Announcements

- Reading
  - Today: Chapter 5 (5.1-5.2)
- Project #2
  - Due on Monday Sept 24th (10 AM)
  - Pthreads book in on reserve on Engineering Library
  - In makefile, need to use –lpthread when linking

Condition Variables

- Allow threads to wait on the value of a variable
  - wait until the list is non-empty for example
  - allows one thread to signal to another thread that something has changed
    - threads may sleep waiting to be notified of this change
- Can unlock and re-lock a mutex before/after suspend

```
wait for count to be >= 1
   pthread_mutex_lock(&count_mutex);
   while (count <= 0) {
      pthread_cond_wait(&count_condvar, &count_mutex);
   }
   pthread_unlcok(&count_mutex);

update count:
   pthread_mutex_lock(&count_mutex);
   count++;
   pthread_mutex_unlock(&count_mutex);
   pthread_cond_signal(&count_condvar);
```

Consider the following program

T1:

count++ -- in C one statement, but really multiple instructions

```
load r1, count
add r1, 1, r1
store r1, count
```

T2:

count++ -- in C one statement, but really multiple instructions

```
load r2, count
add r2, 1, r2
store r2, count
```

What happens when T1 is preempted right after the load

With Synchronization

T1:

```
pthread_mutex_lock(&mylock)
count++
pthread_mutex_unlock(&mylock)
```

T2:

```
pthread_mutex_lock(&mylock)
count++
pthread_mutex_unlock(&mylock)
```

Only one thread at a time gets to update the count

Queue Project

- Need to coordinate access to shared resources
  - use mutex to guard access to a shared data structure
- Queue abstraction is **very** useful
  - enqueue: add item to queue
  - dequeue: remove item, **block** if not ready
  - head: return head of queue without dequeue
  - probe: test if the queue is empty

  - must use a mutex to protect access to queue
  - build a producer/consumer test program
- Multiple application threads
  - our test application is multi-threaded
  - must be able to support multiple threads trying to en-queue

Network Layer

- Responsibility
  - end-to-end delivery of packets to the network
  - selecting routes for the packets to take
    - implies knowledge of the network topology
  - managing utilization of the links
    - provide flow control (across multiple links)
    - spread load among different routes
- Interface Design
  - should be independent of subnet technology
  - hide number, type, and topology of network from upper layers
  - export a common number plan for entire network

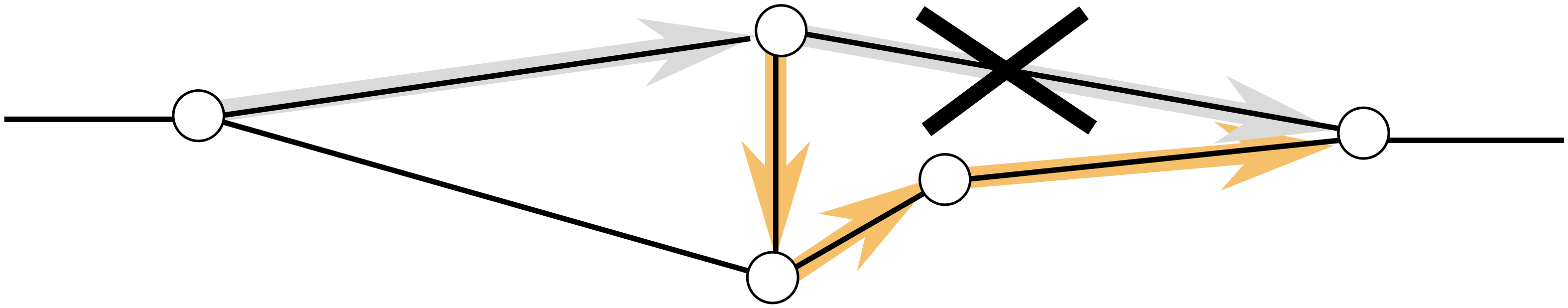
Connection vs. Connectionless

- Two possible designs for network layer
  - connection oriented service (ATM)
    - based on experience of telcos
  - connectionless service (IP)
    - based on packet switching (ARPANET)
- Connectionless
  - transport datagrams from source to destination
    - end-point addresses in every datagram
  - less complex network layer, more complex transport
- Connection oriented
  - also called virtual circuits
  - establish an end-to-end connection with network state
    - can use VCI (global or next hop) in each packet

Datagram vs. VC Addresses

- Datagrams
  - must include full address in each packet
  - addresses must be unqiue for entire network
    - don't re-use too often
    - addresses per src/dest pair
- Virtual Circuit
  - globally unique
    - requires allocation scheme to ensure its unique
    - consumes many bits per packet
  - per link
    - requires translation at each switch
    - uses fewer bits (important for small packets like ATM)

Link Failue in Virtual Circuits



- Re-establish virtual circuit
  - router near failure can patch up link
  - original host/router creates new virtual circuit
- Virtual circuit is dropped
  - transport layer can handle recovery

Virtual Circuit vs. Datagram

Issue	Datagram	Virtual Circuit
Circuit setup	not needed	necessary
Addresses	full source/dest per packet	next hop vc sufficient
state	no state in network	per connection data at each router
routing	each packet individually	once at VC setup
router/link failure	a few packets may be lost	all VCs through router are terminated
congestion control	difficult	many pre-allocation and policing policies permitted

Routing: Goals

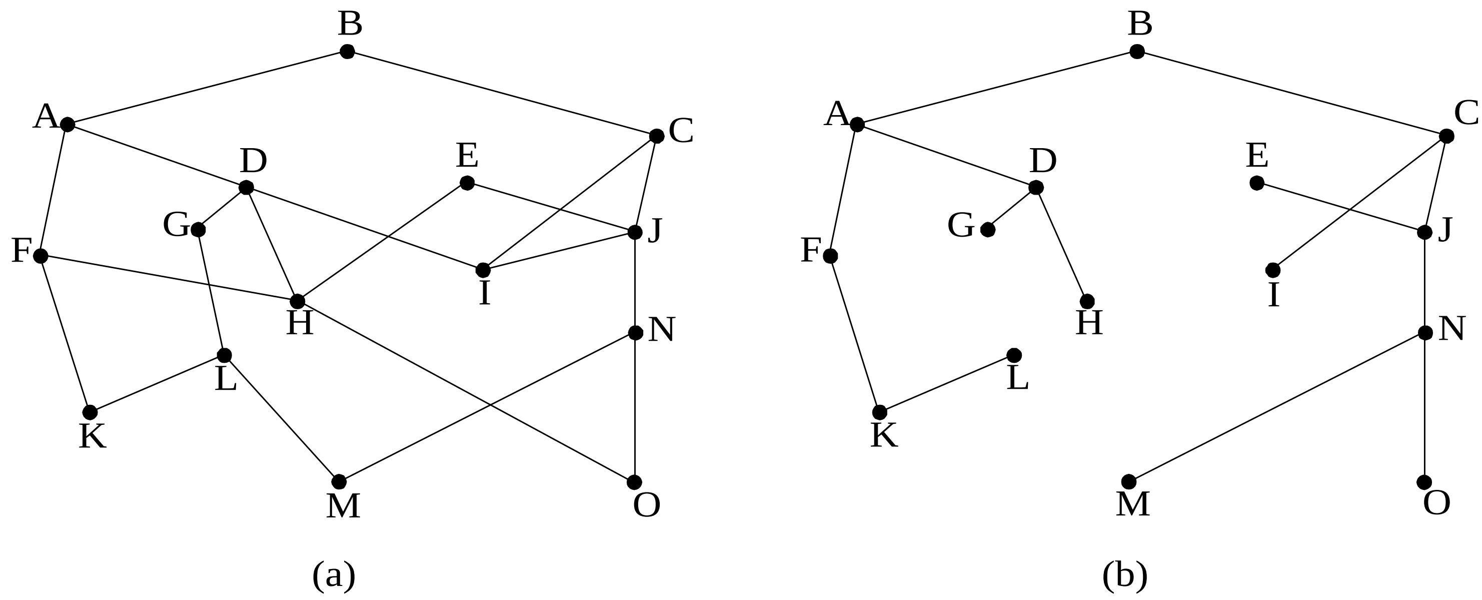
- Correctness
  - packets get where they are supposed
- Simplicity
  - easy to implement correctly
  - possible to make routing choices fast (or updates easy)
- Robustness
  - failures in the network still permit communication
- Stability
  - small changes in link availability results in a small change in the routing information
- Fairness
  - each host, VC, or datagram has the same chance
- Optimality
  - best possible route
  - best utilization of bandwidth

Do Routes Change During Network Operation?

- nonadaptive routing (static routing)
  - information loaded a boot time
  - never changes during network operation
- adaptive routing
  - changes in network operation alter routes
  - issue: where to get this data to make choices
    - locally from neighbors
    - globally from all routers (or a NIC - network information center)
  - issue: when to change routes
    - only on topology changes (links or routers change)
    - in response to changes in load
  - issue: metric to optimize
    - distance, number of hops, estimated latency

Optimality Principal

- If J is on the optimal route from I to K
  - then the optimal route from I to K shares the optimal route from J to K
- transitive result of this is a sink tree
  - can construct a tree from all nodes to a specific node



From: *Computer Networks*, 3rd Ed. by Andrew S. Tanenbaum, (c)1996 Prentice Hall.

Shortest Path Routing

- Graph Representation
  - nodes are routers
  - arcs are links
  - to get between two routes, select a the shortest path
  - need to decide metric to use for minimization
- Dijkstra's Algorithm

```
select source as current node
while current node is not destination
     foreach neighbor of current
          if route via current is better update its tentative route
          label node with <distance, current Node>
     find tentative node with shortest route
          mark a permanent
          make it current
```