Computer Science 432
Spring 2022
Lab 6: Contexts and context switching, due on Friday, April 22.

Objective. To understand execution context and to experiment with context-switching.

Discussion. It is sometimes surprising to people to find out that a program may, at times, not be the focus of a machine's execution. As we learn more about the abstraction of a dedicated machine, it becomes more clear that the state that encodes how a program is currently executing—its *context*—is carried in a small number of the machine's resources. We know, of course, that a locus of execution is maintained only by the state of the machine's general purpose registers, its program counter, its *status registers* (called CSRs in `RISC-V`), and its associated memory. If we can capture the context of a computation, it can be restarted later, picking up where it left off.

In this lab we'll be looking at two interesting programming abstractions: a non-local branching mechanism called a *long jump*, and an execution context that is lighter than a process, called a *thread*. Because context plays an important role in both abstractions, they'll have observably similar implementations.

The Long Jump. For the most part, computation unfolds statement-to-statement in program order. There are some basic exceptions: the `goto` statement—which allows a programmer to branch to another part of the same method to continue execution—and the `break` and `continue` statements—that allow the programmer to short-circuit various aspects of the looping mechanism, either bypassing the loop test or the loop body. These mechanisms can be jarring for someone to encounter when reading the program. They're also jarring for a processor to encounter, and they often go against the natural momentum of the execution.

In C there is one other mechanism called the *long jump*. In this abstraction, the programmer first highlights a *target* for a jump, capturing the context of the machine at that point. Later, in the same routine—or in a child of that routine—the programmer can ask to resume execution at the target.

Here, in the application `divide.c`, we see how the long jump mechanism might be used:

```
#include "kernel/types.h"
#include "user/user.h"
#include "user/thread.h"
struct context target;

// Attempt to divide top by bottom.
int divide(int top, int bottom) {
  if (bottom == 0) longjmp(&target, 1); // complain
  return top / bottom;
}

int main(int argc, char **argv) {
  // grab two values from command line
  int a = atoi(argv[1]), b = atoi(argv[2]);

  if (setjmp(&target) == 0) {  // target context captured
    printf("%d\n",divide(a,b));
  } else {                     // target context restored; returned 1
    printf("You attempted to divide by zero.\n");
  }
  exit(0);
}
```

Ideally, when we run the program a couple of times, we get the following behavior:

```
$ divide 10 7
1
$ divide 10 2
5
$ divide 10 0
You attempted to divide by zero.
$
```

So, what's happening? The C library routine `setjmp` stores the state of the machine in the `target`. When called, `setjmp` always returns 0. Later, the programmer can choose to resurrect the state of the machine at the point of the `setjmp` with a `longjmp`. When the `longjmp` is called, it returns, but not from its own call, but from the call to `setjmp`! To distinguish between the two different returns from `setjmp`, `longjmp` takes a second parameter that describes the return value from setjmp. Typically this value is non-zero. Your first job this week is to build the mechanism that makes this work.

A Little Thread Library. The promise of multiprogramming, of course, is to support lots of concurrent executions. Typically, of course, these different *processes* have very little to do with each other. Indeed, a large part of our semester has been spent thinking carefully how these processes can be *isolated* from each other. Sometimes, however, we would like to have an abstraction where related *threads of execution* run concurrently, trying to collaborate on the solution to some problem. This week we will think about the construction of a very simple library that allows a single monitor process to clone itself in an attempt to perform more than one computation concurrently, or "at the same time".

Our library might support this application, `snap.c`:

```
#include "kernel/types.h"
#include "user/user.h"
#include "user/thread.h"
volatile int counter = 0;  // Why volatile?!?

void snap() {
    for (int i = 0; i < 3; i++) {
        printf("%d: snap!\n",++counter);
        yield();
    }
    fini();
}
void crackle() {
    for (int i = 0; i < 5; i++) {
        printf("%d: crackle!\n",++counter);
        yield();
    }
    fini();
}
void pop() {
    for (int i = 0; i < 4; i++) {
        printf("%d: pop!\n",++counter);
        yield();
    }
    fini();
}
int main(int argc, char **argv) {
    init();         // initialize thread system
    create(snap);   // create runnable threads
    create(pop);
    create(crackle);
    yield();        // let the threads go!
    exit(0);
}
```

Here, the `main` method manages three threads executing the routines `snap`, `crackle`, and `pop`. When the `main` method yields the machine, the threads execute concurrently. Each thread shares the memory of `main`, but each also maintains its own execution context that includes a stack and local variables. One possible output from the above program could be:

```
$ snap
1: snap!
2: pop!
3: crackle!
4: snap!
```

```
5: pop!
6: crackle!
7: snap!
8: pop!
9: crackle!
10: pop!
11: crackle!
12: crackle!
$
```

Notice that, over time, there were 3 snaps, 5 crackles, and 4 pops.

The Assignment. Finish the implementations of `setjmp`/`longjmp` and the `thread` library, as described above. The basis for this lab is found in the `lab6` repository (use your user id rather than `22xyz`):

```
$ git clone ssh://22xyz@lohani.cs.williams.edu/~cs432/22xyz/lab6.git
```

Make sure that you've read Chapter 7 from the book and have had a chance to peruse the code associated with context switching in the kernel.

This lab is composed of two main parts: the `setjmp` implementation and the thread library implementation. They are related, but can be attacked independently. In particular, I do not expect one to be used in the other.

The following tasks are useful for implementing `setjmp` and `longjmp` in `xv6`:

1. Develop a plan. Think about what must be stored in the `struct context` to allow the `setjmp` method to be restored by a call to `longjmp`. How do you remember where the program counter is? How is the stack preserved? Which registers need to be saved?
2. Given your plan, fill out the `struct context` in the file `setjmp.h`.
3. Modify the definition of `setjmp` in `setjmp.S` to return a zero. Remake the system and run. This will silence the persistent error messages from `divide`.
   Dear Mom: Today I wrote some assembly that had a positive impact!
4. Now, carefully write the part of `setjmp` that fills out the `struct context`. You might want to print (from within `main`) out some choice pieces of the saved state to see if they make sense. (Hint: `user/divide.asm` has the assembly associated with the application, along with addresses of each instruction.)
5. Implement the `longjmp` routine in `setjmp.S`. Make sure you remember that this routine takes *two* arguments and that you interpret them appropriately.
6. Push your code to the server. Call Mom.

In the next part of the assignment we'll be implementing the little thread library that supports activities like those found in the `snap` application. Do not make use of any aspect of the `setjmp`/`longjmp` code above; these projects are adjacent to suggest a heavy similarity, but not a shared implementation. Here is an approach to getting things done:

1. Peruse the code found in `snap.c` (the application) and `thread.h`, `thread.c`, and `twitch.S`. Some of these files are incomplete and will require you to add code.
2. Again, have a plan. Think about what must be stored in the `struct thread` to allow the suspension and resumption of individual threads. Notice that every thread has its own stack, but otherwise memory is shared between the threads. How do you remember where the program counter is? Every thread has its own stack. How is the dedicated stack preserved? Which registers need to be saved?
3. Given your plan, fill out the definition of `struct thread`, keeping in mind that you will have to access these fields using the assembly language thread context switcher, `twitch(old,new)`.
4. Write the assembly code for `twitch` in `twitch.S`.
5. Now, work on `thread.c`. Write `init`. Warning: this is your one chance to make sure you initialize everything. Make sure you understand the special nature of thread 0, the *monitor*, or `main`.

6. Write `create`. This routine is given a function that should be associated with a new thread, drawn from those marked `FREE`. Its context should be designed to start executing the function on the next `twitch` to this thread. Mark this new thread `RUNNABLE`. Return from `create` without running the thread.

7. Write `schedule`. This routine should find a process that can be run and perform a thread-switch—a *twitch*—to it. If there are no threads that can be run, `twitch` to the main thread.

8. Write `yield` and `fini`. The difference between the two is that `yield` is a temporary surrender of the CPU, while `fini` surrenders it permanently.

9. At this point, the `snap` application should run and generate output that is similar to what we saw above.

10. Push your work to the server for review and grading!

Thought Questions. Please think about the following questions in preparation for meeting in small groups:

1. What is your precise logic for identifying the makeup of the `struct context` for `setjmp`? Make sure your state-saving is both *necessary* and `sufficient`.

2. What is the relationship between the stack of a call to `setjmp` and its corresponding `longjmp`?

3. How might you go about making sure that each thread *implicitly* calls `fini` as it finishes? (Hint: something similar happens with `main` in a program on more standard forms of unix—the call to `exit` is optional, but it is always performed.)

4. What is the relationship between `setjmp`, `longjmp`, and `twitch`?

⋆