Bogdan Vancea , Shanmuganathan Puspanantha 25.10.2013

Supporting Parallelism in Operating Systems & Programming Languages

Exercise 2: Scheduling

Final Report

1. User Space Scheduling

1.1 API

We have made some changes to the API from the date of the intermediate report. The new API is formed of the functions below:

```
void task_init();
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();
```

void task_init() - Initializes the execution stack and the scheduler.
void task_deinit() - Frees the resources used by the task API.
task_t* task_spawn(**void*** fct_ptr, **void*** arguments, **void** *return_val) - Create a new task to execute the function fct_ptr with the given arguments. The value will be written to the address pointed by *return_val*. A pointer to the newly created task is returned.
int task_sync(task_t** execution_context, **int** count) - Allows the current execution context (task) to wait until the completion of other tasks. This function receives an array of task pointers as arguments and the number of the tasks in the array.
RETURN(value, type) - A macro that allows a task to return a value. The value *value* of type *type* is written at the return pointer address specified in the *task_spanw()* function.

The implementation for the Fibonacci function using our API would looks like in the code presented below:

```
int fib(int n) {
        int result;

        task_t* t[1];
        t[0] = task_spawn(fibo, &n, &result);
        task_sync(t,1);
        return result;
}
```

```
void fibo(void* args) {
	int n = *((int *) args);
	debug("Executing fibo %d", n);
	int* returned;
	ALLOCATE_INT_VALUE(returned, n);

	if (n >= 2) {
		int *x, *y, *arg_x, *arg_y;
		ALLOCATE_INT(x);
		ALLOCATE_INT(y);
		ALLOCATE_INT_VALUE(arg_x, n-1);
		ALLOCATE_INT_VALUE(arg_y, n-2);

		//create 2 contexts for the child functions
		task_t* ct[2];

		//execute the child functions in different execution contexts
		ct[0] = task_spawn( &fibo, arg_x,  x);
		ct[1] = task_spawn( &fibo, arg_y, y);

		//wait for the child contexts to complete
		task_sync(ct, 2);

		//return the result
		*returned = *x + *y;

		free(x);
		free(y);
		free(arg_x);
		free(arg_y);
	}

	RETURN(returned, int);
}
```

We use a wrapper function to abstract the fact that *fibo* function is executed by a collection of concurrent tasks.

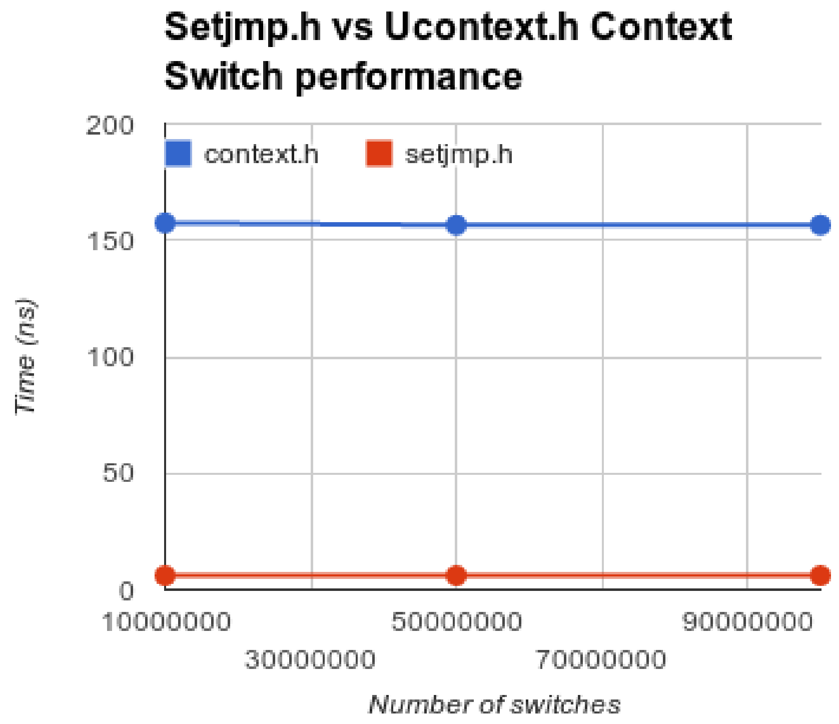
1.2 setjmp.h vs ucontext.h

1.2.1 Benchmark results

For the benchmark we measured the context switch time offered by setjmp/longjmp and setcontext/getcontext. In the intermediate report we predicted that the setjmp/longjmp pair will offer better performance due to the fact it saves less information.
The programs used for the benchmarking measure the time necessary to jump back and forth between two functions for a number of times. To perform this, a volatile int counter variable is incremented between each 2 consecutive jumps. To obtain the time needed for perform a single jump (or context switch) we measure the total time needed to perform x context switches, then subtract the time needed to perform only the increments of the counter and divide by the number of switches.

$$t_{switch} = \frac{total\ execution\ time - execution\ time\ without\ switches}{number\ of\ switches}$$

We implemented the switches using the both setjump/longjmp and getcontext/setcontext. The plot below shows the comparison between context switch times offered by the two implementations. We measured the average time for a switch for a program running on back03.ethz.ch doing 10^7 , 5 * 10^7 and 10^8 switches.



As expected, setjmp/longjmp offers offers a switch time close to 5ns while getcontext/setcontext switch time takes around 150ns. We think this difference might be cause by the implementation of getcontext/setcontext on the machines we tested. We suspect that setcontext performs a system call to a signal handler, which would explain the difference. A table containing the actual times measured is presented below.

Switches	ucontext.h	ucontext.h	setjmp.h	setjmp.h
	Average time (ns)	Standard deviation(ns)	Average time (ns)	Standard deviation(ns)
10000000	157.465	0.069	6.04397	0.000727
50000000	156.548	0.04	6.043894	0.00027
100000000	156.551217	0.027399	6.043791	0.000104

1.3 Implementation

We chose to use the functions defined in ucontext.h for implementing the task scheduling API. The implementation uses a separate *ucontext_t* structure, which we call the scheduler context, to schedule various tasks. The tasks spawned using the API are kept in a run-queue which is only accessed in the *scheduler* context. The most important aspects of our implementation are highlighted below:

- on task spawn, the currently running task (the parent) creates a new task_t structure for the child task and then invokes the scheduler. It is only inside the scheduler context that the current task is placed in a run-queue and the new task (child) is executed.
- when a task returns, it does so using a special macro implemented by us instead of the return instruction. This macro writes the returned value to the required address and jumps to the scheduler context where the current task is marked as completed and a new task is chosen to run.
- inside the *sync* method, a parent task will check if all the tasks received as an argument have returned and yield the control to the scheduler if they have not. When this happens, that task is placed at the end of the run queue.

2 Scheduling OpenMP parallel for loops

OpenMP specifies two models for loop iteration scheduling:

- static - each thread receives a fixed number of loop iterations to execute
- dynamic - the iteration assignment is performed at run-time and threads receive more iterations to execute as the complete previous iterations.

Scheduling example where dynamic scheduling is always superior

```
int a[1000];
#pragma omp parallel for schedule(static, 2)
for (int i = 0; i < 1000;i++) {
   if (i == 1) {
      usleep(10000);
   }
   a[i] = i;
}
```

In the example above, a for loop is executed for 1000 times. However, there is one iteration with a have a penalty, because the thread executing it will sleep for 10 seconds. In the scheduling context presented above, with static scheduling on 2 threads, each thread will receive 500 iterations. Due to the ‘penalty’ iteration, the thread executing it will sleep for 10 seconds and perform another 499 iterations, while the other thread will remain idle after finishing its own iterations. If the scheduling was dynamic, the thread not sleeping would be assigned more iterations and the loop would finish a little faster.

Disadvantages of dynamic scheduling

While dynamic scheduling can prove to be very useful in some situations, especially when the iteration workload is different or not known, it does have some disadvantages.

- Its implementation is more complex, because the decisions of which thread executes a certain task is made at run time. Because of this, dynamic scheduling would usually need more resources than static scheduling (ex. a separate scheduler thread which performs the work scheduling, extra memory to store meta-information about task, etc)
- In situations where the tasks have the same amount of work, dynamic scheduling would actually be slower, because the scheduling introduces some unnecessary delay.
- Dynamic scheduling techniques usually try to find an optimal scheduling of tasks. However, when the optimal assignment of tasks can only be determined after running the program