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 this system. 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 with a fixed pool of worker threads that grab the next task in the queue by incrementing a shared counter next_task_id. Pseudocode for each worker thread is given below.

```c
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 the stub `TASK_FUNCTION` from the code on the previous page: `TASK_A`.

```c
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`. Remember the cache is 16 KB, a cache line is 4 bytes, and the cache implements a LRU replacement policy. Assume no prefetching.**

In order to process a bulk launch of `TASK_A`, you create two worker threads, WT0 and WT1, and assign them to 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 STEADY STATE execution behavior, not just the behavior in the first (“startup”) iteration of the first task.)

There is **not** a substantial difference in performance because both the one and two thread configurations will run at near 100% utilization of the processor and thus operate at about the same speed. The reason for this is that there are essentially no cache misses in this scenario since nearly all memory accesses are serviced by the cache. As a result, there is no latency to hide and thus no benefit from hardware multi-threading in this situation.
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.

The two-thread configuration will execute about 2 times faster. Now all memory accesses are cache misses and incur a 60 cycle latency. The one thread implementation proceeds with 60 cycles of memory stall, followed by 20 cycles of math, then 60 cycles of memory stall, followed by 20 cycles of more math, etc. Therefore, it results in a core utilization of 25%.

The two-thread configuration is able overlap 20 of the 60 stall cycles with execution of arithmetic operations from the other thread. As a result, it achieves 50% utilization and runs *twice as fast*. An interesting note is that a processor supporting four interleaved hardware threads would be required to reach the 100% utilization that was realized in part A.

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, when running one thread, the thread’s working set fits in the cache. The thread takes essentially no cache misses and runs at 100% utilization. The two thread configuration requires 12KB of cache (it only has 8KB) and thus, just like in part B, all memory accesses are cache misses. We know from part B that the two-thread configuration will realize 50% core utilization, and thus the *one thread configuration is twice as fast.*
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?

You want to schedule the worker threads to execution contexts T0 and T2 (on different cores). This assignment allows you to DOUBLE performance over the one-thread configuration since twice as many execution resources could be used. (Two instructions per clock can be executed, not just one!) Note that caching effects were not relevant on this problem since with a 16KB cache like in part A, the working set for two threads can fit in the cache on a single core—although spreading the worker threads out to different cores essentially doubles the effective cache size for each thread, the program does not take advantage of this additional capacity.
E. (3 pts) Imagine you are requested to design a tasking system that maximizes the dual-core processor’s overall \textit{throughput} (in terms of tasks completed) when running bulk launches of many instances of TASK\_A. How many worker threads do you create? Why?

\textit{Run four threads. This configuration maximizes latency hiding ability, utilizes all execution resources, and the working set of the threads on a core does not exceed the capacity of the core’s cache.}

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?

\textit{Run only one worker so it gets all of the core’s processing and cache resources. (It doesn’t have to share with another thread.) We also accepted the answer of running only one worker thread per core.}

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?

\textit{Minimizing the time to compute \textbf{all the work} is akin to maximizing overall throughput of the computation. Use four threads just like in part E, for the same reasons.}