CSE 451: Operating Systems

Simplethreads
Simplethreads

- We give you:
  - Skeleton functions for thread interface
  - Machine-specific code (x86, i386)
    - 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
Simplethreads code structure

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

```c
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
```
**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;
}

...
```
int main(int argc, char *argv[]) {
    sthread_init();
    for(i = 0; i < 3; i++) {
        if (sthread_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;
}
Managing contexts

* (Provided for you in project 2)

* Thread context = thread stack + stack pointer

\[ \text{sthread}_\text{new}_\text{ctx}(\text{func}_\text{to}_\text{run}) \]

  * creates a new thread context that can be switched to

\[ \text{sthread}_\text{free}_\text{ctx}(\text{some}_\text{old}_\text{ctx}) \]

  * Deletes the supplied context

\[ \text{sthread}_\text{switch}(\text{oldctx}, \text{newctx}) \]

  * Puts current context into oldctx

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

\textbf{Xsthread\_switch:}
\begin{itemize}
  \item (push all regs)
  \item \texttt{movq} \%rsp,\%rax
  \item \texttt{movq} \%rdx,\%rsp
  \item (pop all regs)
  \item \texttt{ret}
\end{itemize}

\begin{itemize}
  \item Thread 1 TCB
  \begin{itemize}
    \item \ldots
    \item SP
  \end{itemize}
  \item Thread 2 TCB
  \begin{itemize}
    \item \ldots
    \item SP
  \end{itemize}
\end{itemize}

CPU

Thread 1 running

Thread 2 registers

Want to switch to thread 2…

Thread 1 regs

RSP

Thread 2 ready
Push old context

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

Thread 1 TCB
... SP

Thread 1 running
Thread 1 registers

Thread 2 TCB
... SP

Thread 2 ready
Thread 2 registers
Save old stack pointer

Xsthread_switch:
(push all regs)

```c
movq %rsp, (%rax)
movq %rdx, %rsp
(pop all regs)
ret
```

Thread 1 TCB

Thread 2 TCB

Thread 1 running

Thread 2 ready

CPU

Thread 1 regs

Thread 2 registers

Thread 1 registers
Change stack pointers

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

Thread 1 TCB
...
SP
Thread 1 registers
Thread 1 ready
Thread 2 TCB
...
SP
Thread 2 registers
Thread 2 running

CPU
RSP
Thread 1 regs
4/19/12
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 registers

Thread 1 ready

Thread 2 regs

Thread 2 running

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

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

CPU

Thread 1 TCB
... SP

Thread 1 ready

Thread 2 TCB
... SP

Thread 2 running

4/19/12
Adjusting the PC

- \textbf{ret} pops off the new return address!

Thread 1 TCB
... SP
Thread 2 TCB
... SP

Thread 1 (stopped):
\texttt{sthread\_switch(t1,t2)};
0x400: \texttt{printf(“test 1”)};

Thread 2 (running):
\texttt{sthread\_switch(t2,...)};
0x800: \texttt{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.
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

```c
sthread_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)

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 _sthread_mutex, struct _sthread_cond?
/* Start preemption - func will be called every period microseconds */

void sthread_preemption_init
  (sthread_ctx_start_func_t func,
   int period);

/* Turns interrupts on (LOW) or off (HIGH) * Returns the last state of the * interrupts */

int splx(int splval);
/* atomic_test_and_set - using the native 
 * compare and exchange on the Intel x86. 
 */

* Example usage:
*   lock_t lock;
*   while(atomic_test_and_set(&lock))
*     {} // spin
*   _critical section_
*   atomic_clear(&lock);
*/

int atomic_test_and_set(lock_t *l);
void atomic_clear(lock_t *l);
What you need to do

* Add a call to `sthread_preemption_init()` as the last line in your `sthread_user_init()` function
* `sthread_preemption_init()` takes a pointer to a function that will be called on each timer interrupt
* This function should cause thread scheduler to switch to a different thread!
What you need to do

- Add synchronization to *critical sections* in thread management routines
  - Think: what would happen if the code was interrupted at this point?
    - Would it resume later with no problems?
    - Could the interrupting code mess with any variables that this code is currently using?
  - Don’t have to worry about simplethreads code that you didn’t write (i.e. sthread_switch): already done for you
What you need to do

- Before doing a context switch, interrupts should be disabled to avoid preemption. How can they be reenabled after the switch?
  - Hint: Think of the possible execution paths
Interrupt disabling

Non-thread-safe

/* returns next thread * on the ready queue */

#define splx(x) x

sthread_t

sthread_user_next() {
    sthread_t next;
    next = sthread_dequeue (ready_q);
    if (next == NULL)
        exit(0);
    return next;
}

Thread-safe

sthread_t

sthread_user_next() {
    sthread_t next;
    int old = splx(HIGH);
    next = sthread_dequeue (ready_q);
    splx(old);
    if (next == NULL)
        exit(0);
    return next;
}
Why do we call `splx(old)` after dequeuing instead of just `splx(LOW)`?
Atomic locking

So what is atomic_test_and_set() for?
- Primarily to implement higher-level synchronization primitives (mutexes, CVs)

One way to think about preemption-safe thread library:
- Disable/enable interrupts in “library” context
- Use atomic locking in “application” context
Race conditions and testing

* How can you test your preemption code?
* How can you know that you’ve found all of the critical sections?
Part 5: report

- Covers all parts of project 2
- Discuss your design decisions. In detail. PLEASE!