

# EC440: Project 2 -- User Mode Thread Library

## Project Goals:

- Grok the idea of threads.
- Implement independent, parallel execution within a process.

## Collaboration policy:

- You are encouraged to discuss this project with your classmates/instructors but are required to write and turn in your own solution.
- You must be able to fully explain your solution during oral examination.

## Deadline:

Homework 2 is due on Tuesday, October 22<sup>nd</sup> 23:59:59 ET  
(no deadline extensions or late submissions).

## Project Description:

The main deliverable for this project is a basic thread system for Linux. In the lectures, we learned that threads are independent units of execution that run (virtually) in parallel in the address space of a single process. As a result, they share the same heap memory, open files (file descriptors), process identifier, etc. Each thread has its own context, which consists of a set of CPU registers and a stack. The thread subsystem provides a set of library functions that applications may use to create, start and terminate threads, and manipulate them in various ways.

The most well-known and widespread standard that specifies a set of interfaces for multi-threaded programming on Unix-style operating systems is called POSIX threads (or pthreads). Note that pthreads merely prescribes the interface of the threading functionality. The implementation of that interface can exist either in user-space, take advantage of kernel-mode threads (if provided by the operating system), or mix the two approaches. In this project, you will implement a small subset of the pthread API exclusively in user-mode. In particular, we aim to implement the following three functions from the pthread interface in user mode on Linux (prototypes and explanations partially taken from the respective man pages):

```
int pthread_create(
    pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine) (void *),
    void *arg
);
```

The `pthread_create()` function creates a new thread within a process. Upon successful completion, `pthread_create()` stores the ID of the created thread in the location referenced by `thread`. In our implementation, the second argument (`attr`) shall always be `NULL`. The thread is created (i.e., your library must create a new *Thread context*, cf. Lecture on Processes and Threads) and executes `start_routine` with `arg` as its sole argument. If the `start_routine` returns, the effect shall be as if there was an implicit call to `pthread_exit()` using the return value of `start_routine` as the exit status. Note that the thread in which `main()` was originally invoked differs from this. When it returns from `main()`, the effect shall be as if there was an implicit call to `exit()` using the return value of `main()` as the exit status.

```
void pthread_exit(void *value_ptr);
```

The `pthread_exit()` function terminates the calling thread. In our current implementation, we ignore the value passed in as the first argument (`value_ptr`) and clean up all information related to the terminating thread. The process shall exit with an exit status of 0 after the last thread has been terminated. The behavior shall be as if the implementation called `exit()` with a zero argument at thread termination time.

```
pthread_t pthread_self(void);
```

The `pthread_self()` function shall return the thread ID of the calling thread. For more details about error handling, please refer to the respective man pages.

## Implementation Hints/Details:

- You probably will need to use a data structure that can store information about the state of the thread (its set of registers), its stack (e.g., a pointer to the thread's stack area), and status of the thread (whether it is running, ready to run, or has exited). This data structure is often referred to as a thread control block (TCB). This is the same concept that the operating system uses Process Control Blocks (PCB) for (the OS uses PCBs to manage multiple processes -- cf. Lecture Processes & Threads). As you may need to accommodate multiple threads at the same time (assume a maximum of 128 threads can be alive at the same time), the thread control blocks should be stored in a list or a table (array).
- You should create a helper function that initializes your thread subsystem after/while the

application calls `pthread_create()` for the first time. Before the call to the helper function, there is only one thread running (the main program).

- The process of picking another thread is called scheduling. You may cycle through the available threads in a round robin fashion, giving an equal, fair share to each thread.
  - In order to switch between threads, the currently executing thread needs to call the `setjmp()` library function. `setjmp` will save the current state of the thread into a `jmp_buf` structure. Then your thread subsystem can pick another thread, and use the `longjmp()` function along with saved `jmp_buf` to restore to a previously saved state and resume the execution of the new thread.
- 
- Think about how often you want the thread of a program to call `setjmp` (and thus, give up control of the CPU). In particular, application developers may be unaware of your thread implementation, so it is unlikely that they will periodically call `setjmp` and allow your scheduler to switch to a new process. To solve this issue, you should employ signals and alarms. We can use the `alarm` or the `setitimer` function to set up a periodic timer that sends a `SIGALRM` signal every X milliseconds (assume that X=50ms for this project). Whenever the alarm goes off, the operating system will invoke the signal handler for `SIGALRM`. So, you can install your own, custom signal handler that performs the scheduling (switching between threads) for you. To install this signal handler, you should use `sigaction` with the `SA_NODEFER` flag. Otherwise (e.g., when using the deprecated `signal` function), alarms are automatically blocked while you are running the signal handler.
  - We **require** that your thread system supports thread preemption and switches between multiple threads that are ready. It is **not okay** to run each individual thread to completion before giving the next one a chance to execute.
  - To create a new thread, the system has to properly initialize the TCB for the new thread: create a new thread ID, allocate a new stack (`malloc` can come in handy) of 32,767 byte size and initialize the thread's state so that it "resumes" execution from the start function that is given as argument to the `pthread_create` function. For this, we could use `setjmp` to save the state of the current thread in a `jmp_buf`, and then, modify this `jmp_buf` in two important ways. First, we want to change the program counter (the RIP) to point to the `start_thunk` function we provide in `ec440threads.h`. Second, we want the stack pointer (the RSP) to point to the top of our newly allocated stack. `start_thunk` is a helper function that moves the value stored in R13 to RDI (why? you ask ... remember the SYSV AMD64 calling convention) and then jumps to whatever address is stored in R12. Hence, for `start_thunk` to work we should store the value of `*arg` in R13 and the address of `start_func` in R12 (note that both R12 and R13 are members of `jmp_buf`)
  - To modify the `jmp_buf` directly, we have to first understand that it is a very operating system and processor family-specific data structure that is typically not modified directly. On bandit, we can see the definition of the `jmp_buf` here in this header file: `/usr/include/bits/setjmp.h` (run `cat /usr/include/bits/setjmp.h`). Moreover, `libc` defines the following constants as the eight integer elements of this

structure:

```
#define JB_RBX      0
#define JB_RBP      1
#define JB_R12      2
#define JB_R13      3
#define JB_R14      4
#define JB_R15      5
#define JB_RSP      6
#define JB_PC       7
```

- We can see that the stack pointer has index 6 and the program counter has index 7 into the `jmp_buf`. This allows us to easily write the new values for RSP and PC (or RIP) into a `jmp_buf`. Unfortunately, there is a small complication on the Linux systems in the lab. These machines are equipped with a libc that includes a security feature to protect the addresses stored in jump buffers. This security feature "mangles" (i.e., "encrypts") a pointer before saving it in a `jmp_buf`. To convince yourself of this, generate a `jmp_buf` while stepping through the program with a debugger, and print the contents in the `jmp_buf`. You will see that many register values are exactly as in the debugger, but not RIP and RSP. Thus, we also have to mangle our new stack pointer and program counter before we can write it into a jump buffer, otherwise decryption (and subsequent uses) will fail. (`longjmp` will automatically decrypt these values. Hence it is imperative that they are properly encrypted/mangled beforehand. To mangle a pointer before writing it into the jump buffer, make use of the following function:

```
unsigned long int ptr_mangle(unsigned long int p){
    unsigned long int ret;
    asm("movq %1, %%rax;\n"
        "xorq %%fs:0x30, %%rax;"
        "rolq $0x11, %%rax;"
        "movq %%rax, %0;"
        : "=r"(ret)
        : "r"(p)
        : "%rax"
    );
    return ret;
}
```

- Remember that the start routine of every new thread expects a single argument (a void pointer called `arg`). When a new thread is launched, the start routine should run as if it were invoked by a preceding `call` instruction (but it isn't, we're setting up a "fake" context with the `jmp_buf` and the stack). Hence, we have to basically

"simulate" such a function call to the start routine. To this end, we just have to initialize the new stack for a thread and pass the argument according to the calling convention. Bandit relies on the SYSV AMD64 calling convention, hence the first six arguments are passed in registers. To make this step easier, we provide the `start_thunk` function that behaves as described above. Specifically, whatever value is stored in R13 (hopefully, `*arg`) when `start_thunk` is called, will be passed on as the first argument to the function referenced in R12 (hopefully `start_routine`). For completeness sake, `start_thunk` is defined as:

```
void *start_thunk() {
    asm("popq %rbp;\n"           //clean up the function prolog
        "movq %%r13, %%rdi;\n"   //put arg in $rdi
        "pushq %%r12;\n"        //push &start_routine
        "retq;\n"               //return to &start_routine
        :
        :
        : "%rdi"
    );
    __builtin_unreachable();
}
```

In addition to the argument, we also need to make sure that the new stack has a correct return address, to instruct `start_routine` where to return to after completion. Given that the specification requires that `start_routine` invokes `pthread_exit` upon completion, we can easily achieve this by putting the address of `pthread_exit` on the top of the stack, which will transparently be treated by `start_routine` as the return address. Make sure to adjust the stack pointer (RSP) appropriately.

- Once your stack is initialized and the (mangled) stack pointer (RSP) and program counter (RIP) are written to the `jmp_buf`, your new thread is all ready to go, and when the next SIGALRM arrives, it is ready to be scheduled!
- In your solution you will have to use `ptr_mangle` to store the mangled stack and instruction pointers into the `jmp_buf` structure. If, during development, you want to inspect these values (e.g., to check you actually got a true `jmp_buf`) you can use the following counterpart to the above `ptr_mangle` function (not necessary just useful for convenience, debugging, and sanity)

```
unsigned long int ptr_demangle(unsigned long int p) {
    unsigned long int ret;
    asm("movq %1, %%rax;\n"
        "rorq $0x11, %%rax;"
        "xorq %%fs:0x30, %%rax;"
        "movq %%rax, %0;"
    );
    return ret;
}
```

```

        : "=r" (ret)
        : "r" (p)
        : "%rax"
    );
    return ret;
}

```

## Submission Guidelines:

- The threading library must be implemented in C
- To facilitate grading, you must include the pthreads header file (`#include<pthread.h>`) in your source(s). We will compile your thread library against our test application that will call your pthread functions and check whether threads are properly started, scheduled, and terminated.
- Your makefile should compile your source files into an object threads.o file.  
e.g., `gcc -Wall -Werror -c -o threads.o threads.c`  
Importantly, since you'll want to test your own solution, you have to create another make target that links your threads.o against your test-program.
- In your home directory create a folder (e.g., `project2`) and place all of your source files, makefile and README there. Switch to the `project2` directory and execute `submit2`
- A confirmation mail of your submission is sent to your account on `ec440.bu.edu`. You can read this mail by executing `mail`.
- In the README file explain what you did. If you had problems, tell us why and what.
- You are allowed to resubmit your files. The latest submission before the deadline will be graded.

## Oral Examination:

- You are required to meet with one member of the course staff (excluding Prof. Egele) during office hours to explain your solution.