Name:_____ **SID:** _____

CSCE 351: Operating System Kernels

Lab 2 – User-Level Thread Management (20 pts)

Must be graded by the TA at the beginning of the lab period on Oct 4, 2006.

Basic Setup:

- Accessibility to a Linux workstation
- Copy jumplab.tar from ~witty/share/csce351/ directory

Objectives:

The objectives of this lab are as follows:

- Familiarize students with *siglongjmp* and *sigsetjmp*.
- Expose students to the basic debugging process with GDB.
- Expose students to the timer signal in Linux
- Lay ground work for programming assignment 1

Estimated Lab Time: 75 minutes

Introduction

The objective of this exercise is to familiarize students with two standard C library functions *sigsetjmp* and *siglongjmp*. Function *sigsetjmp* can be used to save the context of the processor (e.g. registers) and stack environment into a buffer. This context can then be retrieved using *siglongjmp*. The first time that *sigsetjmp* is called, it will return 0. However, if it is returning from *siglongjmp*, it would return non-zero value. For more information about these functions, check the UNIX/Linux man page.

Activity 1: Introduction to *sigsetjmp* and *siglongjmp*

For this activity, you will need to log-in to osage.unl.edu. You can use your CSE username and password to log-in to osage. Once log-in, obtain the program from /home/fac/witty/share/csce351/jumplab.tar. You can copy this file directly to your directory. Once in your directory, *execute tar –xvf jumplab.tar* and you should have a folder called *jumplab* in your directory. In *jumplab* directory, we will examine program *jump.c* (listed below)

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>
```

```
#include <string.h>

int main(void)
{
   sigjmp_buf buf_ptr;
   int retval;
   sigset_t sigmask;
   int testval = 100;
   char message[] = "Original Message";
   if (sigsetjmp(buf_ptr,1) == 0)
   {
      fprintf (stderr,"1. message = %s, testval = %d\n",
           message, testval);
      strcpy (message, "New Message");
      testval = 1000;
      fprintf (stderr,"2. message = %s, testval = %d\n",
           message, testval);
   }  else {
      fprintf (stderr,"3. message = %s, testval = %d\n",
           message, testval);
      strcpy (message, "Newest Message");
      testval = 10000;
      fprintf (stderr,"4. message = %s, testval = %d\n",
           message, testval);
      return;
   }
   siglongjmp(buf_ptr, 1);
}
```

By inspecting the code, what do you think will be the output of this program?

Don't worry about the correctness of your answer to the question above. If you are not sure, just provide your best guess as the answer.

Next, compile this program (using *gcc jump.c –o jump*) and run this program (using *./jump*). Is the output similar to your answer above? Also specify the rationales for your provided answer.

To really understand how this program works, we will need to use debugging tools to monitor each step of execution. 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 *EIP* (program counter) and *ESP* (stack pointer) especially around *sigsetjump* and *siglongjump* statements.

Explain the execution path of the program. The first time when *sigsetjmp* is executed, does it go through the "if" or the "else" block? Why?

What happen after *siglongjmp* is executed? To be more specific, what happen to the EIP and ESP registers?

The second time when *sigsetjmp* is executed, does it go through the "if" or the "else" block? Why?

In the man page for *sigsetjmp*, the API for the *sigsetjmp* is provided as follows:

```
int sigsetjmp(sigjmp_buf env, int savemask);
```

The description of the function is stated as:
"The sigsetjmp() function saves the calling process's registers and stack environment see sigaltstack(2)) in env for later use by siglongjmp()."

Do you think that the term "stack environment" include the entire stack space? In other words, is the entire stack saved when *sigsetjmp* is called?

Examine *jump2.c* in your *jumplab* directory. The program is also listed below:

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>
#include <string.h>

void proc1(char*);
sigjmp_buf buf_ptr1;

int main(void)
{
   int retval;
   int count = 0;
   sigset_t sigmask;
   char message[] = "Original Message";
   sigsetjmp(buf_ptr1,1);
   fprintf (stderr,"1. message = %s\n", message);
   if (count >= 4)
      return;
   count++;
   proc1(message);
}
void proc1(char* message)
{
   strcpy (message, "New Message");
   siglongjmp(buf_ptr1, 1);
}
```

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

1. buf_ptr1 is a global variable which means that it is visible across function calls.
2. siglongjmp is called from a function outside of main.
3. we do not care about the return value of sigsetjmp.

Using GDB, monitor the execution of this program. Again pay special attention to EIP and ESP registers.

Based on your observation of this program (*jump2.c*), do you think that the term "stack environment" include the entire stack space? In other words, is the entire stack saved when *sigsetjmp* is called?

What do you think the term “stack environment” include?

Activity 2: Creating user’s level threads

You have already seen that sigsetjmp can be used to create an execution entry point. That is when sigsetjmp is called, the processor’s context and stack environment are saved and siglongjmp 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 stated in the lecture, each thread has its own text section and stack space. Executed code can be the same or different among all threads.

Inside *jumplab* directory, examine and execute *jump3.c*. From the execution, it should be apparent that this program is not correct. Based on your experience with sigsetjmp and siglongjmp, 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 invoked two *sigsetjmp* at the beginning of the program. These two *sigsetjmp* 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()*.

Step 1: you will need to create two stack spaces for each of your execution context. You can use *malloc* or *calloc* routine to do this. I have already declared the variable for your stacks (stack1 and stack2). Let’s set the initial stack size to 4096 bytes.

Step 2: The definition of *sigjmp_buf* is available in */usr/include/bits/setjmp.h*. For x86, *sigjmp_buf* has the size of 24 bytes. Accessing to the data in stored in a variable of type *sigjmp_buf* can be easily done by dereferencing into the buffer. For example, to access the value of the stack pointer stored in buf1 (from our example), you would use:

buf1->__jmpbuf[JB_SP]

Step 3: Overwrite the current values of:

buf1->__jmpbuf[JB_SP],
buf1->__jmpbuf[JB_PC],
buf2->__jmpbuf[JB_SP], and

buf1->__jmpbuf[JB_PC]

with the new stack addresses and the starting address of *insert()* and *delete()*. Notice that stacks do grow downward in Linux/X86 system. Function such as *malloc* or *calloc* returns the starting address, and as more data is stored, the space would go upward. Thus, you want to provide the ending address of your stack and not the starting address so that the stack can grow downward without memory access violation. If you are successful in modifying *jump3i.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.

Activity 3: Timer event

In this activity, we will examine the code to create a timer that can be used to trigger events (e.g. scheduling) after a specific interval.

```
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>
#include <setjmp.h>
#include <string.h>
#include <time.h>
#include <sys/time.h>

#define INTERVAL 600000

struct itimerval clocktimer;
struct itimerval oldclocktimer;

void producer(void);
void signal_processor( int signal);
int main( void )
{
   /* initialize timer to send signal every 300 ms */
   clocktimer.it_value.tv_sec = 0;
   clocktimer.it_value.tv_usec = INTERVAL;
   clocktimer.it_interval.tv_sec = 0;
```

```
   clocktimer.it_interval.tv_usec = INTERVAL;
   setitimer (ITIMER_REAL, &clocktimer, &oldclocktimer);
   sigset (SIGALRM, signal_processor);
   producer();
}

void signal_processor( int signal )
{
   printf ("get a signal\n");
}
void producer()
{
   int i;
   while(1){
      for (i = 0; i < 1000000; i++);
      printf ("Producer\n");
   }
}
```

Step 1: Compile timer.c (available in *jumplab* directory).
Step 2: Execute the binary of *timer.c*.
Step 3: Examine the execution and the output.

End of In-Lab Exercise