Department of Computer Science, ETH Zürich

Supporting Parallelism in Operating Systems & Programming Languages

Fall 2013

Exercise 2: Scheduling

1 User-space scheduling

The goal of this exercise is to take the first steps in building a run-time for scheduling parallel tasks.

1.1 API

We want to create a run-time, where the user can create parallel tasks and wait for their completion. The tasks should be able to return values which will become available after completion to the task's waiter. Define an API that can be used to implement the functionality described above.

For your intermediate report:

- provide a header file with the API
- If you considered different approaches, briefly discuss them
- Consider the following cilk function:

```
cilk int fib(int n)
{
    if (n < 2)
        return (n);
    int x, y;
    x = spawn fib(n - 1);
    y = spawn fib(n - 2);
    sync;
    return (x + y);
}
```

Provide an implementation of `fib()` using your API

1.2 `setjmp.h` vs `ucontext.h`

Create a benchmark that compares the performance of (user-level) context switching using functions `setjmp()` and `longjmp()` from `setjmp.h`, against using the

functions defined in `ucontext.h` (`setcontext()`, `getcontext()`, etc). Note that the former approach cannot be generally used for context switching, and you will need to overcome this limitation for your benchmark. In the context of this exercise, you can ignore any signal management complications.

For more information:

- GNU libc: non-local exits

For your intermediate report:

- Provide a description of the benchmark
- Make a prediction of the benchmark results
- Why `setjmp()` and `longjmp()` cannot be used safely for context switching?
- Is it possible to extend `setjmp()` and `longjmp()`, so that they can be safely used for context switching? How?

For your final report:

- provide and discuss your benchmark and its results
- **For bonus points:** provide an implementation of the `setjmp.h` extension described above

1.3 Implementation

Implement the API described in §1.1 using just one thread. You can redefine the API if you want. You can either use `ucontext.h` functions, or your `setjmp.h` extension.

For your final report:

- Provide a brief description and code for your implementation
- With your code, provide two working examples: (a) a program that creates two tasks that print a message, and (b) a program that computes a Fibonacci number using multiple tasks.

2 Scheduling OpenMP parallel for loops

According to the OpenMP specification[1] you can define different scheduling algorithms for a parallel for loop. Construct an artificial example where using *static* scheduling is always worse than using *dynamic* scheduling.

For your final report:

- Describe the artificial example
- **for bonus points**: implement the example using OpenMP, and verify that using *dynamic* scheduling in this example is better than *static* scheduling
- Discuss the disadvantages of *dynamic* scheduling

[1] Section 2.5.1 in `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`

For bonus points:

- Use the kernelshark tool[2] to analyse how linux schedules threads in the following cases:
  - single OpenMP program that uses threads equal to the number of cores
  - single OpenMP program that uses threads equal to the number of cores plus one
  - two OpenMP programs, each using threads equal to the number of cores

  Discuss the results.

3 Dates & Deliverables

18.10.2012: Intermediate report
25.10.2012: Final version of the code & final report

[2] see https://lwn.net/Articles/425583/