Project 3
Supplemental Lecture

Joe Mongeluzzi
Jason Zhao
Cornell CS 4411, October 12, 2012
Today’s Lecture

- Administrative Information
- Common mistakes on Project 2
- Project 3 FAQ
- Discussion
Project 2

Administrative Information

- Project 2 is being graded
- See a TA if you need help fixing up your P2 before working on P3
- Project 3 deadline is October 19th, 11:59 PM.
Project 2 Common Errors

- Same semaphore used both with and without interrupts disabled (e.g. clean semaphore)
- Interrupt handler decreases alarm count for all alarms
- Only one alarm fired per clock interrupt, even if multiple are ready
- Sleep not implemented using semaphore
- A thread should run again immediately if it is the only thread of its level and it yields (and the level to schedule from hasn’t changed)
- Memory leaks: sleep semaphores, alarm structs
Network header generation

- Having a common header format will be fun later on.
- Use the pack and unpack functions that we provide.
  - Do not use a simple byte copy of the network address, this is incorrect.
  - Port numbers may be stored as ints in your program but must be converted to unsigned shorts when packing.
  - Reminder: do not pack port numbers as ints!
- The protocol field will be useful in the next project.
  - Set the protocol char field to PROTOCOL_MINIDATAGRAM for this project.
Network header generation – the incorrect way

- Given the header specs:
  - 1 byte protocol type
  - 8 bytes source address
  - 2 bytes source port
  - 8 bytes destination address
  - 2 bytes source port
- Allocate a (1+8+2+8+2) byte buffer and manually fill in the contents.

```c
char* header = (char*) malloc(sizeof(protocol_type) +
  sizeof(source_address) + sizeof(source_port) +...
);
memcpy(header, &protocol_type, sizeof(protocol_type));
memcpy(header + sizeof(protocol_type), &source_address,
  sizeof(source_address));
```
Why is this a bad idea?

- Tedious to code.
  - Lots of memcpy() and sizeof() operations.
  - Code looks ugly.

- What if the header specs change later?
  - Must manually change all offsets.
Idea: use a struct to store all fields so they are arranged correctly in memory.

- Compiler arranges a contiguous block of memory for the struct.
- Memory layout of the struct follows the order declared by the struct.

```c
struct header
{
    char protocol_type;
    network_address_t source_address;
    unsigned short source_port;
    network_address_t destination_address;
    unsigned short destination_port;
};

struct header hdr;
network_send_pkt((char*) &hdr, sizeof(hdr), ...);
```
Iteration #2: Close but not quite…

- Padding!
- Computers usually load in units of words.
  - If a multibyte variable spans 2 words, then 2 loads are needed.
  - Align the variable to some word boundary so it requires exactly 1 load.
- Padding is unpredictable and is a waste of resources to transmit.
Idea: Use a struct that cannot possibly have padding.
- Chars require exactly 1 load no matter where they are located.
- Therefore consecutive char fields in a struct are not padded.
- Works regardless of compiler options for padding.

```c
struct header
{
    char protocol_type;
    char source_address[8];
    char source_port[2];
    char destination_address[8];
    char destination_port[2];
};
```

Use packing functions to convert and populate the char arrays.
Use an array for your ports.
- $O(1)$ time when using unbound ports (since user specifies the port he wants).
- $O(1)$ time when creating bound ports before a wraparound; $O(n)$ time afterwards is acceptable (since you need to check each port).

Use semaphore_P and semaphore_V for blocking and unblocking threads.
- Remember how we did this in project 2; consider places where you need to disable interrupts.
Implementation Hints

- Reuse your queue implementation.
  - This is useful for storing data in FIFO order.

- Perform sanity checks.
  - Is the protocol type correct?
  - Are you sure the received packet is meant for you?
  - Is the packet malformed (header too short, invalid port numbers, etc)?
Implementation Hints

- Consider semantics for unused ports.
  - Data sent to unused ports should not actually be transmitted.
  - Data received on an unused port should not be queued.

- Consider reuse semantics for unbound ports.
  - When an unbound port is destroyed and later re-created, any prior queued data should no longer be there.
  - Don’t forget to reset the counting semaphore too.
You may assume the specified port ranges will not change
  - No magic numbers, use `#define`

Dynamic memory responsibilities.
  - Network interrupt handler passes you an `network_interrupt_arg_t`, which you have to eventually free.
  - The user-supplied buffer for both `minimsg_send` and `minimsg_receive` should not be freed by you.
Mutexes and semaphores for unbound ports.

- You will need a counting semaphore.
- But you won’t need a mutex. (Why?)

```c
struct miniport {
    char port_type;
    int port_number;

    union {
        struct {
            queue_t incoming_data;
            semaphore_t datagrams_ready;
        } unbound;

        struct {
            network_address_t remote_address;
            int remote_unbound_port;
        } bound;
    }
};
```
Testing

- Occasional lost packets across machines.
  - This is normal.
  - Try re-executing your program again.
- Unable to communicate between two machines.
  - Make sure both machines can ping each other.
  - Try running on two machines in the CSUG lab.
  - Redrover is known to have problems with machine visibility.
- TAs will set up their solutions to test against in office hours this week.
Questions