

CSCI 491-01

Topics: Internet Programming

Fall 2008

Preliminaries II

Derek Leonard
Hendrix College

August 29, 2008

Preliminaries II: Agenda

- **Sockets**
 - Clients
- Multithreaded Applications
 - Mutex
 - Semaphore
- STL data structures
 - Queue
 - Binary search tree

Preliminaries II: Sockets

- Sockets are interfaces to the TCP/IP protocol stack
 - More on TCP and IP later in the semester
 - HTTP uses TCP
- Communication using sockets is accomplished by a set of system calls
- TCP sockets can be used in two modes:
 - Server (socket listens for incoming connections)
 - **Client** (socket actively establishes outgoing connections)

Preliminaries II: Sockets 2

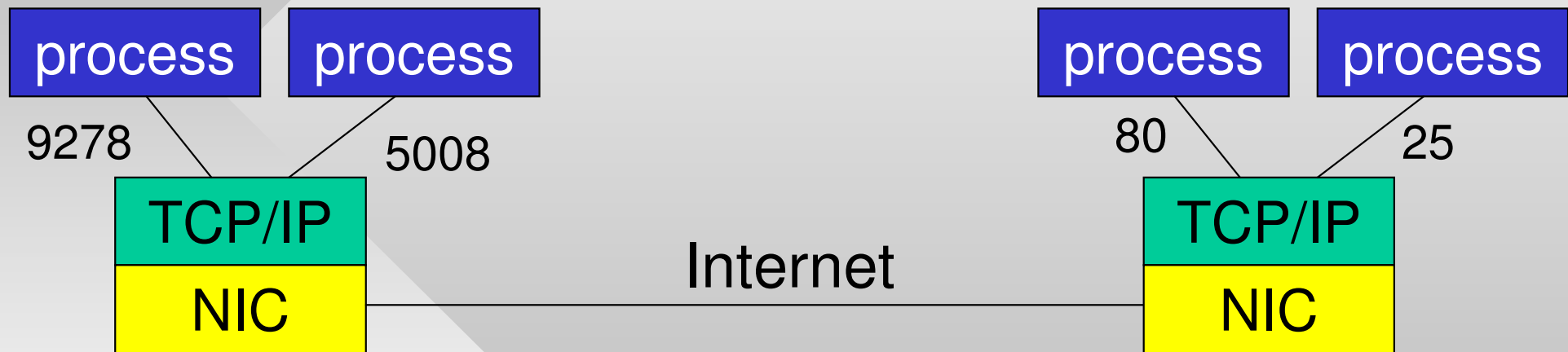
- Addressing remote hosts
 - IP Address: uniquely identifies the host to be contacted
 - 4-byte number written with a dot between each byte
 - E.g., 150.208.10.56 (or 0x80C2873C)
- Localhost has IP address 127.0.0.1
- **What if multiple network applications need to be run simultaneously on one host?**
- Solution: ports
 - Each socket is **bound** to a unique port
 - The OS forwards messages addressed to that port to the correct application
- Implication: to connect to a server, you need the correct IP address and port number

Preliminaries II: Sockets 3

- Ports are 2-byte unsigned integers
 - Port 0 is reserved
 - Ports 1-1023 are system ports; 1024-65535 user ports
- Some well-known ports
 - HTTP: 80
 - Telnet: 23
 - SSH: 22
 - SMTP: 25
- When issuing a connect, the OS implicitly binds the socket to the next available port
 - Clients do not need to worry about their port numbers
 - Ports only meaningful for servers

Preliminaries II: Sockets 4

- Big picture of how this works:



Preliminaries II: Agenda

- Sockets
 - Clients
- Multithreaded Applications
 - Mutex
 - Semaphore
- STL data structures
 - Queue
 - Binary search tree

Preliminaries II: Clients

- Discuss handout

Preliminaries II: Agenda

- Sockets
 - Clients
- **Multithreaded Applications**
 - Mutex
 - Semaphore
- STL data structures
 - Queue
 - Binary search tree

Preliminaries II: Multithreaded Apps

- Allows for multiple threads of execution in one process
- Benefits:
 - If a blocking call is made in one thread, other threads can continue executing
 - Allows for parallelism in a multiprocessor/multicore system
- Issues:
 - Global memory is shared between threads
 - Order of execution of the threads is not known
- Homework note: pass shared parameters to threads instead of using global variables (see `thread.cpp` on course site)

Preliminaries II: Multithreaded Apps 2

- Reasons for using multiple threads in hw #1
 - Web servers usually respond slowly
 - Each thread is suspended for tens of seconds per connection waiting for connect() and recv() to complete
- Multiple threads achieve significant speed-up
 - You could run hundreds of threads, but limit your testing to 5-10 until you know it works correctly
- Consider the code shown on the course webpage
 - What does it do?
 - What does the mutex guarantee?
 - What does the semaphore guarantee?
 - Which thread gets control of the shared memory first?

Preliminaries II: Multithreaded Apps 3

- Synchronization mechanisms used in this code
 - Mutex (mutual exclusion): allows only one thread access to critical section
 - Semaphore: allows up to N concurrent threads during access
- Usage
 - Any data structure modified by parallel threads needs to be protected (e.g., BFS queue, set S, set of visited URLs)
 - If not protected, inconsistencies may result

```
lock mutex // pthread_mutex_lock()  
critical section // e.g., extract URL from Q  
unlock mutex // pthread_mutex_unlock()
```

Preliminaries II: Multithreaded Apps 4

- After a page is crawled by a thread:

```
parse out a URL x of current page
lock mutex
if (x is contained in visited_urls)
    <do nothing>
else {
    add x to Q; add x to S;
}
unlock mutex
```

- Correct version:

```
else {
    add x to Q
    add x to visited_urls
    add x to S
}
```

Preliminaries II: Multithreaded Apps 5

- Semaphores can be used to count events or allow up to N concurrent critical sections
 - A semaphore has a numerical value P attached to it
 - When $P=0$, the semaphore is locked (all threads block)
 - When $P>0$, next thread is allowed access and P is set to $P-1$
- Can be used to wait for all threads to terminate:

```
int n = 100;
... // start n threads
for (int i=0; i<n; i++)
    Lock P
```

```
thread () {
    ...
    release P
}
```

Preliminaries II: Multithreaded Apps 6

- Homework hint: start all N threads at once (dynamic creation/deletion takes a lot of time)
- BFS loop ideas
 - 1) no thread tries to extract from BFS queue unless there is something in it; 2) no periodic polling of queue size
- Many threads write into Q, many read from it (producer-consumer problem)
 - Solution: utilize a semaphore that counts the # of items in the queue (call this semaQ)
 - No upper limit on queue size, so no need for the second semaphore

Preliminaries II: Multithreaded Apps 7

- Pseudo code:

```
while(true) {  
    Lock semaQ  
    Extract URL x from Q  
    Crawl x  
    for each new link y {  
        insert y into Q  
        Release semaQ  
    }  
}
```

- Anything missing?

Preliminaries II: Multithreaded Apps 8

- Improved version:

```
while(true) {
    Lock semaQ
    Lock BFSmutex
    Extract URL x from Q
    Unlock BFSmutex

    Crawl x      // delay here

    Lock BFSmutex
    for each new link y {
        insert y into Q
        Release semaQ
    }
    Unlock BFSmutex
}
```

Preliminaries II: Agenda

- Sockets
 - Clients
- Multithreaded Applications
 - Mutex
 - Semaphore
- **STL data structures**
 - Queue
 - Binary search tree

Preliminaries II: STL

- STL allows implementation of data structures with elements of arbitrary classes
- Example of using queues (O(1) insertion/deletion):

```
#include<deque>           // double-ended queue
using namespace std;
deque<int> myqueue;
myqueue.push_back (100);
int val = myqueue.pop_front();
```

- You can get a stack if you pop from the back:
`myqueue.pop_back()`
- You can use strings as elements

Preliminaries II: STL 2

- Example with strings:

```
#include <string>
string a = "hello world";
deque<string> myQ;
myQ.push_back(a);
```

- Simplest way to implement URL BFS queue

```
class urlInfo { string url; int depth; };
queue<urlInfo> Q; // single-sided queue
```

- Example:

```
urlInfo x = {"www.cnn.com", 1},
        y = {"www.hendrix.edu", 1};
Q.push(x);
Q.push(y);
```

Preliminaries II: STL 3

- For the visited URLs, you need to quickly find whether it has been visited or not
 - Use a balanced binary search tree for this
 - In STL, one option is to use a `set` (red/black trees)
- First, define a comparison operator for the class:

```
class urlInfo{
public:
    string url;
    int depth;
    bool operator () (urlInfo x, urlInfo y) {
        return (x.url != y.url);
    }
};
```

Preliminaries II: STL 4

- Second, specify the comparison predicate

```
set<urlInfo, urlInfo> s;  
urlInfo x = {"a.com", 1}, y = {"b.com", 1};  
s.insert (x);  
s.insert (y);  
s.insert (x);           // only <x, y> are in set
```

- To check if insertion was successful (element not in the set already)
 - Check the set size before and after insertion (s.size())
 - Or use the returned iterator from s.insert()
- Insertion/search in $O(\log N)$ time

Preliminaries II: STL 5

- Iterating through elements of a set

```
set <urlInfo, urlInfo>::iterator iter;  
for (iter = s.begin(); iter != s.end(); iter++)  
{  
    // print the URL and depth  
    printf("%s:%d\n", iter->url.c_str(), iter->depth);  
}
```

- The result looks like this:

```
a.com:1  
b.com:1
```

Preliminaries II: Home Tasks

- Read Chapter 1 of Kurose and Ross
- If you have any questions about the homework, make sure to email me or come by my office
- Achieve working knowledge of STL
- Run sample code from the website, connect to a webpage, download it, write a URL parser