14

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

Name: Michael Xiong Student ID: 404 463 570

1	2	3	4	5	6	7	8	9	sum
6	12	12	4	11	2	12	9	6	74

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).

```
sem_t mutex;
int main() {
    sem_init(&mutex, 0, 1);
    pthread_t T1, T2;
    Pthread_create(&T1, Null, thread, Null);
    Pthread_create(&T2, Null, thread, Null);
    Pthread_detach(T2, Null);
}
void *thread() {
    P(&mutex);   // T1 can execute first
    /* do something */
    V(&mutex);
}
```

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 ~ infectious on NaNs? Explain.

bit-wise or (|) would be infectious. NaN has an exponent of all 1's. applying or to this gives no opportunity for those to be turned off. similarly, there must be an on bit (1 or more) in the fraction. this bit can also not be turned off through or. therefore any number or'd with NaN will yield NaN. &, ^ and ~ allow for those on bits that result in NaN to be turned off and the result may not be NaN (it still might be).
Also, ~~NaN will always be NaN.

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.

Assuming the length of the two strings are constant (they should be), the call to min() in the for loop can be hoisted, and the value for the shorter string (min length) can be stored into a local variable. This reduces the amount of function calls required in running the code.

Another optimization is to use conditional moves.

```
int result = 0;
  for( . . . ){
    result = cvtlower(a[i]) < cvtlower(b[i]) ? -1 : 0;
    result = cvtlower(a[i]) > cvtlower(b[i]) ? 1 : 0;
  }
return result
```

This reduces branch prediction & possible penalties.

The first optimization is probably more impactful as it eliminates many control transfers (function calls), which will cause quadratic complexities for longer strings. The second is a marginal improvement on attempting to reduce misprediction penalties.

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          -jump not below
        movl    %esi, %esi
        movl    (%rdi,%rsi,4), %eax
.L2:
        ret
```

*a i

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

i is unsigned, and will therefore never be less than 0. the first comparison of 0<= i is always true & omitted

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

Although a conditional move is possible, the computation of reading a[i] from memory is much more expensive than returning 0. If the conditional move is not taken, the computation is wasted.

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.

$T_{new} = \frac{1}{(1-.5)+\frac{.5}{2}} = \frac{1}{.5+.25} = \frac{1}{\frac{3}{4}} = \frac{4}{3}$ (1)

$T_{new} = \frac{1}{(1-.3)+\frac{.3}{8}} = \frac{1}{.7+\frac{3}{80}} = \frac{80}{59}$ (2)

The second optimization is faster

$.25 \cdot \frac{3}{80} + .2$ = new speed (both optimizations)

$\frac{1}{.25+\frac{3}{80}+.2}$ = speedup for both optimizations

11

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.

$f = .1f = .00011001$

sign = 0 exp = 01111101 frac = .00011001100...

$r = 0$

X

$r > d$ because d will always be less than the true value of .1

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.

finding out whether or not a cache line was "dirty" would be much slower as it involves a read from memory, then a compare of a whole cache line, instead of looking up a single cached bit. Thus it is slower than write-back

Cache Z's performance in comparison to write-allocate depends on how often a dirty or not check was run. If it is infrequent, cache-z would reduce bus traffic, as each time a write occurred, it wouldn't have to write. However, if the cached address were frequently checked, the efficiency gains would be severely diminished.

One solution is to simply write the cached line to memory each time the line is checked for dirty. This way the comparison is bypassed, and the cache line will not be dirty, so the result is known (not dirty)

12

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?

Assuming starting w/ a cold cache, the instructions must first be written to cache. Cache misses will be cold misses, and be inevitable. Performance is only significantly improved if the kernel is quite a bit bigger and is efficiently exploited in terms of locality.

7

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.

if N is a value such that the resulting memory locations for src and dst do not map to the same sets within the L1 data cache.

32 KiB $2^5 \cdot 2^{10}$ bytes $= 2^{15}$ bytes $\div$ 4 bytes/int $= 2^{13}$ ints held $\div 2 = 2^{12}$ ints per array. 2^6 if N is around 2^6

cache blocks will hold successive rows of the array and there will be many cache-hits.

This avoids thrashing.

6 7

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?

No because applying it increases the number of multiplication operations. This is a costly instruction, and it will be more efficient to reduce the # of multiplies