Operating Systems Programming Lab
Computer Engineering Department
Spring 2023/2024
Lab 7: Threads

Objectives

1. To understand the basics of threads and multi-threaded programming.

Prelab

1. Review Chapter 12 and 13 of the textbook.
2. Read the manual pages of the following functions of the `pthread` library:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine) (void *), void *arg);
int pthread_join(pthread_t thread, void **retval);
void pthread_exit(void *retval);
int isupper(int c);
int islower(int c);
```

Experiment

Part 1: Basic Threads

1. The following program creates a thread to convert a string to uppercase and an integer to its absolute value. Test this program and show the output to your lab instructor.

```
#include<stdio.h>
#include<pthread.h>
#include<string.h>
#include<ctype.h>

struct Data{
int x;
char str[10];
};

void* convert(void* param){
    struct Data* d = (struct Data*) param;
    d->x = d->x > 0 ? d->x : -1*d->x;
    int i;
    for(i=0;i<sizeof(d->str);i++)
    {
      d->str[i] = toupper(d->str[i]);
    }
}
```

```
void main(){
   struct Data data;
   data.x = -7;
   strcpy(data.str, "Converting");
   pthread_t tid;
   pthread_create(&tid, NULL, convert, &data);
   pthread_join(t, NULL);
   printf("X = %d, STR = %s\n", d.x, d.str);
}
```

2. **Returning values from threads:** Modify the program in the previous part as follows. Test and program and make sure it returns the same output as the original one.

```
void* convert(void* param){
    struct Data d = (struct Data*) malloc(sizeof(struct Data));
    struct Data* input = (struct Data*)param;
    d->x = input->x > 0 ? input->x : -1*input->x;
    int i;
    for(i=0;i<sizeof(d->str);i++)
    {
       d->str[i] = toupper(input->str[i]);
    }
    return d;
}

void main(){
    struct Data data;
    data.x = -7;
    strcpy(data.str, "Converting");
    pthread_t tid;
    pthread_create(&tid, NULL, convert, &data);

    struct Data* result;
    pthread_join(t, &result);
    printf("X = %d, STR = %s\n", result.x, result.str);
}
```

Part 2: Thread Synchronization

Some functions are not thread-safe, which means that they cannot be called concurrently by multiple threads. Most common reason for this (being thread-unsafe) is that the underlying function modifies some static\non-local variables. In this part of the experiment, you are required to build a wrapper function called:

`thread_safe_call(void (*f) (int), int)`

that takes as input a pointer to a function `f` and an integer value `param` to be passed to the function `f`. Then, `thread_safe_call` make the call `f(param)` in a thread-safe manner (*how?*).

1. To test the idea, create two functions "`void increment(int inc)`" and "`void decrement(int dec)`" to increment a global variable `size` by `inc` (i.e., `size += inc`) or decrement it by `dec` (i.e., `size -= dec`) respectively. Then, create two threads `incThread` and `decThread`. The `incThread` calls the `increment` function 1000000 times, and the `decThread` calls the `decrement` function 1000000 times. The main thread waits for the two threads to finish and then prints the value of `size`. Run the program and observe the output.

2. Instead of directly calling `increment` and `decrement` functions in the `incThread` and `decThread` threads respectively, call them through the wrapper function `thread_safe_call`. Do you observe any difference with the output in the previous part?