Project 1
Non-Preemptive Multitasking

Ege Mihmanli

Department of Computer Science
Cornell University

September 2, 2016
First things first

• Welcome to PortOS

• Project 1 is already released!

• Due on September 19th at 11:59pm
GitHub

• We are using github.coecis.cornell.edu

• Sign in with Cornell credentials, i.e netID and password

• Projects released and submitted on GitHub
Goals

• Ramp up for C and PortOS

• Learn how threading works

• Implement synchronization primitives

• Large project ➔ bad coding style WILL bite later
Project Overview

Queue  Semaphore  Minithreads
Project Overview

Queue

Semaphore

Minithreads
Queues

• Simple FIFO Queue

• Interface described in queue.h

• Use a linked list under the hood

• Prepend, append and dequeue must be $O(1)$
Project Overview

Queue  Semaphore  Minithreads
What is a semaphore?

• Pillar of concurrent programming

• Actually, just another data structure

• Keeps a count

• Blocks/wakes up threads depending on situation
This is a semaphore
Let’s make the analogy work

• Students $\rightarrow$ Threads

• Bouncers $\rightarrow$ Semaphore

• Legal max capacity $\rightarrow$ Count

• Room space $\rightarrow$ Shared resource

• Line $\rightarrow$ Blocked threads
Concurrency 101

- Client decides how many threads can hold a semaphore (count)

- Counter is incremented/decremented atomically
  - \( P \downarrow \text{ & } V \uparrow \)

- P blocks if count == 0

- V wakes up blocked thread if count == 0
career_fair.c

take_shower();
get_dressed();
sweat_a_lot_on_your_way_over();
semaphore_p(); //attempt to walk in
talk_to_employers();
exaggerate_resume();
get_swag();
semaphore_v(); //walk out
complain_about_career_fair();
Project Overview

Queue  Semaphore  Minithreads
Minithreads

Process

User Threads

Scheduler

Kernel Thread
Scheduler

- First come first serve

- Just yield CPU to thread at the head of queue

- Expect this to get more complicated in Project 2

- Code style matters
Minithreads

• What we call threads in PortOS

• Majority of the project

• Will need a Thread Control Block
  • Stack top pointer
  • stack base pointer
  • thread ID
  • Anything else you want
Useful functions

• We provide some functions we found useful

• Allocate stack → minithread_allocate_stack

• Initialize stack → minithread_initialize_stack

• Switching between threads → minithread_switch

• Make sure to read machineprimitives.h
minithread switch

old_thread_sp  esp  new_thread_sp
minithread_switch

old_thread_sp   esp   new_thread_sp

state

state
minithread_switch

old_thread_sp  esp  new_thread_sp

state  state
minithread_switch

old_thread_sp  esp  new_thread_sp

state
Bootstrapping

```c
void minithread_system_initialize
```

- This bootstraps the system
- Use it to initialize queues, semaphores, global variables or data structures
- You will add more in projects to come
Bootstrapping

• What happens when there is no user thread left?
  
    • System shouldn’t crash! It’s an operating system
  
    • Run the \textit{idle thread}
  
    • Only place where polling is OK!
  
• In our case, the kernel thread is the idle thread
  
• No need to allocate stack for it
Being Non-Preemptive

• What happens when a user thread runs forever?
  • In P1, we let it be!

• Assume that all threads are good and voluntarily yield
  • Threads yield by calling minithread_yield
Life of a minithread

1. `minithread_fork(body_proc, args)`  
   Thread starts, `body_proc(args)` is called.

2. `minithread_yield()`  
   Thread gives up control of CPU

3. Another thread yields, and this thread resumes execution again. Control resumes from instruction after `minithread_yield()`.

4. Thread terminates by executing return from within `body_proc()`.

5. When `body_proc` returns, epilogue code `final_proc(args)` is immediately called. This code should wake up the cleanup thread to free the stack and TCB. A context switch should be made to the next runnable thread.
Testing

• We supply a few primitive tests
  • Use it to see how minithreads work

• Sieve and buffer are good stress tests

• Remove **ALL** of your print statements and dead code before submission!
Coding Style

• Avoid unnecessary polling

```c
while (condition == False) {
    minithread_yield();
}
```

• Unnecessary context switches are bad for you

• Check for NULL arguments! (malloc can return NULL)
Commenting

- Helps us understand your code
- Helps you understand your code
- Helps you notice bugs
- Helps us give partial credit for buggy code
- Notice all the “helps”? Commenting is good!
Coding Style

• Naming convention is important

  • Underscores to delimit words:
    • minithread_switch
    • number_of_eges

  • Constants in ALL_CAPS
Coding in C

• Can’t really say “I know C” without mastering pointers

```c
int *int_ptr = (int*) malloc(sizeof(int));

int_ptr = 5;
```

• What does this do?
Files you need to change

• queue.c/h

• synch.c/h

• minithread.c/h

• Important: you don’t have to change header files!

• DO NOT CHANGE ANY OTHER FILE
```c
int thread3(int* arg) {
    printf("Thread 3.\n");
    return 0;
}

int thread2(int* arg) {
    minithread_fork(thread3, NULL);
    printf("Thread 2.\n");
    minithread_yield();
    return 0;
}

int thread1(int* arg) {
    minithread_fork(thread2, NULL);
    printf("Thread 1.\n");
    minithread_yield();
    minithread_yield();
    return 0;
}

int main(int argc, char * argv[]) {
    minithread_system_initialize(thread1, NULL);
    return 0;
}
```