CSE 451: Operating Systems

Section 4

Project 2 Intro; Threads
Congratulations, you’re all kernel hackers now!

We’re going to give you a break and have you do some userspace work 😊
Project 2: user-level threads

Part A: due Wednesday, May 2 at 11:59pm
- Implement part of a user thread library
- Add synchronization primitives
- Solve a synchronization problem

Part B: due Friday, May 18 at 11:59pm
- Implement a multithreaded web server
- Add preemption
- Get some results and write a (small) report
Project 2 notes

• Start EARLY!
  • It’s looooonng
  • Read the assignment carefully
  • Read it again
  • Understand the skeleton code

• Use the same groups as for project 1
Project 2 tips

* Understand what the provided code does for you

* Division of work
  * Part 3 can be completed without parts 1 and 2

* More tools
  * ddd
  * (Or just gdb if you’re not a fan of GUIs)
Simplethreads

We give you:
- Skeleton functions for thread interface
- Machine-specific code (x86, i386, PowerPC)
  - Support for creating new stacks
  - Support for saving regs/switching stacks
- A queue data structure (why?)
- Very simple test programs
  - You should write more, and include them in the turnin
- A single-threaded web server
Simplethreads code structure

- test/*.c
- Web server (web/sioux.c)
- Other apps
- include/sthread.h
- You write this
  - lib/sthread_user.c
  - lib/sthread_queue.c
  - lib/sthread_ctx.c
  - lib/sthread_preempt.c
  - lib/sthread_switch.S
    - sthread_switch_i386.h
    - sthread_switch_powerpc.h
Pthreads

* Pthreads (POSIX threads) is a preemptive, kernel-level thread library
* Simplethreads is similar to Pthreads
* Project 2: compare your implementation against Pthreads
  * ./configure --with-pthreads
Thread operations

What functions do we need for a userspace thread library?
Simplethreads API

void sthread_init()
  ✴ Initialize the whole system
sthread_t sthread_create(func start_func,
    void *arg)
  ✴ Create a new thread and make it runnable
void sthread_yield()
  ✴ Give up the CPU
void sthread_exit(void *ret)
  ✴ Exit current thread
void* sthread_join(sthread_t t)
  ✴ Wait for specified thread to exit
Simplethreads internals

Structure of the TCB:

```c
struct _sthread {
    sthread_ctx_t *saved_ctx;
    /**<
    * Add your fields to the thread
    * data structure here.
    */
};
```
Sample multithreaded program

* (this slide and next – see test-create.c)

```c
void *thread_start(void *arg) {
    if (arg) {
        printf("in thread_start, arg = %p\n", arg);
    }
    return 0;
}
...
```
Sample multithreaded program

```c
int main(int argc, char *argv[]) {
    sthread_init();
    for(i = 0; i < 3; i++) {
        if (stthread_create(thread_start,
                            (void *)&i) == NULL) {
            printf("sthread_create failed\n");
            exit(1);
        }
    }
    // needs to be called multiple times
    sthread_yield();
    printf("back in main\n");
    return 0;
}
```

4/19/12
Managing contexts

*(Provided for you in project 2)*

Thread *context* = thread stack + stack pointer

```c
stthread_new_ctx(func_to_run)
```

* creates a new thread context that can be switched to

```c
stthread_free_ctx(some_old_ctx)
```

* Deletes the supplied context

```c
stthread_switch(oldctx, newctx)
```

* Puts current context into oldctx

* Takes newctx and makes it current
How sthread\_switch works

Xsthread\_switch:
  (push all regs)
  movq \%rsp,(\%rax)
  movq \%rdx,\%rsp
  (pop all regs)
  ret

Thread 1 TCB
  ...
  SP

Thread 2 TCB
  ...
  SP

Thread 1 running

Thread 2 ready

Want to switch to thread 2…
Push old context

Xstthread_switch:
  (push all regs)
  movq %rsp, (%rax)
  movq %rdx, %rsp
  (pop all regs)
  ret

Thread 1 TCB
  ... SP

Thread 2 TCB
  ... SP

CPU
  RSP

Thread 1 running

Thread 2 TCB
  ... SP

Thread 2 registers

Thread 1 registers

Thread 1 regs

Thread 2 ready
Save old stack pointer

Xsthd_thread_switch:
  (push all regs)
  movq %rsp,(%rax)
  movq %rdx,%rsp
  (pop all regs)
ret

Thread 1 TCB
... SP

Thread 2 TCB
... SP

CPU

Thread 1 TCB
... SP

Thread 1 running

Thread 2 TCB
... SP

Thread 2 ready
Change stack pointers

Xstthread_switch:
  (push all regs)
movq %rsp,(%rax)
movq %rdx,%rsp
  (pop all regs)
ret

Thread 1 TCB
...  SP

Thread 2 TCB
...  SP

Thread 1 ready
Thread 2 running
Pop off new context

Xstthread_switch:
  (push all regs)
  movq %rsp,(%rax)
  movq %rdx,%rsp
  (pop all regs)
ret
Xstthread_switch:
    (push all regs)
    movq %rsp,(%rax)
    movq %rdx,%rsp
    (pop all regs)
    ret

- What got switched?
  - RSP
  - PC (how?)
  - Other registers

Thread 1 TCB
  ... SP

Thread 2 TCB
  ... SP

Thread 1 ready

Thread 2 running
Adjusting the PC

- **ret** pops off the new return address!

---

**Thread 1 TCB**

... SP

**Thread 2 TCB**

... SP

**Thread 1 registers**

ra=0x400

**Thread 1 (stopped):**

sthread_switch(t1,t2);

0x400: printf(“test 1”);

**Thread 2 (running):**

sthread_switch(t2,...);

0x800: printf(“test 2”);
Thread joining

With Pthreads (and Sthreads):
* Master thread calls `join` on worker thread
* Join blocks until worker thread exits.
* Join returns the return value of the worker thread.

![Diagram of thread joining process](image-url)
The need for synchronization

※ Thread safety:
※ An application's ability to execute multiple threads simultaneously without "clobbering" shared data or creating "race" conditions
Synchronization primitives: mutexes

stthread_mutex_t sthread_mutex_init()
void sthread_mutex_free(sthread_mutex_t lock)

void sthread_mutex_lock(sthread_mutex_t lock)
    /* When returns, thread is guaranteed to acquire lock */
void sthread_mutex_unlock(
    sthread_mutex_t lock)
Synchronization primitives: condition variables

```c
sthread_cond_t sthread_cond_init()
void sthread_cond_free(sthread_cond_t cond)
```

```c
void sthread_cond_signal(sthread_cond_t cond)
   ✴ Wake-up one waiting thread, if any
void sthread_cond_broadcast(sthread_cond_t cond)
   ✴ Wake-up all waiting threads, if any
void sthread_cond_wait(sthread_cond_t cond, sthread_mutex_t lock)
   ✴ Wait for given condition variable
   ✴ Returning thread is guaranteed to hold the lock
```
Things to think about

* How do you create a thread?
  * How do you pass arguments to the thread’s start function?
    * Function pointer passed to sthread_new_ctx() doesn’t take any arguments

* How do you deal with the initial (main) thread?

* How do you block a thread?
Things to think about

- When and how do you reclaim resources for a terminated thread?
  - Can a thread free its stack itself?

- Where does sthread_switch return?

- Who and when should call sthread_switch?

- What should be in struct _stthread_mutex, struct _stthread_cond?
Things to think about

∗ Working with synchronization: When does it make sense to disable interrupts?
∗ Which actions are atomic at the application level versus at the thread level?

∗ When using forkbomb, run “ulimit -Su 64” to limit the number of processes/threads
∗ Allows you to log in from another session even if you hit the above limit
∗ Add it to your .bash_profile so it happens automatically