Context Exercises

A Study Guide for the Final Project

Download *context_lab.zip* from Canvas and copy it to a directory in *csce.unl.edu*. Next, login to *csce.unl.edu*. Once in your directory, unzip it and you should have a folder called *context_lab* in your directory. In that folder, take a look at program *context.c* (listed below).

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

int main(void)
{
   ucontext_t context0;
   int counter = 0;
   long long int testval;
   char message[64];
   // try swapping these two lines
   testval = 100;
   getcontext(&context0);
   // What is the purpose of the line above?
   strcpy (message, "Original Message");
   fprintf (stderr,"1. message = %s, testval = %lld\n", message, testval);
   strcpy (message, "New Message");
   fprintf (stderr,"2. message = %s, testval = %lld\n", message, testval);
   testval = testval * 2;
   fprintf (stderr,"3. message = %s, testval = %lld\n", message, testval);
   strcpy (message, "Newest Message");
   testval = testval * 4;
   fprintf (stderr,"4. message = %s, testval = %lld\n", message, testval);
   while (testval > 0)
   {
      setcontext(&context0);
      // What is the purpose of the line above?
   }
}
```

Next, compile this program (using `gcc context.c o context`) and run this program (using `./context`) and observe the output. To really understand how this program works, we will use GNU Debugger (GDB) to perform step-by-step execution. If you have never used GDB, a how-to document is provided in our class website under How-to page. As you are stepping through the program, pay special attention to register RIP (program counter) and RSP (stack pointer) especially around `setcontext` statement. Once you are more familiar with the program, answer the following questions.

Question 1. Explain the execution path of the program. The first time when `setcontext` is executed, what happens to the execution? Why?

Next, examine *context2.c* in your *context_lab* directory. The program is also listed below:

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


void proc1(char*);
void proc2(char*);
ucontext_t GlobalContext;

int main(void)
{
   int retval;
   int count = 0;
   char message[] = "Original Message";
   getcontext(&GlobalContext);
   fprintf (stderr,"%d. message = %s\n", count, message);
   if (count >= 4)
      return;
   // The following line is quite important.
   // Notice that count retains its current value
   // after we set the context back to GlobalContext.
   count++;
   if (count % 2 == 0)
      proc1(message);
   else
      proc2(message);

}
void proc1(char* message)
{
   strcpy (message, "Newer Message");
   setcontext(&GlobalContext);
}
void proc2(char* message)
{
   strcpy (message, "Newest Message");
   setcontext(&GlobalContext);
}
```

The differences between this program and the previous program are as follows:

1. `GlobalContext` is a global variable which means that it is visible across function calls.
2. Two `setcontext` calls are made from functions outside of main.

Using GDB, monitor the execution of this program. Again pay special attention to RIP register. Try to understand how the program works.

Question 2. Explain the execution path of the program. When `setcontext` is executed, which method is executed after the statement? Why?

At this point, you should realize that `getcontext` can be used to create an execution entry point. That is, when `getcontext` is called, the processor's context and stack environment are saved and `setcontext` can be used to restart a program at this exact same point in execution. Our next task is to create necessary components that would allow multiple execution contexts. As you should already know, each thread has its own text section and stack space. Executed code can be the same or different among all threads. Inside *context_lab* directory, examine and execute *context3.c*. From the execution, it should be apparent that this program is not correct.

Question 3. Based on your experience with `getcontext` and `setcontext`, explain why this program stays in an infinite loop?

There are two functions that have been created in this program, `insert()` and `delete()`. These two functions operate on a global variable `i`. We have already made two invocations of `getcontext` at the beginning of the program. These two `getcontext` calls will serve as two different execution contexts. However, these two contexts execute exactly the same code and also share the same stack space. Our goal is to create two different stack spaces for these two execution contexts and then map one context to insert() and another to delete(). To do so, you need to implement the following steps.

Step 1. You will need to create a stack space for each of your execution context. I have already declared two variables for your stacks (`stack1` and `stack2`) as character arrays of size 1024 bytes. You can then evaluate if this size is sufficient after you run the program.

Step 2. The definition of `ucontext_t` is available through the man page for `getcontext`. You need to focus on the following structure and field in the sturcture: `uc_stack` and `uc_link`. Accessing to the data in stored in a variable of type ucontext_t can be easily done by dereferencing into structure. For example, to access the value of the stack pointer stored in buf1 (from our example), you would use `buf1.uc_stack.ss_sp`.

Step 3. Overwrite the current values of:

```
buf1.uc_stack.ss_sp,
buf1.uc_stack.ss_size,
buf1.uc_link,
buf2.uc_stack.ss_sp,
buf2.uc_stack.ss_size, and
buf2.uc_link
```

with the new stack addresses and the starting address of `insert()` and `delete()`. If you are successful in modifying *context3.c* (save it as *context3i.c*), you should be able to compile and get the following output after execution:

```
Main: Jumping to insert
Insert an item
Done inserting! i = 1
Main: Jumping to delete
Delete an item
Done deleting! i = 0
Main thread --- all done
```

At this point, you have created two different execution tasks. Each task has its own stack space. In effect these two tasks are your threads of execution.