COMPSCI 340
Operating Systems

Assignment 2 - Thread Pools and Dispatch Queues

Worth 8%

final date 25th of September, 2018, 9:30pm

Introduction

A problem with using large numbers of threads in a program is the cost in time and memory of repeatedly creating and destroying threads. One way to minimise the cost of using threads is via one or more thread pools. The idea is to create a number of threads your application may use and store them in a pool allocating a thread to a subtask from this pool on demand and returning the thread to the pool when the task is completed. This way the penalty for creating threads is only paid once and threads stay inactive in the pool until they are associated with a new subtask or until the thread pool is dismantled.

One way such pools can be controlled is based on dispatch queues. These are queues on to which tasks are added. The tasks are then processed either concurrently or serially depending on the type of queue and how the tasks are added to them. In this assignment you have to implement a small subset of such a system. Dispatching is the process of selecting a task to run and starting it running by allocating it a thread from a thread pool. Generally tasks can either be blocks (anonymous functions) or normal functions. In this assignment tasks for dispatching will be C functions.

Basic Idea

Rather than having to create threads and deal with them in order to use the multicore environments of modern computers it makes sense to have a library which does this for us in such a way that efficient use is made of all cores and yet our code doesn't have to change whether there is one processor in a system or sixty-four.

Of course because you are creating this library you will have to create threads and deal with them but it should make programming using this library easier.

Types of Queues

There are two types of queues to implement for this assignment, serial queues and concurrent queues. A serial queue dispatches a task and waits for the task to complete before selecting and dispatching the next task. Obviously serial queues don't provide concurrency between the tasks in the same queue as only one task runs at a time but they do provide concurrency between the tasks in the queue and tasks running on other queues (and the main thread of the application).

Concurrent queues dispatch tasks in the order they are added to the queue but they also allow tasks from the same queue to run concurrently. Each concurrent queue has at least 2 threads associated with it (a small thread pool) and as soon as a thread becomes available because it has finished executing its current task, a new task can be dispatched on it from the front of the queue. In this assignment you can create as many concurrent queues as you like.

All tasks on both serial and concurrent queues are dispatched in a FIFO fashion.

Splitting a Program up into Distinct Tasks

Even though dispatch queues can solve some problems of dealing with concurrency on multiple cores they still require the programmer to split the program into mostly independent tasks. All tasks on a concurrent dispatch queue must be able to operate safely in parallel. If you need to synchronize the activity of several tasks you would normally schedule those tasks on a serial dispatch queue.

Two Types of Dispatching

Regardless of the type of queue it is also possible to add tasks to a dispatch queue either synchronously or asynchronously. If a task is added to a dispatch queue synchronously the call to add the task does not return until the task which was added to the queue has completed. If a task is added to a dispatch queue asynchronously the call to add the task returns almost immediately and the calling thread may run in parallel to the task itself. The normal way of adding a task to a queue is the asynchronous method.

Waiting for tasks

If a task was scheduled asynchronously there may be a time when some thread (e.g. the main thread of the program) needs to wait for all the tasks submitted to a dispatch queue to complete. An obvious example here would be when the program waits until all tasks begun in the program have completed before the program terminates. Waiting for a dispatch queue to complete should work both for serial and concurrent tasks. For this assignment if further tasks are added to a dispatch queue after a thread has waited for that queue to complete those new tasks are ignored. Something to think about: is there a race condition here?

How Many Threads?

In such a system the number of active threads in the system could vary according to the number of cores in the machine and the load on the cores. Using some of the same techniques as were used by batch systems to keep processor usage levels high, more threads can be scheduled to carry out tasks when the load level drops and fewer threads used if the cores are being used at close to 100% load. The load level is not just a local function associated with one particular program, it is a global value determined by all of the work being done on the computer.

In this assignment you do not have to concern yourself with the load level of the cores. In fact you should simply allocate the same number of threads to each concurrent dispatch queue as there are cores or processors in the system. The first task of this assignment is to write a short C program which runs on Linux in the labs and prints out the number of cores on the machine the program is running on.

This program should be called `num_cores.c` and it should be compiled, executed and produce output as shown below:

```
gcc num_cores.c -o num_cores
./num_cores
This machine has 2 cores.
```

Because some architectures use hyper-threading the operating system may see 4 processors when only 2 cores are present. Your program should report as the number of cores the same number as shown by the Gnome System Monitor program.

In the `dispatchQueue.h` file you will find some types defined. You must use these types in your program. You may extend the types by adding extra fields or attributes. You may also add your own types (in fact you will probably have to).

You must use locks or semaphores (or both) to control the concurrency in your solutions. `man -k mutex`, `man -k pthread`, and `man -k sem` will give you more than you need to know.

In the `dispatchQueue.c` file you must implement the following functions. None of these functions return error results because you should print an error message and then exit the program if an error occurs.

```
dispatch_queue_t *dispatch_queue_create(queue_type_t queueType)
```

Creates a dispatch queue, probably setting up any associated threads and a linked list to be used by the added tasks. The `queueType` is either `CONCURRENT` or `SERIAL`.

Returns: A pointer to the created dispatch queue.
Example:

```
dispatch_queue_t *queue;

queue = dispatch_queue_create(CONCURRENT);
```

```
void dispatch_queue_destroy(dispatch_queue_t *queue)
```

Destroys the dispatch queue `queue`. All allocated memory and resources such as semaphores are released and returned.

Example:

```
dispatch_queue_t *queue;

…
dispatch_queue_destroy(queue);
```

```
task_t *task_create(void (* work)(void *), void *param, char* name)
```

Creates a task. `work` is the function to be called when the task is executed, `param` is a pointer to either a structure which holds all of the parameters for the `work` function to execute with or a single parameter which the `work` function uses. If it is a single parameter it must either be a pointer or something which can be cast to or from a pointer. The `name` is a string of up to 63 characters. This is useful for debugging purposes.

Returns: A pointer to the created task.

Example:

```
void do_something(void *param) {
      long value = (long)param;
      printf("The task was passed the value %ld.\n", value);
}


task_t *task;
task = task_create(do_something, (void *)42, "do_something");
```

`void task_destroy(task_t *task)`

Destroys the `task`. Call this function as soon as a task has completed. All memory allocated to the task should be returned.

Example:

```
task_t *task;
…
task_destroy(task);
```

`void dispatch_sync(dispatch_queue_t *queue, task_t *task)`

Sends the `task` to the `queue` (which could be either `CONCURRENT` or `SERIAL`). This function does not return to the calling thread until the `task` has been completed.

Example:

```
dispatch_queue_t *queue;
task_t *task;
…
dispatch_sync(queue, task);
```

`void dispatch_async(dispatch_queue_t *queue, task_t *task)`

Sends the `task` to the `queue` (which could be either `CONCURRENT` or `SERIAL`). This function returns immediately, the `task` will be dispatched sometime in the future.

Example:

```
dispatch_queue_t *queue;
task_t *task;
…
dispatch_async(queue, task);
```

`void dispatch_queue_wait(dispatch_queue_t *queue)`

Waits (blocks) until all tasks on the `queue` have completed. If new tasks are added to the `queue` *after* this is called they are ignored.

Example:

```
dispatch_queue_t *queue;
…
dispatch_queue_wait(queue);
```

This is how you get your marks

There are test files and a *make* file you can use to test your code.

Zip all C source files together into a file called `A2.zip`. Do not include the test files but do include your `num_cores.c`, `dispatchQueue.c` and `dispatchQueue.h` files along with any other files you may have added (most people won't have any more).
Put the answer to the questions into a plain text file or pdf called either `A2Answers.txt` or `A2Answers.pdf`.
Submit the zip file and the answers to the questions using the Canvas submission system before 9:30pm on Tuesday the 25th of September.

The work you submit must be your own. Refer to the University's academic honesty and plagiarism information https://www.auckland.ac.nz/en/students/forms-policies-and-guidelines/student-policies-and-guidelines/academic-integrity-copyright.html.

1. `num_cores`

   prints the correct number of cores for the machine it is running on.
   [1 mark]

2. `test1`

   2.1
   works correctly
   [2 marks]
   Produces:

   ```
   test1 running
   Safely dispatched
   ```

   2.2
   `dispatch_queue_destroy` returns the allocated resources
   [1 mark]

   2.3
   `task_destroy` is called by the implementation and releases memory
   [1 mark]

3. `test2`

   works correctly – only get the mark if `test1` worked correctly
   [1 mark]
   Produces:

   ```
   Safely dispatched
   ```

4. `test3`

   works correctly
   [2 marks]
   Produces:

   ```
   Safely dispatched
   test3 running
   ```

5. `test4`

5.1

works correctly

[2 marks]

Produces something like:

```
Safely dispatched
task "A"
task "B"
task "C"
task "D"
task "E"
task "F"
task "G"
task "H"
task "I"
task "J"
The counter is 4939859698
```

The order of the tasks may vary. The counter value will also vary but it should almost never be 10,000,000,000.

5.2

Utilizes all cores

[2 marks]

Open system monitor before you run it and check that all CPUs go to 100%.

6. `test5`

6.1

works correctly

[2 marks]

Produces:

```
Safely dispatched
task "A"
task "B"
task "C"
task "D"
task "E"
task "F"
task "G"
task "H"
task "I"
task "J"
The counter is 10000000000
```

The tasks must be in this order and the counter must always equal 10,000,000,000.

6.2

Mostly utilizes only one core.

[1 mark]

Check by observing system monitor.

7.

Implementation marks (applied if at least 2 of the previous tests passed)

7.1
Good identifiers [1 mark]
7.2
Consistently good indentation [1 mark]
7.3
Useful comments [1 mark]
7.4
No busy waits [1 mark]
7.5
No use of `sleep` to make synchronization work. [1 mark]

8.

The marker will use an unspecified test - using a combination of the implemented functions. [5 marks]

9.

Question:

Explain how your implementation dispatches tasks from the dispatch queues. You must describe how dispatching is done for both serial and concurrent dispatch queues. Give code snippets from your solution as part of the explanation.
[5 marks]
2 marks for clarity (can the marker easily understand what you are saying).
2 marks for making sense.
1 mark for including code snippets in the explanation.