In-class: User-level threads

In this assignment you will complete a simple user-level thread package by implementing the code to perform context switching between threads.

Please feel free to collaborate with others on these exercises.

Switching threads

Download uthread.c and uthread_switch.S into your xv6 directory. Make sure `uthread_switch.S` ends with `.S`, not `.s`. Add the following rule to the xv6 Makefile after the _forktest rule:

```
_uthread: uthread.o uthread_switch.o
	$(LD) $(LDFLAGS) -N -e main -Ttext 0 -o _uthread uthread.o uthread_switch.o $(ULIB)
	$(OBJDUMP) -S _uthread > uthread.asm
```

Make sure that the blank space at the start of each line is a tab, not spaces.

Add `_uthread` in the Makefile to the list of user programs defined by UPROGS.

Run xv6, and run `uthread` from the xv6 shell, and you will observe a panic.

Your job is to complete `thread_switch.S`, so that you see output similar to this (make sure to run with CPUS=1):

```
~/classes/6828/xv6$ make CPUS=1 qemu-nox
dd if=/dev/zero of=xv6.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes transferred in 0.037167 secs (137756344 bytes/sec)
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes transferred in 0.000026 secs (19701685 bytes/sec)
dd if=kernel of=xv6.img seek=1 conv=notrunc
307+1 records in
307+1 records out
157319 bytes transferred in 0.003590 secs (43820143 bytes/sec)
qemu -nographic -hdb fs.img xv6.img -smp 1 -m 512
Could not open option rom 'sgabios.bin': No such file or directory
xv6...
cpu0: starting
init: starting sh
$ uthread
my thread running
my thread 0x2A30
my thread running
my thread 0x4A40
my thread 0x2A30
my thread 0x4A40
my thread 0x2A30
my thread 0x4A40
....
```

Before jumping into `uthread_switch.S`, first understand how `uthread.c` uses `thread_switch`. `uthread.c` has two global variables `current_thread` and `next_thread`. Each is a pointer to a `thread` structure. The thread structure has a stack for a thread and a saved stack pointer (`sp`, which points into the thread's stack). The job of `uthread_switch` is to save the current thread state into the structure pointed to by `current_thread`, restore `next_thread`'s state, and make `current_thread` point to where `next_thread` was pointing to, so that when `uthread_switch` returns `next_thread` is running and is the `current_thread`.

You should study `thread_create`, which sets up the initial stack for a new thread. It provides you good hint what `thread_switch` should do. In particular, note that `thread_create` allocates 32 bytes of space for saving registers. That is all x86 registers, and the intent is that `thread_switch` use the assembly instructions `popal` and `pushal` to restore and save registers.

To write the assembly in `thread_switch`, you need to know how the C compiler lays out `struct thread` in memory, which is as follows:

```
    --------------------
    | 4 bytes for state|
    --------------------
    | stack size bytes |
    | for stack        |
    --------------------
    | 4 bytes for sp   |
    --------------------  <--- current_thread
         ......

         ......
    --------------------
    | 4 bytes for state|
    --------------------
    | stack size bytes |
    | for stack        |
    --------------------
    | 4 bytes for sp   |
    --------------------  <--- next_thread
```

The variables `next_thread` and `current_thread` each contain the address of a `struct thread`.

To write the `sp` field of the struct that `current_thread` points to, you should write assembly like this:

```
   movl current_thread, %eax
   movl %esp, (%eax)
```

This saves `%esp` in `current_thread->sp`. This works because `sp` is at offset 0 in the struct. You can study the assembly the compiler generates for `uthread.c` by looking at `uthread.asm`.

To test your code it might be helpful to single step through your `thread_switch` using `gdb`. You can do this as follows:

```
(gdb) symbol-file _uthread
Load new symbol table from "/Users/kaashoek/classes/6828/xv6/_uthread"? (y or n) y
Reading symbols from /Users/kaashoek/classes/6828/xv6/_uthread...done.
(gdb) b thread_switch
Breakpoint 1 at 0x204: file uthread_switch.S, line 9.
(gdb) c
```

In your xv6 shell, type "uthread", and gdb will break at `thread_switch`. Now you can type commands like the following to inspect the state of `uthread`:

```
(gdb) print /x *next_thread
$3 = {sp = 0x49e8, stack = {0x0 , 0x66, 0x1, 0x0, 0x0},
  state = 0x1}
```

What address is `0x166`, which sits on the top of the stack of `next_thread`?

Submit: your modified uthread_switch.S

Challenge exercises

The user-level thread package interacts badly with the operating system in several ways. For example, if one user-level thread blocks in a single call, another user-level thread won't run, because the user-level threads scheduler doesn't know that one of its threads has been descheduled by the xv6 scheduler. As another example, two user-level threads will not run concurrently on different cores, because the xv6 scheduler isn't aware that there are multiple threads that could run in parallel. Note that if two user-level threads were to run truly in parallel, this implementation won't work because of several races (e.g., two threads on different processors could call `thread_schedule` concurrently, select the same runnable thread, and both run it on different processors.)

There are several ways of addressing these problems. One is using scheduler activations and another is to use one kernel thread per user-level thread (as Linux kernels do). Implement one of these ways in xv6.

Add locks, condition variables, barriers, etc. to your thread package.