Preemption and Networking

415 Project 2

Emin Gun Sirer

What you need to do

- Add Preemption
- Add Alarms
- Add minithread_sleep
- Add datagram networking

Interrupts

- In project 2, we have interrupts
  - Much more realistic machine model
- Two kinds
  - Clock: periodic interrupt; call clock_initialize to start
  - Network: when packets arrive; call minimsg_initialize
- Interrupts behave like they do on native hardware

How it works

- When you start, interrupts are disabled
- You need to initialize the clock device

```
#define PERIOD 50*MILLISECOND
typedef void (*clock_handler_t)();
void minithread_clock_init(clock_handler_t h)
```

- Clock frequency defined in clock.h
  - Initially set it to something like 5 seconds…
- Once the clock device is initialized, interrupts are still disabled

Enabling interrupts

- Turn interrupts on or off with:

```
interrupt_level_t set_interrupt_level(interrupt_level_t newlevel )
```

- For example:

```
set_interrupt_level(ENABLED);
```

- You will now start getting an interrupt every 5 seconds
- The interrupts arrive on the stack of whatever thread happens to be currently executing

Interrupt Stack

- Whatever the thread used to be doing is interrupted
- Old state is saved onto the stack
- Your handler is called
- If your handler returns, the old state is resumed

Preemption

- Everything that you run inside your interrupt handler will execute in the context of the interrupted thread
- What would happen if the interrupt handler yields ?

Interrupt Handlers

- You cannot do anything you want inside an interrupt handler
- You can’t take too long
  - it takes CPU time away from real computation,
  - if you take far too long, you will get a second clock interrupt
- You cannot use spin locks
  - The thread you interrupted may be holding the lock you are spinning on
  - Consequently, you’ll spin forever and the machine will hang
- You cannot block (I.e. call P)
  - It’ll block the thread the interrupt arrived on
- You CAN signal (I.e. V) other threads though

Disabling Interrupts

- At critical points in your **system** code, you may not want to take interrupts
  - E.g. while manipulating the run queue
- You can disable interrupts for short periods of time
  - Make sure you reenable them properly
  - On all paths, back to their prior value

```
Oldlevel = set_interrupt_level(DISABLED);
        … critical code …
set_interrupt_level(Oldlevel);
```

Make sure you do not accidentally execute application code with interrupts disabled

Preemption Testing

- Once you have implemented preemption:
  - Reduce your quantum to ~100 ms.
  - No printfs, no unnecessary tasks inside the clock interrupt handler
- Your FCFS scheduler became RR
  - Optional: Implement multi-level queue scheduling
- Run the idle thread only when the system is truly idle

Alarms

- Often, you need to schedule something to happen in the future
  - E.g. Wake me up in 30 ms.
  - E.g. If my network packet is not acknowledged by the remote side, resend it
- You need to implement an alarm facility where people can register functions to be executed in the future
- You need to keep track of time
- You then need to call the alarm functions when they expire

Alarm Interface

- Two calls:

```
int register_alarm(int delay, void (*func)(void*), void *arg);
void deregister_alarm(int alarmid);
```

- Keep track of how many ticks have elapsed
- Execute the given function when enough ticks have gone by
- Assume that the functions are interrupt safe, I.e. you can call them from within the interrupt handler

Sleep

- Now implement a call by which threads can sleep for a specified amount of time

  minithread_sleep_with_timeout(int timeout)
- You need to take the calling thread out of the run queue, have it wait someplace until timeout
- Semaphores can help here
  - Make sure that if more than one thread calls minithread_sleep, they each sleep the right amount of time

Networking

- You need to implement a datagram protocol
- We provide a packet send primitive and a network interrupt when a packet arrives
- You need to place the appropriate header on the way out
- Perform the appropriate decoding on the way in
- Alert the right thread waiting to receive packets
- Buffer packets if no thread is currently waiting

Ports

- Ports are connection endpoints
  - How you name where your packet ought to go
  - How you name which packets you want to listen to
- Two kinds:
  - Local: On the local machine
  - Remote: Bound to a particular remote host
- Internally, each port needs a unique id, an incoming packet queue, and necessary synchronization variables

Network Operations

- On the outgoing path, you need to define a packet header
- You need to attach just enough information to the header so you can decode it on the receive side
  - E.g. id of the destination port

Network Receive Path

- The network interrupt handler gets called with a pointer to a packet
- Need to look inside, attach it to the right port queue, alert the threads that are waiting for data
  - You need to keep track of all ports in the system
  - Can simply use a big array to hold all of them, it's ok to support only a small (<64K) number of ports at a given time
- Datagrams are unreliable, so some packets may be lost
  - You don't NEED to implement timeouts and retransmits
  - But they are not too hard with alarms and semaphores

Cleanup

- As always, you should not leak memory
- Ports should be cleaned up entirely when destroyed
  - Any unclaimed data should be freed
- Same goes for alarms

Testing

- Test preemption with your pokemon implementation and other tests from Project 1
- Use network[1-6].c to test your networking code
- The interface is rich, the code shows how it can be used
- Makefile : change the MAIN variable to indicate which main() you want to link against

Conclusions

- After this project, you will have a mostly functional prototype OS
  - Threads
  - Synchronization
  - Networking
- Built on top of a realistic machine model
- For help: cs414@cs.cornell.edu