Project 3: Unreliable datagrams

Owen Arden
owen@cs.cornell.edu
Upson 4126
All slides stolen from previous year.
What do you have to do?

- Implement unreliable communication
  - Simulate (parts of) the UDP/IP protocol
  - Build a datagram networking stack
    - Use the provided pseudo-network interface (see "network.h")
    - Interface in "minimsg.h", skeleton code in "minimsg.c" provided to fill in
    - Implement ports to identify the endpoints
    - Build a *minimessage* layer for thread I/O
A glimpse at interface to implement

```c
#define MINIMSG_MAX_MSG_SIZE (4096)
typedef struct miniport* miniport_t;
typedef char* minimsg_t;

void minimsg_initialize();

miniport_t miniport_local_create();
miniport_t miniport_remote_create(network_address_t addr, int id);
void miniport_destroy(miniport_t miniport);

int minimsg_send(miniport_t local, miniport_t remote, minimsg_t msg, int len);
int minimsg_receive(miniport_t local, miniport_t* remote, minimsg_t msg, int *len);
```
Networking pseudo-device (1)

- Allows communication between minithreads systems
- Interrupt-driven implementation
  - `Network_handler`
    - Similar to clock handler, same interrupts used
    - Executed separately for each received packet
    - Uses the stack of the current thread
    - Should finish as soon as possible
    - Initialized with "network_initialize()"
Networking pseudo-device (2)

- **Network_handler** receives a structure:
  ```c
  typedef struct
  {
      network_address_t addr; // sender
      char buffer[MAX_NETWORK_PKT_SIZE]; // hdr+data
      int size; // size
  } network_interrupt_arg_t;
  ```

  - Need to strip the header off the buffer

- Call **“network_initialize”** function
  - After clock_initialize()
  - Before enabling interrupts and running threads
Networking functions

- **Network_send_pkt** – sends a packet
  - Destination
  - Header (body, length)
  - Data (body, length)

- **Header:**
  - Extra information
    - About the sender
    - About the receiver
  - As small as possible
Miniports

- **Data structures** that represents endpoints
  - Network Device does not control which thread processes a received packet

- **Local ports:**
  - Usually, used for listening
  - Not bound to any remote ports
  - Can receive from any remote port

- **Remote ports:**
  - Created when a packet is received
  - Bound to a “remote” port
  - Allows the receiver to reply
Miniports example (1)

- Ports 1,3 – local ports; 2 – remote port
- A,B - Threads
- Sender A sends a message to Receiver B
Miniports example (2)

- Minithread system creates the remote port 100
- Message is delivered to the local port
- B receives the message;
Miniports example (3)

- B replies to A using the newly created remote port
- The message is relayed to A’s local port
typedef struct miniport {
    char port_type;
    int port_number;
    queue_t msg_queue;
    semaphore_t msg_sem;
    semaphore_t msg_mutex;
    network_address_t remote_address;
    int remote_port;
    int remote_is_local;
} miniport;
Miniports – you can use unions

```c
struct miniport {
    char type;
    unsigned int portno;
    union {
        struct {
            queue_t receiver_queue;
            semaphore_t queue_lock;
            semaphore_t data_ready;
        } loc;
        struct {
            unsigned int portno;
            network_address_t addr;
        } rem;
    } u;
};
```
Miniports - hints

- **Local communication**
  - Note that `miniport_destroy` function will be called by the receiver
  - `remote_miniport` as a pointer to a local port
  - `miniport_send` implemented based on the "remote port"

- **Miniports**
  - Identified by numbers
  - Assigned them successive numbers
  - Local miniports – start from 0
  - Remote miniports – start from 32768
Minimsg layer

- Identifies the end-points of the communication (ports)
  - The sender assembles the header used to identifies the endpoints
  - The receiver
    - examines the header
    - Identifies destination
    - Enqueues the packet in the right place, wakes up any sleeping threads
Minimsg functions

- **Minimsg_send:**
  - Non-blocking
  - Parameters:
    - local and remote ports
    - The message and its length
  - Appends the header to the message
  - Sends the entire data using `network_send`

- **Minimsg_receive:**
  - Blocks the thread until it receives a message on the specified port
  - Receives information about the remote port – used to reply
Implementation hints

- Do not add unnecessary data to the header
  - Include the address of the sender (used later by the ad-hoc routing protocol later)
- Port operations must be O(1)
- Do not waste resources
- Make sure a port in use is not reassigned
- Remote miniports are destroyed by the application
- `network_initialize` returns the ip address of the machine
- Build other test cases