Supporting Parallelism in Operating Systems & Programming Languages

Exercise 2: Scheduling

Intermediate Report

1. User Space Scheduling

1.1 API

The API we plan to use is highlighted below.

```
/**
 * Identifies a single execution context.
 */
typedef struct context context_t;

/**
 * Run the function pointed by fct_ptr with the given arguments in a different execution
context.
 *
 * Arguments:
 *
 * execution_context  :        a pointer to an execution context used for this task
 * fct_ptr                :        a pointer to the function executed for this task
 * arguments          :        an array of arguments which should be passed to the function
 * argc                           :         the number of arguments in the array
 *
 * returns                :        a void pointer containing the address of the result
 */
void* span_worker(context_t* execution_context, void* fct_ptr, void* arguments, int argc);

/**
 * Barrier method, used to make the current thread wait for the completion of the
execution context
 * received as argument.
 *
 * Arguments
 *
 * execution_context  :        a pointer to an execution context used for this task *
 */
void* sync_worker(context_t* execution_context);
```

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

```
#include "rts_api.h"

int fib(int n) {

        if (n < 2) {
                return n;
        }
        int *x, *y;
        //create 2 contexts for the child functions
        context_t ct_x, ct_y;

        //execute the child functions in different execution contexts
        x = (int *) span_worker(&ct_x, (void) &fib, (void*) (n-1), 1);
        y = (int *) span_worker(&ct_y, (void) &fib, (void*) (n-2), 1);

        //wait for the child contexts to complete
        sync_worker(&ct_x);
        sync_worker(&ct_y);

        //return the result
        return (*x + *y);
}
```

1.2 setjmp.h vs ucontext.h

1.2.1 Benchmark description

For the benchmark, we propose measuring the execution time of a parallel program executed by a run time system implemented using setjmp.h and a run time system implemented using ucontext.h. Because the program will execute the same work, the only difference that should be seen between the execution times is the difference between the context switch times. To ensure we can accurately measure this difference, we will opt for a program with a large number of context switches.

1.2.2 Benchmark result prediction

Functions defined in ucontext.h save more values for the execution context and are generally slower than setjmp()/longjmp(). We therefore expect that cooperative multitasking context switching implemented with ucontext.h functions will be slower than the setjmp.h alternative.

1.2.3 Why setjmp() and longjmp() cannot be used safely for context switching?

setjmp()/longjmp() does not provide any mechanism for stack overflow detection or stack

overflow protection. Because of this, it is possible to for a task running in one context to overwrite the context of another task.

1.2.4 Is it possible to extend setjmp() and longjmp() so that they can be safely used for context switching?

It is possible to use setjmp() and longjmp() for context switching as long as we can ensure that the context used by longjmp() is valid. This could be done by manually allocating a “stack” for each context. This stack should be large enough for the context to execute without having to overwrite other stack frames. In short, some sort of stack overflow protection can be ensured.