CS 416 + CS 518 Project 1: Understanding the Basics (Warm-up Project)

Due: 09/26/2025, Time: 11:59pm EST

Points: 100 (5% of the overall course points.)

This simple warm-up project will help you recall basic systems programming concepts before tackling the second project. In the first part of this project, you will use the Linux pThread library to write a simple multi-threaded program. Part 2 involves writing functions for simple bit manipulations. Part 3 introduces user-level context switching with ucontext.h and asks you to debug a non-terminating program. We have provided three code files: thread.c, bitops.c, and ucontext.c for completing the project.

We will perform plagiarism checks throughout the semester. So, please do not violate Rutgers submission policy.

Part 1: pThread Basics (30 points)

In this part, you will learn how to use Linux pthreads and update a shared variable. We have given a skeleton code (*thread.c*).

To use the Linux pthread library, you must compile your C files by linking to the pthread library using *-lpthread* as a compilation option.

1.1 Description In the skeleton code, there is a global variable `x`, which is initialized to 0. You are required to use pThread to create 2 threads to increment the global variable by 1.

Use *pthread_create* to create two worker threads, and each of them will execute *add_counter*. Inside *add_counter*, each thread increments `x` 10,000 times.

After the two worker threads finish incrementing counters, you must print the final value of `x` to the console. Remember, the main thread may terminate before the worker threads; avoid this by using *pthread_join* to let the main thread wait for the worker threads to finish before exiting.

Because `x` is a shared variable, you need a pthread mutex to guarantee that each thread exclusively updates that shared variable. You can read about using pthread mutex in the link [3]. In later lectures and projects, we will discuss how to use other complex synchronization methods.

If you have implemented the thread synchronization correctly, the final value of `x` would be 2 times the loop value.

NOTE: When evaluating your projects, we will test your code for different loop values.

We have also provided a makefile to compile all project files.

1.2 Tips and Resources

- [1] pthread_create: http://man7.org/linux/man-pages/man3/pthread_create.3.html
- [2] pthread_join: http://man7.org/linux/man-pages/man3/pthread_join.3.html
- [3] pthread_mutex: https://man7.org/linux/man-pages/man3/pthread_mutex_lock.3p.html

1.3 Report You do not have to describe this part in the project report.

Part 2: Bit Manipulation (35 points)

2.1 Description Understanding how to manipulate bits is a crucial part of systems/OS programming and will be required for subsequent projects. As a first step toward mastering bit manipulation, you will write simple functions to extract and set bits. We have provided a template file **bitops.c**.

Setting a bit means updating a bit to 1 if it is currently 0, while clearing a bit means changing a bit from 1 to 0.

2.1.1 Extracting Top-Order Bits Your first task is to complete the *get_top_bits()* function to find the top-order bits. For example, if the global variable *myaddress* is set to `4026544704`, and you need to extract the top (outer) 4 bits (`1111`, which is decimal 15), your function *get_top_bits()*, which takes *myaddress* and the number of bits (*GET_BIT_INDEX*) to extract as input, and should return 15.

2.1.2 Setting and Getting Bits at a Specific Index Setting and getting bits at a specific index is widely used across all OSes. You will use these operations frequently in Projects 2, 3, and 4. You will complete two functions: *set_bit_at_index()* and *get_bit_at_index()*. Remember that each byte consists of 8 bits.

Before calling *set_bit_at_index()*, a bitmap array (i.e., an array of bits) will be allocated as a character array, specified using the *BITMAP_SIZE* macro. For example, if *BITMAP_SIZE* is set to 4, you can store 32 bits (8 bits per character element). The *set_bit_at_index()* function passes the bitmap array and the index (*SET_BIT_INDEX*) at which a bit must be set. The *get_bit_at_index()* function checks if a bit is set at a specific index. Note that setting the 0th bit could make a char either 128 or 1 (either you set 1000_0000 or 0000_0001); for this project, it does not matter which one you pick as long as it's consistent. However, going with little-endian is the recommended method as it reflects how data is usually stored.

For example, if you allocate a bitmap array of 4 characters, `bitmap[0]` would refer to byte 0, `bitmap[1]` to byte 1, and so on.

Note that you will not modify the main function, only the three functions: *get_top_bits()*, *set_bit_at_index()*, and *get_bit_at_index()*.

During project evaluation, we may change the values of *myaddress* or other macros. You don't need to handle too many corner cases for this project (e.g., setting *myaddress* or other macros to 0).

2.2 Report In your report, describe how you implemented the bit operations.

Part 3: User-level Contexts Switching — Debugging (35 points)

In this part, you will learn how to create and switch between execution contexts using the ucontext.h library. We provide a file *ucontext.c* that demonstrates the use of getcontext, makecontext, and setcontext.

getcontext(ucontext_t ucp)*

Saves the current execution context (registers, program counter, stack pointer, signal mask) into the structure ucp. You can later resume execution from this snapshot with setcontext.

makecontext(ucontext_t ucp, void (*func)(), int argc, ...)*

Converts a context (previously captured with getcontext) into one that will begin executing at the function func. You must first allocate and assign a stack (ucp->uc_stack). When func returns, execution continues at ucp->uc_link.

setcontext(const ucontext_t ucp)*

Restores the execution context from ucp. Execution immediately jumps into that context; if successful, setcontext does not return.

More details can be found in the references listed in Section 3.4.

3.1 Description The provided code attempts to:

- Saves the current context in main_ctx with getcontext().
- Allocates a separate stack for a worker context.
- Uses makecontext() to set the worker context to start at a function.
- Transfers control from main to the worker with setcontext().
- Returns from the worker to main via uc_link.

However, the program as given does not terminate. Your job is to understand why and debug the fix for this behavior, so the program terminates properly after returning from the worker.

3.2 Debugging the context set operations. Run the provided code and observe the bug. Explain why the execution loops when the worker returns. Modify the code so that:

- The worker runs and returns once.
- Control flows back to main exactly once.
- The program terminates cleanly after freeing the allocated stack.
- Verify your output matches the expected format as shown below. Just so you know, the last print statement is something that you will add after fixing the code.

```
In main: saving main context with getcontext
In main: transferring control to worker using setcontext
In worker: started
In worker: returning (uc_link will switch back to main)
In main: back after worker returned via uc_link
//The last  print statement you need to add to print after fixing the code.
```

3.3 Report In your report, include a short explanation of the bug and the change you made to ensure termination.

3.4 Tips and Resources

- [1] getcontext: http://man7.org/linux/man-pages/man3/getcontext.3.html
- [2] makecontext: http://man7.org/linux/man-pages/man3/makecontext.3.html
- [3] setcontext: http://man7.org/linux/man-pages/man3/setcontext.3.html

Project 1 Submission

For submission, create a zip file named *project1.zip* and upload it to Canvas. The zip file should contain the following files. Please ensure that the report file is a PDF (do not submit Word or text files).

project1.zip (in lowercase) that contains thread.c, bitops.c, ucontext.c report.pdf

- Only one group member should submit the code. However, in your project report and at the top of your code files, list the names and NetIDs of all group members, the course number (CS 416 or CS 518), and the iLab machine on which you tested your code, as a comment at the top of the file.
- Your code must work on one of the iLab machines. You can use the C code provided as a base and its functions. You can change the function signature for Part 2 if you need to.