Bogdan Vancea, Shanmuganathan Puspanantha 6.12.2013

Supporting Parallelism in Operating Systems & Programming Languages

Exercise 4: Memory Management

Final 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:

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

In the above equation, n is the number of threads in the system. 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 $threshold_{global}$.

1.4 Performance comparison with malloc, tcmalloc

For the performance comparison, we used the recursive Fibonacci implementation as a test program. We measured the execution time for computing the 12th Fibonacci number. The configurations we have used are:

- our per-thread free list implementation
- a naive malloc implementation with no free list management
- a naive method implementation linked with tcmalloc

Both in the case of the malloc and tcmalloc configuration, no per-thread free list is kept. Each task allocation allocates a task descriptor, a ucontext descriptor and a stack for the new task. For these configurations, in case of a free, the aforementioned structures are free with calls to malloc.
For each of the aforementioned configuration we ran tests with 1, 2 and 4 scheduling threads. Each of the spawned threads is a pthread affined to a dedicated processor core.

Configuration	**1 thread**		**2 threads**		**4 threads**	
	Mean	Std. dev.	Mean	Std. dev.	Mean	Std. dev.
Our system	1812.6	46	1189.8	351	2096.6	691
Tcmalloc	1613	456	1938	1121	1885	419
Malloc	3651	1061	3429.8	841	4403.4	101

The results we have obtained are presented in the table above. Figure 1 also plots the average execution times for the configurations described.

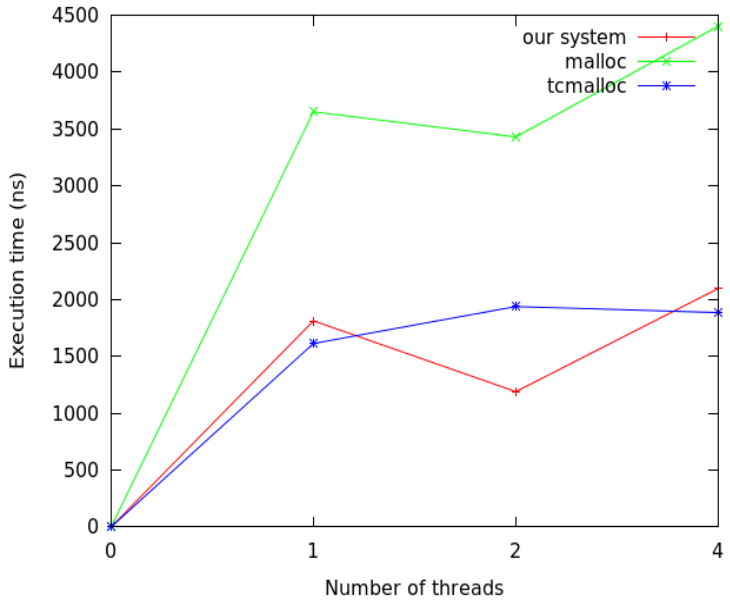

Figure 1. Comparison between the response times of different allocators

From the plot above we can see that the per-thread free-list model described and implemented by us provides better performance than the standard malloc provided by glibc. That is because in the malloc implementation, the objects are not cached at all. Also, we noticed that our system provides comparable performance with the tcmalloc implementation. While in the tcmalloc

implementation we do not use free-lists, tcmalloc implements caching of allocated block internally. While the response times seem similar, tcmalloc seems to offer a lower standard deviation for the response times.

2. NUMA Allocation

The most important consideration that needs to be taken into account for a NUMA architecture is the fact the local memory access will be faster than the remote memory access. Therefore, the design goal would be to make each worker thread access memory from its local node in most, if not all of the cases.

The factors that need to be taken in consideration are:

- worker threads steal tasks from each other. Because of this, worker threads will access memory allocated by other worker threads.
- tasks are cached inside free lists before being freed. The global free list is the second manner in which a worker thread can access memory allocated by another worker thread.

In a context where threads access only memory accessed by themselves, it would make sense to use a local allocation policy, in which each thread allocates memory on its own local node.

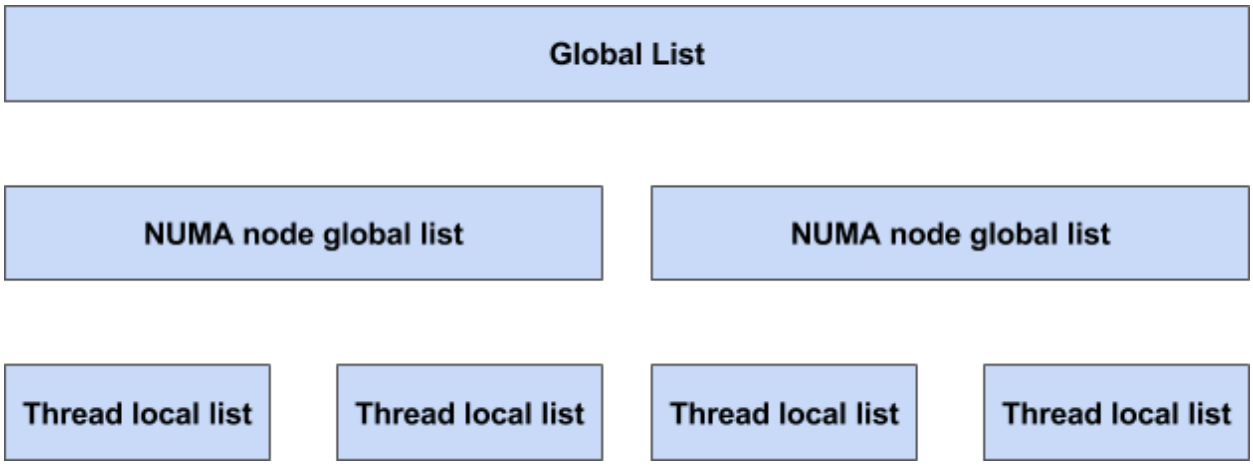

Figure 2. NUMA free list design

To reduce the instances when a worker thread has to access remote memory, the following steps should be taken:

1. Each worker thread should be affined to a NUMA node, to prevent it from being moved to a different CPU.
2. Each task descriptor should be annotated with the NUMA node it was allocated on.

3. The free lists which are local to each worker thread should be allocated on the worker thread's corresponding NUMA nodes.
4. The global free list should be split into several global lists, one per NUMA node. Therefore, when a thread local list is empty, a task can be taken from the NUMA global list, which contains only tasks allocated in that NUMA node. If the NUMA node global list is empty, a task can be taken from the global list, which contains tasks allocated from more NUMA nodes. When a task is retrieved from this global list, the calling thread should try to retrieve a task that belongs to it's NUMA node. If no such task is found, a task from another node can be retrieved, or, if the global list is empty, a new task is allocated through an OS specific API call. This memory for this new task should be allocated from the calling thread's NUMA node.
5. Like in the previous design, when one of the free lists grows over a certain size, it should return some tasks to a list right above it in the hierarchy. In case of the global list, memory should be freed.
6. The stealing algorithm can also be modified so that a worker thread will first try to steal from another thread in it's NUMA node, and only if that fails, it should try to steal from other threads.

3. Stack Size

A static task stack size can cause problems in the context of a parallel task execution. First of all, the stack size should be large enough to provide enough space for the most memory consuming tasks. However, in this case, tasks which do not require very much memory will be allocated the same amount of memory as the memory intensive tasks. This problem would limit the number of tasks that could exists in memory in the same time.

A simple way to dynamically allocate stacks for parallel tasks would be to make some modifications to the compiler to determine to required stack size. This can be done by performing an extra analysis of the code before the actual compilation. In this extra step, the functions which are defined as parallel tasks can be analyzed to determine the necessary stack size. The stack size needed by these tasks will include the local variables defined in the function as well as the necessary function calls. However, the calls to other task functions should not be included in this stack size, as each parallel task will have a different task.