UCLA Computer Science 33 (Spring 2015)
Midterm
108 minutes total, open book, open notes
Questions are equally weighted (12 minutes each)

Name:_____ Student ID:_____

1	2	3	4	5	6	7	8	9	sum

1. Suppose you want a thread T1 to wait until thread T2 finishes, and that T2 is detached. Explain how to arrange for this reliably, assuming the threads cooperate by executing code that you specify. Your code may invoke any of the pthread_* or sem_* primitives discussed in the book or in class. Explain any assumptions you make and any race conditions that you couldn't fix (these should be reasonable and few).

NOTE: This question deals with the topic of synchronization, which you (the Fall '15 CS 33 class) will not be tested on in your midterm.

Thread T1 should not execute until T2 is complete. T2 is a detached thread, which means you can't use pthread_join(tid2,..) and must use a semaphore. Assume that T1 will run the code specified by thread1 and T2 will run the code specified by thread2.

```
int main( ) {
     sem_init(&mutex, 0, 0);    // Initialize sem value to 0
     ...
}
void* thread1(void* arg) {
     sem_wait(&mutex);          //  T1 does P(s) before executing
code
     <CODE FOR T1>
}
```

```
void* thread2(void* arg) {
     <CODE FOR T2>
     sem_post(&mutex);          // T2 does V(s) after executing code
}
```

If we initialize the semaphore "mutex" to be 0, then the code in thread1 will wait with the sem_wait function until the mutex is incremented. The mutex is only incremented by T2 which will execute its code and then call sem_post once it is ready to exit. Note: technically, there is no guarantee that T1 will wait until T2 has properly finished because this configuration will allow T1 to execute as soon as T2 calls sem_post. As a result, the scheduler could allow T1 to execute immediately after T2 calls sem_post but before T2 actually exits the thread2 function.

2. Suppose we extend the bitwise operations ^, &, | and ~ to operate on floating-point values by applying these bitwise operations to their representations. For example, since the binary representation of 0.1f is 0x3dcccccd, and ~0x3dcccccd == 0xc2333332, and 0xc2333332 represents -44.799995f, then ~0.1f would yield -44.799995f and ~-44.799995f would yield 0.1f. Recall that the general rule for floating point operations is that NaNs are infectious, i.e., that if one or both inputs to a floating-point operation is a NaN, then the operation yields a NaN. Which (if any) of the bitwise operations ^, &, | and ~ are infectious on NaNs? Explain.

OR is the only infectious bitwise operator. NaNs are identified by an exponent field that is all 1s and a non-zero fractional field. Thus, we are looking for a bitwise operation that will, with certainty, result in an exponent field of all 1s and a non-zero fractional field. With |, if either of the operands is a NaN, the resulting value must be a NaN because anything OR'd with a 1 is a 1. Any of the other operations can or must yield non-NaN values when applied to a NaN. For example: <bitwise ~>NaN will result in an all 0 fractional field, NaN <bitwise ^> 0xFFF...FF has the same effect as <bitwise ~>NaN, NaN <bitwise &> 0 = 0.

3. The function stricmp(A, B) compares the two strings A and B ignoring case, and returns an

int. If we let a = the lowercased version of A and b = the lowercased version of B, then stricmp(A, B) returns a negative number if a compares less than b, a positive number if a compares greater than b, and zero otherwise. This function compares strings byte by byte, and assumes only the 52 ASCII letters.

Consider the following stricmp implementation. Assume that it's running on the x86.

```
#include <string.h>

#define min(a, b) ((a) < (b) ? (a) : (b))

char
cvtlower (char c)
{
  if ('A' <= c && c <= 'Z')
    return c - 'A' + 'a';
  return c;
}

int
stricmp (char const *a, char const *b)
{
  for (size_t i = 0;
       i < min(strlen(a), strlen(b));
       i++)
    if (cvtlower(a[i]) < cvtlower(b[i]))
      return -1;
    else if (cvtlower(a[i]) > cvtlower(b[i]))
      return 1;
  return 0;
}
```

Propose two optimizations of this code, at least one of which is likely to improve performance greatly and the other at least somewhat. Explain why the former is likely to be better than the latter.

Significant Improvement:

Hoist out strlen(a) - This can be done either by doing:

```
int min = min(strlen(a), strlen(b));
for (size_t i = 0; i < min; i++)
```

...or frankly even:

```
int len_a = strlen(a);
int len_b = strlen(b));
for (size_t i = 0; i < min(len_a, len_b); i++)
```

The reason that this provides such a considerable improvement is because strlen will perform n operations where n is the length of the string. Thus, for each iteration of the loop (say there are m iterations), we are performing two strlens. This means the function is doing 2*n*m operations as a result of the strlen. When strlen is hoisted out, the function only does 2*n operations as a result of the strlen.

Minor Improvement:
Reduce the number of calls to cvtlower - This can be done as follows:

```
for (size_t i = 0; i < min(strlen(a), strlen(b)); i++)
{
  char lower_a = cvtlower(a[i]);
  char lower_b = cvtlower(b[i]);
  if (lower_a < lower_b))
    return -1;
  else if (lower_a > lower_b)
    return 1;
}
```

Previously, each loop was performing 4 calls to cvtlower(), which is an O(1) function. This means cvtlower was contributing a total of 4*m operations. Now, it only performs 2 calls per iteration for a total of 2*m operations. Loop unrolling could also have also provided a minor decrease in the number of loop overhead operations.

4. Consider the following C function and its translation to x86-64 assembly language. The C function returns a[i] in the typical case where i is in range, and returns 0 otherwise:

```
int
subscript (int *a, unsigned i,
         unsigned int n)
{
  if (0 <= i && i < n)
    return a[i];
  return 0;
}

subscript:
      xorl  %eax, %eax
      cmpl  %edx, %esi
      jnb   .L2
      movl  %esi, %esi
      movl  (%rdi,%rsi,4), %eax
.L2:
      ret
```

4a. How can this code be correct? The source has two comparisons, but the assembler has just one.

The comparison of 0 <= i is unnecessary and will be optimized out since i is unsigned which means this condition will always be true.

4b. Why aren't conditional moves helpful for improving this code's performance? Explain.

The typical case for subscript is that the index is within the bounds. In this code snippet, dynamic branch prediction will generally predict that the branch will not be taken and as a result, it will speculatively perform the instructions for returning a[i]. Assuming a typical access pattern for arrays, this prediction will be successful most of the time, so conditional moves won't improve much.

5. Suppose your program has three parts that are done in sequence and take 0.5, 0.3, and 0.2 of the time respectively. You can parallelize the first part and speed it up by a factor of 2. Or you can parallelize the second part and speed it up by a factor of 8. Use Amdahl's law to calculate which of these two will give you better performance and why. Suppose you can do both

parallelizations: how much will your performance improve compared to the original, or to either parallelization alone? Show your work.

Original:

T0 = 0.5 + 0.3 + 0.2

Parallelize the first part by a factor of 2:

T1 = 0.5/2 + 0.3 + 0.2 = 0.75

Parallelize the second part by a factor of 8:

T2 = 0.5 + 0.3/8 + 0.2 = 0.5 + 0.0375 + 0.2 = 0.7375

Parallelize both parts:

T3 = 0.5/2 + 0.3/8 + 0.2 = 0.4875

T3 < T2 < T1

6. Let d = the number 0.1 in C (i.e., the 'double' value 0.1). Let f = the number 0.1f in C (i.e., the 'float' value 0.1f). And let r = the number 0.1 in mathematics (i.e., the real number equal to 1 divided by 10). Recall that the binary representation of r is the repeating sequence 0.0001100110011001100110011... base 2. Sort the values d, f, and r into nondescending order. If two or more of these three values are equal, say so. Assume x86-64 arithmetic with default rounding. Show your work.

The key observation is that r is infinitely repeating while f and d must terminate. Because of this, we will not be able to represent 1/10 precisely with a single or double precision floating point. As a result, f and d may be rounded up (to be greater than 1/10) or rounded down (to be less than 1/10). In order to see which way it rounds, we need to examine the binary.

r = .000110011001100...

The greatest power of 2 in this number is 2^-4. An exponent of that range will required normalized representation in both float and double form. Because the fractional contribution of normalized form includes an implicit 1, we have to rewrite the binary to be of this form:

```
f/d = 2^-4 * (1 + ...)
f/d = 2^-4 * 1.frac
frac = 100110011001... (or sequence of 1001s repeated)
```

There are 23 fractional bits in a float. This means that the sequence of repeated 1001s will truncate with the third (23 % 4 = 3) digit of 1001 (or 100|1). There are 52 fractional bits in double. This means that the sequence ends in the last digit of the sequence. Thus (the vertical bar represents the cutoff):

```
float:  frac = 10011001...1001100|110011001...
double: frac = 10011001...1001100 110011001...10011001|10011001
```

The default rounding mode is "round-to-even", which actually rounds to the closest value and rounds to even only to break a tie (when the actual value is exactly in the middle of two representable numbers). In order to determine if there is a tie, consider the bits to the right of the cutoff. If the bit immediately to the right is 1 and the rest of the bits are 0, then there is a tie. However in this instance, because the bit immediately to the right is 1 AND there are repeated 1s beyond that, both values are actually closer to the rounded-up value. Thus, the actual values will be:

```
float:  frac = 10011001...1001101
double: frac = 10011001...10011001...10011010
real:   frac = 10011001...10011001...100110011001...
```

As a result, both f and d are greater than r. However, f rounds up at a much greater position than the double because the floating point has less precision:

```
float: 10011001...100110100000...
doub:  10011001...100110011001...
```

As a result, the ascending order is r,d,f or what is known in the industry as "The Reverse Franklin Delano Roosevelt".

7. We have a special kind of SRAM cache called Cache Z. Cache Z uses a write-back approach, but omits the dirty bit. Instead, whenever it needs to know whether a cache line is dirty, it loads the corresponding data from RAM and compares it to the data in the cache line: if they compare equal, Cache Z acts as if the dirty bit were zero, and if they compare nonequal, Cache Z acts as if the dirty bit were nonzero. Compare the pros and cons of Cache Z to an ordinary write-back write-allocate cache. Propose an improvement to Cache Z's performance that does not involve adding a dirty bit.

According to Cache Z's policy, each time a block must be evicted, you need to go to memory to read the corresponding block. Then, you compare the value of the block in the cache to the value that was fetched from memory. If the values are different, you write the updated block to memory. This means that for each eviction, you MUST read from memory and you might have to write back to memory.

Pros of Cache Z:

- Unlike in a traditional write-back cache, you do not need to store the dirty bit. You save a single bit per block

Cons of Cache Z:

- Each eviction costs at least a memory access and requires 2 memory accesses if the block is dirty. A traditional write back cache will only require 1 memory access if the block is dirty.

Improvement:

- When evicting, simply write the block in cache back to memory without doing an additional check.
- Consider a case where there are n evictions and n/2 of those evictions need to be written back to memory. In Cache Z, this will require 1.5*n memory accesses.
- With this improvement, n memory accesses are necessary, which means that it will always perform at least as well as Cache Z which will require between n and 2n memory accesses.

8. Suppose you have an x86-64 machine with the following characteristics:

two sockets
12 CPU cores per socket
L1 instruction cache: 32 KiB per core
L1 data cache: 32 KiB per core
L2 cache: 256 KiB per core
L3 cache: 30 MiB per socket
64 GiB DRAM per socket

L1 and L2 caches are private to each core.
L3 cache is shared by all cores in a socket.
All caches are writeback.

8a. The single instruction cache at L1 is fast but very small: why won't performance suffer greatly if your program's kernel doesn't fit into 32 KiB?

The architecture includes a unified L2 and L3 cache. As a result, if the working set of instructions doesn't fit into L1, we can always use the L2/L3 caches. Since the L2 and L3 caches are still quite fast, performance won't be greatly impacted.

8b. Consider the following program, and assume its x86-64 code fits entirely within the L1 instruction cache, and assume that the source and destination do not overlap.

```
#define N <<you pick the constant>>
void transpose(int dst[N][N], int src[N][N]) {
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
    dst[j][i] = src[i][j];
}
```

Suppose this function is often executed in your multithreaded application on the specified machine. What values of N do you recommend for good performance, and why? Look for local sweet spots for N. State any further assumptions you're making.

The best N will be one that will allow both the src and dst arrays to fit entirely in one of the caches (ie L1, L2, or L3). In general, we want to find an N that satisfies the equation:

4 bytes/int * N^2 ints/array * 2 arrays <= sizeof(cache)

For example, if we want to fit src and dst in the L1 cache:
4 * N^2 * 2 = 32 KiB
2^3 * N^2 = 2^15
N^2 = 2^12
N = 2^6 = 64

So, any N<64 will result in the src and dst arrays fitting entirely in the L1 cache.

9. Section 5.9.2 of the book shows how to apply the associative law to improve performance while unrolling a loop. Can we use a similar idea to improve performance by applying the distributive law A*(B + C) == A*B + A*C? Why or why not?

There are two cases to consider. The first case is when A is the accumulator.

(1) a = a*(b[i] + c[i])
(2) a = a*b[i] + a*c[i]

For (1), the bottleneck would be the multiplication operation, since we can perform the addition of iteration i + 1 before waiting for the multiplication of iteration i to finish. Thus, the critical path would consist only of the multiplication.

For (2), the bottleneck involves the multiplication operations (both of which can be done in parallel) and the addition operation. In this case, the two multiplications and the addition must be done sequentially; we cannot do the multiplication of iteration i + 1 while doing the addition of iteration i because the multiplication of iteration i + 1 relies on the updated value of "a" from the addition of iteration i. Thus, the critical path would consist of a multiplication operation and an addition operation.

Since (2) is not as efficient as (1), the distributive law doesn't improve performance.

However, if you assume that the accumulator is b, we would have:

```
(1)   b = a[i] * (b + c[i])
(2)   b = a[i] * b + a[i] * c[i]
```

Since the bottleneck is the same for both cases, applying the distributive law doesn't matter. In (1), the critical path involves an addition and then a multiplication. The addition of iteration i + 1 cannot be done in parallel with the multiplication of iteration i because the addition uses "b" which relies on the result produced by the multiplication of the previous iteration. This is also true of (2).