Threads - roll your own

Johan Montelius

VT2016

1 Introduction

This is an experiment where we will explore the concept of an *execution context* and implement concurrency to better understand how multi-threading works. You will basically do the first steps of what is needed to create your own thread library.

The program that you will write does probably not look like anything that you will ever do again but it's fun to see that it works. After this assignment you will have a far better understanding of what an execution context is and how multi-threading works.

2 An execution context

You know that an execution context consists of a heap, a stack, code, an instruction pointer, registers etc When you're programming in C you only use the heap explicitly, the stack etc are implicitly used when doing procedure calls and declaring local variables. The *context of the process* also includes open file descriptors, signal handlers and a myriad things that the operating system keeps track of. In this assignment we will only deal with the things that are specific for an *execution* i.e. the stack and instruction pointer.

We will explore what an execution context means by the support found in the library `ucontext`. Here we find a set of procedures that allows us to capture the current execution context, change it and even start using another context. As you will see we will be able to switch from one execution to another by switching context.

2.1 get the context

The magic in this assignment is the data structure `ucontext_t` that among other things holds a pointer to an execution stack, copy of the registers and instructions pointer. We will only touch the execution stack explicitly and let the procedures of the `ucontext` library do the rest.

The header files might not be installed on your system from start so first make sure that we have all the header files needed. Run the following command in the shell:

```
>sudo apt-get install linux-headers-$(uname -r)
```

x

We then turn to our program. The first ting we will try, is to get hold of the current execution context and swap one context for another. Try the following in a file called `switch.c`.

```
#include <stdlib.h>
#include <stdio.h>
#include <ucontext.h>

int main() {

  int done = 0;
  ucontext_t one;
  ucontext_t two;

  getcontext(&one);

  printf("hello \n");

  if(!done) {
    done = 1;
    swapcontext(&two, &one);
  }
  return 0;
}
```

We allocate two contexts, `one` and `two`, and then use the procedure `getcontext()` to populate the first context with the state of the running context. This will store among other things the program counter in the context data structure. We then print `hello` to the standard output and move to the strange looking section where we call `swapcontext()`.

The first time we enter this section `done` will be zero so the test will succeed and we will select the alternative. We set the variable `done` to one since we only want to do this once and then call `swapcontext()`. This call will save the current execution context in structure `two` and copy the context of `one` into the proper registers of the CPU. When we continue we will of course continue from the position where `one` was saved. Compile the program and explain what you see.

2.2 registers and optimizations

To see that it is more than just the instruction pointer that is stored in the context we can try the following small change to our program.

```
  register int done = 0;
```

The `register` modifier will ask the compiler to store this variable in a register (if possible). If we now compile and run our program something else will happen (be quick on the Ctrl-C).

To complicate things even further we can try to compile the original program using some compile optimizations. Try the following and see how things change, what is going on?

```
$ gcc -O2 switch.c
$ ./a.out
  :
  :
```

So we learn that the compiler can do strange things. Most of the time it does things that works, and works better, but sometime things does not really work as we think it works. Let's not use compiler optimizations for the rest of this assignment.

3 allocating a stack

We will now make the situation a bit more complicated, we will allocate a new stack. We will have three contexts all in all. One is the *main context*, that will have the regular C stack. The other two contexts will have their own stacks that we have allocated on the heap. We will also explicitly set the program pointers of the other two contexts to make them start their execution on some more interesting place.

Create a new file called `yield.c` and try the following. We will go through it step by step. The first section is include directives and the declarations of our contexts. We will no keep these as global data structures since we want to make the code simpler.

```
#include <stdlib.h>
#include <stdio.h>
#include <ucontext.h>

#define MAX 10

static int running;

static ucontext_t cntx_one;
static ucontext_t cntx_two;
static ucontext_t cntx_main;
```

We then define a procedure `yield()` that will allow us to switch between the two contexts. We assume that the variable `running` is either `1` or `2` depending on which context that is executing, switching between them is then a simple task.

```
void yield() {
  printf(`` - yield -\n'');
  if(running==1) {
    running = 2;
    swapcontext(&cntx_one, &cntx_two);
  }  else {
    running = 1;
    swapcontext(&cntx_two, &cntx_one);
  }
}
```

Next we define a procedure `push()` that only serves the purpose of making the stack grow. We do a print out when we push a call on the stack and then another on the way down. The format string will print a number followed by an indent "push" or "pop" where the indentations is given by the recursion depth.

```
void push(int p, int i) {
  if(i<MAX) {
    printf("%d%*s push\n", p, i," ");
    push(p, i+1);
    printf("%d%*s pop\n", p, i, " ");
  } else {
    printf("%d%*s top\n", p, i, " ");
  }
}
```

Now for the thing that will *tie the room together*, setting up the two contexts and take it for a spin. We allocate two stacks, 8Kbyte each, called `stack1` and `stack2` (are they allocated on the stack or on the heap?). Next we initialize each context using the procedure `getcontext()`, this will populate the contexts with all the information that the operating system needs. Then we explicitly change the `uc_stack` elements of the contexts. This is a structure holding pointers and size of the stack. We are thus giving each context a stack of their own.

We then call `makecontext()` providing a reference to each context and the procedure that they should call when starting the execution. We provide the name of the procedure, the number of arguments and the two arguments as parameters. The two contexts will both use the `push` procedure, one using the arguments `1` and `1` and the other `2` and `1`.

```
int main() {

  char stack1[8*1024];
  char stack2[8*1024];

  /* The first context. */

  getcontext(&cntx_one);
  cntx_one.uc_stack.ss_sp = stack1;
  cntx_one.uc_stack.ss_size = sizeof stack1;
  makecontext(&cntx_one, (void (*) (void)) push, 2, 1, 1);

  getcontext(&cntx_two);
  cntx_two.uc_stack.ss_sp = stack2;
  cntx_two.uc_stack.ss_size = sizeof stack2;
  makecontext(&cntx_two, (void (*) (void)) push, 2, 2, 1);

  running = 1;

  swapcontext(&cntx_main, &cntx_one);

  return 0;
}
```

When you compile and run the program you might not be super exited since nothing much happens. We start the program and then pass control over to context one. This context will of course call `push()` and push and pop the stack before terminating but the second context is never given time to execute. However, everything is prepared to swap between the contexts. The only thing we need to do is add a call to `yield()` in the `push()` procedure. Let's call yield when were at the top of the stack.

```
     :
  } else {
    printf("%d%*s top\n", p, i, " ");
    yield();
  }
}
```

Now things are starting to look more interesting; the two contexts take turn executing. You can do a yield in every recursion if you like and see how the execution jumps from one context to the other. Before we're done we shall fix one more thing.

4 termination

The switching between contexts works fine but there is one problem with our example that might not be obvious. When one context terminates the whole execution terminates. If we yield the execution when at the top of the stack we will see that only the first context will finish the execution. The first context will push all entries on its stack, then the second context will do the same but then when the first context resumes execution it will pop all entries and terminate the whole program. We need a way to capture the termination and schedule the second context again.

The solution that we will implement is not the most general solution. It is, as the `yield()` procedure, hardwired into only handle our two contexts. We will later discuss a more general solution but this will do for now.

We begin by adding yet another context, `cntx_done`, this will be the context that we switch to when a context terminates. We also add two flags that will keep track of which contexts that have terminated their execution.

```
static int done1;
static int done2;

static ucontext_t cntx_done;
```

Next we define a procedure `done()` that will be the procedure of `cntx_done`. This procedure will be scheduled when one of the contexts terminate and we should then switch to the remaining but we need to keep track of which context is still alive so we do some bookkeeping using the flags `done1`, `done2` and `done`. When both contexts have terminated we're done.

```
void done() {

  int done = 0;

  while(!done) {
    if(running == 1) {
      printf(" - process one terminating -\n");
      done1 = 1;
      if(!done2) {
        running = 2;
        swapcontext(&cntx_done, &cntx_two);
      } else {
        done = 1;
      }
    } else {
      printf(" - process two terminating -\n");
      done2 = 1;
      if(!done1) {
        running = 1;
        swapcontext(&cntx_done, &cntx_one);
      } else {
        done = 1;
      }
    }
  }
  printf(" - done terminating -\n");
}
```

Now for the part where we set this up. We need another stack for `cntx_done` so we allocate it as the other stacks. Next we do the trick that will make things work, we set a element called `uc_link` in each of the contexts `cntx_one` and `cntx_two`. This should be done before the call to `makecontext()`. This will set u the context in such way that when the context terminates we will automatically switch to the context`cntx_done`.

```
    :
  char stack_done[8*1024];
    :

  cntx_one.uc_link = &cntx_done;
    :

  cntx_two.uc_link = &cntx_done;
    :
```

The last thing we need to do is initialize the new context. We here also give it the context `cntx_main` to be the context that it should switch to when terminating.

```
  getcontext(&cntx_done);
  cntx_done.uc_link = &cntx_main;
  cntx_done.uc_stack.ss_sp = stack_done;
  cntx_done.uc_stack.ss_size = sizeof stack_done;
  makecontext(&cntx_done, (void (*) (void)) done, 0);
    :
```

We still start the execution by switching to the first context but we add some printout to follow the execution. Note that when we switch to the first context we store the current execution environment in `cntx_main`. This is the context that we switch to once the `cntx_done` is terminating. We will if everything works out see the last printout before the whole execution terminates.

```
  running = 1;

  printf(" - let's go! \n");
  swapcontext(&cntx_main, &cntx_one);
  printf(" - that's all folks -\n");
```

If everything works you should now see how the two contexts both execute to termination. Add a call to `yield()` at several places in the procedure `push()` to switch several times, it works right?

5 a library

Before you started this assignment you probably had no idea of how to implement your own threads library. Now it should be, at least partly, clear on how to start. The experiment that we have done has actually done the hard part, switching between different context. Tet has of course been hardwired into only work for the two contexts that we have used but we could generalize it to work for several contexts, how hard could it be?

We would need to keep track of several contexts so why not keep a list of contexts that should be scheduled. When a context calls `yield()` we would simply place the context last and schedule the next context in the list.

If a context terminated, the *done context* would be scheduled and it would deallocate the context and schedule the next context in the list. If the list is empty the execution terminates. We would of course have to provide a procedure `spawn()` that would allocate a new context on the heap (with it's own stack) and insert it in the list of contexts to schedule.

We could continue to rely on explicit yielding of execution but we could also add a timer interrupt that would make a call to `yield()` every 100 ms. Adding support to wait for a context to terminate could be one feature but would require some bookkeeping; we need to attach the context to a list of suspended threads attached to the thread that we are waiting for. There are probably more features that we would like to have but nothing that is too hard to solve.

You might wonder why one would like to implement threads library but it could have its advantages. The switching between contexts could be more efficient since the kernel is not involved. There are however many things that could go wrong so one should have very good reasons for not using the standard pthreads library. The idea with this assignment is not to make you implement a new threads library but to get some insight into threads scheduling and that it's not all magic.