Make a thread

Amir H. Payberah

Introduction

In this assignment, you will work with threads by creating and passing arguments to them. You will use the `pthreads` library to do this assignment. When you compile a program that uses this library, you need to indicate your desire to use `pthreads` by adding the `-pthread` option at the end of the compiler command line. This option specifies that the `pthreads` library should be linked.

```
$ gcc -o myprog myprog.c -pthread
```

1 Creating and destroying threads

The first step to understanding how to build a multi-threaded program is to understand how to create and destroy threads. To create a new thread you need to use the `pthread_create()` function. It gives back a thread identifier that can be used in other calls. The second parameter is a pointer to a thread attribute object, and `NULL` means to use the default attributes (suitable for many cases). The third parameter is a pointer to the thread's function to execute, and the final parameter is the argument passed to the thread function. Your program should then wait for each thread to terminate and collect its results by calling `pthread_join()`.

```c
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void *threadfunction(void *arg) {
  printf("%s\n", (char *)arg);
  return 0;
}

int main() {
  pthread_t thread;
  char *msg = "Hello world!";

  int createerror = pthread_create(&thread, NULL, threadfunction, msg);

  pthread_join(thread, NULL);

  return 0;
}
```

Now, you have two threads, each gets a number, and the main process waits for them to finish:

```c
void *thread_func(void *arg) {
    printf("I am thread #%d\n", *(int *)arg);
    return NULL;
}

int main() {
    pthread_t t1, t2;
    int i = 1;
    int j = 2;

    pthread_create(&t1, NULL, &thread_func, &i);
    pthread_create(&t2, NULL, &thread_func, &j);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("In main thread\n");
    return 0;
}
```

2 Returning results from threads

Note that the thread functions are declared to return a pointer to `void`. However, there are some pitfalls involved in using that pointer. The code below shows one attempt at returning an integer status code from a thread function.

```
void *thread_func(void *arg) {
    int code = 5;
    return (void *)code ;
}
```

This method will only work on machines where integers can be converted to a pointer and then back to an integer without losing information. On some machines, such conversions are dangerous. This method will fail in all cases where one attempts to return an object, such as a structure, that is larger than a pointer. In contrast, the code below does not fight the type system and returns a pointer to an internal buffer where the return value is stored. While the example shows an array of characters for the buffer, one can easily imagine it being an array of any necessary type or a single object, such as an integer status code or a structure with many members.

```
void *thread_func(void *arg) {
    char buffer[64] = "Hello world!";
    return buffer;
}
```

The code above fails because the internal buffer is automatic and vanishes as soon as the thread function returns. The pointer, which is given back to the calling thread, points at undefined memory. This is another example of the classic dangling pointer error. In the next attempt, the buffer is made `static` so that it will continue to exist even after the thread function terminates. This gets around the dangling pointer problem.

```
void *thread_func(void *arg) {
    static char buffer[64] = "Hello world!";
    return buffer;
}
```

This method might be satisfactory in some cases, but it does not work in the common case of multiple threads running the same thread function. In such a situation, the second thread will overwrite the static buffer with its own data and destroy the data left by the first thread. Global data suffers from this same problem since global data always has a static duration. The version below is the most general and most robust.

```
void *thread_func(void *arg) {
    char *buffer = (char *)malloc(64);
    buffer = "Hello world!";
    return buffer;
}
```

This version allocates buffer space dynamically. This approach will work correctly even if multiple threads execute the thread function. Each will allocate a different array and store the address of that array in a stack variable. Every thread has its own stack, so automatic data objects are different for each thread. In order to receive the return value of a thread, the higher-level thread must join with the subordinate thread.

```
void *threadresult;

// Wait for the thread to terminate.
pthreadÙjoin(threadID, &threadresult);
```

The `pthread_join()` function blocks until the thread specified by its first argument terminates. It then stores into the pointer pointed at by its second argument the value returned by the thread function. To use this pointer, the higher-level thread must cast it into an appropriate type and dereference it.

```
char *message ;
message = (char *)threadresult ;
printf("I got %s back from the thread.\ n", message);
free(threadresult);
```

If the thread function dynamically allocates the space for the return value, then the higher-level thread needs to free that space when it no longer needs the return value. If this is not done, the program will leak memory.

Below, you can see a complete code of returning results from threads.

```c
struct thread_args {
    int a;
    double b;
};

struct thread_result {
    long x;
    double y;
};

void *thread_func(void *args) {
    struct thread_args *args = args;
    struct thread_result *res = malloc(sizeof *res);

    res->x  = 10 + args->a;
    res->y = args->a * args->b;
    return res;
}

int main() {
    pthread_t threadL;
    struct thread_args in = {.a = 10, .b = 3.141592653};
    void *out_void;
    struct thread_result *out;

    pthread_create(&threadL, NULL, thread_func, &in);
    pthread_join(threadL, &out_void);
    out = out_void;
    printf("out -> x = %ld\tout -> b = %f\n", out->x, out->y);
    free(out);

    return 0;
}
```

3 Acknowledge

The main part of this assignment is derived from the “pthread tutorial” by Peter C. Chapin.