



Advanced Operating Systems

Part 1: Process, Signal, Pipe

Sam
Luis
Avenir
Kaan

Agenda

- Introduction
- Time Functions
- Process Control
- Signals and Signal Processing
- Pipes

Introduction

System Calls - how a program requests a service from an operating system's kernel

- **may include:**
 - hardware-related services (e.g. accessing a hard disk drive)
 - creation and execution of new processes
 - communication with integral kernel services such as process scheduling

System calls provide an essential interface between a process and the operating system

Introduction

- System calls provide an essential interface between a process and the operating system
- UNIX has about 60 system calls that are at the *heart* of the operating system

Introduction

- The UNIX manual has an entry for all available functions. Function documentation is stored in *section 3* of the manual, and there are many other useful system calls in *section 2*

Time Functions

- **We can access the clock time with UNIX system calls.**
- **Uses of time functions include:**
 - Telling the time
 - Timing programs and functions
 - Setting number seeds

Time Functions

- **DateTime** is a predefined class in C# from the **System** library
 - accesses system date & time with **DateTime.Today**

```
using System;

class Program
{
    static void Main()
    {
        Console.WriteLine("Today: {0}", DateTime.Today);

        DateTime y = GetYesterday();
        Console.WriteLine("Yesterday: {0}", y);
    }

    /// <summary>
    /// Gets the previous day to the current day.
    /// </summary>
    static DateTime GetYesterday()
    {
        // Add -1 to now
        return DateTime.Today.AddDays(-1);
    }
}
```

Output:

```
Samiks-MacBook-Pro:Desktop SamAvan$ mono time.exe
Today: 3/31/2015 12:00:00 AM
Yesterday: 3/30/2015 12:00:00 AM
Samiks-MacBook-Pro:Desktop SamAvan$ █
```

Process

In order to access system calls from C#, we must include the following namespaces (collections of classes):

```
using System;  
using System.Diagnostics;
```

“using” is a keyword that imports a namespace

Process Control

- A process is a single running program.
 - Unix Process ID
 - Unix Process Status
 - Unix Process Name
- A program usually runs as a single process.
 - However later we will see how to make programs run as several separate communicating processes.

Process

- Creating a new process in C# (use Process class)

```
Process myProcess = new Process();
```

- Retrieving Process ID

```
Console.WriteLine(myProcess.Id);  
//prints process Id
```

- Start process

```
myProcess.Start()  
//starts specified process (arguments optional)
```

Introduction

Executing a process

```
using System.Diagnostics;

class Program
{
    static void Main()
    {
        // Open the file "example.txt".
        // ... It must be in the same directory as the .exe file.
        Process.Start("example.txt");
    }
}
```

Process

- Accessing process name
myProcess.ProcessName()
//returns String of process name
- Accessing process status
myProcess.Responding
//returns boolean true if running, false if
//idle

Process

Accessing Process Information

```
using System;
using System.Diagnostics;

class Program
{
    static void Main()
    {
        // Show all processes on the local computer.
        Process[] processes = Process.GetProcesses();

        // Display count.
        Console.WriteLine("Count: {0}", processes.Length);

        // Loop over processes.
        foreach (Process process in processes)
        {
            Console.WriteLine(process.Id);
        }
    }
}
```

Process

Accessing Process Information

```
using System;
using System.Diagnostics;
using System.Threading;

class Program
{
    static void Main(string[] args)
    {
        Process[] myProcs = Process.GetProcesses();

        Console.WriteLine("-----");
        Console.WriteLine("{0, -8} {1, -30} {2, -10}", "PID", "Process Name", "Status");
        Console.WriteLine("-----");

        foreach (Process p in myProcs)
        {
            try
            {
                Console.WriteLine("{0, -8} {1, -30} {2, -10}", p.Id,
                    p.ProcessName, p.Responding ? "Running" : "IDLE");
            }
            catch (Exception)
            {
                continue;
            }
        }
    }
}
```

Process

Accessing Process Information

```
Samiks-MacBook-Pro:Desktop SamAvan$ mcs processByName.cs
Samiks-MacBook-Pro:Desktop SamAvan$ mono processByName.exe
```

PID	Process Name	Status
12537	mono-sgen	IDLE
12516	Office365Servic	IDLE
12502	Google Chrome H	IDLE
12394	com.apple.iClou	IDLE
11825	bash	IDLE
11592	com.apple.sbd	IDLE
11590	com.apple.WebKi	IDLE
11588	com.apple.WebKi	IDLE
11583	Mail	IDLE
11429	QuickLookSatell	IDLE
11428	quicklookd	IDLE
11303	Google Chrome H	IDLE
11300	Google Chrome H	IDLE
11032	Google Chrome H	IDLE
10877	mdworker	IDLE
10876	mdworker	IDLE
10573	com.apple.dt.Xc	IDLE
10572	com.apple.dt.Xc	IDLE
10571	com.apple.dt.Xc	IDLE
10562	mdworker	IDLE
10558	mdworker	IDLE
10549	com.apple.Cores	IDLE
10461	Xcode	IDLE

Process

- Waiting for a process to end
Process waitProcess = myProcess.Start();
waitProcess.WaitForExit();
//exits after myProcess is complete
- Process Exit Code
myProcess.ExitCode
//returns 0 if normal termination occurs
// any other value indicates an error or unusual
//occurrence

Process

- Forcing a process to end
myProcess.Kill();
//force the process to terminate immediately

```
using System.Diagnostics;
using System.Threading;

class Program
{
    static void Main()
    {
        // Start notepad.
        Process process = Process.Start("notepad.exe");
        // Wait one second.
        Thread.Sleep(1000);
        // End notepad.
        process.Kill();
    }
}
```

Process(C/C++)

- As discussed earlier
 - Each process in a Linux system has it's own process ID, commonly referred to as "pid"
 - Every process also has a parent process (except init)
- Think of processes on a Linux system as if they form a tree, with init at it's root
 - The parent process ID, "ppid", is simply the process ID of the process's parent
 - When referring to process ID's in C/C++ code, always use `pid_t` typedef

Process(C/C++)

- The command “ps”, gives us a list of processes and some info on them such as their process ids

```
kaan@Kubuntu:~/Documents/school/540$ ps
```

PID	TTY	TIME	CMD
-----	-----	------	-----

3008	pts/3	00:00:00	bash
------	-------	----------	------

8238	pts/3	00:00:00	ps
------	-------	----------	----

```
kaan@Kubuntu:~/Documents/school/540$ ps
```

PID	TTY	TIME	CMD
-----	-----	------	-----

3008	pts/3	00:00:00	bash
------	-------	----------	------

8239	pts/3	00:00:00	ps
------	-------	----------	----

Process(C/C++)

- We can easily and separately access that information by getpid() and getppid()
- We definitely need <stdlib.h> and <unistd.h>

```
#include <iostream>
#include <string>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
using namespace std;
```

Process(C/C++)

- There are two ways to create a new process:
 - Using the system function is easy
 - system creates a subprocess, hands the command to the Bourne shell for execution

```
#include <stdlib.h>
int main ()
{
    int return_value;
    return_value = system ("ls -l /");
    return return_value;
}
```

Process(C/C++)

- Although using system is relatively simple but it's inefficient and has security risk. Using fork and exec methods are preferred for creating new processes
- Linux's fork function makes a child process that is an exact copy of it's parent process
- If you want the process to be the instance of another program, use exec
- To create a new process, use fork and then exec to make a copy of the current process and then change the process to be an instance of a program

Process(C/C++)

- When fork is called, a duplicate of the process is created
 - Referred to as the “child” of the “parent” process
 - Executes the same program from the same place
 - Has it's own unique process ID
- Fork function provides different return values to the parent and the child
 - one process(parent) goes in to fork and two(parent and child) come out
 - Child ID is returned zero while parent ID is returned with the child ID

Process(C/C++)

- exec functions replace the program running with another program
 - When exec is called by a program, the process stops executing that program and starts a new one
 - While using execvp and execlp, a program name is used to search for a program in the current path
 - While using execv and execve, an argument list for the new program as a NULL-terminated array of pointers to strings is used
 - Since exec replaces the program with another one, it never returns unless an error occurs

Process(C/C++)

- When a program is invoked from the shell, the shell sets the first element of the argument list to the name of the program, the second element of the argument list to the first command-line argument and so on.
- When using an exec function in programs, the name of the function should be passed as the first element of the argument list

Process(C/C++)

```
int global=2;
pid_t process_ID = 0;
main(){
    string process_type;
    int iStack=50;
    pid_t pID = fork();
    if(pID==0){
        process_type = "Child Process: ";
        global++;
        iStack++;
        process_ID = (int) getpid();
        execlp("ls", "ls", "-l", "/", NULL);
        cout<<"Main program is over"<<endl;
    }
}
```

Process(C/C++)

```
else if(pID<0){
    cerr<<" Something's wrong! "<<endl;
    exit(1);
}
else{
    process_type = "Parent Process: ";
    process_ID = (int) getppid();

    }
cout<<process_type;
cout<<" Global variable: "<<global;
cout<<" Stack Variable: "<<iStack<<endl;
cout<<" Process ID is "<<process_ID<<endl;
}
```

Process(C/C++)

Parent Process: Global variable: 2 Stack Variable: 50
Process ID is 3008

kaan@Kubuntu:~/Documents/school/540\$ total 96

drwxr-xr-x 2 root root 4096 Apr 5 02:44 bin

drwxr-xr-x 3 root root 4096 Apr 5 02:53 boot

drwxrwxr-x 2 root root 4096 Apr 5 02:18 cdrom

drwxr-xr-x 13 root root 3880 Apr 5 03:02 dev

drwxr-xr-x 135 root root 12288 Apr 5 03:17 etc

drwxr-xr-x 3 root root 4096 Apr 5 02:21 home

lrwxrwxrwx 1 root root 33 Apr 5 02:52 initrd.img ->

boot/initrd.img-3.16.0-33-generic

lrwxrwxrwx 1 root root 33 Apr 5 02:30 initrd.img.old ->

boot/initrd.img-3.16.0-23-generic

Process(C/C++)

Parent Process: Global variable: 2 Stack Variable: 50
Process ID is 3008

kaan@Kubuntu:~/Documents/school/540\$ Child
Process: Global variable: 3 Stack Variable: 51
Process ID is 8884

- Here, we're trying to demonstrate the effects of calling an exec family function on the child process of a parent process.
 - The child process is terminated after the output is posted

Process(Java)

- Before JDK 5.0, the only way to fork off and execute a process was to use the `exec()` method of the `java.lang.Runtime` class
- New ways of executing a command in a separate process involves a class called `ProcessBuilder`.

Process(Java)

- Before calling the `exec()`
 - Specify the command and its arguments
 - Environment variable settings
 - Working directory
- All versions of the method return a `java.lang.Process` object for managing the created process
 - After this, we can get the input/output stream of the subprocess and other information

Process(Java)

- The `ProcessBuilder` class has two constructors
 - `public ProcessBuilder(List command)`
 - Accepts a `List` for the command and its argument
 - `public ProcessBuilder(String... command)`
 - Accepts a variable number of `String` arguments
- Call `start()` to execute the command

Process(Java)

public static String getc_pID() throws IOException,
InterruptedException{

```
    List<String> commands = new ArrayList<String>();  
    commands.add("/bin/bash");  
    commands.add("-c");  
    commands.add("echo $PPID");  
    ProcessBuilder pb = new ProcessBuilder  
(commands);
```

```
    Process pr = pb.start();  
    pr.waitFor();
```

Process(Java)

```
if(pr.exitValue()==0){  
    BufferedReader outReader = new  
BufferedReader(new InputStreamReader(pr.  
getInputStream()));  
    return outReader.readLine().trim();  
}  
else{  
    System.out.println("Error while getting PID");  
    return "";  
}  
}
```

Process(Java)

```
public static void main(String[] args) throws  
IOException, InterruptedException{
```

```
    System.out.println("Process ID of the current process  
is "+getc_pid());  
}
```

- OUTPUT

```
kaan@Kubuntu:~/Documents/school/540$ java  
ProcessTest
```

```
Process ID of the current process is 16751
```

Signals

- Signals are a limited form of interprocess communication used in Unix and other POSIX-compliant operating systems
- Signals are sent to a process when an event occurs
- Signals are how the OS communicates with application process

Signals

- A signal is an asynchronous notification
- Signals have been around since the 1970s Bell Labs Unix
- Recently specified in the POSIX standard.

Signals

- When a signal is sent, the operating system interrupts the target process's normal flow of execution to deliver the signal
- Execution can be interrupted during any non-atomic(uninterruptable) instruction.

Signals Cont

1. Event gains attention of OS
2. OS stops application process immediately, sending it a signal
3. Signal Handler executes
4. Application Resumes where it left off

Signal Options

1. Default action - The process does the default action, which is to stop executing immediately.
2. Ignore - The process can be programmed to catch the signal (called trapping the signal) and ignore it. The process continues running.

Signal Options

3. Catch - The process can be programmed to catch (trap) the signal and do something sensible such as open a file.

Some Signal Examples

Signal	Code	Default Action	Description
SIGHUP	1	TERMINATE	HANGUP
SIGINT	2	TERMINATE	INTERRUPT
SIGQUIT	3	TERMINATE CORE DUMP	QUIT DUMP CORE
SIGILL	4	TERMINATE CORE DUMP	ILLEGAL INSTRUCTION

Common Signals

- /* hangup */
- SIGHUP 1
- /* interrupt */
- SIGINT 2
- /* quit dump core */
- SIGQUIT 3
- /* illegal instruction */
- SIGILL 4

Common Signals Continued

- /* used by abort */
- SIGABRT 6
- /* hard kill */
- SIGKILL 9
- etc
- Signals can be numbered from 0 to 31.

Example 1

This Example prints out some different signals and the UNIX code.

C++ Example

```
#include <csignal>
#include <signal.h>
#include <iostream>
void signal_handler(int signal)
{
    gSignalStatus = signal;
}
```

C++ Cont.

```
int main()
{
    // Install a signal handler
    std::signal(SIGINT, signal_handler);
    std::cout << "SignalValue: " <<
gSignalStatus << '\n';
    std::cout << "SIGNIT " << '\n';
    std::cout << "Sending signal " << SIGINT
<< '\n';
```

C++ Cont.

```
std::raise(SIGINT);  
std::cout << "SignalValue: " <<  
gSignalStatus << '\n';  
std::cout << "SIGABRT " << '\n';  
std::cout << "Sending signal " << SIGABRT  
<< '\n';
```


C++ Cont.

```
std::raise(SIGABRT);  
std::cout << "SignalValue: " <<  
gSignalStatus << '\n';  
  
}
```

C++ Output

```
campos_comp:CS540 campos_luis$ g++  
example2.cpp
```

```
campos_comp:CS540 campos_luis$ ./a.out
```

```
SignalValue: 0
```

```
SIGNIT
```

```
Sending signal 2
```

```
SignalValue: 2
```

```
SIGABRT
```

```
Sending signal 6
```

```
Abort trap: 6
```

Signal Handling

- A Signal Handler is a function in an Application Program that depends on some signal.
- Once a signal is detected the handler has three options

3 options

- The process can let the default action happen for the signal
- The process can block/ignore the signal (some signals cannot be ignored)
- Some action like printing out a message.

Sending Signals

- `kill()` -The kill command can be used to send a signal to a running process
- If you know the process id then you can be more specific
- `int kill(int pid, int signal)` - a system call that send a signal to a process, pid.
- The `raise()` library function sends the specified signal to the current process.

Example 2 Signal Handling

In this example the normal SIGINT signal is disabled.

Normally in terminal CTRL+C quits any process in the terminal.

When CTRL+C is pressed instead of quitting the program prints a message

C++ Example 2

```
#include <csignal>
#include <iostream>
using namespace std;
// SIGINT handler
void int_handler(int x)
{
    cout <<"Your pressed CTL+C which is
disabled to quit press return"<<endl; // after
pressing CTRL+C you'll see this message

}
```

C++ Cont.

```
int main()
{
    signal(SIGINT,int_handler);

    cout <<"Press ctrl+c now\n";
    cin.get();
}
```


Example 2 Output

```
campos_comp:CS540 campos_luis$ g++  
example3.cpp
```

```
campos_comp:CS540 campos_luis$ ./a.out  
Press ctrl+c now
```

```
^CYour pressed CTL+C which is disabled to  
quit press return
```

```
^CYour pressed CTL+C which is disabled to  
quit press return
```

```
^CYour pressed CTL+C which is disabled to  
quit press return
```

Output cont.

^CYour pressed CTL+C which is disabled to
quit press return

^CYour pressed CTL+C which is disabled to
quit press return

^CYour pressed CTL+C which is disabled to
quit press return

^CYour pressed CTL+C which is disabled to
quit press return

Unix Signals with C# (Mono)

C# in UNIX and Mac environments can be possible by using Mono Framework

- Install MonoDevelop on Mac
- Install Xamarin on Linux

Signals (C#)- Ctrl + C Handler

using System

class SampleSignals

```
{  
    public static void Main()  
    {  
        ConsoleKeyInfo cki;  
  
        Console.Clear();  
  
        // Establish an event handler to process key press events.  
        Console.CancelKeyPress += new ConsoleCancelEventHandler  
(myHandler);  
        while (true) {  
            Console.Write("Press any key, or 'X' to quit, or ");  
            Console.WriteLine("CTRL+C to interrupt the read operation:");
```

Signals (C#)- Ctrl + C Handler

```
// Start a console read operation. Do not display the input.  
cki = Console.ReadKey(true);
```

```
// Announce the name of the key that was pressed .  
Console.WriteLine(" Key pressed: {0}\n", cki.Key);
```

```
// Exit if the user pressed the 'X' key.  
if (cki.Key == ConsoleKey.X) break;
```

```
}
```

```
}
```

Signals (C#)- Ctrl + C Handler

```
protected static void myHandler(object sender, ConsoleCancelEventArgs args)
{
    Console.WriteLine("\nThe read operation has been interrupted.");

    Console.WriteLine(" Key pressed: {0}", args.SpecialKey);

    Console.WriteLine(" Cancel property: {0}", args.Cancel);

    // Set the Cancel property to true to prevent the process from terminating.
    Console.WriteLine("Setting the Cancel property to true...");
    args.Cancel = true;

    // Announce the new value of the Cancel property.
    Console.WriteLine(" Cancel property: {0}", args.Cancel);
    Console.WriteLine("The read operation will resume...\n");
}
```

Signals (C#)- Ctrl + C Handler

Output

```
Press any key, or 'X' to quit, or CTRL+C to interrupt the read operation:  
Key pressed: Enter
```

```
Press any key, or 'X' to quit, or CTRL+C to interrupt the read operation:  
Key pressed: Y
```

```
Press any key, or 'X' to quit, or CTRL+C to interrupt the read operation:
```

```
The read operation has been interrupted.
```

```
Key pressed: ControlC
```

```
Cancel property: False
```

```
Setting the Cancel property to true...
```

```
Cancel property: True
```

```
The read operation will resume...
```

```
Key pressed: D
```

```
Press any key, or 'X' to quit, or CTRL+C to interrupt the read operation:
```

```
Key pressed: G
```

```
Press any key, or 'X' to quit, or CTRL+C to interrupt the read operation:
```



Signals(Java)

- Java provides no standard interface for a Java application to hear or react to POSIX signals
- To write a java signal handler, define a class that implements `sun.misc.SignalHandler` interface
- Register the handler by using the `sun.misc.Signal.handle()` method

Signals(Java)

- There are certain advantages of using Java signal handlers instead of native signal handlers
 - implementation can be completely in Java, keeping the application simple
 - with Java signal handling, you keep an object-oriented approach and refer to signals by name
 - more readable code than C equivalent

Signals(Java)

- sun.misc package contains undocumented support classes that may change between releases of Java
 - Hence the warning while the code is being compiled
 - “warning: Signal Handler is internal proprietary API and may be removed in a future release”

Signals(Java)

```
import sun.misc.Signal;
import sun.misc.SignalHandler;
public class signals implements SignalHandler {
    private SignalHandler oldHandler;

    public static SignalHandler install(String signalName)
    {
        Signal diagSignal = new Signal(signalName);
        signals instance = new signals();
        instance.oldHandler = Signal.handle(diagSignal,
instance);
        return instance;
    }
}
```

Signals(Java)

```
public void signalAction(Signal signal) {  
    System.out.println("Handling " + signal.getName());  
    System.out.println("Just sleep for 5 seconds.");  
    try {  
        Thread.sleep(5000);  
    } catch (InterruptedException e) {  
        System.out.println("Interrupted: "  
            + e.getMessage());  
    }  
}
```

Signals(Java)

```
public void handle(Signal signal) {  
    System.out.println("Signal handler called for signal "  
        + signal);  
    try {  
  
        signalAction(signal);  
  
        // Chain back to previous handler, if one exists  
        if (oldHandler != SIG_DFL && oldHandler !=  
SIG_IGN) {  
            oldHandler.handle(signal);  
        }  
    }  
}
```

Signals(Java)

```
} catch (Exception e) {  
    System.out.println("handle|Signal handler failed,  
reason " + e.getMessage());  
    e.printStackTrace();  
}  
}
```

```
public static void main(String[] args) {  
    signals.install("TERM");  
    signals.install("INT");  
    signals.install("ABRT");  
}
```

Signals(Java)

- ```
System.out.println("Signal handling example.");
try {
 Thread.sleep(50000);
} catch (InterruptedException e) {
 System.out.println("Interrupted: " + e.
getMessage());
}
}
```
- ```
kaan@Kubuntu:~/Documents/school/540$ java signals
Signal handling example
^CSignal handler called for signal SIGINT
Handling INT
Just sleep for 5 seconds
```

Pipelines

In computing, a **pipeline** is a set of data processing elements connected in series, where the output of one element is the input of the next one.

The elements of a pipeline are often executed in parallel or in time-sliced fashion; in that case, some amount of buffer storage is often inserted between elements. (Wiki)

Pipelines

Software pipelines, where commands can be written where the output of one operation is automatically fed to the next, following operation. The Unix system call **pipe** is a classic example of this concept, although other operating systems do support pipes as well.

Pipelines in Linux systems

Pipes were first suggested by **M. Doug McIlroy**, when he was a department head in the Computing Science Research Center at Bell Labs, the research arm of AT&T (American Telephone and Telegraph Company).

McIlroy had been working on *macros* since the latter part of the 1950s, and he was a ceaseless advocate of linking macros together as a more efficient alternative to series of discrete commands. A macro is a series of commands (or keyboard and mouse actions) that is performed automatically when a certain command is entered or key(s) pressed.

McIlroy's persistence led **Ken Thompson**, who developed the original **UNIX at Bell Labs in 1969**, to rewrite portions of his operating system in **1973** to include pipes.

This implementation of pipes was not only extremely useful in itself, but it also made possible a central part of the *Unix philosophy*, the most basic concept of which is *modularity* (i.e., a whole that is created from independent, replaceable parts that work together efficiently).

Pipelines in Linux systems

Linux commands:

cat - Concatenate files

sort - Sort lines of text

uniq - Report or omit repeated lines

grep - Print lines matching a pattern

wc - Print newline, word, and byte counts for each file

head - Output the first part of a file

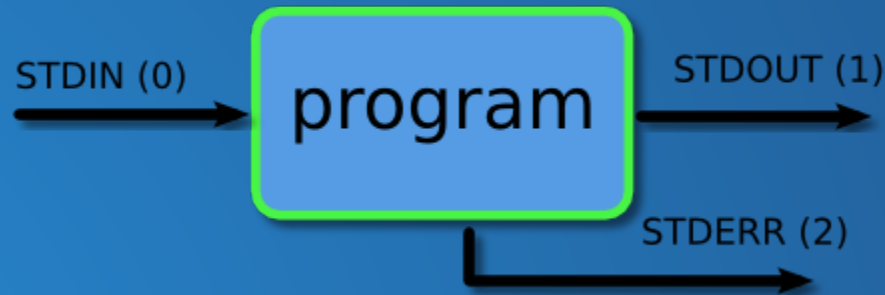
tail - Output the last part of a file

tee - Read from standard input and write to standard output and files

Pipelines in Linux systems

Every program we run on the command line automatically has three data streams connected to it.

- STDIN (0) - Standard input (data fed into the program)
- STDOUT (1) - Standard output (data printed by the program, defaults to the terminal)
- STDERR (2) - Standard error (for error messages, also defaults to the terminal)



Piping

The general syntax for pipes is:

```
command_1 | command_2 [| command_3 . . . ]
```

1. **user@user: ls**
2. barry.txt bob example.png firstfile foo1 myoutput video.mpeg
3. **user@user: ls | head -3**
4. barry.txt
5. bob
6. example.png

1. **user@user: ls | head -3 | tail -1**
2. example.png

This shows how you can run an external system command from within a Java program.

```
import java.io.*;
public class JavaRunCommand {
    public static void main(String args[]) {
        String s = null;
        try {
            // run the Unix "ps -ef" command
            // using the Runtime exec method:
            Process p = Runtime.getRuntime().exec("ps -ef");
            BufferedReader stdInput = new BufferedReader(new
                InputStreamReader(p.getInputStream()));
            BufferedReader stdError = new BufferedReader(new
                InputStreamReader(p.getErrorStream()));
            // read the output from the command
            System.out.println("Here is the standard output of
the command:\n");
            while ((s = stdInput.readLine()) != null) {
                System.out.println(s);
            }
        }
    }
}
```


```
// read any errors from the attempted command
    System.out.println("Here is the standard error of
the command (if any):\n");
    while ((s = stdError.readLine()) != null) {
        System.out.println(s);
    }
    System.exit(0);
}
catch (IOException e) {
    System.out.println("exception happened - here's what
I know: ");
    e.printStackTrace();
    System.exit(-1);
}
}
```

Command line in Linux Using Java

ProcessBuilderExample

```
import java.io.IOException;
import java.util.*;


public class ProcessBuilderExample
{
    public static void main(String[] args) throws
    Exception
    {
        new ProcessBuilderExample();
    }
    // can run basic ls or ps commands
    // can run command pipelines
    // can run sudo command if you know the password
    is correct
```

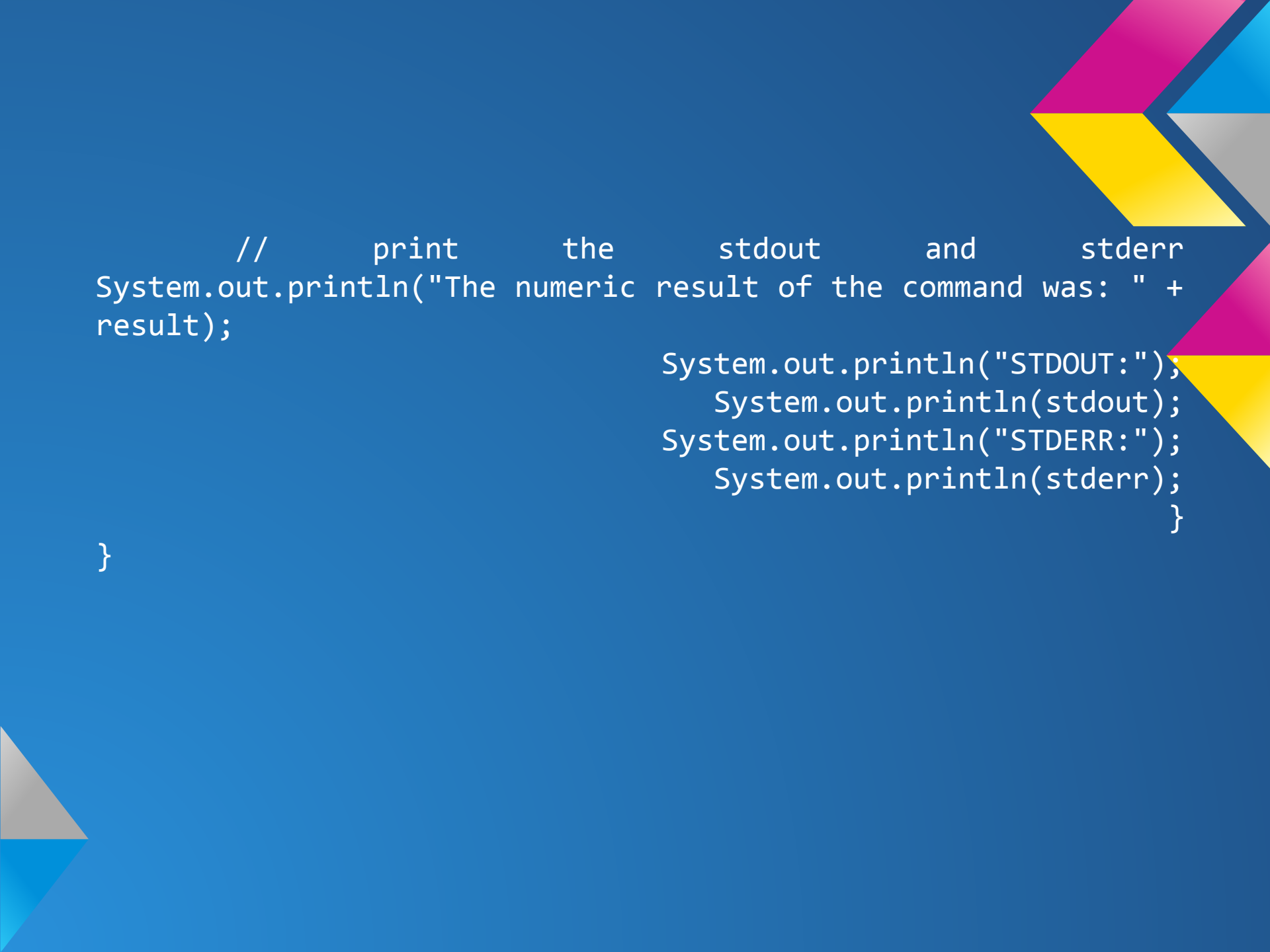



```
public ProcessBuilderExample() throws IOException,
InterruptedException
{
    // build the system command we want to run
    List<String> commands = new ArrayList<String>();
    commands.add("/bin/sh");
    commands.add("ls | head -3 | tail -1");
    // execute the command

    SystemCommandExecutor commandExecutor = new SystemCommandExecutor
(commands);

    int result = commandExecutor.executeCommand();
    // get the stdout and stderr from the command that was run
    StringBuilder stdout = commandExecutor.
getStandardOutputFromCommand();
    StringBuilder stderr = commandExecutor.
getStandardErrorFromCommand();
```



The background is a solid blue color. In the top right corner, there are several overlapping, semi-transparent geometric shapes in shades of pink, yellow, and light blue. In the bottom left corner, there are similar overlapping shapes in shades of light blue and white.

```
//      print      the      stdout      and      stderr
System.out.println("The numeric result of the command was: " +
result);

System.out.println("STDOUT:");
System.out.println(stdout);
System.out.println("STDERR:");
System.out.println(stderr);
}

}
```

SystemCommandExecutor

```
import java.io.*;
import java.util.List;

public class SystemCommandExecutor
{
    private List<String> commandInformation;
    private String adminPassword;
    private ThreadedStreamHandler inputStreamHandler;
    private ThreadedStreamHandler errorStreamHandler;

    /**
     * Pass in the system command you want to run as a List of Strings, as shown
     here:
     *
     * List<String> commands = new ArrayList<String>();
     * commands.add("/sbin/ping");
     * commands.add("-c");
     * commands.add("5");
     * commands.add("www.google.com");
     * SystemCommandExecutor commandExecutor = new SystemCommandExecutor(commands);
     * commandExecutor.executeCommand();
     *
     * @param commandInformation The command you want to run.
     */
}
```

```
public SystemCommandExecutor(final List<String> commandInformation)
{
    if (commandInformation==null) throw new NullPointerException("The
commandInformation is required.");
    this.commandInformation = commandInformation;
    this.adminPassword = null;
}
public int executeCommand()
throws IOException, InterruptedException
{
    int exitValue = -99;

    try
    {
        ProcessBuilder pb = new ProcessBuilder(commandInformation);
        Process process = pb.start();

        // you need this if you're going to write something to the command's input
stream
        // (such as when invoking the 'sudo' command, and it prompts you for a
password).
        OutputStream stdout = process.getOutputStream();

        InputStream inputStream = process.getInputStream();
        InputStream errorStream = process.getErrorStream();
```

```
// these need to run as java threads to get the standard output and error
from the command.
// the inputStream handler gets a reference to our stdout in case we need
to write
// something to it, such as with the sudo command
inputStreamHandler = new ThreadedStreamHandler(inputStream, stdout,
adminPassword);
errorStreamHandler = new ThreadedStreamHandler(errorStream);

// TODO the inputStreamHandler has a nasty side-effect of hanging if the
given password is wrong; fix it
inputStreamHandler.start();
errorStreamHandler.start();

// TODO a better way to do this?
exitValue = process.waitFor();

// TODO a better way to do this?
inputStreamHandler.interrupt();
errorStreamHandler.interrupt();
inputStreamHandler.join();
errorStreamHandler.join();
}
catch (IOException e)
{
    throw e;
}
catch (InterruptedException e)
{
    throw e;
}
```

```
finally
{
    return exitValue;
}

/**
 * Get the standard output (stdout) from the command you just exec'd.
 */
public StringBuilder getStandardOutputFromCommand()
{
    return inputStreamHandler.getOutputBuffer();
}

/**
 * Get the standard error (stderr) from the command you just exec'd.
 */
public StringBuilder getStandardErrorFromCommand()
{
    return errorStreamHandler.getOutputBuffer();
}
}
```

ThreadedStreamHandler

```
import java.io.*;

class ThreadedStreamHandler extends Thread
{
    InputStream inputStream;
    String adminPassword;
    OutputStream outputStream;
    PrintWriter printWriter;
    StringBuilder outputBuffer = new StringBuilder();
    private boolean sudoIsRequested = false;

    /**
     * A simple constructor for when the sudo command is not necessary.
     * This constructor will just run the command you provide, without
     * running sudo before the command, and without expecting a password.
     *
     * @param inputStream
     * @param streamType
     */
}
```

```
ThreadedStreamHandler(InputStream inputStream)
{
    this.inputStream = inputStream;
}

/**
 * Use this constructor when you want to invoke the 'sudo' command.
 * The outputStream must not be null. If it is, you'll regret it. :)
 *
 * TODO this currently hangs if the admin password given for the sudo command is
wrong.
 *
 * @param inputStream
 * @param streamType
 * @param outputStream
 * @param adminPassword
 */
ThreadedStreamHandler(InputStream inputStream, OutputStream outputStream, String
adminPassword)
{
    this.inputStream = inputStream;
    this.outputStream = outputStream;
    this.printWriter = new PrintWriter(outputStream);
    this.adminPassword = adminPassword;
    this.sudoIsRequested = true;
}
```



```
public void run()
{
    // on mac os x 10.5.x, when i run a 'sudo' command, i need to write
    // the admin password out immediately; that's why this code is
    // here.
    if (sudoIsRequested)
    {
        //doSleep(500);
        printWriter.println(adminPassword);
        printWriter.flush();
    }

    BufferedReader bufferedReader = null;
    try
    {
        bufferedReader = new BufferedReader(new InputStreamReader(inputStream));
        String line = null;
        while ((line = bufferedReader.readLine()) != null)
        {
            outputBuffer.append(line + "\n");
        }
    }
}
```

```
catch (IOException ioe)
{
    ioe.printStackTrace();
}
catch (Throwable t)
{
    t.printStackTrace();
}
finally
{
    try
    {
        bufferedReader.close();
    }
    catch (IOException e)
    {
        // ignore this one
    }
}
}
```

```
private void doSleep(long millis)
{
    try
    {
        Thread.sleep(millis);
    }
    catch (InterruptedException e)
    {
        // ignore
    }
}

public StringBuilder getOutputBuffer()
{
    return outputBuffer;
}
}
```

Fake Pipes

Sometimes the pipe-like mechanism used in **MS-DOS** is referred to as ***fake pipes*** because, instead of running two or more programs simultaneously and channeling the output data from one continuously to the next, **MS-DOS** uses a temporary ***buffer file*** (i.e., section of memory) that first accumulates the entire output from the first program and only then feeds its contents to the next program.

- takes more time (because the second program cannot begin until the first has been completed)
- consume more system resources (i.e., memory and processor time).
- if the first command produces a very large amount of output and/or does not terminate.

Pipe in Java

Pipes in Java IO provides the ability for two threads running in the same JVM to communicate. Therefore pipes can also be sources or destinations of data.

You cannot use a pipe to communicate with a thread in a different JVM (different process). The pipe concept in Java is different from the pipe concept in Unix / Linux, where two processes running in different address spaces can communicate via a pipe. In Java, the communicating parties must be running in the same process, and should be different threads.

Java class:

PipedOutputStream

PipedInputStream

```
import java.io.*;
public class PipeExample {
    public static void main(String[] args) throws IOException {
        final PipedOutputStream output = new PipedOutputStream();
        final PipedInputStream input = new PipedInputStream(output);
        Thread thread1 = new Thread(new Runnable() {
            public void run() {
                try {
                    output.write("Hello world, pipe!".getBytes());
                } catch (IOException e) {
                }
            }
        });
        Thread thread2 = new Thread(new Runnable() {
            public void run() {
                try {
                    int data = input.read();
                    while(data != -1){
                        System.out.print((char) data);
                        data = input.read();
                    }
                } catch (IOException e) {
                }
            }
        });
        thread1.start();
        thread2.start();
    }
}
```

Pipes in C#

- Pipes provide a means for interprocess communication
 - Two Types:
 - Anonymous pipes
 - Named pipes

Anonymous Pipes in C#

- provide interprocess communication on a local computer
- require less overhead than named pipes but offer limited services
- one-way and cannot be used over a network
- support only a single server instance

Anonymous Pipes in C#

- useful for communication between threads, or between parent and child processes where the pipe handles can be easily passed to the child process when it is created.
- implemented using:
 - *AnonymousPipeServerStream*
 - *AnonymousPipeClientStream*

Named Pipes in C#

- provide interprocess communication between a pipe server and one or more pipe clients
- can be one-way or duplex
- support message-based communication and allow multiple clients to connect simultaneously to the server process using the same pipe name

Named Pipes in C#

- support impersonation, which enables connecting processes to use their own permissions on remote servers.
- implemented using:
 - *NamedPipeServerStream*
 - *NamedPipeClientStream*

Pipeline C# Example - Server

```
using System;
using System.IO;
using System.IO.Pipes;
using System.Diagnostics;

class PipeServer
{
    static void Main()
    {
        Process pipeClient = new Process();

        pipeClient.StartInfo.FileName = "pipeClient.exe";

        // PIPE DIRECTION OUT
        // Handle Inheritability determines whether the underlying handle can
        be inherited by the child
        using (AnonymousPipeServerStream pipeServer = new
        AnonymousPipeServerStream(PipeDirection.Out,
            HandleInheritability.Inheritable))
        {
```

Pipeline C# Example - Server

```
Console.WriteLine("SERVER: Tran Mode: {0}.",  
    pipeServer.TransmissionMode);  
  
// Pass the client process a handle to the server.  
pipeClient.StartInfo.Arguments =  
    pipeServer.GetClientHandleAsString();  
pipeClient.StartInfo.UseShellExecute = false;  
pipeClient.Start();  
  
pipeServer.DisposeLocalCopyOfClientHandle();  
Console.WriteLine("SERVER: Tran Mode: {0}.",  
    pipeServer.TransmissionMode);  
  
// Pass the client process a handle to the server.  
pipeClient.StartInfo.Arguments =  
    pipeServer.GetClientHandleAsString();  
pipeClient.StartInfo.UseShellExecute = false;  
pipeClient.Start();  
  
pipeServer.DisposeLocalCopyOfClientHandle();
```

Pipeline C# Example - Server

```
try{
    using (StreamWriter sw = new StreamWriter(pipeServer))
    {
        sw.AutoFlush = true;

        // Send a 'sync message' and wait for client to receive
        sw.WriteLine("CONNECT");
        pipeServer.WaitForPipeDrain();
        string temp = "";

        do{
            // Ask user to input text
            Console.Write("SERVER: Enter text to send to client");

            //Read user console input and send to client (via pipe)
            temp = Console.ReadLine();

            if(!temp.Equals("quit")){sw.WriteLine(temp);}
            //Sleep for 2 seconds
            System.Threading.Thread.Sleep(2000);
        }while (!temp.Equals("quit"));
    }
}
```

Pipeline C# Example - Server

```
catch (IOException e)
{
    Console.WriteLine("SERVER: Somethings wrong! Error: {0}", e.Message);
}

pipeClient.WaitForExit();
pipeClient.Close();
Console.WriteLine("SERVER: Client quit. Server terminating.");
}
```

Pipeline C# Example - Client

```
using System;
using System.IO;
using System.IO.Pipes;

class PipeClient
{
    static void Main(string[] args)
    {
        if (args.Length > 0)
        {

            // PIPE DIRECTION IN
            using (PipeStream pipeClient =
                new AnonymousPipeClientStream(PipeDirection.In, args[0]))
            {

                Console.WriteLine("CLIENT: Trans Mode: {0}.",
                    pipeClient.TransmissionMode);
            }
        }
    }
}
```

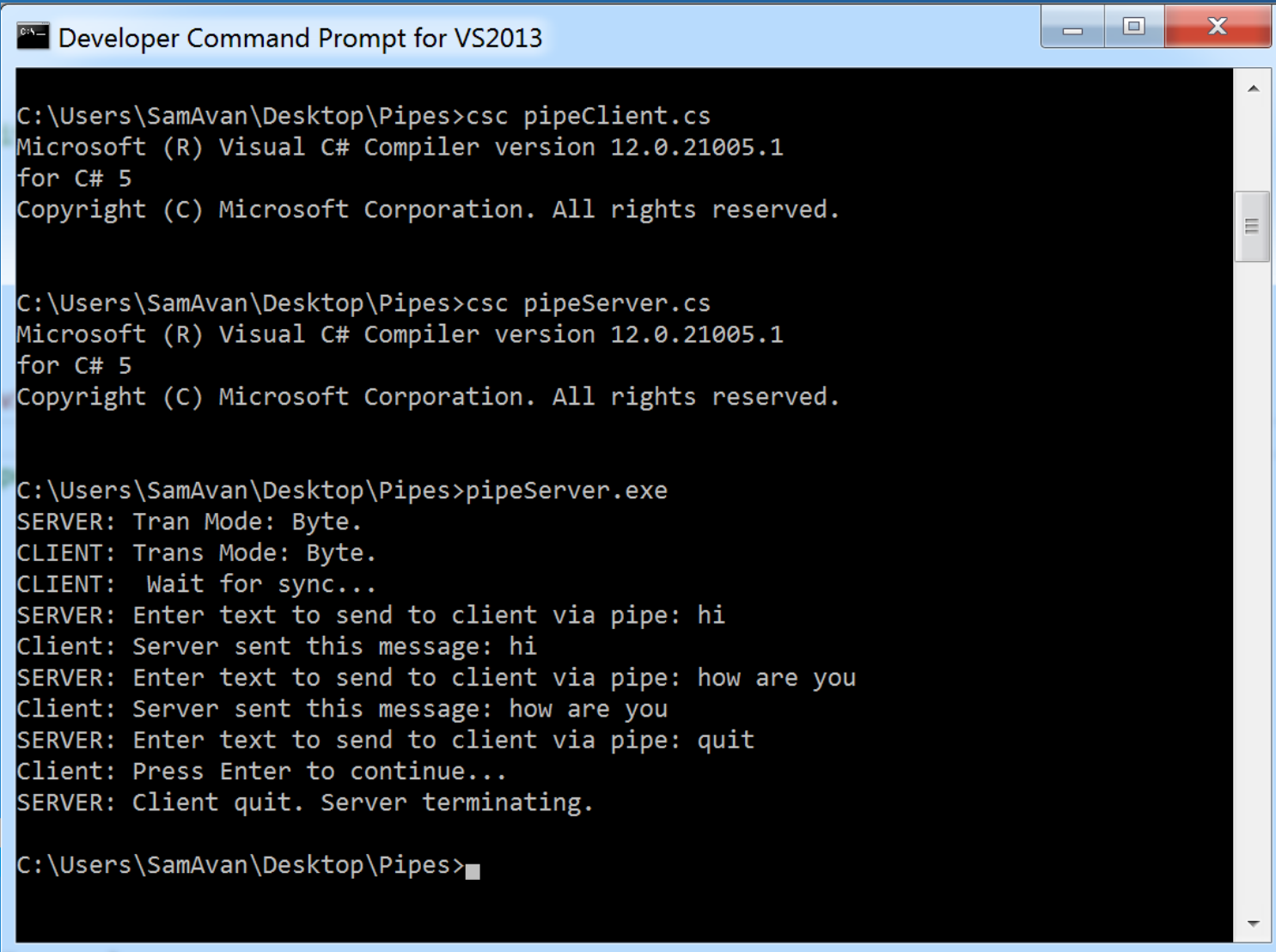
Pipeline C# Example - Client

```
using (StreamReader sr = new StreamReader(pipeClient))
{
    string temp;

    // Wait for 'sync message' from the server.
    do
    {
        Console.WriteLine("CLIENT:  Wait for sync...");
        temp = sr.ReadLine();
    }
    while (!temp.StartsWith("CONNECT"));

    // Read the server data and echo to the console.
    while ((temp = sr.ReadLine()) != null)
    {
        Console.WriteLine("Client: Server sent this message: " +
temp);
        }}
    }
    Console.Write("Client: Press Enter to continue...");
    Console.ReadLine();
}
```


Pipeline C# Example - Output



```
Developer Command Prompt for VS2013

C:\Users\SamAvan\Desktop\Pipes>csc pipeClient.cs
Microsoft (R) Visual C# Compiler version 12.0.21005.1
for C# 5
Copyright (C) Microsoft Corporation. All rights reserved.

C:\Users\SamAvan\Desktop\Pipes>csc pipeServer.cs
Microsoft (R) Visual C# Compiler version 12.0.21005.1
for C# 5
Copyright (C) Microsoft Corporation. All rights reserved.

C:\Users\SamAvan\Desktop\Pipes>pipeServer.exe
SERVER: Tran Mode: Byte.
CLIENT: Trans Mode: Byte.
CLIENT: Wait for sync...
SERVER: Enter text to send to client via pipe: hi
Client: Server sent this message: hi
SERVER: Enter text to send to client via pipe: how are you
Client: Server sent this message: how are you
SERVER: Enter text to send to client via pipe: quit
Client: Press Enter to continue...
SERVER: Client quit. Server terminating.

C:\Users\SamAvan\Desktop\Pipes>
```

Pipes C++

```
#include <iostream>  
using namespace std;
```

```
#include <stdio.h>  
#include <unistd.h>  
int main()  
{
```

C++ Cont.

```
int status, pid, pipefds[2];  
char instring[20];
```

```
status = pipe(pipefds);  
if (status == -1)  
{  
    perror("Trouble");  
    exit(1);  
}
```

C++ Cont.

```
pid = fork();  
    if (pid == -1)  
    {  
        perror("Trouble");  
        exit(2);  
    }
```

C++ Cont.

```
else if (pid == 0)
{
    close(pipefds[0]);
    cout << "About to send a message: " <<
endl;
        write(pipefds[1], "Hello World!", 12);

    close(pipefds[1]);
    exit(0);
}
```

C++ Cont.

```
else
{
    close(pipefds[1]);
    read(pipefds[0], instring, 12);
    cout << "Just received a message that
says: " << instring << endl;
    close(pipefds[0]);
    exit(0);
}
```

Pipes Output

```
campos_comp:CS540 campos_luis$ g++  
pipes.cpp
```

```
campos_comp:CS540 campos_luis$ ./a.out
```

About to send a message:

Just received a message that says: Hello
World!?

CS 540 Spring 2015 GitHub Repository

<https://github.com/burgui/CS540-Spring-2015>