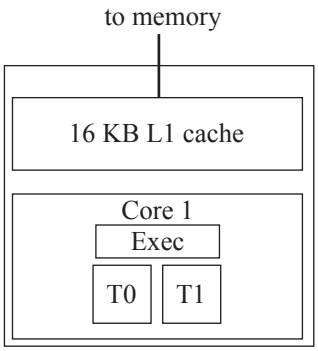
CMU 15-418/618: Parallel Computer Architecture and Programming
Practice Exercise 1

A Task Queue on a Multi-Core, Multi-Threaded CPU

The figure below shows a simple single-core CPU with an 16 KB L1 cache and execution contexts for up to two threads of control. Core 1 executes threads assigned to contexts T0-T1 in an interleaved fashion by switching the active thread only on a memory stall); **Memory bandwidth is infinitely high in this system, but memory latency is 60 clocks. A cache hit is only 1 cycle. A cache line is 4 bytes. The cache implements a least-recently used (LRU) replacement policy.**



You are implementing a task queue for a system with this CPU. The task queue is responsible for executing large batches of independent tasks that are created as a part of a bulk launch (much like how an ISPC task `launch` creates many independent tasks). You implement your task system using a pool of worker threads, all of which are spawned at program launch. When tasks are added to the task queue, the worker thrads grab the next task in the queue by atomically incrementing a shared counter `next_task_id`. Pseudocode for the execution of a worker thread is shown below.

```
mutex queue_lock;
int   next_task_id;              // set to zero at time of bulk task launch
int   total_tasks;               // set to total number of tasks at time of bulk task launch
int*  task_args[MAX_NUM_TASKS];  // initialized elsewhere

while (1) {

   int my_task_id;

   LOCK(queue_lock);
   my_task_id = next_task_id++;
   UNLOCK(queue_lock);

   if (my_task_id < total_tasks)
       TASK_A(my_task_id, task_args[my_task_id]);
   else
       break;
}
```

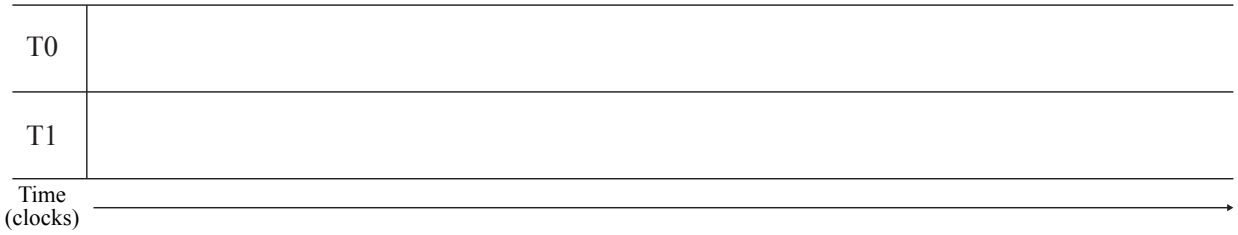
A. (3 pts) Consider one possible implementation of TASK_A from the code on the previous page:

```
function TASK_A(int task_id, int* X) {
   for (int i=0; i<1000; i++) {
      for (int j=0; j<1536; j++) {
         load X[j]   // assume this is a cold miss when i=0
         // ... 20 non-memory instructions using X
      }
   }
}
```

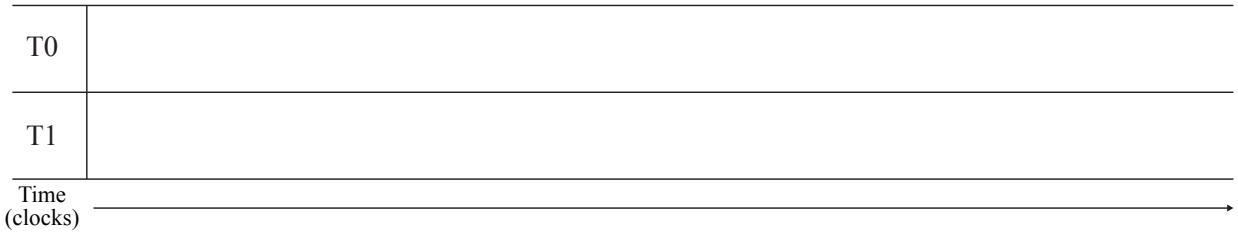
The inner loop of TASK_A scans over 6 KB of elements of array X, performing 20 arithmetic instructions after each load. This process is repeated over the same data 1000 times. **Assume there are no other significant memory instructions in the program and that each task works on a completely different input array X (there is no sharing of data across tasks). Remember the cache is 16 KB, a cache line is 4 bytes, and the cache implements a LRU replacement policy. Assume the CPU performs no prefetching.**

In order to process a bulk launch of TASK_A, you create two worker threads, WT0 and WT1, and assign them to CPU execution contexts T0 and T1. Do you expect the program to execute *substantially faster* using the two-thread worker pool than if only one worker thread was used? Why or why not?

(Careful: please consider the program's execution behavior on average over the entire program's execution ("steady state" behavior). Past students have been tricked by only thinking about the behavior of the first loop iteration of the first task.) It may be helpful to draw when threads are running and stalled waiting for a load on the diagram below.

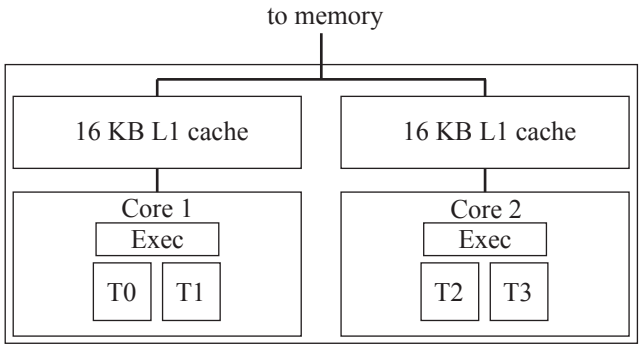


B. (3 pts) **Now consider the case where the L1 cache size is changed to 4 KB. (Keep in mind different tasks operate on different data.)** When running the program from part A on this new machine, do you expect your two-thread worker pool to execute the program *substantially faster* than a one thread pool? If so, please calculate how much faster (your answer need not be exact, a back-of-the envelop calculation is fine). If not, explain why.



C. (3 pts) **Now consider the case where the L1 cache size is changed to 8 KB.** Assuming you cannot change the implementation of `TASK_A` how should your system schedule tasks to improve program performance by nearly a factor of two over the two-worker pool approach? Why does this improve performance?

Now consider the case where the task system is running programs on a dual-core processor. Each core is two-way multi-threaded, so there are a total of four execution contexts (T0-T3). Each core has a 16 KB cache.



D. (3 pts) If you maintain your two-worker thread implementation of the task system as discussed in prior questions, to which execution contexts do you assign the two worker threads WT0 and WT1? Why? Given your assignment, how much better performance do you expect than if your worker pool contained only one thread?

E. (3 pts) Imagine you are requested to design a tasking system that maximizes the dual-core processor's overall *throughput* (in terms of tasks completed) when running bulk launches of `TASK_A`. How many worker threads do you create? Why?

F. (3 pts) Imagine you are requested to design a tasking system that minimizes start-to-end latency of any one single task in a bulk launch of `TASK_A`. How many worker threads do you create? Why?

G. (2 pts) Imagine you are requested to design a tasking system that minimizes start-to-end latency of an *entire bulk launch* of many instances of `TASK_A`. How many worker threads do you create? Why?