Supporting Parallelism in Operating Systems & Programming Languages

Exercise 3: Synchronization

Intermediate Report

1. Multi-threaded task scheduling

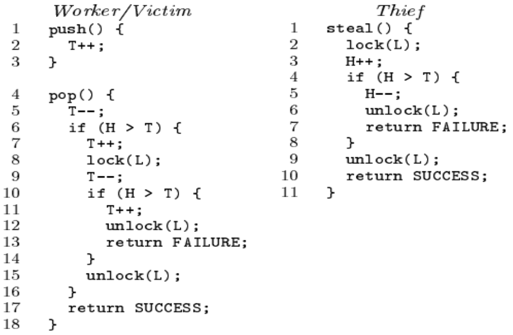
1.1 Design overview

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. Like in the case of Cilk, we choose to make the task stealing be the slower operation and use deques for managing the task queues.

1.2 Synchronization scheme

The synchronization scheme we used is based on the the THE protocol used by Cilk. Therefore, each thread maintains a deque of ready tasks. The owner of the queue interacts only with the bottom of the queue, by pushing and popping new tasks to that queue. When a thread wants to steal a task from another queue, it will only interact with the tail of the queue. This approach does not eliminate the need for synchronization, as it is possible that both the owner of the queue and a thief to try to execute the same task. Therefore, we choose to protect each individual dequeue with a lock.



Worker/Victim

```
push() {
  T++;
}

pop() {
  T--;
  if (H > T) {
    T++;
    lock(L);
    T--;
    if (H > T) {
      T++;
      unlock(L);
      return FAILURE;
    }
    unlock(L);
  }
  return SUCCESS;
}
```

Thief

```
steal() {
  lock(L);
  H++;
  if (H > T) {
    H--;
    unlock(L);
    return FAILURE;
  }
  unlock(L);
  return SUCCESS;
}
```

Fig 1. Cilk work stealing strategy

The approach we plan to use for the push/pop/steal operations will be based on the simplified THE protocol described in the Cilk paper.

1.3 System API

The API we will use is the same one we have used in the last exercise:

```
void task_init(int nr_threads);
void task_deinit();
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. If the aforementioned operations would be made from the contexts of the tasks executed, strange behaviour might appear if the current task would be stolen by another thread right when performing a push/pop on the queue.

1.4 Implementation

This section describes the implementation modifications for the APi presented at the previous point.

- **task_init(int nr_threads) -** This function needs to initialize both the scheduler contexts and the task queues for all the threads created by the run time system.
- **task_deinit() -** Terminates the threads used for scheduling and frees any memory used by the run time system.
- **task_spawn()** - When a new task is created, control is first passed to a scheduler which will choose a thread to which it will pass the task. If the spawn happens in the context of a thread in the run time system, the task will be assigned to the deque of the thread which was execution the parent task. If the spawn happened in the context of the main program thread, it will be assigned to a thread at random. In all of these cases, it is the new task (child) that will be executed before the parent task. As a note, the fact that we use a random strategy when assigning the first task to a random thread and when selecting the victim for work stealing could lead to an unbalanced load for the threads of the system. However, work stealing will balance the workloads if we assume a high parallel slack for the programs executed.

- **task_sync() -** During the sync operation, the parent will check if all the child tasks have executed successfully. If that is the case, the child tasks are cleaned up and the parent resumes the executes. However, if that is not the case and there are some child tasks which are not done, the parent will be marked as sleeping and then it will yield to the scheduler. Once the last child task has finished the execution**,** the parent is woken up by the scheduler and it continues execution.
- **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 checks if the task that just ended has siblings which need to execute. If not, the parent is woken up, else the scheduler assigns the current thread a new task from the bottom of the queue.