Bogdan Vancea , Shanmuganathan Puspanantha 22.11.2013

Supporting Parallelism in Operating Systems & Programming Languages

Exercise 3: Synchronization

Final Report

1. Multi-threaded task scheduling

1.1 Design overview

The run time system we propose is based on the Cilk runtime system model and allows the user to spawn new tasks and wait for their completion. Our system uses a fixed number of threads for executing the tasks created. We plan to use the pthread library for the threads. Therefore, when a new task is created, it is assigned to one of the threads used by the system. Because the number of tasks is usually larger than the number of threads, each running thread maintains a queue of tasks it has to execute.

The system will also implement work stealing at the thread level. That is, when a thread's running queue is empty, it will choose a "victim" thread and execute one of the tasks assigned to that thread.

1.2 Synchronization scheme

In our implementation we use the following synchronization methods:

- we use a lock to protect the access in each of the queues task queues.
- we use a counter on each task descriptor to keep track of how many child tasks have been created. When a child is spawned, the counter in increased, when the child returns, the counter is decreased. We use a separate lock associated with each task descriptor to protect the access to the counter variables.
- we use a global condition variable to make the "main" execution thread wait until all of the tasks have been executed. When a task returns, if it has no parent, we signal the condition variable. This conditional variable is also protected using a single, global lock.

1.3 System API

The API we will use is similar to what we used in the previous exercise:

```
void task_init(int nr_threads);
void task_end();
task_t* task_spawn(void* fct_ptr, void* arguments, void *return_val);
int task_sync(task_t** execution_context, int count);
int task_return();
#define RETURN(value, type) task_t* current_task = task_current();          \
                              *((type*)current_task->result)= *value;        \
                                        task_return()
```

```
task_t* task_current();
```

The modifications we will make are the following:

- instead of maintaining a single queue for all the task, the system will maintain a number of task queues, each associated with a pthread.
- each pthread will have it's own scheduler context. All push/pop/steal operations on the task deques are made from these contexts.
- we have replaced the task_deinit() function with a task_end() function.

1.4 Implementation

Before going into detail about the implementation of the API functions, we will briefly describe the structure of the task and thread descriptors.

Task descriptors contain:

- an ucontext_t object describing its context.
- a status variables. A task can have the following statuses: STARTED, COMPLETED.
- a pointer to the function that needs to be executed.
- a pointer to an argument structure that needs to be passed to the aforementioned function.
- a pointer to the memory address where there the result should be stored.
- a counter for the not yet completed children task and a lock used to protect the access to this function.
- a pointer to the parent task.

Thread descriptors contain:

- an ucontext_t object describing its context.
- an action object describing the current state. Possible actions are: ADD_TASK, YIELD, RETURN_TASK and IDLE.
- a lock protecting the access to the action object.
- an integer identifier
- a pointer to the currently executing task
- a queue of tasks ready to execute.

Each created pthread then proceeds to loop through a function and execute an action based on the action object value. If the current action is ADD_TASK, it will add the task pointed by the new_task pointer and add it to the ready queue. If the current action is YIELD, it will retrieve a task for the ready queue and execute it. If the current action is RETURN_TASK it will set the status of its currently executing task to COMPLETED and decrement the children pointer of the parent. If after the decrement the parent's counter is 0, it will add the parent task to it's ready queue. If the action object is IDLE, which happens only when its ready queue is empty, it will attempt to steal a task from one of the other threads.

Furthermore, we use the pthread_getspecific/pthread_setspecific functions to maintain the id of the thread descriptor for each thread. This id indexes the thread descriptor in a global array of thread descriptors.

- **task_init(int nr_threads) -** This function initializes all resources for the threads used to execute the tasks and then creates the threads.
- **task_spawn(void* fct_ptr, void* arguments, void *return_val)** - This function constructs a new task object using its arguments and creates a new ucontext_t for it. It then modifies the pointer for the new_task for the currently executing thread and sets the action variable to ADD_TASK. This access is synchronized using target thread descriptors lock. If this function is called from the "main" context, a random thread is chosen from the pool.
- **task_sync(task_t** execution_context, int count) -** During the sync operation, the current execution is stopped until all tasks received as arguments are completed. This is done by checking the status of all these tasks in a loop. If all tasks are completed, the resources associated to them are freed. If some tasks have not finished, the following thing happens:
  - if we are in the "main" context, the main thread calls wait on a condition variable
  - if we are in the context of a pthread, a yield operation is done. This means that the current thread jumps in its scheduler context and will execute a different task.
- **task_return() -** During the return operation, the current task writes the value it needs to return to the return address specified in the spawn instruction and then yields to the scheduler. The scheduler decrements the children counter of its parent and checks if that value is 0. If that is the case, the parent task is moved on the thread's ready queue.

1.5 Implementation bottlenecks

We have benchmarked our implementation using the Linux *perf* tool. From the report on a Fibonacci function, with 4 threads on a 4 core CPU, we have obtained the following numbers:

Function	Percent
pthread_getspecific() - called to obtain the thread descriptor of the current thread	44.43%
task_sync()	21.48%
inside_main() -- boolean function that indicates wether the curren context is the main context.	17.44%
kernel.kallsymms -- system calls	8.29%

The large percent corresponding to pthread_getspecific is caused by the fact that the Fibonacci function does not require a lot of computation and thus the overall switching takes longer.

Considering the current implementation, we consider that the bottlenecks for the implementation are:

- the work stealing implementation. We only use simple locks to protect the ready queues. Furthermore, the threads will always try to steal work from another thread when it has no more threads on the task queue, even if there are no other task on the queues of the other threads. When using a large number of threads, this contention will take a lot of CPU and will make it hard for new tasks to start executing. Ideally, if there is no work to be stolen, the threads should block in some way.
- the ucontext jumps. Due to the signal handling system call in setcontext, each time a task is started or finished, our system suffers from the performance penalty of a system call.
- the locks. Even if all the previous issues are resolved, the needs for synchronization will eventually become the bottleneck. Considering a large number of threads, all of them trying to acquire locks (even if not the same lock) will cause a large contention on the memory bus.

2. Locking strategy

Normal mutexes (mutexes) are preferable over adaptive mutexes in case of single-processor systems. In such a system, in case a thread attempts to take a mutex and fails because the mutex is already taken by a different thread, it does not make sense to spin and check for the lock value, as it will not change. Therefore, spinning even for a short period of time will be a waste of time.

3. Transactional Memory

Transactional memory is suitable to applications that:

- do not have any I/O operations. I/O operations cannot be undone if the transaction needs to be rolled back.
- there are a lot of shared variables but the frequency at which they are changed is low. In this manner, most of the transactions will commit successfully and there will be few cases where a roll-back is necessary. In a situation where we expect that these shared variables will change frequently, we can also expect that most transactions will need to be rolled-back.
- require composable atomic operations. There are situations when the operations that need to be performed atomically are more complex and could be composed of simpler atomic operations. For example, if an item needs to be removed from a list and then inserted into a different list and the whole operations should appear atomic. In this case, it is possible to implement the delete and insert operations to be thread-safe using locks, however, the whole operation would require a large lock, which would prove to be slow in the cases when other threads would only like to read values from only one of the lists. However, since in the most of the cases no thread would read a value between the delete

and the insert, one could wrap these two operations in a transaction. In this case, the penalty of a roll-back is only paid in those rare cases.