|  |  |
| --- | --- |
| 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).  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. | 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); ...}  void\* thread1(void\* arg) {  sem\_wait(&mutex);  code <CODE FOR T1>  }  // Initialize sem value to 0  // T1doesP(s)beforeexecuting  void\* thread2(void\* arg) { <CODE FOR T2>  sem\_post(&mutex); // T2 does V(s) after executing code } |

**2.** Pretend ^, &, | and ~ work on floating ­point. 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. Remember 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 of the bitwise operations are infectious on NaNs? Explain.

**ANSWER:** OR is the only infectious bitwise operator. NaNs are identified by an exponent field that is all 1s and a non­zero fractional field. Thus, we are looking for a bitwise operation that will, with certainty, result in an exponent field of all 1s and a non­zero 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 non­NaN 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. 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.

|  |  |
| --- | --- |
| #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; } | **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 becorrect? The source has two comparisons, but the assembler has just one.  **ANSWER:**  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.  **ANSWER**  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.**

d = 0.1 in C (double 0.1) f= 0.1f in C (float value) r= 0.1 (real value)

Binary representation of r: 0.000110011001100110011...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.

r is infinitely repeating while f and d must terminate, so you can’t represent precisely. f and d may be rounded up (to be greater than 1/10) or rounded down (to be less than 1/10).

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

default rounding mode is "round­to­even", 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 rounded­up 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 writeback 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 write­back 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 x86-­64machine 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 L1is fast but very small: why won't performance suffer greatly if your program's kernel doesn't fit into 32 KiB?  **ANSWER:**  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 forN. State any further assumptions you're making. |

**ANSWER:**

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?

**Answer:**

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).

Concurrency:

multiple things at once: parallelism, but better.

Every program gets a process, which can be thought of as a shell for everything your program needs

wherever your stack is is also our process space

heavy-weight full chunk of memory

Threads are inside processes, with their own tiny little stack spaces

I/O Multiplexing - event-based listing. Instead of multiple threads spread across, it waits for an event to happen.

Threaded vs Event-based.

Processeses have massive bunch of space, and threads share this space.

Process vs thread:

threads are lightweight

Threads share a memory space, while processes have their onwn memory, but threads have their own registers

Thread level parallelism (TLP)

Two kinds - either breaking into sections of own program, or hardware built for TLP - like *hyperthreading*. If you have one core, how to break among them?

Race condition!

Threads can act on the same variable. WHat if you want to break up this addition into parallel add? There might be a chance that both threads try to update at the same time. Essentially, threads collide while sharing a resource.

The BSS segment, also known as *uninitialized data*, is usually adjacent to the data segment. The BSS segment contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code. For instance, a variable defined as static int i; would be contained in the BSS segment.

The *.data* segment contains any global or static variables which have a pre-defined value and can be modified. That is any variables that are not defined within a function (and thus can be accessed from anywhere) or are defined in a function but are defined as *static* so they retain their address across subsequent calls. Examples, in C, include:

int val = 3;

char string[] = "Hello World";

The values for these variables are initially stored within the read-only memory (typically within *.text*) and are copied into the *.data* segment during the start-up routine of the program.

Offset is the difference between two addresses, or how far away it is from a certain point.

Alignment is a property of the memory address itself. An address of alignment 4-bytes is positioned such that is lines up within a 4-byte chunk of memory (for faster access by the CPU).

Basically, if mem\_addr % n == 0, mem\_addr is aligned n-bytes

struct

{

    int k;

    char c;

} p;

In this case, c has offset 4-bytes because it is positioned 4 bytes from the beginning of the struct, and is 1-byte aligned because a char is of size 1 byte (alignment is usually (but not always) the size of the data itself).

//

Alignment also affects offsets. Like if you were to define

struct k

{

char c;

int i;

};

C will have an offset of 0, alignment is 1 (size of integral type in bytes)

I will have an offset of 4 (assuming 32-bit), alignment is 4 (size of integral type in bytes)

The struct itself will have an alignment of the largest alignment, so 4.

So say we now make another struct and define it as

struct m

{

 struct k {

   int i;

   char c;

} a, b;

};

This new struct has a size of 13, and an alignment of 4.

a, offset 0, align 4

a.i offset 0, align 4

a.c offset 4, align 1

b, offset 8, align 4

b.c offset 8, align 4

b.i offset 12, align 1

When exponent in range 1 <= e <= 254

result = +- 2(e - 127) \* 1.f         (in base 2)

When e == 0

result = +- 2(e - 127) \* 0.f         (in base 2)

* Infinity
  + e = 255, f = 0
* Zero
  + s = 0 or 1, e = 0, f = 0
* NaN
  + e = 255, f != 0
  + Anything a NaN touches turns into NaN (except boolean)
  + Comparing NaN to a number as boolean result = 0
    - (NaN == 4) // equals 0
    - (NaN == NaN) // also equals 0
* for (i = 0; i < 100; ++i) {
* // Do something with i
* }
* vs
* for (i = 0; i < 100; i += 2) {
* // Do something with i
* // Do something with i + 1
* }

Out-of-Order Processor

1. Fetch the instruction
2. Dispatch instruction to instruction queue
3. Instruction waits in queue until inputs available. Allowed to leave queue before older instructions
4. When inputs available, instruction sent to functional unit and executed
5. Results queued
6. After older instructions have written their results, this instruction get to write its results

The general principle behind **caching** is that for each level *k*, a smaller and faster device at *k* serves as a cache for a larger slower device at *k+1*.

* 1. For example, DRAM is a cache to the disk, and the L1 – L3 caches are caches to DRAM.

At the L1 cache, we often see a distinction between the L1 data and L1 instruction caches. The reasoning for this is twofold: first, as explored in ILP, we can fetch both the instruction and data in parallel. Secondly, it allows us to take advantage of ... locality!

* Cache blocks are generally stored in some format *like:*[set][valid][dirty][tag][-- data --]
  + The set and tag fields indicate the block of data being cached
  + We cache at a coarser granularity than single bytes

The distribution of bits to set and tag can heavily influence the likeliness that we're storing the right things in the cache.  *(Although, this goes down a discussion about associativity that isn't too important for this class!)*

L1 Cache (2-4 cycles)

Between CPU Register File and Main Memory

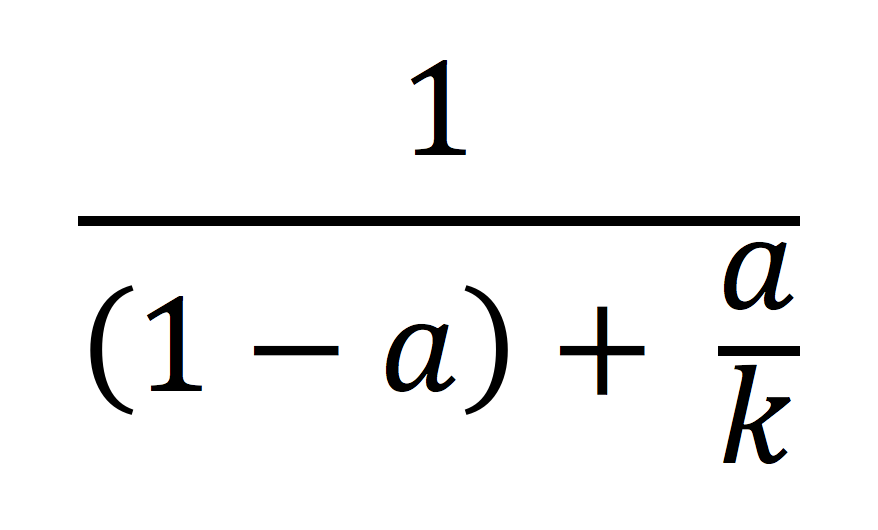
L2 Cache (10 cycles)

Between L1 Cache and Main Memory

L3 Cache (30-40 cycles)

Between L2 Cache and Main Memory

SRAM is faster than DRAM!

* General Caching Organization
  + m bits for addressing: M = 2m **addresses**
  + A cache has S = 2s **cache sets**
  + Each set has E **cache lines**
  + Each line has 1 valid bit, t tag bits, B = 2b bytes per **cache block**
  + Cache Size: C =  B x E x S
* Types of Caches
  + Direct-mapped Cache
    - E = 1, only one line per set
  + Set-associative Cache
    - 1 < E < C/B, more than one line per set
  + Fully Associative Cache
    - E = C/B **AND** S = 1, only one set with C/B lines
* Write Hit Policies:
  + Write Back
    - Write to cache, and set the **Dirty Bit**
    - When data is about to be overwritten by cache eviction, send dirty data to lower level
  + Write Through
    - When write to a cache, write through to the lower level
* Write Miss Policies:
  + Write Allocate
    - Load corresponding block from lower level to cache, then write
    - Takes advantage of space locality, but misses lead to a load
  + No Write Allocate
    - Skip the cache and write directly to the lower level
* We can even split caches up even more!
  + i-cache
    - Only stores instructions
  + d-cache
    - Only stores data
  + Unified cache
    - Stores both instructions and data!
* This allows the processor to read instructions and data at the same time
* Performance of Caches
  + Miss Rate
    - #misses / #references
  + Hit Rate
    - 1 - Miss Rate
  + Hit Time
    - Time to deliver word from cache to CPU
  + Miss Penalty
    - Any additional time due to a cache miss
* Only when the compiler is *sure* the behavior would be the same as the un-optimized case.
* Use Amdahl’s Law!
* a = what fraction of your program is the hog
* k = hog speedup factor
* 
* Speedup =
* Under the **write-through** policy, for each write hit, write to the cache and also to the backing memory.
  + Memory and cache are always synchronized and consistent
  + However, we incur the penalty of writing to memory each time we write to cache, which negates the benefit of the cache for writes.
* Under the **write-back** policy, for each write hit, write only to the cache but mark the cache block as dirty. We write back to memory upon eviction.
  + We benefit immensely from temporally localized programs!
  + Although we're guaranteed that we perform less or equally many writes as with the write-through policy, we run into the issue of unpredictability in our writes. For example, if we're developing a program for a nuclear plant, we might want to be sure of how long a write will take.
  + In the case that we have a program that performs many random reads, we may perform just as many writes but occupy extra space for the dirty bit.