ECE459: Programming for Performance Winter 2013

Lecture 16 — March 7, 2013

Patrick Lam *version 1*

Midterm Solutions

Question 1: Short-Answer.

1. single vs master: single, doesn't have to wait for a particular thread to become free. (surprisingly many people missed this; I thought I'd start the exam with an easy question.)
2. main reasons: context switches and cache misses.
3. I'd expect the 9-thread server to be faster, because responding to requests ought to be I/O-bound.
4. start a new thread (or push to a thread pool) for each of the list elements; can't parallelize list traversal.
5. `volatile` ensures that code reads from memory or writes to memory on each read/write operation; does not optimize them away. Does not prevent re-ordering, impose memory barriers, or protect against races.
6. oops, this was a doozy. Sorry!
   First, weak consistency: run `r2 = x` from T2 first,
   then `x = 1; r1 = y` from T1, and finally `y = 1`.
   Get `x = 1, r1 = 0, r2 = 0, y = 1`.

   Sequential consistency: at end, always have `x = 1, y = 1`. Let's talk about other variables.
   In T1, if you reach `r1 = y`, then you had to execute `x = 1`.
   Maybe you've executed `y = 1` already, maybe you haven't.
   If you have, then `r1 = 1`; eventually get `r1 = 1, r2 = 0`.
   If you haven't, then `r1 = 0, r2 = 1` eventually.
   In neither case is `r1 = r2 = 0` possible as above with weak consistency.
7. No, a race condition is two concurrent accesses to a shared variable, one of which is a write. If you're making a shared variable into a private variable, there's no way to make a new concurrent access to a shared variable—you're not creating a new shared variable.
8. Yes, the behaviour might change. I expect something a bit more detailed, but basically you can put a variable write of a shared variable in a parallel section; changing it to private will change the possible outputs.

9. Simple:

```
int * x = malloc(sizeof(int));
foo(x, x);
```

It was fine to pass in the address of an `int` as well.

10. Some possibilities: the outer loop has a loop-carried dependence, or maybe it only iterates twice and you'd like to extract more parallelism.

Question 2: Zeroing a Register. Source: `http://randomascii.wordpress.com/2012/12/29/the-surprising-subtleties-of-zeroing-a-register/`

- (2a): the `mov` requires a constant value embedded in the machine language, which is going to be more bytes, so imposes more pressure on the instruction cache;
  all else being equal (it is), `xor` is therefore faster.

- (2b): first write down a set of reads and writes for each instruction.

#	W set	R set
1	`eax`	`eax`
2	`ebx`	`eax`
3	`eax`	`eax`
4	`eax`	`eax, ecx`

  Now, read off that set for each pair of instructions; no OOO possible.

  - 1–2: RAW on `eax`
  - 1–3: RAW on `eax`, WAR on `eax`, WAW on `eax`
  - 1–4: RAW on `eax`, WAR on `eax`, WAW on `eax`
  - 2–3: WAR on `eax`
  - 2–4: WAR on `eax`
  - 3–4: RAW on `eax`, WAR on `eax`, WAW on `eax`

- (2c): behind the scenes, register renaming is going on. We'll assume that we can rename the instructions here.

```
add eax, 1
mov ebx, eax
xor edx, edx
add edx, ecx
```

#	W set	R set
1	`eax`	`eax`
2	`ebx`	`eax`
3	`edx`	`edx`
4	`edx`	`ecx, edx`

  Changed dependencies:

  - 1–3: all gone
  - 2–3: all gone
  - 1–4: all gone
  - 2–4: all gone
  - 3–4: RAW on `edx`, WAR on `edx`, WAW on `edx`

  Now we can run (1, 3) or (2, 3) out of order. There are still dependencies between (1, 2) and (3, 4), preventing (1, 4) or (2, 4).

Question 3: Dynamic Scheduling using Pthreads. Once again, this was more complicated than I thought. Sorry. I felt that it was very doable, but it just took more time than you had.

First, let's start by writing something that handles a chunk. Actually, your solution can be much simpler; I overlooked the fact that OpenMP only parallelizes the outer loop, so you don't need to have a chunk, just an index for `i`.

```
#define CHUNK_SIZE 50

typedef struct chunk { int i; int j; } chunk;

void handle_chunk(chunk * c) {
  int i = c->i, j = c->j;
  int count = 0;
  for (; j < 200 && count < CHUNK_SIZE; ++j)
    data[i][j] = calc(i+j);
  for (i++; i < 200 && count < CHUNK_SIZE; ++i) {
    for (j = 0; j < 200 && count < CHUNK_SIZE; ++j) {
      data[i][j] = calc(i + j);
    }
  }
}

// code to pull work off the queue
pthread_mutex_t wq_lock = PTHREAD_MUTEX_INITIALIZER;

chunk * extract_work() {
  chunk * c;

  pthread_mutex_lock(&wq_lock);
  if (!work_queue.is_empty())
    c = work_queue.pop();
  pthread_mutex_unlock(&wq_lock);
  return c;
}

void * worker_thread_main(void * data) {
  while (1) {
    chunk * c = extract_work();
    if (!c) pthread_exit();
    handle_chunk(c);
  }
}
```

```
// populate threads
queue work_queue;

int main() {
  for (int i = 0; i < 200; ++i) {
    for (int j = 0; j < 200; j += CHUNK_SIZE) {
       // relied on 200 % CHUNK_SIZE == 0
      chunk * c = malloc(sizeof(chunk));
      c->i = i; c->j = j;
      work_queue.enqueue(c);
    }
  }

  pthread_t threads[NUM_THREADS];
  for (int i = 0; i < NUM_THREADS; ++i) {
    pthread_create(&threads[i], NULL,
                   worker_thread_main, NULL);
  }
  for (int i = 0; i < NUM_THREADS; ++i) {
    pthread_join(&threads[i], NULL);
  }
}
```

Question 4a: Race Conditions. The race occurs because one thread is potentially reading from, say, `pflag[2]` while the other thread is writing to `pflag[2]`.

You can fix the race condition by wrapping accesses to `pflag[i]` with a lock. The easiest solution is to use a single global lock.

This race is benign because it causes, at worst, extra work; if the function reads `pflag[i]` being 1 when it should be 0, it'll still check `v % i == 0`.

Question 4b: Memory Barriers. I was happy to see that I didn't actually say that these were `pthread` locks, so maybe they don't have memory barriers.

I made this question a bonus mark, since it relies on extra information which I didn't give you. I was generous in marking the question and gave marks for answers that were plausible, even if they weren't actually right.

For more information: `http://erdani.com/publications/DDJ_Jul_Aug_2004_revised.pdf`

(4b (i)) The `new` operation contains three steps: allocate; initialize; set pointer. Thread A might allocate and set the pointer (due to reordering). Thread B could return the un-initialized object.

(4b (ii) is actually really easy: just don't double-check the lock. Protect the singleton with a single lock and be done with it.

I accepted answers which just said to insert fences, if they were at least somewhat detailed (more than just "insert a fence") and showed some understanding of why it would help.