CSE 511 - Project 1 Report

Group Members:

Ankush Aniket Mishra (aam6386@psu.edu)
Vinay Kumar Vemuri (vvv5079@psu.edu)

Introduction

This report serves as a brief summary of the approach to the project, the lessons learned, and how the work was distributed among the both of us. The goal of this project was to start a program/function on one machine until one section, then transfer to another machine and continue execution.

Overview

Our application provides a library ***psu_thread.h*** which allows users to migrate running functions from one machine to another. This was developed at the W135 Westgate Machines. The application utilizes TCP Sockets to transfer data across machines including contextual data about the running application.

When a context is transferred it also means all the stack variables and registers are transferred across machines. So it is expected that once an application is transferred it resumes where it left off.

Files in the code base

Along with the provided test cases, we also added our own more comprehensive test cases as well to validate the application.

- `psu_thread.h`: Contains functions which starts and migrates threads
- `socket.h`: Helper utilities for handling socket based programming.
- `app4.c`: Extra test case using pointer variables on the stack
- `app5.c`: Test case using nested function migrations.

Development Approach and Code Explanations

We developed this application with the following logic which we felt intuitive to us.

Client:

1. Our initial setup on the client included, only setting up the initial state variables that we use for storing the context of the client.
2. Once the client calls **psu_thread_create** we then launch a new thread where we set a new context with our own stack of size 16834 bytes. Then the ***user_func*** is invoked after that.
3. Once the user function calls the **psu_thread_migrate** we then retrieve the context that was used for this user's function execution. Based on the current context's **EBP** we then set the **ESP** as ***(EBP + 4)** and **EIP** as ***(EBP + 1)** and the current **EBP** as the previous stack frame's EBP which is ***(EBP)**

   In essence our application restores the previous stack frame and sets the next instruction to be the one which will be called after **psu_thread_migrate** returns.
4. Then, we transfer our stack, along with the context to the other host and exit from the thread.

Server

1. Similar to the client, our initial setup includes the setting of the initial values of the variables and also the socket binding for the server.
2. On invocation of the **psu_thread_create** we restore the context that was transferred to our application, by means of **makecontext** and **swapcontext** . This also includes the stack from the other host
3. Since the context structure was already modified to contain the correct values, the function resumes as expected.

Distribution of work

The distribution of work was equal among partners. Both partners researched, developed, and tested the project.

Difficulties

One of the most challenging things for us was trying to get app1 working correctly. Although it took time, we realized it was extremely important to understand stack frames in c so that we can understand what to extrapolate and where.

References

For sockets:
https://www.geeksforgeeks.org/socket-programming-cc/

MakeContext Man Pages:
https://pubs.opengroup.org/onlinepubs/009695399/functions/makecontext.html

Glibc x86 ucontext Source Code:
https://github.com/bminor/glibc/blob/master/sysdeps/unix/sysv/linux/x86/sys/ucontext.h

X86 Registers and their information:
https://www.eecg.utoronto.ca/~amza/www.mindsec.com/files/x86regs.html

Beej's Guide to Network Programming:
https://beej.us/guide/bgnet/html/