P3
Unreliable Datagram Communication

Drew Zagieboylo

CS 4411 - March 9th, 2018

P2 Postmortem

- Great Job!
- Don’t stress about grades :)
  - We’re strict on interrupt safety now so that you don’t have to worry later
  - (this will improve your P3-P5 grades)

P2 Postmortem

- Common Bugs
  - In minithread_exit()
    - semaphore_V() doesn't create lock around cleanup queue (need another semaphore or disable interrupts)
    - interrupts need to be disabled until mt_switch()

P2
Postmortem

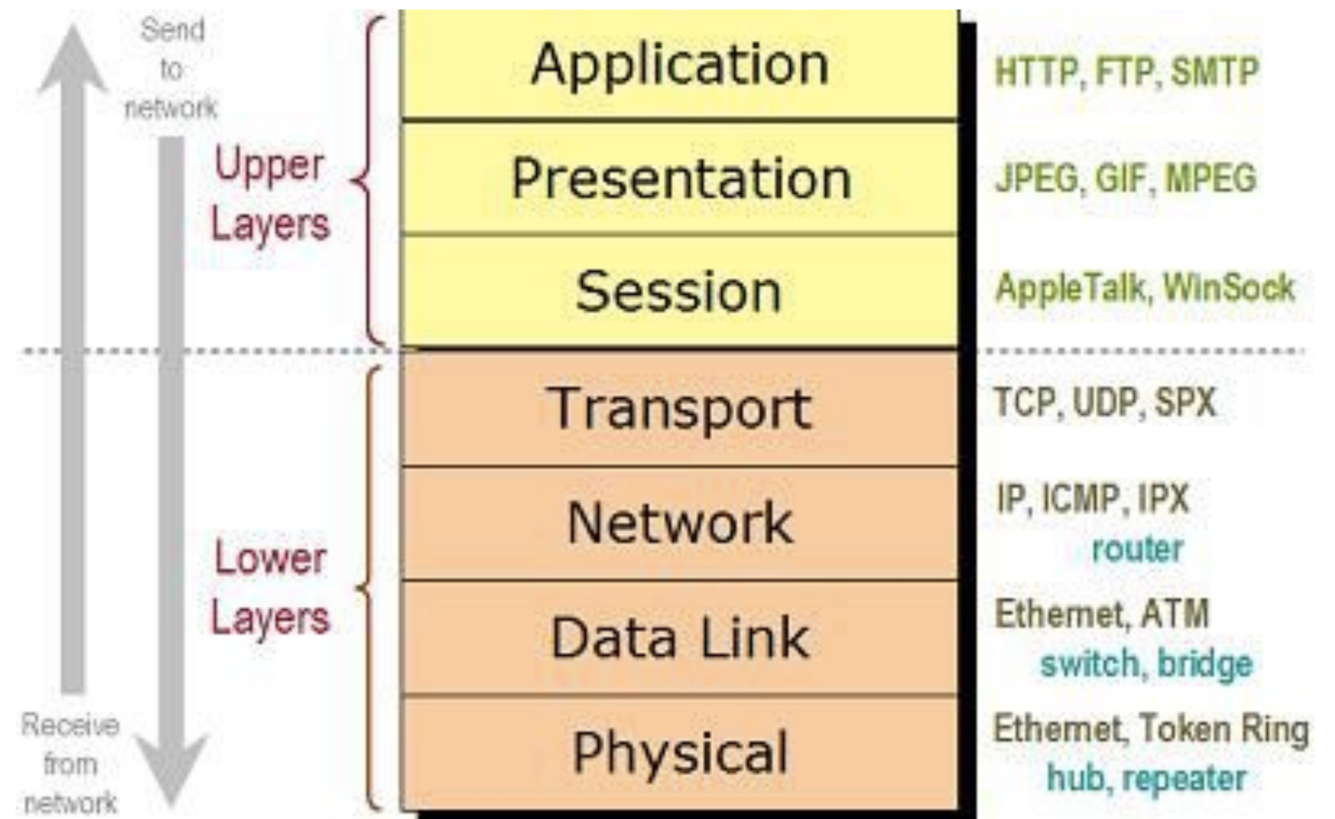
- Common Bugs
    - Scheduling the idle thread
        - Only run if there's nothing on *any* level of the run queue
        - Use a single `schedule_next_thread()` function

P2 Postmortem

- Common Bugs
  - Alarm Queue Interrupt Safety
    - Alarm is User-Facing
    - Must implement interrupt-safety when accessing alarm queue

Networking

- Processes and Machines
- Protocol
  - Agreement for how to communicate
- Many-layered stack
  - OS -> Transport Layer



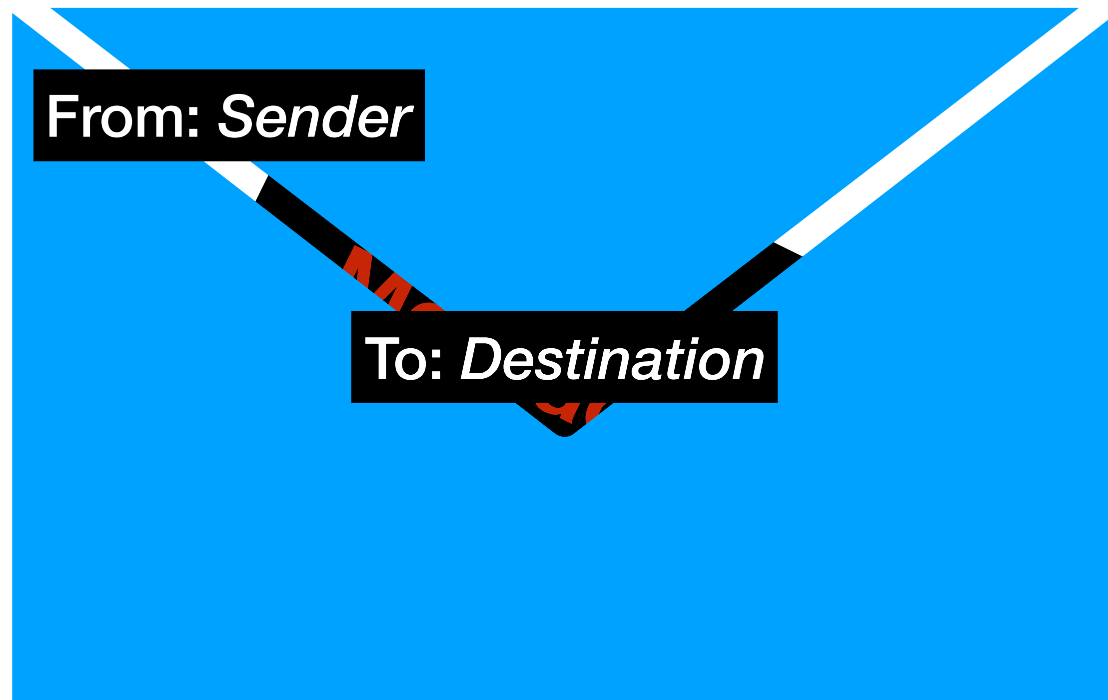
Datagram Protocol

From: *Sender*

To: *Destination*

- Simplest Transport Layer Protocol
  - “Here are some bytes!”
- Message + Sender + Receiver

Datagram Protocol



- Datagrams could be delivered:
  - *out of order*
  - *not at all*
- Have max size, larger messages must be broken into multiple datagrams
- Handling ^ is the application’s problem

Datagram Protocol

- `miniheader.h`
- `mini_header_t`
- ```
  {
     protocol
     src_port
     src_address
     dest_port
     dest_address
  }
  ```

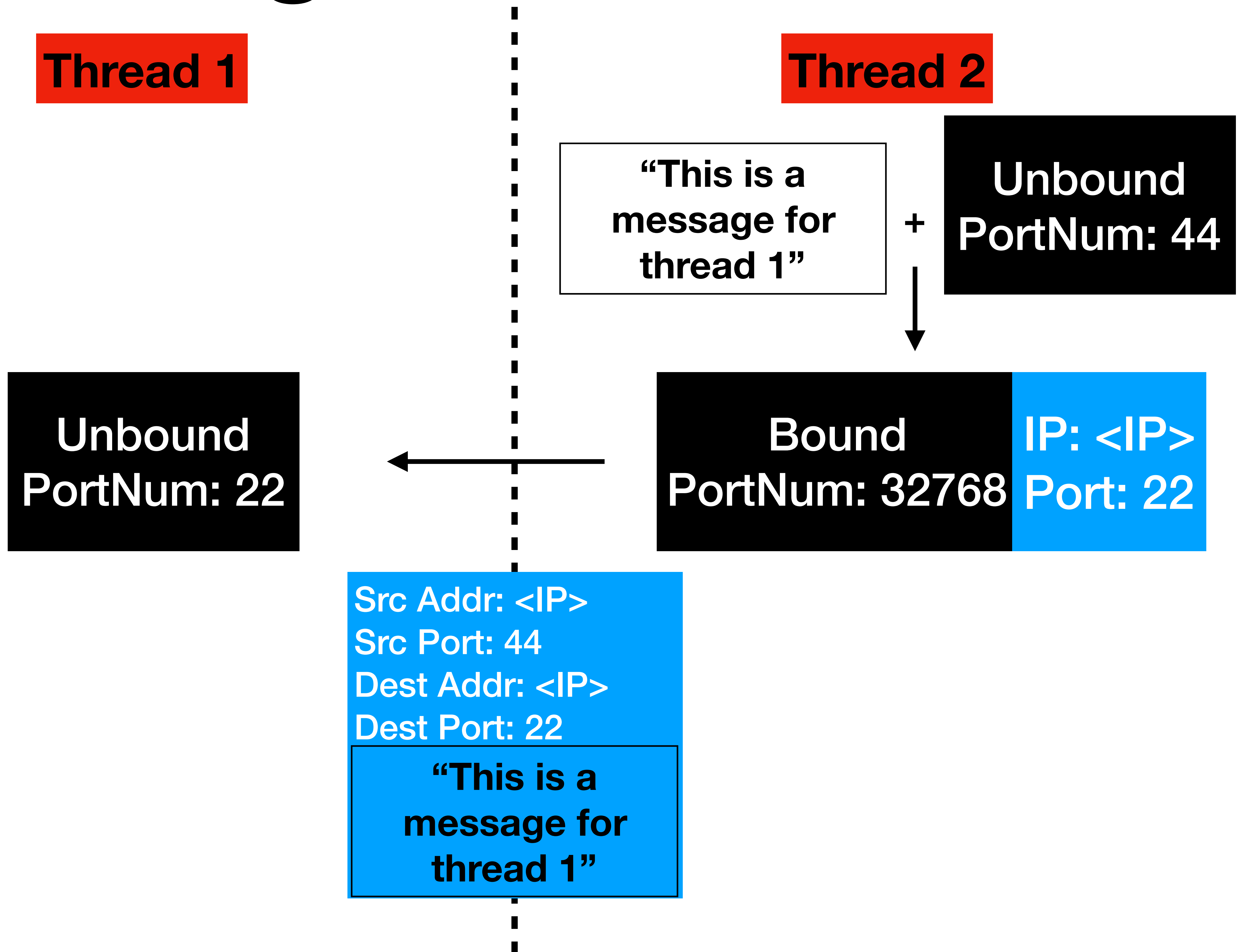
- Header
  - All ‘metadata’ about message
  - *address* identifies the physical machine
  - port (usually) identifies the process/thread on the machine
  - **port is NOT the same as Linux ports**
  ```

Datagram Protocol

- Interface:
  - create/destroy miniport
  - send
  - receive

- bound vs. unbound miniports
- **unbound**
  - used for ‘listening’ (like a server)
  - used to receive responses
- **bound**
  - used to send messages
  - need to specify a remote *unbound* port

Datagram Protocol



Datagram Protocol

- send vs. receive
- **send**
  - `minimsg_send(local_unbound_port, local_bound_port, msg, len)`
- **receive**
  - `minimsg_recv(local_unbound_port, new_local_bound_port, msg, len)`

Datagram Protocol

- send vs. receive
- **send**
  - `minimsg_send(local_unbound_port, local_bound_port, msg, len)`
- **receive**
  - `minimsg_recv(local_unbound_port, new_local_bound_port, msg, len)`

Used to identify the destination

Used to receive response

Datagram Protocol

- send vs. receive
- **send**
  - `minimsg_send(local_unbound_port, local_bound_port, msg, len)`
- **receive**
  - `minimsg_recv(local_unbound_port, new_local_bound_port, msg, len)`

Used to identify the listener

Used to send future responses

Minimsg Send

```
main() {
 char[] msg = “hello_world”;
 mp* remote_mp = miniport_create_bound(
              addr(“123.123.123”),22);
 mp* local_mp = miniport_create_unbound(44);
 minimsg_send(local_mp, remote_mp, msg, 12);
}
```

Minimsg Send

- Fire & Forget
- Create header; then send packet
- We supply a ‘send packet’ primitive
- `network_send_pkt(dest_ip, header_len, header, msg_len, msg)`

Minimsg Receive

```
main() {
 mp* local_mp = miniport_create_unbound(22);
 mp* remote_mp;
 char[] test;
 minimsg_receive(local_mp, &remote_mp, test, 20);
 if(strcmp(test, “hello_world”) == 0) {
   minimsg_send(local_mp, remote_mp, “HI!”, 4);
 }
}
```

Minimsg Receive

- How do we receive messages?
- What does minimsg_recieve look like?
- Busy waiting for I/O is wasteful!
- Receive a notification whenever a datagram arrives! (interrupt)
- Multiple threads can ‘listen’ on the same port -> each datagram is just received by any one of them

Network Handler

- For each unbound port number:
  - See if it’s been created
  - Maintain a queue of received datagrams
- For each bound port number:
  - See if it’s been created
  - Maintain info on port to which it’s bound

Network Handler

1. Triggered by network interrupt (packet received)
2. Need to:
    1. Disable interrupts
    2. Check header contents
    3. Save packet on appropriate miniport queue
    4. Notify any waiting threads that packet has arrived

Real Network Impl

- network.c implements a virtual network, using the Unix sockets API
- Can introduce virtual unreliability and re-ordering
- You can *actually communicate over the internet*
- ^ The real network really *will drop packets*.
- Use local communication (e.g. between PortOS threads) to ensure reliability when testing

Misc

- miniheader.h has functions for reading/writing headers
- Don’t modify any of the new header files
- Grading will be autograded with a large suite of tests
  - We provide a very small number
  - We will much more extensively stress test your sends, error handling, etc.
- Other guidelines will be in the README