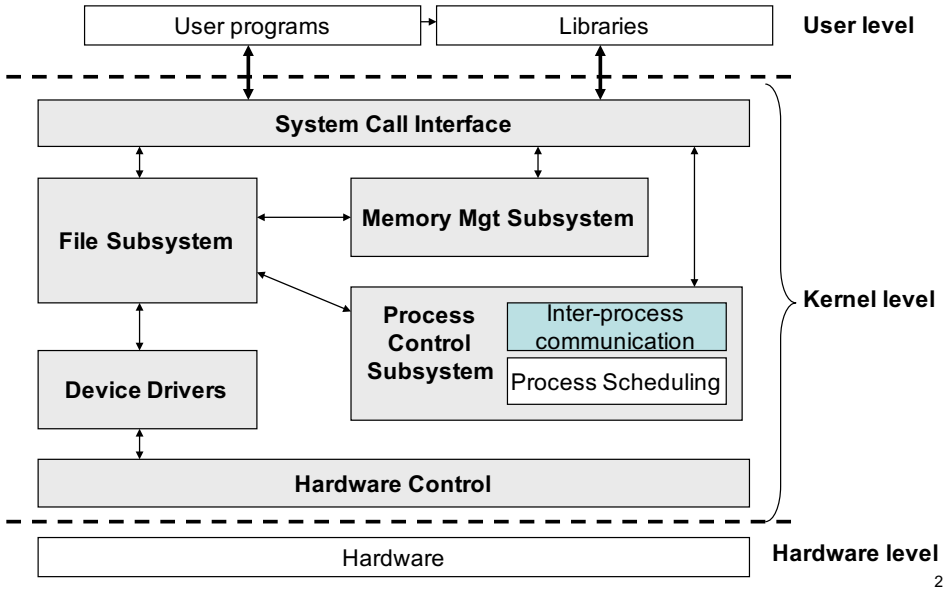
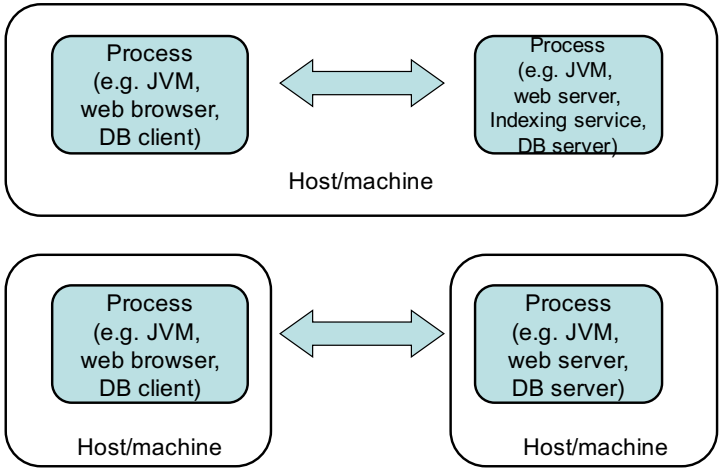
Network Programming with Sockets

An Architectural View of an OS



Inter-Process Communication



- Same system calls (APIs) and protocols can be used for both types of communication.

Protocol Stack

- Application layer (L7)
  - e.g., HTTP, POP, SMTP, SSH, SIP, RTP, Skype
- Session layer (L5)
  - e.g., SSL
- Transport layer (L4)
  - e.g., TCP, UDP
- Network layer (L3)
  - e.g., IP
- MAC (data link) and physical layers (L2 and L1)
  - e.g., Ethernet, WiFi, FDDI, ATM, LTE, 3G

Network Protocols

- A protocol allows multiple processes to talk with each other in an unambiguous way.
- Each protocol defines…
  - Communication primitives (or commands)
  - Pairs of request and response messages
  - Message format
    - Header
    - Payload

An Example: HTTP

- GET /index.html HTTP/1.0
- HTTP/1.0 200 OK
  Server: Apache…..
  Date: Wed, 11 April 2007 HH:MM:SS GMT
  Content-Type: text/html:charset=ISO…
  Set-cookie: XXXXX=ZZZZZ

  <html>
  <h1>Welcome to my home page!</h1>
  …
  </html>

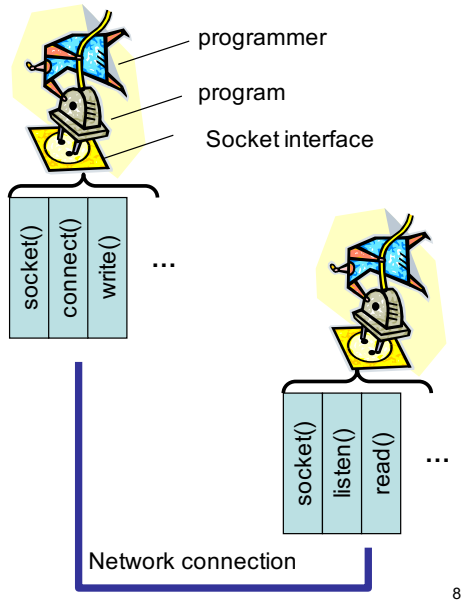
- GET /index2.html HTTP/1.0
- HTTP/1.0 404 Not Found
  Date: …
  …

Another Example: POP

- USER jxs <-- client
- +OK Password required for jxs <-- server
- PASS opensesame
- +OK jxs has 2 messages (300 octets)
- STAT
- +OK 2 300
- RETR 1
- +OK 200 octets
  email text included here
- DELE 1
- +OK message 1 deleted
- …
- QUIT
- +OK POP server signing off

Network-related System Calls

- Socket interface
  - A part of OS system call interface
  - A set of functions specific to networking
  - implements the transport layer
  - socket()
    - Creates a socket
  - bind()
    - Names a created socket
  - connect()
    - Sends out a connection request
  - listen()
    - Waits for connection requests
  - accept()
    - Accepts a connection request
  - select(), read(), write(), close()

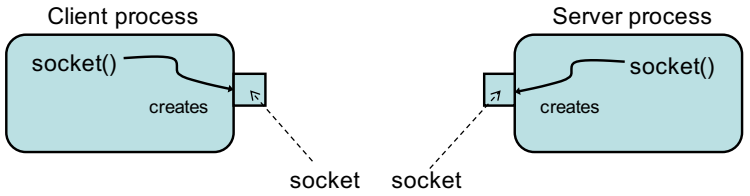


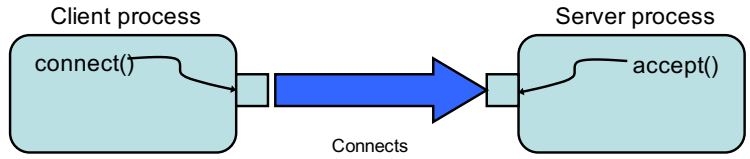
Java Networking API

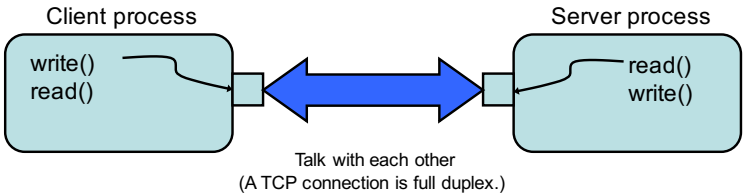
- A set of classes/methods in the java.net package
  - follows the the socket interface's design.
  - offers TCP and UDP communication capabilities.
  - glues Java programs to the socket interface
  - makes it easier to implement network systems than using socket system calls directly

Sockets

- Socket
  - A communication channel to transmit TCP/UDP packets between processes
    - on the same machine, or
    - on different machines

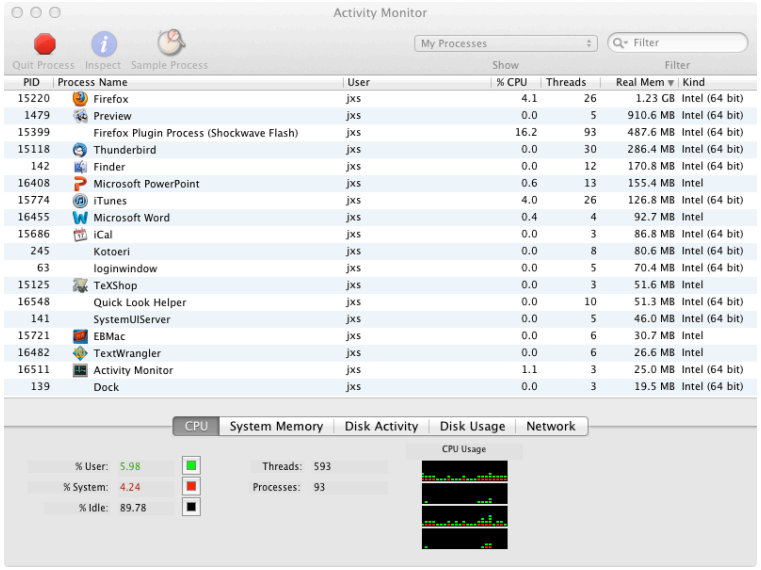






IP and Port Number

- How does a process identify and access a remote process?
  - Hundreds of processes on a machine
  - A huge number of machines in the Internet
- How about using process IDs?
  - Not good
    - The same program uses different pids when running at different times.
      - If a program is rebooted, it uses a different pid from the one it was using before the reboot.



- ps command
- GUI frontend

- A combination of an IP address and a port number
  - An IP address uniquely identifies a particular machine in the network
    - 216.243.167.53 (www.umb.edu)
  - A port number uniquely identifies a particular process on a machine.
    - A process can use the same port number at different times.
      - e.g., before and after a reboot.
      - http://www.cs.umb.edu:80

Java Socket

- Server
- ServerSocket serverSocket = new ServerSocket(9000);

  Socket socket = serverSocket.accept();

  Scanner scanner = new Scanner(serverSocket.getInputStream());

  PrintWriter writer = new PrintWriter(serverSocket.getOutputStream());

- serverSocket.close();
  - Easy to forget closing an open socket.
  - Put it in a finally clause.

- Client
- Socket socket = new Socket(“localhost”, 9000);

  PrintWriter writer = new PrintWriter(socket.getOutputStream());

  Scanner scanner = new Scanner(socket.getInputStream());

- socket.close();
  - Put it in a finally clause.

Make Sure to Close ALL Sockets

- A serer program closes the server-side socket.

```
try{
    try{
        ServerSocket serverSocket = new ServerSocket(9000);
        while(true){
            Socket socket = serverSocket.accept();
            ...
        }
    }finally{
        serverSocket.close();
    }
}catch(IOException exception){...}
```

- A client program closes the client-side socket.

```
try{
    try{
        Socket socket = new Socket("localhost", 9000);
        ...
    }finally{
        socket.close();
    }
}catch(IOException exception){...}
```

Closing a Socket with a try-with-resources Statement

- `ServerSocket` implements the `AutoCloseable` interface.
  - c.f. lecture note #1
  - *Try-finally*

```
try{
    serverSocket socket = new ServerSocket(...);
    // Open a (server-side) socket
    doSomething();
}finally{
    socket.close(); // Close the socket
}
```

  - *Try-with-resources*

```
try(ServerSocket serverSocket = new ServerSocket(...)){
    doSomething();
}
```

  - `close()` is automatically called on a socket when existing the try block.
    - No explicit call of close() necessary on `socket` in the finally block.

- A serer program closes the server-side socket.

```
try(ServerSocket serverSocket = new ServerSocket(9000)){
    while(true){
        Socket socket = serverSocket.accept();
        ...
    }
}catch(IOException exception){...}
```

- A client program closes the client-side socket.

```
try(Socket socket = new Socket("localhost", 9000)){
    ...
}catch(IOException exception){...}
```

- The catch block is called when an exception occurs in the try block.
  - It runs after close() is called on a socket.

Sample Code

- Networked bank account
  - A bank account at the server side
  - A client accesses the bank account through via TCP socket
- Simple Banking Protocol (SBP)
  - Commands from a client
    - BALANCE
      - Get the current balance. The current balance is returned.
    - DEPOSIT X
      - Deposit amount X. The current (updated) balance is returned.
    - WITHDRAW X
      - Withdraw amount X. The current (updated) balance is returned.
    - QUIT
      - Close a TCP connection

- Run SBP code.
  - Run two shells (terminal windows)
    - One for a server and the other for a client
  - java edu.umb.cs.threads.net.BankServer in the server shell
    - Kill the program with Ctrl-C
  - java edu.umb.cs.threads.net.BankClient in the client shell
    - A *batch* program that checks the current balance, deposit $100 and withdraw $50.
- Access BankServer *interactively* with telnet
  - telnet localhost 8888 (telnet 127.0.0.1 8888)
  - Type in and send commands

- Run BankServer and BankClient on different hosts
  - Replace “localhost” with a remote host name/IP in BankClient
    - new Socket(“localhost”, BANKPORT)
  - Test BankServer with telnet as well.
- Experience HTTP with telnet
  - See a set of instructions in an email I will send you tonight.

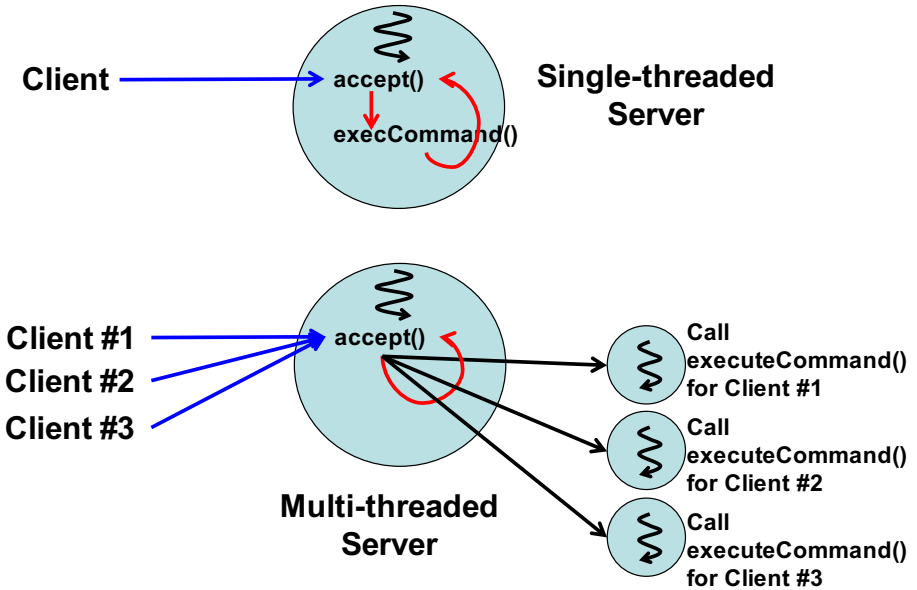
HW 26

- Implement an echo server.
  - Client
    - Connects to a server
    - Sends some string data to the server
    - Receives a returned data from the server and prints it out.
  - Server
    - Prints out incoming string data from a client
    - Sends (or echos) back the data to the client.

HW 27

- Transform bank server code to be multi-threaded.
  - Modify BankServer.init() to implement a thread-per-client policy

```
try(ServerSocket serverSocket = new ServerSocket(BANKPORT)){
      while(true){
            Socket socket = serverSocket.accept();
            executeCommand( socket );
      }
}catch(IOException exception){}
```

- Define a Runnable class
  - Its run() calls executeCommand().
    - BankServer.init() can look like:

```
while(true){
    Socket socket = serverSocket.accept();
    new Thread(new YourRunnable(socket)).start(); }
```

  - executeCommand() can receive and process multiple commands from the same client.
  - executeCommand() returns when it receives QUIT.
- Run multiple BankClient instances on different terminals
  - Have each client sends multiple SBP commands