Homework #3

Advanced Programming in the UNIX Environment

Due: May 20, 2022

Extend the Mini Lib C to Handle Signals

In this homework, you have to extend the mini C library introduced in the class to support signal relevant system calls. You have to implement the following C library functions in Assembly and C using the syntax supported by yasm x86_64 assembler.

1. setjmp: prepare for long jump by saving the current CPU state. In addition, preserve the signal mask of the current process.
2. longjmp: perform the long jump by restoring a saved CPU state. In addition, restore the preserved signal mask.
3. signal and sigaction: setup the handler of a signal.
4. sigprocmask: can be used to block/unblock signals, and get/set the current signal mask.
5. sigpending: check if there is any pending signal.
6. alarm: setup a timer for the current process.
7. write: write to a file descriptor.
8. pause: wait for signal
9. sleep: sleep for a specified number of seconds
10. exit: cause normal process termination
11. strlen: calculate the length of the string, excluding the terminating null byte ('\0').
12. functions to handle sigset_t data type: sigemptyset, sigfillset, sigaddset, sigdelset, and sigismember.

The API interface is the same to what we have in the standard C library. However, because we are attempting to replace the standard C library, our test codes will only be linked against the library implemented in this homework. We will use the following commands to assemble, compile, link, and test your implementation.

```
$ make          # Use your provided Makefile to generate `libmini.so`
$ yasm -f elf64 -DYASM -D__x86_64__ -DPIC start.asm -o start.o
$ gcc -c -g -Wall -fno-stack-protector -nostdlib -I. -I.. -DUSEMINI test.c
$ ld -m elf_x86_64 --dynamic-linker /lib64/ld-linux-x86-64.so.2 -o test test.o start.o -L. -L.. -lmini
```

- test.c: the testing code (prepared by the TAs).
- start.asm: the program start routine (start.asm) we have introduced in the class (prepared by the TAs).

Please notice that there is a -nostdlib parameter passed to the compiler, which means that you **could not** use any existing functions implemented in the standard C library. Only the functions you have implemented in the libmini64.asm and libmini.c file can be used. In addition to your library source code, you also have to provide a corresponding libmini.h file. The testing codes will include this file and use the function prototypes and data types defined in the header file.

You can build your libsmini.so by the following commands:

```
$ yasm -f elf64 -DYASM -D__x86_64__ -DPIC libmini64.asm -o libmini64.o
$ gcc -c -g -Wall -fno-stack-protector -fPIC -nostdlib libmini.c
$ ld -shared -o libmini.so libmini64.o libmini.o
```

To ensure that your library can handle signals properly, your implemented setjmp function *must save the signal mask* of the current process in your customized jmp_buf data structure. The saved signal mask *must be restored* by the longjmp function. *This requirement is different from the default setjmp/longjmp implementation*.

Homework Submission

We will compile your homework by simply typing 'make' in your homework directory. You have to ensure your Makefile required library files including ***libmini.a***, header files ***libmini.h***, and the shared object ***libmini.so***. Please make sure your Makefile works and the output executable name is correct before submitting your homework.

Please pack your C/C++/Assembly code and Makefile into a **zip** archive. The directory structure should follow the below illustration. The *id* is your student id. Please note that you don't need to enclose your id with the braces.

```
{id}_hw3.zip
└── {id}_hw3/
        ├── Makefile
        └── (any other c/c++/assembly files if needed)
```

You have to submit your homework via the E3 system. Scores will be graded based on the completeness of your implementation.

Basic Test Cases (70%)

We will implement several simple test programs and link them against your mini C library. Here are some sample test cases: write1.c (write1.c), alarm1.c (alarm1.c), alarm2.c (alarm2.c), alarm3.c (alarm3.c), and jmp1.c (jmp1.c).

write1 (10%)

```
$ make write1
gcc -c -g -Wall -fno-stack-protector -nostdlib -I. -I.. -DUSEMINI write1.c
ld -m elf_x86_64 --dynamic-linker /lib64/ld-linux-x86-64.so.2 -o write1 write1.o start.o -L. -L.. -lmini
rm write1.o
$ LD_LIBRARY_PATH=. ./write1
Hello world!
```

alarm1 (15%)

The commands to assemble, compile, and link alarm1.c (alarm1.c), as well as the corresponding runtime output are pasted below.

```
$ make alarm1
gcc -c -g -Wall -fno-stack-protector -nostdlib -I. -I.. -DUSEMINI alarm1.c
ld -m elf_x86_64 --dynamic-linker /lib64/ld-linux-x86-64.so.2 -o alarm1 alarm1.o start.o -L. -L.. -lmini
rm alarm1.o
$ LD_LIBRARY_PATH=. ./alarm1
(3 seconds later ...)
Alarm clock
```

alarm2 (15%)

The commands to assemble, compile, and link alarm2.c (alarm2.c), as well as the corresponding runtime output are pasted below.

```
$ make alarm2
gcc -c -g -Wall -fno-stack-protector -nostdlib -I. -I.. -DUSEMINI alarm2.c
ld -m elf_x86_64 --dynamic-linker /lib64/ld-linux-x86-64.so.2 -o alarm2 alarm2.o start.o -L. -L.. -lmini
rm alarm2.o
$ LD_LIBRARY_PATH=. ./alarm2
(5 seconds later ...)
sigalrm is pending.
```

alarm3 (15%)

The commands to assemble, compile, and link alarm3.c (alarm3.c), as well as the corresponding runtime output are pasted below.

```
$ make alarm3
gcc -c -g -Wall -fno-stack-protector -nostdlib -I. -I.. -DUSEMINI alarm3.c
ld -m elf_x86_64 --dynamic-linker /lib64/ld-linux-x86-64.so.3 -o alarm3 alarm3.o start.o -L. -L.. -lmini
```

```
rm alarm3.o
$ LD_LIBRARY_PATH=. ./alarm3
^Csigalrm is pending.
```

jmp1 (15%)

The commands to assemble, compile, and link jmp1.c (jmp1.c), as well as the corresponding runtime output are pasted below.

```
$ make jmp1
gcc -o jmp1.o -c -g -Wall -fno-stack-protector -nostdlib -I. -I.. -DUSEMINI jmp1.c
ld -m elf_x86_64 --dynamic-linker /lib64/ld-linux-x86-64.so.2 -o jmp1 jmp1.o start.o -L. -L.. -lmini
rm jmp1.o
$ LD_LIBRARY_PATH=. ./jmp1
This is function a().
This is function b().
This is function c().
This is function d().
This is function e().
This is function f().
This is function g().
This is function h().
This is function i().
This is function j().
$
```

Advanced Test Cases (30%)

There will be three advance test cases, each worth **10 points**.

Hints

This might be a relatively difficult homework, so we have several hints for this homework. Reading these hints carefully may reduce your efforts to implement this homework.

1. x86_64 system call table: It should be easy for you to find one on Internet. Here is the one we demonstrated in the course (x86_64 syscall table (http://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/)).
2. If you need precise prototypes for system calls, you may refer to an online Linux cross reference (LXR) site. For example, this page shows the official prototypes form the linux kernel (include/linux/syscalls.h (https://elixir.bootlin.com/linux/v4.16.8/source/include/linux/syscalls.h#L603)).
3. You will have to define all the required data structures and constants by yourself. If you do not know how to define a data structure, try to find them from the Linux kernel source codes.
4. With LXR, you may also check how a system call is implemented, especially when an error code is returned from a system call. For example, here is the implementation for sys_rt_sigaction (https://elixir.bootlin.com/linux/v4.16.8/source/kernel/signal.c#L3711) system call in the kernel. By reading the codes, you would know that passing an incorrect sigset_t size would lead to a negative EINVAL error code.
5. For implementing setjmp with a preserved process signal mask, the recommended data structure for x86_64 is given below:

   ```
   typedef struct jmp_buf_s {
           long long reg[8];
           sigset_t mask;
   } jmp_buf[1];
   ```

   The minimal eight 64-bit values you have to preserve in the *reg* array are: RBX, RSP, RBP, R12, R13, R14, R15, and the return address (to the caller of setjmp). The current process signal mask can be preserved in the *mask* field.
6. To ensure that a signal handler can be properly called without *crashing* a process, you have to do the following additional setup in your implemented sigaction function as follows (illustrated in C language):

   ```
   long sigaction(int how, struct sigaction *nact, struct sigaction *oact) {
           ...
           nact->sa_flags |= SA_RESTORER;
   ```

```
        nact->sa_restorer = /* your customized restore routine, e.g., __myrt */;
        ret = sys_rt_sigaction(how, nact, oact, sizeof(sigset_t));
        ...
}
```

The implementation of the *__myrt* function is simply making a system call to *sigreturn* (rax = 15).

7. Please notice that the sigaction data structure used in the C library may be *different* from that used in the kernel. If the user space and the kernel space data structure are inconsistent, you will have to perform the convertion in your wrapper functions.
8. The assembly instructions we used in this homework are pretty simple. All the required instructions have been introduced in the class. Here we enumerate the possible required assembly instructions for this homework.

```
add     call    cmp     inc     jge     jmp     jne
jnz     lea     mov     neg     or      pop     push
ret     sub     syscall xor
```

9. Please ensure that your header file (libmini.h) exports additional required macros, constants, data types, and function prototypes. You may refer to this file (export.txt) to see what are the minimal requirements that would be used in the sample test codes.