Bogdan Vancea , Shanmuganathan Puspanantha 29.11.2013

Supporting Parallelism in Operating Systems & Programming Languages

Exercise 4: Memory Management

Intermediate Report

1. Per thread object free lists

1.1 Allocator design

Our implementation of the run time system uses the following objects:

- scheduler_t - object that models the threads which schedule the tasks that need to be ran. One scheduler_t object is allocated for each thread, at the start of the run time system.
- queue_t - object that models the thread execution queues. Each thread is allocated a queue_t object at the start of the run time system.
- task_t - object that models the task which needs to be executed. Tasks are created during the **sync()** method and free by the last thread that accesses them. Because in our run time system we assume a great parallel slackness, we know that a lot of task objects will be created and freed during the lifetime of the program.

Based on the facts presented above, we propose keeping per-thread free-lists for the *task_t* objects. Our system will contain the following free lists:

- each thread will maintain it's own list of free *task_t* objects. We will implement this using an array of lists of *task_t* objects. We maintain an integer *id* for each thread which will be used to access the thread's task free list.
- a global free lists containing task_t objects which can be used by each thread if necessary.

The allocation and reclamation of the task_t objects will be done through the use of the following functions:

```
task_t* task_alloc();
void task_free(task_t* task);
```

The allocation process is the following:

- when a call to *task_alloc*() is made, the calling thread will retrieve a *task_t* object from its free list.
- If the free list is empty, the thread will attempt to retrieve a *task_t* object from the global free list.
- If the global free list is also empty, a new task_t object will be allocated using malloc().

The reclamation process will be implemented as follows:

- when a call to *task_free*(*task_t** task) is made, the calling thread will place the task object received as argument in its free list. To prevent the free list from becoming very large, we will also set a threshold $threshold_{local}$ for the size of each thread's free list. If after an insertion the thread list goes over the aforementioned threshold, a fraction of the task objects will be move to the global free list. We believe that a good starting value for the fraction of returned tasks is half of the thread's free list.

- if during the move of task objects from a thread's free list to the global free list the size of the global free list surpassed a threshold $threshold_{global}$, a portion of the global free list will be freed. We believe a good starting value of this potion would be 1/n, where n is the number of threads in the run time system.

1.2 Synchronization scheme

We will use the following synchronization scheme for the operations described:

- no synchronization for operations involving the calling thread and it's own free list. The calling thread will be the only thread that will access the free list and no synchronization is necessary.
- a global lock for operations involving the calling thread and the global free list. It is possible that more than one thread can access the global free list in the same time, either to retrieve *task_t* objects from it or to add such objects to it. To preserve the consistency of this global list we will use a global lock to protect the list during the task addition and removal operations. This approach could make the global list become a bottleneck if there are many thread attempting to mutate the global list in the same time. However, we assume that in most of the cases a thread will only interact with it's own free list.

1.3 Unused memory management

As discussed in the previous paragraph, our approach maintains a bound of the unused memory. Considering the thresholds described above, the maximum amount of unused memory in our system is:

$$n * threshold_{local} + threshold_{global}$$

To reiterate, the amount of unused memory used by a single thread will never surpass $threshold_{local}$, while the amount of unused memory in the global free list will never be more than $threshold_{global}$.