CSE451 – Section 6

Project 2 (parts 4,5,6)
Short recap of a few topics for midterm

Project 2, part 4 - web server

- web/sioux.c – singlethreaded web server
  - Read in command line args, run the web server loop
- web/sioux_run.c – the webserver loop
  - Open a socket to listen for connections (`listen`)
  - Wait for a connection (`accept`)
  - Handle it
    - Parse the HTTP request
    - Find and read the requested file (www root is ./docs)
    - Send the file back
    - Close the connection

What you need to do

- Make the web server multithreaded
  - Create a thread pool
    - A bunch of threads waiting for work
    - Number of threads = command-line arg
  - Wait for a connection
  - Find an available thread to handle connection
    - Current request waits if all threads busy
  - Once a thread grabs onto connection, it uses the same processing code as before.

Hints

- Each connection is identified by a socket returned by `accept`
  - Which is just an int
  - Simple management of connections among threads
- Threads should sleep while waiting for a new connection
  - Condition variables are perfect for this
- Don't forget to protect any global variables
  - Use part 2 mutexes, CVs
- Develop + test with pthreads initially
- Mostly modify sioux_run.c and/or your own files
- Stick to the sthread.h interface!

Part 5 - Preemption

- What we give you:
  - Timer interrupts
  - Primitives to turn interrupts on and off
  - Synch primitives atomic_test_and_set, atomic_clear
- What you have to do:
  - Add code that will run every time a timer interrupt is generated
  - Add synchronization to your part 1 and part 2 implementations so that everything works with preemptive thread scheduling
- WARNING: The code that we have provided to support preemption works correctly *only* on the x86 architecture! Do not attempt using this portion of the assignment on a Mac or other non-x86 architecture!

What we give you:

- sthread_preempt.h (implemented in .c):

```
/* start preemption - func will be called every period microseconds */
void sthread_preemption_init(sthread_ctx_start_func_t func, int period);

/* Turns interrupts ON and off
 * Returns the last state of the inturrupts
 * LOW = interrupts ON
 * HIGH = interrupts OFF
 */
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);
```

Signals

- Asynchronous notification mechanism
- Every process has a signal handler table
- When a signal is sent to a process, OS interrupts that process and calls the handler function registered for that signal
- A process can:
  - Override the default signal handlers using the *signal()* system call (or *sigaction()*)
  - Enable / disable signals via *sigaddset()* / *sigdelset()*
- To send a signal, use *kill(N)*, where N is signal #
  - E.g.: SIGINT (CTRL-C), SIGQUIT (CTRL-\), SIGKILL* (kill -9), SIGSEGV, SIGFPE, SIGALRM, SIGIO, SIGUSR1

* Handler for SIGKILL cannot be replaced by process.

What you need to do

- Add a call to sthread_preemption_init() as the last line in your sthread_user_init() function.
  - init specifies a function that is called on each timer interrupt (done for you, but instructive to look at!)
  - This func should cause thread scheduler to switch to a different thread
- Add synchronization to thread management routines
  - Where are the critical sections from part 1 and 2?

Preemption + Critical Sections

- Safest way: disable interrupts before critical sections
- Example:

NON-threadsafe soln

```
/*returns the next thread on the ready queue*/
sthread_t sthread_user_next() {
  sthread_t next;
  next = sthread_dequeue(ready_q);
  if (next == NULL)
    exit(0);
  return next;
}
```

threadsafe soln

```
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;
}
```

- Where is the critical section? Why?

Part 6 – Report

- Design discussion & functionality
  - Make it short
- Results
  - Run a few experiments with the new webserver
    - Use given web benchmark: /cse451/projects/webclient
  - Present results in a *graphical easy-to-understand* form.
  - Explain
    - Are the results what you expected?
    - Try to justify any discrepancies you see

Project 2 questions?

Midterm

- Review lecture tomorrow

Top topics:

- Processes
- Scheduling
- Synchronization
- Virtual memory

Scheduling review

- FIFO:
  - \+ simple
  - \- short jobs can get stuck behind long ones; poor I/O device utilization
- RR:
  - \+ better for short jobs
  - \- hard to select right time slice
  - \- poor turnaround time when jobs are the same length
- SJF:
  - \+ minimal average waiting time
  - \- hard to predict the next CPU burst length
  - \- unfair
- Multi-level feedback:
  - \+ approximate SJF (gives preference to short jobs)
  - \+ establishes the nature of a process quickly
  - \- unfair to long running jobs

A simple scheduling problem

Thread	Arrival Time	Burst Time
A	0	10
B	1	5
C	3	2

- FIFO Turnaround time:
- FIFO Waiting Time:

A simple scheduling problem

Thread	Arrival Time	Burst Time
A	0	10
B	1	5
C	3	2

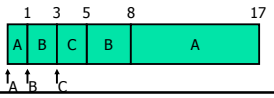
- FIFO Turnaround Time:
  - A: (10-0) = 10
  - B: (15-1) = 14
  - C: (17-3) = 14
  - (10+14+14)/3 = 12.66
- FIFO Waiting Time:
  - A: 0
  - B: (10-1) = 9
  - C: (15-3) = 12
  - (0+9+12)/3 = 7

A simple scheduling problem

- What about SJF?

Thread	Arrival Time	Burst Time
A	0	10
B	1	5
C	3	2

- Ave Turnaround Time:
  - B: 8-1 = 7
  - C: 5-3 = 2
  - A: 17-0 = 17
  - (17+2+7)/3 = 8.67
- Ave Waiting Time:
  - B: 2
  - C: 0
  - A: 2+2+3 = 7
  - (2+0+7)/3 = 3



Priority Inversion

- Have three processes
  - P1:Highest priority; P2:Medium; P3:Lowest
  - P1 and P3 have this code:

```
P(mutex);
critical section;
V(mutex);
```

  - P2 is a long-running task
- P3 acquires mutex; preempted
- P1 tries to acquire mutex; blocks
- P2 enters the system at medium priority; runs
- P3 never gets to run; P1 never gets to run!!
- This happened on Mars Pathfinder in 1997!
- Solutions?

Deadlock-related questions

- Q1: Can there be a deadlock with only process?
- Q2: Given two threads, what sequence of calls to transfer(...) causes the following to deadlock?

```
/* transfer x dollars from a to b */
void transfer(account *a, account *b, int x)
  P(a->sema);
  P(b->sema);
  a->balance += x;
  b->balance -= x;
  V(b->sema);
  V(a->sema);
```