Bogdan Vancea , Shanmuganathan Puspanantha 18.10.2013

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:

```
typedef struct context context_t;
```

We need a context structure to store state information about the tasks. This information should be populated right before switching the context to another task.

```
void spawn_worker(context_t* execution_context, void* fct_ptr, void* arguments);
```

This function is used to create a new execution context for a certain task. It is similar to the pthread_create function from the pthread library.
context_t execution_context* - specifies the execution context for the current task.
void fct_ptr* - a pointer to the function that should be executed in the context of the new task
void arguments* - a void pointer to the arguments of the function. This could be a pointer to a structure or a single variable, similar to the pthread library.

```
void* sync_worker(context_t* execution_context, void *return_val);
```

This function is used to stop the execution of the current task until the task identified by the *execution_context* parameter.
context_t execution_context* - specifies the execution context for the current task.
*void *return_val* - a pointer to an address where the result returned by the task is.

We also considered small variations of the presented functions. For example, it could be possible to add a *sync* function that would take as parameter an array of execution context and wait until all the respective tasks are completed, but that approach implies a change in how the return values are presented to the user of the API. It would be possible to return an array of pointers to where the results are stored, or simply to place the result parameter in the *spawn* function.

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

```
#include "rts_api.h"
#include <stdlib.h>

#define ALLOCATE_INT(x) x = (int *) malloc(sizeof(int));

int fib(int n) {

	if (n < 2) {
		return n;
	}
	int *x, *y, *arg_x, *arg_y;
	ALLOCATE_INT(x);
	ALLOCATE_INT(y);
	ALLOCATE_INT(arg_x);
	ALLOCATE_INT(arg_y);

	//create 2 contexts for the child functions
	context_t ct_x, ct_y;

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

	//wait for the child contexts to complete
	sync_worker(&ct_x, (void*) x);
	sync_worker(&ct_y, (void*) y);

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

1.2 setjmp.h vs ucontext.h

1.2.1 Benchmark description

For benchmarking we will use a simple cooperative multitasking program where two tasks will take turns in incrementing a variable. Each task will thus increment a variable and then yield the execution to the other task, until a certain number of switches happened. To measure the execution switch time, we mark the variable incremented as volatile, to ensure a memory access at each incrementation (and thus a constant execution time between switches). We will implement this program using both setjmp.h and ucontext.h functions and measure the execution times in both cases. Upon comparing the execution times, the only difference in execution time should be caused by the different duration of the 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. This happens because longjmp invalidates the stack frame of the function it is called from. However, because most implementations do not erase the variables in that stack frame, it is possible to jump back to that task as long as no other task overwrites it (this happens of that other task allocates a lot of memory and needs more stack frames).

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 large enough block of memory for each task. This memory should be large enough for the context to execute without having to overwrite other stack frames. In short, some sort of stack overflow protection should be ensured.

Another approach would be to manually allocate stack frames on the heap for each task and the using some assembler instructions to manipulate the stack pointer to point to the desired locations.