails to check whether the return value from the call to malloc is NULL.

One could argue the call to strcpy() should be replaced with a call to strncpy(), but if gets() is correct, and if the problem in the call to malloc() is corrected, then the call to strcpy() is safe.

3. Compilation & Linking (25 pts)

Consider the following programs:

prog_a.c:	prog_b.c:	prog_c.c:

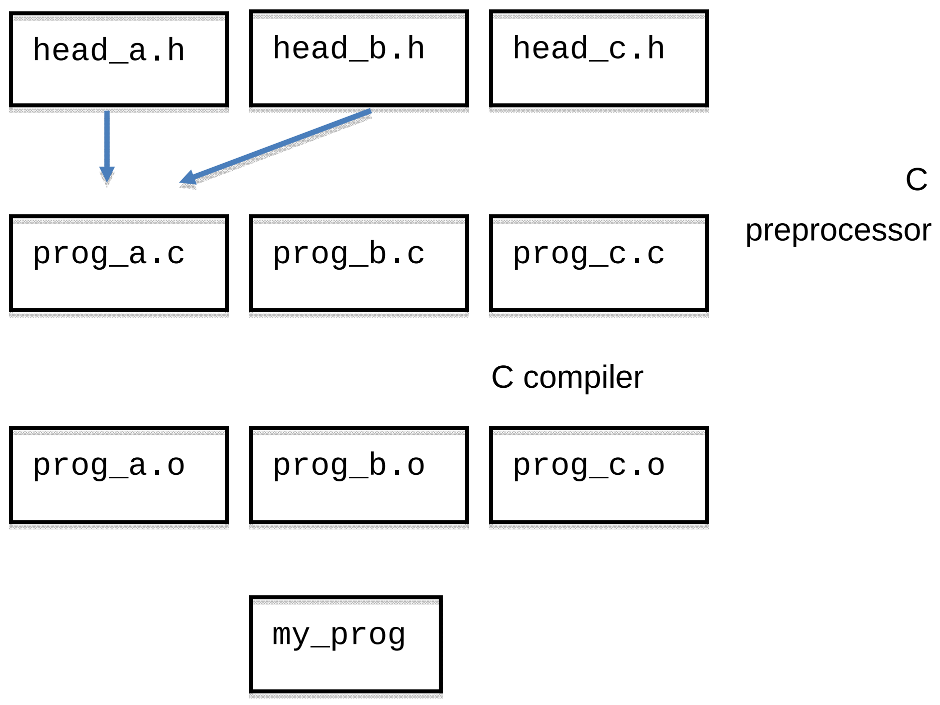
`#include <head_a.h>`
`#include <head_b.h>`
 	
`int main ()`
`{`
`my_func_c();`
`my_func_b();`
`}`
	`#include <head_a.h>`
	
	`void my_func_b()`
	`{`
	`lib_func_a();`
	`my_func_c();`
	`}`
		`#include <head_a.h>`
		`#include <head_b.h>`
		`#include <head_c.h>`
		
		`void my_func_c()`
		`{`
		`lib_func_c();`
		`lib_func_d();`
		`}`

Assume all undefined functions are defined in a *standard* library `lib_alpha.so`

a) (2 pts) Write the GCC command to compile the programs into a binary named `my_prog` using dynamic linking.

```
gcc prog_a.c prog_b.c prog_c.c lib_alpha.so -o my_prog
```

b) (6 pts) Draw a flow graph showing how the different files are processed and converted into the binary. Clearly mark the dependencies between the files, and name the tools that are used at each stage. Also show how `lib_alpha.so` is accessed during loading.



c) A user wants to run `my_prog` but wants to use her own version of `lib_func_d()` and not the one defined in the library `lib_alpha.so`. Outline and give commands she will use to achieve this if:

i. (3 pts) the above source files (not the library source) are available;

Change either the source file to use new custom function, e.g., `my_lib_func_d()`, or redefine the function in the c code.

ii. (3 pts) the source files are not available.

Use LD_PRELOAD to override the function defined in the standard library

d) (3 + 3 pts) Now assume that `my_prog` is created using static linking, repeat (c).

(i) Same as before
(ii) It is not possible to do this as LD_PRELOAD only works for dynamically linked libraries.

e) In *NIX environments, when a user `bob` executes a binary, say `gimp`, `gimp` executes with the privileges of `bob`, even though `gimp` is owned by the user `root`. A special case occurs when the `setuid` attribute of the binary is enabled. In this case, the binary will run with the privileges of the user who owns the binary and not as `bob` who runs the program. For instance, `ping` is owned by `root` with `setuid` enabled. Thus, when `bob` executes `ping`, it runs with `root` privileges.

i. (3 pts) Based on your answers to part (c) and (d), discuss one problem that can arise in dynamically linked programs that have `setuid` enabled.

A user can use LD_PRELOAD and inject malicious code into a binary, which will then run with root privileges.

ii. (2 pts) Suggest an efficient solution for addressing this problem.

Do not allow LD_PRELOAD for setuid programs.

4. IA32 Execution and Optimizations (25 pts)

Consider the following function compiled using `gcc`:

```
calculate:
 1:      push   %ebp
 2:      mov    %esp,%ebp
 3:      sub    $0x10,%esp
 4:      movl   $0x0,-0x10(%ebp)
 5:      movl   $0x1,-0xc(%ebp)
 6:      movl   $0x0,-0x4(%ebp)
 7:      jmp    <line 22>
 8:      mov    -0xc(%ebp),%eax
 9:      add    -0x10(%ebp),%eax
10:      mov    %eax,-0x8(%ebp)
11:      mov    -0xc(%ebp),%eax
12:      mov    %eax,-0x10(%ebp)
13:      mov    -0x8(%ebp),%eax
14:      mov    %eax,-0xc(%ebp)
15:      mov    -0x4(%ebp),%eax
16:      shl    $0x2,%eax
17:      mov    %eax,%edx
18:      add    0xc(%ebp),%edx
19:      mov    -0x8(%ebp),%eax
20:      mov    %eax,(%edx)
21:      addl   $0x1,-0x4(%ebp)
22:      mov    -0x4(%ebp),%eax
23:      cmp    0x8(%ebp),%eax
24:      jl     <line 8>
25:      leave
26:      ret
```

`calculate_hand_opt`:

Uses registers instead of accessing memory again and again.
Uses `leal` to calculate array address

a) (10 pts) Write a C version of the function `calculate`.

```
void calculate (int n, int *result)
{
    int a,b,c,i;
    a=0;
    b=1;

    for(i=0;i<n;i++) {
       c=a+b;
       a=b;
       b=c;
       result[i]=c;
    }
}
```

b) (2 pts) Explain what line 3 of the assembly code does and why.

Reserves space for temporary/local variables on the stack.

c) (2 pts) Based on your examination of the code, what can you say about the `gcc` optimization level used for compiling `calcuate` as shown in the provided code?

No optimization. Reason: code contains repetitive access to memory, thus code is emitted for each instruction.

d) (2 pts) What is the main performance bottleneck in the provided code for `calculate`?

Excessive memory accesses.

e) (9 pts) In the space provided above under `calculate_hand_opt`, ***sketch*** what you think would be a more optimized assembly code version of `calculate`. If you do not recall the exact format of specific IA32 instructions, you can use pseudo code, i.e., use plain English to express what you want an instruction to do.

Shown in the space above.