# Homework: 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. Submit your solutions before the beginning of the next lecture to the submission web site.

## 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, then run `uthread` from the xv6 shell. The xv6 kernel will print an error message about `uthread` encountering a page fault.

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 (5.1 MB, 4.9 MiB) copied, 0.0190287 s, 269 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 7.2168e-05 s, 7.1 MB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
291+1 records in
291+1 records out
149040 bytes (149 kB, 146 KiB) copied, 0.000528827 s, 282 MB/s
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 1 -m 512
xv6...
cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ uthread
my thread running
my thread 0x2D68
my thread running
my thread 0x4D70
my thread 0x2D68
my thread 0x4D70
my thread 0x2D68
my thread 0x4D70
my thread 0x2D68
...
```

`uthread` creates two threads and switches back and forth between them. Each thread prints "my thread ..." and then yields to give the other thread a chance to run.

To observe the above output, you need to complete `thread_switch.S`, but 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 hints about what `thread_switch` should do. The intent is that `thread_switch` use the assembly instructions `popal` and `pushal` to restore and save all eight x86 registers. Note that `thread_create` simulates eight pushed registers (32 bytes) on a new thread's stack.

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 get started in this way:

```
(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)
```

The breakpoint may (or may not) be triggered before you even run `uthread`. How could that happen?

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

```
(gdb) p/x next_thread->sp
$4 = 0x4ae8
(gdb) x/9x next_thread->sp
0x4ae8 :	0x00000000	0x00000000	0x00000000	0x00000000
0x4af8 :	0x00000000	0x00000000	0x00000000	0x00000000
0x4b08 :	0x000000d8
```

What address is `0xd8`, which sits on the top of the stack of `next_thread`?

> **Submit**: your modified uthread_switch.S

## Optional challenges

The user-level thread package interacts badly with the operating system in several ways. For example, if one user-level thread blocks in a system 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.