THE UNIVERSITY OF AUCKLAND
NEW ZEALAND

COMPSCI 340 & SOFTENG 370
Operating Systems

Assignment 2 - Thread Pools and Dispatch Queues

Worth 8%

final dates 21st and 25th of September, 2018, 9:30pm

JUMP to the bottom of page 5 for marking instructions

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).

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);
```

Extra for SOFTENG 370 students

`void dispatch_for(dispatch_queue_t *queue, long number, void (*work)(long))`

Executes the `work` function `number` of times (in parallel if the `queue` is `CONCURRENT`). Each iteration of the `work` function is passed an integer from `0` to `number-1`. The `dispatch_for` function does not return until all iterations of the `work` function have completed.

Example:

```
void do_loop(long value) {
      printf("The task was passed the value %ld.\n", value);
}
dispatch_queue_t *queue;
…
dispatch_for(queue, 100, do_loop);

/* This is sort of equivalent to:
for (long i = 0; i < 100; i++)
      do_loop(i);
Except the do_loop calls can be done in parallel.
*/
```

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 (or answers if you are a SOFTENG 370 student) 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 Monday the 17th of September for COMPSCI 340 and Friday the 21st of September for SOFTENG 370.

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.

Text in bold added for the marker's version. You must run submissions on 64-bit Linux in the labs (or 64-bit Linux on your own machine).
Unzip the student's source files into a directory empty except for the Makefile.
Type "make". Not all tests will compile for all students. You can briefly look to see if it is something simple to make a test compile, but don't spend more than a few minutes.

In the mark sheet fill in the marks and total - I'll check the additions automatically.

If there is a reason, apart from not attempted, that you don't give full marks on a section, please add a helpful comment.

1. `num_cores` - **run with "./num_cores"**

   prints the correct number of cores for the machine it is running on.
   Should compile with "gcc num_cores.c -o num_cores", then run with "./num_cores"
   Check the code - make sure the number of cores is not hard coded.
   [1 mark]

Students who used named semaphores may have problems with semaphores hanging around between runs. If you see semaphore errors appear you can rerun the program. If they have reused my example code, running the first time should clear semaphores for a second attempt.

2. `test1` - **run with "./test1"**

   2.1
   works correctly
   [2 marks]
   Produces:

   ```
   test1 running
   Safely dispatched
   ```

   To get both marks *the output must appear in this order because even though the task sleeps for a second before doing anything it was dispatched synchronously.*

   2.2
   `dispatch_queue_destroy` returns the allocated resources
   Must at least call free and destroy or close semaphores.
   [1 mark]
   Look at the `dispatch_queue_t` type and check that `dispatch_queue_destroy` releases semaphores and threads and frees memory.

   2.3
   `task_destroy` is called by the implementation and releases memory
   Just search for it. Should at least be called after a task has run.
   [1 mark]

3. `test2` - **run with "./test2"**

   works correctly
   [1 mark]
   Produces:

   ```
   Safely dispatched
   ```

   The output from the task must not appear as the main thread should have finished by then. If errors occur after the message appears that is probably ok because they may be caused by releasing the threads associated with the dispatch queue while one of them is still working.

4. `test3` - **run with "./test3"**

   works correctly
   [2 marks]
   Produces:

   ```
   Safely dispatched
   ```

```
test3 running
```
"Safely dispatched" must occur before "test3 running".

5. `test4` - **run with "./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. If it is run it again.

5.2
Utilizes all cores
[2 marks]
Open system monitor before you run it and check that all CPUs go to 100%. ***2 marks if all CPUs stay at 100% for most of the time (except the end). 1 mark if they do for some of the time.***

6. `test5` - **run with "./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. Or 1,410,065,408 if run on a 32 bit machine.

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. **A quick scan is good enough. Any waiting should be with sem_wait or pthread_mutex_lock etc.** [1 mark]
7.5
No use of `sleep` to make synchronization work. **Search for "sleep".** [1 mark]

8. **- run with "./test6"**

The marker will use an unspecified test - using a combination of the implemented functions.
Should produce output like:

```
Serial tasks safely dispatched. counter: 0
task "Serial A"
task "Concurrent A"
task "Serial B"
task "Concurrent B"
task "Serial C"
Concurrent task safely dispatched and completed. counter:
1761520771
task "Serial D"
task "Serial E"
task "Serial F"
task "Serial G"
task "Serial H"
task "Serial I"
task "Serial J"
The final counter is 9248488365
```

The counter values will be different each time of course.
Things to look for: It is likely that Serial and Concurrent will be interleaved at first, but the Concurrent tasks should complete before the Serial tasks finish.
2 of the marks depend on the system monitor. You should see two cores used at first, but after the concurrent tasks have gone it should go down to one core. This may be hard to spot as there is no control over which cores are used.
[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.

The next two parts are extra for SOFTENG 370 students (COMPSCI 340 students get no additional marks for doing these).

10.

The marker will use a test program similar to `testFor` but with more intensive computation.
works correctly
Output should be something like (order may be different):

```
The number is 999999999
The number is 1000000000
The number is 999999998
The number is 999999997
The number is 999999996
The number is 999999995
The number is 999999994
The number is 999999993
The number is 999999992
The number is 999999991
```

[2 marks]
All cores should be used when running.

11.

Question:

Using the *time* command, time running *test4* and *test5*. Give the results and discuss them. Is there anything unusual? Explain what you observe.
[3 marks]
1 mark for the results.
1 mark for comments.
1 mark for explanations.
Their explanation should make sense according to their results. In general they should find that there is not as much speed up as they may have expected (on my machine the concurrent times were actually slower). Plausible reasons include contention for the shared variable, in particular cache line bounce and the overheads associated with semaphores and threads.
If their results do show a good speed up then they should talk about the benefits of using all cores etc.

If the assignment was submitted late (I think it will be highlighted in pink) take off a percentage as indicated in the following table. There is no penalty if submitted between 9:30pm and 11:59pm on the due date so please check that.

Day submitted	SoftEng 370	CompSci 340
22nd of September	-5%	
23rd of September	-10%	
24th of September	-15%	
26th of September		-5%
27th of September		-10%
28th of September		-15%