Programming Assignment 4

[LAST REVISION 12/21/20, 6:12AM]

Due 11:59pm Sunday, March 7

This programming assignment is on the topic of implementing threads, and consists of 3 parts, A, B, and C. Each part consists of a set of exercises which you should do.

You will hand in ONE FILE: `mycode4.c`.

Each of the programs below can be found in separate files, `pa4a.c`, `pa4b.c`, and `pa4c.c`.

To install this assignment using your account on ieng9:

1. Log in to ieng9.ucsd.edu using your class account.
2. Enter in the command, "`prep cs120w`". This configures your account for this class. You should always do this when doing CSE120 work.
3. Enter in the command, "`getprogram4`". This will create a directory called "`pa4`" and will copy the relevant files into that directory. It is important that you do all your work in this directory (i.e., called "pa4" in your home directory). Do NOT create another directory (i.e., with a name different from pa4) to do your work, as this may have an unknown effect on how UMIX runs. You may certainly save your work by copying it to another directory, but your latest work with any changes you make should be in, and all your executions should be run from within, pa4.
4. To compile, enter the command "`make`" (from within the pa4 directory). This will compile all of the programs. You can compile a particular program by specifying it as a parameter to make, e.g., "`make pa4a`".
5. To turn in the assignment, make sure that your `mycode4.c` file that you need to turn in is in the current directory and type "`turninprogram4`".

Notes on Grading

1. You will be graded primarily on whether your code works and how well it works (is it efficient in time and/or space). While you will not be graded specifically on code structure and commenting/documentation, it is to your benefit to use good software engineering principles in structuring and commenting your code. This will especially benefit you if your code requires visual inspection by a grader. If a problem arises and we can't understand your code, or it is difficult to do so, your grade may suffer.

2. Unless indicated otherwise, you must NOT use any library routines or any code that you did not write yourself other than the routines given to you. For example, if you need a data structure like a linked list or queue, you should create it yourself rather than relying on a C library.

3. You should NOT use dynamic memory allocation in your kernel. For example, you should not use malloc (both because it is a dynamic memory allocator, and it is a C library routine which, as indicated above, you should not use). Since any dynamic memory allocator may fail (if all the memory is used), the kernel cannot depend on it (otherwise it might fail, which would be catastrophic).

4. It is your responsibility to proactively come up with YOUR OWN tests that you think are necessary to convince yourself that your kernel is doing what is asked for in the specification. If your code passes tests provided by anyone on the CSE 120 teaching staff, you should not assume that your code "works" and you are done. What ultimately matters as far as what your code is expected to do will be based on the specification you are given (i.e., these instructions). It is up to YOU to interpret it and devise any test cases you think are applicable. This mimics the experience of a real operating system designer/implementer, who has no idea what kind of applications will be written for their operating system, and how their code will be exercised. The real operating system implementer must test for anything and everything, as best they can. So, you must test robustly, and be creative in coming up with tests. You are free to ask questions about cases that you think may matter.

5. All your code must be contained in *** `mycode4.c` ***. ALL OTHER CHANGES YOU MAKE TO ANY OTHER FILES WILL BE COMPLETELY IGNORED!

   Consequently, do not put declarations, function or variable definitions, defined constants, or anything else in other files that you expect to be part of your solution. We will compile your submitted code with our test programs, and so your entire solution must be contained in what you submit. If you code does not compile with our test programs because you made changes to other files that required inclusion, YOU WILL RECEIVE ZERO CREDIT, so please be careful!

6. There should be no `Printf` statements in your `mycode4.c`, as if there are, it will disrupt the autograder's ability to check your program and you may lose points. For debugging, you may wish to use `DPrintf`, which is used and works exactly like `Printf`, except that it generates no output during the autograder's testing, so it is very safe to use.

Programming Assignment 4: Exercise A

In this assignment, you will implement a thread package.

In this first exercise, you will learn how to use some basic mechanisms for building threads, specifically, the functions **`setjmp`** and **`longjmp`**. These are standard C library functions that support "non-local jumps". You can learn more about them by reading their man pages ("`man setjmp`" and "`man longjmp`"), though you should be able to do this assignment solely with the information provided here.

`setjmp(env)` causes a process to save parts of its context in `env`, which is a "`jmp_buf`" structure defined in `<setjmp.h>`. (What is actually saved in env is machine architecture and compiler dependent; examples include the PC, SP, FP, return address, etc. We need not be concerned with these details here, though you may find these examples helpful to answer the "challenge questions" below.)

`setjmp(env)` returns twice (similar to `Fork()`), returning 0 the first time, and a value other than 0 the second time. More on this below.

`longjmp(env, t)` causes a process to do a non-local jump to the location where `setjmp(env)` was called. This is similar to a goto instruction, except that it is non-local, i.e., transfer may be across previous procedure calls that have not returned yet.

For example, given three procedures A, B, and C, say `setjmp(env)` is called within A, and then A calls B. If B then calls C, and then `longjmp(env, t)` is called within C, control returns to the point where `setjmp(env)` was previously called by A. Thus, the call stack is reduced because A then continues to run as if B and C had returned (they are no longer pending). This is because `env` contains the stack pointer SP as it was when `setjmp` was called.

It is **VERY IMPORTANT** to note that `longjmp(env, t)` can only jump into a pending procedure that had called `setjmp(env)`. In our example, A called `setjmp(env)` and THEN called B, which THEN called C, which THEN called `longjmp(env, t)`. Note that A did not return before `longjmp(env, t)` was called. If A had called `setjmp(env)`, but returned (to whatever called it), a later call to `longjmp(env, t)` would generally not work properly because the context described by env no longer exists; in particular, the activation record for A is gone.

To be more specific as to where control transfers after a call to `longjmp(env, t)`, it is to a location actually WITHIN the previous call to `setjmp(env)`. Thus, `setjmp(env)` will return twice. The first time it returns, it had just set `env` and returns a value of 0. When it returns the second time as a result of calling `longjmp(env, t)`, it (i.e., `setjmp`) returns the value `t` if it is not 0, and 1 if `t` is 0. How you use `t` is up to you.

An example will clarify the above discussion. Study the program below. To run the program, use the supplied Makefile and run `make`.

Questions

1. Can you account for the output order?
   - A: t = 1
   - B: t = 2
   - C: t = 4
   - D: t = 5

2. Make the following minimal changes so that the order is ACBD rather than ABCD: (a) change the test in the conditional to != rather than ==, and (b) move the longjmp statement so that it is just after Point C. Can you explain why the printed values of t are 1, 2, 2, 3?

3. Change the test in the conditional back to ==, and move the `longjmp` statement to just after Point D. Can you account for the process's behavior?

If you were able to answer the questions above, you now have a basic understanding for how setjmp and longjmp works. You may now proceed to Exercise B, where you will build on this knowledge, and things will get more difficult, and more interesting. For those who want an additional challenge (not necessary to complete the rest of the assignment), see if you can answer the following optional questions.

4. Change back to the original setup (conditional test is ==, `longjmp` is immediately after Point B). Change `setjmp` to `Setjmp` and `longjmp` to `Longjmp`. What do `Setjmp` and `Longjmp` do? When you run the program, you should notice different behavior; why? Hint: consider how the stack of activation records changes over time.

5. Now change `Setjmp` to `Setjmp1` and `Longjmp` to `Longjmp1`. When you run the program, do you notice any different behavior? Can you explain why? Hint: this behavior can only be justified by considering what must be saved in (and restored from) `env`.

```
#include <setjmp.h>
#include "aux.h"
#include "umix.h"

void Main()
{
      jmp_buf env;
      int t = 1;
      int Setjmp(), Setjmp1(), Longjmp(), Longjmp1();

      Printf("A: t = %d\n", t);              // Point A

      if ((t = setjmp(env)) == 0) {          // conditional test
            t = 2;
            Printf("B: t = %d\n", t);        // Point B
            longjmp(env, t);
      } else {
            t = t + 2;
            Printf("C: t = %d\n", t);        // Point C
      }
      t = t + 1;
      Printf("D: t = %d\n", t);              // Point D
}

int Setjmp(jmp_buf env)
      // env: to contain saved state
{
      Printf("Inside Setjmp\n");
      return(Setjmp1(env));
}

int Setjmp1(jmp_buf env)
      // env: to contain saved state
{
      Printf("Inside Setjmp1\n");
      return(setjmp(env));
}

int Longjmp(jmp_buf env, int t)
      // env: state to restore
      // t: thread to resume
{
      Printf("Inside Longjmp\n");
      Longjmp1(env, t);
}
```

```
int Longjmp1(jmp_buf env, int t)
      // env: state to restore
      // t: thread to resume
{
      Printf("Inside Longjmp1\n");
      longjmp(env, t);
}
```

Programming Assignment 4: Exercise B

We will now begin studying a simple thread package. You are given the file `mycode4.c`, which contains a rudimentary thread package that will only work for two threads. It contains the following functions:

`MyInitThreads()`: initializes the thread package. Must be the first function called by any user program that uses the thread package.

`MyCreateThread(f, p)`: creates a new thread to execute function `f(p)`, where `f` is a function with no return value and `p` is an integer parameter. However, the new thread does not begin executing until another thread yields to it.

`MyYieldThread(t)`: causes the running thread to yield to thread `t`. Should return the ID (integer identifier) of the thread that yielded to the thread being given control, i.e., the one returning from its call to `MyYieldThread`. More on this below.

`MyGetThread()`: returns the ID of the currently running thread.

`MySchedThread()`: causes the running thread to simply give up the CPU and allow another thread to be scheduled. Selecting which thread to run is determined in this procedure. Note that the same thread may be chosen (as will be the case if there are no other threads).

`MyExitThread()` causes the currently running thread to exit.

Let us study **`MyCreateThread`** and **`MyYieldThread`**, both of which contain some preliminary code that you may use. (Note that for the next part, you will need to change some of this code in a significant way. If you develop a good understanding of how this preliminary code works, it will help you immensely.)

First, there is a very simple thread management table, `thread[]`. You may add fields as needed. The only existing field is to save a thread's context (filled by `setjmp`, and later used by `longjmp`).

Next, `MyCreateThread(f, p)` is what does the bulk of the work. It first calls `setjmp(thread[0].env)`, which saves the context of thread 0. Thread 0 already exists by default; it is the "Main" thread.

Hence it will be creating a new thread, thread 1 (thread IDs range from 0 to `MAXTHREADS-1`, where `MAXTHREADS` is defined in `mycode4.h`).

Each thread requires its own stack. Here is perhaps the most interesting part of the code, where we simply use the current stack (for this single UMIX process, which is currently being used by thread 0), and so there is no need to actually allocate space. However, to ensure that thread 0's stack may grow and not bump into thread 1's stack, the top of the stack is effectively extended automatically by declaring a local variable `stack[]` (a large "dummy" array, which is never actually used).

Two additional automatic local variables are declared, `func` and `param`. These are used to store the passed parameters `f` and `p`, but note that `func` and `param` are near the top of the stack whereas `f` and `p` are somewhere below the "cushion" provided by `stack[]`. The importance of this will become more evident.

Next, a check is made to ensure that the stack was indeed extended. It may happen that an optimizing compiler will notice that `stack[]` is not being used, and therefore gets automatically removed. Here, just by referencing it in the conditional statement avoids this problem.
But if this conditional were removed, the compiler would likely remove `stack[]` also. You should try developing an experiment to see if this would happen, by testing the address of, say, `param`, with and without the conditional. Its address should not change, but if it does, it would indicate the absence of `stack[]` (especially if the address changed by the size of `stack[]`, i.e., `STACKSIZE`).

Now that the stack is properly set up, the bulk of the work that needs to be done by `MyCreateThread(f, p)` is complete. Thread 1 is ready to run and execute `f(p)`, but `MyCreateThread` stops short of actually doing this at this point in the code. Since this is the point where we want thread 1 to begin executing whenever thread 0 yields to it in the future, the context is saved by calling `setjmp`, this time supplying `thread[1].env` as a parameter. Since a 0 is returned the first time `setjmp` returns, the call to `longjmp(thread[0].env, 1)` occurs. Later, when `longjmp(thread[1].env, 1)` is called, thread 1 again return from this setjmp, but this time since the returned value is 1, the body of the conditional does not execute. Indeed, what Thread 1 will do is execute `f(p)`, as given by `func` and `param`.

But what happens during the first return from setjmp, when a call is made to `longjmp(thread[0].env, 1)`? The last saved context for Thread 0 was at the very beginning of the body of `MyCreateThread`, where `setjmp(thread[0].env)` was called. This now returns for a second time, with a return value of 1 (the second parameter of `longjmp`), and so the body of the conditional is skipped, and `MyCreateThread` returns. Thread 0 has now successfully created Thread 1, and can now yield to it.

This brings us to `MyYieldThread(t)`. Here, the context of the calling thread is saved. Note that the code is currently hard-wired to work with only two threads whose IDs must be 0 and 1. So when Thread 0 calls `MyYieldThread(1)`, then `1 - t` = 0 and so Thread 0's context is saved in the call to `setjmp(thread[1-t].env)`. Since the return value is 0, `longjmp(thread[t].env, 1)` is called. And where does Thread 1 begin executing from? Answer: wherever its context was last saved.

Finally, if `MyYieldThread(0)` is called by Thread 1, notice that the roles are reversed.

The problems with this code are twofold. First, it assumes there are only two threads. Second, it assumes the calling thread always yields to the other (when in fact, we must allow for the possibility that a thread might yield to itself). Hence, these issues need to be addressed.

The code below is a simple program that creates two threads, Thread 0 which is created by default and Thread 1 which is created by `MyCreateThread`. Control is then passed back and forth by calls to `MyYieldThread`.

Study this program carefully, and make sure you understand how it interacts with the code in `mycode4.c` Experiment with it extensively. (In fact, it is a good idea to make a copy so that you can modify it at will and always retrieve a clean version.)

Here are some things to try:

1. Try removing the dummy array `stack[STACKSIZE]`. What happens to the program's behavior? Try difference sizes for `STACKSIZE`.

2. Try removing the automatic local variables `func` and `param`, and replace the call to `(*func)(param)` with `(*f)(p)`. What happens to the program's behavior?

3. How would you generalize the code for `MyYieldThread(t)`? Implement your solution so that the program below continues to work properly. Make sure your code allows a thread to yield to itself.

```
#include "aux.h"
#include "umix.h"
#include "mycode4.h"

#define NUMYIELDS     5

static int square;               // global variable, shared by threads

void Main()
{
      int i, t;
      void printSquares();

      MyInitThreads();      // initialize, must be called first

      MyCreateThread(printSquares, 0);

      for (i = 0; i < NUMYIELDS; i++) {
            MyYieldThread(1);
            Printf("T0: square = %d\n", square);
      }

      MyExitThread();
}

void printSquares(int t)
      // t: thread to yield to
{
      int i;

      for (i = 0; i < NUMYIELDS; i++) {
            square = i * i;
            Printf("T1: %d squared = %d\n", i, square);
            MyYieldThread(0);
      }
}
```

Programming Assignment 4: Exercise C

You are ready to begin building a thread package for UMIX. Your task is to implement the functions:

- `MyInitThreads()`
- `MyCreateThread(f, p)`
- `MyYieldThread(t)`
- `MyGetThread()`
- `MySchedThread()`
- `MyExitThread()`

They should work in a general way, supporting `MAXTHREADS` active threads. Note that a program may create more than `MAXTHREADS` threads, as long as no more than `MAXTHREADS` threads are active at any point in time.

You are given a test program below. It currently references thread functions that are properly implemented, i.e., `CreateThread`, `YieldThread`, etc., which mirror the ones you are to implement. Eventually, you should replace all of them with reference to your functions by simply prefixing each one with "My", e.g., "`CreateThread`" becomes "`MyCreateThread`".

Note that you cannot mix your functions with the working ones, so you must replace all of them.

Some Notes:

1. **`MyInitThreads()`** should initialize all your thread management data structures. Here is where you may wish to reserve stack space **FOR THE MAXIMUM NUMBER OF THREADS** that may be active at any one time (`MAXTHREADS`).

2. **`MyCreateThread(f, p)`** should return the ID of the thread just created (assuming no errors). In Exercise B, it was assumed the return value was 1 because it could only create a single thread (with ID 1); this needs to be generalized to any value between 0 and `MAXTHREADS - 1`.

   If there is an error, such as if there are already `MAXTHREADS` active threads (and no more can be created until one or more exit), `MyCreateThread` should simply return (-1).

   IMPORTANT: Threads IDs should be integers that are assigned in increasing order.

   The initial thread (that exists by default) is thread 0. The first time `MyCreateThread` is called it should create thread 1, and each subsequent (successful) call to `MyCreateThread`, regardless of which thread makes the call, should assign IDs 2, 3, ... 9, i.e., up to and including `MAXTHREADS-1` (for `MAXTHREADS` equal to 10 in this example).

Values should be reused AFTER having reached `MAXTHREADS-1`, again starting from 0 and incrementing by 1, but a value that is in use should be skipped over (and thus not assigned to a new thread since it is already the ID of an active one).

The result is that each active thread will have a unique ID between 0 and `MAXTHREADS-1` inclusive. Note the increasing order of assignment and incrementing by 1 (this is very important, as our test programs will expect this order of ID assignment).

Here is an example for clarification: Say `MAXTHREADS = 10`, and seven threads are created: 0, 1, 2, ..., 6.
Next, threads 2 and 5 exit.
Next four threads are created: 7, 8, 9, and 2.
Since 0 and 1 still exist, those IDs are skipped over. Next 0 and 3 exit. Next three threads are created. Since the last ID assigned was 2, and since 3 is the next available, the three threads are assigned the following IDs: 3, 5 and 0.

3. **`MyYieldThread(t)`** needs to be generalized so that any thread can yield to any other thread, including itself. Also, the ID of the calling thread must be properly returned, or -1 should be returned if t is invalid or if there is no calling thread (i.e., if the return from `MyYieldThread` is due to a call to `MySchedThread`).

To clarify this further, consider a program of many threads, two of which are threads 3 and 7.

Thread 3 contains the statement:
```
x = MyYieldThread(7);       // causes thread 3 to yield to thread 7
```

Thread 7 contains the statement:
```
x = MyYieldThread(t);       // causes thread 7 to yield to thread t
```

Assume that at some point in the past, thread 7 had run and had executed its yield statement as shown above. To what it yields to is not shown as it depends on the value of `t`.

- If `t` equalled 3, then control would have gone to thread 3.
- If it were another value, another thread would have gotten control.

Regardless, assume that thread 3 is now running, and executes its yield statement as shown above. At this point, control is given to thread 7, which returns from its yield statement with `x` set to 3 (because it was thread 3 that yielded to thread 7). The value of `x` that will be set in thread 3 when its yield statement returns will depend on whatever other thread eventually yields to it.

---

4. **`MySchedThread()`** is similar to **`MyYieldThread`**, except that **`MySchedThread`** determines which thread to yield to, rather than this being specified via a parameter as in `MyYieldThread`. `MySchedThread` does not return any value.

   IMPORTANT: **`MySchedThread`** should implement the FIFO (first-in-first-out) scheduling discipline.

   Thus, if a thread calls `MySchedThread`, it should be placed at the end of a queue, and whichever thread is at the front should be selected for execution (and of course removed from the queue).
   If a thread calls `MyYieldThread(t)`, then the calling thread should be placed at the end of the queue, and t should be removed from the queue, regardless of its position, and treated as if it were at the front, i.e., selected for execution and when it gives up the CPU, it should go to the end of the queue.

5. **`MyExitThread()`** should cause the currently running thread to exit, i.e., it should never run again, and its resources, such as its entry in the thread table, should be reclaimed so that another thread may use them.

   Finally, it may call `MySchedThread()` to pass control to another thread, unless there is no thread to run, in which case it should call `Exit()` so that the UMIX process properly completes.

*

What to Turn In

You must turn in one file: `mycode4.c`, which contains your thread implementation.

Your programs will be tested with various Main programs that will exercise your threads implementation. As always, make sure you do as much rigorous testing yourself to be sure your implementations are robust.

While this assignment may be conceptually difficult, it does not require a large amount of code. The solution is roughly 100 lines of C code (which include some comments) more than the version of `mycode4.c` that was given to you.

Good luck!

```
#include "aux.h"
#include "umix.h"
#include "mycode4.h"

#define NUMYIELDS     5

static int square, cube;    // global variables, shared by threads

void Main()
{
      int i, t, me;
      void printSquares(), printCubes();

      InitThreads();

      me = GetThread();
      t = CreateThread(printSquares, me);
      t = CreateThread(printCubes, t);

      for (i = 0; i < NUMYIELDS; i++) {
            YieldThread(t);
            Printf("T%d: square = %d, cube = %d\n", me, square, cube);
      }

      ExitThread();
}

void printSquares(int t)
      // t: thread to yield to
{
      int i;

      for (i = 0; i < NUMYIELDS; i++) {
            square = i * i;
            Printf("T%d: %d squared = %d\n", GetThread(), i, square);
            YieldThread(t);
      }
}

void printCubes(int t)
      // t: thread to yield to
{
      int i;

      for (i = 0; i < NUMYIELDS; i++) {
```

```
                cube = i * i * i;
                Printf("T%d: %d cubed = %d\n", GetThread(), i, cube);
                YieldThread(t);
        }
}
```