

# CSCI 3310

# Programming

# Introduction

MARK GREEN

ONTARIO TECH UNIVERSITY

# Linux

- ▶ All the programming in this course will be done in the Linux environment
- ▶ I assume that you have some basic familiarity with Linux from a previous course
- ▶ If you already have Linux on your laptop, then you are ready to go for the assignments and labs
- ▶ If you are only running Windows, you will need to prepare a Linux environment

# Linux

- ▶ For Windows users there are two approaches that you can use
- ▶ The first is to run a virtual machine, at the present time the best one seems to be VirtualBox (<https://www.virtualbox.org/>)
- ▶ I don't particularly like Oracle, but the alternatives seem to be worse
- ▶ I don't recommend using a virtual machine, students who tried this last year had problems with some of the labs and assignments
- ▶ An alternative is to use Windows Subsystem for Linux(WSL) (<https://docs.microsoft.com/en-us/windows/wsl/install-win10>)
- ▶ You want to use version 2.0 of WSL, this seems to work well, but it only gives you a command line interface

# POSIX

- ▶ The POSIX (Portable Operating System Interface) is an attempt to provide a uniform API for Unix like operating systems, and IEEE standard
- ▶ Linux is largely POSIX compliant, there are aspects of Windows that are as well
- ▶ It was originally developed during the Unix wars to assist with program portability, at least at the source level
- ▶ We will largely be sticking to POSIX and ignore many of the Linux extensions

# Dilemma

- ▶ I'm in a bit of a dilemma for this course
- ▶ Normally we simplify examples so they are easy to understand
- ▶ We almost always ignore error checking, and quite often don't allocate memory correctly
- ▶ These things can take up a lot of code, hard to see the main flow of the logic
- ▶ In this course we are really concerned with these things, production quality code that has fewer security holes
- ▶ We need to check for all possible error situations

# Dilemma

- ▶ If I'm illustrating an important concept I may be sloppy and omit error checking
- ▶ At other times we will discuss all the possible errors that could occur and how we can handle them
- ▶ You will be responsible for handling all possible error conditions in your assignments
- ▶ It is important that you don't just cut and paste code from the labs, you need to understand what's happening and where you may need to add extra error checking
- ▶ This is a warning to those of you who code by Google

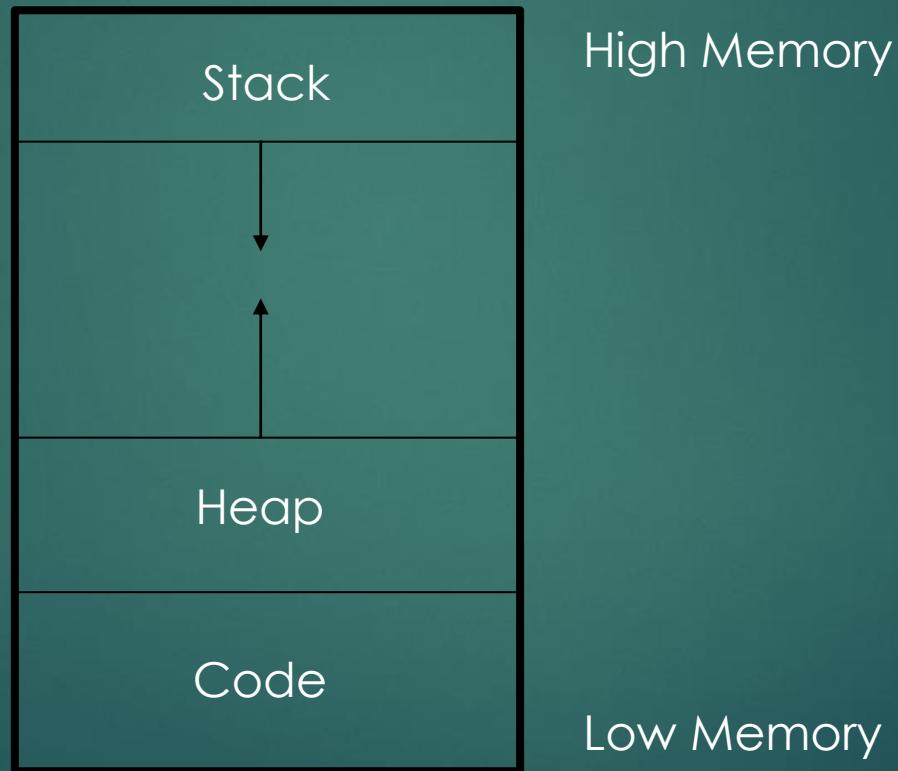
# C

- ▶ We will be using the C programming language
- ▶ The Linux API is a C API, so that pretty much forces the issue
- ▶ Some of the concepts are easier to explain with C than higher order languages that use more complicated techniques
- ▶ You've already seen C++, C is just a subset of C++ without objects and a few other features
- ▶ There is a difference with memory allocation, which we will cover next
- ▶ The first two chapters of Dive Into Systems provides a good review of the C programming language

# C Memory Allocation

- ▶ Linux system calls assume that all of their parameters are words, so we can only pass integers, floating point numbers and pointers
- ▶ If we need to pass a buffer or a structure we need to pass a pointer
- ▶ So we need a good understanding of pointers and memory allocation
- ▶ Start by examining the runtime structure of a process
- ▶ We have three types of memory as shown on the next slide
- ▶ The program code is in low memory, this is usually write protected so the program can't accidentally overwrite itself

# C Memory Allocation



# C Memory Allocation

- ▶ Above the code segment is the heap segment, this is where global variables are stored along with any dynamically allocated memory
- ▶ The heap has a structure that we won't go into in detail
- ▶ The heap grows up through memory space as memory is dynamically allocated
- ▶ The stack segment is at the top of memory and grows down as procedures are called
- ▶ This is where local variables are stored
- ▶ Whenever a procedure is called a stack frame is constructed, contains all the information for that procedure call

# C Memory Allocation

- ▶ Stack frame:
  - ▶ Return address
  - ▶ Return value
  - ▶ Parameters to procedure
  - ▶ Procedures local variables
  - ▶ Any temporary values needed by the procedure
- ▶ If the stack and heap collide you are out of memory and the process terminates

# C Memory Allocation

- ▶ On a modern 64bit computer a process has a larger memory space than there is real memory on the computer
- ▶ When the stack and heap grow new memory is allocated by the OS, this allocation is in terms of pages
- ▶ There isn't real memory backing up the entire memory space
- ▶ This has several implications:
  - ▶ A pointer can access a memory location that hasn't been allocated – this causes a segmentation fault
  - ▶ You will never get the full address space, the process will be terminated long before that

# C Memory Allocation

- ▶ There are three memory allocation procedures:
  - ▶ malloc – allocate a new block of memory
  - ▶ free – release a block of memory
  - ▶ realloc – change the size of a allocated block of memory
- ▶ malloc and free are the procedures that are most commonly used
- ▶ If we go to the man page for malloc we find the following declaration

```
void *malloc(size_t size);
```
- ▶ Remember man is your friend, it has the documentation for procedures you need

# C Memory Allocation

- ▶ The `size_t` type is the size of a machine word, this makes code portable between 32bit and 64bit operating systems
- ▶ The parameter to `malloc` is the number of bytes of memory that is required
- ▶ If `malloc` is successful you will get at least this amount of memory
- ▶ `malloc` returns a `void*`, which is a pointer to nothing, or a pointer to everything, depending upon your view of life
- ▶ This value can then be cast to any type of value that you like

# C Memory Allocation

- ▶ The pointer returned by malloc is aligned to the largest common data structure used by the OS
- ▶ For 64bit operating systems, this will typically be on an 8 byte boundary, it will not be smaller than this
- ▶ This ensures that all operations through the pointer will be time efficient, but could be space inefficient
- ▶ Lesson: don't allocate memory one byte at a time
- ▶ Think carefully before you allocate memory, this is a relatively expensive operation

# C Memory Allocation

- ▶ The free procedure is used to return allocated memory, its declaration is:
- ```
void free(void *ptr);
```
- ▶ The parameter to this procedure is a pointer to the block of memory to be freed
  - ▶ You can only free a block of memory once, otherwise the data structures used by malloc and free become corrupt
  - ▶ The realloc procedure is used to change the size of a block of allocated memory, its declaration is:

```
void *realloc(void *ptr, size_t size);
```

# C Memory Allocation

- ▶ The first parameter is a pointer to the block of memory to be changed and the second parameter is the new size
- ▶ The new size can be either smaller or larger than the current size
- ▶ The return value is a pointer to the new block of memory, this may not be the same as the pointer that is passed into this procedure
- ▶ `realloc` will copy as much of the old block of memory into the new block as possible
- ▶ If the new block is larger the whole contents on the old block will be copied, otherwise it will be truncated

# C Memory Allocation

- ▶ What can go wrong?
- ▶ If the OS cannot allocate more memory then malloc and realloc will return 0 and set errno to ENOMEM
- ▶ What is errno?
- ▶ This is a global variable that is available to all programs that indicates if an error occurred on a previous call to a system procedure, more on this later
- ▶ If you abuse a pointer then all bets are off and your program is likely to crash in an unexpected way
- ▶ This is one of the hardest bugs to find

# C Memory Allocation

- ▶ To see where problems can occur with dynamically allocated memory we need to examine how malloc and free keep track of blocks of memory that have been allocated
- ▶ When a block of memory is returned using free(), we need to know how big this block is
- ▶ Usually this information is stored just before or just after the block of memory, it rides along with the memory
- ▶ This is why you don't want to go outside of the memory block that you allocated, this will overwrite the information used by malloc() and free()

# C Memory Allocation

- ▶ Example:

```
char *p;
```

```
p = (char*) malloc(100);
```

```
p[-1] = 5; // bad, overwrites data before beginning of block
```

```
p[100] = 5 //bad, overwrites data after the block
```

- ▶ Problems will occur when the memory block is returned, if you are really lucky free() will immediately crash
- ▶ Most of the time the problem won't occur until later, called heap corruption
- ▶ This is very hard to debug

# C Memory Allocation

- ▶ `free()` will try to merge blocks of memory when they are returned
- ▶ Usually has a linked list of free memory blocks, when block is returned will merge into this list
- ▶ Try to form large blocks, otherwise the heap becomes fragmented
- ▶ May not be able to allocate a block of memory, even if there is enough free memory
- ▶ `malloc()` will also look for a block that is close to the requested size, again to reduce fragmentation
- ▶ Thus, these operations are expensive

# C Memory Allocation

- ▶ Trick: if there are several common memory block sizes build a layer on top of malloc() and free()
- ▶ Example: use a lot of 512 and 1024 byte blocks
- ▶ Maintain a linked list of both block sizes, this can be pre-allocated at the start of the program
- ▶ When a request comes in check the list first and use the first block on the list
- ▶ When the block is freed add it to the front of the list
- ▶ This is at least and order of magnitude faster than using malloc() and free()

# C Memory Allocation

- ▶ Another problem: memory leaks, memory is allocated but never freed
- ▶ For a short assignment this isn't a problem, you've got lots of memory on your laptop
- ▶ But, for a server that must run continuously for months even a small leak can be a big problem
- ▶ This will eventually cause a crash and the service will not be available
- ▶ For a complex program memory leaks can be very hard to fix, careful design is a must

# C Memory Allocation

- ▶ How can you test for a memory leak?
- ▶ There are several programs that will point to a memory leak:
  - ▶ top under the %mem heading will give an indication of the percentage of available memory being used
  - ▶ ps l under the VSZ heading gives the total virtual memory space used by the program
  - ▶ memusage will print memory statistics as long as the program terminates normally
- ▶ If these numbers are increasing over time you have a memory leak

# C Memory Allocation

- ▶ There are a number of tools that assist with finding memory allocation problems, just examine a few of them
- ▶ malloc and free normally crash when they find that the heap has been corrupted
- ▶ This behavior can be changed in two ways:
  - ▶ The M\_CHECK\_ACTION parameter to mallopt
  - ▶ The MALLOC\_CHECK\_ environment variable
- ▶ The later approach can be used without changing the program code and recompiling
- ▶ Both use the same set of values

# C Memory Allocation

- ▶ The possible values and their effects are:

```
0 Ignore error conditions; continue execution (with undefined results).  
1 Print a detailed error message and continue execution.  
2 Abort the program.  
3 Print detailed error message, stack trace, and memory mappings, and abort the program.  
5 Print a simple error message and continue execution.  
7 Print simple error message, stack trace, and memory mappings, and abort the program.
```

- ▶ You are not allowed to use 0 in your assignments, this is useful when you have a large number of bugs and you want to skip the memory ones for now

# C Memory Allocation

- ▶ This gives some information, but probably not where the error actually occurred
- ▶ More detailed information is provided by the mtrace() function
- ▶ This should be called at the start of your program and it will instrument all the malloc and free calls
- ▶ It records each call on the file specified by the MALLOC\_TRACE environment variable
- ▶ This is a text file that can be analyzed by the mtrace Perl script
- ▶ The mcheck() family of procedures are also useful in tracking down memory problems
- ▶ valgrind is another good option, but is slow, see Dive Into Systems Chapter Three for a description of how to use it

# System Calls

- ▶ The interface between your program and the operating system
- ▶ On Linux there are more than 100 system calls, we probably won't look at all of them
- ▶ They are all described in section two of the manual
- ▶ A system call is more complicated than a normal procedure call
- ▶ It starts out as a procedure call, usually to a stub procedure that is at least partially written in assembler
- ▶ This procedure needs to handle the transition from user mode to supervisor mode

# System Calls

- ▶ Supervisor mode has access to the entire machine, all of memory, all devices, instructions that user programs aren't allowed to use
- ▶ We can't simply call a procedure in the kernel, this would be a security hole and difficult to implement
- ▶ There is a special instruction, usually called TRAP that does this transition for us
- ▶ Depending upon the machine architecture and OS, the parameters to the system call are put into registers along with the number of the system call
- ▶ Then the TRAP instruction is executed

# System Calls

- ▶ Executing TRAP switches to kernel space and calls the trap handler
- ▶ This could involve setting up a new set of registers and memory mapping hardware, this is an expensive operation, depending upon the processor
- ▶ The trap handler then determines the kernel procedure to execute
- ▶ If the procedure will take a long time, for example a read(), your program loses control of the CPU and its handed to another program
- ▶ When the kernel procedure is finished the whole process is reversed

# System Calls

- ▶ There are several implications of this process:
  - ▶ System calls are more expensive than normal procedures
  - ▶ Limited amount of information can be passed, must fit in a register, usually only word length data items
  - ▶ A system call will usually force a context switch
- ▶ Be careful with how you use system calls
- ▶ Don't read one byte at a time, read a reasonable size buffer
- ▶ This is an area where you can significantly improve program performance

# System Calls

- ▶ How do we know whether an error has occurred?
- ▶ There is limited bandwidth between the kernel and our program, the kernel can't print an error message
- ▶ Where would it print the message, particularly for a server that isn't associated with a terminal??
- ▶ The kernel does return error information
- ▶ Most system calls have a return value that indicates whether an error occurred and in many cases the actual error
- ▶ This is a negative integer

# System Calls

- ▶ To make error checking somewhat easier there is a global variable, `errno` that is set to a non-zero value to indicate the error that has occurred
- ▶ All the possible error codes are listed in ***man 3 errno*** along with a short description of the error
- ▶ Note: a successful system call doesn't change the value of `errno`, so if you are not careful the error could have been generated by a previous system call
- ▶ The standard procedure is to set `errno` to 0 before a call and then check its value afterwards

# System Calls

- ▶ Some of the errors that are returned are not really errors, but an indication of non-standard behavior
- ▶ We will see examples of this later
- ▶ A program can handle these situations by examining the value of errno and then taking an appropriate action
- ▶ While the integer error number is useful within our program this isn't something we would like to tell the user
- ▶ We need something that is more informative

# System Calls

- ▶ There are two easy ways of getting a reasonable error message
- ▶ The variable `sys_errlist[]` is an array of error messages that can be indexed using `errno`
- ▶ These are static strings that should not be changed
- ▶ The `strerror()` procedure returns the same error message with `errno` as its only parameter
- ▶ Note: you cannot use the return value from a system call here, it is usually a negative number which cannot be used as an array index

# Make

- ▶ You will need to use make for your assignments, you must include a makefile with your assignment that builds your solution
- ▶ It is a good idea to use make for the labs as well, this will make development easier and is good practice for when you need it for assignments
- ▶ Make is a build system that is driven by a text file, it tracks the timestamps on your files and attempts to do the minimum work required to build your solution
- ▶ There are versions of make for most platforms, so it is a good tool to learn

# Make

- ▶ There are several standard file names that make automatically searches for when asked to make a solution
- ▶ We will use Makefile for this file name, you can use any text editor that you like to construct this file
- ▶ The basic unit of a makefile is a rule that has the following format:

```
target ... : prerequisite ...
          recipe
          ...

```
- ▶ There can be multiple targets, but usually there is just one

# Make

- ▶ The prerequisites are the things that must exist before the target can be made
- ▶ The recipe is something can be executed to assist with making the target
- ▶ There can be multiple recipes, one per line, with each line starting with a tab
- ▶ Each line is run in a separate shell instance in the current directory, so there is little interaction between the lines

# Make

- ▶ A simple make file:

```
program : file1.o file2.o
```

```
        cc -o program file1.o file2.o
```

- ▶ The first rule in a makefile determines the target that will be made
- ▶ In this case it is program, which needs file1.o and file2.o
- ▶ The recipe shows how program is built
- ▶ But, how does make know how to produce file1.o and file2.o?
- ▶ It has a number of default rules that show how a .o file can be built from a .c or .cpp file

# Make

- ▶ We can add an include file defs.h to our example to give:

```
program : file1.o file2.o
```

```
cc -o program file1.o file2.o
```

```
file1.o : defs.h
```

```
file2.o : defs.h
```

- ▶ Now if defs.h changes make knows that it has to rebuild file1.o and file2.o
- ▶ This is enough to get you started, other features will be added in the labs

# Summary

- ▶ Covered two important general topics, memory allocation and system calls
- ▶ Memory allocation is a major source of bugs that can be very hard to find, best solution is to program carefully
- ▶ System calls are expensive, need to carefully plan their use
- ▶ Simple changes can drastically change program performance

# CSCI 3310

# Build Process

MARK GREEN  
FACULTY OF SCIENCE  
ONTARIO TECH

# Introduction

- ▶ Examine the process of going from source code to machine code
- ▶ How our program text gets converted into something that can be executed
- ▶ Use gcc and Linux as examples
- ▶ We start with the compiler, this is the program that does the initial step of converting our program
- ▶ The input is our program text, the output is assembly code, which is passed to the assembler

# Compiler

- ▶ Compilers are usually divided into two main parts; a front end and a back end
- ▶ The front end is language dependent:
  - ▶ It reads the program text
  - ▶ Converts it into a tree that represents the program
  - ▶ Traverses the tree to produce intermediate code
- ▶ Intermediate code is largely independent of the language being compiled, resembles a high level assembly language

# Compiler

- ▶ The back end is language independent, but processor dependent
- ▶ It performs optimizations on the intermediate code and then generates assembly code for the program
- ▶ There are standards for intermediate code
- ▶ All compilers can use the same back end to generate code for a particular processor
- ▶ All compiles can use the same front end for compiling the same language
- ▶ This avoids combinatorial explosion when we are supporting multiple languages on multiple processors

# Assembler

- ▶ The assembler takes the compiler output and converts it into binary machine instructions
- ▶ Why not generate machine code directly from the compiler?
- ▶ This is possible and has been done, but it makes the compiler more complicated
- ▶ In most cases we need to write an assembler, so why duplicate most of the effort in the compiler?
- ▶ Pass on some of the machine level details to the assembler, cleaner code in the compiler

# Loader

- ▶ Typically a program will consist of multiple files, which can be processed separately by the compiler
- ▶ Most programs use library functions
- ▶ The loader combines the machine code from multiple files, and adds in the library code
- ▶ Compiler assumes that each file will start at location 0, loader needs to relocate code to where it will actually be in the program
- ▶ Also needs to relocate library code and link it to the program
- ▶ The result is an actual binary file that can be executed

# Loader

- ▶ It's a bit more complicated
- ▶ There are two types of machine language files; hosted and free standing
- ▶ A hosted file runs under an operating system, this is the most common
- ▶ The loader provides small pieces of code that interface the binary with the OS
- ▶ A free standing file can execute on its own, this is how operating system kernels are produced

# Libraries

- ▶ There are two types of libraries; static and dynamic
- ▶ In the case of static libraries the machine code from the library is added to the output produced by the loader
- ▶ With a dynamic library the machine code isn't added until the program runs, at load time only stubs for the procedures are added to the program
- ▶ This can be done efficiently with modern operating systems, we will see how this is done later in the course
- ▶ Note, this is more complicated

# Libraries

- ▶ Why do we use dynamic libraries?
- ▶ Later we will see that they save main memory space
- ▶ In addition, they make software maintenance easier
- ▶ With static libraries, any change to the library requires recompiling all the programs that use it
- ▶ With dynamic libraries this isn't necessary as long as the public interface doesn't change
- ▶ Internal changes require rebuilding the library, but the programs that use it don't need to change

# Example

- ▶ We will use a simple program to illustrate the process

```
*****
/*
 * A very simple C program that illustrates the build process
 */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    int i;

    i = 2+2;
    printf("i=%d\n",i);
    exit(0);
}
```

# Example

- ▶ This program doesn't do very much
- ▶ We can run  
    gcc build.c
- ▶ And we will get an a.out file that can be executed
- ▶ If we add the -v option to gcc, it will print the programs that it executes to produce the a.out file
- ▶ An edited version of this is on the next slide
- ▶ gcc is really a driver program and not the compiler itself

# Example

```
mark@MSI: ~/examples
Using built-in specs.
COLLECT_GCC=cc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/5/lto-wrapper
Target: x86_64-linux-gnu
Thread model: posix
gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.12)
COLLECT_GCC_OPTIONS='-v' '-mtune=generic' '-march=x86-64'
 /usr/lib/gcc/x86_64-linux-gnu/5/cc1 quiet -v -imultilibarch x86_64-linux-gnu build.c -quiet -dumpbase build.c -march=generic -march=x86-64 -auxbase build -version -fstack-protector-strong -Wformat -Wformat-security -o /tmp/cc70gLJN.s
GNU C11 (Ubuntu 5.4.0-6ubuntu1~16.04.12) version 5.4.0 20160609 (x86_64-linux-gnu)
 compiled by GNU C version 5.4.0 20160609, GMP version 6.1.0, MPFR version 3.1.4, MPC version 1.0.3
GCC heuristics: --param ggc-min-expand=100 --param ggc-min-heapspace=131072
GNU C11 (Ubuntu 5.4.0-6ubuntu1~16.04.12) version 5.4.0 20160609 (x86_64-linux-gnu)
 compiled by GNU C version 5.4.0 20160609, GMP version 6.1.0, MPFR version 3.1.4, MPC version 1.0.3
GCC heuristics: --param ggc-min-expand=100 --param ggc-min-heapspace=131072
Compiler executable checksum: 8087146d2ee737d238113fb57fabb1f2
COLLECT_GCC_OPTIONS='-v' '-mtune=generic' '-march=x86-64'
as -v --64 -o /tmp/cc1Ls9i9.o /tmp/cc70gLJN.s
GNU assembler version 2.26.1 (x86_64-linux-gnu) using BFD version (GNU Binutils for Ubuntu) 2.26.1
COLLECT_GCC_OPTIONS='-v' '-mtune=generic' '-march=x86-64'
 /usr/lib/gcc/x86_64-linux-gnu/5/collect2 -plugin /usr/lib/gcc/x86_64-linux-gnu/5/liblto_plugin.so -plugin-opt=/usr/lib/gcc/x86_64-linux-gnu/5/lto-wrapper -plugin-opt=-fresolution=/tmp/cc70gLJN.s -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s -plugin-opt=-pass-through=-lc -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s -sysroot=/ --build-id --eh-frame-hdr -m elf_x86_64 --hash-style=gnu --as-needed -dynamic-linker /lib64/ld-linux-x86-64.so.2 -z relro /usr/lib/gcc/x86_64-linux-gnu/5/.../.../x86_64-linux-gnu/crt1.o /usr/lib/gcc/x86_64-linux-gnu/5/.../.../x86_64-linux-gnu/crti.o /usr/lib/gcc/x86_64-linux-gnu/5/crtbegin.o -L/usr/lib/gcc/x86_64-linux-gnu/5 -L/usr/lib/gcc/x86_64-linux-gnu/5/.../.../x86_64-linux-gnu -L/usr/lib/gcc/x86_64-linux-gnu/5/.../.../lib -L/lib/x86_64-linux-gnu -L/lib/../lib -L/usr/lib/x86_64-linux-gnu -L/usr/lib/.../lib -L/usr/lib/gcc/x86_64-linux-gnu/5/.../.../ /tmp/cc1Ls9i9.o -lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed /usr/lib/gcc/x86_64-linux-gnu/5/crtend.o /usr/lib/gcc/x86_64-linux-gnu/5/.../.../x86_64-linux-gnu/crtn.o
```

Compile

Assembly

Load

# Example

- ▶ If we use the `-S` flag to `gcc` it will stop after producing the assembly version of our program
- ▶ This will be stored in a `.s` file
- ▶ The assembly output from our simple program is shown on the next slide

# Example

```
mark@MSI: ~/examples$ more build.s
    .file    "build.c"
    .section      .rodata
.LC0:
    .string   "i=%d\n"
    .text
    .globl   main
    .type    main, @function
main:
.LFB2:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $32, %rsp
    movl    %edi, -20(%rbp)
    movq    %rsi, -32(%rbp)
    movl    $4, -4(%rbp)
    movl    -4(%rbp), %eax
    movl    %eax, %esi
    movl    $.LC0, %edi
    movl    $0, %eax
    call    printf
    movl    $0, %edi
    call    exit
    .cfi_endproc
.LFE2:
    .size   main, .-main
    .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.12) 5.4.0 20160609"
    .section      .note.GNU-stack,"",@progbits
mark@MSI:~/examples$
```

# gcc

- ▶ gcc supports multiple languages, the language is usually determined by the file suffix
- ▶ There is also g++, which is essentially the same as gcc, but uses different libraries to support C++
- ▶ There are different versions of the C standard, it is updated on a regular basis
- ▶ Unfortunately, the default version seems to change from one version of the compiler to the next
- ▶ This was a problem last year when we went to test the assignment one programs

# gcc

- ▶ The version of C is controlled by the `-std` (two -) compile option
- ▶ I recommend that you set this explicitly in your make files
- ▶ Good options to use are `c99` and `c11`
- ▶ For other options you can consult the gcc manual
- ▶ The manual is quite long, but it does have a good table of contents

# Cross Compiling

- ▶ Since we have separate front and back ends, we can use the back end for a different processor
- ▶ This is called cross compiling, producing code for a different processor than the one we are running on
- ▶ This is how we get operating systems, etc. for new processors
- ▶ It can also be used to produce code for processors that are too small to host a compiler on their own

# Summary

- ▶ Examine the process of going from source language text to machine code that can be executed
- ▶ Division of compilers into front and back ends
- ▶ Some of the options that can be used with gcc, and how it works
- ▶ The basic idea of cross compilation

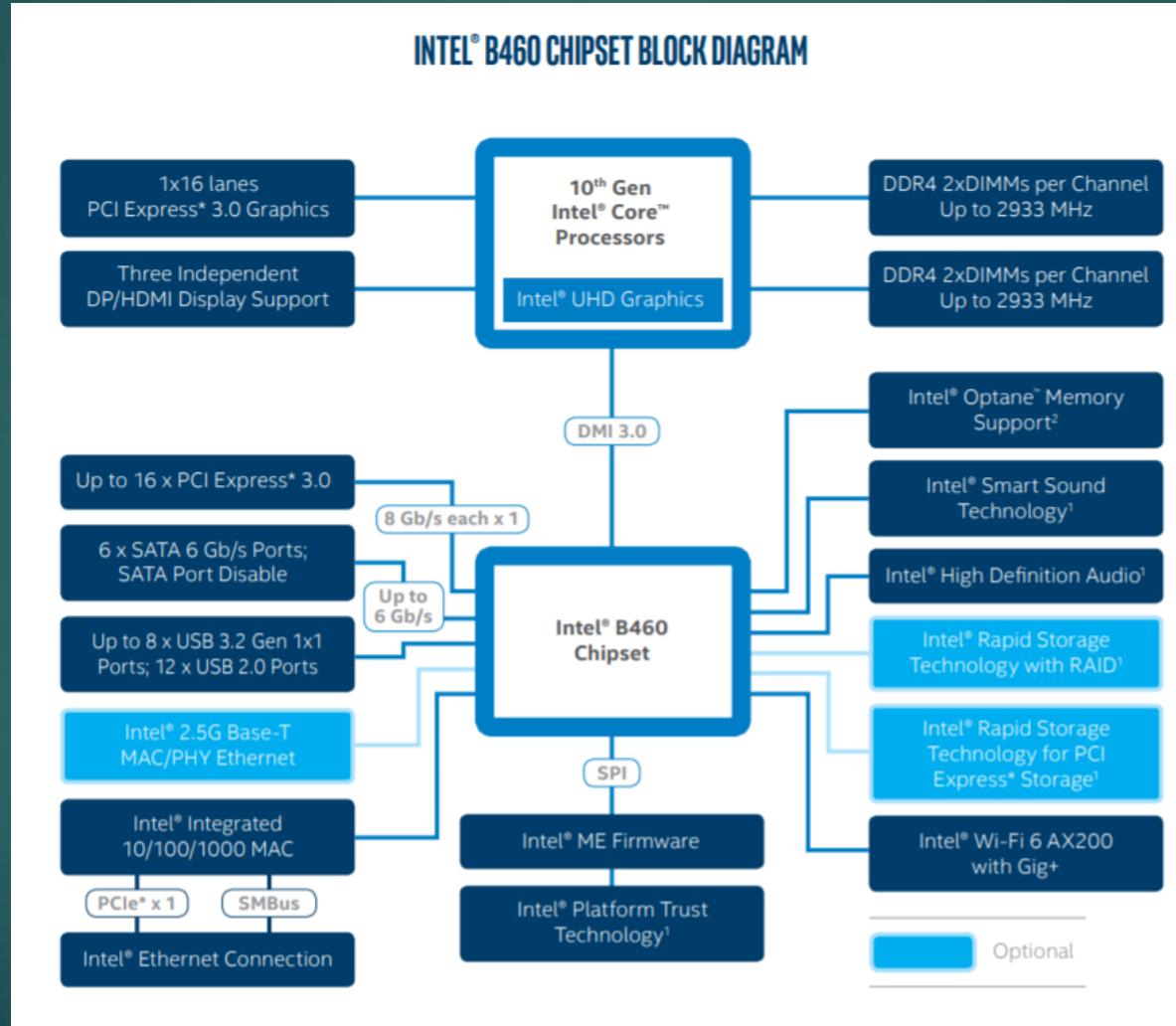
# CSCI 3310 Computer Architecture Review

MARK GREEN  
FACULTY OF SCIENCE  
ONTARIO TECH

# Introduction

- ▶ Operating systems deal with the computer hardware, they manage the hardware, the software layer just above the hardware
- ▶ At this point we will review computer architecture, in particular the parts of computer architecture that impact the OS
- ▶ At the introductory level we provide a simple model of a computer, processor, memory and I/O
- ▶ This is good enough for high level programming, but we need a more accurate model
- ▶ The next slide show a recent Intel chip set

# PC Architecture



# PC Architecture

- ▶ This is a slightly more complicated model 😊
- ▶ Note that there are multiple buses, there are two main reasons for this:
  - ▶ Different devices have different speeds, a USB keyboard is much slower than DRAM
  - ▶ Some devices need the complete bus bandwidth
- ▶ Over the years we have evolved from a single bus with all the memory and devices, to multiple buses that support much higher performance

# Processors

- ▶ There are two things of interest to us:
  - ▶ Instruction execution cycle
  - ▶ Memory access
- ▶ Start with the instruction execution cycle, the simple model of this is:
  - ▶ Fetch next instruction
  - ▶ Decode the instruction
  - ▶ Execute the instruction
- ▶ For most of the programming that we do, we view this as executing one instruction at a time, but that's not reality

# Processors

- ▶ Hardware designers want to get as much performance as possible, so they started processing all three steps in parallel
- ▶ Once we did three steps in parallel, why not do more??
- ▶ For example, if the instruction needs data from memory, that can be a separate pipeline step
- ▶ Modern processors can have up to 19 steps in their pipeline
- ▶ This means that 19 instructions are at some stage of execution at the same time

# Processors

- ▶ But life can get more complicated
- ▶ The circuits for floating point arithmetic, integer operations and Boolean operations are separate, so why not run all three at the same time?
- ▶ The processor can change the order of instruction execution to keep all three of these units busy
- ▶ So, we really don't know which instruction is being executed at any one point in time

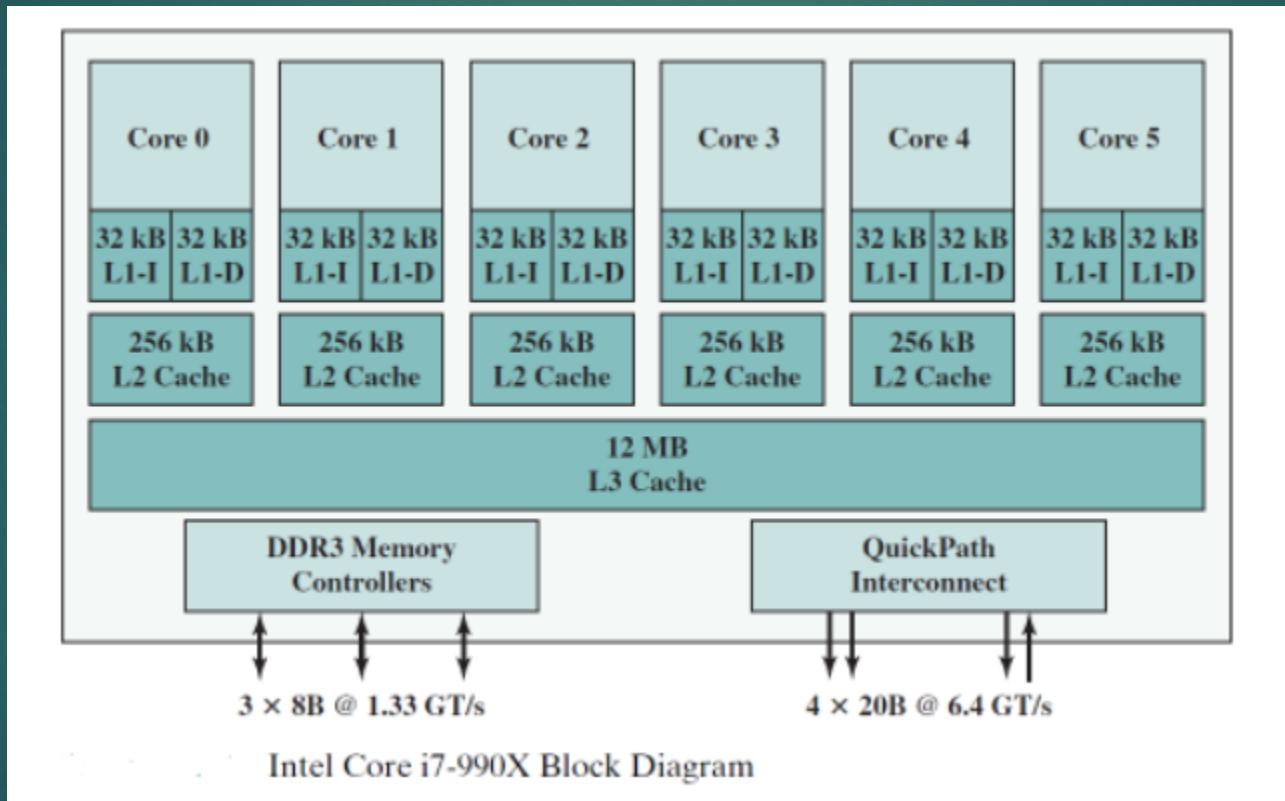
# Processors

- ▶ Why is this a problem?
- ▶ If the processor receives an interrupt, it transfers to the OS kernel, it processes the interrupt and then returns to the program that is running
- ▶ But, what instruction does it start at?
- ▶ We may need to drain the pipeline before we process the interrupt
- ▶ Similarly if an instruction causes a TRAP or requires some OS service, what happens to the other instructions in the pipeline
- ▶ Note: this could occur midway through the pipeline

# Processors

- ▶ In the good old days processors and memory worked at the same speed, and we could count on CPU speed doubling every three years
- ▶ Unfortunately, memory didn't keep up with processor speed, and CPUs are no longer doubling in speed
- ▶ Since we can't make our CPUs run faster, we put multiple CPUs on the same chip
- ▶ Now our OS needs to deal with this
- ▶ To deal with memory speed we introduce a memory hierarchy

# Processors



# Processors

- ▶ The CPU has a number of registers, clearly as fast as the CPU
- ▶ All operations occur on registers, so no slow down
- ▶ But, there is a small number of registers, 64 at max
- ▶ Hardware designers noticed that certain instruction sequences would repeat, for example a for or while loop
- ▶ Instead of storing these instructions in DRAM, they are stored in an on chip cache that runs at CPU speed
- ▶ The same can be done with data, called an L1 cache

# Processors

- ▶ L1 cache is fast, but it's expensive, so we add an L2 cache
- ▶ L2 cache is slower, twice as slow as L1 cache, but several times larger
- ▶ On top of this we add an L3 cache, again much larger, but slower
- ▶ Cache is organized as cache lines, typically 64 bytes of contiguous memory
- ▶ When the CPU needs data from memory, it first examines the caches in order L1, L2, then L3
- ▶ This is very fast, typically one or two clock cycles

# Processors

- ▶ If the data is available in a cache, the CPU retrieves it
- ▶ Otherwise, it must be loaded from DRAM
- ▶ The DRAM transfer is in units of cache lines, you can't transfer an individual byte
- ▶ When this happens a cache line must be removed and the new one stored in its place
- ▶ This works okay for reading instructions and static data, but what happens when we write data to memory?

# Processors

- ▶ If the memory location is in cache, the value in the cache is updated and the cache line is marked as dirty
- ▶ This means there is an inconsistency between what's stored in the cache and what's stored in DRAM
- ▶ At some point this cache line needs to be written back to DRAM, this will probably not occur immediately
- ▶ Now we have a problem, note that each core has its own L1 and L2 cache, what if two cores have the same cache line and one changes a value?
- ▶ We hope that the CPU designers handled this correctly

# I/O

- ▶ I/O occurs over one of the buses attached to a hub, which is then attached to the CPU
- ▶ The hub is a simplified processor that handles all the interactions with the buses for the CPU, doesn't need to worry about the different bus standards
- ▶ There are basically two types of I/O that we see on modern systems:
  - ▶ Interrupt driven
  - ▶ DMA – direct memory access
- ▶ Interrupts are usually used with slower devices

# I/O

- ▶ With interrupts the OS sends a command to the device, and then continues with its work
- ▶ When the device has finished the command it interrupts the CPU, which can then send another command
- ▶ When an interrupt occurs the process that is executing is suspended and control transfers to the OS
- ▶ At this point the OS must save all the CPU registers and any other CPU state before processing the interrupt
- ▶ When finished, the OS restored the registers and state and returns to the interrupted process

# I/O

- ▶ What happens if OS is processing an interrupt when another interrupt occurs?
- ▶ We could get into a situation where we never exit from interrupts
- ▶ During interrupt processing the CPU or OS turns off interrupts, so it can complete the interrupt without being interrupted
- ▶ When it's finished the interrupt, it turns interrupts back on
- ▶ The hub will hold any interrupts that occur while interrupts are off, and then forward them, in priority order, once interrupts are turned on again

# I/O

- ▶ Processing interrupts is expensive, so they are not suitable for fast devices
- ▶ Instead we use DMA where the device can directly access memory instead of going through the CPU
- ▶ When starting a DMA transfer the CPU gives the device the address in memory where the data is to be transferred to or from
- ▶ The device then carries out the transfer without involving the CPU
- ▶ When it's finished it can interrupt the CPU, or the OS can check a status word to see if the transfer is complete

# Booting

- ▶ The CPU has to start somehow, this can be a bit of a complicated process
- ▶ There will be an address that the CPU will start at, this will typically be an address in ROM, called the BIOS on PCs
- ▶ In the early days of PCs the BIOS was the operating system, it provided basic I/O routines for the standard devices, and handled starting up the CPU
- ▶ Its role as an operating system is long gone, but it still handles starting up the CPU

# Booting

- ▶ What does the BIOS need to do?
  - ▶ It initializes some of the CPU registers
  - ▶ It determines the amount of memory
  - ▶ It determines the devices that are connected to the CPU and performs basic initialization
  - ▶ It does a number of basic checks to determine if the system is okay
  - ▶ Finally it goes to the boot device and reads the first sector, this contains the initial boot and information on how the boot device is configured

# Summary

- ▶ Examined the basic architecture of a modern PC
- ▶ Examined how the CPU executes instructions and the problem that causes for the OS
- ▶ Examined how memory is accessed and different types of caches
- ▶ Examined how the CPU interacts with I/O devices
- ▶ Briefly summarized how the system boots

# CSCI 3310

# Introduction to

# Operating Systems

MARK GREEN

FACULTY OF SCIENCE

UOIT

# Introduction

- ▶ The software layer between the hardware and your application
- ▶ Two views:
  - ▶ Hardware abstraction- applications aren't concerned with the low level details of computer architecture, easier to move programs between computers
  - ▶ Resource sharing – multiple programs can run on the same hardware, don't interfere with each other, looks like they have the machine to themselves

# Hardware Abstraction

- ▶ Modern hardware is quite complicated, don't want to program at this level
- ▶ Like to use the simple model from first year:
  - ▶ Large uniform memory space
  - ▶ Simple CPU
  - ▶ Uniform interface to I/O devices
- ▶ This is not the way modern hardware works:
  - ▶ Multiple memory levels
  - ▶ Large variety of I/O devices
  - ▶ Complicated instructions sets

# Hardware Abstraction

- ▶ Linux runs on both Intel and ARM CPUs, we don't care about the difference, the details are hidden by the operating system
- ▶ Writing to the console and a disk file uses the same system call, we don't need to worry about the differences between these devices
- ▶ We don't care where our programs are located in memory, as long as we have enough memory
- ▶ This makes writing our applications so much easier

# Resource Sharing

- ▶ Modern computers run more than one program at a time, they share the CPU, memory, etc.
- ▶ This is done by quickly switching the CPU between programs
- ▶ When you are editing a program the CPU spends most of its time waiting for you to press a key, this time can be used by other programs
- ▶ The operating system coordinates this sharing of resources, we don't need to worry about it as programmers
- ▶ Can make the most of our computer system

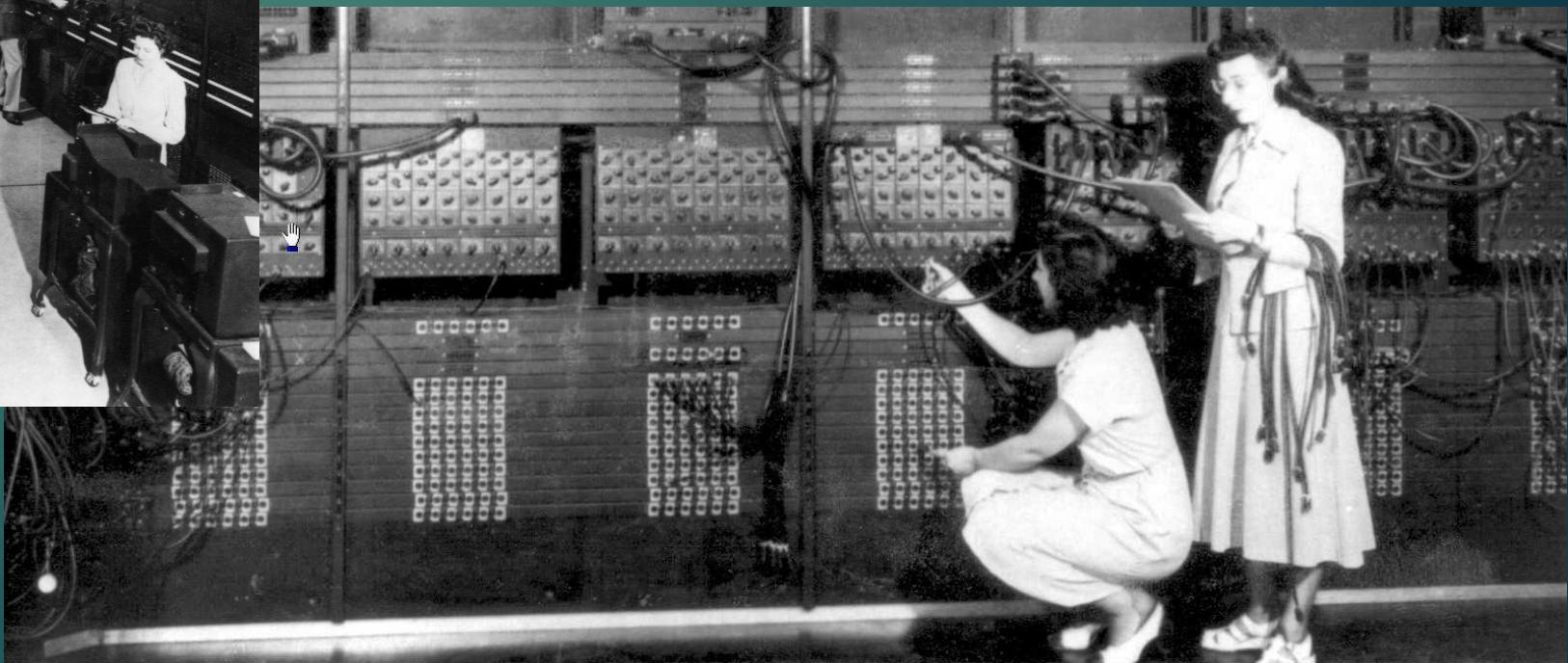
# Resource Sharing

- ▶ How do we want resource sharing to work?
  - ▶ It should be fair, every program gets its fair share of resources
  - ▶ The system should be responsive, it quickly responds to the user
  - ▶ Maximize the use of resources, we want to keep every part of the system as busy as possible
  - ▶ Reliable, programs must be protected from each other, one program can't cause the entire system to crash

# History

- ▶ Operating systems evolved over several decades, in some cases needed hardware assistance
- ▶ Early computers had no operating systems, programs ran on the bare metal
- ▶ Programmers would reserve a block of time:
  - ▶ Read in program from cards or paper tape
  - ▶ Run the program
  - ▶ Print the output
- ▶ Note: early I/O devices were quite slow

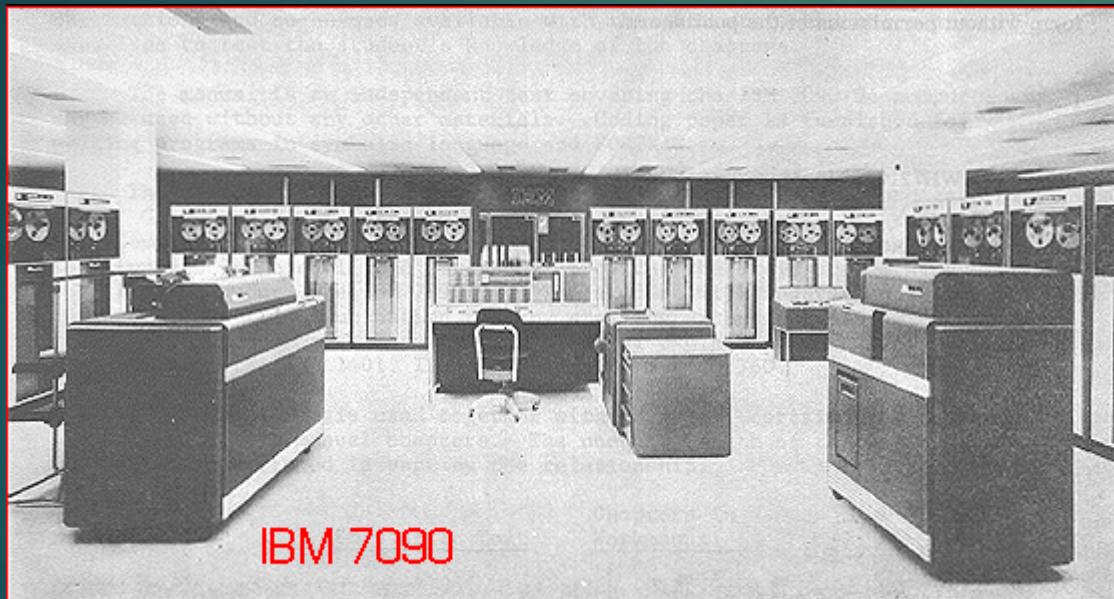
# ENIAC – Early Computer



# History

- ▶ Didn't make very good use of CPU, it was idle most of the time
- ▶ Early computers used many tubes, would only run for ~15 minutes before a tube blew and the hardware crashed
- ▶ When transistors were introduced computers would run much longer before they had hardware problems
- ▶ Computers were expensive, wanted to make the best use of them, don't want to waste time reading programs and printing results
- ▶ First operating systems – batch processing
- ▶ All the programs were collected together on a tape using a small low cost computer

# IBM 7090



# History

- ▶ The CPU would read the programs one at a time from the tape, relatively fast, and run the program
- ▶ The output from the program was written to another tape
- ▶ The output tape was then taken to the printer
- ▶ Much higher CPU utilization, overlapping input, compute and output
- ▶ JCL (Job Control Language) used to specify how each of the programs was to be run
  - ▶ Resources required
  - ▶ Input/output tapes and disks

# History

- ▶ What happens if the job before yours ran for a long time and did a lot of I/O?
- ▶ CPU time is wasted, smaller jobs could make use of the CPU, but they have to wait their turn
- ▶ Next innovation – multiple programs in memory, when one is waiting for I/O the CPU is given to another program
- ▶ Can keep the CPU busy all the time
- ▶ Divide memory into partitions, one program per partition
- ▶ Needed to compile the program for the partition it would use

# IBM 360



# History

- ▶ The IBM 360 (and several other computers) introduced hardware support for partitions
  - ▶ Base register – the physical memory address where the program is located
  - ▶ Extent register – the amount of physical memory allocated to the program
- ▶ Programs were compiled with a starting address of 0, when memory was referenced, address compared to extent register, if valid the base register was added to the address to get the real address
- ▶ Trivia – many computers through the 1980s had more physical memory than a program could address

# History

- ▶ With multiple programs in memory how do we protect the operating system?
- ▶ Don't want to give applications complete control
- ▶ Introduction of multiple execution modes:
  - ▶ Supervisor mode – has access to all of the hardware
  - ▶ User mode – has restricted access to hardware
- ▶ The operating system runs in supervisor mode, while all the other programs run in user mode

# History

- ▶ How do programs communicate with operating system, can't be a simple procedure call due to multiple modes
- ▶ A trap instruction is used to transfer control to the operating system, parameter is the operation to be performed
- ▶ Trap instruction sets supervisor mode, switches stack and registers, then transfers control to a known location
- ▶ The operating system can then process the trap, when finished it returns control to the user program
- ▶ This is secure, but much more expensive than a simple procedure call

# History

- ▶ In the 1980s memory became much cheaper, so address spaces started to expand
- ▶ Pre-1980 16 bit and 18 bit address spaces were common, had enough physical memory for the entire address space
- ▶ The 1980s saw 32 bit address spaces, a decade later 64 bit address spaces started to appear
- ▶ Now the address space was much larger than available physical memory
- ▶ Now what happens?? Program crashes when it accesses non-existent memory??

# History

- ▶ This lead to virtual memory
- ▶ Each program could have as much memory as it likes, but only part of the program will be in main memory, the rest will be on disk
- ▶ A table mapped program addresses into addresses in physical memory
- ▶ To be efficient this table was based on pages, where a page could be 4k of adjacent memory locations
- ▶ If a page wasn't in main memory it was read from disk, this usually meant that another page was removed from main memory

# History

- ▶ As memory got even larger, the page table got to be too large, remember they needed to be stored somewhere
- ▶ Many algorithms and schemes have been developed for virtual memory, we will look at some of them later
- ▶ Can have a major impact on program and system performance, something that application programmers need to be aware of

# History

- ▶ For decades we have relied on Moore's law, allowed us to double CPU performance every 3 years – not quite true but a good approximation
- ▶ This was based on the scaling of VLSI and clock speeds
- ▶ Unfortunately, physics stated that this can't happen forever, about a decade ago we started reaching the limit
- ▶ We can't use clock speed to make CPUs faster, for a short period of time cache was added to improve performance, but again we reach a limit

# History

- ▶ The only way forward was to use more processors
- ▶ We have to write our programs to make use of multiple processors
- ▶ The current trend is to put multiple cores on the same chip, this simplifies cache and memory management
- ▶ We will talk more about writing parallel programs later in the course
- ▶ Our first attempt at doing this was to have multiple CPU cards that shared one or more memory cards
- ▶ Since CPUs were faster than memory this didn't work particularly well
- ▶ Note: there needs to be a copy of the operating system on each CPU card

# History

- ▶ Next a CPU and memory were combined on the same card
- ▶ CPU had fast access to its memory, much slower access to memory on other cards
- ▶ Got to the point of putting multiple CPUs on each card
- ▶ Programs could use memory on multiple CPU cards, high speed communications over a common bus
- ▶ This worked very well up to about 8 CPUs, could scale to 256 CPUs, but failed when we got to 512 CPUs
- ▶ In this case the operating system is distributed over all the CPUs, needed to coordinate memory access

# History

- ▶ For large systems we now have discrete computers with high speed communications links
- ▶ This makes programming harder, but greatly simplifies the operating system, only responsible for its computer
- ▶ The hardware also scales well, can easily add more computers to the system
- ▶ It's now the programmers job to coordinate all of the computers

# Types of OS

- ▶ Desktop operating systems are the ones you are most familiar with, but there are other kinds
- ▶ Desktop is general purpose optimized for supporting a single user interface
- ▶ At the low end are embedded operating systems
- ▶ They are embedded in some device and have limited or no user interface
- ▶ Highly reliable, may not be able to reboot, usually can't be upgraded, have a limited set of applications

# Types of OS

- ▶ Closely related is real-time OS
- ▶ Support time critical applications, code that must be executed before a deadline
- ▶ Respond quickly to events and conditions
- ▶ Usually have a fixed set of applications, could be built right into the OS
- ▶ Example: autopilot for cars, power station control systems

# QNX

- ▶ Canadian developed operating system for the automotive market
- ▶ <https://www.youtube.com/watch?v=-9iggiTTbC8>
- ▶ <https://youtu.be/U1d1VX68GPM>

# Types of OS

- ▶ Server or batch operating systems, process a large number of transactions, user support isn't important
- ▶ Optimized for reliability, security and I/O
- ▶ May have a minimal load on the CPU and applications that are resident for a long time, static application mix
- ▶ Throughput is a major concern, must be able to deal with a large amount of disk traffic
- ▶ Usually found on large mainframe systems

# Types of OS

- ▶ At the high end is operating systems that support large clusters of processors
- ▶ The challenge is to make the best use of the resource:
  - ▶ Keep all of the CPUs busy
  - ▶ Minimize stalls for disk access
- ▶ There could be many jobs running on different parts of the cluster
- ▶ Need to schedule jobs based on the resources that they need

# OS Architecture

- ▶ Key design decision: how much of the OS runs in supervisor mode
- ▶ The part that runs in supervisor mode is often called the kernel
- ▶ All operating systems have some non-kernel code, they are often called utilities
- ▶ Modern desktop OSs have a monolithic kernel, most of the OS is in the kernel, whether it needs supervisor mode or not
- ▶ The OS is viewed as a single executable, loaded when the system boots
- ▶ This makes the OS easier to distribute, particularly automatically to a large number of sites

# OS Architecture

- ▶ But, there is a problem here
- ▶ Most of the code in the kernel doesn't need to run in supervisor mode, this could be as much as 90% of the code
- ▶ This causes two problems:
  - ▶ Security – the more code in the kernel the more avenues of attack
  - ▶ Updates – need a new kernel whenever a bug is found in non supervisor code
- ▶ Why use monolithic kernels?? It's easier!

# OS Architecture

- ▶ The opposite approach is called a micro kernel
- ▶ In this case only the part of the OS that needs supervisor mode is in the kernel, the rest of it is in user mode
- ▶ This produces a much smaller kernel, but it can involve multiple executables
- ▶ A monolithic OS is easier to architect since everything is in one executable and there is no mode switching, which can be more efficient
- ▶ But, a large amount of the code doesn't need supervisor mode, this leads to more serious bugs and security holes

# OS Architecture

- ▶ In a micro kernel there is less opportunity for these problems, since there is much less code in the kernel
- ▶ Usually less than 10% of the code in a monolithic kernel needs to be in supervisor mode
- ▶ A smaller kernel needs less memory, a good fit for embedded and real-time systems
- ▶ But, there needs to be an efficient way for the two parts of the OS to communicate
- ▶ This requires very careful design of the OS to avoid calls between parts or an efficient calling mechanism

# OS Architecture

- ▶ Another architecture is layered
- ▶ Different parts of the OS need access to different parts of the hardware
- ▶ Some parts can make use of lower level parts that have complete access, they may need little to no access
- ▶ Architecture based on a set of layers, with the lowest layer being the most secure, and higher levels having less access to the hardware
- ▶ Access based on need to know

# OS Architecture

- ▶ Bottom layer could implement processes, the next layer up implements memory access
- ▶ The third layer interacts with the I/O devices
- ▶ These are the layers that require the most security and the most access to the hardware
- ▶ Things like the file system can be built on top of these layers
- ▶ Again careful design is required to minimize communications between the layers
- ▶ Can result in more flexibility in system updates, ability to update on the fly for the higher layers

# Summary

- ▶ Examined the roles that operating systems play
- ▶ A brief history of operating systems, how they developed
- ▶ Overview of the different types of operating systems
- ▶ Discussed three basic operating system architectures



# CSCI 3310

# Input and Output

# Part One

MARK GREEN

ONTARIO TECH UNIVERSITY

# Introduction

- ▶ Linux views many things as a file, even things that aren't
- ▶ Provides a uniform interface to many types of resources
- ▶ Reading from a file and a network connection is done in the same way
- ▶ This reduces the number of things that we need to learn and produces more general code
- ▶ Start with the basics of IO programming and then add to it later in the course

# Files in Linux

- ▶ Start with an overview of how files are organized on disk and then look at how they are represented in the kernel
- ▶ Background for understanding the related system calls
- ▶ All physical devices have an entry in the /dev directory, internally a device is identified by a major and minor device number
- ▶ In the case of disks the major device number is the disk drive and the minor device number is the partition
- ▶ A disk is known as a block device, data is read and written one block at a time
- ▶ The other type of device is a character device, data is read and written one character at a time

# Files in Linux

- ▶ On disk a file is represented by a inode, an on disk data structure that describes the file
- ▶ The inode along with the major and minor device numbers uniquely identify a file
- ▶ An inode doesn't have a file name
- ▶ Filenames are stored in directories, basically a list of filename and the associated inodes
- ▶ Thus, an inode can have multiple filenames, they could be in the same or different directories
- ▶ Multiple programs can be accessing the same file under different names at the same time

# Files in Linux

- ▶ Linux has a simple file access control scheme that it inherited from Unix, this is inode based
- ▶ This is based on three levels of access and three types of users
- ▶ The access levels are:
  - ▶ read – the file can be read
  - ▶ write – the file can be written
  - ▶ execute – the file can be executed, or searched if it is a directory
- ▶ Each of these permissions is represented by a bit in a bit vector, so an inode can have zero permission for a particular type of user

# Files in Linux

- ▶ The three types of users are:
  - ▶ owner – the user that owns the inode
  - ▶ group – members of the group that owns the inode
  - ▶ world – anyone of the system
- ▶ There are three bits for each type of user, so we typically represent the permissions as a 3 digit octal number
- ▶ The higher order digit is owner, the next is group and the final is world
- ▶ The bits are ordered: read, write and execute from the left

# Files in Linux

- ▶ Example: 0754 allows the owner to do everything, the group members to read and execute and everyone else can just read
- ▶ Remember these permissions are at the inode level and not the filename level
- ▶ This is much more to the Linux file system than we will come back to later
- ▶ Each process has a file table in the kernel, one entry for each open file
- ▶ The index into this table is called a file descriptor, what our programs use to refer to files, a small non-negative integer

# Files in Linux

- ▶ Each process starts with three standard file descriptors:
  - ▶ 0 – standard input
  - ▶ 1 – standard output
  - ▶ 2 – standard error
- ▶ The process is free to close any of these files and open them as another file
- ▶ This is enough to get us started

# Open and Close

- ▶ Except for the three standard files, a file must be opened before it can be used
- ▶ The open system call is used for this:

```
int open(const char *filename, int flags, mode_t mode);
```

- ▶ The third parameter is optional
- ▶ The first parameter is the name of the file to be opened
- ▶ Open returns a small non-negative integer on success, if the result is negative an error has occurred and errno should be examined to determine the error

# Open and Close

- ▶ The flags parameter is a bit vector that indicates how the file should be opened
- ▶ It must include one of the following:
  - ▶ O\_RDONLY – open the file for reading only
  - ▶ O\_WRONLY – open the file for writing only
  - ▶ O\_RDWR – open the file for reading and writing
- ▶ There are many other values that can be or'ed with these, man 2 open contains a complete list of them
- ▶ We will look at a few that are useful

# Open and Close

- ▶ If O\_CREAT is used the file will be created if it doesn't already exist
- ▶ In this case the mode parameter gives the permissions for the new file
- ▶ By default writes will occur at the start of the file, if O\_APPEND is used the writes will start at the end of the file
- ▶ If O\_TRUNC is specified the file contents are discarded and the length is set to zero
- ▶ If O\_SYNC is used the data will be immediately written to disk instead of being buffered by the kernel

# Open and Close

- ▶ Once you are no longer using a file it should be closed:  
`int close(int fd);`
- ▶ The parameter is the file descriptor of the file to be closed
- ▶ Zero is returned if the close operation was successful
- ▶ Files are automatically closed when a program exits, but this is not the best style
- ▶ The file table is a limited size, for long running programs the file table could fill up if files aren't closed
- ▶ This can be a problem for servers

# Read and Write

- ▶ The read system call is used to read from a file descriptor:  
`ssize_t read(int fd, void *buf, size_t count);`
- ▶ The first parameter is the file descriptor to read from
- ▶ The second parameter is a pointer to a buffer that will receive this data
- ▶ You must allocate memory for this buffer
- ▶ The final parameter is the number of bytes to read, this must not be larger than the size of the buffer

# Read and Write

- ▶ If there are no errors, the value returned by read is the number of bytes that have been read
- ▶ This may not be the same as count
- ▶ If there is an error a negative value is returned, errno will identify the error
- ▶ There are several reasons why read will return fewer bytes than requested
- ▶ The most common one is when the end of file is reached
- ▶ In this case you will get a partial buffer, if there are no bytes left to read you will get zero

# Read and Write

- ▶ If the read operation is interrupted you will also get fewer bytes, more on this later
- ▶ There are several other cases where you could get zero or fewer bytes, this quite often happens in network communications, more on this later
- ▶ The write system call is used to write to a file descriptor:  
`ssize_t write(int fd, void *buffer, size_t count);`
- ▶ Buffer contains the data to be written, the third parameter is the number of bytes to write

# Read and Write

- ▶ `write()` returns the number of bytes written, or a negative number if an error occurs
- ▶ Again you may have fewer bytes written than requested
- ▶ This can occur if the `write()` is interrupted, or there is no more room on the device
- ▶ It can also occur if the kernel has run out of memory to buffer the data, particularly in network communications
- ▶ It is tempting to use large buffers, but you then need to pay careful attention to the return values

# File Positioning

- ▶ In the case of disks (and tapes) the file doesn't need to be read or written sequentially
- ▶ When a file is opened the file pointer is normally at the start of the file, if O\_APPEND is used when the file is opened it is positioned at the end of the file
- ▶ When a file is read or written the file pointer is moved according to the number of bytes read or written
- ▶ This is basic sequential file processing
- ▶ Linux also supports random file access as well using the lseek system call

# File Positioning

- ▶ The man page for lseek states:  
`int lseek(int fid, off_t offset, int whence);`
- ▶ The first parameter is the file descriptor and the second parameter is the number of bytes the file pointer should be move, which can be negative or positive
- ▶ The whence parameter gives the reference for the pointer movement:
  - ▶ SEEK\_SET – the offset is from the beginning of the file
  - ▶ SEEK\_CUR – the offset is from the current position
  - ▶ SEEK\_END – the offset is from the end of file

# File Positioning

- ▶ The return value is the new value of the file pointer, in bytes from the start of the file
- ▶ We can find the length of a file in the following way:  
`length = lseek(fd, 0, SEEK_END);`
- ▶ Random access can produce holes in a file, Linux doesn't allocate actual disk blocks for parts of the file that are not written
- ▶ Example, a file can have a length of 10,000 bytes, but only have 10 bytes of data
- ▶ A read of a block that hasn't been written returns zeros

# File Status

- ▶ While not strictly I/O, the stat family of system calls provides valuable information about files
- ▶ The declarations of these system calls are:

```
int stat(const char *filename, struct stat *buf);
```

```
int fstat(int fid, struct stat *buf);
```

```
Int lstat(const char *filename, struct stat *buf);
```

- ▶ The first two system calls return information about the file, the third one returns information about the symbolic link if filename is a symbolic link
- ▶ All three system calls return 0 on success

# File Status

- ▶ The following slide shows the declaration of the stat structure
- ▶ It contains some interesting information, most of which is obvious from the comments, or will be explained later when we discuss file systems
- ▶ The st\_mode field is the one that requires some explanation, it contains all the information related to file protection
- ▶ The lowest order 9 bits are the file access permissions
- ▶ There are also bits that identify the type of file, more on this later

# File Status

```
struct stat {
    dev_t      st_dev;          /* ID of device containing file */
    ino_t      st_ino;          /* inode number */
    mode_t     st_mode;         /* protection */
    nlink_t    st_nlink;        /* number of hard links */
    uid_t      st_uid;          /* user ID of owner */
    gid_t      st_gid;          /* group ID of owner */
    dev_t      st_rdev;         /* device ID (if special file) */
    off_t      st_size;         /* total size, in bytes */
    blksize_t  st_blksize;      /* blocksize for filesystem I/O */
    blkcnt_t   st_blocks;       /* number of 512B blocks allocated */

    /* Since Linux 2.6, the kernel supports nanosecond
       precision for the following timestamp fields.
       For the details before Linux 2.6, see NOTES. */

    struct timespec st_atim;    /* time of last access */
    struct timespec st_mtim;    /* time of last modification */
    struct timespec st_ctim;    /* time of last status change */

#define st_atime st_atim.tv_sec    /* Backward compatibility */
#define st_mtime st_mtim.tv_sec
#define st_ctime st_ctim.tv_sec
};
```

# Summary

- ▶ This gives us a start at doing I/O under Linux
- ▶ There are a lot of features that we haven't discussed that we will come back to later
- ▶ We will also need to talk about directory structure, but again that will come later

# CSCI 3310 Processes

MARK GREEN  
ONTARIO TECH

# Introduction

- ▶ A process is the basic unit of execution, so this is a good place to start
- ▶ A process is what brings a program to life
- ▶ A program is static, it is a description of a computation, while a process is the computation
- ▶ They are closely related, but they aren't the same thing
- ▶ There can be multiple processes executing the same program, there is one program, but many processes

# Processes

- ▶ What does a process need?
  - ▶ It needs a program to run
  - ▶ It needs memory for the process
  - ▶ It has a complete set of registers
  - ▶ It has a program counter and a stack pointer
  - ▶ It has open files, and file pointers for each file
  - ▶ A process ID
  - ▶ ...
- ▶ All this information is stored in a process table, one entry for each process

# Processes

- ▶ Where does a process come from?
- ▶ With Linux there is an initial process called init, it is created when the operating system is booted
- ▶ All other processes are created by init or its descendants
- ▶ How does a process create a new process?
- ▶ In Linux the fork() system call is used for this purpose
- ▶ After calling fork() there will be two almost identical processes, the parent process that called fork() and the child process that was created

# Processes

- ▶ The child starts with a copy of its parent's memory, and all of its open files
- ▶ The only difference between the two process is the return value from `fork()`
- ▶ In simple cases this is good enough, but most of the time we would like to run a different program in the child process
- ▶ The `execve()` system call is used for this, the first parameter to this function is the file where the program is stored

# Processes

- ▶ In Linux we have this two step creation process:
  - ▶ Create a new process
  - ▶ Load the program to run in that process
- ▶ In Windows this is done in a single step, the system call creates a new process and specifies the program to run in that process
- ▶ In Linux, a process and its descendants is called a process group
- ▶ Now that we've started a process how do we stop it?
- ▶ Process termination can be voluntary or involuntary

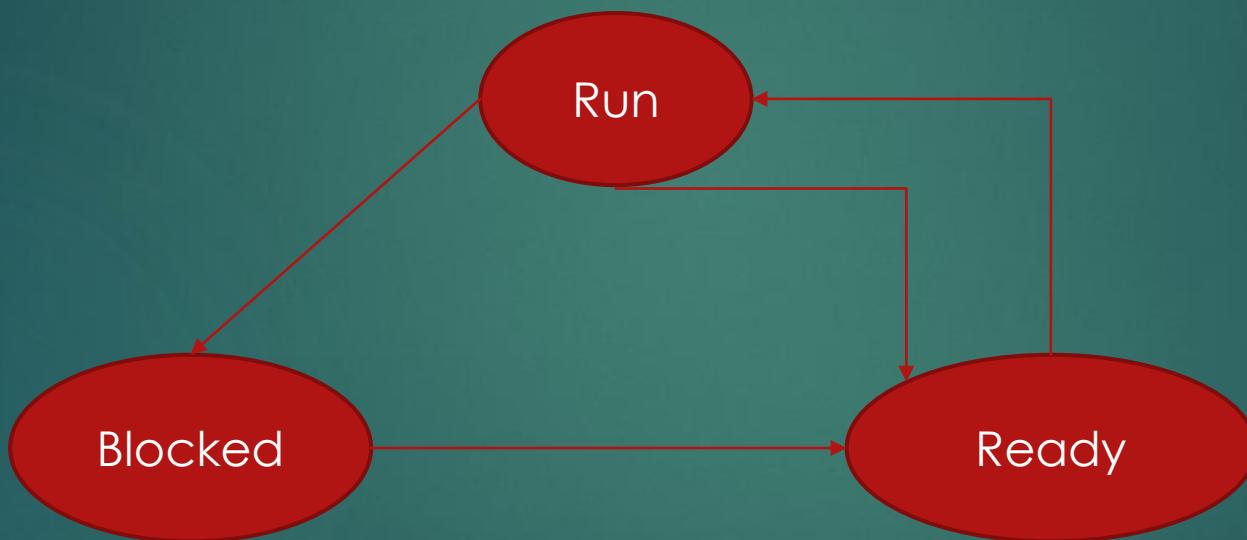
# Processes

- ▶ A process can terminate itself by calling the exit() system call
- ▶ This would normally happen when a process finishes its work, but can also happen when there is an error the process can't recover from
- ▶ If a process commits a fatal error that is detected by the operating system, it can be terminated, this is involuntary
- ▶ Also one process can terminate another process, for this to happen both processes must have the same owner

# Multiprogramming

- ▶ When a process calls a read() system call, it must wait for the data to be available, this can take a long time
- ▶ We don't want the CPU to be idle while the process waits for its data, we would like to run another process
- ▶ The process waiting for data is said to be in the **blocked** state
- ▶ Any process that is in memory and ready to run is said to be in the **ready** state
- ▶ The single process that is actually running is said to be in the **run** state – we are only concerned with a single processor now
- ▶ These states and their transitions are shown on the next slide

# Multiprogramming



# Multiprogramming

- ▶ Assuming that we have only a single CPU there can be only one process in the run state
- ▶ If this process performs an action that requires it to wait it goes into the blocked state
- ▶ When the process becomes unblocked it then goes into the ready state
- ▶ When the running process is blocked one of the ready processes is selected to run

# Multiprogramming

- ▶ The currently running process can also be interrupted by the OS, its time slice is over
- ▶ Typically a process will be given on the order of 10-100msec to execute
- ▶ When this happens the process goes into the ready state and one of the other ready processes is selected for the run state
- ▶ The part of the operating system that does all of this is called the **scheduler**, there are many algorithms for this, some we will investigate later

# Multiprogramming

- ▶ Switching a process between states, a context switch, is expensive:
  - ▶ Registers need to be saved and restored
  - ▶ The memory management system needs to be reset
  - ▶ The cache needs to be flushed
- ▶ We want a process to run for a reasonable length of time between context switches
- ▶ If there are several cores this becomes more complicated, but it's the same basic idea

# Threads

- ▶ Processes are very heavy weight:
  - ▶ They take time to create
  - ▶ Switching between processes is a lot of work
- ▶ If computations are completely independent then processes are good, for example text editor, web browser and compiler
- ▶ They don't share resources in any way, so a process for each of them is a good idea
- ▶ But, what happens if we have a computation that shares a lot of data, requires a lot of communications

# Threads

- ▶ Consider a web server, it could receive many requests per second and users want fast responses
- ▶ Reading a web page from disk takes time, so web servers cache common requests in main memory
- ▶ When a request comes in first check the cache, if a page isn't there read it from disk
- ▶ If this is purely sequential the web server can only serve a few pages per second
- ▶ Could wait 0.1 second for disk read, could serve at most 10 pages per second

# Threads

- ▶ What if the web server didn't have to wait? It could do something else while the file is read
- ▶ Have multiple threads of execution, like processes within a process
- ▶ Threads share all the resources of a process, memory, open files, etc.
- ▶ But they have their own register set, program counter and stack
- ▶ These are all registers, and there is a small number of them
- ▶ Switching between threads is very fast, avoid all the memory operations that make process switching expensive

# Threads

- ▶ Back to the example: when the web server gets a request it creates a new thread to process the request, it then waits for the next request to come in
- ▶ The worker thread first checks the cache, if the page is there it serves it immediately, otherwise it issues a read and blocks
- ▶ The web server can have many threads at the same time, when a request comes in the server immediately starts processing it
- ▶ New requests don't need to wait for slower requests to finish first

# Threads

- ▶ As another example consider a program that processes a large data file
- ▶ It reads the input file, processes the data, and writes an output file
- ▶ We know that the read and write will block, so a sequential program will waste a lot of CPU time
- ▶ Instead we could use three threads:
  - ▶ One for reading the input file
  - ▶ One for processing the data
  - ▶ One for writing the output file

# Threads

- ▶ The input thread reads data into a buffer
- ▶ The processing thread reads data from this buffer as soon as there is data available
- ▶ The processing thread saves the processed data in another buffer that the output thread then writes to disk
- ▶ Once the input thread has read enough data all three threads can be working at the same time
- ▶ Clearly only one thread is really executing at a time, but we can switch rapidly between them

# Threads

- ▶ With multiple cores we can have all three threads truly executing in parallel
- ▶ This is one way that we can take advantage of multi core processors
- ▶ Many operating systems provide a native threading capability, but there is a standard: POSIX Threads or Pthreads
- ▶ We will take a look at Pthreads later in detail and use them in the lab, but there is more that we need to learn about threads

# Threads

- ▶ What is the advantage of threads?
- ▶ The examples have a large amount of shared data that is stored in memory
- ▶ Since the threads share memory they all have access to it
- ▶ Communicating data between processes is expensive, so this saves resources
- ▶ Switching between threads is much cheaper than switching between processes
- ▶ This is due to memory management, so again we are saving resources

# Threads

- ▶ There are two ways that we can implement threads
  - ▶ Completely in user space
  - ▶ In kernel space
- ▶ User space threads have the advantage that they are operating system independent, the complete implementation can be handled in a user level library
- ▶ This is attractive from a portability point of view, it can be implemented once and used many places
- ▶ In this case the thread scheduler is in user space

# Threads

- ▶ But there are some problems with this approach
- ▶ Both of our examples involved blocking system calls
- ▶ If a thread calls one of these system calls it will block, the whole process will block and none of the other threads will execute
- ▶ This removes one of the major advantages of threads
- ▶ The thread library could provide its own versions of the system calls that first checks to see if the call would block before it is issued
- ▶ This is possible with some system calls, but not all of them

# Threads

- ▶ Another problem occurs when one of the threads hogs the processor
- ▶ There is no way to interrupt it and switch execution to another thread
- ▶ All the threads in a program must be cooperative and periodically release the processor
- ▶ This adds an extra burden to the programmer and can make the program harder to debug

# Threads

- ▶ The alternative is to implement threads in the kernel, this is where the thread scheduler is
- ▶ This solves the blocking problem, threads can use the standard system calls
- ▶ If a call blocks, the scheduler can pick another thread to execute, either in the same process or another process
- ▶ The kernel level scheduler already handles time slices, so a thread can't hog resources
- ▶ A kernel scheduler had a global view of resources, may be able to make better decisions

# Threads

- ▶ There is a problem here, all the thread API calls now become system calls, this will be much more expensive
- ▶ We can partially solve this problem by dividing the thread API into two parts:
  - ▶ The part that requires kernel support, thread creation, thread destruction and blocking system calls
  - ▶ All the other API calls
- ▶ This will reduce the amount of interaction with the kernel and improve performance

# Threads

- ▶ Kernel threads could also reduce portability, a tendency for each OS to go its own way
- ▶ This is where standards like Pthreads helps, implementation will tend to stick to the standard
- ▶ The tendency is towards kernel threads, introduce ways of reducing system call overhead

# Threads

- ▶ Two issues with threads regardless of the implementation
- ▶ With multiple threads there are multiple stacks, where do they go?
- ▶ With a single thread the stack grows down and the heap grows up, but we can't do this with more than one thread
- ▶ One solution is to allocate a block of memory each time a thread is created, used a the thread's stack
- ▶ If the block is too small the thread could crash on a procedure call when it runs out of stack space
- ▶ Making the block large wastes memory
- ▶ This is something the programmer needs to think about

# Threads

- ▶ Alternatively, the stack could be reallocated when it grows too large
- ▶ This can be quite expensive
- ▶ Another approach is to use a single stack and have all the threads share it
- ▶ This requires compiler support to change the way the stack is used so it is not a general solution
- ▶ If a thread encounters an error it could be hard to recover the stack, a fragile solution

# Threads

- ▶ The other key issue is libraries, particularly system libraries
- ▶ If there are multiple threads there could be multiple calls to the same procedure active at the same time
- ▶ Example: library procedure foo()
- ▶ Thread A calls foo(), while it is executing foo(), thread B also calls foo()
- ▶ We now have two threads executing foo() at the same time, what happens to memory?

# Threads

- ▶ There are two possibilities:
  - ▶ foo() only uses local variables
  - ▶ foo() manipulates a global or shared data structure
- ▶ In the first case we don't have any problems, both threads have their own stack, so the two copies of foo() will have their own local variables
- ▶ It's the second case that causes a problem, there is only one copy of the global data structure
- ▶ If both try to change it at the same time, it could end up in an inconsistent state

# Threads

- ▶ Procedures that can be called from multiple threads without causing problems are called **thread safe**
- ▶ The documentation for a procedure should tell you whether it is thread safe or not
- ▶ The second case brings up the problem of interprocess communications and synchronization
- ▶ This is a big topic, which is next on our agenda

# Interprocess Communications

- ▶ Even with a single core we have the problem of interprocess communications
- ▶ One process could be interrupted while manipulating a shared resource, and the next process could also manipulate the same resource
- ▶ With multiple cores this problem becomes worse, since the two process could be executing at the same time
- ▶ Add threads to the mixture with shared global variables and the problem becomes even worse

# Interprocess Communications

- ▶ This has been studied for a long time and there are many possible solutions
- ▶ Some of this is quite tricky, so you need to think carefully about it
- ▶ This is also where a lot of bugs can occur
- ▶ Let's start with a simple example: we have a global counter, count, and two threads that increment this count
- ▶ Both threads have the following statement:

```
count = count + 1
```

# Interprocess Communications

- ▶ This is one statement, but it will be compiled into three machine language instructions:
  - Load count
  - Add one
  - Store count
- ▶ With two threads there are many ways that these instructions could be interleaved
- ▶ If we are lucky, one thread will execute all three instructions followed by the next thread executing all three, no problem

# Interprocess Communications

- ▶ Unfortunately, there are many other ways that these three instructions could be executed
- ▶ The next slide shows one possibility
- ▶ The first thread loads count, adds one and then stores the new value of count
- ▶ But, the second thread loads the old value of count before the first thread can store the new value
- ▶ This thread works with the old value, adds one and stores the value
- ▶ In this case count is only incremented once and not twice

# Interprocess Communications

- ▶ Load count
- ▶ Add one
- ▶ Store count
- ▶ ...
- ▶ Load count
- ▶ Add one
- ▶ Store count

# Interprocess Communications

- ▶ With this short sequence of code what is the chance of this occurring?
- ▶ In practice the computation could be much longer, instead of one instruction there could be thousands, a much higher probability
- ▶ There could even be a system call somewhere in the sequence
- ▶ I've just kept the example simple so you can see the principle
- ▶ Problems like this are called **race conditions**, two or more processes are updating the a global resource at the same time

# Interprocess Communications

- ▶ There are many examples of race conditions that have been developed over the years
- ▶ Some common examples including updating bank accounts, and transferring funds between bank accounts
- ▶ Now we will turn our attention to ways of solving this problem
- ▶ The basic idea is to prevent two or more processes from manipulating the same resource at the same time
- ▶ And, the solution mustn't introduce its own race condition

# Interprocess Communications

- ▶ The place where we are manipulating a common resource is called a **critical region**
- ▶ We only want one process at a time in a critical region, this is called **mutual exclusion**
- ▶ Any technique for this must be fair, a process cannot be blocked forever
- ▶ We would also like it to be efficient, we don't want to waste CPU time
- ▶ The key to this is an **atomic** operation, an operation that can't be interrupted, this is typically done in hardware

# Interprocess Communications

- ▶ An example atomic operation is the XCHG instruction on the x86
- ▶ This instruction swaps the values of two locations, in the process it freezes both location so no other processor can change them
- ▶ To see how this works we will introduce a variable called lock
- ▶ If lock is zero we can enter the critical region and if it is non-zero we must wait
- ▶ With this we can write two short assembly functions, enter\_region and leave\_region

# Interprocess Communications

enter\_region:

```
MOVE REGISTER, #1  
XCHG REGISTER, LOCK  
CMP REGISTER, #0  
JNE enter_region  
RET
```

leave\_region:

```
MOVE LOCK, #0  
RET
```

# Interprocess Communications

- ▶ These two procedures solve our problem, we can even use them in user space
- ▶ But, there is a problem
- ▶ If lock is 1 when we call enter\_region it will loop until lock becomes 0
- ▶ This is called a busy wait and it wastes CPU time
- ▶ If we have a single core, everything will stop while a process is in a busy wait
- ▶ We will see later how we can get around this problem

# Interprocess Communications

- ▶ One of the common concurrency examples is the producer-consumer problem
- ▶ The producer process produces items and stores them in a shared buffer
- ▶ The consumer process removes items from the shared buffer
- ▶ The buffer has a fixed size,  $N$ , when the buffer is full the producer must wait for the consumer
- ▶ Similarly if the buffer is empty, the consumer must wait for the producer
- ▶ Start with the basic code that doesn't work very well

# Interprocess Communications

## Producer

```
#define N 100
int count = 0;
void producer() {
    int item;
    while(TRUE) {
        item = produce_item()
        while(count == N) ;
        insert_item(item);
        count = count+1;
    }
}
```

## Consumer

```
void consumer() {
    int item;
    while(TRUE) {
        while(count == 0) ;
        item = remove_item();
        count = count-1;
        consume_item(item);
    }
}
```

# Interprocess Communications

- ▶ Both processes have a busy wait, which is not good
- ▶ In addition, both processes have a critical region where the buffer is manipulated and count is changed
- ▶ These critical regions are not protected
- ▶ We have two things going on here:
  - ▶ The critical region where the buffer is updated
  - ▶ The number of items in the buffer
- ▶ Both of these need to be handled correctly for concurrency

# Interprocess Communications

- ▶ One solution to both of these problems is **semaphores**
- ▶ Semaphores have two atomic operations, up and down, that operate on integer variables, the variable is called a semaphore
- ▶ The down operation first examines the current value of the semaphore:
  - ▶ If the value is greater than zero, the semaphore is decremented and down returns
  - ▶ If the value is zero, the process is blocked, it is put on a list of processes waiting for the semaphore and another process will run

# Interprocess Communications

- ▶ The up operation increments the value of the semaphore, if its value was zero one of the blocked processes is chosen at random to run
- ▶ Note, there is no busy waiting, there is always a process running
- ▶ Now let's rewrite the producer-consumer example:

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
```

# Interprocess Communications

## Producer

```
void producer() {  
    int item;  
    while(TRUE) {  
        item = produce_item();  
        down(&empty);  
        down(&mutex);  
        insert_item(item);  
        up(&mutex);  
        up(&full);  
    }  
}
```

## Consumer

```
void consumer() {  
    int item;  
    while(TRUE) {  
        down(&full);  
        down(&mutex);  
        item=remove_item();  
        up(&mutex);  
        up(&empty);  
        consume_item(item);  
    }  
}
```

# Interprocess Communications

- ▶ We have three semaphores, let's examine what they are doing
- ▶ The mutex semaphore is called a **binary semaphore**, it guards the critical region where the buffer is updated
- ▶ The other two semaphores keep track of the buffer state
- ▶ The empty semaphore is the number of empty slots in the buffer
- ▶ The producer decrements this semaphore and the consumer increments it
- ▶ The full semaphore is the number of full slots in the buffer, it is manipulated in the opposite way

# Interprocess Communications

- ▶ Binary semaphores are so common that we have a special name for them, **mutex**, and a simplified implementation
- ▶ A mutex is used to guard a critical region, so the procedures are called mutex\_lock and mutex\_unlock
- ▶ These procedure are placed around the critical region
- ▶ A possible implementation is shown on the next slide, this is suitable for user space implementation
- ▶ A more efficient implementation can be done in the kernel where a list of waiting processes can be maintained

# Interprocess Communications

Mutex\_lock:

```
MOVE REGISTER, #1  
XCHG REGISTER, MUTEX  
CMP REGISTER, #0  
JZE ok  
CALL thread_yield  
JMP mutex_lock
```

ok: RET

Mutex\_unlock:

```
MOVE MUTEX, #0  
RET
```

# Interprocess Communications

- ▶ The synchronization techniques that we have seen so far are low level and can be hard to use
- ▶ The advantage is they are fairly easy to implement and are widely available
- ▶ There are other approaches that are higher level, but they require programming language support
- ▶ A good example of this is monitors
- ▶ A **monitor** is similar to a class, it has data and procedures, and the data inside a monitor cannot be accessed by code outside of the monitor

# Interprocess Communications

- ▶ The only way to change the data inside a monitor is through its procedures
- ▶ They guard the data and make sure it is consistent
- ▶ The important feature of monitors is that only one procedure within the monitor can be active at a time
- ▶ If a thread calls a monitor procedure, no other thread can call a monitor procedure until the first thread leaves the monitor
- ▶ In other words, only one thread at a time can be in the monitor

# Interprocess Communications

- ▶ In the case of our producer-consumer example, a monitor can be used for the buffer
- ▶ The producer calls a procedure, `add_item`, in the monitor to add an item to the buffer, and the consumer calls a monitor procedure, `remove_item` to remove an item
- ▶ This protects the buffer from any race conditions
- ▶ But, what happens if the producer calls `add_item` and the buffer is full?
- ▶ It must wait for an empty slot in the buffer, but that won't happen until `remove_item` is called, but the consumer is now blocked from entering the monitor

# Interprocess Communications

- ▶ If a thread blocks in a monitor, no other thread can enter the monitor, we need some way of dealing with this
- ▶ A thread needs to be able to give up its control of the monitor until it can move forward again
- ▶ **Condition variables** are used for this, they allow a thread to wait for a condition, and then resume when the condition is true
- ▶ There are two operations that act on condition variables, wait and signal

# Interprocess Communications

- ▶ When a thread calls wait, it stops executing and is placed on a list of threads waiting on that condition
- ▶ It also gives up control of the monitor, so another thread can enter the monitor
- ▶ When signal is called on a condition variable, one of the threads waiting on that condition is removed from the list and is allowed to execute
- ▶ There is a problem here, after calling signal there could be two threads executing in the monitor, which we don't want

# Interprocess Communications

- ▶ There are several solutions to this problem
- ▶ One is to only allow signal as the last statement in a monitor procedure
- ▶ In this way the thread will immediately exit the monitor after calling signal
- ▶ Another approach is to only allow the thread switch to occur when the procedure calling signal returns
- ▶ But, what happens if the procedure blocks before it returns?
- ▶ Need to make sure that a wait isn't called after a signal is called

# Interprocess Communications

- ▶ One of the problems with monitors is that the programming language must support them, this is not the case with C or C++
- ▶ This is a programming language feature and not an operating system feature
- ▶ You can build them in Java, but this requires some programmer effort
- ▶ If all the procedures in a Java class are synchronized then the resulting class is a monitor
- ▶ Monitors are mainly useful for threads, they are not very useful for synchronizing processes, since it's difficult to share the monitor between multiple processes

# Interprocess Communications

- ▶ The last technique we will examine is **barriers**
- ▶ this isn't a general technique and is only useful in certain situations
- ▶ Consider a computation on a large array, we can use multiple threads to compute different parts of the array
- ▶ Periodically we need to synchronize these computations, say when we go from one time step to the next
- ▶ A barrier causes all the threads using the barrier to wait until all the threads have reached the barrier
- ▶ That is, none of the threads can proceed until all of the threads have reached the barrier

# Deadlocks

- ▶ One of the difficult problems in concurrency is deadlocks
- ▶ This is when two or more processes are waiting for each other and none of them can make progress
- ▶ Consider the code on the following slide where the two threads both make use of two mutex
- ▶ Thread A starts executing and gets the lock on mutexA, at the same time thread B gets the lock on mutexB
- ▶ At this point neither thread can progress, since they are both waiting on each other

# Deadlocks

Thread A

```
down(&mutexA);  
down(&mutexB)
```

...

some computation

...

```
up(&mutexB);  
Up(&mutexA);
```

Thread B

```
down(&mutexB);  
down(&mutexA)
```

...

some computation

...

```
up(&mutexA);  
up(&mutexB);
```

# Deadlocks

- ▶ The solution in this case is quite simple, we just need to swap the calls to down in one of the threads
- ▶ If we do this for thread B, both threads will try to acquire the lock on mutexA at the start
- ▶ Only one of them will get the lock and can then acquire the lock on mutexB
- ▶ Once this thread has completed, it will release both locks and the other thread can enter the critical region

# Deadlocks

- ▶ In general, it is impossible to determine if a program has a deadlock, this is a theoretical result
- ▶ Deadlocks can involve multiple threads, not just the two in this example
- ▶ They usually are not in adjacent pieces of code, so they are not easy to see
- ▶ This best plan is careful design and programming to make sure that deadlocks won't occur
- ▶ There are some tools that can assist with finding deadlocks, but they can't find all of them

# Dining Philosophers

- ▶ Classical concurrency problem, suggested by Dijkstra in 1965
- ▶ We have 5 philosophers, 5 bowls of spaghetti and 5 forks
- ▶ Philosophers alternate between thinking and eating, and they need two forks in order to eat



# Dining Philosophers

- ▶ A philosopher can use the fork on their left and on their right
- ▶ Clearly there are not enough forks for all the philosophers to eat at the same time
- ▶ We want none of the philosophers to go hungry, so each philosopher must get a chance to eat at some point in time
- ▶ This is a resource management problem, with forks being the scarce resource
- ▶ We will model each philosopher by a thread
- ▶ An incorrect procedure for each philosopher is shown on the next slide, N is the number of philosophers

# Dining Philosophers

```
void philosopher(int i) {  
    while(TRUE) {  
        think();  
        take_fork(i);          // left fork  
        take_fork((i+1) % N);  // right fork  
        eat();  
        put_fork(i);  
        put_fork((i+1) % N);  
    }  
}
```

# Dining Philosophers

- ▶ Clearly this doesn't work, since two philosophers could be holding the same fork at the same time
- ▶ Another incorrect solution is based on the following idea
- ▶ The philosopher picks up their left fork and then checks to see if the right fork is free:
  - ▶ If it is free the philosopher picks it up and eats
  - ▶ If it is not free the philosopher puts down the left fork and waits for a period of time before repeating the process

# Dining Philosophers

- ▶ The solution appears to work, and it will for some time, but there is a problem
- ▶ What happens if all the philosophers pick up their left fork at the same time?
- ▶ There will be no right fork to pick up, so they will all put down their left work and wait
- ▶ After waiting they will all pick up their left fork again ...
- ▶ None of the philosophers will get to eat
- ▶ This is called **starvation**, the threads can run, but they don't make any progress, similar to deadlock, but not quite the same

# Dining Philosophers

- ▶ Why not have each philosopher wait a random length of time?
- ▶ This will work for a long period of time, but there is always the chance of starvation, it is not guaranteed
- ▶ This is okay in some situations, but not in others
- ▶ We would like a solution that is guaranteed to work
- ▶ Our first attempt at a guaranteed solution is shown on the next slide
- ▶ We introduce a binary semaphore, mutex, and put a critical region around the forks

# Dining Philosophers

```
void philosopher(int i) {  
    while(TRUE) {  
        think();  
        down(&mutex);  
        take_fork(i);          // left fork  
        take_fork((i+1) % N); // right fork  
        eat();  
        put_fork(i);  
        put_fork((i+1) % N);  
        up(&mutex);  
    }  
}
```

# Dining Philosophers

- ▶ This works, but there is a problem
- ▶ Only one philosopher can eat at a time, we've converted eating into a serial process, there is no concurrency
- ▶ With 5 philosophers there are enough forks for two philosophers to eat at the same time
- ▶ We are wasting resources
- ▶ There is a solution that maximizes concurrency and avoid deadlocks and similar problems

# Dining Philosophers

- ▶ Add a semaphore per philosopher, used to wait for free forks
- ▶ We also add a state for each philosopher:
  - ▶ THINKING
  - ▶ HUNGRY – waiting for the forks
  - ▶ EATING – has both forks and eating
- ▶ The first part of the solution is shown on the next two slides
- ▶ Again there is one philosopher() procedure for each thread

# Dining Philosophers

```
#define N 5
#define LEFT (i+N-1) % N
#define RIGHT (i+1) %N
#define THINKING 0
#define HUNGRY 1
#define EATING 2
typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];
```

# Dining Philosophers

```
void philosopher(int i) {  
    while(TRUE) {  
        think();  
        take_forks(i);  
        eat();  
        put_forks(i);  
    }  
}
```

# Dining Philosophers

- ▶ We have three procedures that are shared by all of the threads
- ▶ This is where we put the critical regions
- ▶ The semaphore for each philosopher,  $s[i]$ , is used to wait until both forks are available and the philosopher can eat
- ▶ The test procedure is used to test for this condition
- ▶ The semaphore mutex is used to check that only one philosopher at a time is updating their state and calling test()
- ▶ The down(& $s[i]$ ) at the end of take\_forks() waits until both forks are free

# Dining Philosophers

```
void take_forks(int i) {  
    down(&mutex);  
    state[i] = HUNGRY;  
    test(i);  
    up(&mutex);  
    down(&s[i]);  
}  
  
void put_forks(int i) {  
    down(&mutex);  
    state[i] = THINKING;  
    test(LEFT);      // see if the left neighbour can now eat  
    test(RIGHT);     // see if the right neighbour can now eat  
    up(&mutex);  
}
```

# Dining Philosophers

```
void test(int i) {  
    if(state[i]==HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {  
        state[i] = EATING  
        up(&s[i]);  
    }  
}
```

- ▶ if both forks are free we signal s[i] that the philosopher can now eat, which will cause that philosopher to exit the take\_forks() procedure

# Dining Philosophers

- ▶ Since this problem is so famous there are many different solutions using different concurrency techniques
- ▶ You can use a monitor to solve this problem as well, the forks are the resource that needs to be protected

# Readers and Writers

- ▶ Another well known problem motivated by database
- ▶ Consider a large database that can be accessed by many processes
- ▶ Example: airline reservation system, many people will be looking for flights, but only a fraction of them will be booking flights
- ▶ The first group only need to read the database, they don't cause concurrency problems, several can be reading at the same time
- ▶ The second group will be modifying the database, this is where the concurrency problem occurs, only one should modify the database at a time, and there can be no readers

# Readers and Writers

- ▶ One solution to this problem is based on keeping a count of the number of active readers
- ▶ The writer can only modify the database if this count is zero
- ▶ This give us one semaphore that controls access to the count and another that blocks the writer until there is no readers
- ▶ One possible solution is given on the next two slides

# Readers and Writers

```
typedef int semaphore;  
semaphore mutex = 1;      // control access to count  
semaphore db = 1;         // control write access to database  
Int count = 0;  
  
void writer(void) {  
    while(TRUE) {  
        produce_data();  
        down(&db);  
        write_data();  
        up(&db);  
    }  
}
```

# Readers and Writers

```
void reader(void) {  
    while(TRUE) {  
        down(&mutex);  
        count = count+1;  
        if(count == 1) down(&db);      // first reader, block any writers  
        up(&mutex);  
        read_data();  
        down(&mutex);  
        count = count-1;  
        if(count == 0) up(&db);      // last reader, the writer can access db  
        up(&mutex);  
        use_data();  
    }  
}
```

# Readers and Writers

- ▶ This solution protects the database, but there is a problem
- ▶ The writer can't access the database while a reader is accessing it
- ▶ Start with one reader, it will block the writer
- ▶ Before the first reader exits, another reader comes along, so the writer remains blocked
- ▶ If the readers arrive fast enough there will always be at least one reader accessing the database and the writer will never have a chance
- ▶ This solution starves the writer

# Readers and Writers

- ▶ How do we solve this problem?
- ▶ We need to block new readers from accessing the database while a writer is waiting
- ▶ This can be done by adding another semaphore that blocks new readers while a writer is waiting
- ▶ This reduces the amount of concurrency, but ensures that writers don't starve
- ▶ There are several other solutions to this problem that give writers more priority

# Lessons

- ▶ Concurrency is very subtle, a solution that looks good may not be
- ▶ Bugs can be very hard to find, programs can run for a long time before the bug appears
- ▶ Very hard to test for the absence of bugs
- ▶ Fixing one problem can lead to another, for example starvation or deadlock
- ▶ Sometimes there is a trade off between performance and correctness, can we live with a rare bug in order to get more performance?

# Scheduling

- ▶ Determining which process to run next
- ▶ A basic part of any operating system
- ▶ Many things to consider:
  - ▶ When does the scheduler run, how often?
  - ▶ The types of processes, job mix
  - ▶ Scheduling policy, which processes have priority
  - ▶ Resource utilization
  - ▶ efficiency

# Scheduling

- ▶ A process performs a system call and traps to kernel
- ▶ At this point registers and state are saved, possibly change memory mapping
- ▶ If the system call is short, do we return to the calling process?
- ▶ If the system call will block, we need to run another process
- ▶ Switching processes, context switch, is expensive:
  - ▶ Change memory mapping
  - ▶ Flush cache
  - ▶ Potentially read new process into memory

# Scheduling

- ▶ Don't want to switch processes too frequently, eats into computing time
- ▶ Switching every 10msec is too short, 100msec is too long, usually somewhere in between
- ▶ What happens when an interrupt occurs?
- ▶ Process caused a disk read, the data is now available, should we run the blocked process?
- ▶ What about a clock interrupt?
- ▶ This is a good time for the scheduler to examine the running process and possibly switch

# Scheduling

- ▶ Different processes behave differently:
  - ▶ Computational processes just want to sit on the CPU, use as much CPU time as possible, very few system calls, **compute bound**
  - ▶ Other process do a lot of I/O, file read and write, many system calls, very little use of CPU, **I/O bound**
  - ▶ Other processes interact with the user, long periods of waiting, but must respond quickly when the user does something
- ▶ We could have a mix of these processes on the same computer, want to fairly schedule all of them

# Scheduling

- ▶ Consider a super computer
- ▶ Some processes run for very long time, days, and are mainly CPU bound, make very good use of CPU
- ▶ Also need to compile and test programs
- ▶ These processes run for a relatively short period of time, several seconds
- ▶ Don't want to wait days for a compile to finish, should have priority over long running processes
- ▶ At least be able to get a slice of the processor so they can complete in a reasonable time

# Scheduling

- ▶ This introduces scheduling policies, which processes get priority
- ▶ With multiple groups of users this can be quite controversial
- ▶ Management groups for super computers deal with this all the time, very heated discussions
- ▶ For a web server priority might be given to new requests that have just come in
- ▶ If they involve little work they can run quickly and get out of the system

# Scheduling

- ▶ As another example consider a batch system, this is a system with no interactive users that processes large jobs
- ▶ A large company will have payroll, inventory, sales reports, accounting, etc. that they need to run on a daily basis
- ▶ These jobs can take a long time to run, deal with large files or databases
- ▶ This introduces efficiency, we would like to see each job take the minimal amount of time, at the same time we want to make sure the CPU is fully utilized – **CPU utilization**
- ▶ The number of jobs that are completed in a unit of time is called **throughput**

# Scheduling

- ▶ We have a similar set of issues for real-time systems
- ▶ Computations have a deadline, they should be completed by the deadline - constraints
- ▶ There are two types of constraints:
  - ▶ Hard – these constraints must be met, usually for safety reasons, or continued performance of the system
  - ▶ Soft – every effort should be made, but if the constraint is missed there won't be a catastrophe
- ▶ In a car a hard constraint would be avoiding a collision, a soft constraint is the entertainment system

# Scheduling

- ▶ There are many scheduling algorithms, start by assuming we have a single core on a single processor
- ▶ Two main types of schedulers:
  - ▶ **Preemptive** – the OS can interrupt a process and give control to another process
  - ▶ **Non-preemptive** – the process only gives up control of the CPU on a system call, no way to force it to switch
- ▶ Non-preemptive scheduling has largely disappeared, now only seen on very small systems, for example appliances

# Scheduling

- ▶ **First come first served** is a non-preemptive scheduling algorithm
- ▶ The first process to arrive is given the CPU, all subsequent processes are placed on a queue
- ▶ When the current processes is blocked, it is removed from the CPU and put on the end of the queue, the process at the front of the queue is then given the CPU
- ▶ This is very easy to implement, but it does have the problem that a long running process could hog the CPU
- ▶ If the process has an infinite loop, everything comes to a stand still, the system may need to be rebooted

# Scheduling

- ▶ **Shortest process first** – the process with the least amount of execution time will run first
- ▶ In batch systems it's easy to determine the amount of time a process will take, done by a human
- ▶ This could be preemptive or non-preemptive
- ▶ A related approach is **shortest remaining time** – the process that will finish soonest is chosen
- ▶ These are easy to implement and they increase the throughput of a system at the expense of longer running processes

# Scheduling

- ▶ The previous approaches work well for batch systems but not interactive ones
- ▶ **Round robin scheduling** maintains a list of all the processes in the ready state
- ▶ It then goes through this list giving each process a slice of computing time, this time is called a **quantum**
- ▶ If the process blocks before it has used its quantum, the next process on the list executes
- ▶ When a process reaches its quantum, it is interrupted and the next process on the list gets the CPU

# Scheduling

- ▶ This is a fair system, each process gets an equal share of the CPU and they all get a chance to run
- ▶ If there are a large number of processes in the ready state there may be a long gap between when the processes run
- ▶ This is not good for interactive systems, it could take seconds for a process to respond to a user
- ▶ One solution to this problem is to have **multiple queues**
- ▶ Interactive process are placed in the high priority queue, and other processes are placed in the low priority queue

# Scheduling

- ▶ All the processes in the high priority queue are given a chance to execute
- ▶ When all these processes are blocked, the processes in the low priority queue are given a chance to run
- ▶ There can be many more than two priority levels
- ▶ An alternative is to pick one low priority process to execute once all the high priority ones have executed
- ▶ Problem: the low priority processes may never get a chance to execute as long as there are high priority processes

# Scheduling

- ▶ A variation on this is **priority based scheduling**
- ▶ When a process enters the system it is assigned a priority
- ▶ Interactive processes can be given a higher priority than background processes
- ▶ The process list is ordered based on priority, with the highest priority process first
- ▶ The process with the highest priority is executed, at the end of its quantum its priority is lowered
- ▶ If its priority is now lower than the next process, the next process gets to execute and the older process is placed on the list in priority order

# Scheduling

- ▶ Periodically all processes on the list are given a priority increase
- ▶ Thus, a process that initially had a low priority will be given a chance to execute
- ▶ This is fair in the sense that all process will eventually get a chance to execute
- ▶ There are many variations on this scheme depending upon how the priorities are manipulated
- ▶ This is the scheduling approach that is used in most desktop systems
- ▶ On Linux the nice and renice commands can be used to change a process's priority, there is also a setpriority system call

# Scheduling

- ▶ Scheduling algorithms for real-time systems can be more complicated since they must ensure that hard real-time constraints are honoured
- ▶ For closed real-time systems the scheduling can be done once before the system runs
- ▶ In this case there is no scheduler, the OS may still check for abnormal conditions, but this usually indicates a hardware error and the system may shut down with an error message

# Multicore

- ▶ With multicore things get more complicated, since we have true parallelism
- ▶ The next slide shows the implementation we had for mutex
- ▶ There are two issues with this code
- ▶ First, what if two core execute the XCHG instruction at the same time?
- ▶ We could end up with both cores having the lock, no more mutual exclusion
- ▶ This didn't happen before since there could only be one thread executing at a time

# Interprocess Communications

Mutex\_lock:

```
MOVE REGISTER, #1  
XCHG REGISTER, MUTEX  
CMP REGISTER, #0  
JZE ok  
CALL thread_yield  
JMP mutex_lock
```

ok: RET

Mutex\_unlock:

```
MOVE MUTEX, #0  
RET
```

# Multicore

- ▶ We could attempt to solve this problem in software, but it gets quite complicated
- ▶ With modern processors there is hardware assist
- ▶ At the start of the XCHG instruction the memory is locked, so only one core can access memory
- ▶ This first core to do this wins and gets the lock
- ▶ Second, the process running on the second core will yield the CPU to another thread or process
- ▶ This made sense in a single core, since the thread that held the lock must eventually execute to free the lock

# Multicore

- ▶ Consider the case where the process with the lock only needs to execute a few instructions and then frees the lock
- ▶ In general critical sections are fairly short, so the lock won't be held for long
- ▶ On the other hand switching processes takes a long time
- ▶ By the time the new process starts running the mutex is probably free, maybe a long time before
- ▶ This is a waste of CPU time

# Multicore

- ▶ Maybe a busy loop might be better
- ▶ The CPU won't be doing anything constructive, but it will be less time than a process switch
- ▶ This involves a lot of memory accesses, some processors have instructions that wait for a block of memory to be written, reducing power consumption during the wait
- ▶ With hyperthreading or similar technologies, the core keeps the information for two threads in registers so it can quickly switch between them
- ▶ In this case the busy wait isn't needed

# Multicore

- ▶ Scheduling also becomes more difficult
- ▶ Before we just needed to worry about which process runs next, now we need to worry about which core it runs on
- ▶ If we have independent single thread processes this is easy, the process runs on the next available core
- ▶ This is an easy addition to a scheduling algorithm, but may not the best
- ▶ Consider the case of multiple threads that are communicating, we would like these threads to all be executing at the same time

# Multicore

- ▶ With kernel threads, the scheduler knows the process that the threads belong to, it can try to schedule the process's threads at the same time
- ▶ If there are fewer threads than cores, the scheduler can block the process until the required number of cores become free, it can then run all the threads at the same time
- ▶ This could cause the process to block for a long period of time
- ▶ If there are more threads than cores, the scheduler can try to run as many threads as possible at the same time, but it doesn't block the process in this case

# Multicore

- ▶ There is one other issue
- ▶ If a thread is executing on one core and it is interrupted it may be scheduled on any other core in the system
- ▶ This may not be the best choice
- ▶ The thread will have filled the cache with its pages, the next thread may not use all of the cache space, so some of the first threads pages may still be in cache
- ▶ It would make sense to schedule the first thread on the original core to reduce the amount of cache traffic

# Multicore

- ▶ This is called **processor affinity** a thread or process wants to run on a particular core
- ▶ In a multi CPU system there may be several busses, one for each CPU, and there may be devices that are on one bus, but not the other
- ▶ In this case it better if the processes that use that device reside on the CPU that has direct access to it, this saves transferring data between CPUs
- ▶ In multi CPU systems its common for each CPU to have its own ethernet port, sometimes only one CPU interacts with the GPU

# Multicore

- ▶ We can still use the same basic scheduling algorithms, but they will need to be modified
- ▶ For example, we could have a ready queue per core, instead of one for the entire system, this would deal with processor affinity
- ▶ Similarly priority could be based on where the thread needs to run, or if multiple threads need to run at the same time

# Summary

- ▶ Examined processes and threads
- ▶ Examined the common techniques used to synchronize processes and threads
- ▶ Examined some of the classical concurrency problems
- ▶ Examined some of the standard scheduling algorithms
- ▶ Examined the complications that multicore adds to all of this

# CSCI 3310

# Processes and

# Threads in Linux

MARK GREEN  
ONTARIO TECH

# Access Control

- ▶ File permissions were covered in the file section, but how do we find out what permissions a process should have?
- ▶ This is based on users, the user who is running the process and the user who created the program
- ▶ Each user has an integer ID, assigned when the user account is created
- ▶ There is a special user ID, 0, called the root user who has permission to access everything
- ▶ There used to be an actual root account, but on most systems it can no longer login, this was a major security hole

# Access Control

- ▶ When a user logs in, their user ID is determined, and all the processes they run normally use this ID
- ▶ This works well for most programs, but there are cases where a more sophisticated mechanism is required
- ▶ Consider the passwd program, used to change user passwords
- ▶ The passwords are stored in the /etc/passwd file, which can only be written by the root user
- ▶ So how can passwd update this file?
- ▶ Programs like passwd are setuid, when they run they take on the permissions of the file owner

# Access Control

- ▶ The passwd is a setuid program, and its owner is root
- ▶ When passwd runs, it runs as root and not the user that started the process
- ▶ There is a bit in the file permissions that determines whether a program is setuid
- ▶ These programs must be written with care since they are an attack vector
- ▶ The problem is more complicated than this, consider the ftp daemon, ftpd, that processes ftp requests

# Access Control

- ▶ This daemon starts by running as root, but when a user logs in it will switch to the user's account
- ▶ Now the process is running as a regular user and not root
- ▶ The problem is that ftpd sometimes needs to run as root again, but it can't do that because it's now a regular user process, it's lost that ability
- ▶ To solve this problem we introduce more user IDs for the process, when the process is running it uses the effective user ID
- ▶ It also has a real user ID, which in the case of ftp is root
- ▶ There is also a saved user ID

# Access Control

- ▶ A process can switch its effective user ID to either its real or effective user ID, but unless it's root it cannot change to any other user ID
- ▶ There are several system calls that can be used to change the user ID, the most flexible is setreuid():

```
int setreuid(uid_t ruid, uid_t euid);
```
- ▶ This system call sets the real and effective user IDs to its parameters
- ▶ If either parameter is -1, that user ID isn't changed
- ▶ Note, the process must have proper permissions to change the user IDs

# Fork

- ▶ The fork() system call is used to create a new process:  
`pid_t fork(void);`
- ▶ This system call produces a child process with a copy of the parent process's memory, it also has all the parent's open files
- ▶ In the parent process fork() returns the process ID of the child process, in the child it returns 0
- ▶ A process can determine its process ID by calling getpid() and it can determine its parent's process ID by calling getppid()

# Exec

- ▶ The exec family of procedures are used to start the execution of a new program
- ▶ This program replaces the currently running process
- ▶ If these procedures are successful they don't return
- ▶ The only one of these procedures that is a system call is execve:  
`int execve(const char *filename, char const argv[], char const envp[]);`
- ▶ The first parameter is the filename for the program that will be executed, this could be a binary executable or an interpreter script

# Exec

- ▶ If it's a binary executable it must have a main() procedure that is declared in the following way:

```
int main(int argc, char *argv[], char *envp[]);
```
- ▶ The argv parameter is the list of parameters to the program, and argc is the number of parameters
- ▶ By convention, the first parameter to a program is the name of the program, the file it is stored on
- ▶ Why? A single executable can be used for several different purposes, each one having a different filename, but all sharing the same inode

# Exec

- ▶ The argv array is terminated by a NULL pointer when it is passed to execve
- ▶ There are no constraints put on the format of the entries in the argv array, it is up to the program to interpret them
- ▶ The envp array is similar, it is an array of strings with the last entry being a NULL pointer
- ▶ This array contains the environment variables that are active when the program is started

# Environment Variables

- ▶ Both Linux and Windows have environment variables
- ▶ These are variables that are global to programs and are maintained by the shell
- ▶ In Linux the `printenv` program can be used to list all the environment variables and their values
- ▶ Each entry in `envp` is a string with a particular format:  
`name=value`
- ▶ The most important of these variables is `PATH`, a colon separated list of directories to search for programs

# Interpreter Scripts

- ▶ The filename passed to execve can be a text file with execute permission
- ▶ In this case the first line of the file must be:  
`#! Interpreter [optional arguments]`
- ▶ In this case interpreter must be a binary executable and it will be started with the optional arguments, the filename, and then the parameters in the argv array
- ▶ The most common use of this is to start a shell script, but it could also be used for a python program or the input to any other interpreter

# Exec

- ▶ execve ignores the PATH variable, it just looks for the file in the current directory
- ▶ There are a number of library functions that call execve, but behave slightly differently or provide a different interface to execve
- ▶ Three of them don't use the argv array, instead they provide the arguments as parameters:

```
int execl(const char *path, const char *arg, ... (char*) NULL);
```

```
int execlp(const char *file, const char *arg, ... (char*) NULL);
```

```
int execle(const char *path, const char *arg, ... (char*) NULL, char*  
const envp);
```

# Exec

- ▶ For these procedures the list of program arguments is terminated by a NULL pointer
- ▶ The three other procedures in this family are:

```
int execv(const char *path, char* const argv[]);
```

```
int execvp(const char *file, char*const argv[]);
```

```
int execvpe(const char *file, char* const argv, char* const argv[]);
```

- ▶ The execlp(), execvp() and execvpe(), search the PATH environment variable if the filename they are passed doesn't contain the / character
- ▶ There is quite a bit of flexibility here

# Wait

- ▶ There are cases where the parent process needs to know when the child process has finished
- ▶ There are two system calls that can be used for this purpose:

```
int wait(int *status);
```

```
int waitpid(pid_t pid, int *status, int options);
```

- ▶ The `wait()` system call suspends the process until one of its children has terminated.
- ▶ The status parameter points to an integer variable that provides information on the terminated process, this can be NULL

# Wait

- ▶ The waitpid() system call gives us more control
- ▶ If the value of pid is greater than 0, it waits for that particular process
- ▶ If the value is -1, it will wait for any child
- ▶ The man page gives details on how you can wait for any process in a process group
- ▶ There are several options that can be passed, the only one of interest now is WNOHANG
- ▶ If this option is used waitpid() will return immediately instead of waiting for a child to terminate

# Wait

- ▶ If status is used there are several macros that can be used to retrieve information, these macros take the integer and not the pointer to it
- ▶ WIFEXITED(status) returns true if the process terminated normally, by calling exit or returning from main
- ▶ WEXITSTATUS(status) returns the value passed to exit when the process terminated
- ▶ There are several other conditions that can be tested, see the man page for details

# Pthreads

- ▶ Pthreads is the standard thread package that is used by Linux
- ▶ There is a lot of basic information on pthreads that is available on the internet, one of the best sources is <https://computing.llnl.gov/tutorials/pthreads/>
- ▶ Some of the information on this page is particular to their lab, but it is a good source of general information
- ▶ We will review the basic pthreads procedures and look at a number of examples program that are available on Canvas
- ▶ The examples are in a tar file that contains the C source code plus a make file

# Pthreads

- ▶ A new thread is created by calling pthread\_create():

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,  
void* (*start_routine) (void*), void *arg);
```
- ▶ The first parameter is a pointer to a variable that will contain the ID for the thread that is created
- ▶ The second parameter is a pointer to attributes for the new thread
- ▶ The default attributes are okay for most threads so this parameter is usually NULL
- ▶ The third parameter is the procedure that is used to start the thread, this is where the thread starts

# Pthreads

- ▶ The thread procedure has a single parameter, which is a void\*
- ▶ The final parameter to pthread\_create() is the parameter to be passed to the thread procedure
- ▶ Thus, a thread procedure will have the following form:

```
void *proc(void *parameter) {  
    ...  
}
```

- ▶ The parameter will usually be cast to a variable inside the thread procedure

# Pthreads

- ▶ When a thread is finished it should call `pthread_exit();`  
`void pthread_exit(void *retval);`
- ▶ The parameter can be used to pass a value back to the procedure that created the thread
- ▶ This is useful in programs that do a lot of computation, but we won't need it, there is an example that shows how it can be used
- ▶ This is enough to get things started with an example, just create some threads and exit
- ▶ All programs that use pthreads must include `pthread.h` and use `-lpthread` to link with the pthreads library

# Pthreads

- ▶ The first part of our example program is:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 5

void *printHello(void *threadid) {
    long tid;
    tid = (long) threadid;
    printf("Hello World from thread #%-ld\n", tid);
    pthread_exit(NULL);
}
```

# Pthreads

- ▶ This is a simple thread procedure, its single parameter is a long integer that identifies the thread
- ▶ This parameter is then cast to the tid variable inside the thread procedure
- ▶ The main function for this example is shown on the next slide
- ▶ It has an array threads where the ID of the created threads are stored
- ▶ It is basically a simple loop that creates the threads
- ▶ An example output is shown on the following slide

# Pthreads

```
int main(int argc, char *argv[]) {  
    pthread_t threads[NUM_THREADS];  
    int rc;  
    long t;  
    for(t=0; t<NUM_THREADS; t++) {  
        printf("In main: creating thread %d\n", t);  
        rc = pthread_create(&threads[t], NULL, printHello, (void*) t);  
        if(rc) {  
            printf("Error in pthread_create: %d\n",rc);  
            exit(-1);  
        }  
    }  
    pthread_exit(NULL);  
}
```

# Pthreads

In main: creating thread 0

In main: creating thread 1

Hello World from thread #0

In main: creating thread 2

Hello World from thread #1

In main: creating thread 3

Hello World from thread #2

In main: creating thread 4

Hello World from thread #3

Hello World from thread #4

# Pthreads

- ▶ You need to be careful when passing data to a thread
- ▶ In this case we are passing an integer value, each thread will get a different value
- ▶ For simple values that are passed by value everything is okay
- ▶ But, if we pass a pointer to a data structure we need to be careful, we could end up with all of the threads sharing the same data structure
- ▶ We can illustrate this with our example program, instead of passing the value of  $t$ , we pass a pointer to  $t$

# Pthreads

In main: creating thread 0

In main: creating thread 1

Hello World from thread #1

In main: creating thread 2

Hello World from thread #2

In main: creating thread 3

Hello World from thread #3

In main: creating thread 4

Hello World from thread #4

Hello World from thread #5

# Pthreads

- ▶ We know that a thread can pass a value back through `pthread_exit`, but how do we retrieve that value in the thread that created it
- ▶ The `pthread_join` procedure is used to wait for a thread to terminate and retrieve its value:

```
int pthread_join(pthread_t thread, void **reval);
```

- ▶ The first parameter is the thread ID of the thread that the caller is waiting on
- ▶ The second parameter is the location where the return value will be stored

# Pthreads

- ▶ To see how this works make a simple change to our example to return the thread number:

```
void *printHello(void *threadid) {  
    long tid;  
    tid = (long) threadid;  
    printf("Hello World from thread #%"PRIld"\n", tid);  
    pthread_exit(threadid);  
}
```

# Pthreads

- ▶ To the main procedure we add a new variable status and then the following code at the end of the procedure:

```
for(t=0; t<NUM_THREADS; t++) {  
    rc = pthread_join(threads[t], &status);  
    if(rc) {  
        printf("Error in pthread_join: %d\n",rc);  
        exit(-1);  
    }  
    printf("Main: thread %ld completed with status %ld\n",  
          t, (long) status);  
}
```

# Pthreads

- ▶ When we run this program we get the following output:

In main: creating thread 0

In main: creating thread 1

Hello World from thread #0

In main: creating thread 2

Hello World from thread #1

In main: creating thread 3

Hello World from thread #2

In main: creating thread 4

Hello World from thread #4

Main: thread 0 completed with status 0

Main: thread 1 completed with status 1

Main: thread 2 completed with status 2

Hello World from thread #3

Main: thread 3 completed with status 3

Main: thread 4 completed with status 4

# Pthreads

- ▶ Critical regions can be protected in Pthreads using a mutex
- ▶ A mutex variable is declared in the following way:  
`pthread_mutex_t mutex;`
- ▶ Before a mutex can be used it must be initialized, there are two ways to do this
- ▶ One is when the variable is declared:  
`pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`
- ▶ This is the easiest way to do it if you have individual mutex variables

# Pthreads

- ▶ This approach doesn't work very well if you have an array of mutex variables
- ▶ In this case the `pthread_mutex_init()` procedure can be used:

```
int pthread_mutex_init(pthread_mutex_t *mutex, const  
pthread_mutexattr_t *attr);
```

- ▶ While we can specify attributes for the mutex, none of them are particularly interesting and `NULL` is typically used

# Pthreads

- ▶ The two main procedures that are used by mutex variables are:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);  
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```
- ▶ These two procedures do the obvious thing
- ▶ To illustrate how this work we will return to our readers and writers problem
- ▶ In this program we will have a single writer thread and several reader threads
- ▶ The number of reader threads is given by the READERS constant

# Pthreads

- ▶ There are three global variables in this program, the two mutex variables and the count on the number of readers in the database
- ▶ Their declarations are:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;  
pthread_mutex_t db = PTHREAD_MUTEX_INITIALIZER;  
int count = 0;
```

- ▶ The procedure for the writer thread is shown on the next slide, the reader procedure is on the following two slides

# Pthreads

```
void *writer(void *threadid) {  
    int i;  
    for(i=0; i<10; i++) {  
        sleep(1);      // prepare the data for writing  
        pthread_mutex_lock(&db);  
        printf("writer in database\n");  
        printf("write out of database\n");  
        pthread_mutex_unlock(&db);  
    }  
    pthread_exit(NULL);  
}
```

# Pthreads

```
void *reader(void *threadid) {  
    long tid;  
    int i;  
    tid = (long) threadid;  
    for(i=0; i<10; i++) {  
        pthread_mutex_lock(&mutex);  
        count = count+1;  
        if(count == 1) pthread_mutex_lock(&db);  
        pthread_mutex_unlock(&mutex);  
        printf("reader %ld in database, total readers %d\n",tid,count);  
        printf("reader %ld leaving database\n",tid);  
    }  
}
```

# Pthreads

```
pthread_mutex_lock(&mutex);
count = count -1;
if(count == 0) pthread_mutex_unlock(&db);
pthread_mutex_unlock(&mutex);
sleep(1);    // process the data
}
pthread_exit(NULL);
}
```

# Pthreads

- ▶ This looks very similar to the algorithm that we had earlier in class
- ▶ The main procedure shown on the next slide creates one writer thread and then enters a loop to create READERS reader threads
- ▶ There is nothing really new here
- ▶ Some of the output from the program is shown on the following slide

# Pthreads

```
int main(int argc, char *argv[]) {
    pthread_t theWriter;
    pthread_t readers[READERS];
    long t;
    int rc;
    rc = pthread_create(&theWriter, NULL, writer, (void*) t);
    if(rc) {
        printf("Error, writer failed to start: %d\n",rc);
        exit(-1);
    }
    for(t=0; t<READERS; t++) {
        rc = pthread_create(&readers[t], NULL, reader, (void*) t);
        if(rc) {
            printf("Error, reader failed to start: %d\n",rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

# Pthreads

```
reader 0 in database, total readers 1
reader 0 leaving database
reader 1 in database, total readers 2
reader 1 leaving database
reader 2 in database, total readers 3
reader 2 leaving database
reader 4 in database, total readers 1
reader 4 leaving database
reader 3 in database, total readers 1
reader 3 leaving database
writer in database
write out of database
reader 2 in database, total readers 1
reader 1 in database, total readers 3
reader 1 leaving database
```

# Pthreads

- ▶ In pthreads a mutex is used to protect a critical region
- ▶ Pthreads uses condition variables to signal that a condition has occurred
- ▶ A condition variable is manipulated within a critical region, so it is always associated with a mutex
- ▶ The condition variable type is `pthread_cond_t`
- ▶ These variables can be initialized in two ways, the first way is:  
`pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`
- ▶ The second way is to use `pthread_cond_init`, which has two parameters, a pointer to a condition variable and NULL

# Pthreads

- ▶ The `pthread_cond_wait()` procedure has two parameters:
  - ▶ A pointer to a condition variable
  - ▶ A pointer to a mutex
- ▶ The mutex must be locked when the procedure is called, that is it is within a critical region protected by the mutex
- ▶ The thread gives up the mutex when `pthread_cond_wait` is called
- ▶ The procedure waits until the condition is signalled
- ▶ The `pthread_cond_signal()` procedure is used to signal a condition variable
- ▶ Its single parameter is a pointer to the condition variable

# Pthreads

- ▶ The call to `pthread_cond_signal` must occur in a critical region that is protected by the same mutex as the `pthread_cond_wait`
- ▶ The thread will be unblocked from the `pthread_cond_wait`, but it will not be able to execute until the thread that called `pthread_cond_signal` unlocks the mutex
- ▶ There is also a `pthread_cond_broadcast()` procedure that will unblock all the threads waiting for the condition variable
- ▶ Only one of these threads will be able to execute when the mutex is unlocked

# Pthreads

- ▶ Earlier we examined a solution to the producer-consumer problem using semaphores
- ▶ We can also solve this problem using condition variables that play a similar role to the semaphores
- ▶ The first solution used semaphores empty and full to keep track of the number of items in the buffer and the number of free slots
- ▶ We can replace them by condition variables full and empty and a regular integer variable, count, which is the number of items in the buffer
- ▶ The first bit of the solution is on the next slide

# Pthreads

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define SIZE 5

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t empty = PTHREAD_COND_INITIALIZER;
pthread_cond_t full = PTHREAD_COND_INITIALIZER;
int count = 0;
```

# Pthreads

- ▶ The following two slides show the procedures for the producer and consumer threads
- ▶ Note that these are close to what we had before with semaphores
- ▶ The following slide has the main procedure followed by the output from the program
- ▶ The output isn't all that interesting
- ▶ With a buffer of size 5 and the producer producing 20 items, we expect it to fill up approximately three times

# Pthreads

```
void *producer(void* tid) {  
    int i;  
    for(i=0; i<20; i++) {  
        pthread_mutex_lock(&mutex);  
        if(count == SIZE) {  
            printf("Producer: buffer is full\n");  
            pthread_cond_wait(&empty, &mutex);  
        }  
        count = count+1;  
        pthread_cond_signal(&full);  
        pthread_mutex_unlock(&mutex);  
    }  
    pthread_exit(NULL);  
}
```

# Pthreads

```
void *consumer(void *tid) {  
    int i;  
    for(i=0; i<20; i++) {  
        pthread_mutex_lock(&mutex);  
        if(count == 0) {  
            printf("Consumer: buffer is empty\n");  
            pthread_cond_wait(&full, &mutex);  
        }  
        count = count-1;  
        pthread_cond_signal(&empty);  
        pthread_mutex_unlock(&mutex);  
    }  
    pthread_exit(NULL);  
}
```

# Pthreads

```
int main(int argc, char *argv[]) {  
    pthread_t pro;  
    pthread_t con;  
    int rc;  
    rc = pthread_create(&pro, NULL, producer, NULL);  
    if(rc) {  
        printf("Error: can't create producer\n");  
        exit(-1);  
    }  
    rc = pthread_create(&con, NULL, consumer, NULL);  
    if(rc) {  
        printf("Error: can't create consumer\n");  
        exit(-1);  
    }  
    pthread_exit(NULL);  
}
```

# Pthreads

Producer: buffer is full

Consumer: buffer is empty

Producer: buffer is full

Consumer: buffer is empty

Producer: buffer is full

Consumer: buffer is empty

# Semaphores

- ▶ Semaphores are a more recent addition to pthreads and are used outside of pthreads
- ▶ There are two types of semaphores in Linux, those that are used within a single process to synchronize threads, and those that can be used between processes
- ▶ We will only look at the first kind, since they are easier to use
- ▶ The semaphore data types and procedures are in the semaphore.h include file, they are not part of pthread.h
- ▶ A semaphore is declared in the following way:  
`sem_t semaphore;`

# Semaphores

- ▶ Semaphores cannot be initialized when they are declared, instead the `sem_init()` procedure must be called:

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- ▶ The first parameter is the semaphore to be initialized, and the third parameter is the initial value of the semaphore
- ▶ The `pshared` parameter indicates whether the semaphore is shared between processes, the value 0 indicates that it is only used within a single process

# Semaphores

- ▶ The two main procedures that are used to manipulate semaphore are:

```
int sem_wait(sem_t *sem);  
int sem_post(sem_t *sem);
```
- ▶ When `sem_wait()` is called it decrements the value of the semaphore, if it is greater than zero
- ▶ If the semaphore becomes zero, `sem_wait()` blocks the thread and waits for it to become greater than zero
- ▶ The `sem_post()` procedure increments the value of the semaphore, this procedure never blocks

# Semaphores

- ▶ With this in hand we can go back to our producer-consumer solution and produce a semaphore version of it in pthreads
- ▶ This solution is very similar to the previous one, the program starts with:

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define SIZE 5
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
sem_t empty;
sem_t full;
```

# Semaphores

## Producer

```
void *producer(void* tid) {  
    int i;  
    for(i=0; i<20; i++) {  
        sem_wait(&empty);  
        pthread_mutex_lock(&mutex);  
        printf("produce item\n");  
        pthread_mutex_unlock(&mutex);  
        sem_post(&full);  
    }  
    pthread_exit(NULL);  
}
```

## Consumer

```
void *consumer(void *tid) {  
    int i;  
    for(i=0; i<20; i++) {  
        sem_wait(&full);  
        pthread_mutex_lock(&mutex);  
        printf("consumer item\n");  
        pthread_mutex_unlock(&mutex);  
        sem_post(&empty);  
    }  
    pthread_exit(NULL);  
}
```

# Semaphores

```
int main(int argc, char *argv[]) {  
    pthread_t pro;  
    pthread_t con;  
    int rc;  
    sem_init(&empty, 0, SIZE);  
    sem_init(&full, 0, 0);  
    rc = pthread_create(&pro, NULL, producer, NULL);  
    if(rc) {  
        printf("Error: can't create producer\n");  
        exit(-1);  
    }  
    rc = pthread_create(&con, NULL, consumer, NULL);  
    if(rc) {  
        printf("Error: can't create consumer\n");  
        exit(-1);  
    }  
    pthread_exit(NULL);  
}
```

# Summary

- ▶ Examined the system calls that are associated with processes:
  - ▶ fork()
  - ▶ exec procedures
  - ▶ wait procedures
- ▶ Introduced pthreads and the main procedures in that package
- ▶ Examined how some of the standard concurrency problems can be solved in pthreads



# CSCI 3310

# Input and Output

# Part Two

MARK GREEN  
ONTARIO TECH

# Introduction

- ▶ Already examined basic I/O operations like opening and reading files
- ▶ Now examine more directory and file descriptor oriented operations
- ▶ Earlier mentioned that a file was really an inode on disk and multiple directories can refer to the same inode
- ▶ Multiple directory entries referring to the same inode are called links
- ▶ There are two kinds of links:
  - ▶ Hard – the inode is stored in the directory entry
  - ▶ Soft – the file path is stored in the directory entry

# Links

- ▶ Hard links came first, so we will start there
- ▶ In this case there are multiple directory entries that refer to the same inode
- ▶ When the file is first created there is a single directory entry with the inode
- ▶ This first entry is used to create other entries, this can be done with the link program, or the link system call:

```
int link(const char *origpath, const char *newpath);
```
- ▶ The inode is copied from the original path to the new path

# Links

- ▶ Hard links can only be used in the same file system, since they are manipulating inodes
- ▶ All directories with the same inode are equal, they share all the file properties, changing one changes all the others
- ▶ Note, link should only be used to link files and not directories
- ▶ A link can be removed by calling unlink:  
`int unlink(const char *pathname);`
- ▶ The removes the entry from the directory and reduces the inode's link number by 1

# Links

- ▶ Each inode keeps track of the number of links to it, when this number reaches 0 the file is removed from the file system
- ▶ A more general form of link is a symbolic link, in this case the directory entry stores the path to the file or directory that is being linked
- ▶ This can be on the same or different file system
- ▶ The inode doesn't know about its symbolic links, the linked to directory entry and inode can be removed and the symbolic link will continue to exist
- ▶ Referencing this symbolic link will lead to an error

# Links

- ▶ A symbolic link can be created with the `ln` program or the `symlink` system call:

```
int symlink(const char *origpath, const char *newpath);
```
- ▶ A new directory entry is created at `newpath`, this directory entry contains `oldpath` as its value, this is basically a text string
- ▶ A symbolic link behaves more or less like a regular file, except that when it is referred to the path is replaced by the textual content of the symbolic link
- ▶ Thus, the linked to file doesn't need to exist, this can be confusing, the file clearly exists in the directory, but `open` returns an error saying the file doesn't exist!

# Renaming Files

- ▶ A file can be renamed using the mv program or the rename system call:

```
int rename(const char *oldpath, const char *newpath);
```

- ▶ The directory entry oldpath is removed and the directory entry newpath is created, the inode from oldpath is moved to newpath
- ▶ This operation is atomic, so there will be no unexpected results
- ▶ In programs, files within the same directory are often renamed, but rename can be used on different directories

# Duplicating File Descriptors

- ▶ There are two system calls that can be used to duplicate file descriptors
- ▶ The simplest is dup():

```
int dup(int oldfd);
```
- ▶ This creates a new file descriptor that points to the same file table entry as the old file descriptor
- ▶ Both file descriptors share all the information about the file
- ▶ This is different from opening the file twice, this creates two table entries, with different file pointers

# Duplicating File Descriptors

- ▶ The dup() system call returns the lowest free file descriptor
- ▶ The dup2() system call specifies what the new file descriptor should be:

```
int dup2(int oldfd, int newfd);
```

- ▶ The old file descriptor is duplicated and the new file descriptor references it
- ▶ If the new file descriptor referenced an open file, that file is automatically closed

# Duplicating File Descriptors

- ▶ The dup2() system call is used extensively by the shell to handle I/O redirection
- ▶ Consider the following shell command:

```
program >output
```

- ▶ In this case standard output is redirected to the file output, in program file descriptor 1 no longer refers to the terminal, but the file output
- ▶ This can be done in the following way:

```
fd = open("output", O_WRONLY, 0);  
dup2(fd, 1);
```

# Pipes

- ▶ Pipes are a unidirectional communications channel between processes
- ▶ A pipe has two ends, two file descriptors, one can be used to write to the pipe and the other can be used to read from the pipe
- ▶ The data is buffered in the kernel, it is not part of the file system
- ▶ There is a limited amount of buffer space, so a write to a pipe can block, similarly a read will block if there is no data in the pipe
- ▶ Pipes have been part of Unix since the very early versions and they are used extensively by the shell

# Pipes

- ▶ A pipe is created by calling the pipe() system call:  
`int pipe(int fds[2]);`
- ▶ `fds[0]` is used for reading and `fds[1]` is used for writing
- ▶ Consider the following shell command:  
`program1 | program2`
- ▶ To handle this the shell will create a pipe
- ▶ Before starting `program1` it will `dup2(fds[1],1)`
- ▶ Before starting `program2` it will `dup2(fds[0],0)`

# fcntl

- ▶ The fcntl system call can be used to modify the behavior of an open file, its declaration is:  

```
int fcntl(int, fd, int command, long arg);
```
- ▶ The first parameter is the file descriptor to operate on
- ▶ The second parameter is the operation to be performed, the man page for fcntl lists many possible commands
- ▶ The third parameter is the parameter to the operation
- ▶ Some operations have no parameter, others interpret the parameter as a pointer

# fcntl

- ▶ An example of the use of fcntl is the close on exec flag
- ▶ If this flag is set the file descriptor will be closed when an execve call occurs, this includes all the procedures in the exec family
- ▶ This can be done in the following way:

```
fcntl(fd, F_SETFD, 1);
```
- ▶ It can be turned off by passing 0 as the parameters
- ▶ The current value of this parameter can be retrieved by calling fcntl with F\_GETFD as the command
- ▶ A non-zero return indicates that the flag has been set

# Summary

- ▶ Introduced the notion of hard and soft links
- ▶ Examined the system calls that can be used to manipulate links
- ▶ Introduced the notion of pipes
- ▶ Examined the system call that can be used to create pipes
- ▶ Shown how the shell can implement redirection and pipelines of commands

# CSCI 3310

# Memory

# Management

MARK GREEN  
ONTARIO TECH

# Introduction

- ▶ Processes need memory in order to run
- ▶ Memory is a limited resource
- ▶ Thus, it must be managed
- ▶ The part of the operating system responsible for this is the **memory manager**

# Memory Hierarchy

- ▶ Modern computers have several layers of memory, called the **memory hierarchy**:
  - ▶ Cache – fastest memory, part of the CPU chip, expensive and volatile
  - ▶ Main Memory – slower memory, but much larger amount, on a bus separate from the CPU chip, also volatile memory
  - ▶ Disk – very slow memory, but very cheap, non-volatile
- ▶ The hierarchy goes from fast and expensive, to slow and cheap, want to carefully manage the movement between levels

# Memory Hierarchy

- ▶ The cache is largely managed by the CPU chip, the operating system plays a minor role
- ▶ It is important for performance, but we won't consider it
- ▶ Disks and file systems are our next major topic, wait until then to discuss
- ▶ Current part of the course restricted to management of main memory

# Address Spaces

- ▶ At the lowest hardware level there is a linear address space for main memory
- ▶ Starts at zero and bytes have consecutive addresses up to the amount of installed memory
- ▶ This is what the memory bus and chips see
- ▶ Early computers, and some microcontrollers still use this memory model
- ▶ Really only supports one program running at a time, the program has the whole address space and can access all of memory

# Address Spaces

- ▶ With multiple processes this doesn't work:
  - ▶ Translation of program addresses into physical addresses
  - ▶ Restrict processes from accessing memory of other processes
- ▶ If a program can be located anywhere in memory there must be some way of relocating its addresses to its address space
- ▶ Program is compiled assuming its code will start at address 0
- ▶ All jumps, branches and procedure calls are based on this starting address
- ▶ But, this is not where the program will eventually reside in main memory

# Address Spaces

- ▶ Want program to reference the correct memory without a lot of cost
- ▶ Can't recompile the program each time it moves in memory!
- ▶ Add **a base register** to the hardware, the location where the program has been placed in memory
- ▶ Add the base register to all the program's addresses whenever they are used
- ▶ This is done in hardware in the CPU, the program isn't aware of this

# Address Spaces

- ▶ The base register protects programs with lower memory addresses, but it doesn't protect programs running in higher memory
- ▶ Solved by adding a **limit register**, the highest memory address that the program can access
- ▶ Each memory access is compared to the limit register, if it is higher the program halts with an error
- ▶ This is a simple approach that requires very little extra hardware, but allows several programs to share main memory
- ▶ This solution was introduced in the 1960s, starting with CDC 6000 family of computers

# Address Spaces

- ▶ With this model a process still needs to be smaller than main memory
- ▶ On early systems (into the 1980s) the program address space was smaller than the amount of main memory, so this wasn't a problem
- ▶ But, we want to have many processes running at the same time, more processes than there is memory
- ▶ Can't have all the processes in memory at the same time
- ▶ The first solution to this problem was swapping

# Swapping

- ▶ The main idea behind swapping is that processes would reside on disk
- ▶ When it was time to run the processes it was copied into main memory and possibly another process would be swapped out to disk
- ▶ Typically a part of the disk drive was reserved for swapping, not user accessible
- ▶ Swapping a process wasn't fast, so schedulers gave priority to the processes that were already in memory

# Swapping

- ▶ First question is how much memory do we give a process?
- ▶ We could give it the whole address space for a program, but this could waste a lot of memory
- ▶ Want to give a process the smallest amount of memory possible, but give it some room to grow:
  - ▶ Program code – know this from compiler
  - ▶ Heap – global variables from compiler, some room for dynamically allocated memory
  - ▶ Stack – enough from some procedure calls

# Swapping

- ▶ If the process remains small everything is okay, but what happens if it needs more memory, the heap or stack grows?
- ▶ If there isn't an adjacent process, could expand into the empty memory between processes
- ▶ This will work some of the time
- ▶ Another approach is to first swap the process out, change the size of the program and then swap it back in again
- ▶ Note that this is expensive
- ▶ Some programs started by malloc'ing a large block of memory and then freeing it, this forces the process to grow so there won't be subsequent swaps on memory allocation

# Managing Free Memory

- ▶ In order to swap a process in the memory manager must be able to find free memory for it
- ▶ Thus, the memory manager must keep track of all the free and allocated memory
- ▶ There are two basic approaches to this
- ▶ The first approach is the use of a bit vector
- ▶ Physical memory is divided into blocks, usually on the order of a kilobyte
- ▶ Each block has a bit in the bit vector, 0 if the block is free, 1 if the block is occupied

# Managing Free Memory

- ▶ When a process leaves memory, all its bits are set back to 0, and when a process is allocated memory its bits are set to 1
- ▶ This is quite efficient
- ▶ The problem comes when a process must be swapped into memory and we need to find blocks for it
- ▶ The memory manager must search through the bit vector looking for enough adjacent blocks with 0 values to accommodate the process
- ▶ This can be an expensive operation

# Managing Free Memory

- ▶ The second approach is based on maintaining linked lists of free and allocated blocks of memory
- ▶ In the allocated list there will be one block for each process in memory, the size of the block is the size of the process
- ▶ On the free list the blocks are made as large as possible, blocks can be merged when a process is removed from memory
- ▶ Adding and removing blocks can be quite quick, the size of these lists is limited by the number of processes that can fit into main memory, they won't be very long

# Managing Free Memory

- ▶ The complicated part comes when we need to allocate memory for a process
- ▶ There are a number of algorithms that have been proposed for this, all search the free list
- ▶ In the **first fit** algorithm the memory manager starts at the beginning of the free list and looks for the first block large enough to hold the process
- ▶ In the **next fit** approach the search starts at the point where the last block was found
- ▶ Equalize the chance of a block being selected

# Managing Free Memory

- ▶ In the **best fit** approach the memory manager searches for the block that is closest in size to the process
- ▶ This involves searching the entire free list
- ▶ There are several other variations on this general theme
- ▶ Lots of debate and few facts
- ▶ Some experimental studies show that first fit is the best
- ▶ Best fit results in too many small blocks between processes that are hard to use – memory fragmentation

# Managing Free Memory

- ▶ There are two issues here:
- ▶ First, with swapping the entire program must be in memory, in most cases only a subset of program memory is being used at any one time
- ▶ A large part of the allocated memory isn't being used, a waste of resources
- ▶ Second, what happens if there isn't enough free memory to swap in a process?
- ▶ The process will block until enough memory becomes available, this favours small processes, large processes may need to wait a long time

# Virtual Memory

- ▶ We now have two problems:
  - ▶ Only need part of a process to be in memory at any one time
  - ▶ What happens to processes that are larger than available memory?
- ▶ These two problems are related, if we only need to have part of the process in memory we can easily handle larger processes
- ▶ This is what motivated virtual memory
- ▶ The key idea is to divide the process's memory space into **pages**, which can be mapped into physical memory

# Virtual Memory

- ▶ The pages vary in size from 4kb to 1mb or even 1gb
- ▶ Main memory is also divided into pages, called **page frames**, which are the same size as the program pages
- ▶ The page frames are typically the same size as the process pages
- ▶ The pages are mapped to page frames using a **page table**
- ▶ there is one entry in this table for each of the pages in the process, this table has two important entries:
  - ▶ A bit that indicates whether the page is in memory
  - ▶ The page frame the page maps into if its in memory

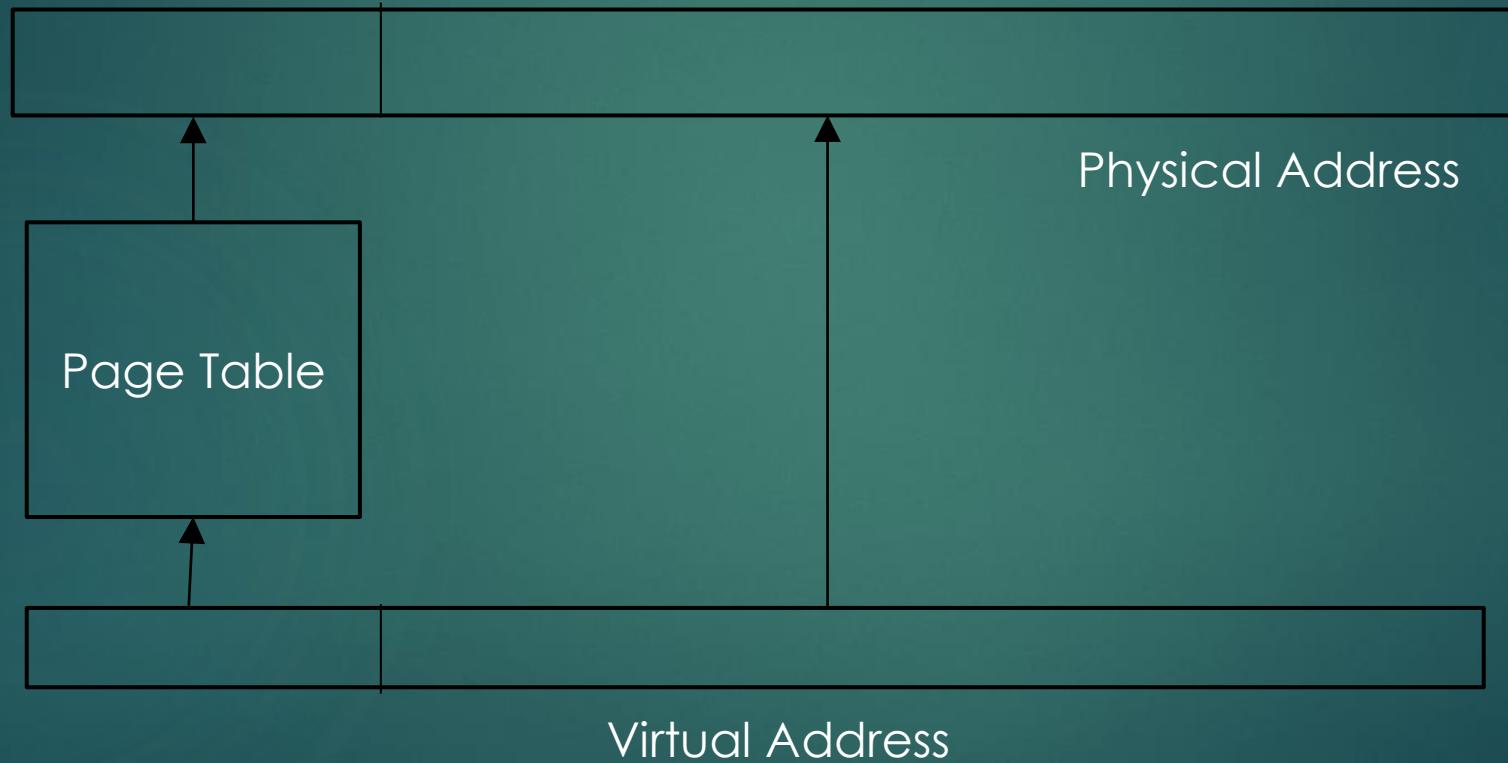
# Virtual Memory

- ▶ The page table is part of the CPU and the mapping is handled by the **memory management unit (MMU)** that is part of the CPU
- ▶ How do we do this efficiently?
- ▶ To start with page sizes are always a power of two, that way we can easily divide the address into two parts
- ▶ The lower bits give the location within a page, and the higher bits give the page number
- ▶ The page table can then be indexed by the page number, this gives the page frame number, which then becomes the high order bits of the physical memory address

# Virtual Memory

- ▶ Consider a simple example, a 16 bit address space
- ▶ Assume a 4kb page, this requires 12 bits, the bottom 12 bits of the address
- ▶ This leaves 4 bits as the page number
- ▶ The lower 12 bits are used directly to form the lower 12 bits of the physical address
- ▶ The top 4 bits are used as an index into the page table, that will have 16 entries
- ▶ 4 bits will be taken from the page table entry to make up the high order bits of the physical address

# Virtual Memory



# Virtual Memory

- ▶ In this example we assumed that the physical address space was the same size as the virtual address space
- ▶ This doesn't need to be the case
- ▶ In early virtual memory systems we had more physical memory than a program could address
- ▶ In that case the top 4 bits could be converted into 8 bits
- ▶ Now our virtual address space is much larger than the physical address space
- ▶ In this case the page table could produce fewer bits for the physical address

# Virtual Memory

- ▶ This works nicely if all the pages are in memory, but that's not what we want
- ▶ We add an extra bit to the page table entry that indicates whether the page is in memory, 1 if in memory and 0 if not
- ▶ If a process references a page that isn't in memory the CPU causes a **page fault** which will be trapped by the operating system
- ▶ The process will then be blocked while the requested page is read into memory, some other process could run during this time
- ▶ When the page is in memory, the instruction that caused the page fault will be executed again, since the data is now available

# Virtual Memory

- ▶ There are a number of other things that can go in the page table:
  - ▶ Modified bit – this bit is set if a write has occurred to the page, if the modified bit is 0 the page will not need to be written back out to disk if its memory is needed
  - ▶ Referenced bit – indicates if the page has been referenced recently, if a page hasn't been used recently it's a candidate for being page out
  - ▶ Protection bits – indicates whether the page is write protected, whether it can be executed, etc.

# Virtual Memory

- ▶ This works well for 16 to 20 bit address spaces, but for larger address spaces we start running into trouble
- ▶ With 32 bit address spaces and 4kb pages we would have 20 bits for the page table indices
- ▶ The page table would have  $2^{20}$  entries, which means it probably wouldn't fit on the CPU chip and would need to be stored in memory
- ▶ With a 64 bit address space the situation is even worse, the page table would no longer fit in main memory
- ▶ Even with 32 bits, we will need to do an extra memory request for each process memory request

# Virtual Memory

- ▶ There are two issues we need to deal with here:
  - ▶ The page table is getting too large to store
  - ▶ With the page table in main memory we double the number of memory requests, since we need to read the page table
- ▶ We want page table access to be fast, so we have a problem
- ▶ An observation assists with this problem, programs don't jump all over memory
- ▶ Programs are basically sequential, they are lists of statements, for loops, etc.

# Virtual Memory

- ▶ When we are executing a program, the instructions will be in a small number of pages
- ▶ For a procedure, all the local variable are adjacent to each other on the stack, likely all in the same page
- ▶ Thus, at any point in time we only reference a small number of pages, we only need a small amount of memory in the CPU to store this information
- ▶ This is the main idea behind **translation lookaside buffers**, or **TLB**
- ▶ The is a small table in the CPU that maps virtual pages to page frames

# Virtual Memory

- ▶ How are we going to do this? We have a 20 bit page number, we can't have a table that we index
- ▶ Instead we store both the page number and the page frame in the TLB, but we can no longer use the page number as the index, do we need to search the table?
- ▶ No, the TLB typically has between 64 and 256 entries, the CPU does the search in parallel
- ▶ This is smart memory, the CPU broadcasts the page number to all the entries in the TLB, they compare it to their page number, and if an entry has it, it responds to the request with the page frame

# Virtual Memory

- ▶ Since the TLB is relatively small, we can have dedicated circuitry on each entry to do the comparison
- ▶ If the page isn't in the TLB, then the CPU must go to the page table in main memory
- ▶ This will create a new entry in the TLB, if the TLB is full, one of its entries will be selected for removal
- ▶ Most of the information in the old entry will be in the page table, there are only a few bits that need to be copied to it

# Virtual Memory

- ▶ A TLB entry typically contains the following information:
  - ▶ The virtual page number
  - ▶ The page frame
  - ▶ The protection bits
  - ▶ The modified bit
  - ▶ A valid bit – indicates whether this entry in the TLB is currently in use
- ▶ Of all this information only the modified bit needs to be copied back to the page table

# Virtual Memory

- ▶ If we have reasonable locality of reference, the TLB can efficiently support very large address spaces
- ▶ The TLB is usually part of the CPU and handled in hardware
- ▶ Some older CPUs had a software implementation, but this is no longer used
- ▶ We still have the problem of a very large page table
- ▶ Its unlikely that a process with a 64 bit address space will need all of that memory, where will it get it?
- ▶ Even 32 bit programs rarely used their entire address space

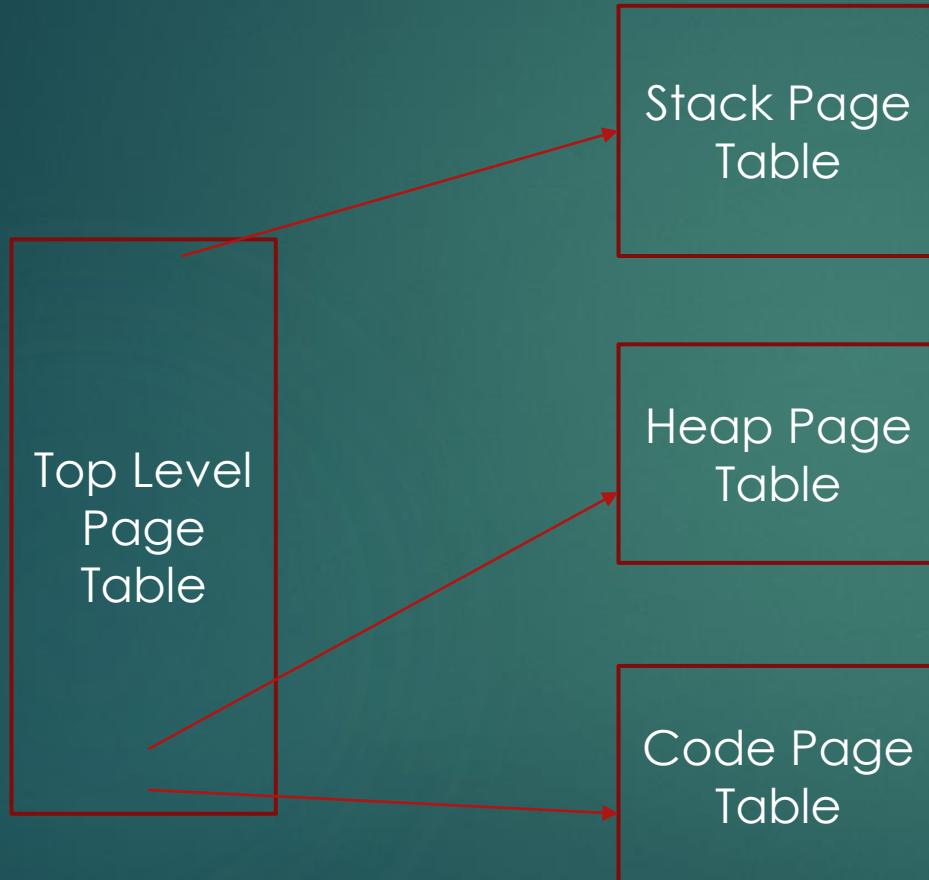
# Virtual Memory

- ▶ This means that large parts of the page table will be empty, we don't need to store that information
- ▶ Consider our typical program model, the code and heap are in low memory, the stack is in high memory, the stack and heap grow towards each other
- ▶ With this model there will be a few pages in low memory, and a few pages in high memory, and nothing in between
- ▶ This suggests a different type of page table organization, one that can accommodate large amounts of empty space

# Virtual Memory

- ▶ Consider a 32 bit address space, the bottom 12 bits can be the address within a page
- ▶ This leaves us with the top 20 bits, this can be divided into two 10 bit fields
- ▶ One of the fields can serve as the page table index, this gives a page table with 1024 entries
- ▶ The other field can be used as an index into a table of page tables
- ▶ It is used to select the page table that the other field serves as an index into
- ▶ This is called a **multilevel page table**

# Virtual Memory



Have only 4 tables with  
1024 entries each, much  
less space than a single  
table with  $2^{20}$  entries

# Virtual Memory

- ▶ This approach saves a considerable amount of memory
- ▶ Each process will have its own page table, there will be one in memory for each process that is in memory
- ▶ While a process is in memory the complete page table should also be in memory
- ▶ A smaller set of page tables makes it possible to fit more processes in memory
- ▶ Easier to allocate memory for the page tables since it can be done incrementally as the process grows

# Page Replacement Algorithms

- ▶ At some point memory will be full, we need to bring a page into memory, select a page to be removed
- ▶ Where do these pages go?
- ▶ If it's a code page, we don't have to worry, program can't write to it and there is a copy on disk
- ▶ Data pages do need to be saved somewhere, the swap file or partition can be used for this
- ▶ We are no longer swapping, but we still need to save pages that aren't in memory, the swap space is good for this

# Page Replacement Algorithms

- ▶ Now we have the problem of selecting the page that will be swapped out
- ▶ We don't want to select a page that will need to be swapped in very soon, it doesn't make sense to swap it out and then almost immediately swap it back in again
- ▶ Ideally, we want to swap out the page that won't be used for the longest time, it will be needed sometime in the distant future, as far as the CPU is concerned
- ▶ Problem: we don't know which page this is, and there is no way of determining it

# Page Replacement Algorithms

- ▶ There are many algorithms for making the selection, they are generally useful in many different areas
- ▶ Anything that uses a cache must deal with this problem, such as the TLB
- ▶ A web server caches frequently used pages, will need to remove pages from the cache at some point
- ▶ A network game has a cache of the regions that most players are in, as players move will need to bring new regions into the cache and select an old one to remove
- ▶ These are generally interesting algorithms

# Page Replacement Algorithms

- ▶ There is an important consideration for most of these algorithms:
  - ▶ Do we only consider pages for the current process? - **local**
  - ▶ Do we consider all the pages that are in memory? – **global**
- ▶ Some of the algorithms can do either, others are local only
- ▶ The problem with local algorithms is that processes always have the same size – they cannot grow or shrink
- ▶ This can lead to performance problems, if a process doesn't have enough pages it will be continually swapping pages
- ▶ If it has too many pages, we are wasting memory

# Page Replacement Algorithms

- ▶ Local algorithms are somewhat easier and tend to be more efficient, fewer pages to consider
- ▶ Each page table entry has two useful bits:
  - ▶ R – the page has been referenced recently
  - ▶ M – the page has been modified
- ▶ When a process starts none of its pages are in memory
- ▶ As it starts executing instructions there will be page faults and pages will be brought into memory
- ▶ These pages have their R bit set, and marked Read Only

# Page Replacement Algorithms

- ▶ If the process writes to a page there will again be a page fault
- ▶ At this point the M bit is set, and the page is marked Read/Write
- ▶ On clock interrupts the R bits are set back to zero, in this way we can track whether a page has been used in the last clock interval, approximately 20msec
- ▶ We can now divide the page into four classes:
  1. Not referenced, not modified
  2. Not referenced, modified
  3. Referenced, not modified
  4. Referenced, modified

# Page Replacement Algorithms

- ▶ The **Not Recently Used (NRU)** algorithm randomly selects a page from the lowest numbered class that isn't empty
- ▶ Remember if the M bit is 0, the page doesn't need to be written, saving one disk access
- ▶ This algorithm is easy to understand and is reasonably efficient, in most cases it gives adequate performance
- ▶ The **FIFO** algorithm is based on maintaining a list of all the pages in memory
- ▶ When a page is brought into memory it is placed on the end of the list, the page on the front of the list has been in memory the longest

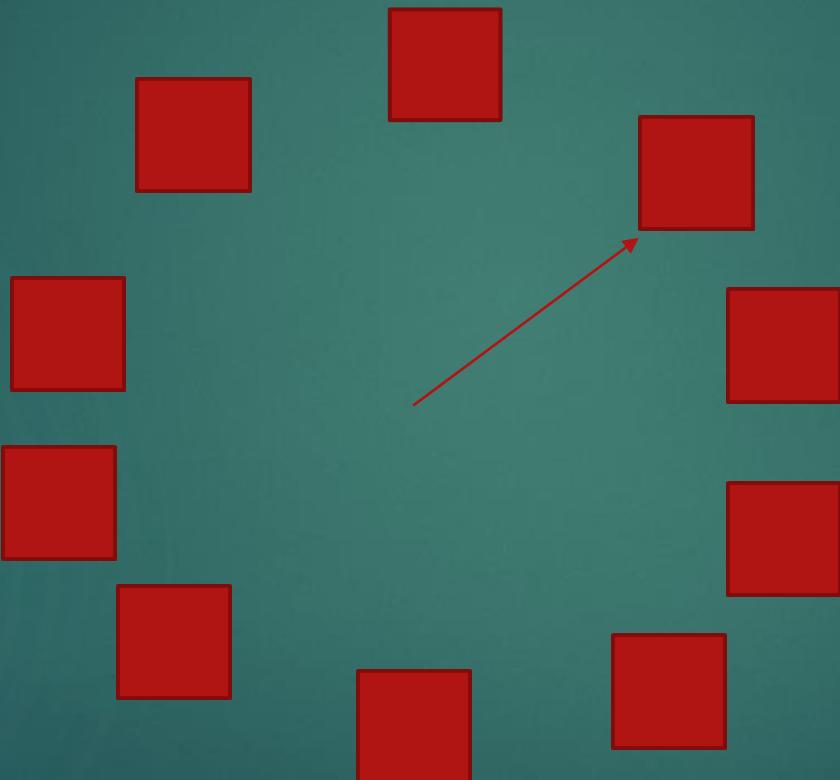
# Page Replacement Algorithms

- ▶ When a page needs to be removed from memory, the one at the front of the list is chosen
- ▶ It's been in memory the longest, so it's a good candidate for removal
- ▶ The problem is this page may still be referenced a lot, as soon as its removed from memory it will be brought back in again, not very efficient
- ▶ The **second chance** algorithm examines the R bit, if its 1, the page is moved to the end of the list and the R bit is set to zero
- ▶ The algorithm then considers the new front of the list

# Page Replacement Algorithms

- ▶ In the worst case all the pages have their R bit set, then we are back to pure FIFO
- ▶ One of the problems with this algorithm is its constantly manipulating a list
- ▶ There is a fixed number of page frames in memory, so once memory is full the list won't change in size, there will be one entry for each page frame
- ▶ This suggests keeping the pages in a circular list, as shown on the next slide
- ▶ The pointer points to the oldest page on the list

# Page Replacement Algorithms



# Page Replacement Algorithms

- ▶ When a page needs to be removed the page at the pointer is examined, if its R bit is 0 it is removed
- ▶ If its R bit is 1, it is set back to 0 and the pointer is move to the next page in the clockwise direction
- ▶ Thus, this is called the **clock** algorithm, and the pointer is call the **hand**
- ▶ The operation is basically the same as the second chance algorithm, but it is more efficient since it involves few pointer manipulations

# Page Replacement Algorithms

- ▶ Observation:
  - ▶ If a page has been heavily used recently, it is likely to be used again
  - ▶ If a page hasn't been used for a long time, its not likely to be used again
- ▶ The is the idea behind the **Least Recently Used (LRU)** algorithm
- ▶ It is close to optimal, but hard to implement
- ▶ Need to keep a list of all pages in memory and update this list on every memory reference
- ▶ We need to have a good approximation

# Page Replacement Algorithms

- ▶ A hardware approach is to have a 64 bit counter, C, that is incremented on each memory reference
- ▶ Each entry in the page table also has a 64 bit field, when the page is referenced C is stored in that field
- ▶ The page to be removed is the one with the lowest value in this field
- ▶ Unfortunately, this rarely exists in hardware, but this is the motivation for the **Not Frequently Used (NFU)** algorithm, which can be implemented in software
- ▶ In this approach each page table entry has a counter, on each clock interrupt R is added to this counter

# Page Replacement Algorithms

- ▶ The page to be removed is the one with the lowest counter value
- ▶ There are two problems with this algorithm:
  - ▶ Pages that have been referenced a lot, but are no longer used will have a large counter value and won't be removed
  - ▶ The most recently added pages will have small counter values and are likely to be removed, but they will almost immediately be paged in again
- ▶ These problems are solved by the **aging** algorithm, which is also counter based

# Page Replacement Algorithms

- ▶ In this approach, on each clock interrupt the counter is shift right one position and R is added to the leftmost (high order) bit
- ▶ In this way pages referenced in the last clock interval will have high values, those that haven't been reference recently will be close to zero
- ▶ The counter can be as small as 8 bits and still get good performance
- ▶ With a 20msec clock and 8 bits, the counter will be zero after 160msec if the page isn't used, that's quite a few instructions
- ▶ Efficient, easy to implement and very good performance

# Page Replacement Algorithms

- ▶ The **working set model** is an important idea in paging algorithms
- ▶ Consider a process that is just starting and none of its pages are in memory
- ▶ It tries to execute the first instruction, causes a page fault, the page with instructions needs to be paged in
- ▶ Soon it will need data and a stack, so more pages will come in
- ▶ This is called **demand paging**
- ▶ Over time more pages will be required, but the number of page faults decreases over time

# Page Replacement Algorithms

- ▶ As the computation evolves, some pages are no longer needed, and others need to be paged in
- ▶ At any point in time the process only uses a fraction of its pages, called locality of reference, this is the same idea behind TLB
- ▶ The pages that a process is currently using is called its **working set**
- ▶ The pages in this set will change over time, it can also change in size
- ▶ A process should be allocated enough page frames to accommodate its working set
- ▶ If not, it will do a lot of paging, called **thrashing**, which will reduce the overall system performance

# Page Replacement Algorithms

- ▶ When a process is swapped out, a scheduling decision, its pages will be removed from memory
- ▶ When its time to be swapped in again, the process could start with none of its pages in memory, but this will cause a lot of paging
- ▶ Instead the paging system can keep track of the working set and swap it all in at once, this will save a lot of paging activity, this is called **prepaging**
- ▶ When it comes time to remove a page, select one of the pages that isn't in the working set, since its not likely to be used

# Page Replacement Algorithms

- ▶ A pure working set based page replacement algorithm is a nice idea, but it's hard to implement
- ▶ For this to work the paging system needs to know the working set for all of its processes
- ▶ It needs to keep track of which pages are being referenced, ideally on an instruction by instruction basis
- ▶ This is just not practical, so we need some kind of approximation
- ▶ The **virtual time** for a process is the length of time that the process has been running, we can use this to approximate our working set

# Page Replacement Algorithms

- ▶ This approach is based on the R bit and a time field in the page table
- ▶ On each page fault the page table is scanned, if the R bit for an entry is set it is reset to 0 and the current virtual time is placed in the time field
- ▶ This is an approximation to the last time the page was referenced, it was sometime in the clock interval
- ▶ When it comes time to evict a page, we set a time interval  $t$  that is at least several time clicks and start examining the page table

# Page Replacement Algorithms

- ▶ If the R bit is set, we skip the entry
- ▶ If R is zero, and the time in the time field is within our threshold  $t$ , the page is also skipped
- ▶ The first page with R=0, and a time field greater than  $t$  is removed from memory
- ▶ If there isn't such a page we look for one with M=0, otherwise a page is picked at random
- ▶ The pages that are left in memory are a good approximation to the working set

# Page Replacement Algorithms

- ▶ The problem with this approach is we need to scan the entire page table, which is not particularly efficient
- ▶ A better approach is to combine working sets with the clock algorithm to produce **WSClock**
- ▶ This is based on constructing a ring from all the page frames for the process, and a pointer to the current page frame under consideration
- ▶ If  $R=1$ , we set  $R=0$  and continue to the next page frame
- ▶ If  $R=0$ ,  $M=0$ , and the page is older than our threshold  $t$ , we just claim this page, since it doesn't need to be written to disk

# Page Replacement Algorithms

- ▶ If  $R=0$ ,  $M=1$  and the page is older than the threshold, the page is scheduled to be written to disk and we advance to the next page frame
- ▶ We could end up scheduling multiple pages to be written, this saves time on the next page request
- ▶ If we go all the way around the ring there is a chance that at least one of the page writes is complete, if so we can use that page
- ▶ Otherwise, we look for any page with  $M=0$ , regardless of its age and claim that page
- ▶ If no page satisfies this condition, a page is selected at random

# Page Replacement Algorithms

- ▶ This approach, or a modification of it is used in many operating systems
- ▶ It is quite efficient and results in very good performance
- ▶ With local page replacement algorithms we have a problem, how many page frames to we allocate to a process?
- ▶ We essentially need a second layer of page management
- ▶ If a process has too few page frames it will do a lot of paging, negative impact on system performance
- ▶ If a process has too many page frames it reduces the number of processes in memory

# Page Replacement Algorithms

- ▶ One simple solution is to give every process the same number of page frames
- ▶ Doesn't really solve our problem, but it does get things started
- ▶ Another simplistic approach is when a process is removed from memory its page frames are given to other processes, proportional to their sizes
- ▶ Not a bad idea
- ▶ The aim of this level of page management is to reduce the overall number of page faults, thus we should be tracking the page faults for each process

# Page Replacement Algorithms

- ▶ **Page Fault Frequency (PFF)** is the number of page faults in a process over a period of time, for example a second
  - ▶ If the PFF of a process is high it needs more page frames
  - ▶ If the PFF of a process is low it has too many page frames
- ▶ A policy decision is to determine the ideal PFF for a system and try to maintain all the processes in this range
- ▶ What is the maximum amount of paging that the system can sustain?
- ▶ If the PFF is too high we need to remove a process from memory

# Page Replacement Algorithms

- ▶ Page frames are a scarce resource, ways of making better use of them should be considered
- ▶ Example: a text editor, many users could be using the test editor at the same time, multiple copies of the program code in memory at the same time
- ▶ Program code is read only, process can't change it
- ▶ It make sense for all the processes to share the program code, reduce the number of page frames required
- ▶ This can be done through the page table

# Page Replacement Algorithms

- ▶ When a process starts, the OS determines if that program is already in memory
- ▶ If it is, the page table for the program code is loaded with the page frames that are already used for the program code
- ▶ The OS also keeps track of the number of processes that are using the same program
- ▶ It doesn't release the page frames until after the last program exits, or is no longer in memory
- ▶ For common programs this can save a significant number of page frames

# Page Replacement Algorithms

- ▶ A similar idea is shared libraries, which can have a bigger impact
- ▶ Many programs use large libraries, for example graphics and GUI libraries
- ▶ While the programs might be different, they are all using the same libraries, so they can share the page frames these libraries use
- ▶ On Linux they are called shared objects and on Windows they are called DLLs
- ▶ When the program is linked, they are not linked to the actual library, but to a stub that will connect with the library at run time, much smaller

# Page Replacement Algorithms

- ▶ The shared library has some aspects of a process, when its loaded into memory it will have its own set of page frames
- ▶ On Windows a DLL even has its own main() procedure
- ▶ The pages for a shared library are shared by all the processes that use it, there is only one copy in memory
- ▶ For large libraries this can be a significant savings of page frames
- ▶ Shared libraries have a number of other advantages
- ▶ If a library has a bug, with a static linking of the library the program must at least be relinked, which can be hard to do if many programs use the library

# Page Replacement Algorithms

- ▶ If a shared library is used, in many cases only the shared library file needs to be modified, one file
- ▶ As long as the external data structures and procedures have the same signatures this is possible
- ▶ This makes maintenance much easier, most operating systems put all of their libraries into shared libraries, making OS updates much easier
- ▶ There is another trick that can be done with shared libraries, the programs that use them don't need to know the procedures in them when they are compiled

# Page Replacement Algorithms

- ▶ When a program connects with a shared library it can enquire about the procedures in the library
- ▶ It can then link to these procedures at run time
- ▶ This is how program extensions work
- ▶ The program is distributed in executable form, but has a well defined interface to shared libraries
- ▶ If a programmer follows this interface, they can provide new features that are added to the program at run time
- ▶ This is a common practice with large applications

# Page Replacement Algorithms

- ▶ So far we have been sharing program code, since it is read only this is quite safe
- ▶ Processes can also share data pages, this can be done by having the page table entries point to the same page frame
- ▶ There are several ways this can be done
- ▶ One of through the use of memory mapped files, the file resides in memory and not on disk
- ▶ Then two or more processes can open the file and perform reads and writes to it
- ▶ This is a controlled form of memory sharing

# Page Replacement Algorithms

- ▶ Some operating systems allow two or more processes to map to the same page frames
- ▶ It is then up to these processes to coordinate references to memory, recall the synchronization primitives
- ▶ Another example is fork()
- ▶ After fork() is called the page tables for the two processes have the same contents, all the data pages are marked read only
- ▶ When one of the processes writes to a data page, a copy of the page is made and marked read/write in both processes
- ▶ This avoids a complete memory copy on fork()

# Implementation Details

- ▶ When a process goes from the ready state to the run state a number of things need to occur:
  - ▶ The TLB needs to be flushed, since it is referencing the pages from the previous process
  - ▶ The MMU now needs to use the page table for the new process
- ▶ We can't avoid flushing the TLB, but this is fairly small, so this isn't a major problem
- ▶ The most efficient way to handle this is to have a pointer to the page table in the MMU
- ▶ We also may need to page the page table into memory

# Implementation Details

- ▶ Examine the process of handling a page fault
- ▶ The page fault is detected by the CPU, but its handled by the operating system
- ▶ The first thing that must occur is saving the state of the CPU when the page fault occurred, this will involve both the CPU and the operating system
- ▶ The OS determines the virtual address that caused the page fault, and whether it was a legal reference
- ▶ If its legal, a page frame must be found for the page that will be brought in from disk

# Implementation Details

- ▶ Note this will involve at least one disk transfer, if not two
- ▶ At this point the process causing the page fault will block, another process will execute
- ▶ When the page is in memory, the page table is updated and the process can start running again
- ▶ At this point the state of the CPU must be restored and the process returns to the instruction that caused the page fault
- ▶ This introduces **instruction backup**, which is largely handled by the CPU and is a complex process

# Implementation Details

- ▶ When the page fault occurs, the CPU is in the process of executing the instruction
- ▶ With modern pipelined CPUs, there are multiple instructions in the pipeline and each instruction takes multiple clock cycles to execute as it flows through the pipeline
- ▶ The instruction may be partially executed when the page fault occurs, this what causes the problems
- ▶ It may have already changed a register value or a location in memory
- ▶ What do we do here?

# Implementation Details

- ▶ There are two things we could do:
  - ▶ Undo anything the instruction has already done, if we are lucky the CPU keeps track of this making it somewhat easy
  - ▶ Record the complete state of the CPU, so the instruction can be safely restarted
- ▶ We must chose one and do it right, this has not been the case in the past
- ▶ If it's not done correctly the program ends up crashing for unknown reasons, pretty much impossible to debug
- ▶ Fix: compiler doesn't generate the instruction that could cause the problem

# Implementation Details

- ▶ Some operating systems like Linux have a paging daemon that periodically scans the page tables for pages that can be paged out
- ▶ The idea is to produce a list of free pages that can quickly be allocated when a page fault occurs
- ▶ If the daemon finds a page with the M bit set it will schedule it for a write to disk
- ▶ If it finds an old page without the M bit set it will immediately add it to the free page list

# Implementation Details

- ▶ One of the issues that must be addressed is DMA
- ▶ When a user process initiates a read or write it will be blocked and another process will start executing
- ▶ If the I/O operation uses DMA to user space there is the chance the page to receive the data will be swapped out
- ▶ In this case the data will be written over top of another process
- ▶ This is bad for both processes
- ▶ There are two solutions to this problem
- ▶ The first is to pin the user page in memory so it can't be swapped out

# Implementation Details

- ▶ With a large amount of physical memory and a small transfer this isn't much of a problem
- ▶ But it could result in pinning down a significant number of page frames
- ▶ Another approach is to do the DMA operation to a kernel buffer, which can't be swapped out
- ▶ When the process runs again, the data is copied from the kernel to the user process
- ▶ This is not as efficient, but is easier to implement and gives more predictable performance

# Summary

- ▶ Covered the techniques used to manage main memory
- ▶ Early operating systems used swapping, since it is relatively easy to implement
- ▶ Swapping doesn't make the best use of memory, so paging has replaced it
- ▶ Discussed ways of organizing the page table, algorithms for page replacement and some of the implementation details

# CSCI 3310

# File Systems

MARK GREEN  
ONTARIO TECH

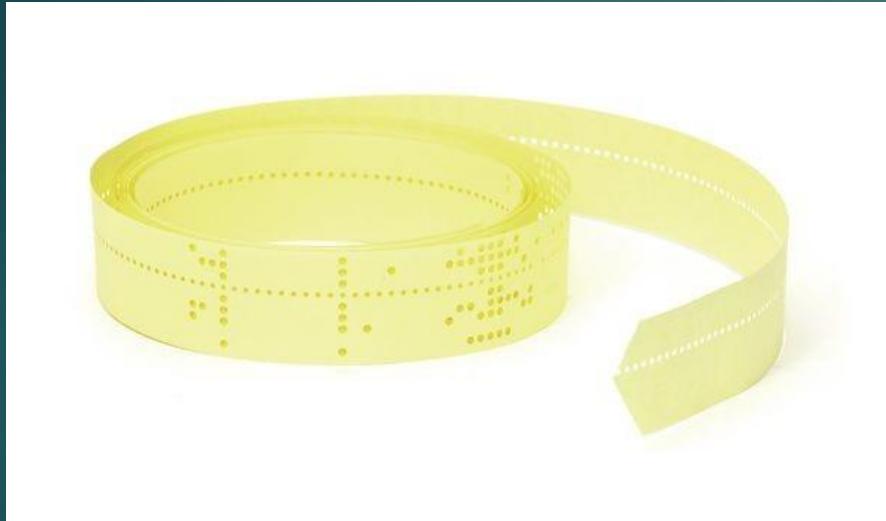
# Introduction

- ▶ Examined memory management that dealt with main memory
- ▶ Several times mentioned paging memory to disk
- ▶ Noted that main memory was volatile, values lost when power is removed
- ▶ We need non-volatile memory, which brings us to file systems, how we can store data on disk

# Hardware

- ▶ Several non-volatile memory techniques have been used in the past:
  - ▶ Paper tape – narrow strip of tape with holes punched in it, both paper tape punches and readers
  - ▶ Punched cards – paper cards with 80 columns of data, again readers and punches
  - ▶ Magnetic tape – tape that can be written by changing magnetic properties and then read – still used today
- ▶ The first two are write once and not eco friendly

# Hardware



# Hardware

- ▶ Disks are now the standard technology, hold large amount of data, reasonably fast and cheap
- ▶ Based on rotating magnetic material, circular in shape
- ▶ Rotation speeds of 5400, 7200 and 10000 rpm
- ▶ A read/write head moves across the surface, concentric tracks that hold data
- ▶ Each track is divided into a number of sectors, all the same size
- ▶ Must read and write in units of sectors

# Hardware



# Hardware

- ▶ To read or write the head must first be moved to the track
- ▶ The time required to do this depends upon the distance the head must move, called the **seek time**, in the 2msec range
- ▶ Then must wait for the data to rotate around to be under the head, called **rotational latency**
- ▶ On average this is half the time required for a rotation, depends upon the rotation speed
- ▶ Time required to read the data is minimal compared to other two
- ▶ Modern disk drives have sizable caches

# Hardware

- ▶ CD-ROM and DVD have a similar structure, except they are read and written optically
- ▶ Not as fast as magnetic disk, but removable
- ▶ Semiconductor versions are more recent, things like flash drives
- ▶ From a software perspective they all behave like a magnetic disk, use a lot of the structures and file systems

# Files

- ▶ A disk is made up of files and directories, examine them before looking at how file systems are implemented
- ▶ Modern operating systems view files as a stream of bytes with no structure
- ▶ Files can grow as long as there is room on the disk
- ▶ Not always the case, older operating systems structured their files with the misguided aim of making them easier to use
- ▶ Imposed a **record** structure on files, typically 80 columns, matching punched card
- ▶ Read or write a whole record at a time

# Files

- ▶ In other cases would automatically add an index to a file, programmer had little control over the index structure
- ▶ In addition programmer had to pre-allocate the file, specify the size of the file, so the OS could allocate space on the disk before the file was used
- ▶ The byte stream model is more flexible, the programmer imposes whatever structure they like on the file
- ▶ Not restricted by what the operating system gives them, the OS gets out of the way

# Directories

- ▶ We need some way of finding files, where they are located on disk
- ▶ Don't expect the programmer to know track and sector number
- ▶ Directories are used for this purpose, associate a name with a location on the disk
- ▶ Came along later than files
- ▶ If I have a deck of cards I don't need a directory, the name of the file is written on the card deck! Just needed to look at the card deck
- ▶ As disks became more popular and larger directories became necessary, before disks stored a single file

# Directories

- ▶ Started with a single directory for the whole disk, frequently only one or two disks
- ▶ All users shared the same directory, disks were small so not a lot of files, maybe several hundred
- ▶ Early PC used this approach MS-DOS
- ▶ On main frames users would have a temporary directory while they were using the computer, files in this directory would disappear when their session was over
- ▶ Needed to back up their files to “permanent files”, handled by a different file system

# Directories

- ▶ One system I used had files and directories that were stored for a single day
- ▶ Load files from cards in the morning, in the evening punch a new deck of cards, typically thousands of cards – real eco friendly
- ▶ All of this was motivated by charging schemes
- ▶ File names have also evolved
- ▶ Initially file names had a maximum length, typically 8 characters, could also have a three character extension
- ▶ Made directory structures simpler, each entry had a fixed size

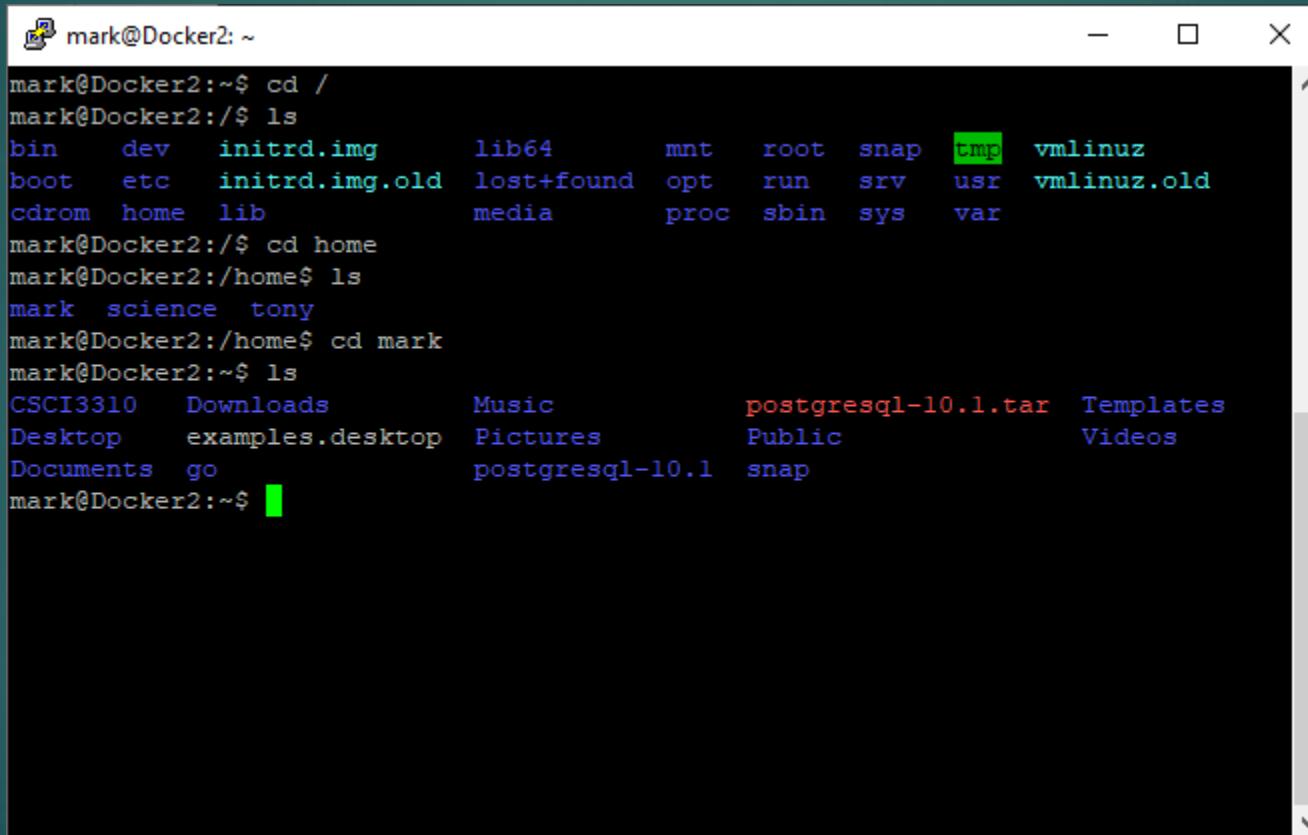
# Directories

- ▶ Some OS's attach meaning to the extension, Windows is a good example of this
- ▶ Others, like Linux allow extension but don't care about them
- ▶ Then there was the original Mac OS, files had two forks:
  - ▶ One for data
  - ▶ One for information on the file
- ▶ Modern OS's place few restrictions on file name length, there usually is a limit, but its typically around 255

# Directories

- ▶ Modern OS's use a hierarchical directory structure
- ▶ The directories are arranged in a tree structure, there is a root directory that has the high level directories underneath it
- ▶ These in turn have directories underneath them
- ▶ On Linux the root directory is /
- ▶ The next slide shows an example walk through of part of a Linux directory tree
- ▶ On windows the drive letter is used as the root directory, for example C: or D:

# Directories



A screenshot of a terminal window titled "mark@Docker2: ~". The terminal displays a series of commands and their outputs:

```
mark@Docker2:~$ cd /
mark@Docker2:/$ ls
bin dev initrd.img lib64 mnt root snap tmp vmlinuz
boot etc initrd.img.old lost+found opt run srv usr vmlinuz.old
cdrom home lib media proc sbin sys var

mark@Docker2:/$ cd home
mark@Docker2:/home$ ls
mark science tony

mark@Docker2:/home$ cd mark
mark@Docker2:~/mark$ ls
CSCI3310 Downloads Music postgresql-10.1.tar Templates
Desktop examples.desktop Pictures Public Videos
Documents go postgresql-10.1 snap

mark@Docker2:~/mark$
```

# Directories

- ▶ On Unix directories were initially normal files, you could use open() and read() on them
- ▶ This produced a very consistent file system, a nice logical model
- ▶ Programs could process directories like normal files
- ▶ Problem: if the directory structure changed all the programs that used directories needed to be changed
- ▶ On modern Unix and Linux systems there are special procedures for manipulating directories

# Directories

- ▶ Directories often include extra information about the file:
  - ▶ Owner
  - ▶ Protection
  - ▶ Creation date
  - ▶ Modification date
  - ▶ Access date
  - ▶ Shareable
  - ▶ Archivable
- ▶ On Linux most of this information is stored in the Inode

# Disk Structure

- ▶ Before examining file systems take a quick look at disk structure
- ▶ In practice we divide a disk into a number of partitions and place a different file system in each partition
- ▶ We could even put a different operating system in each partition, how we get dual boot systems
- ▶ Partitions have two purposes
- ▶ One is to divide a disk up into logical units, each unit has its own purpose
- ▶ Example: have separate partitions for OS, user files, and swap

# Disk Structure

- ▶ The second purpose is to support larger disks
- ▶ Disk size grew faster than file systems, the file system assumed that a disk could only have so many blocks, only allocated enough space for this
- ▶ As disks got bigger a single file system couldn't cover the whole disk, the addresses were too small
- ▶ Solution was to divide the disk into multiple partitions, each one small enough for a file system
- ▶ We went through this several times over the past few decades

# Disk Structure

- ▶ The first block on a disk is called the **Master Boot Record (MBR)**, it contains two things:
  - ▶ The initial boot program that gets everything started, this is a small program
  - ▶ The partition table, lists where all the partitions on the disk are located – one will be the active partition
- ▶ If anything happens to the MBR your system is dead
- ▶ The boot program then goes to the active partition, reads its first block, and then uses it as the second boot program

# Disk Structure

- ▶ This could be a boot loader, like GRUB, or it could be the program that boots the operating system
- ▶ For reliability there can be a second disk which mirrors the first disk in case there is a problem
- ▶ Or a copy of the MBR can be made on another disk so it can be recovered
- ▶ Under normal operation the MBR is not written, it is set up once when the disk is configured

# File Implementation

- ▶ Now we get to the problem of actually storing the file data on disk
- ▶ A file consists of a number of blocks of data, this is the unit of data that the OS uses for files
- ▶ A block is at least one sector, typically it is a small number of sectors, somewhere between 1kb and 4kb
- ▶ The simplest way to store the file is as contiguous blocks on the disk
- ▶ The OS finds a part of the disk that is large enough to hold the whole file and copies it to that location
- ▶ In order to do this we need to know the size of the file when it is created

# File Implementation

- ▶ If the OS can't find a large enough area for the whole file, the file creation operation fails
- ▶ The file cannot grow beyond this length
- ▶ This has the advantage that large reads and writes are very efficient
- ▶ Just need to seek to the start of the file and write the blocks one after another
- ▶ When the file is deleted its blocks are freed
- ▶ One of the problems with this approach is fragmentation
- ▶ If a large file is deleted and then a smaller one comes along to take its place there will be a small number of blocks left over

# File Implementation

- ▶ Eventually there will be lots of small holes between the files and no new files can be created
- ▶ The solution to this is to defragment the disk, all the files are copied to one end of the disk creating one large unallocated area
- ▶ This approach is no longer used on disks due to the fragmentation problem
- ▶ But, it is used on CD-ROMs and a variation is used on DVDs
- ▶ In most cases they are written once, and the sizes of all the files are known before they are written

# File Implementation

- ▶ We want an approach where files can easily grow and they can take advantage of all the disk blocks
- ▶ One idea that has been suggested is to use a linked list of blocks
- ▶ The first few bytes of each block is used as a pointer to the next block in the file
- ▶ As the file is read the OS follows the pointers to find the next block in the file
- ▶ All the free blocks on the disk are also linked together, called the **free list**

# File Implementation

- ▶ When a file is written, blocks are taken off of the free list to grow the file size
- ▶ When a file is deleted its blocks are put back on the free list
- ▶ There is one problem with this, the pointers take up space in the block, so the part of the block available for data is no longer a power of two
- ▶ Programs need to know the pointer size to optimally read and write data, this could change from one file system to another
- ▶ Programs won't be as portable as we would like them to be
- ▶ Also random access is slow, since we need to traverse the linked list to find the block, requires reading blocks from disk

# File Implementation

- ▶ This is a good start and points us in the right direction
- ▶ We can improve on this by storing the pointers in main memory
- ▶ We construct a table, a **File Allocation Table (FAT)** that has one entry for each block on the disk
- ▶ The next pointer that was being stored in the block is now stored in the FAT entry for that block
- ▶ Our blocks are now an even power of two
- ▶ Random access is now better, since we can follow the linked list in memory instead of having to read the blocks from disk

# File Implementation

- ▶ Problem: the entire FAT must be stored in memory, this is okay for small file systems
- ▶ With large disks, the FAT will take up too much room
- ▶ The solution to this is to make the blocks larger so we need fewer entries
- ▶ As the blocks get larger we waste more disk space, the last block of the file won't be completely filled
- ▶ Also the FAT needs to be written to disk occasionally, otherwise if the system crashes, the file system will be corrupt

# File Implementation

- ▶ This is the approach that was used in MS-DOS and is still supported by many operating systems
- ▶ It is still used on flash drives, it is easy to implement and reasonably efficient
- ▶ The problem with FAT is the whole table has to be in main memory, but we only need the entries for the files that are currently open
- ▶ This is the idea behind **index nodes** or **Inode**
- ▶ Each file has an Inode, stored on disk, that contains the file attributes and then pointers to the blocks that belong to the file
- ▶ When the file is opened the Inode is read into memory

# File Implementation

- ▶ If a file is small, pointers to all of its blocks can be stored in the Inode
- ▶ But, as the file grows it will eventually run out of room
- ▶ To solve this problem the last few entries of the Inode point to blocks that contain more pointers
- ▶ In Linux, the first of these pointers points to a block of pointers
- ▶ The second one points to a block that points to blocks of pointers
- ▶ The third one points to a block that points to blocks that point to blocks that contain the pointers
- ▶ This is good enough for the largest disks that we currently have

# File Implementation

- ▶ This is clearly the approach that is used in Linux
- ▶ It is also the approach that is used in the Windows NTFS file system
- ▶ This is significantly more complicated than FAT, so there is the chance that something will go wrong
- ▶ Linux has a program, fsck that checks a file system to determine if it has been damaged
- ▶ It is also capable of repairing some problems
- ▶ This program is typically run when the system boots, though it may not be done every time

# File Implementation

- ▶ One approach to this problem is to use a **journaling file system**
- ▶ Modern disk drives have large buffers, so a block may not be actually written to disk for several seconds, this is to improve efficiency
- ▶ Unfortunately if the system crashes data will be lost
- ▶ For regular file data this is not a big problem, it can usually be restored
- ▶ But, if it involves directories and Inodes we have a problem, the file system could end up being inconsistent or blocks will be lost

# File Implementation

- ▶ Consider deleting a file from a directory, the following operations will be performed:
  - ▶ Remove the file from the directory
  - ▶ Return the Inode to the pool for free Inodes
  - ▶ Return all the disk block to the pool of free disk blocks
- ▶ If a crash occurs after the first operation has been performed there will be no way to reach the Inode and its blocks
- ▶ They will be lost to the file system

# File Implementation

- ▶ The idea behind a journaling file system is that these operations are recorded in a journal file that is forced out to disk when it is updated
- ▶ When the system crashes the file system replays the journal to restore the file system to the point where it crashed
- ▶ The information is stored in the journal in such a way that it can be performed more than once without causing problems
- ▶ In that way the file system doesn't need to know exactly how far the disk got before the crash
- ▶ This is the difference between the ext2 and ext3 file systems on Linux, NTFS also uses this approach

# Directories

- ▶ Directories aren't near as complicated, but there has been some evolution over time
- ▶ Most file systems have settled on each directory entry having the same length
- ▶ There have been some experiments with variable length directory entries, but the decreasing cost of disk space have removed the need for this complication
- ▶ The main thing that was of variable length was the file name, now we just allocate enough room for long file names, say 256 bytes, and not worry about the wasted space

# Directories

- ▶ Now there are basically two broad classes of directory organizations, depending upon how much information is stored in the directory
- ▶ In the Linux world most of the file information is stored in the Inode
- ▶ The directory only contains the file name, the file type (regular file, directory, symbolic link, etc.) and a pointer to the Inode
- ▶ In other OS's the directory contains all the file attributes, and the Inode just contains pointers to disk blocks
- ▶ Hard links aren't possible in this approach, since the file attributes aren't shared and the number of directories that reference the file can't be determined

# File System Administration

- ▶ As a user or system administrator we don't have much control over the scheduler and memory management, they are usually left alone
- ▶ But, we do have a lot of control over file systems
- ▶ It's easy to add a new disk to a computer, so we will walk through the process on Linux, the process on Windows is similar
- ▶ With a new disk the first thing you need to do is partition it
- ▶ This is done with the fdisk program
- ▶ This is an interactive program that will prompt you for how you want the disk partitioned, it is relatively easy to use, but you need to follow the instructions

# File System Administration

- ▶ Be careful that you don't partition an existing disk, you could loose all the information on the disk
- ▶ The next step is to build the filesystem, initialize the partition so it can be used as a file system, this is called format in the Windows world
- ▶ In Linux, the mkfs family of programs is used for this
- ▶ Linux supports a wide range of file systems, mkfs is a front end to the programs that are specific to the file system
- ▶ It is recommended to use the file system specific program, the man page for mkfs points you to the correct program

# File System Administration

- ▶ In most cases you will want a ext2 or ext3 file system, the main difference between the two is that ext3 does journaling
- ▶ An important parameter is the block size for the file system
- ▶ The main Linux file systems support block sizes of 1kb, 2kb and 4kb
- ▶ Other file systems such as the FAT family support a much wider range of block sizes
- ▶ The block is our unit of allocation, the smallest thing that we can store on disk
- ▶ If a file is only one byte long it will still occupy a whole block on disk

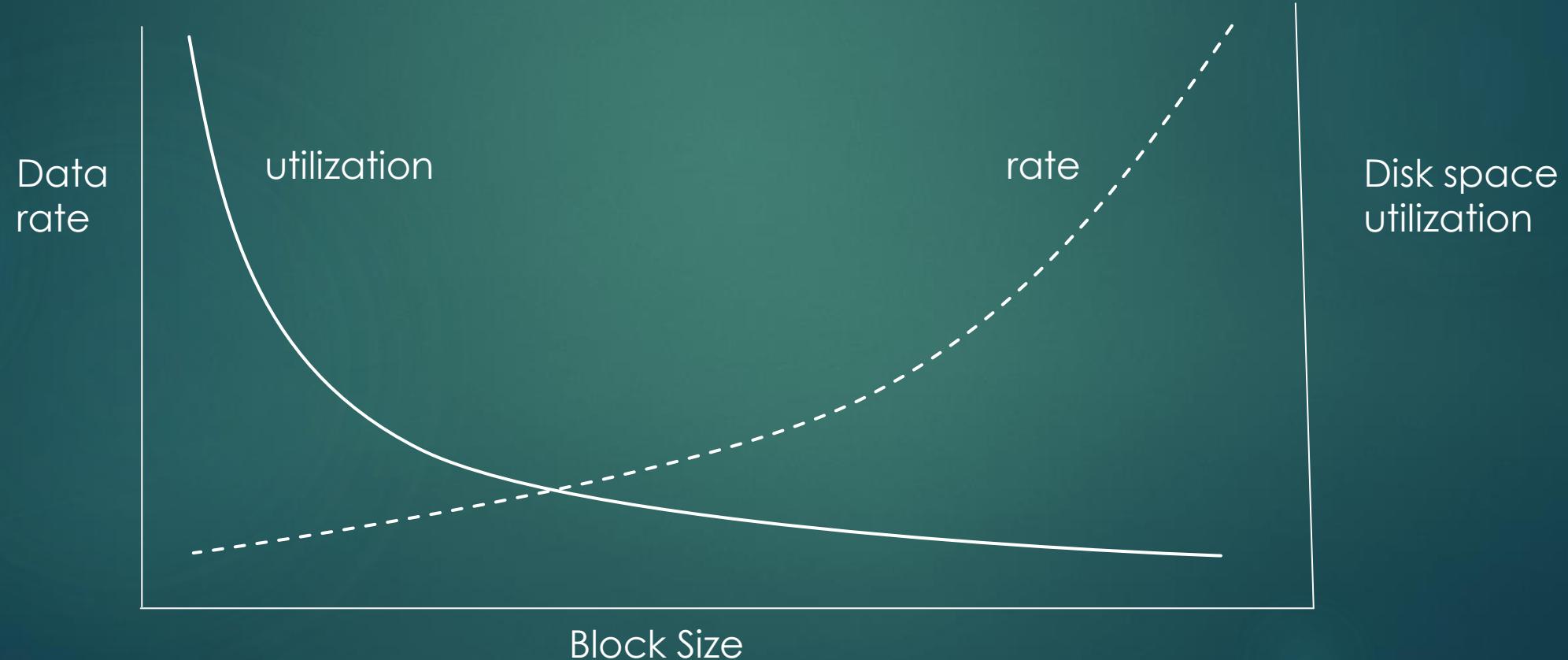
# File System Administration

- ▶ The larger the block size, the more wasted space on the disk, small files will always be a block
- ▶ Even for large files the last block may be largely empty
- ▶ This argues for a small block sizes
- ▶ Most of the time required to read or write a block is seek and rotational latency
- ▶ If a file requires 4 1kb blocks, this overhead occurs 4 times, if the block size is 4kb, the overhead occurs once
- ▶ Thus a 4kb block size is potentially 4x faster than a 1kb block size

# File System Administration

- ▶ This argues for large block sizes
- ▶ Now we have a conflict, we want both large and small block sizes, but we can't have both
- ▶ This conflict is shown in the graph on the next slide
- ▶ A compromise could be the place where the two curves intersect, but this is a bad choice for both!
- ▶ The block size really depends upon how the file system will be used
- ▶ For program development a small block size is good, for video editing a large block size would be better

# File System Administration



# File System Administration

- ▶ The next thing to be considered is the number of Inodes
- ▶ We need one Inode per file, so this limits the number of files that we can have
- ▶ The mkfs program will estimate the number of Inodes based on the size of the partition and the block size, but you can adjust this value
- ▶ Some file systems allow you to specify the size of the Inodes, useful if you have a lot of large files
- ▶ Remember, Inodes are stored on disk, so they take up space that could be used for data

# File System Administration

- ▶ Some file systems allow you to set quotas, this is useful on multi-user systems and servers
- ▶ The quota is the number of files and the total number of disk blocks that a user can have
- ▶ This prevents a single user from hogging all the disk space
- ▶ On a server it prevents one application from filling the disk and crashing the system
- ▶ Most systems have both hard and soft quotas
- ▶ A user can exceed their soft quota, but they will be warned

# File System Administration

- ▶ There is a fixed number of warnings before the account is locked
- ▶ This allows for the creation of temporary files that are deleted at the end of a session
- ▶ If the user exceeds the hard quota the account is immediately locked
- ▶ To unlock the account the user must come up with a plan to decrease their disk usage
- ▶ Quotas can be placed on other resources, but disk is the most common one

# File System Administration

- ▶ Another administrator job is backups
- ▶ For single user systems this isn't as important, there are things like GitHub for saving versions of the things that you are working on
- ▶ In the case of servers and multi-user systems this is more important
- ▶ Some servers will have uptime guarantees if a disk crashes they need to get it up as fast as possible
- ▶ RAID configurations is one way of doing this, multiple disks where the same information is stored on several disks
- ▶ Problem: disks tend to fail at the same time, the recovery of a single disk causes extra stress on the others

# File System Administration

- ▶ There are two reasons for doing backups:
  - ▶ Recover from disk crash, or other disaster
  - ▶ Recover from stupidity – someone deleted a file by mistake
- ▶ Windows handles the second one nicely by placing the file in the recycle bin where it can be recovered
- ▶ The backup procedure depends upon which of the two reasons you are concerned about
- ▶ Historically backups have gone to tape, this is still a common practice since tapes are cheap and easy to store
- ▶ Backups to disk are now becoming more common

# File System Administration

- ▶ The lower price of disks make them more attractive and the backup will be quicker, NAS is commonly used for this
- ▶ Tape is about 20% or less than the cost of disk
- ▶ Tape is not as reliable as disk, and can only be written so many times before it starts to fail
- ▶ The simplest type of dump is a full dump (the whole disk), this basically copies the raw disk starting at block zero, called a **physical dump**
- ▶ This can be done using a program like dd, or a slightly more sophisticated program
- ▶ This is hard to get wrong, but people will

# File System Administration

- ▶ There are three issues with a physical dump
- ▶ First, it dumps the entire disk, no matter how full it is
- ▶ This is a waste of time and space
- ▶ Second, when restoring it must go to a disk of the same size
- ▶ The copy includes all the partition information from the MBR and all the file system information
- ▶ If this is copied to a larger disk, it will only use part of the disk, the rest will not be available
- ▶ Copying to a small disk just won't work

# File System Administration

- ▶ Third, the copy program could have problems with bad blocks
- ▶ No disk is perfect, there will always be places on the disk that cause errors, these are called **bad blocks**
- ▶ When a disk is manufactured extra space is allocated for blocks that will replace the bad blocks
- ▶ The disk drive itself will have a map of the bad blocks and will automatically replace them with good blocks, the OS doesn't know about this
- ▶ Unfortunately, as the disk ages more bad blocks will appear, these ones the OS will know about

# File System Administration

- ▶ Quite often the OS will gather them into a bad block file so they won't be used in a file
- ▶ If the dump program can't handle bad blocks it will crash, some programs make use of the bad block file for this, others ignore errors in reads
- ▶ The other type of dump is a **logical dump**, this dump starts at the root directory and follows all the directories copying out directories and files
- ▶ This has the advantage of dumping only data and not the unused blocks on the disk

# File System Administration

- ▶ Logical dumps are more complicated, but have more features
- ▶ One of the most important ones is the ability to do **incremental dumps**
- ▶ The idea is that a full dump of the file system is done periodically, say once a week
- ▶ For the other days only the files that have changed are dumped
- ▶ This results in a much smaller dump, that can be done more quickly
- ▶ Typically the incremental dumps are done at the end of the day
- ▶ Incremental dumps require some OS support

# File System Administration

- ▶ The dump program needs to know if a file has been changed since the last dump
- ▶ One approach is to add a flag to the Inode, this flag is set when the file is modified and cleared when the file is dumped
- ▶ Another approach is to know the date and time of the last dump, inspect the modified time in the Inode and dump the file if it occurred after the last dump
- ▶ One of the advantages of logical dumps is its possible to restore individual files, but this may involve examining multiple tapes

# File System Administration

- ▶ Doing a full disk restore is more complicated with logical dumps
- ▶ Must start with the most recent full dump and restore it
- ▶ Then all the incremental dumps must be restored in the order they were produced
- ▶ If a file is quite active it may appear on multiple incremental dumps
- ▶ This is much slower than a physical dump, maybe 10x or more slower
- ▶ This is why a full dump is done every week, or maybe more often
- ▶ This is the price you pay for more flexibility

# File System Administration

- ▶ What about file system activity while the dump is in progress?
- ▶ The file system cannot be active during a physical dump, this can be done by unmounting the file system
- ▶ Typically a maintenance period is schedule for this
- ▶ Since a physical dump is relatively fast, this may not be a major problem
- ▶ With some care a logical dump can be done while the file system is in use, in this case the dump won't be a snap shot of the file system
- ▶ Directories cannot be freed during this process or the dump program might get lost

# File System Administration

- ▶ The next question is where you put the disk or tape you just dumped to, this is important
- ▶ If the dump is to recover from disaster, say a fire, storing them next to the computer is of no use
- ▶ At least some of the dump media should be stored at a different location, this can be a security problem, theft of media
- ▶ If the dump is at a remote location, restoring files could take much longer
- ▶ Different schemes deal with this, for example a physical dump once a week stored at a remote location, at worst you will be 7 days out of date, not completely wiped out

# File System Administration

- ▶ Inode based file systems can become inconsistent, particularly when there is a system crash
- ▶ This is a problem for Linux and Windows NTFS file systems
- ▶ Both operating systems have programs that repair these problems, will examine the Linux one, called fsck
- ▶ This program always runs after a system crash while the system is booting
- ▶ It will occasionally run at boot time after a clean shutdown
- ▶ The file system must not be mounted, no programs can be running in the file system

# File System Administration

- ▶ On Linux there are conceptually two passes, these used to be separate programs
- ▶ The first pass works at the block level and is based on two tables of counters, with one entry for each Inode, both tables are initialized to zero:
  1. Counts the number of times the block appears in an Inode
  2. Counts the number of times the block appears in the free list
- ▶ This pass starts by examining all the Inodes in the file system, for each block mentioned in an Inode its entry in the first table is incremented
- ▶ A similar process happens with the free list, each time a block is found its counter in the second table is incremented

# File System Administration

- ▶ At the end of this pass each block should have 1 in either the first table or the second table, but not both
- ▶ If a block doesn't appear in either list it's a **missing block** and its added to the free list
- ▶ If a block appears more than once in the free list, the free list needs to be rebuilt
- ▶ If the counter in the first table is greater than 1, the block appears in more than one file, which is a serious problem
- ▶ If one of these files is removed, the block will be placed back on the free list, even though it is still used by the other file

# File System Administration

- ▶ The fix to this problem isn't perfect
- ▶ A new block is removed from the free list and the contents of the problem block are copied to this block and it replaces the problem block in one of the files
- ▶ This produces a consistent file system, but one or both of the files will now be garbled, hopefully there is a backup that can fix this
- ▶ The second pass is based on the directory structure, it also has a table with one entry per Inode, initialized to zero
- ▶ This pass starts at the root of the file system and visits every directory
- ▶ Each entry in the directory is examined and the entry for its Inode is incremented

# File System Administration

- ▶ The entries in this table can be greater than 1 if the Inode has multiple hard links
- ▶ If the entry is 0, the file isn't referenced by any directory, it is clearly lost, the file is placed in the lost and found directory, so it can be retrieved later by a user
- ▶ Next the count in the table is compared to the hard link count in the Inode
- ▶ If the counts are different the one in the Inode is changed to agree with the table

# Performance

- ▶ There are two things that a file system can do to attempt to improve performance
- ▶ One is to cache blocks, similar to what was done with pages in memory management, and similar algorithms can be used
- ▶ While there may be multiple partial reads from a block, or partial writes, it's unlikely that a block will be read multiple times
- ▶ This might occur with programs, but the memory management system has probably already locked those blocks in memory
- ▶ The techniques get quite complicated and its not clear that this is a big win, causes problems if there is a crash

# Performance

- ▶ The other technique is **block read ahead**, the idea here is that most files are read sequentially
- ▶ If there is a request for block  $k$ , a request for block  $k+1$  will soon follow, thus it makes sense to schedule a read for that block
- ▶ The hope is that when the read comes for the next block it is already in memory
- ▶ This works well for sequential file access, but is counter productive for random access
- ▶ The OS can keep track of the access pattern and switch to block read ahead when its clear that sequential access is being used

# Summary

- ▶ Examined file systems and the algorithms that are used with them:
  - ▶ How files are stored on disk
  - ▶ Directory structure
- ▶ Examine different types of file systems
- ▶ Examined some of the system administrator tasks associated with file systems
- ▶ Quickly looked at two techniques that attempt to improve performance



# CSCI 3310

# Network Programming

# Part One

MARK GREEN

ONTARIO TECH

# Introduction

- ▶ This is a big topic, will cover both the theory and practice, but will start with some programming
- ▶ The main purpose of networking is to communicate between different devices
- ▶ There is a wide range of networking technologies
- ▶ Start with the programmers view of the Internet
- ▶ This is based on the TCP/IP family of protocols
- ▶ IP is the base protocol, and TCP and UDP are built on top of it
- ▶ Start with TCP, view the network as a byte stream

# Network Addresses

- ▶ There are millions of computers on the Internet, we need some way of addressing them
- ▶ The initial widely distributed version of IP, IPv4, uses a 32 bit address to identify a computer
- ▶ A new version of IP, IPv6, uses a 128 bit address, both versions are in common use today
- ▶ To keep things simple we will concentrate on IPv4, the extension to IPv6 is fairly trivial from a programmers point of view

# Network Addresses

- ▶ A 32 bit IPv4 address is viewed as 4 bytes and is typically written as:  
w.x.y.z
- ▶ Where each of the numbers is between 0 and 255, called **dotted-decimal notation**
- ▶ This is the human readable version of it, the computer treats is as a 32 bit integer
- ▶ This gives us a way of addressing a computer, but we want to interact with a program running on that computer
- ▶ If we only had the address we could only interact with one program at a time, and there would be no way of determining which program

# Network Addresses

- ▶ This is handled by using **port numbers**, a 16 bit integer
- ▶ This integer can then be mapped to a program that is listening to that port
- ▶ To make connections easier there is a set of **well-known port numbers** that are associated with standard services
- ▶ This is maintained by **the Internet Assigned Numbers Authority (IANA)**, <https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml>
- ▶ You can find a local list in /etc/services

# /etc/services

```
mark@MSI: ~
# Network services, Internet style
#
# Note that it is presently the policy of IANA to assign a single well-known
# port number for both TCP and UDP; hence, officially ports have two entries
# even if the protocol doesn't support UDP operations.
#
# Updated from http://www.iana.org/assignments/port-numbers and other
# sources like http://www.freebsd.org/cgi/cvsweb.cgi/src/etc/services .
# New ports will be added on request if they have been officially assigned
# by IANA and used in the real-world or are needed by a debian package.
# If you need a huge list of used numbers please install the nmap package.

tcpmux      1/tcp          # TCP port service multiplexer
echo        7/tcp
echo        7/udp
discard    9/tcp          sink null
discard    9/udp          sink null
sysstat    11/tcp         users
daytime     13/tcp
daytime     13/udp
netstat    15/tcp
qotd       17/tcp          quote
msp         18/tcp          # message send protocol
msp         18/udp
chargen    19/tcp          ttyst source
chargen    19/udp          ttyst source
ftp-data   20/tcp
ftp        21/tcp
fsp        21/udp          fspd
--More-- (5%)
```

# Network Addresses

- ▶ Many of these port numbers have been assigned for many decades
- ▶ You cannot just randomly select a port number and used it for your application
- ▶ It is a bad idea to use a port number less than 2048, most OS's will not let you use a port number less than 1024 unless you have supervisor privileges
- ▶ This is to prevent rogue applications from spoofing well known applications, like web servers
- ▶ It is usually safe to use port numbers greater than 50,000
- ▶ This is what I normally do to avoid clashes with other applications

# Network Addresses

- ▶ When a connection is established between two computers, both sides with have an IP address and a port number
- ▶ On the client side the IP address and port number is typically assigned by the OS
- ▶ On the server side, the server will select the port number, but it usually lets the OS select an appropriate IP address
- ▶ All computers have at least 2 IP addresses, at least one of these addresses is not visible to the Internet
- ▶ Why? A hidden IP address can be used by local applications and not the rest of the world

# Network Addresses

- ▶ The following structure is used to store connection information, from man 7 ip:

```
struct sockaddr_in {  
    sa_family_t    sin_family; /* address family: AF_INET */  
    in_port_t      sin_port;   /* port in network byte order */  
    struct in_addr sin_addr;  /* internet address */  
};  
  
/* Internet address. */  
struct in_addr {  
    uint32_t        s_addr;    /* address in network byte order */  
};
```

- ▶ The following include files are required:

```
#include <sys/socket.h>
```

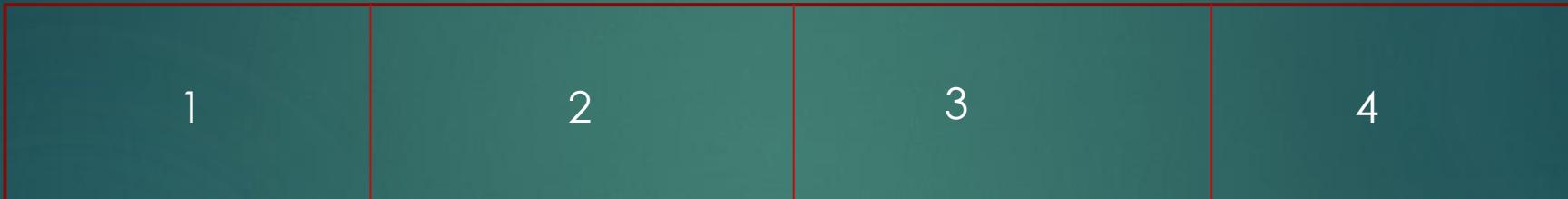
```
#include <netinet/in.h>
```

# Network Addresses

- ▶ The sin\_family will be AF\_INET for IPv4
- ▶ Now we get into a bit of a nasty detail
- ▶ Memory in modern computer is byte addressable, there is no problem with accessing a byte
- ▶ But how do we interpret a 4 byte integer?
- ▶ There are two common ways of doing this called **big-endian** and **little endian**
- ▶ On big-endian computers the most significant byte has the lowest address, on little-endian computers it has the higher address

# Endian

Big Endian



Little Endian



# Network Addresses

- ▶ If all computers were the same endian everything would be okay, but the world doesn't act that way
- ▶ The x86 architecture is little-endian, but other computers are big-endian
- ▶ Most of the computers at the time when the Internet was developed were big-endian, so this became the **network byte order**
- ▶ This is a standard that you must follow, otherwise you will not be able to connect to servers
- ▶ The port numbers and IP address must at least follow this standard

# Network Addresses

- ▶ Luckily there is a set of functions that come to the rescue:

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

- ▶ You should always use these functions regardless of the endian of your computer, this makes your program code more portable

# Mapping Names to Addresses

- ▶ Most people can't remember 32 IPv4 addresses, little on 128 bit IPv6 addresses
- ▶ To solve this problem most computers on the Internet have a human readable name
- ▶ The **Domain Name Service (DNS)** is a distributed database that is used to map these names into IP addresses
- ▶ Most computer also have an /etc/hosts file that is searched first before the DNS is used
- ▶ This is used to maintain computer names on a local network

# Mapping Names to Addresses

- ▶ The getaddrinfo function can be used to map host names and services names to IP addresses and port numbers:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *node, const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);

void freeaddrinfo(struct addrinfo *res);

const char *gai_strerror(int errcode);
```

- ▶ This is a bit complicated and doesn't follow the standard Linux error reporting process

# Mapping Names to Addresses

- ▶ The first two parameters are fairly easy, they are the host name and service that is required as text strings
- ▶ Either of these parameters could be NULL – but not both, which saves looking up that name
- ▶ The following structure is used for both the third and fourth parameters

```
struct addrinfo {  
    int          ai_flags;  
    int          ai_family;  
    int          ai_socktype;  
    int          ai_protocol;  
    socklen_t    ai_addrlen;  
    struct sockaddr *ai_addr;  
    char         *ai_canonname;  
    struct addrinfo *ai_next;  
};
```

# Mapping Names to Addresses

- ▶ The hints parameter is used to control the amount of information that is returned
- ▶ The ai\_family field can be PF\_INET or PF\_INET6 to restrict the type of addresses returned
- ▶ The ai\_socktype field can be either SOCK\_STREAM or SOCK\_DGRAM, we are only interested in the first for now
- ▶ The ai\_protocol field is usually 0, unless special processing is required
- ▶ The ai\_flags field further controls the information returned, it is the OR of the flags on the next slide

# Mapping Names to Addresses

- ▶ AI\_ADDRCONFIG – returns only addresses that can be used with the current system configuration, for example IPv6 address won't be returned for a system that only supports IPv4
- ▶ AI\_CANONNAME – the ai\_canonname field will contain the canonical name for the host, this is the name that the host is normally known by
- ▶ AI\_NUMERICHOST – the host name is already in dotted-decimal form, no host name lookup is required, the text string is just converted into the appropriate integer
- ▶ AI\_PASSIVE – if the host name is NULL, return an address that can be used for all interfaces on the system – a wild card address

# Mapping Names to Addresses

- ▶ The last parameter is a pointer to a pointer to a list of addrinfo structures
- ▶ In your program you declare a pointer to an addrinfo structure, then you pass a pointer to this variable by putting an & in front of it in the getaddrinfo call
- ▶ You will get a list of addresses back, usually the first address on the list is the one that you want to use

# Mapping Names to Addresses

- ▶ On success getaddrinfo returns 0, otherwise it returns an error code, it doesn't use errno like other Linux functions
- ▶ An error message can be retrieved by passing the returned value to gai\_strerror()
- ▶ When we are finished we need to free the memory used by the linked list of addresses
- ▶ The freeaddrinfo() procedure can be used for this
- ▶ Now that we have addresses under control, we can look at setting up a connection between two computers

# Sockets

- ▶ The socket API is used for all Unix/Linux networking, all network connections are sockets
- ▶ There is also a version of sockets on Windows, works in basically the same way
- ▶ With a bit of care you can write socket code that runs on both Linux and Windows
- ▶ The basic architecture is you have a server process and a client process
- ▶ The client will connect to the server to exchange information, use a service, etc.

# Sockets

- ▶ The server will advertise a service in terms of an IP address and a port number
- ▶ The client will connect to this service using the IP address and port number of the server
- ▶ The API calls that the server uses are different from the ones that the client uses
- ▶ They all start with a socket
- ▶ A socket is a file descriptor with read and write privileges
- ▶ That is, we can use our standard read and write system calls, and they go both directions

# Sockets

- ▶ The socket system call:  
`int socket(int domain, int type, int protocol);`
- ▶ There are a number of possible domains, the two that we are interested in are:
  - ▶ PF\_INET – TCP/IP version 4
  - ▶ PF\_INET6 – TCP/IP version 6
- ▶ The two standard values for type are:
  - ▶ SOCK\_STREAM – a stream of bytes
  - ▶ SOCK\_DGRAM – packets of data

# Sockets

- ▶ The protocol parameter is used in special cases, the value 0 is typically used
- ▶ The next thing the client does is specify the server that it wants to connect to, this is done with the connect system call:

```
int connect(int socket, struct sockaddr *server, socklen_t addrlen);
```

- ▶ The first parameter is the socket we just created
- ▶ The second parameter is the server address returned by getaddrinfo, and the third parameter is the length of this structure
- ▶ Note: the address structures for IPv4 and IPv6 are of different lengths

# Sockets

- ▶ Connect() blocks until the server has accepted the connection, at that point the socket can be used to read and write
- ▶ On the server side, the first thing to do is specify the server's IP address and port, this is done with the bind system call:

```
int bind(int socket, struct sockaddr *myAddr, socklen_t addrlen);
```
- ▶ The first parameter is the socket, the second parameter is the address that the server uses, and the third parameter is the size of this address structure
- ▶ The parameters are basically the same as connect

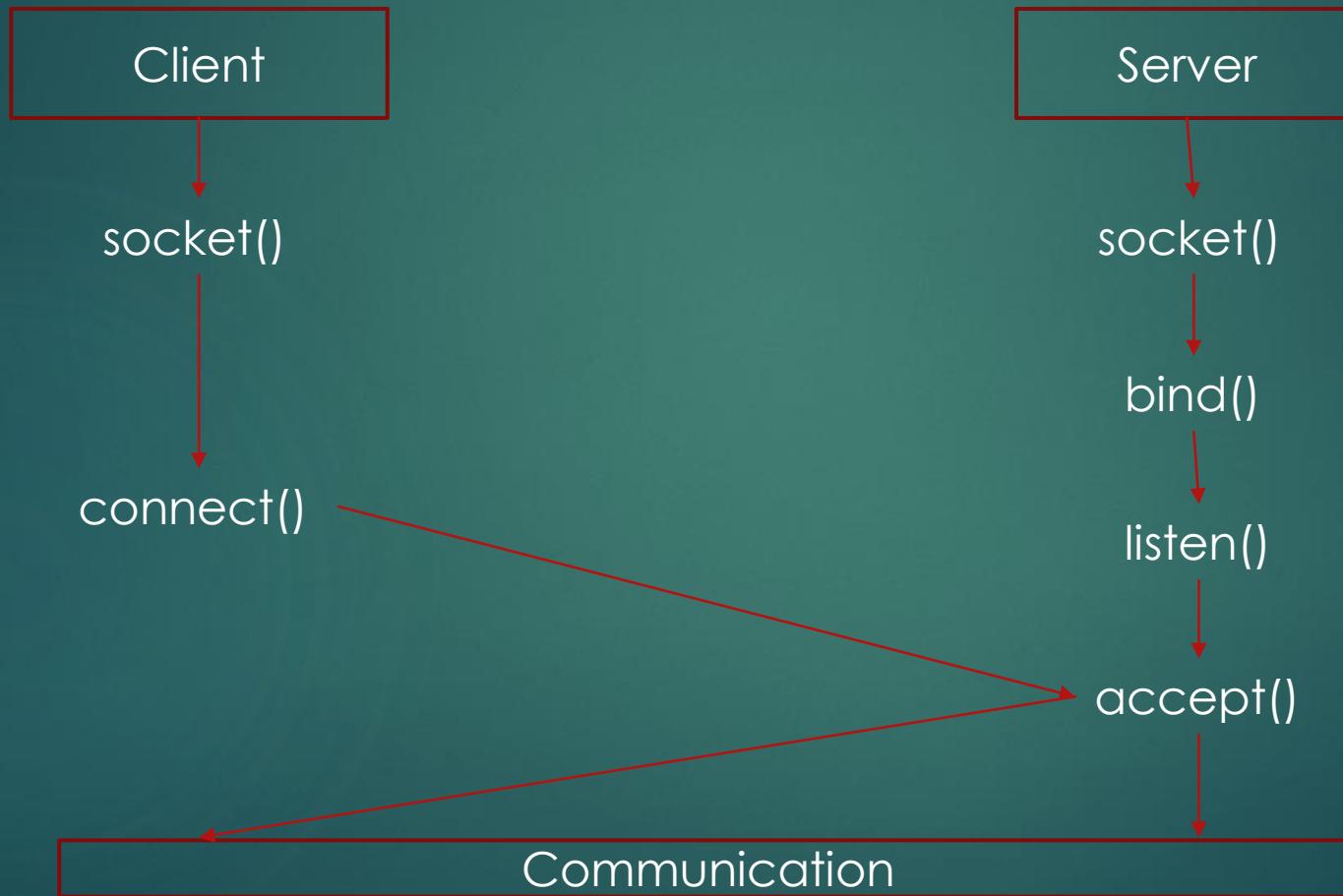
# Sockets

- ▶ The server has two more steps
- ▶ First, it needs to listen for connections from a client, this is done with `listen`:  
`int listen(int socket, int backlog);`
- ▶ The first parameter is the socket for the service, the second parameter specifies the number of clients that can be waiting for service
- ▶ Several connection requests could arrive at close to the same time, it takes the server some time to process each one
- ▶ This parameter is typically set to 5

# Sockets

- ▶ The second step is accepting the connect, this is done with accept:  
`int accept(int socket, struct sockaddr *addr, socklen_t addrlen);`
- ▶ The first parameter is the socket we are listening on
- ▶ The second and third parameters will receive the IP address and port number of the client
- ▶ Accept blocks until a connection request is received from the client
- ▶ It can then returns a new socket that can be used to read and write to the client
- ▶ The whole process is shown on the next slide

# Sockets

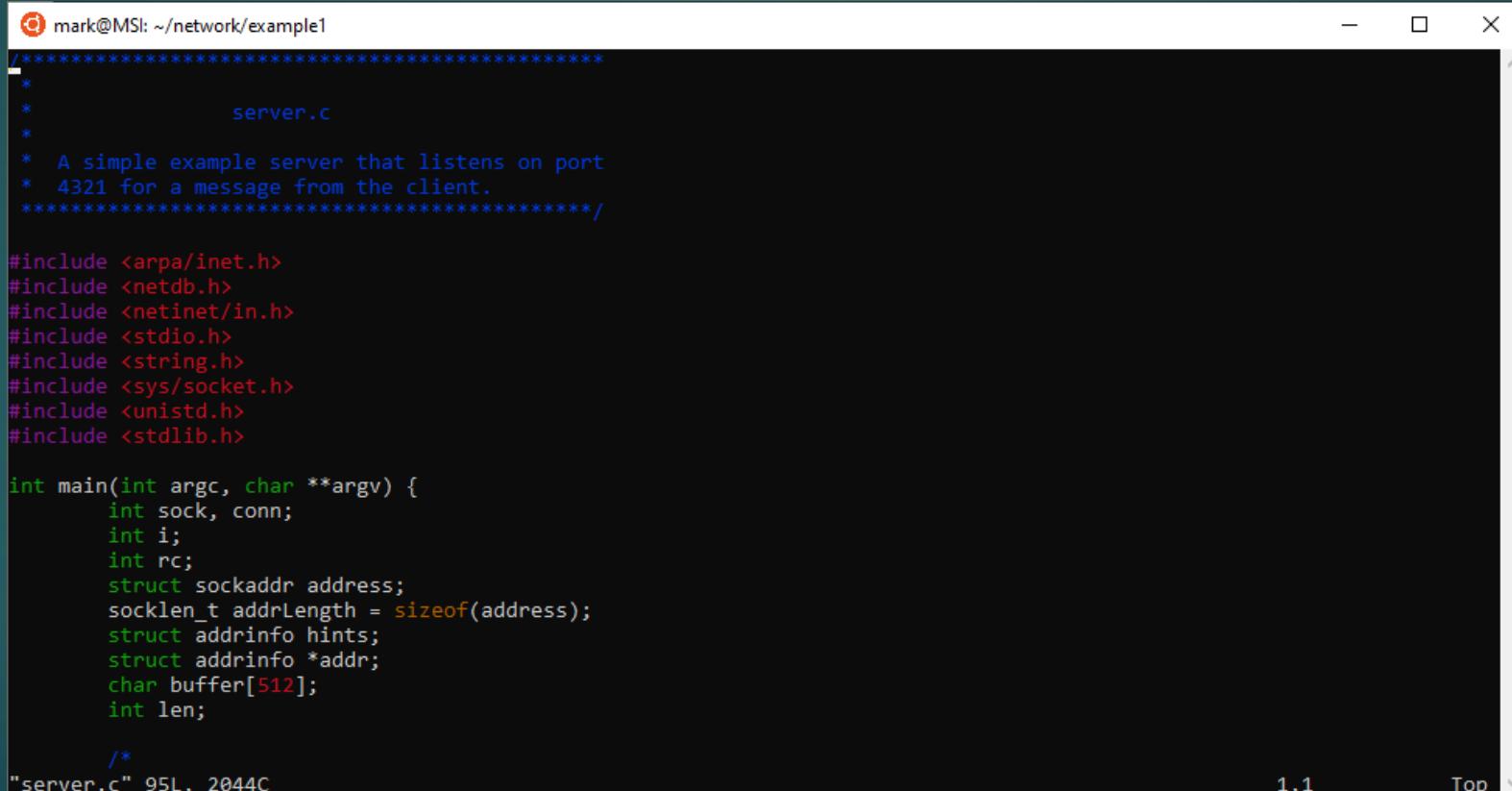


# Example

- ▶ Start by looking at a simple example of a client and a server using sockets
- ▶ This example mainly shows how sockets are set up, the application doesn't do much of interest, it just exchanges messages between the two programs
- ▶ In this example both the client and server run on the same computer
- ▶ The name localhost is used for the local computer, this is the host name that the client program will use to find the server

# Example - Server

- ▶ Start with the server code, include files and variable declarations:



A screenshot of a terminal window titled "mark@MSI: ~/network/example1". The window displays the beginning of a C program named "server.c". The code includes comments explaining it's a simple example server that listens on port 4321 for a message from the client. It then lists various header files required for socket programming. The main function starts with variable declarations for a socket, connection, indices, return codes, and structures. A buffer for receiving data is also declared. The code ends with a copyright notice and the file name "server.c".

```
mark@MSI: ~/network/example1
/**
 *          server.c
 *
 *  A simple example server that listens on port
 *  4321 for a message from the client.
 *****/
#include <arpa/inet.h>
#include <netdb.h>
#include <netinet/in.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    int sock, conn;
    int i;
    int rc;
    struct sockaddr address;
    socklen_t addrLength = sizeof(address);
    struct addrinfo hints;
    struct addrinfo *addr;
    char buffer[512];
    int len;

    /*
 "server.c" 95L, 2044C

```

# Example - Server

- ▶ The first thing we need to do is obtain the network address for the server, this is done with getaddrinfo:

```
/*
 *  set the hints structure to zero
 */
memset(&hints, 0, sizeof(hints));

/*
 *  want a stream, also address that will accept all
 *  connections on this host
 */
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE | AI_ADDRCONFIG;
if((rc = getaddrinfo(NULL, "4321", &hints, &addr))) {
    printf("host name lookup failed: %s\n", gai_strerror(rc));
    exit(1);
}
```

# Example - Server

- ▶ The memset procedure is used to initialize a block of memory to a constant value
- ▶ In this case we are setting the contents of hints to zero
- ▶ The three parameters to this function are the pointer to the memory block, the constant value to use and number of bytes to initialize
- ▶ For the hints we want a TCP stream, SOCK\_STREAM
- ▶ We also want to use all the addresses on the host that are supported by network hardware
- ▶ We are essentially requesting a wild card address for this host

# Example - Server

- ▶ The next bit of code creates the socket and binds it to the address, all this information is provided by getaddrinfo

```
/*
 *  use the first entry returned by getaddrinfo
 */
sock = socket(addr->ai_family, addr->ai_socktype, addr->ai_protocol);
if(sock < 0) {
    printf("Can't create socket\n");
    exit(1);
}

/*
 *  want to be able to reuse the address right after
 *  the socket is closed. Otherwise must wait for 2 minutes
 */
i = 1;
setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &i, sizeof(i));

rc = bind(sock, addr->ai_addr, addr->ai_addrlen);
if(rc < 0) {
    printf("Can't bind socket\n");
    exit(1);
}
```

# Example - Server

- ▶ The setsockopt procedure is used to control how sockets behave
- ▶ There are a large number of options to this procedure, we only need one of them
- ▶ The first parameter is the socket, and the second parameter is the level where the option is
- ▶ In this case we use SOL\_SOCKET, since we are setting a general socket option
- ▶ The third parameter is the name of the option
- ▶ The fourth parameter is a pointer to the option value, and the fifth parameter is the length, in bytes, of the value

# Example - Server

- ▶ In this case we are setting the SO\_REUSEADDR option
- ▶ Normally the OS doesn't release the address and port when a server terminates, instead it waits several minutes to release them
- ▶ This gives clients an opportunity to terminate the connection cleanly
- ▶ This is a good idea for production servers, but causes problems for debugging
- ▶ If we quickly change the server and then try to run it again, we will find that we can't get the port – it is still held by the OS
- ▶ By setting this option the OS will immediately release the address and port

# Example - Server

- ▶ The next bit of code frees the result that we returned by getaddrinfo and then starts listening on the server's port

```
/*
 * free results returned by getaddrinfo
 */
freeaddrinfo(addr);

rc = listen(sock, 5);
if(rc < 0) {
    printf("Listen failed\n");
    exit(1);
}
```

# Example - Server

- ▶ We are now ready to accept connections from the client and process them
- ▶ This is basically an infinite loop waiting for connections

```
/*
 *  accept an arbitrary number of connections in a loop
 */
while((conn = accept(sock, (struct sockaddr*) &address, &addrLength))
    >= 0) {
    /*
     *  read message from client and respond
     */
    len = read(conn, buffer, 512);
    printf("Received from client: %s\n",buffer);
    strcpy(buffer,"Hello Client");
    write(conn, buffer, strlen(buffer));
    close(conn);
}
```

# Example

- ▶ Servers are designed to run continuously until they are explicitly killed by the server owner or administrator, so we use an infinite loop
- ▶ Note how we close the socket to the client after we send the message, this terminates the connection between the two processes
- ▶ The client code is simpler than the server code
- ▶ It starts with a similar set of include files and variable declarations
- ▶ The code is on Canvas, you can examine it and run it yourself

# Example - Client

- ▶ The client starts with a similar call to getaddrinfo, in this case to get the address of the server

```
/*
 *  clear the hints structure to zero
 */
memset(&hints, 0, sizeof(hints));

/*
 *  want a stream on a compatible interface
 */
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_ADDRCONFIG;
/*
 *  localhost is the name of the current computer
 */
rc = getaddrinfo("localhost", NULL, &hints, &addr);
if(rc != 0) {
    printf("Host name lookup failed: %s\n", gai_strerror(rc));
    exit(1);
}
...
```

# Example - Client

- ▶ Next create the socket and connect to the server, note how htons is user for the server port

```
/*
 *  use the first result from getaddrinfo
 */
addrinfo = (struct sockaddr_in *) addr->ai_addr;

sock = socket(addrinfo->sin_family, addr->ai_socktype, addr->ai_protocol);
if(sock < 0) {
    printf("Can't create socket\n");
    exit(1);
}

/*
 *  specify the port number
 */
addrinfo->sin_port = htons(4321);

rc = connect(sock, (struct sockaddr *) addrinfo, addr->ai_addrlen);
if(rc != 0) {
    printf("Can't connect to server\n");
    exit(1);
}
```

# Example - Client

- ▶ The final piece frees the getaddrinfo result and interacts with the server

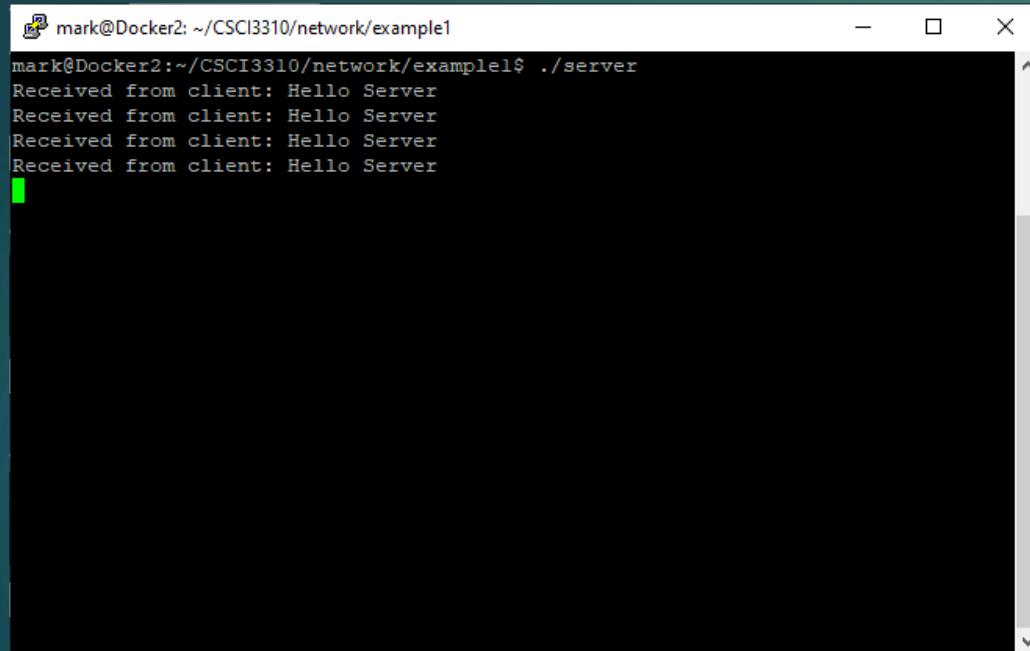
```
/*
 *  free the results returned by getaddrinfo
 */
freeaddrinfo(addr);

/*
 *  send a message to the server and echo the response
 */
strcpy(buffer, "Hello Server");
write(sock, buffer, strlen(buffer));
len = read(sock, buffer, 512);
printf("Message from server: %s\n",buffer);

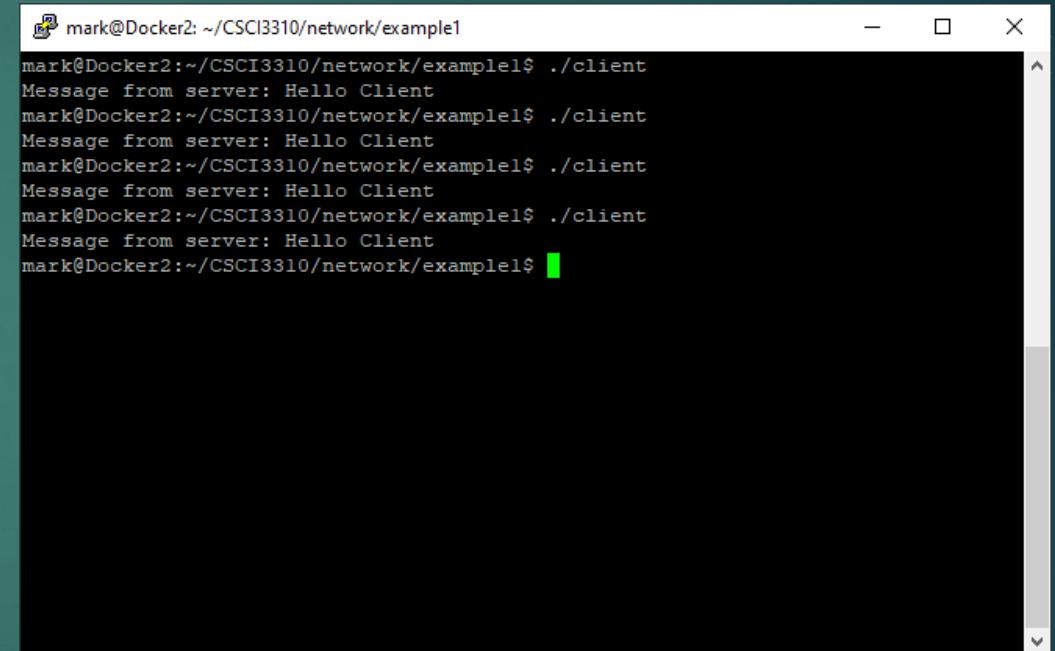
close(sock);

exit(0);
```

# Example



```
mark@Docker2: ~/CSCI3310/network/example1
mark@Docker2:~/CSCI3310/network/example1$ ./server
Received from client: Hello Server
```



```
mark@Docker2: ~/CSCI3310/network/example1
mark@Docker2:~/CSCI3310/network/example1$ ./client
Message from server: Hello Client
mark@Docker2:~/CSCI3310/network/example1$
```

# Debugging

- ▶ To debug network programs you quite often need to know what the network is doing
- ▶ If your client or server can't make a connection, the problem might originate in the network and your program needs to work around it
- ▶ There are a number of programs that assist with this
- ▶ The first of these is ifconfig, it tells us which network interfaces we have on our computer and some information about each interface
- ▶ An example of ifconfig output is shown on the next slide
- ▶ The equivalent program on Windows is ipconfig

# Debugging

```
mark@Docker2: ~/CSCI3310/network/example1
    RX packets:0 errors:0 dropped:0 overruns:0 frame:0
    TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
    collisions:0 txqueuelen:0
    RX bytes:0 (0.0 B)   TX bytes:0 (0.0 B)

enp0s7      Link encap:Ethernet  HWaddr 00:30:67:71:20:3b
            inet  addr:192.168.0.155  Bcast:192.168.0.255  Mask:255.255.255.0
            inet6 addr: fe80::134:767f:1b39:650b/64 Scope:Link
            UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
            RX packets:815174 errors:0 dropped:0 overruns:0 frame:0
            TX packets:182419 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:1000
            RX bytes:597853334 (597.8 MB)   TX bytes:19026909 (19.0 MB)

lo         Link encap:Local Loopback
            inet  addr:127.0.0.1  Mask:255.0.0.0
            inet6 addr: ::1/128 Scope:Host
            UP LOOPBACK RUNNING  MTU:65536  Metric:1
            RX packets:2052 errors:0 dropped:0 overruns:0 frame:0
            TX packets:2052 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:1000
            RX bytes:279687 (279.6 KB)   TX bytes:279687 (279.6 KB)

mark@Docker2:~/CSCI3310/network/example1$
```

# Debugging

- ▶ There are two interface shown here
- ▶ The first one enp0s7 is an ethernet interface
- ▶ We can see its IPv4 address and IPv6 address along with statistics on the number of packets and bytes that have passed through the interface
- ▶ The second interface is the loopback interface, this the interface with host name localhost, it always has the IPv4 address of 127.0.0.1
- ▶ It also tells us that each interface is up and running
- ▶ This is the easiest way to determine the address that has been assignment to your computer

# Debugging

- ▶ The next useful program is ping, on both Linux and Windows
- ▶ This program determines if a given host can be reached, and if it can how long it takes for packets to get there and back
- ▶ This program has many options, the only required one is the name of the host to contact
- ▶ An example of ping output is shown on the next slide
- ▶ If your program can't connect to another computer, try using ping to see if that computer is reachable

# Debugging

```
mark@Docker2: ~/CSCI3310/network/example1
    inet6 addr: ::1/128 Scope:Host
      UP LOOPBACK RUNNING  MTU:65536  Metric:1
        RX packets:2052 errors:0 dropped:0 overruns:0 frame:0
        TX packets:2052 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
        RX bytes:279687 (279.6 KB)  TX bytes:279687 (279.6 KB)

mark@Docker2:~/CSCI3310/network/example1$ ping www.uoit.ca
PING atlas-c.uoit.net (205.211.180.33) 56(84) bytes of data.
64 bytes from atlas-c.uoit.net (205.211.180.33): icmp_seq=1 ttl=59 time=2.60 ms
64 bytes from atlas-c.uoit.net (205.211.180.33): icmp_seq=2 ttl=59 time=2.42 ms
64 bytes from atlas-c.uoit.net (205.211.180.33): icmp_seq=3 ttl=59 time=1.58 ms
64 bytes from atlas-c.uoit.net (205.211.180.33): icmp_seq=4 ttl=59 time=2.40 ms
64 bytes from atlas-c.uoit.net (205.211.180.33): icmp_seq=5 ttl=59 time=3.15 ms
64 bytes from atlas-c.uoit.net (205.211.180.33): icmp_seq=6 ttl=59 time=2.61 ms
64 bytes from atlas-c.uoit.net (205.211.180.33): icmp_seq=7 ttl=59 time=2.63 ms
64 bytes from atlas-c.uoit.net (205.211.180.33): icmp_seq=8 ttl=59 time=2.38 ms
64 bytes from atlas-c.uoit.net (205.211.180.33): icmp_seq=9 ttl=59 time=2.13 ms
64 bytes from atlas-c.uoit.net (205.211.180.33): icmp_seq=10 ttl=59 time=2.38 ms
^C
--- atlas-c.uoit.net ping statistics ---
11 packets transmitted, 10 received, 9% packet loss, time 10014ms
rtt min/avg/max/mdev = 1.589/2.432/3.152/0.382 ms
mark@Docker2:~/CSCI3310/network/example1$
```

# Debugging

- ▶ The third program is netstat, on both Linux and Windows
- ▶ This is quite a general program that can tell us a lot of things
- ▶ At this point just look at one thing it can do
- ▶ Netstat can tell us the connections that are currently active and servers that are listening for connections
- ▶ The next slide shows a typical netstat usage, in this case we are looking at IPv4 connections (--inet) and servers that are listening (-l)
- ▶ We can see that our server is listening on port 4321

# Debugging

```
mark@Docker2: ~/CSCI3310/network/example1
64 bytes from atlas-c.uoit.net (205.211.180.33): icmp_seq=5 ttl=59 time=3.15 ms
64 bytes from atlas-c.uoit.net (205.211.180.33): icmp_seq=6 ttl=59 time=2.61 ms
64 bytes from atlas-c.uoit.net (205.211.180.33): icmp_seq=7 ttl=59 time=2.63 ms
64 bytes from atlas-c.uoit.net (205.211.180.33): icmp_seq=8 ttl=59 time=2.38 ms
64 bytes from atlas-c.uoit.net (205.211.180.33): icmp_seq=9 ttl=59 time=2.13 ms
64 bytes from atlas-c.uoit.net (205.211.180.33): icmp_seq=10 ttl=59 time=2.38 ms
^C
--- atlas-c.uoit.net ping statistics ---
11 packets transmitted, 10 received, 9% packet loss, time 10014ms
rtt min/avg/max/mdev = 1.589/2.432/3.152/0.382 ms
mark@Docker2:~/CSCI3310/network/example1$ netstat --inet -l
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address          Foreign Address        State
tcp      0      0 Docker2:domain          *:*                  LISTEN
tcp      0      0 *:ssh                 *:*                  LISTEN
tcp      0      0 localhost:ipp           *:*                  LISTEN
tcp      0      0 *:4321                *:*                  LISTEN
udp      0      0 *:mdns                *:*                  LISTEN
udp      0      0 *:38687               *:*                  LISTEN
udp      0      0 *:51205               *:*                  LISTEN
udp      0      0 Docker2:domain          *:*                  LISTEN
udp      0      0 *:bootpc              *:*                  LISTEN
udp      0      0 *:ipp                 *:*                  LISTEN
mark@Docker2:~/CSCI3310/network/example1$
```

# Summary

- ▶ Examined network addressing, how to lookup the address of a host
- ▶ Standard client server architecture
- ▶ Introduced sockets and looked at a simple example
- ▶ This isn't the way that we normally build a server, need to examine server architecture next
- ▶ Examined some of the tools that can assist with debugging network programs



# CSCI 3310

# Network Programming

# Part Two

MARK GREEN

ONTARIO TECH

# Introduction

- ▶ Covered the very basics of getting a client and server running
- ▶ There is a lot more to do in terms of reliability and performance
- ▶ The current version of the server can only handle one client at a time
- ▶ It finishes processing the current client before accepting the next one
- ▶ The only works in the simplest cases, where the processing is very short
- ▶ In addition, we have assumed that everything in the network is perfect, how do we recover from errors?

# Server Architecture

- ▶ Start by examining the first problem, efficiently handling requests in the server
- ▶ Client processing can be time consuming, it can even involve blocking system calls
- ▶ With our current approach nothing is accomplished while the server is blocked processing a client request
- ▶ This isn't the most efficient way of handling things
- ▶ We want the server to do multiple things at the same time
- ▶ We already know two ways of doing this, and there is a third way

# Server Architecture

- ▶ One approach is to use the fork() system call to create a child process to handle the client request
- ▶ Recall that accept returns a new socket for interacting with the client, we don't use the socket that we originally created
- ▶ Each request has a new socket, so we can be processing several of them at the same time
- ▶ A child process will inherit the socket returned by accept, so it can easily handle the request
- ▶ This is outlined on the following slide

# Server Architecture

```
int sock, conn;  
...  
listen(sock, 5);  
While(TRUE) {  
    conn = accept(sock, ...);  
    if((childpid = fork()) == 0) {  
        close(sock);           // child process  
        client processing  
        exit(0);  
    } else {  
        close(conn);  
    }  
}
```

# Server Architecture

- ▶ With this structure the server can be serving multiple clients at the same time
- ▶ On the parent side the fork() returns almost immediately, so it is ready to accept the next client request
- ▶ But, there is a problem here
- ▶ When the child is finished, it calls exit(), it still sits in the process table until the parent retrieves its exit status
- ▶ We call these processes zombies, they aren't alive, but they are not quite dead
- ▶ We know we can use wait() and waitpid() to get the process status, but we don't know when to call them

# Signals

- ▶ Signals can be used for this purpose
- ▶ A signal is like an interrupt, but it is at the software level
- ▶ When a process receives a signal, it interrupts the current program code, and control can be transferred to another procedure
- ▶ This is a general mechanism that can be used for multiple conditions and has been part of Unix since the very beginning
- ▶ One of the signals that our process can receive occurs when a child process calls the exit() system call
- ▶ This is exactly what we want

# Signals

- ▶ Unfortunately signals are quite complicated and in the past they have not been implemented correctly, or at least some details have been ignored
- ▶ Part of the problem is they introduce concurrency where we don't expect it
- ▶ The other problem is that multiple signals that happen close together are not handled in the way we would expect
- ▶ Examine both of these problems as we explore the current implementation of signals

# Signals

- ▶ There are several things that can cause a signal:
  - ▶ The process itself can raise the signal
  - ▶ Another process can raise the signal
  - ▶ The kernel can generate the signal
- ▶ Once this happens the signal enters the pending state
- ▶ There are three things that can happen:
  - ▶ The signal can be ignored
  - ▶ A signal handler can be called to process the signal
  - ▶ The default action, depends on signal, defined by OS

# Signals

- ▶ If a signal handler is called we say the signal has been caught
- ▶ There are some signals that we can't ignore or catch
- ▶ In these cases the OS terminates the program
- ▶ The `sigaction()` system call is used to specify what happens when a signal occurs:

```
int sigaction(int signum, struct sigaction *sig, struct sigaction *old);
```

- ▶ This system call and related data structures are declared in `signal.h`
- ▶ The first parameter to this system call is the signal that we want to handle, each signal has a defined constant that should be used here

# Signal

- ▶ The sigaction structure specifies what happens when the signal occurs:

```
struct sigaction {  
    __sighandler_t sa_handler;  
    sigset_t sa_mask  
    int sa_flags;  
};
```

- ▶ The signal handler procedure takes a single integer parameter and returns void

# Signals

- ▶ There are two special values that can be used for `sa_handler`:
  - ▶ `SIG_IGN` – ignore the signal
  - ▶ `SIG_DFL` – use the default action
- ▶ There are two `sa_flags` values of interest:
  - ▶ `SA_NOCLDSTOP` – don't send a `SIGCHLD` signal when a child process is stopped, only when it terminates
  - ▶ `SA_RESTART` – if a slow system call is interrupted, restart the system call after the signal has been processed
- ▶ The `sa_mask` field specified the signals that will be blocked during the signal handler

# Signals

- ▶ The sa\_mask field is a sigset\_t, a data structure that represents a set of signals
- ▶ Some of the procedure that can be used to manipulate this data structure are:

|                                           |                                  |
|-------------------------------------------|----------------------------------|
| int sigemptyset(sigset_t *set);           | - clear the set                  |
| int sigfillset(sigset_t *set) ;           | - add all the signals to the set |
| int sigaddset(sigset_t *set, int signum); | - add signum to set              |
| Int sigdelset(sigset_t *set, int signum); | - delete signum from set         |

# Signals

- ▶ There are two procedures that can be used to generate signals within programs
- ▶ The raise() procedure is used to send a signal to the current process:  
`int raise(int signum);`
- ▶ The kill procedure can be used to send a signal to another process:  
`int kill(pid_t pid, int signum);`
- ▶ For regular users they must be the owner of the process that they send a signal to

# Signals

- ▶ You need to be careful when writing signal handlers
- ▶ The interrupt could occur in the middle of a C statement, or in a system call
- ▶ This can cause problems with global data and even procedures like `printf()`
- ▶ Since a signal can arrive at any time we need to guard against these problem in our program
- ▶ One way of doing this is to use a signal mask, the signals in the signal mask will be blocked

# Signals

- ▶ The `sigprocmask()` procedure is used to manipulate the signal mask:  
`int sigprocmask(int what, const sigset_t *set, sigset_t *old);`
- ▶ The `what` parameter can have the following values:
  - ▶ `SIG_BLOCK` – add the signals in `set` to the signal mask
  - ▶ `SIG_UNBLOCK` – remove the signals in `set` from the signal mask
  - ▶ `SIG_SETMASK` – set the signal mask to `set`
- ▶ The `set` parameter is the signals to be set or unset, and the `old` parameter is the previous value of the signal mask

# Signals

- ▶ To guard part of our code construct a signal set with the signals that could interrupt it
- ▶ Before the code call `sigprocmask()` with `SIG_BLOCK`
- ▶ After the code call `sigprocmask()` with `SIG_UNBLOCK`
- ▶ This essentially converts the code block into a critical region for signals
- ▶ There is one last problem, if a sequence of signals with the same signal number is received, they are converted into one pending signal
- ▶ This is important for our server

# Server Architecture

- ▶ Now we can return to our server
- ▶ The SIGCHLD signal is sent when a child is terminated
- ▶ We can catch this signal and then use something like wait() to completely terminate the zombie process
- ▶ The problem is we can't use wait(), since we need to handle the case where several child processes terminate at the same time
- ▶ We can use waitpid() to solve this problem:

```
pid_t waitpid(pid_t pid, int status, int options);
```
- ▶ If the pid parameter is -1, this procedure waits for any child to terminate

# Server Architecture

- ▶ If we use the WNOHANG option, waitpid() will return immediately if no process has terminated, if no processes have terminated, waitpid() returns zero
- ▶ With this in mind our signal handler is:

```
void handler(int signum) {  
    pid_t pid;  
    int status;  
  
    while(((pid = waitpid(-1, &status, WNOHANG)) > 0);  
        return;  
    }
```

# Server Architecture

- ▶ Now we've killed our zombies, but we now have a new problem
- ▶ What happens if the interrupt occurs in a system call like `read()` or `accept()`?
- ▶ We could hope that `SA_RESTART` was specified when the signal handler was established, but what if someone forgot?
- ▶ In this case `EINTR` will be returned as the value of `errno`
- ▶ We can check for this value and then restart the system call if this is the case
- ▶ We will first examine `accept()` and then look at `read` and `write`

# Server Architecture

- ▶ We need to change the loop in our server to:

```
While(TRUE) {  
    conn = accept(sock, ...);  
    if(conn < 0)  
        if(errno == EINTR)  
            continue;  
        else  
            print error message  
    if((childpid = fork()) == 0) {  
        ...  
    } else {  
        close(conn);  
    }  
}
```

# Read() and Write()

- ▶ The situation with read() and write() is more complicated
- ▶ For a robust application we need to be careful with how we handle these system calls
- ▶ The following slide shows the start of the man page for read()
- ▶ Note that it says it will attempt to read the number of bytes requested, but there is no guarantee
- ▶ It can return fewer bytes than we requested, this can lead to problems in our program

# Read() and Write()

```
mark@MSI: ~/network/example1
```

READ(2) Linux Programmer's Manual READ(2)

**NAME**  
read - read from a file descriptor

**SYNOPSIS**

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

**DESCRIPTION**  
read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf. ←

On files that support seeking, the read operation commences at the current file offset, and the file offset is incremented by the number of bytes read. If the current file offset is at or past the end of file, no bytes are read, and read() returns zero.

If count is zero, read() may detect the errors described below. In the absence of any errors, or if read() does not check for errors, a read() with a count of 0 returns zero and has no other effects.

If count is greater than SSIZE\_MAX, the result is unspecified.

**RETURN VALUE**  
On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or because read() was interrupted by a signal. See also NOTES.

Manual page read(2) line 1 (press h for help or q to quit)

Important

# Read() and Write()

- ▶ Consider the following piece of code:

```
int i;  
read(fd, &i, sizeof(i));
```

- ▶ We could use this code to read a binary integer
- ▶ If we were reading from disk there would be no problem
- ▶ But, if we are reading from a network there is no guarantee that we will get all sizeof(i) bytes in one read
- ▶ The OS doesn't know that we are trying to read an integer, it just knows about bytes, it will return what it has available

# Read() and Write()

- ▶ On top of this there are errors that we could get, read() returns -1, that really aren't errors
- ▶ We have already seen EINTR, when the call is interrupted
- ▶ We could also get EAGAIN and EWOULDBLOCK depending upon how we set up the socket, we will see this later
- ▶ All of these are a source of interesting bugs in our program
- ▶ It will work if both client and server are on same host, but not when they are on different hosts
- ▶ Or, the bug might depend on network load

# Read() and Write()

- ▶ We get all the same problems with write(), there is no guarantee that the requested number of bytes will be written
- ▶ If we are not careful the other end of the connection will only get part of the data
- ▶ These problems can even occur with text data
- ▶ For example if we send a name over a TCP connection, if the other end of the connection isn't careful it might not get the entire name if it does a standard read
- ▶ We need to have more robust versions of read() and write() that can handle these problems

# Read() and Write()

- ▶ There are several things that can be done
- ▶ One is to construct a reliable version of read(), say readn() that makes a better attempt to return the requested number of bytes
- ▶ We can also construct special purpose read procedures for different data types
- ▶ For strings the exchange could consist of two parts:
  - ▶ An integer, the number of characters in the string
  - ▶ The characters in the string
- ▶ This ensures that we get the complete string

# Read() and Write()

- ▶ When reading data, if we miss a few bytes in a read, the rest of the stream is likely to be garbage
- ▶ Consider the case of reading two integers
- ▶ If the first read misses the last two bytes of the integer, the first integer will be garbage
- ▶ Now when we read the second integer we will get the first two bytes of the first integer, plus part of the second integer
- ▶ The second integer will be garbage as well
- ▶ It's likely that everything else will be wrong and you will end up with strange bugs in your program

# Read() and Write()

- ▶ Build a reliable read() function, called readn()
- ▶ Our first step is to write a read procedure that will attempt to read the number of bytes that we have requested
- ▶ There are only two conditions where we won't be able to read the requested number of bytes:
  - ▶ We have reached the end of file
  - ▶ read() returns an error
- ▶ Both of these conditions indicate that something has gone wrong at the other end of the socket

# Read() and Write()

```
int readn(int fd, char *buffer, int count) {
    char *ptr;
    int n;
    int left;
    ptr = buffer;
    left = count;
    while(left > 0) {
        n = read(fd, ptr, left);
        if(n == 0) break;
        left -= n;
        ptr += n;
    }
    return(count - left);
}
```

# Read() and Write()

- ▶ Notice how this procedure keeps reading as long as we have not filled the buffer
- ▶ It also checks for the end of file condition, in this case it returns what we have read and we will deal with the problem later
- ▶ Now we have the problem of dealing with errors
- ▶ For most errors we want to return -1, but for the three that we indicated earlier we want to keep on reading
- ▶ The next slide shows the statements we need to add to our readn() procedure

# Read() and Write()

```
n = read(fd, ptr, left);
if(n == 0) break;
if(n < 0) {
    if(errno == EINTR || errno == EAGAIN || errno == EWOULDBLOCK) {
        n = 0;
    } else {
        return(-1);
    }
}
```

# Read() and Write()

- ▶ This handles the errors that really aren't errors
- ▶ We need to do the exact same thing for write, the code looks basically the same
- ▶ We can use this to build a procedure that will reliably read a string, where the length of the string is sent first followed by the characters in the string
- ▶ Assume that the length includes the '\0' character at the end of the string
- ▶ The code is on the next slide

# Read() and Write()

```
char *readString(int fd) {  
    short len;  
    char * buffer;  
    int ret;  
    len = readn(fd, &len, sizeof(len));  
    buffer = (char*) malloc(len);  
    ret = readn(fd, buffer, len);  
    if(ret != len)  
        return(NULL);  
    else  
        return(buffer);  
}
```

# Read() and Write()

- ▶ Note, that its an error if we don't get all of the characters in the string, in this case we return the NULL pointer
- ▶ The write procedure for strings looks basically the same
- ▶ There is one detail that we have left out, network byte order
- ▶ If we want to truly be general we should convert len from network byte order to the native byte order, do the reverse of writing
- ▶ If we can guarantee that we are going between x86 machines we can drop this
- ▶ But, ARM processors can be big endian, so buyer beware!

# Example

- ▶ Our next example illustrates how some of these techniques work
- ▶ There is example2.tar on Canvas that contains the code
- ▶ For the time being we will ignore adding a fork() to the server, we will come back to that later
- ▶ At this point we are only interested in reliably moving data
- ▶ In this example the client sends two integers to the server, which adds the two numbers and sends the result back to the client
- ▶ The client then responds with a message
- ▶ This illustrates both numeric and text transfer

# Example

- ▶ The first thing we do is construct a small library with the I/O procedures
- ▶ This is in lib.h and lib.c
- ▶ This library contains the four functions we've discussed:

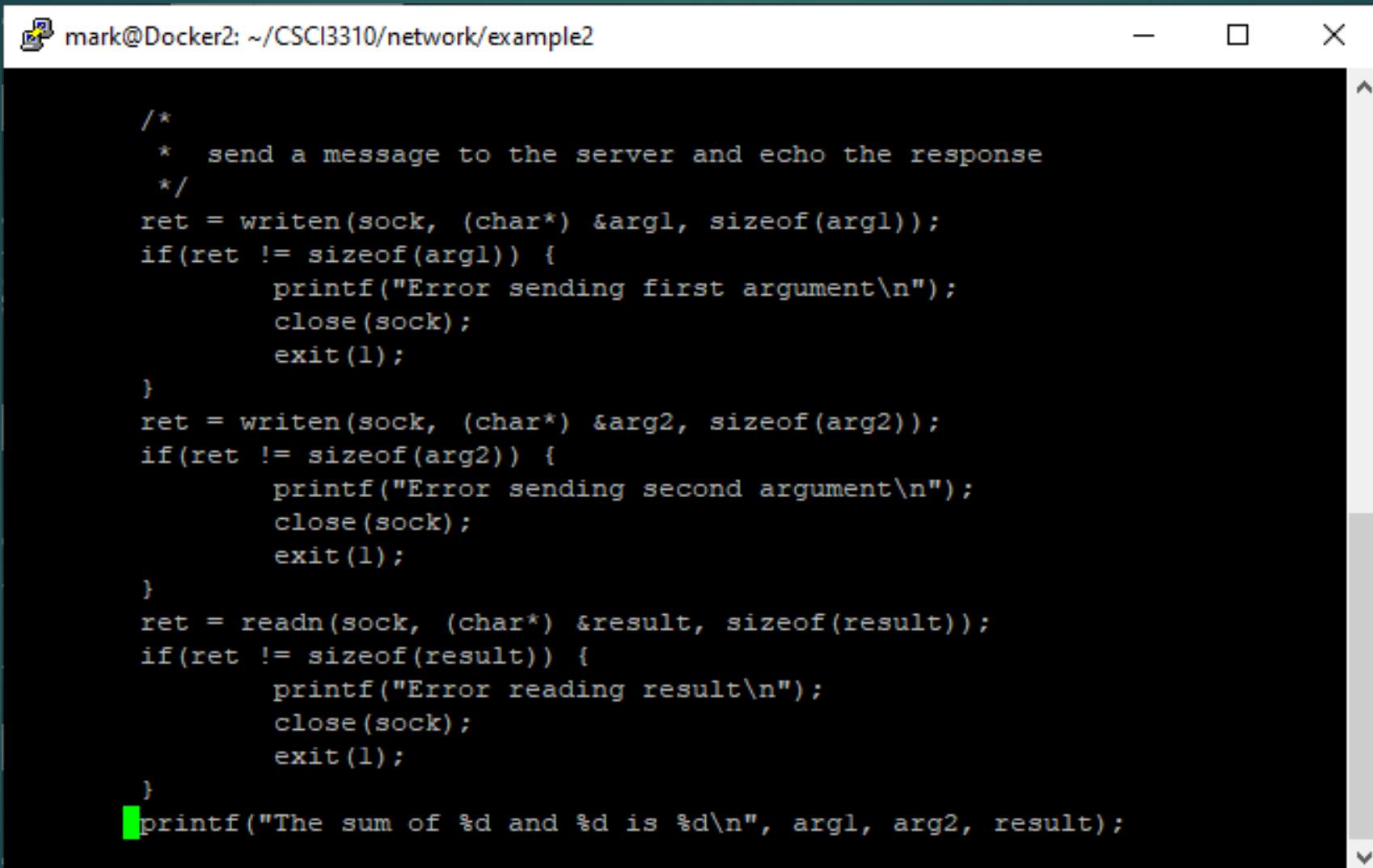
```
int readn(int fd, char *buffer, int count);  
int written(int fd, char *buffer, int count);  
char *readString(int fd);  
int writeString(int fd, char *string);
```

- ▶ These procedures are used in both the client and server

# Example

- ▶ The first part of both the client and server programs are the same, the only part that has changed is where they exchange data
- ▶ The next slide shows the first part of the client code
- ▶ The heart of this code is calling `writen()` twice to send the integers to the server and the `readn()` once to get the result
- ▶ The bulk of the code is checking the results of these functions for errors
- ▶ The following slide shows how `writeString()` can be used
- ▶ Again we are careful about error checking

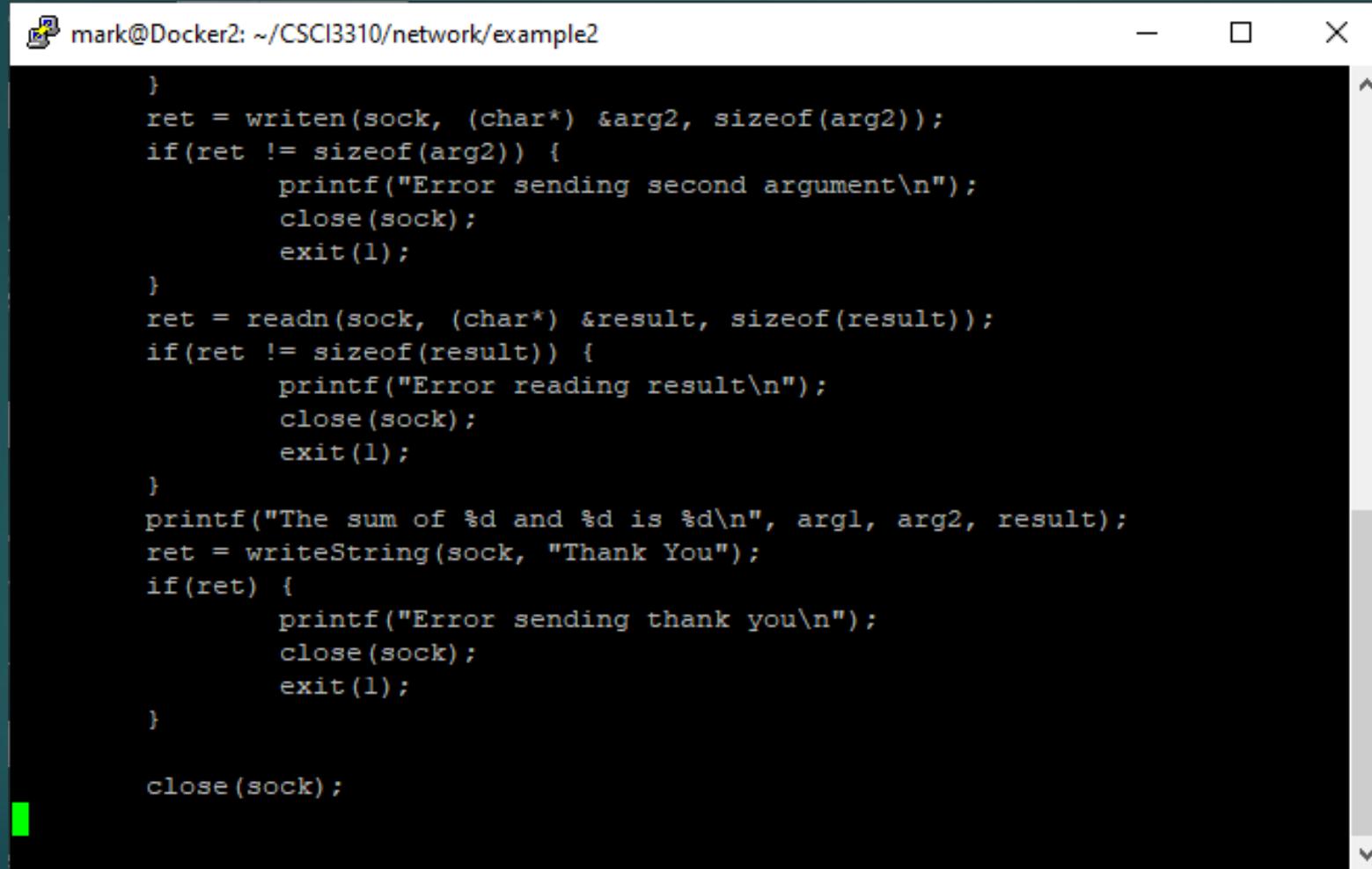
# Example



A screenshot of a terminal window titled "mark@Docker2: ~/CSCI3310/network/example2". The window contains the following C code:

```
/*
 *  send a message to the server and echo the response
 */
ret = writen(sock, (char*) &arg1, sizeof(arg1));
if(ret != sizeof(arg1)) {
    printf("Error sending first argument\n");
    close(sock);
    exit(1);
}
ret = writen(sock, (char*) &arg2, sizeof(arg2));
if(ret != sizeof(arg2)) {
    printf("Error sending second argument\n");
    close(sock);
    exit(1);
}
ret = readn(sock, (char*) &result, sizeof(result));
if(ret != sizeof(result)) {
    printf("Error reading result\n");
    close(sock);
    exit(1);
}
printf("The sum of %d and %d is %d\n", arg1, arg2, result);
```

# Example



A screenshot of a terminal window titled "mark@Docker2: ~/CSCI3310/network/example2". The window contains C code for a network application. The code includes file operations like writing to a socket, reading from a socket, and closing it. It also includes error handling for these operations and a final printf statement.

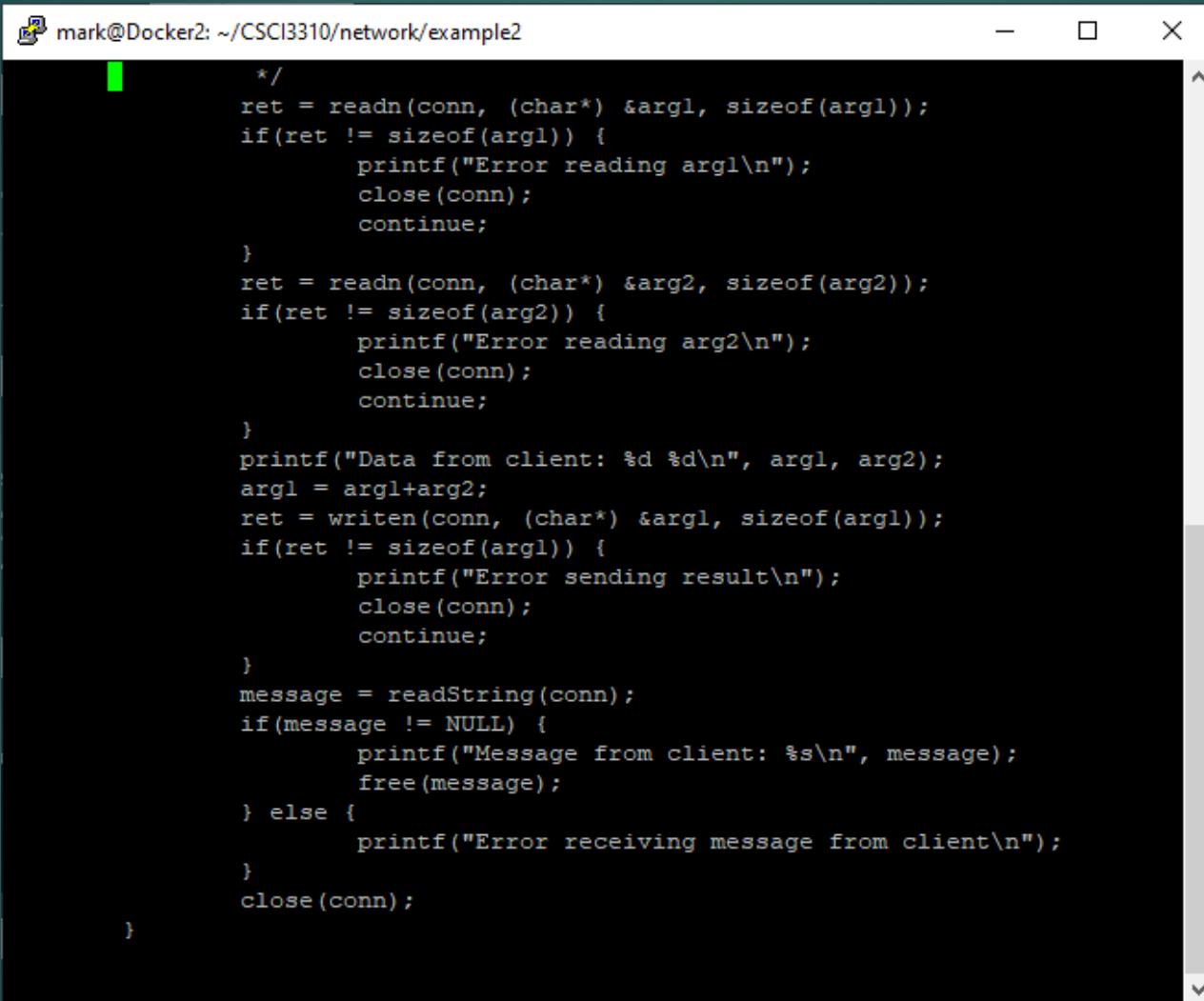
```
    }
    ret = writen(sock, (char*) &arg2, sizeof(arg2));
    if(ret != sizeof(arg2)) {
        printf("Error sending second argument\n");
        close(sock);
        exit(1);
    }
    ret = readn(sock, (char*) &result, sizeof(result));
    if(ret != sizeof(result)) {
        printf("Error reading result\n");
        close(sock);
        exit(1);
    }
    printf("The sum of %d and %d is %d\n", arg1, arg2, result);
    ret = writeString(sock, "Thank You");
    if(ret) {
        printf("Error sending thank you\n");
        close(sock);
        exit(1);
    }

    close(sock);
```

# Example

- ▶ The following slide shows the corresponding code for the server
- ▶ In this case it uses `readn()` twice to read in the numbers and `written()` once to send them to the client
- ▶ The server then reads the string from the client and displays it
- ▶ The following slide shows some interactions between the client and the server
- ▶ In this example I originally used an `int` instead of a `short` in one of the procedures, the strings didn't transfer correctly
- ▶ This took some time to find, so you need to be careful when writing network code

# Example



A screenshot of a terminal window titled "mark@Docker2: ~/CSCI3310/network/example2". The window contains the following C code:

```
/*
 * 
ret = readn(conn, (char*) &arg1, sizeof(arg1));
if(ret != sizeof(arg1)) {
    printf("Error reading arg1\n");
    close(conn);
    continue;
}
ret = readn(conn, (char*) &arg2, sizeof(arg2));
if(ret != sizeof(arg2)) {
    printf("Error reading arg2\n");
    close(conn);
    continue;
}
printf("Data from client: %d %d\n", arg1, arg2);
arg1 = arg1+arg2;
ret = writen(conn, (char*) &arg1, sizeof(arg1));
if(ret != sizeof(arg1)) {
    printf("Error sending result\n");
    close(conn);
    continue;
}
message = readString(conn);
if(message != NULL) {
    printf("Message from client: %s\n", message);
    free(message);
} else {
    printf("Error receiving message from client\n");
}
close(conn);
}
```

# Example

```
mark@Docker2: ~/CSCI3310/network/example2
mark@Docker2:~/CSCI3310/network/example2$ ./client 5 7
The sum of 5 and 7 is 12
mark@Docker2:~/CSCI3310/network/example2$ ./client 100 300
The sum of 100 and 300 is 400
mark@Docker2:~/CSCI3310/network/example2$ ./client -5 -7
The sum of -5 and -7 is -12
mark@Docker2:~/CSCI3310/network/example2$
```

```
mark@Docker2:~/CSCI3310/network/example2
mark@Docker2:~/CSCI3310/network/example2$ ./server
Data from client: 5 7
Message from client: Thank You
Data from client: 100 300
Message from client: Thank You
Data from client: -5 -7
Message from client: Thank You
[green bar]
```

# Server Architecture

- ▶ Returning to our server architecture, we've already handled the case where accept is interrupted by a signal
- ▶ There are other things that accept detects as errors, but we can easily recover from
- ▶ Consider the case of a busy server, the client sends a connect request, but it is queued by listen()
- ▶ By the time the server gets to the request, the client has canceled the request for taking too long
- ▶ In this case accept returns ECONNABORTED, but this isn't fatal for the server, it can just go on to the next connection request

# Server Architecture

- ▶ There are a number of other accept errors that are safe to ignore, but they tend to be system dependent
- ▶ The best thing to do is look at the man page for accept() on the system you are using
- ▶ In the case of Linux there are 8 errors that can be ignored, and just go on to the next client
- ▶ Most of these errors deal with protocol errors or hardware errors that effect a single client
- ▶ It is safe to ignore the request and go on to the next client
- ▶ But, this does result in a lot of condition checking in our server code

# Server Architecture

- ▶ Now we need to address the problems that can occur when either the client or server crash
- ▶ We can deal with some of these issues now, and return to them when we have covered more of the networking API
- ▶ Start with the client crashing, in this case the client end of the socket is closed, the server cannot read and write to it
- ▶ The child process is handling this interaction, so it will either crash or terminate itself and the main server process will be informed
- ▶ An attempt to read on a socket that has been closed will detect an end of file and return 0 (or fewer bytes than requested)

# Server Architecture

- ▶ In the case of a read() this is easy to detect and handle
- ▶ A write to a closed socket produces a SIGPIPE signal, which by default terminates the process
- ▶ We can specify that this signal is ignored, in this case the write will get an EPIPE error
- ▶ At this point the child process should clean up and terminate
- ▶ This is important if there is client data to be saved
- ▶ Similar things happen on the client side, but there is a problem
- ▶ It may take some time to detect these conditions

# Server Architecture

- ▶ If the child process on the server dies the client will face a similar situation and will need to recover from it
- ▶ But things can get more complicated, largely to do with timing
- ▶ If the client is blocked, for example waiting for user input, it may not detect that the server is down
- ▶ The user could do a considerable amount of work without knowing this is the case
- ▶ For example, the client could spend several minutes attempting to send data to the server before concluding that something is wrong

# Server Architecture

- ▶ The problem may not be with the server
- ▶ Any data that is sent over the Internet goes through many routers and switches, if one of these fails the server may become unreachable
- ▶ It could even be someone cutting a wire, this can happen during construction work
- ▶ In this case an attempt is made to find a different route from the client to server
- ▶ This is why there are multiple tries and a wait of several minutes before an error is reported

# Server Architecture

- ▶ Another interesting timing bug occurs when a server crashes and is then restarted and the client is not aware of this
- ▶ Think about a web browser, a user can spend several minutes reading a web page, the server could crash and recover during this time
- ▶ When the server crashes all of the TCP connections are lost
- ▶ In this case the client must reconnect with the server
- ▶ We would like this to happen without the user knowing
- ▶ This requires detecting these situations before an error occurs

# Server Architecture

- ▶ We can do basically the same thing with threads
- ▶ Instead of creating a new process after an accept a new thread is created
- ▶ There is one thing we need to be careful about: signals
- ▶ In the case of a child process a signal to the child process only goes to the child process
- ▶ In the case of threads, it goes to the single process, all the threads will see it
- ▶ This requires a little bit of care

# I/O Multiplexing

- ▶ So far all the I/O we have done is blocking
- ▶ When we do a read or write the process blocks
- ▶ With networking we can be using multiple file descriptors, if we block for one we can miss things on the other descriptors
- ▶ We've seen that this can cause problems when interacting with the user and not realizing that the server is down
- ▶ There are several solutions to this problem
- ▶ The first one we will look at gives us our third server architecture as well

# I/O Multiplexing

- ▶ Consider the case where we have two descriptors that we could read, we don't know which one will have data
- ▶ This is a problem, if we read the wrong descriptor the process will block even if there is data to read on the other descriptor
- ▶ We would like to know if there is data to read before we call the `read()` system call
- ▶ Similarly, a `write()` can block if there isn't buffer space in the kernel for the data to be written
- ▶ This can happen with networks, again a situation that we would like to avoid

# I/O Multiplexing

- ▶ The solution to this problem is the select() system call, this system call tells us which file descriptors are ready to use
- ▶ The declaration of this system call is:

```
#include <sys/select.h>
#include <sys/time.h>
int select(int nfds, fd_set *readset, fd_set *write_set, fd_set
*exceptset, struct timeval *timeout);
```

- ▶ This is a fairly complicated system call
- ▶ The second, third, and fourth parameters specify the file descriptors we are interested in

# I/O Multiplexing

- ▶ fd\_set used to be a bit vector, but the OS is free to implement it any way that it likes
- ▶ If we are interested in reading a descriptor we add it to readset, if we are interested in writing to a descriptor we add it to writeset
- ▶ We can add as many descriptors as we like to these sets
- ▶ When we call select, it will check to see if any of the descriptors in these sets are ready
- ▶ If they are, their entry in the set will be set, this is how we know which descriptor to read or write

# I/O Multiplexing

- ▶ There are four macros that can be used to manipulate fd\_set values:

`void FD_ZERO(fd_set *set)` - clear the set

`void FD_SET(int fd, fd_set *set)` - add fd to the set

`void FD_CLR(int fd, fd_set *set)` - remove fd from the set

`int FD_ISSET(int fd, fd_set *set)` - is fd in the set?

- ▶ The fd\_set parameters are both input and output, they are changed by select
- ▶ On return if a file descriptor is in readset then it is ready for reading

# I/O Multiplexing

- ▶ The first parameter to select is one greater than the maximum file descriptor number
- ▶ This assists the kernel with being more efficient, it knows the number of file descriptors it needs to check
- ▶ It also knows how much of readset, writeset and exceptset needs to be transferred to the kernel
- ▶ The last parameter specifies how long select should wait before returning
- ▶ There is a wide variety of options here
- ▶ If this parameter is NULL, select is blocking

# I/O Multiplexing

- ▶ Select blocks until one of the file descriptors becomes active, or it is interrupted, say by a signal
- ▶ The other options are based on the timeval structure:

```
struct timeval {  
    long tv_sec;      //seconds  
    long tv_usec;     //microseconds  
};
```

- ▶ We can use this parameter to set how long select should wait, if its zero select will return immediately with its result
- ▶ Select can always be terminated early by an interrupt

# I/O Multiplexing

- ▶ The value returned by select is the number of file descriptors that have been set in its fd\_sets
- ▶ If the number is zero, nothing is ready
- ▶ As usual a negative result indicates an error
- ▶ Any of the fd\_set parameters can be NULL
- ▶ Assume that we have two descriptors, fd1 and fd2, that we want to read from
- ▶ The following code shows how we can use select to determine which of these file descriptors is ready to read

# I/O Multiplexing

```
int fd1, fd2, maxfd, nfd;  
fd_set readset;  
...  
FD_ZERO(&readset);  
FD_SET(fd1, &readset);  
FD_SET(fd2, &readset);  
maxfd = max(fd1, fd2)+1;  
nfd = select(maxfd, readset, NULL, NULL, NULL);  
if(nfd > 0) {  
    if(FD_ISSET(fd1, readset))  
        read from fd1  
    if(FD_ISSET(fd2, readset))  
        read from fd2  
}
```

# Server Architecture

- ▶ A socket that we've called listen on can be used with select
- ▶ It is added to the readfds set and select will check if accept() can be called without blocking
- ▶ With this in mind we can build a server that doesn't use fork() or threads, yet can still process multiple clients at that same time
- ▶ When sockets were first introduced processes were quite expensive and limited in number, there were no thread package as well
- ▶ This was the only way of producing efficient and reliable servers, our third architecture

# Server Architecture

- ▶ Our server is based on a readfds set that includes the socket waiting for an accept as well as each of the client sockets
- ▶ The server has a loop that starts with a select statement waiting for a new connection or something to read from a client
- ▶ It then processes the accept, or the client interaction and then returns to a select statement
- ▶ This architecture is outlined on the following slide

# Server Architecture

```
fd_set readfds, allfds;  
...  
while(TRUE) {  
    readfds = allfds;  
    select(maxfds, &readfds, NULL, NULL, NULL);  
    if(FD_ISSET(acceptsock, &readfds))  
        add new client to allfds  
    for(i=0; i<NumClients; i++) {  
        if(FD_ISSET(sock[i], &readfds))  
            read from client i  
    }  
}
```

# Server Architecture

- ▶ Construct a server based on select(), examine that approach to server architecture
- ▶ The example is based on an echo server, a server that echoes everything that is sent to it
- ▶ A client can send multiple lines, and each line is echoed one after the other
- ▶ There is a standard echo server on most computers, but this is a nice example that has a longer term connection than the ones we have already examined

# Server Architecture

- ▶ In our previous examples, the child process did the bookkeeping for us, the same thing would happen with threads
- ▶ Each child was responsible for one connection and kept track of the socket
- ▶ With a select based server we need to do all of the bookkeeping ourselves
- ▶ We need to keep track of all the sockets connected to the server
- ▶ Need to check when they are ready to read, when they are no longer connected

# Server Architecture

- ▶ We use two data structures for this purpose
- ▶ The first data structure is an array for storing the socket file descriptors, the second data structure is an fd\_set for determining when a socket can read
- ▶ How do we know how big to make the array?
- ▶ We will be using select, and an fd\_set to know when a client socket is ready to read, the maximum size of fd\_set is FD\_SETSIZE, so that should be the size of the array:

```
int client[FD_SETSIZE];  
fd_set allfds;
```

# Server Architecture

- ▶ When a client connects to the server we add its socket to client[] and when they disconnect we remove the socket
- ▶ How do we do this efficiently?
- ▶ File descriptors are positive integers, so we initialize all the entries in client[] to -1
- ▶ When a client connects, we find the first entry in client that is -1 and store the socket there
- ▶ When the client disconnects we set the entry back to -1
- ▶ Since the client array could be large, maxi, is the largest index where there is a file descriptor

# Server Architecture

- ▶ Similarly when a client connects its entry in allfds is set and when they disconnect its entry is cleared
- ▶ We also add the listening socket to allfds, so it can be used in select
- ▶ We use maxfd to keep track of the largest socket file descriptor
- ▶ The first part of the server is the same as our other servers, the part after the call to listen is different
- ▶ It starts by initializing the data structures and then we enter the infinite loop that processes connections and responding the clients

# Server Architecture

- ▶ The following is the initialization code:

```
maxfd = sock;  
maxi = -1;  
for(i=0; i<FD_SETSIZE; i++)  
    client[i] = -1;  
FD_ZERO(&allfds);  
FD_SET(sock, &allfds);
```

# Server Architecture

- ▶ At the start of the while loop we set readfds to allfds, all the sockets we could read or do an accept on
- ▶ The call select() in blocking mode to see if anything is ready to be processed:
- ▶ `nready = select(maxfd+1, &readfds, NULL, NULL, NULL);`
- ▶ When select() returns either there is some action to perform, or select() was interrupted
- ▶ Since we aren't using signals, interrupts should be very rare
- ▶ The first thing we do is check to see if another client wants to connect

# Server Architecture

```
if(FD_ISSET(sock, &readfds)) {
    conn = accept(sock, (struct sockaddr*) &address, &addrLength);
    /*
     * find an entry in the client table for it
     */
    for(i=0; i<FD_SETSIZE; i++) {
        if(client[i] < 0) {
            client[i] = conn;
            break;
        }
    }
    FD_SET(conn, &allfds);
    if(conn > maxfd)
        maxfd = conn;
    if(i > maxi)
        maxi = i;
    if(--nready <= 0)
        continue;
}
```

# Server Architecture

- ▶ This approach works reasonably well if we don't have a lot of simultaneous connects
- ▶ If there were thousands of connections, finding a slot in client[] could be slow, but given network speed this probably isn't much of an issue
- ▶ Note that nready is the number of sockets that need to be serviced, this gives us a way of terminating the process early
- ▶ The code for processing the client connections is on the next slide

# Server Architecture

```
for(i=0; i<=maxi; i++) {
    conn = client[i];
    if(conn < 0)
        continue;
    if(FD_ISSET(conn, &readfds)) {
        message = readString(conn);
        /*
         * has the client disconnected
         */
        if(message == NULL) {
            close(conn);
            client[i] = -1;
            FD_CLR(conn, &allfds);
        } else {
            writeString(conn, message);
            free(message);
        }
    }
}
```

# Server Architecture

- ▶ Basically loop through all the client sockets to see if one is ready to read
- ▶ An end of file on a socket also makes it ready to read, which is the first thing we check for
- ▶ In this case we close the socket, remove the socket from the client array and allfds
- ▶ If the connection is still active, the line is echoed back to the client
- ▶ In this case the client processing was quite short, so its in the main loop
- ▶ Otherwise, it should be a separate procedure

# Server Architecture

- ▶ Next examine the client
- ▶ Again the first part of the client is the same, it still needs to connect to the server
- ▶ The code after the connection is different
- ▶ It is a loop that continues to an end of file
- ▶ It reads a line from the terminal, sends it to the server and prints the line returned from the server
- ▶ This part of the client is shown on the next slide

# Server Architecture

```
while(1) {
    ret = fgets(buffer, 512, stdin);
    /*
     * check for user entering end of file
     */
    if(ret == NULL) {
        shutdown(sock, SHUT_WR);
        break;
    } else {
        writeString(sock, buffer);
    }
    ret = readString(sock);
    printf("%s", ret);
}
```

# Server Architecture

- ▶ The main interesting part is the call to shutdown() when the end of file is reached
- ▶ In this example it isn't necessary, but for more complicated clients it is
- ▶ We want to tell the server there is no more input, with a normal connection we would just close the file
- ▶ But, recall the sockets are two way, if we close the file the server can no longer write to the client
- ▶ This causes a signal that could terminate the server

# Server Architecture

- ▶ Our server doesn't use child processes, so this would crash the server
- ▶ In the client we only want to close the write side of the socket, so the server sees the end of file
- ▶ We don't want to close the read side until the server is finished sending data to us
- ▶ The shutdown() procedure closes just one side of the socket, so we get a clean termination of the connection

# Summary

- ▶ Examined more the details of TCP servers and clients
- ▶ Discussed three types of server architectures, using child processes, threads or select()
- ▶ Examined reliable way of transferring data
- ▶ Examined what can happen when either the client or server crashes

# CSCI 3310

# Networking

# Introduction

MARK GREEN  
ONTARIO TECH

# Introduction

- ▶ Now that we've seen some network programming time to move on to network theory
- ▶ There is a lot of terminology in this area, partly due to the wide range of technologies
- ▶ Also due to the fact that part of the network infrastructure has been around for over a century
- ▶ Digital information exchange started in the 1930s with systems like **Telex**
- ▶ This predates computers by decades and has an impact on some of the terminology

# Telex – Teletype ASR33



# Types of Networks

- ▶ There are many ways of classifying networks
- ▶ One way is by the type of connection:
  - ▶ **Point-to-point** – there is a direct link between the peers, a private conversation
  - ▶ **Broadcast** – the information is broadcast over the network, anyone can see it
- ▶ WiFi is an example of broadcast, while wired ethernet is an example of point-to-point
- ▶ In general broadcast is cheaper than point-to-point

# Types of Networks

- ▶ Another classification is based on distance
- ▶ **Personal area network** (PAN) is close to a person, usually a few meters at most
- ▶ A common example of this is Bluetooth
- ▶ This type of network has become very popular in the last 5 to 10 years, but it took a long time for it to develop, the standard is over 20 years old
- ▶ Before that we had IRDA, based in infrared signals, this is still used in remote controls

# Types of Networks

- ▶ The next level up is **local area networks** (LAN), these are typically restricted to a single building, such as a house
- ▶ There are quite a few technologies in this area
- ▶ WiFi has become popular since it's broadcast technology that doesn't need wires, reduces cost
- ▶ Wired ethernet is another common technology, it involves laying wires throughout the building, which is more expensive
- ▶ Power lines can also be used, the wires already exist, just need to buy an adapter for each device, cost about \$30 each

# Types of Networks

- ▶ Home networks have evolved considerably over the past 2 decades, in ways the networking industry didn't expect
- ▶ Initially one Internet connection came into the house, a modem with a single ethernet connection
- ▶ Contract stated it could only be used by one computer
- ▶ Not long before people started setting up networks in their own homes
- ▶ Now home modems have multiple ethernet connections, plus WiFi, has become a selling point

# Types of Networks

- ▶ Ethernet will always outperform WiFi
  - ▶ Basic signalling is at least an order of magnitude faster
  - ▶ The connection isn't shared, you get the full bandwidth
- ▶ Since WiFi is broadcast the signal can be intercepted
- ▶ There is encryption, but some people don't know how to set it up correctly, or even turn it off, it's a pain entering WiFi passwords
- ▶ WiFi encryption can be broken, this is why you should be careful with public WiFi
- ▶ Home networks now include TVs, appliances, and other devices

# Types of Networks

- ▶ The next level up is **metropolitan area networks** (MAN), connect multiple sites within metropolitan area
- ▶ This is what comes into your house and what Bell and Rogers manage on a city scale
- ▶ Based on wires or fibre, which means cables must be laid, particularly if you want high speed
- ▶ This is cost effective in urban areas with dense population
- ▶ In rural areas this is much harder, can't justify the cost of laying cables when houses are some distance apart

# Types of Networks

- ▶ This has become a very hot political issue in rural areas that currently lack affordable Internet connections
- ▶ There is a push for governments to provide funding
- ▶ ADSL is a technology that runs over standard phone lines, it typically gets 3Mbps
- ▶ Only works if you are close to a telephone concentrator box, usually within 3 to 5km
- ▶ An alternative is satellite, which is more expensive than ADSL and has higher latency, but it offers considerably higher bandwidth

# Types of Networks

|           | <b>Download<br/>(Mbps)</b> | <b>Upload<br/>(Mbps)</b> | <b>Latency<br/>(msec)</b> | <b>Jitter (msec)</b> |
|-----------|----------------------------|--------------------------|---------------------------|----------------------|
| Xplornet  | 15.2                       | 1.46                     | 610                       | 39                   |
| Bell ADSL | 2.08                       | 0.58                     | 13                        | 7.5                  |
| Rogers    | 79.9                       | 16.1                     | 11                        | 4.2                  |

# Types of Networks

- ▶ One of the main challenges in MANs is the last mile
- ▶ This is the technology that takes the signal from the local switching box to your house
- ▶ This is the expensive part
- ▶ There have been multiple attempts to use some form of wireless technology for this
- ▶ None of it seems to have panned out
- ▶ There was a trial in Oshawa, I'm not sure what happened to it

# Types of Networks

- ▶ The largest distance scale is **wide area networks** (WAN)
- ▶ There is a wide range of technologies used here
- ▶ One common approach is fibre, a single cable will have many fibres, each of which can carry many signals
- ▶ These cables were originally laid by the telephone companies, you can rent a fibre, or part of a fibre
- ▶ Monthly rents are in the \$10,000 range
- ▶ Microwave is also used for this, eliminates the need to lay cables, but you need to build a tower every 50km

# Types of Networks

- ▶ Satellite is also an option here, can afford powerful ground stations that have high bandwidth
- ▶ All of these technologies are point-to-point, so we need some way of routing the data
- ▶ Devices called **switches** and **routers** are used for this purpose
- ▶ They route the packets from one router to the next
- ▶ There are many algorithms for routing and packet forwarding
- ▶ Want to find the best route, but have very little time to compute this
- ▶ Needs to be dynamic in case there are failures

# Types of Networks

- ▶ WANs use many incompatible networking technologies, sometimes over cables that were laid over 50 years ago
- ▶ Need to be able to convert between the different technologies, called **internetworking**
- ▶ Based on gateways that can convert from one network technology to another
- ▶ This could be a dedicated computer, or part of a switch

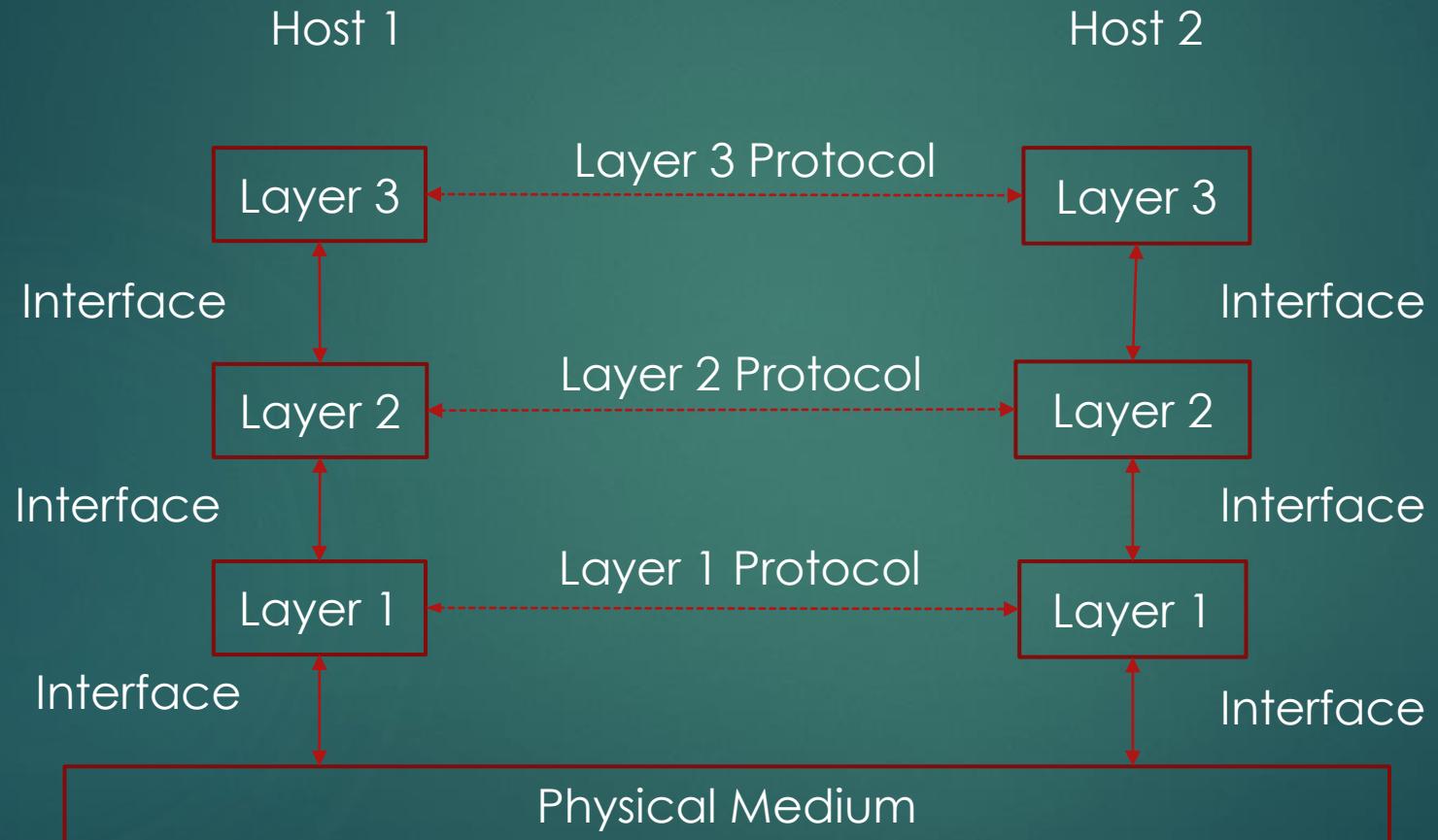
# Network Architecture

- ▶ Designing networks to be fast and reliable isn't easy
- ▶ On top of this there can be multiple technologies of different ages
- ▶ Technologies also change, sometimes quite rapidly, want to minimize the impact this has on our network
- ▶ Want to have a modular architecture, can easily replace one component without changing the rest of the network
- ▶ This goes for both the hardware and software
- ▶ Since we aren't electrical engineers, we will concentrate on the software

# Network Architecture

- ▶ The approach that is now used is a layered one
- ▶ Multiple layers handle different aspects of the network, one builds on top of the other
- ▶ The bottom layer deals directly with the hardware, while the top layer interacts with the application
- ▶ There are between 4 and 7 layers in most of these architectures
- ▶ An abstract view of this is shown on the next slide

# Network Architecture



# Network Architecture

- ▶ There is a lot going on in this model
- ▶ If an application on Host 1 sends a message to an application on Host 2, the process starts at the top of the left side
- ▶ The message gets passed down the layers until it reaches the physical medium, the wire
- ▶ It then goes up the right side starting at the bottom and working its way up to the application on Host 2
- ▶ The two applications only interact at layer 3, they aren't aware of the other layers or the physical medium

# Network Architecture

- ▶ Each layer has a well defined **interface** that it exports to the layer above it
- ▶ The layer provides a set of **services** and there is a well defined interface to these services
- ▶ The implementation of a layer can be replaced by another implementation of that layer, and none of the other components will know
- ▶ This is how we can evolve the network as technology changes, we just needs to replace one of the boxes and not the whole network
- ▶ Very similar to the idea of object oriented programming

# Network Architecture

- ▶ **Protocols** are another important part of this model
- ▶ There is a protocol between layers on the two sides
- ▶ A protocol is a set of rules that governs the conversation between corresponding layers on the two sides
- ▶ The layers don't directly communicate with each other, this is done by passing information up and down the model
- ▶ But, abstractly they are communicating, they act like there is a direct connection between them
- ▶ This is called a **protocol stack**, there are multiple layers of protocols, each dealing with one aspect of the network

# Network Architecture

- ▶ Protocol design is a difficult process and it can take some time to get it right
- ▶ There are a number of things that must be considered
- ▶ Two important ones are correctness and performance
- ▶ A protocol is correct if it accurately moves information from one host to another
- ▶ Bad protocol design can result in race conditions, deadlocks, infinite loops, lost data, etc.
- ▶ In addition, we don't want a protocol to introduce a large amount of overhead

# Network Architecture

- ▶ A protocol stack will need to deal with error detection and correction if it is to provide a reliable service
- ▶ It needs to deal with **addressing or naming**, need to know the destination of the data
- ▶ Once we have a name we need to deal with **routing**, the path between the hosts
- ▶ **Flow control** handles the case where one host generates data faster than the other host can process it
- ▶ If the receiving host can't process the data quick enough, some will be lost and require retransmission

# Network Architecture

- ▶ **Congestion** occurs when there is too much traffic on a network
- ▶ This is also bad for performance
- ▶ Protocol stacks need to build in a mechanism for controlling how fast hosts can send data
- ▶ Want to be able to share the network infrastructure fairly, make sure that some hosts are not blocked
- ▶ **Quality of service** (QoS) is another important consideration
- ▶ If you are watching a video you want the data to come at a uniform rate to give a good experience

# Network Architecture

- ▶ With video or sound, a few dropped packets don't matter
- ▶ It is always better to have slower bandwidth, but uniform rate, if the rate changes a lot the video will be jerky
- ▶ For a file transfer a lot of these issues don't matter, it is a bulk transfer for data
- ▶ The protocol stack should provide some way of specifying the QOS that your application needs
- ▶ Security is also an important consideration that can be addressed at the protocol level

# Important Concepts

- ▶ There are two important concepts:
  - ▶ **Protocol** – how adjacent layers on different hosts communicate, the rules they use for an orderly conversation
  - ▶ **Interface** – how adjacent layers in the same stack communicate, well defined services and interfaces that are provided by each layer
- ▶ Protocols go horizontal and interfaces go vertical
- ▶ Interfaces are within the same stack/host, protocols go between stacks/hosts

# Reference Models

- ▶ An abstract way of representing network structure
- ▶ The layers, interfaces and protocols that are required, without going into the details of them
- ▶ A way of talking about the different levels or components of a network, provides a common vocabulary for people working in the area
- ▶ Can also be a guide to network implementation
- ▶ Reference models look similar to the layer diagrams we have already seen

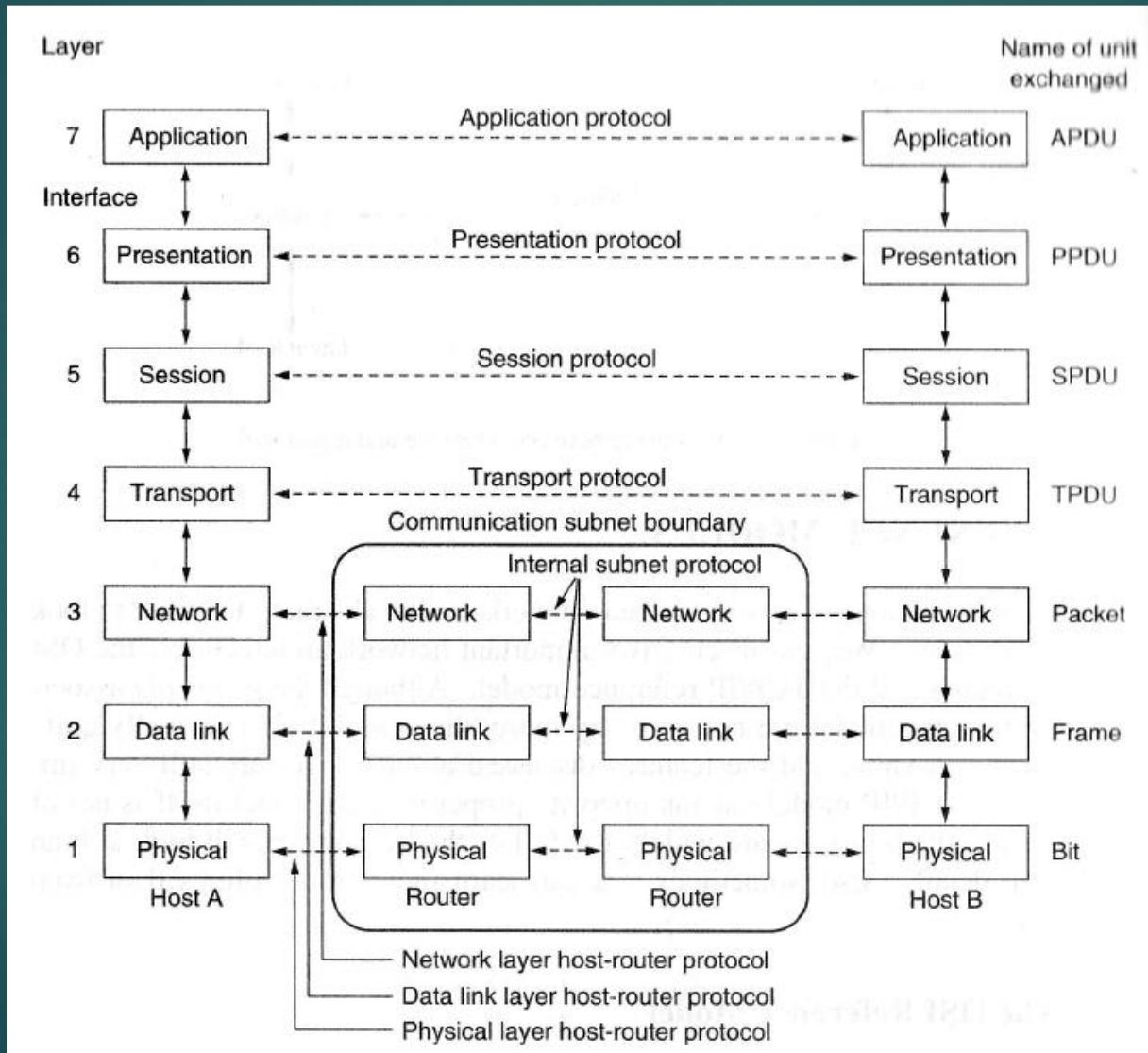
# OSI Reference Model

- ▶ Start by examining two standard reference models, that are very different in the way they were produced
- ▶ The **Open Systems Interconnection** or **OSI** model is a good example of top down development
- ▶ This model was developed by a large committee in the early 1980s
- ▶ There was little experience with developing large scale networks at that time, few real examples to draw upon
- ▶ This model has a number of good ideas and concepts, but is no longer used as an implementation guide

# OSI Reference Model

- ▶ The OSI reference model is a standard in the networking community, it is well known by most people, so it is well worth knowing
- ▶ It is where a lot of the terminology comes from
- ▶ The most important part of this model is its well defined layers, the services within each layer, the interfaces between layers and the need for protocols between peers
- ▶ This model has seven layers, shown on the next slide, briefly examine what each of these layer does, starting at the bottom

# OSI Reference Model



# OSI Reference Model

- ▶ The **physical** layer is the actual electrical connections and signals that are used to transfer information
- ▶ How 1 and 0 bits are represented electrically
- ▶ The types of wires that are used, and the connectors at the ends of the wires
- ▶ How transmissions occur over the wires and how connections are made
- ▶ The **data link** layer builds structure on top of the physical layer
- ▶ It deals with the errors that can occur at the transmission level

# OSI Reference Model

- ▶ It takes the data from the higher levels and packages it into a form that the physical layer understands
- ▶ This could involve dividing the data into packets that are an appropriate size for transmission
- ▶ Handles access to the physical layer
- ▶ The abstraction is a link between two connected computers or network devices, no notion of addressing
- ▶ The **network** layer provides the routing from the data source to the destination
- ▶ This provides the network abstraction where computers are not directly connected

# OSI Reference Model

- ▶ It also deals with the differences between different network technologies, such as address formats and packet sizes
- ▶ The **transport** layer deals with how data is transported over the network
- ▶ Whether it's a point-to-point connection or a data gram service
- ▶ It deals with broadcasting information to multiple receivers
- ▶ This is a true peer-to-peer connection between computers
- ▶ The **session** layer deals with sessions, not a very well defined concept
- ▶ Really comes from time sharing systems, where you login to a session

# OSI Reference Model

- ▶ The **presentation** layer deals with how data structures are represented on the network
- ▶ A common representation for data structures, similar to what we've seen with big endian vs. little endian
- ▶ How computers with different internal representations can communicate
- ▶ The **application** layer is where application level protocols reside, such as http, file transfers, email, etc.
- ▶ The application specific protocols

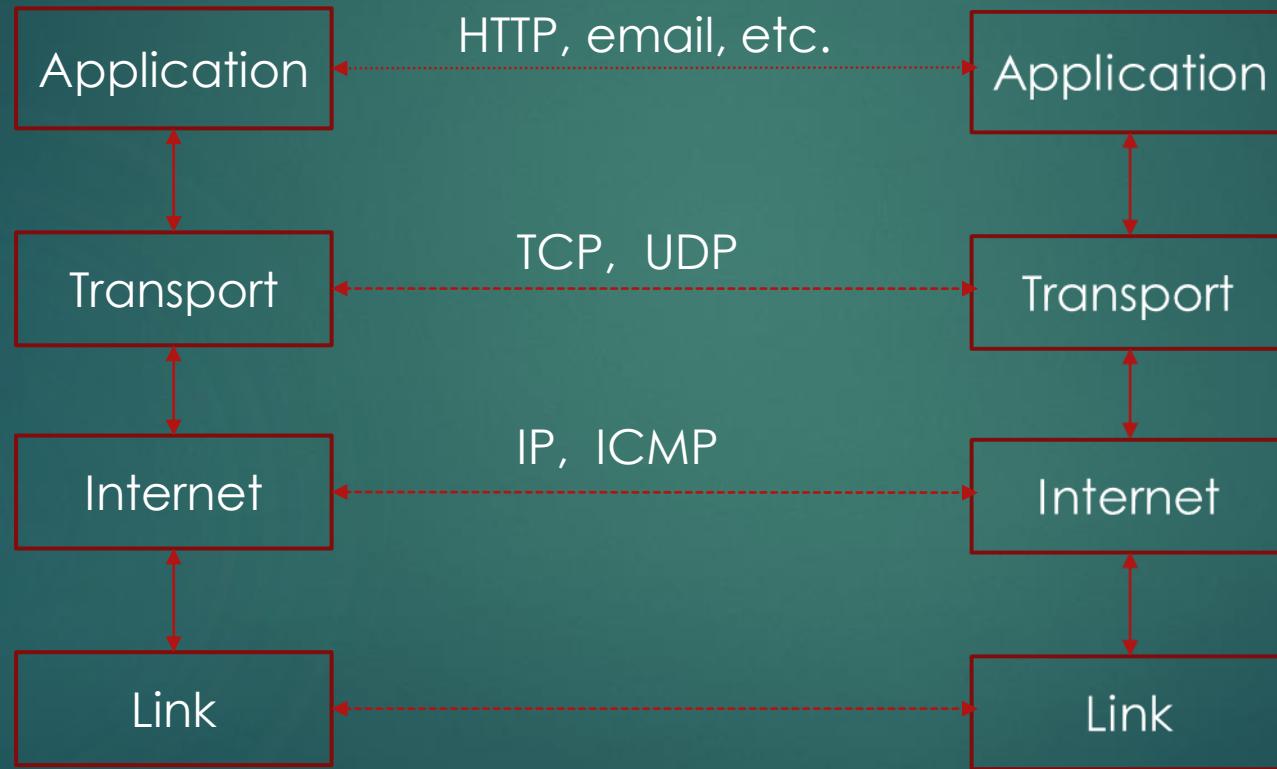
# OSI Reference Model

- ▶ One of the criticisms of the OSI model is it has too many layers
- ▶ Are the session and presentation layers really required?
- ▶ It is also quite difficult to understand, the total standard required a meter of paper
- ▶ There are protocols associated with OSI, but they were hard to implement and didn't perform very well
- ▶ By the time OSI was finalized TCP/IP had become a de facto standard so no one was really interested in an OSI implementation

# TCP/IP Reference Model

- ▶ The **TCP/IP** reference model was developed in the exact opposite way
- ▶ It started with a working network and protocols and the reference model was added as an after thought
- ▶ Initially designed for ARPANET, early network support by US defense department
- ▶ Wanted a network that would be resilient in war, would function even if some of the nodes were destroyed
- ▶ This lead to a packet based network where packets were routed dynamically

# TCP/IP Reference Model



# TCP/IP Reference Model

- ▶ All the hardware related issues were put in the **link** layer, the reference model says very little about this
- ▶ It exists and it carries packets of bytes
- ▶ The **Internet** layer is roughly equivalent to the network layer in the OSI model
- ▶ It handles delivering packets from their source to their destination
- ▶ They can be delivered by different routes and can arrive out of order
- ▶ They can also be lost in transmission

# TCP/IP Reference Model

- ▶ There are two protocols at this level:
  - ▶ **Internet Protocol** (IP) – the packet format that transmits the data
  - ▶ **Internet Control Message Protocol** (ICMP) – assists with managing the network
- ▶ The **transport** layer provides the abstraction that allows programs on different computers to communicate with each other
- ▶ The two main protocols here are:
  - ▶ **Transmission Control Protocol** (TCP)
  - ▶ **User Datagram Protocol** (UDP)

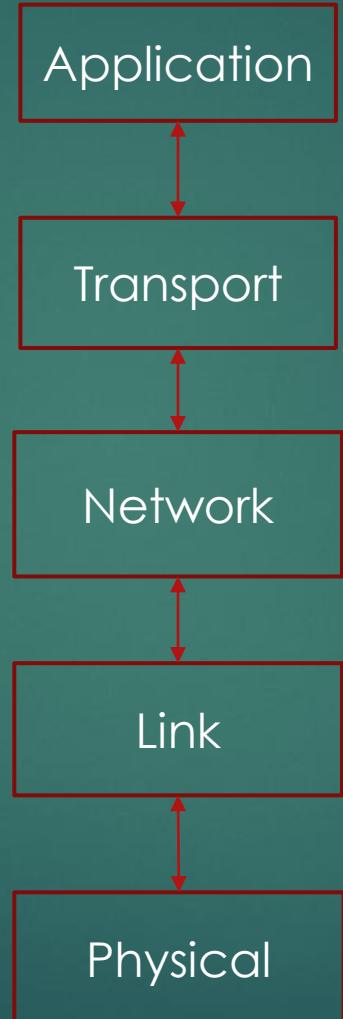
# TCP/IP Reference Model

- ▶ The **application** layer is the protocols that are defined by individual applications, this is the same as the OSI model
- ▶ The TCP/IP model could definitely be implemented efficiently, that was shown before the model was created
- ▶ Unfortunately it's not very general, it can really only describe TCP/IP and very closely related networks
- ▶ The layers aren't as cleanly defined as in OSI
- ▶ In addition, it says very little about the hardware and how the protocols interface with it

# Tanenbaum's Model

- ▶ The OSI model has too many layers and the TCP/IP model is missing layers
- ▶ The OSI layers are well defined, the TCP ones are not
- ▶ The OSI model is general, TCP is not
- ▶ To get around this problem we will use Tanenbaum's model
- ▶ Tanenbaum is the author of several of the main networking textbooks
- ▶ He uses a model in his books that lies between OSI and TCP/IP and provides a nice way of talking about networks

# Tanenbaum's Model



# Tanenbaum's Model

- ▶ We will basically follow this model, with more emphasis on the higher layers
- ▶ There are a lot of interesting things going on in the lower layers, but they really aren't of interest to application programmers
- ▶ We will mainly concentrate on the TCP/IP protocol stack, since this is by far the most common protocol stack

# Summary

- ▶ Examined the main terminology in the networking field
- ▶ Classified networks based on the distance traveled
- ▶ Introduced reference models, two main ones:
  - ▶ OSI
  - ▶ TCP/IP
- ▶ This is the starting point for our study of networks

# CSCI 3310

# Physical Layer

MARK GREEN  
ONTARIO TECH

# Introduction

- ▶ This layer deals with the physical movement of bits along the transmission medium -> the wire
- ▶ How a 1 and a 0 are represented
- ▶ The connectors that are used at either end of the wire
- ▶ These are low level concerns
- ▶ It is not concerned with how the data is packaged into bytes or frames
- ▶ Need a basic understanding of the different types of media to know which ones are appropriate in a given situation

# Wires

- ▶ The most basic way of carrying a signal is a wire
- ▶ We need at least two wires, the voltage in a signal must be measured against some standard, we call this ground
- ▶ There must be at least one wire that carries the ground reference
- ▶ For low speeds we can use straight wires to carry the signals, particularly over short distances
- ▶ This is used in serial communications standards such as RS-232
- ▶ With careful design serial communications can reach 100Kbps, this is not very high

# Wires

- ▶ The problem with straight wires is they act like an antenna, they can be a source of radiation, and detect radiation, they are radio transmitters and receivers
- ▶ If we run two wires close to each other, the signal will be transmitted from one wire to the other, this is called crosstalk
- ▶ As the speed increases this becomes more important
- ▶ The solution to this is **twisted pair**, two wires are twisted together to carry one signal

# Twisted Pair



# Twisted Pair

- ▶ The twisting of the wires breaks the antenna, so the wire doesn't radiate
- ▶ The signal is the difference in voltage levels between the two wires, this increases noise resistance
- ▶ If there is an electrical noise it will effect both wires, it will shift the absolute voltage levels, but not the relative voltage between the two wires
- ▶ The quality of twisted pair depends upon the number of twists per unit length
- ▶ They can also be shielded, a grounded wire mesh encasing the wires, further reduces noise

# Twisted Pair

- ▶ The most common example of twisted pair is Category 5 (Cat 5) cable for ethernet
- ▶ This cable has 4 twisted pairs and can be terminated at a patch panel or with an RJ45 connector
- ▶ It can handle up to 1Gbps ethernet, Cat 6 has higher standards and can be used for 10Gbps ethernet



# Cables

- ▶ A channel that can transmit in both directions at the same time is called **full duplex**, two pairs of twisted pair can be used to construct this
- ▶ A channel that can transmit in one direction at a time, but can support both directions, just not simultaneously, is called **half duplex**
- ▶ This can be done with one twisted pair and some coordination
- ▶ A channel that can communicate in only one direction is called **simplex**
- ▶ This applies to all types of communications channels

# Coax

- ▶ In the case of wires the bits are sent as simple voltage levels:
  - ▶ 0 volts for 0
  - ▶ 5 volts for 1
- ▶ This limits the amount of information that can be sent down a wire
- ▶ Coax works in a different way and is more flexible
- ▶ Think about the coax that is used for cable TV, a single coax cable carries a wide range of TV channels
- ▶ This is done by allocating each channel a frequency band on the cable, its signal is modulated to that band

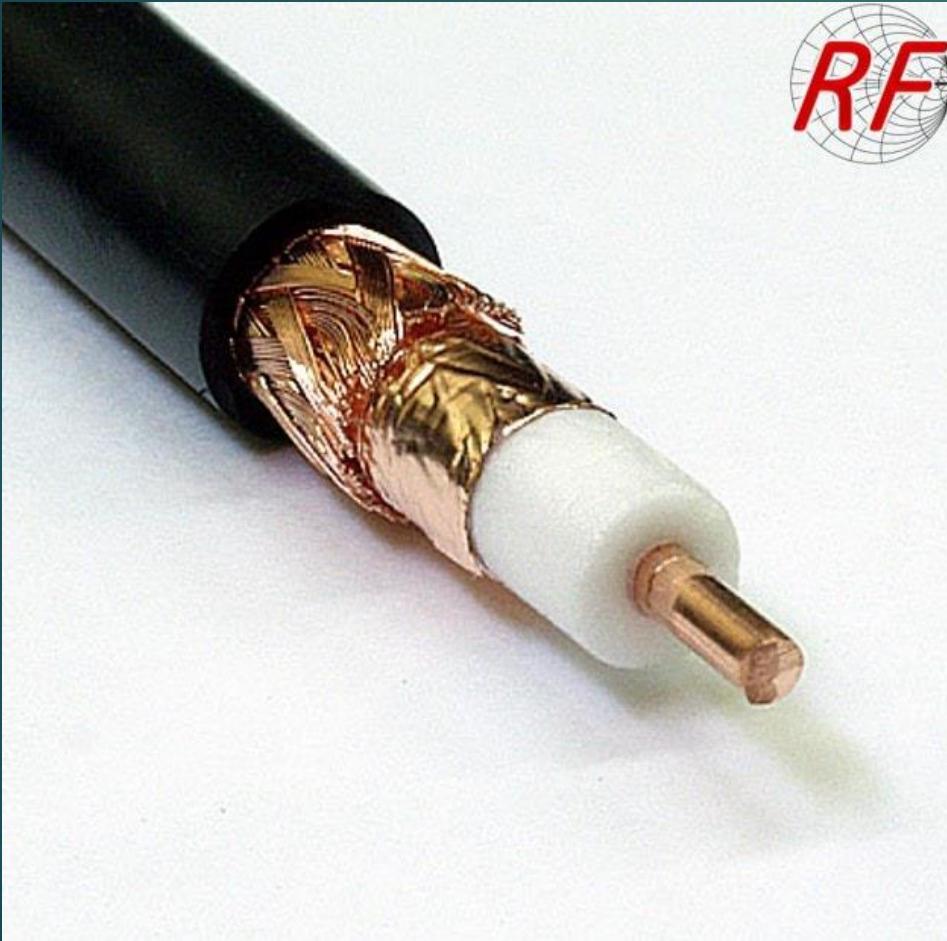
# Coax

- ▶ We can do the same thing with digital signals
- ▶ Frequency bands are allocated for different digital signals
- ▶ One band is for download and the other is for upload
- ▶ These bands can share the coax with TV channels
- ▶ The size of the frequency band determines the amount of information that can be sent
- ▶ It is not unusual to have the download frequency band larger than the upload one
- ▶ People are more likely to download than upload

# Coax

- ▶ Coax has a single copper wire in the center, this is surrounded by an insulator, this is covered by a copper shielding and finally this is wrapped in a plastic coating
- ▶ The copper shielding is grounded to reduce noise
- ▶ The wire is typically thicker than twisted pair, so it can support higher frequencies and much longer cable lengths
- ▶ There are many different kinds of coax, started with radio and TV
- ▶ You cannot mix and match, the whole system must use the same type of coax

# Coax



# Power Line

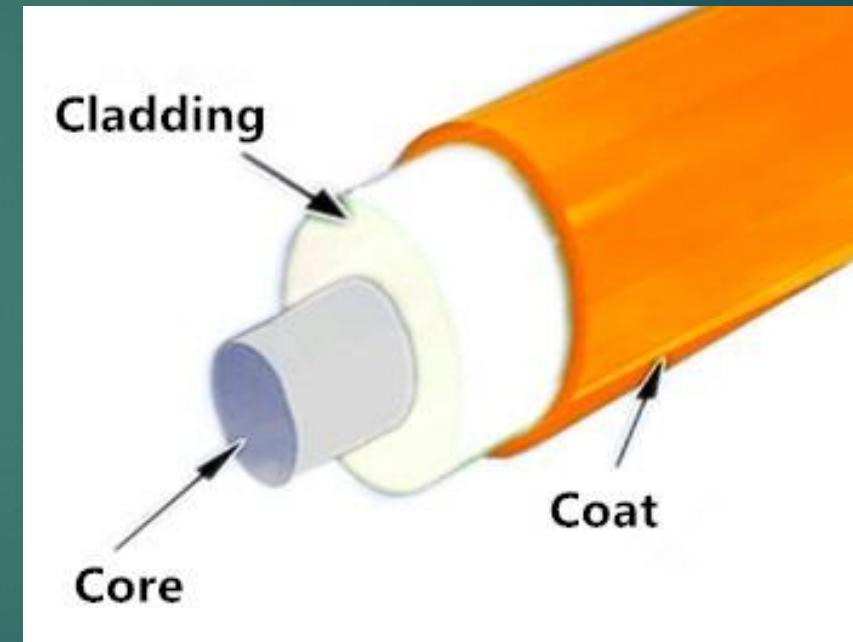
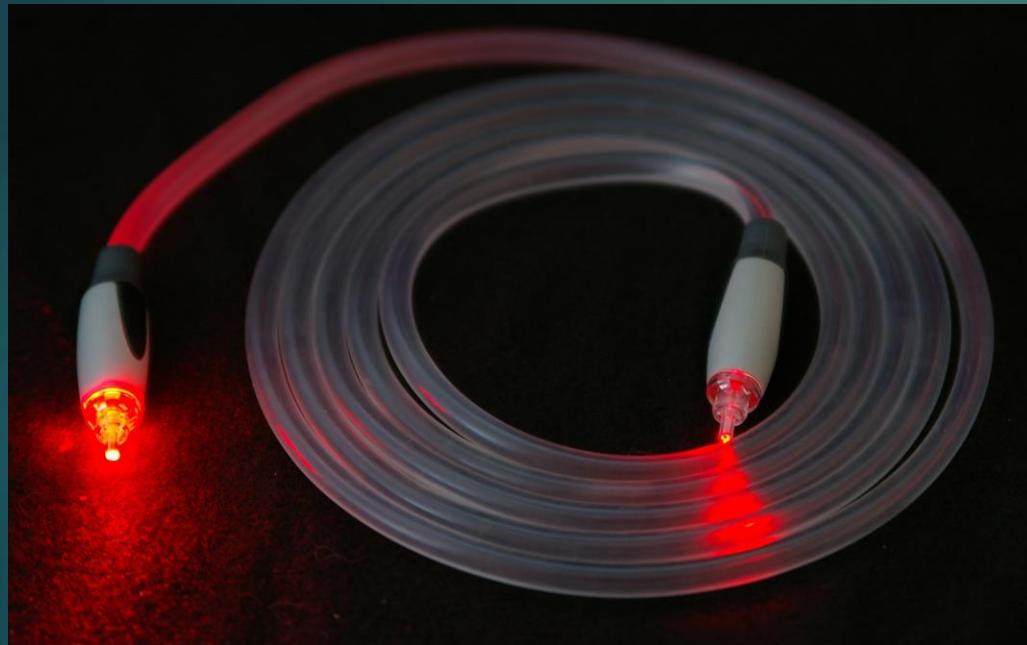
- ▶ We already have power lines in our house, copper wires, can use them for networking, can only be used within a single house
- ▶ There are some noise issues, but can reach 1Gbps rates
- ▶ Usually purchased in pairs, \$50 to \$100 per pair



# Fibre Optic

- ▶ Fibre optics is the main technology for long distances
- ▶ Light is used as the transmission medium, electrical interference isn't a problem
- ▶ The heart of a fibre optic cable is a very thin strand of glass
- ▶ This is surrounded by a layer of glass that has a different index of refraction than the fibre
- ▶ This is surrounded by a plastic coating
- ▶ Usually many fibres are packaged together with another coat around them

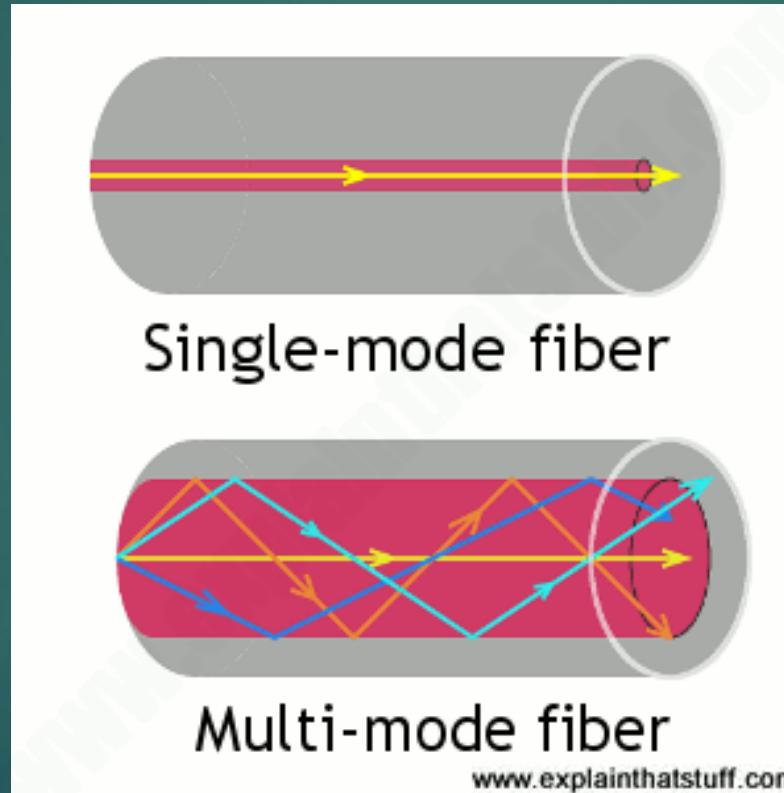
# Fibre Optic



# Fibre Optic

- ▶ There are two ways of using a fibre, both are based on a light source
- ▶ In multimode fibre the light bounces off the sides of the fibre, this is based on total internal refraction, there is a critical angle where the light bends enough to remain in the fibre
- ▶ Can have multiple light beams, as long as the angle is greater than this critical angle
- ▶ In single mode fibre, the fibre is only a few wavelengths wide so the light travels in a straight line

# Fibre Optic



# Fibre Optic

- ▶ At one end of the fibre is a light source, at the other end is a photodetector
- ▶ The individual bits are transmitted as light pulses
- ▶ Either a LED or laser is used as the light source
- ▶ The main bottleneck is the electronics, we currently only use a portion of the possible bandwidth, can be as much as 100Gbps
- ▶ There is very little signal lose in a fibre, it needs to have an amplifier every 50 to 100km
- ▶ Optical amplifiers will allow us the increase this bandwidth

# Fibre Optic

- ▶ Fibre optics has a number of significant advantages over copper
- ▶ It is much lighter and smaller than copper cables, this reduces the cost of installation significantly
- ▶ There is no light lose from a fibre, can't eavesdrop on the data being sent, more secure
- ▶ Since the cable is made of glass it can be easily damaged
- ▶ The fibres used for communications cannot be bent beyond a certain angle

# Wireless

- ▶ The technologies we've looked at require cables, there is the expense of laying cables
- ▶ Wireless avoids the need to lay cables, use radio waves to send the data, a cost advantage
- ▶ There are several significant problems with wireless
- ▶ First, at the frequencies available for data communications radio waves travel in a straight line, limits the distance they can travel due to the earth's curvature
- ▶ In addition, at higher frequencies the radio waves are absorbed by buildings and geography

# Wireless

- ▶ Second, radio waves are a shared resource, cannot do whatever you want, it will interfere with other users
- ▶ International agreements on how the radio spectrum can be used
- ▶ Governments have discovered that they can make money from allocating spectrum, cell phone spectrum is in the \$1B range
- ▶ Due to allocation there is limited bandwidth, so each channel cannot carry a lot of data
- ▶ Low power radio can be used on some frequency bands without license fees, this can be used for short range communications

# Wireless

- ▶ Microwaves have been used for decades for voice and data communications, need to install towers so this is largely a WAN technology
- ▶ For satellites line of sight is an advantage, many ground stations can use the same frequencies since they are beaming to different satellites, their signals won't interfere
- ▶ Problem: the atmosphere absorbs certain frequencies, and the frequency bands change during the day
- ▶ Solar flares are also a major problem that can disrupt communications

# Telephone System

- ▶ Telephone system has wires into the home, unfortunately they are only Cat 3 twisted pair
- ▶ Originally designed for voice communications, very small bandwidth, can support up to 56Kbps
- ▶ Telephones lines have filters that restrict the bandwidth
- ▶ The last mile is analog, this goes to a switch or concentrator that is digital, the rest of the system is now digital
- ▶ The last mile is our problem for digital communications, telephone companies didn't care until cable modems appeared

# Telephone System

- ▶ Introduced various forms of DSL (Digital Subscriber Lines)
- ▶ This requires a hardware upgrade at the concentrators to remove the bandwidth limit
- ▶ Special filtering done to remove bandwidth limit, the bandwidth depends upon the distance to the concentrator
- ▶ Small change done in the home to split voice from data
- ▶ Typical bandwidths of 3 to 5Mbps, this isn't too bad and is cost effective for the telephone companies

# Quantum Internet

- ▶ The next big jump in technology could be the quantum internet
- ▶ There are two main advantages:
  - ▶ Very high speed and bandwidth
  - ▶ Very secure, any attempt to intercept data results in its destruction
- ▶ Currently building research networks on MAN scale to determine feasibility
- ▶ Long time before this becomes a standard technology

# Physical Layer

- ▶ There are many ways of organizing the bits that are transmitted
- ▶ Arrange them to reduce the error rate
- ▶ Many coding schemes have been developed
- ▶ There is an attempt at this level to do some error detection and recovery, but this isn't a main concern
- ▶ At this point we just want to move the bits as fast as possible, leave it to the other layers to worry about errors

# Summary

- ▶ Examined some of the technologies that are used at the physical layer
- ▶ Have not gone into detail on any of the technologies, this is a large topic
- ▶ There are techniques for encoding bits at this level that reduce the error rate, you just need to know that they exist, you won't encounter them in practice

# CSCI 3310

# Data Link Layer

MARK GREEN  
ONTARIO TECH

# Introduction

- ▶ The physical layer just does bits, the upper layers want to work in terms of bytes and strings and other types of data
- ▶ The network layer wants to send packets of information, it needs something that will convert them into bits for the physical layer
- ▶ The data link layer is responsible for this packaging
- ▶ It takes a packet from the network layer and packages it into a **frame** that will be sent over the physical layer
- ▶ A frame consists of a **header** and a **trailer** that encloses the packet

# Frame



# Data Link Layer

- ▶ The main things performed by the data link layer:
  - ▶ Deal with transmission errors
  - ▶ Control the flow of data
  - ▶ Package data to be sent over the physical layer
- ▶ Again, the physical layer only knows about bits, it doesn't understand any larger unit of information
- ▶ The data link layer must provide this higher structure, impose structure on a stream of bits

# Data Link Layer

- ▶ At this point we are only interested in moving a frame from one end of the wire to another
- ▶ In the simplest case this is between two directly connect computers
- ▶ It could also be between a computer and a switch
- ▶ This is a bit restrictive, we are not dealing with routing packets over disconnected computers, this is the responsibility of the network layer
- ▶ The routing will be covered later in a higher layer

# Data Link Layer

- ▶ The data link layer can provide the following services to the network layer:
  - ▶ Unacknowledged connectionless service
  - ▶ Acknowledged connectionless service
  - ▶ Acknowledged connection-oriented service
- ▶ The bare minimum is the first service, but the other two can also be supported
- ▶ In the first case the source computer sends frames to the destination computer, assuming that they will get there okay

# Data Link Layer

- ▶ No attempt is made to detect errors or recover from them
- ▶ The frames are sent and we hope for the best
- ▶ This is okay over a physical layer with few errors, such as ethernet
- ▶ This doesn't work very well on physical layers, such as WiFi where there can be a significant number of errors
- ▶ In this case we need to know whether a frame got through
- ▶ This is accomplished by having the destination send an **acknowledgement**, a message that says the packet was received

# Data Link Layer

- ▶ The acknowledged connectionless service, provides this type of acknowledgement
- ▶ In its most basic form the sender transmits the frame and then waits for the receiver to send an acknowledgement
- ▶ It doesn't send another frame until the acknowledgement is received
- ▶ If the time required to receive the message is  $\tau$  the sender must wait at least  $2\tau$  for the acknowledgement, it should wait a bit longer in case the receiver is busy
- ▶ If it doesn't get the acknowledgement it sends the frame again

# Data Link Layer

- ▶ What happens if the acknowledgement is lost, but not the original frame?
- ▶ The frame gets sent twice, the receiver doesn't know that it has a duplicate!
- ▶ Clearly this works best over a LAN, where the transmission time is short
- ▶ The acknowledged connect-oriented service solves the problem of duplicated frames
- ▶ The two parties first establish a connection, involves several packet exchanges

# Data Link Layer

- ▶ To solve the duplicate frame problem each frame can be given a sequence number
- ▶ The acknowledgement contains the sequence number
- ▶ If the receiver gets two frames with the same sequence number it knows they are duplicated
- ▶ The sender no longer needs to wait for an acknowledgement, it can send the frames as fast as possible, the acknowledgement will tell it which frames made it through
- ▶ The receiver can order the frames based on their sequence number

# Data Link Layer

- ▶ Establishing the connection involves some network bandwidth, but after this can use the full physical layer bandwidth
- ▶ This approach works on much longer connections where the wait for the acknowledgement would slow things down too much
- ▶ The next problem that we get to is **framing**
- ▶ The physical layer sends bits, it doesn't have any way of identifying the start of a transmission or the boundary between two frames
- ▶ This is a problem that needs to be solved at the data link layer, one of the main things the data link layer does

# Framing

- ▶ There are many solutions to this problem, quite often use a combination of them
- ▶ Examine two approaches
- ▶ If the physical link is quiet and then starts sending bits, we know that it's the start of a frame
- ▶ A byte count can be included in the frame header, the number of bytes in the frame
- ▶ After this number of bytes have been seen, it must be the start of the next frame

# Framing

- ▶ This works on a physical layer with few errors
- ▶ But if there is an error in the byte count, or a few bits are dropped we will lose our place in the bit stream
- ▶ The only solution is to wait for the physical layer to be quiet again, so you can find the start of a frame
- ▶ A second approach is the use of **flag** bytes, a byte value that isn't used in the frame
- ▶ These flag bytes are placed at the beginning and end of a frame
- ▶ Common values are 01010101 and 10101010

# Framing

- ▶ When the receiver see a flag byte it knows that it is seeing the beginning or end of a frame
- ▶ This would work okay if all of our packets contained only text, we could choose a value that isn't a text character
- ▶ If we are sending binary data this doesn't work, in this case we use **byte stuffing**
- ▶ If the flag byte occurs in the data, we add an extra byte, called an **escape** before the byte, this increases the length of our frame by one byte

# Framing

- ▶ When the receiver sees the byte sequence “escape flag” it knows to remove the escape byte
- ▶ If the escape byte appears in the message we replace it with “escape escape”
- ▶ We use multiple flag bytes at the start and end of a frame, just in case a flag byte is lost
- ▶ We can combine this with a byte count if the physical layer is noisy, this guards against escape bytes being lost

# Flow Control

- ▶ If the sender sends frames faster than the receiver can process them frames will be lost, and the sender will need to retransmit them
- ▶ This is a waste of network bandwidth, a problem called **flow control**
- ▶ An acknowledged connectionless service handles this automatically, it won't send the next frame until it gets an acknowledgement from the previous one
- ▶ Straight unacknowledged connectionless services have no way of knowing this problem is occurring
- ▶ The sender will just keep blasting frames, whether the receiver can handle them or not

# Flow Control

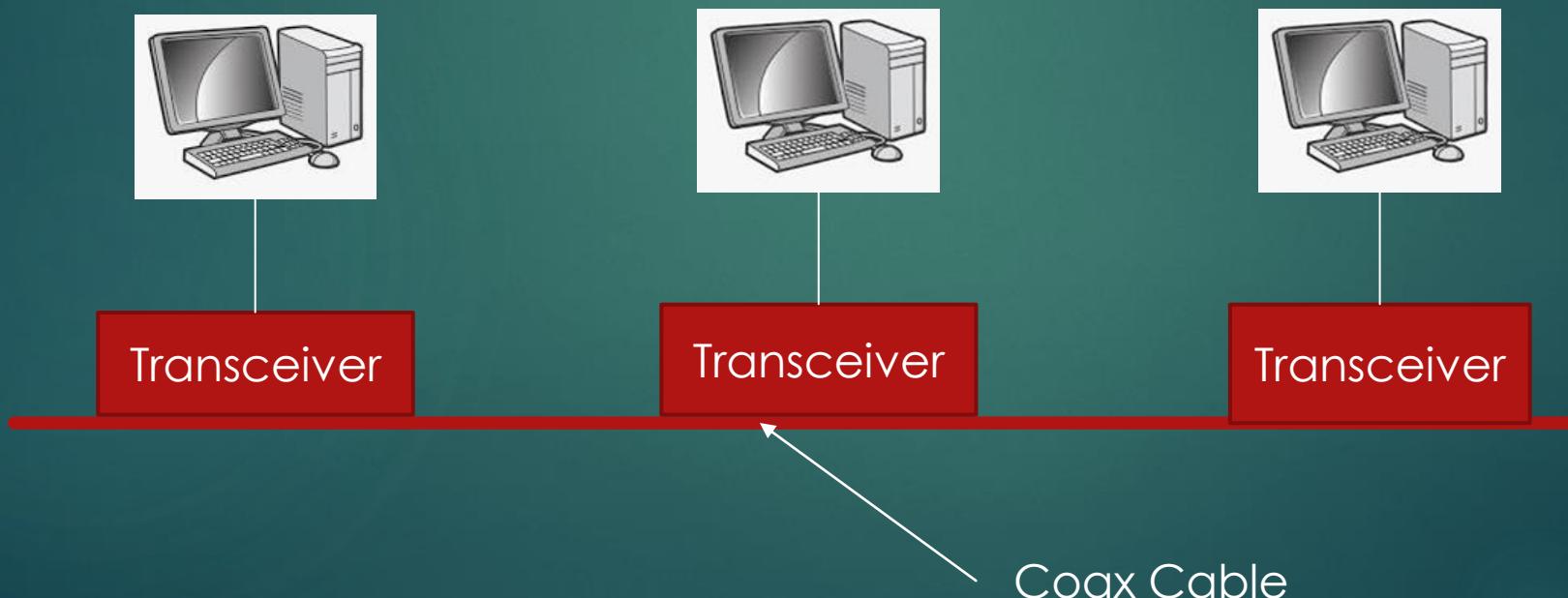
- ▶ We could leave it to a higher level layer to detect the packet loss
- ▶ The receiver could periodically send a frame to the sender telling it how busy it is, or send a frame requesting the sender to stop
- ▶ A connection-oriented service could send this information in an acknowledgement, saving sending an extra frame

# Ethernet

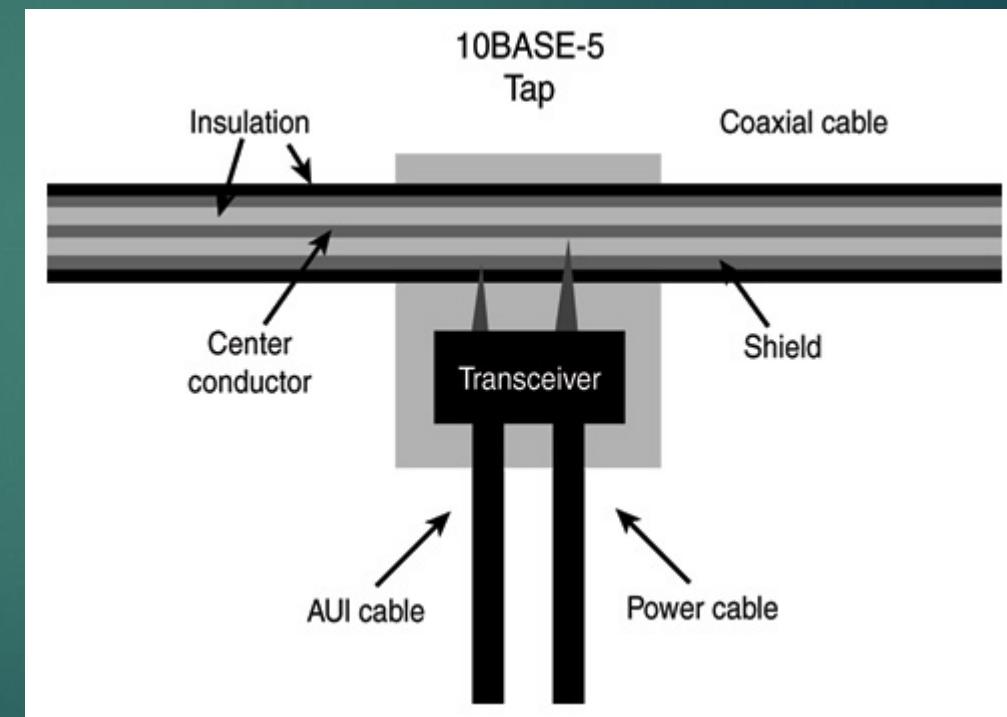
- ▶ Ethernet is a good example of a data link protocol, it is also quite common, use it as an example
- ▶ There are two types of ethernet, classical ethernet and switched ethernet
- ▶ Will start with classical ethernet, though it is not used now
- ▶ Developed in 1976 by Bob Metcalfe and David Boggs at Xerox PARC, the first LAN
- ▶ Initial version was based on coax able and ran at 3Mbps, this was fast by 1976 standards
- ▶ Bob Metcalfe went on to form 3com

# Ethernet

- ▶ DEC, Intel and Xerox worked together in 1978 to produce the first ethernet standard, called the DIX standard, this was at 10Mbps
- ▶ Based on a single coax cable that all the computer tapped into, using a vampire tap



# Vampire Tap



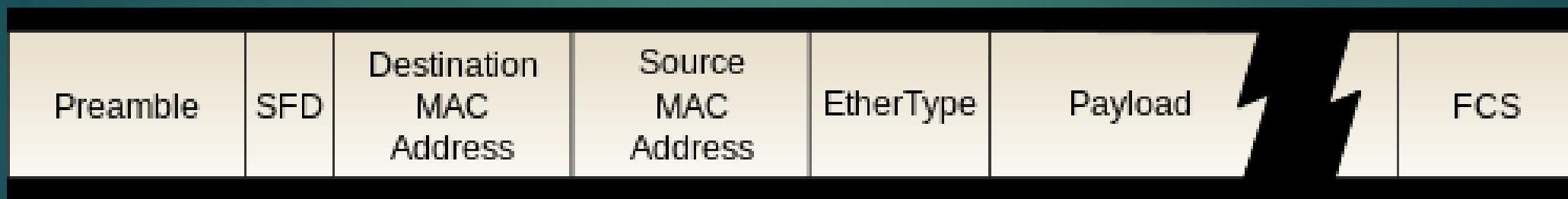
# Ethernet

- ▶ The coax cable can be at least 500 metres long and at most 4 repeaters, each repeater decreased the maximum cable length
- ▶ There were a number of restrictions on how the cable could be laid out
- ▶ Later IEEE 802.3 appeared which was almost identical to DIX and they could interoperate
- ▶ This was called thick ethernet due to the diameter of the cable
- ▶ Later a thin ethernet standard appeared, based on a thinner cable, but with reduced length

# Ethernet

- ▶ An ethernet frame is shown on the next slide
- ▶ There is a 7 byte preamble at the start of the frame, the byte values are 10101010, this is followed by SFD that marks the end of the preamble, a single byte with value 10101011
- ▶ The preamble is used to mark the start of the frame and synchronize the clocks on the sender and receiver
- ▶ This is followed by the destination and source addresses, both of which are 6 bytes long, called **MAC addresses**, the address is unique across all ethernet interfaces

# Ethernet Frame



# Ethernet

- ▶ If the first bit of the address is 0, it is a unique interface
- ▶ If the first bit is 1, the address is used for multicast or broadcast
- ▶ The first three bytes of the address is the **Organizational Unique Identifier** or **OUI**, this is assigned to the manufacturer of the interface
- ▶ The remaining bits of the address are assigned by the manufacturer, they are unique for each interface produced by that manufacturer
- ▶ The next two bytes is where the two standards differ:
  - ▶ In DIX it is the type of frame
  - ▶ In 802.3 it is the length of the frame

# Ethernet

- ▶ Since the length must be less than 1536, and type values are greater than this, the two standard don't conflict
- ▶ Up to 1500 bytes of data follow
- ▶ An ethernet frame must be at least 64 bytes long, so there can be up to 46 bytes of padding to reach this minimum length
- ▶ This is followed by a 4 byte checksum, which is used to detect errors in the frame
- ▶ The coax cable is shared by all the stations (interfaces) on the network, so there must be some way of sharing the bandwidth that isn't centralized

# Ethernet

- ▶ This is done in the following way:
  - ▶ The sender waits until the coax is quiet
  - ▶ It then starts transmitting the frame
  - ▶ At the same time it's listening to the coax, if it's the only one transmitting it will just see its own frame
  - ▶ If another sender starts transmitting, since the original frame hadn't reach it yet, there will be a collision
  - ▶ Both frames will turn into garbage and the power on the coax will be higher
  - ▶ Both senders will see this and stop sending

# Ethernet

- ▶ This is why we need a minimum frame length
- ▶ Consider two stations are opposite ends of the coax
- ▶ Station A starts transmitting at time zero, and it takes  $\tau$  to reach the opposite end of the coax
- ▶ At time  $\tau - \epsilon$  station B starts transmitting, since A's frame hasn't reached it yet
- ▶ Station B soon detects the collision and sends a 48 bit noise burst to warn other stations
- ▶ This burst arrives at station A at time  $2\tau$ , in order for station A to detect a collision it must still be transmitting, otherwise it doesn't know that its frame that caused the collision

# Ethernet

- ▶ Given the maximum length of an ethernet segment this results in a 64 byte frame size
- ▶ Now that we have a collision what do the two stations do?
- ▶ They stop transmission and wait to transmit again
- ▶ We divide our timeline into slots of length  $2\tau$ , the maximum time to detect a collision
- ▶ On the first round both stations randomly select a delay of 0 or 1 slot times
- ▶ If they still detect a collision, they randomly select a delay of 0, 1, 2 or 3 slot times

# Ethernet

- ▶ On the  $i^{\text{th}}$  round they randomly select a delay from 0 to  $2^{i-1}$  slot times
- ▶ This continues up to a delay of 1023 slot times
- ▶ If after 16 tries they are still getting collisions they declare that the frame can't be delivered
- ▶ This algorithm is called **binary exponential backoff**
- ▶ This works surprisingly well in practice for 10Mbps ethernet
- ▶ Unfortunately as the bandwidth increases the performance get worse, so we are essentially limited to 10Mbps

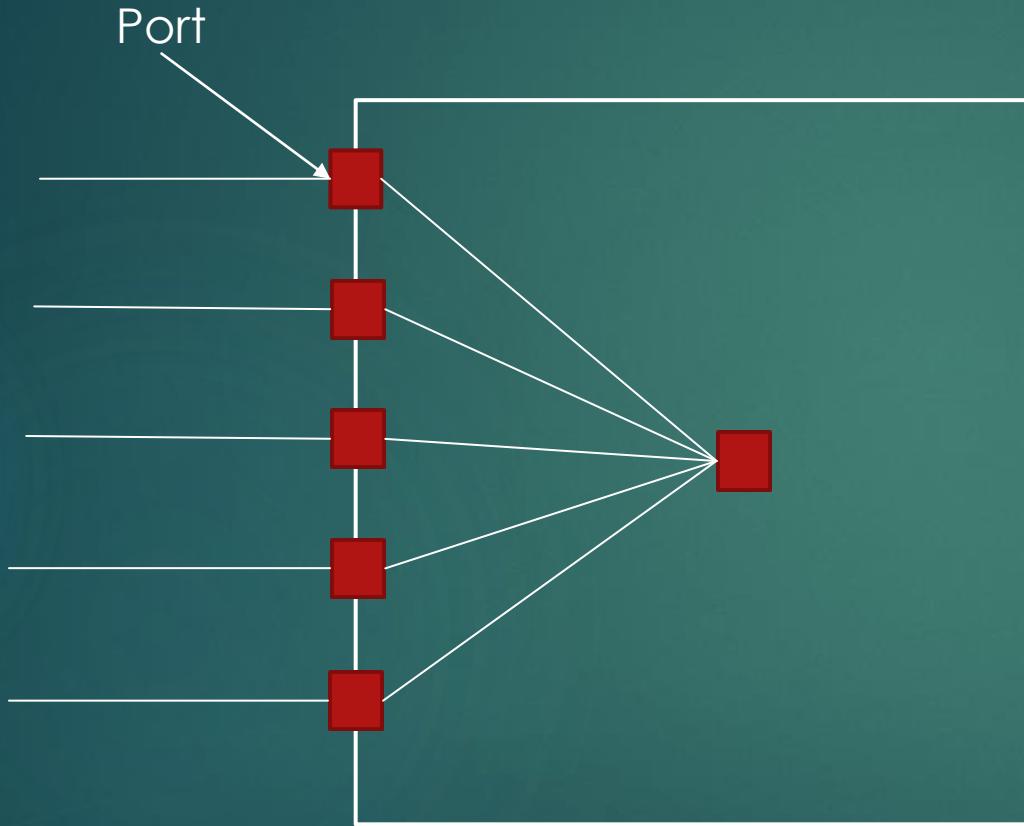
# Switched Ethernet

- ▶ Running coax and connecting computers to it was a lot of work and if something happened to one of the taps the whole network would die
- ▶ System administrators could spend many hours trying to track down the tap that caused the problem, quite often it was hidden in the ceiling making it hard to inspect
- ▶ To solve this problem coax was replaced by twisted pair, the standard ethernet cable that we now know
- ▶ To make it behave like a single coax cable the ethernet hub was invented

# Switched Ethernet

- ▶ The basic idea behind an ethernet hub is that a number of cables were plugged into to it, typically between 5 and 48, using standard RJ45 connectors
- ▶ Internally all of the cables were connected together so they acted like a single length of coax
- ▶ The whole system behaved in essentially the same way, it was just a lot easier to manage all of the cables
- ▶ There were still collisions and a limited bandwidth
- ▶ This is now obsolete technology

# Ethernet Hub



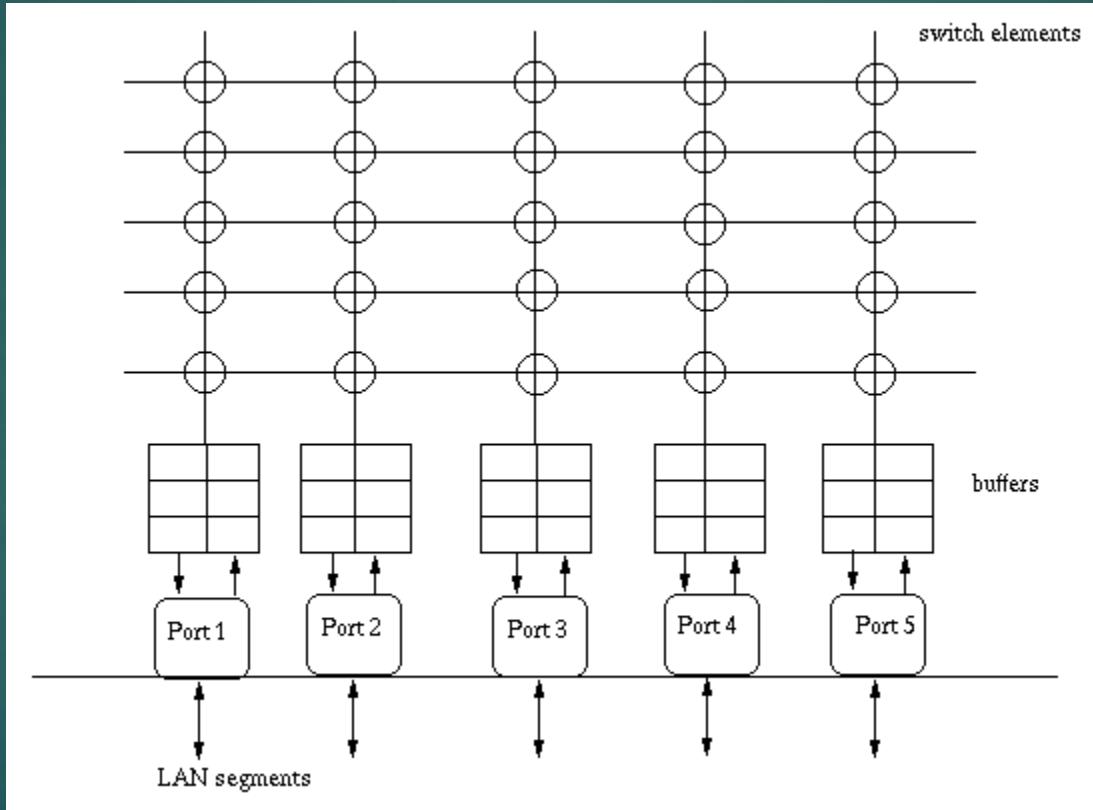
# Switched Ethernet

- ▶ There were two problems with a hub:
  - ▶ It was using only one pair out of the four in a Cat 5 cable, this was basically a half duplex channel
  - ▶ The medium was still being shared by all the computers, only one could transmit at a time
- ▶ We could use two pairs in the Cat 5 cable to produce a full duplex channel
- ▶ It would be more efficient to directly connect the two computers that are communicating and not share the medium

# Switched Ethernet

- ▶ This is the idea behind switched ethernet
- ▶ Instead of connecting all the cables together they are connected to a matrix switch
- ▶ If we have 5 cables, they are connected to a 5x5 switch
- ▶ When computer 1 is talking to computer 5, a direct connection is set up between them
- ▶ At the same time computer 2 can be talking to computer 3
- ▶ That is, we can have multiple transmissions occurring at the same time, they can all go at full ethernet speed

# Ethernet Switch



# Switched Ethernet

- ▶ If we use two pairs in the cable a computer can be receiving and transmitting at the same time
- ▶ This gives the LAN a much higher bandwidth
- ▶ We can now go to 100Mbps with the same cables
- ▶ Switches are intelligent, so they can detect whether the connection is at 10Mbps or 100Mbps
- ▶ Thus, older ethernet interfaces can be used with newer ones, the switch does the conversion between speeds
- ▶ It is now standard to add 1Gbps to switches, again we can have all three speeds on the same LAN

# Switched Ethernet

- ▶ This approach made it possible to easily grow a LAN with new technology, you didn't have to replace all the existing hardware
- ▶ How does a switch know how to route a frame, it needs to know which computer is attached to which port
- ▶ We want to do this without human intervention, just plug in the cables and let it work, no management required
- ▶ There are several ways that we can do this, the simplest way of doing this is a technique called **backward learning**
- ▶ This technique is also very robust

# Switched Ethernet

- ▶ Backward learning is based on examining the ethernet frame, note that each frame has a source address as well as a destination address
- ▶ When a frame arrives at a port the switch examines the source address of the frame, this must be the address of the computer at the other end of the cable
- ▶ This is stored in a hash table inside the switch
- ▶ When the switch receives a frame it extracts the destination address and looks it up in the hash table
- ▶ If the address is in the hash table, it knows which port to send the frame to

# Switched Ethernet

- ▶ If the address is not in the hash table one of two thing is happening:
  - ▶ It belongs to a port that we don't have an address for yet
  - ▶ It belongs to a computer that isn't on this LAN segment
- ▶ If we know the address for each port, we know that we have the second case and we send the frame on an outside link, to another switch
- ▶ If there are ports that we haven't identified we do a **flood**, we send the frame to all of the unidentified ports
- ▶ That way the frame will get to its destination, even though we are wasting bandwidth

# Switched Ethernet

- ▶ With RJ45 we can easily unplug a computer and move it to a different port, now we are sending frames to the wrong port
- ▶ We want to recover from this automatically
- ▶ The switch will periodically clear its hash table and then rebuild it
- ▶ This will happen every few minutes
- ▶ With reasonable network load it should only take a few seconds to rebuild the hash table
- ▶ If a computer isn't used frequently, it doesn't really matter that it misses a hash table rebuild

# Switched Ethernet

- ▶ There is another problem that we need to deal with, what happens when several computers are transmitting to the same receiver at the same time?
- ▶ The switch has a limited amount of buffer space where it can save frames waiting to be switched
- ▶ This is one of the reasons for having a reasonably short frame length
- ▶ Most switches don't have a lot of buffer space, they depend on short bursts of traffic
- ▶ If the buffer is full, the frame is dropped

# Switched Ethernet

- ▶ Most low end switches are called unmanaged, you plug them in and they just work
- ▶ A managed switch has more features, accessed through a user interface or the SNMP protocol
- ▶ System administrator can turn ports on and off, determine the ports that forward frames to the rest of the Internet
- ▶ Can also be used to set of VLANs, or virtual LANs, the ability to break up the ports into several logical LANs
- ▶ A high end switch can have 48 ports and they can be stacked to produce a larger switch

# Summary

- ▶ Examined some of the issues in the data link layer
- ▶ A high level view, didn't get down into the technical details
- ▶ Examined ethernet as an example:
  - ▶ Classical ethernet
  - ▶ Switched ethernet

# CSCI 3310

# Network Layer

MARK GREEN  
ONTARIO TECH

# Introduction

- ▶ The data link layer worked at the LAN level, it was responsible for getting packets from one end of a wire to the other
- ▶ The network layer works at the whole network level, it gets packets from one site to another regardless of where they are located on the network
- ▶ The main piece of hardware that is responsible for this is called a **router**
- ▶ Like a switch, a router has a number of ports and a matrix switch that connects these ports
- ▶ It also has buffer space to hold packets waiting for delivery

# Introduction

- ▶ Routers are more sophisticated than switches, they need to know how to get the packet from one end of the network to the other
- ▶ This is a much harder job, since the router needs to have an idea of the network structure, so it knows how to route the packet in the most efficient way
- ▶ The basic technique that is used by a router is called **store-and-forward**
- ▶ The router collects the packet in memory as it is received, it will check the packet for validity, determine its destination and then send it to the next router on its path

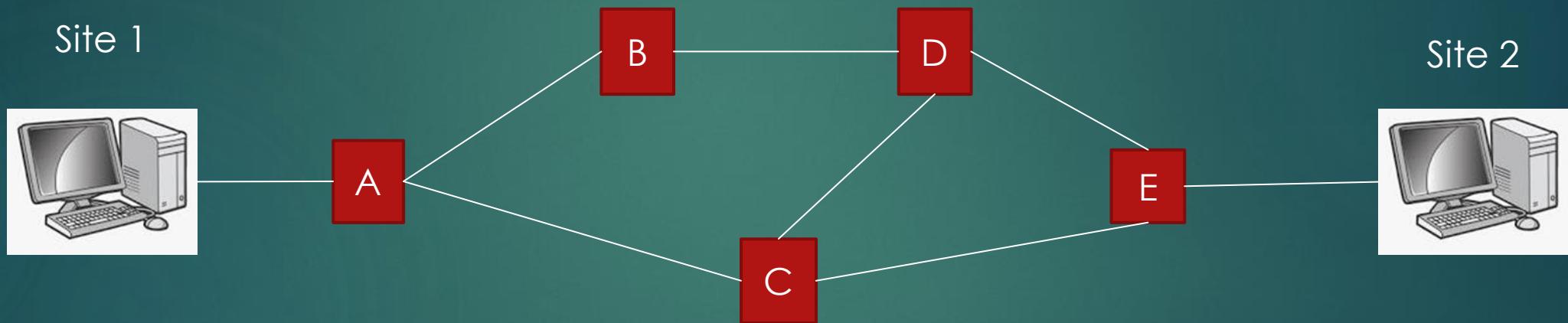
# Network Structure

- ▶ We can view a network as a graph:
  - ▶ The nodes of the graph are the routers
  - ▶ The edges are the links between the routers
- ▶ A LAN connects to one of the ports on a router
- ▶ A router must at least know all the routers its connect to, quite often it will know more than that
- ▶ The whole routing problems is complicated by the fact that the Internet is very large, can't know the entire graph
- ▶ In addition, it is always changing

# Network Structure

- ▶ The following slide show a small network, there are 5 routers and the connections between them
- ▶ We want to send a packet from site 1 to site 2
- ▶ There are multiple routes through the network that we could use:
  - ▶ A -> B -> D -> E
  - ▶ A -> C -> E
  - ▶ A -> B ->D -> C -> E
- ▶ The **routing** problem is which one of these routes do we use?
- ▶ The choice might depend upon which links are up and traffic levels

# Network Structure



# Services

- ▶ There are two types of service that the network layer could provide:
  - ▶ Connectionless
  - ▶ Connection-oriented
- ▶ There have been lots of debate about which is better, it has gone back and forth over the decades
- ▶ My view and that of others, is that you need both
- ▶ The dominate service now is connectionless, but connection-oriented services are making a comeback in some applications
- ▶ In general routers should support both

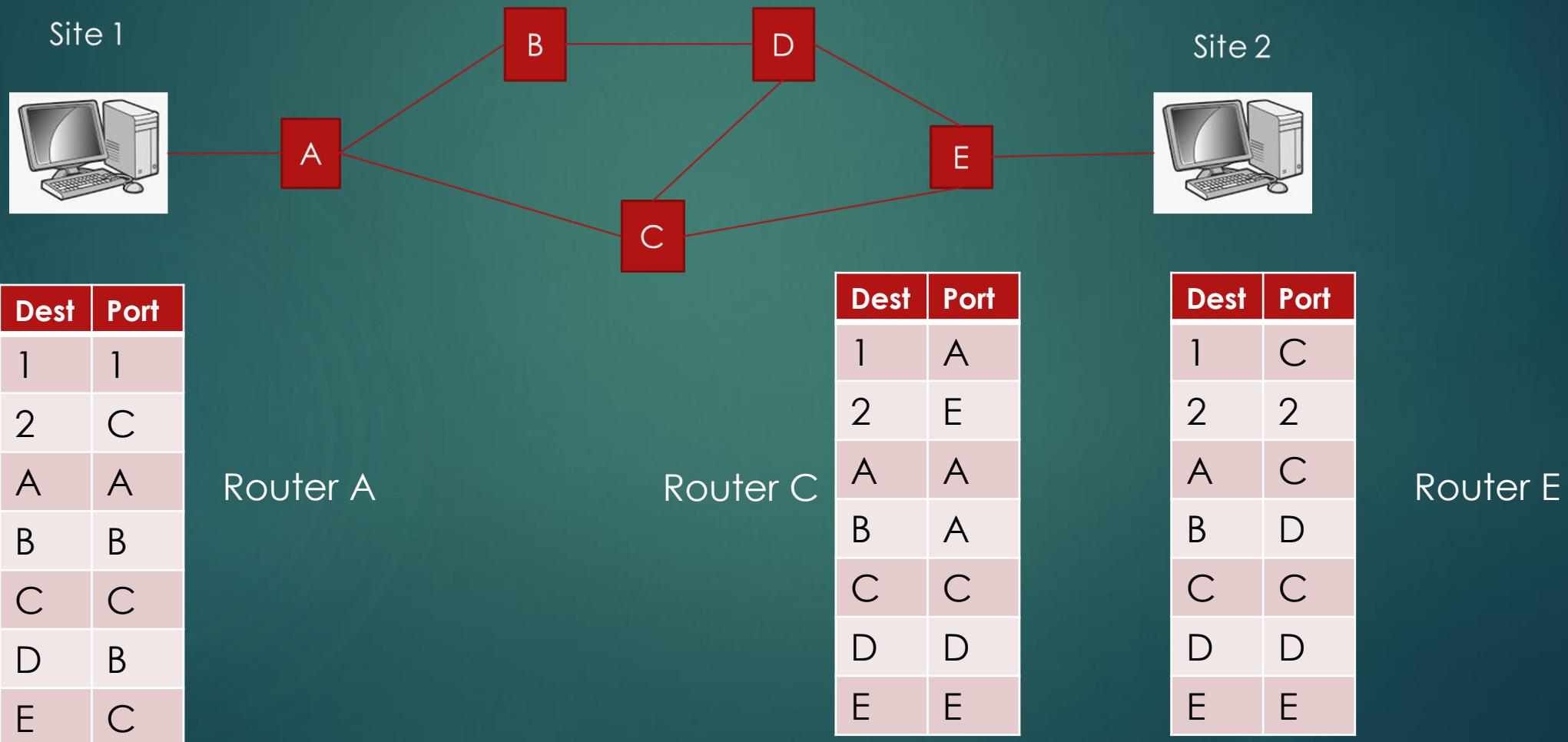
# Connectionless Service

- ▶ In a connectionless service the packets are routed independently of each other
- ▶ The packets are called datagrams, and the network is called a datagram network
- ▶ To see how this works go back to our example network
- ▶ A computer at site 1 wants to send data to a computer at site 2
- ▶ The network layer gets this data from the transport layer, which includes the destination address
- ▶ It now breaks it up into a sequence of packets for the data link layer to go to router A

# Connectionless Service

- ▶ Router A will get these packets, on the port coming from site 1
- ▶ It will examine the destination address and decide where to send the packets, this is a packet by packet decision
- ▶ Router A can only send packets to routers B and C, it has direct connections to them
- ▶ At some point a **routing algorithm** has decided which path is best and provided A with a table
- ▶ This table has two columns, the first is the destination address, the second is the port to send the packet on

# Connectionless Service



# Connectionless Service

- ▶ When a packet arrives at router A, it examines its routing table and finds that for site 2 the packet is forwarded to Router C
- ▶ When the packet gets to Router C, it examines its routing table and finds that the packet is forwarded to Router E
- ▶ When it gets to Router E its routing table sends it to the port for site 2 and the packet is delivered
- ▶ At each router there is a simple table lookup to determine the port that the packet should be sent on
- ▶ A hash table or special purpose hardware is used, since this must be done quickly

# Connectionless Service

- ▶ Now, mid way through transmitting the packets someone with a backhoe cuts through the cable between router A and C
- ▶ We can no longer use the route that we have been using
- ▶ The routing algorithm quickly determines that Router E can be reached by going through routers B and D
- ▶ The table in Router A is then changed to point to Router B instead of Router C
- ▶ This is the advantage of connectionless service, we can quickly change the route to account for failures or high traffic levels

# Connection-Oriented Service

- ▶ Connection-oriented people argue that connectionless services need to make routing decisions on a packet by packet basis
- ▶ If you are sending a large number of packets between two sites it makes sense to do the routing once and create a **virtual circuit** for all the packets to follow
- ▶ Example: site 1 is a video streaming service and site 2 is a home viewing videos, there will be a large number of packets sent over an extended period of time
- ▶ It makes sense to establish a connection, the set of routers to be followed at the beginning and then not worry about it

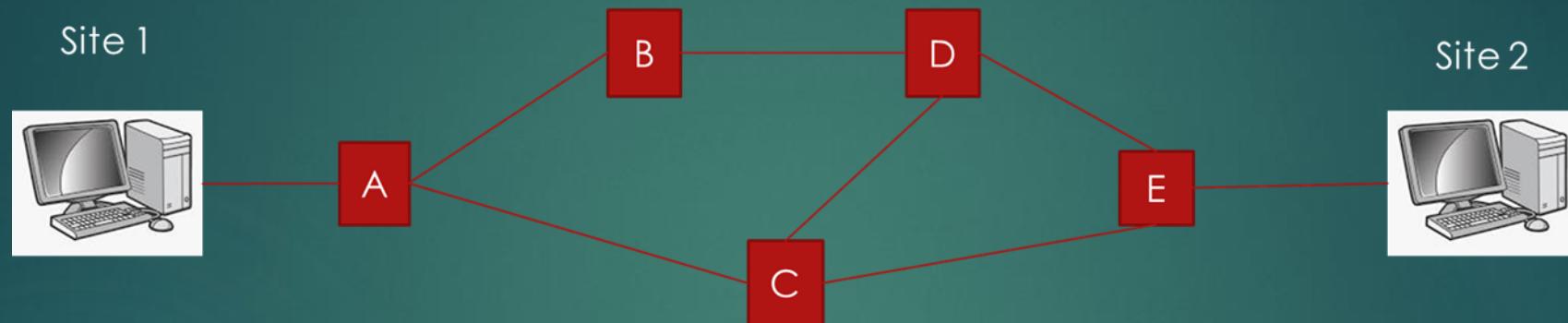
# Connection-Oriented Service

- ▶ We need some way of identifying connections, we do this by assigning an identifier to each connection
- ▶ Since we have a distributed system we can't always assign a connection identifier that is constant from source to destination
- ▶ When we get to a router it may already have that connection identifier assigned, so it needs to be able to change the connection identifier for the next hop
- ▶ This produces a more complicated table structure, with four columns:
  - ▶ The incoming port and connection number
  - ▶ The outgoing port and connection number

# Connection-Oriented Service

- ▶ Returning to our example network, site 1 wants to establish a connection to site 2
- ▶ This is its first connection, so we use the connection number 1
- ▶ If we examine the table for Router A, we get connection 1 from site 1 and then send it as connection 1 to Router C
- ▶ It turns out the Router C already has a connection number 1, its coming from Router D
- ▶ If it uses connection number 1 for the connection from Router A the rest of the network will be confused, since there will be two separate connections coming from Router C labelled connection 1

# Connection-Oriented Service



Router A

| In | # | Out | # |
|----|---|-----|---|
| 1  | 1 | C   | 1 |
| 1  | 2 | B   | 2 |

Router C

| In | # | Out | # |
|----|---|-----|---|
| A  | 1 | E   | 2 |
| D  | 1 | E   | 1 |

Router E

| In | # | Out | # |
|----|---|-----|---|
| C  | 2 | 2   | 2 |
| C  | 2 | 2   | 1 |

# Connection-Oriented Service

- ▶ To solve this problem Router C changes the connection number from Router A to 2, and then sends it on its way to Router E
- ▶ At Router E we just pass the packets on the site 2 with the new connection number
- ▶ Once the connection is established the routing algorithm isn't required
- ▶ In addition the routers can allocate resources for the connection
- ▶ They can guarantee a certain level of service, if a router is over committed it can refuse a connection
- ▶ If our backhoe operator gets busy again and breaks one of the fibres the connection is down, it can't recover

# Comparison

- ▶ Both services have their advantages and disadvantages
- ▶ Connectionless works for a lot of Internet services, there are few packets exchanged, so the overhead of a connect-oriented service isn't worthwhile
- ▶ The resiliency of the network is more important
- ▶ If a large number of packets are transferred between sites a connection-oriented service is better
- ▶ Can guarantee the bandwidth by dedicated router resources
- ▶ But error recovery is more difficult
- ▶ To set up a connection-oriented service we need something like a connectionless service to establish the connection

# Comparison

- ▶ There is also an economic consideration
- ▶ For network operators it's much easier to charge for connection-oriented services
- ▶ It's easy to account for network bandwidth and time of connection and develop a charging scheme
- ▶ With connectionless service its hard to track every packet and charge for it, this is an impossible accounting problems
- ▶ Thus, the network operators prefer connection-oriented services and will push for them

# Routing Algorithms

- ▶ This is a big topic, we will only look at part of it to give a feeling for what's going on
- ▶ The problem is complicated by the fact that the Internet is large and constantly changing
- ▶ This means that any realistic routing algorithm can't have perfect knowledge of all the computers on the Internet
- ▶ It must also be resilient, in the sense that routers will come and go due to various hardware issues
- ▶ Must be able to recompute routes when this happens

# Routing Algorithms

- ▶ We would like our routing algorithms to be correct, the packet is always delivered to its destination
- ▶ Also would like simplicity, must be implemented on all routers, complicated algorithms encourage bugs
- ▶ The routes should be stable, don't want them changing all the time
- ▶ Determine a good route and stay with it, otherwise packets might get lost
- ▶ Also want it to be fair and efficient
- ▶ The problem is, what do we mean by efficient

# Routing Algorithms

- ▶ Again go back to our graph view of the network
- ▶ Each edge is labelled by a cost or distance, usually call it a distance in the algorithms, but it rarely is
- ▶ The best route is the one with the lowest cost:
  - ▶ The time required to deliver the packet
  - ▶ The monetary cost
  - ▶ The number of hops through routers
  - ▶ The possibility of network congestion
- ▶ The cost is usually a combination of several of these

# Routing Algorithms

- ▶ The **optimality principle** assists with several of our algorithms:
  - ▶ Consider an optimal path from router I to router K
  - ▶ Consider router J that lies on this optimal path
  - ▶ The optimal path from J to K must also lie along this optimal path
- ▶ Put another way, if there is an optimal path from I to J and an optimal path from J to K, we just need to join these two paths to get an optimal path from I to K
- ▶ This suggests that we can use local information at a router to build an optimal path

# Routing Algorithms

- ▶ A consequence of this is the **sink tree**
- ▶ Consider a single destination router and all the possible source routers
- ▶ Construct a path from each of the sources to the destination
- ▶ This results in a tree with the destination at the root
- ▶ This tree may not be unique, if there are several paths with the same cost
- ▶ If we include all of these paths we get a directed acyclic graph
- ▶ There will be no loops in the sink tree, so packets will not go around in circles

# Routing Algorithms

- ▶ For the time being assume we have perfect knowledge of the network structure, all the routers, their connections and the costs of these connections
- ▶ Given a source router and an destination router we can use one of the shortest path graph algorithms
- ▶ There are quite a few of them, this is the subject of an algorithms course
- ▶ But, we can construct optimal paths, which is a good sign
- ▶ In the worst case we will need to repeat this for every pair of routers
- ▶ This would work if our network changes slowly

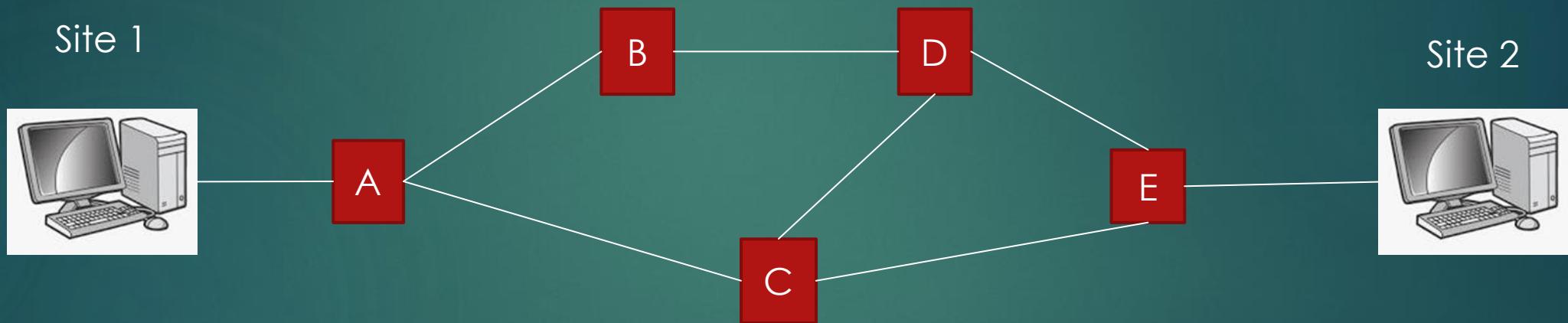
# Flooding

- ▶ A brute force approach is to use flooding like we did at the data link layer
- ▶ To flood a packet it is sent out on all of a router's ports except the one it originally arrived on
- ▶ This will basically send the packet to all of the routers on the network and the final destination
- ▶ It will follow all the possible paths to the destination, so it will find the shortest path, but at a great waste of bandwidth
- ▶ But this is more complicated than the LAN case where there is only one switch

# Flooding

- ▶ If we aren't careful the packet will circulate forever in the network
- ▶ Consider our small example network again, on the next slide
- ▶ If we start flooding from router C, the packet will go to E, then to D, and then back to C again, where it will again be sent out
- ▶ Eventually the network will be full of flooding packets
- ▶ There is a solution to this, the source of the packet assigns a sequence number to the packet
- ▶ When a router receives the packet it records the source of the packet and its sequence number
- ▶ If it sees a packet with the same source and sequence number again it won't forward it

# Network Structure



# Flooding

- ▶ There is a problem here, we will eventually need a large table to store all of these sequence numbers
- ▶ If we've seen all the sequence numbers (from a given source) less than k, all we need to do is store k
- ▶ If we see a sequence number less than k, we know that it can be dropped
- ▶ Rarely use flooding as a routing algorithm, but it is part of other algorithms
- ▶ A way of determining the network structure

# Distance Vector Routing

- ▶ This algorithm is based on local information available at each router
- ▶ It uses this information to construct a routing table
- ▶ This table has one entry for each router in the network, this entry has the port to use to reach that router and the estimated time to reach it
- ▶ While it is called distance vector it is usually based on time
- ▶ The router can determine the time it takes to get to each neighbour by sending an echo packet
- ▶ This packet has a time stamp, and the neighbour returns it immediately, from this we can estimate the time to the neighbour

# Distance Vector Routing

- ▶ Periodically each router sends its routing table to all of its neighbours
- ▶ It will also receive a routing table from all of its neighbours
- ▶ It will use this information to build a new routing table in the following way:
  - ▶ The time required to reach router  $i$  is  $m_i$
  - ▶ The routing table for router  $i$  has entries  $t_{ij}$  the time to go from router  $i$  to router  $j$
  - ▶ For each value of  $j$  find the minimum of  $m_i + t_{ij}$  for all routers  $i$
  - ▶ Say that this minimum occurs at router  $k$ , this is our optimal route

# Distance Vector Routing

- ▶ With this information the new routing table entry for router j is port k with time  $m_k + t_{kj}$
- ▶ This is repeated over all the table entries
- ▶ Note, that the old routing table isn't used at the current router, but is used at all its neighbours
- ▶ This algorithm has a number of important advantages
- ▶ It doesn't require a lot of storage or processing, each router has a small number of neighbours
- ▶ When a new router appears it is quickly incorporated into all the tables, it converges quickly

# Distance Vector Routing

- ▶ But, there is one major problem, if a router becomes disconnected it stays in the routing tables for a long time
- ▶ To see why this is the case consider router A that is connected to router B, all the other routers in the system know that they can get to router A through router B
- ▶ Now the link between the routers fails, router B cannot directly reach router A, but all of B's neighbours know they can reach it through B
- ▶ When it comes time to update the routing tables B will still think it can reach A, say through its neighbour C
- ▶ But, C assumes it can get to A through B

# Distance Vector Routing

- ▶ The problem comes from each router only knowing one hop of the route, router B doesn't know that all the other routers think that the shortest route to A is through B
- ▶ All B knows is that there is another route, but of course it will take longer
- ▶ Each time the routing tables are updated the time to A will get longer, this is called the **count-to-infinity problem**
- ▶ If this continues the network will never discover that router A can't be reached and packets will go in circles

# Distance Vector Routing

- ▶ A solution to this problem is to set a maximum time for packet transmission
- ▶ Once this time is exceeded in one of the routing tables that router is declared to be down
- ▶ In summary, new routers are discovered very quickly, but failed ones take a long time to detect
- ▶ This algorithm has been used on the Internet in the RIP protocol

# Link State Routing

- ▶ The link state routing algorithm solves the dead router problem and is now currently used on the Internet
- ▶ The idea behind this algorithm is for each router to build a complete graph of the network and then use the shortest path algorithm to construct the routing tables
- ▶ The key problem here is constructing the graph, since each router initially only knows about its neighbours
- ▶ This algorithm assumes that each router has a network unique name, since the algorithm builds up a map of the whole network

# Link State Routing

- ▶ The first step in the algorithm is to send an HELLO packet to each of the routers connected to the current router
- ▶ This packet returns the name of the router on the other end of the line
- ▶ The second step is to determine the cost associated with each of the neighbours
- ▶ This could be the inverse of the network speed, or the measured time to reach the neighbour
- ▶ At this point the router knows about its local neighbourhood

# Link State Routing

- ▶ The third step is to package all of this information to send to the rest of the network, this is called a **link state packet**, it contains:
  - ▶ The name of the router
  - ▶ A sequence number
  - ▶ The age of the packet
  - ▶ For each neighbour, the neighbour's name and the cost of reaching it
- ▶ These three steps are repeated periodically, or when the router detects a problem with one of its neighbours

# Link State Routing

- ▶ The fourth step is the hardest part
- ▶ All the link state packets need to be distributed to all the routers in the network
- ▶ Each router must have the most recent packet, otherwise it could have an inconsistent picture of the network:
  - ▶ It could be missing new routers
  - ▶ It could still have old routers that no longer exist
- ▶ This could cause the routing information to be wrong
- ▶ The basis of this step is the flooding algorithm, the router send its link state packet to all of its neighbours, which send it on

# Link State Routing

- ▶ As we saw before we need to control the flooding
- ▶ Each packet has a sequence number, this allows routers to detect packets they've already received
- ▶ But, there's a problem here, if a router crashes and comes back up again its sequence numbers will restart at zero
- ▶ These packets will be ignored by the other routers, since they have seen higher numbered packets
- ▶ The solution is to add an age to the packet
- ▶ Internally, each router decrements this age value periodically
- ▶ When it reaches zero, the router forgets about the sending router

# Link State Routing

- ▶ In this way the new packets will eventually get through
- ▶ This step can be made more robust by sending acknowledgement packets whenever a link state packet is received
- ▶ There are a number of techniques for minimizing the number of packets that must be sent, we won't go into them
- ▶ The final step once all the link state packets have been received is to construct the network graph from this information and then run the shortest path algorithm to construct the routing table for the router
- ▶ This whole process must occur frequently to handle changes to the network

# Link State Routing

- ▶ The problem with this algorithm is it requires a lot of memory and processing time
- ▶ These resource requirements grow with the number of routers and the number of connections between the routers
- ▶ This means that each router must have a significant amount of processing power and this process runs in the background
- ▶ On the scale of the Internet this would require each router to be a powerful computer, something that isn't practical

# Hierarchical Routing

- ▶ We can't build the graph for the complete Internet in each router, it takes up too much space and time
- ▶ Hierarchical routing is the answer to this problem
- ▶ Each router just constructs a graph for the local region, such as the GTA or Ontario
- ▶ This reduces the size of the network
- ▶ For destinations outside of the region, each region has a link to a higher level router
- ▶ These higher level routers build the graph for the network at that level

# Hierarchical Routing

- ▶ When a packet goes to another region, the packet is routed to the higher level routers
- ▶ The router for the destination region is determined and it is forwarded to that region
- ▶ There can be multiple levels of this hierarchy, the aim is to keep the number of routers at each level to a reasonable number
- ▶ Using multiple levels makes it more likely that each router will have an accurate map of its level

# Issues

- ▶ There are several issues with routing algorithms
- ▶ They rely on all the routers behaving properly, if a small number of routers are faulty or don't implement the algorithm correctly routing may fail
- ▶ Routers will get an incorrect picture of the network, so the routing tables will be wrong
- ▶ Similarly if a router is causing errors in the packets it may take too long to get a complete picture of the network, or it could be incorrect

# Issues

- ▶ The process relies on routers being honest
- ▶ In the case of connection-oriented services, a router may advertise better performance in order to attract customers, this will increase the revenue to its owners
- ▶ There are also security issues, a router may want to monitor the traffic to a particular site, government or commercial, so it claims to have a direct connection to the site
- ▶ It will then become part of the shortest path to the site and will be able to copy the traffic going to that site
- ▶ It may just be interested in who is talking to who, doesn't need to know the packet contents

# Internet

- ▶ Before examining IP and routing take a look at the Internet architecture
- ▶ By necessity the Internet is hierarchical, there are far too many sites for it to be flat
- ▶ Each country has a backbone network, which is a high speed network with high end routers that spans the country
- ▶ Regional routers connect to the backbone network, and each region is then viewed as a subnet
- ▶ There may even be smaller networks within each region

# Internet

- ▶ The country networks, their backbones are then linked together to form a higher level network that moves packets between countries
- ▶ If a packet goes to another site within its region, it stays within the regional network, this reduces traffic at the country level
- ▶ The idea is to keep routing as local as possible
- ▶ This is a nice idea, but it's quite often broken
- ▶ For example, Canada has two backbones, one for commercial traffic and the other for research/educational traffic
- ▶ There is a similar structure in the US with multiple backbones

# Internet

- ▶ To make things more complicated, there can be direct connections between sites in different countries
- ▶ For example, the research networks in Canada and the US have direct connections
- ▶ The hierarchical picture is just an approximation, it is helpful, but there are exceptions
- ▶ We need to have some way of controlling the number of sites, routers can't possibly hold that much information

# Internet

- ▶ The standards for the Internet are controlled by the Internet Engineering Task Force, IETF, <https://www.ietf.org/>
- ▶ The standard documents are called RFC's Request For Comment, except these are standards and they don't want any comments
- ▶ They are numbered, and are ASCII documents, no fancy pictures
- ▶ Part of a page from the IP RFC is shown on the next slide

# Internet

## 2. OVERVIEW

### 2.1. Relation to Other Protocols

The following diagram illustrates the place of the internet protocol in the protocol hierarchy:

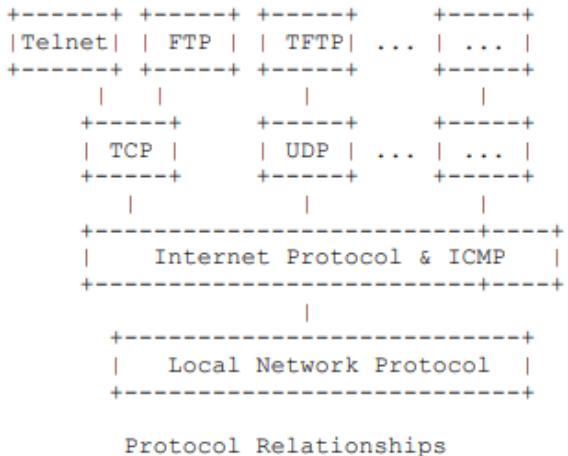


Figure 1.

Internet protocol interfaces on one side to the higher level host-to-host protocols and on the other side to the local network protocol. In this context a "local network" may be a small network in a building or a large network such as the ARPANET.

# IP Protocol

- ▶ The network protocol that is used on the Internet is called IP, Internet Protocol
- ▶ This is one of the original Internet protocols and has been around for many decades, we will start with the IPv4 protocol
- ▶ The key part of this protocol is the IP header, the network level receives data from the transport layer, typically TCP or UDP and adds the header to the front of it
- ▶ The header has changed a bit over the decades, mainly in response to the growth of the Internet

# IPv4 Protocol

- ▶ Start by examining the parts of the protocol that haven't changed, the ones that the routers rely on
- ▶ The current version of the header is shown in the next slide
- ▶ It is made up of 32 bit words in big endian format
- ▶ The header must be at least 5 words long, but can be as long as 15 words, depending on the options
- ▶ The initial motivation for the options was to provide a way to evolve the protocol, this really hasn't work out and many modern routers just ignore them, as will we

# IPv4 Header

# IPv4 Protocol

- ▶ The first byte has two important fields, the version field is the version of the protocol, in our case it is 4, for IPv6 it is 6
- ▶ Since this appears in the first byte, there can be significant differences between the protocol versions
- ▶ The second field in this byte is the header length, in 32 bit words
- ▶ Since this is a 4 bit field the maximum header length is 15 words, so there are 10 words for options
- ▶ The total length field gives the total length of the packet in bytes,
- ▶ This is a 16 bit field so packets could be 64K bytes long, but this isn't common

# IPv4 Protocol

- ▶ The next two important fields are the source and destination addresses
- ▶ Both of these fields are 32 bits, the interpretation of the address bits has changed over time, will come back to this later
- ▶ The fields in the second word largely deal with fragmentation
- ▶ A IPv4 packet can be 64K long, but some data link protocols, such as ethernet can't deal with this length
- ▶ In this case the packet must be divided into smaller packets, which are called fragments
- ▶ Each fragment is a IPv4 packet

# IPv4 Protocol

- ▶ The identification field identifies the packet that the fragment comes from
- ▶ The fragment offset field gives the position of this fragment within the original packet
- ▶ This is measured in units of 8 byte blocks, since this field is only 13 bits long
- ▶ The third word has three interesting fields
- ▶ The time to live (TTL) field is decremented each time the packet goes through a router, when it reaches 0 the packet is dropped
- ▶ This keeps packets from circulating forever

# IPv4 Protocol

- ▶ The protocol field is the transport level protocol that is being carried in the packet, this is usually TCP or UDP, but it can have other values
- ▶ The header checksum field is a checksum of the header, this is used to check for errors within the header
- ▶ These fields have stayed constant though the evolution of IPv4, they are the fields that routers need to process the packets
- ▶ The ECN field in the second byte is used to communicate end to end congestion information, this is optional

# IPv4 Protocol

- ▶ The DSCP field is used for classifying services that can run over IPv4
- ▶ The aim is to give certain services higher priority than other services
- ▶ For example video and phone need to process packets at a constant rate, but can tolerate a higher error level
- ▶ A file transfer cannot tolerate errors, but doesn't have the same real time constraints
- ▶ This is called **differentiated service** and there are 64 possible values
- ▶ This is optional for routers and network operators can define their own values for these fields

# IPv4 Protocol

- ▶ The 3 bit flags field is used with fragmentation
- ▶ Bit 0 must always be zero, it isn't used
- ▶ Bit 1 is the don't fragment (DF) flag, if this flag is set the packet won't be fragmented and an error will be returned
- ▶ This is used by network management software to determine the largest packet that can be sent over a path
- ▶ Bit 2 is the more fragments (MF) flag, this flag is set in all except the last fragment in a packet
- ▶ This indicates when all the fragments in a packet should have been received

# IPv4 Addresses

- ▶ IP addresses are 32 bits, or 4 bytes
- ▶ We've already seen dotted decimal notation where each byte is written as an integer between 0 and 255
- ▶ The interpretation of these addresses has evolved over time, as the network has grown we've had to add structure to the addresses
- ▶ When there were very few computers on the Internet the routers could know about all the computers, each one would have an entry in the routing table
- ▶ This didn't last very long, needed to impose a hierarchy on the address

# IPv4 Addresses

- ▶ At the most abstract level an address consists of a network address and a host address
- ▶ The network identifies a **subnet**, or the router for a LAN or group of LANs
- ▶ The host address identifies the individual computer within this network
- ▶ The high order bits are the network address and the low order bits are the host address
- ▶ Several scheme have been developed for telling the two types of addresses apart

# IPv4 Addresses

- ▶ The first was a static division, this was called network classes and they were based on the size of the subnet
- ▶ This soon became unworkable and other approaches were developed
- ▶ The **subnet mask** is one way of doing this
- ▶ This is a 32 bit word, with 1 bits where the network address is and 0 bits where the host address is
- ▶ This is again written in dotted decimal notation
- ▶ When an interface is set up both its IP address and subnet mask must be specified

# IPv4 Addresses

```
mark@cloud2:~/CSCI3310
mark@cloud2:~$ cd CSCI3310
mark@cloud2:~/CSCI3310$ ifconfig
eth2      Link encap:Ethernet HWaddr 00:1a:a0:cb:7e:12
          inet addr:199.212.32.242 Bcast:199.212.32.243 Mask:255.255.255.252
          inet6 addr: fe80::21a:a0ff:fecb:7e12/64 Scope:Link
                  UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
                  RX packets:46012715 errors:0 dropped:0 overruns:0 frame:0
                  TX packets:45750088 errors:0 dropped:0 overruns:0 carrier:0
                  collisions:0 txqueuelen:1000
                  RX bytes:28454617700 (28.4 GB) TX bytes:8391332416 (8.3 GB)
                  Interrupt:16

eth3      Link encap:Ethernet HWaddr 00:14:d1:1d:35:50
          inet addr:192.168.0.1 Bcast:192.168.0.255 Mask:255.255.255.0
          inet6 addr: fe80::214:d1ff:feld:3550/64 Scope:Link
                  UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
                  RX packets:11438416 errors:0 dropped:0 overruns:0 frame:0
                  TX packets:25024017 errors:0 dropped:0 overruns:0 carrier:0
                  collisions:0 txqueuelen:1000
                  RX bytes:1409957892 (1.4 GB) TX bytes:24908477878 (24.9 GB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1 Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
```

# IPv4 Addresses

- ▶ In this example for eth3 the subnet mask is easy, but for eth2, its not
- ▶ How many host address can I have on eth2?? Can't really tell by looking at the subnet mask
- ▶ Need to convert the last byte to hex, which is FC, there are only 2 bits for host addresses
- ▶ We can make this a lot simpler by observing that high order bits are always the network address
- ▶ This is the idea behind **CIDR** (Classless InterDomain Routing) notation
- ▶ This is given by a / followed by the number of bits in the network part of the address

# IPv4 Addresses

- ▶ This can be used on both the IP address and the subnet mask
- ▶ In CIDR notation my eth2 address is 199.212.32.242/30
- ▶ So how does this help the routers?
- ▶ They don't need to store host addresses, they only need to store the network addresses
- ▶ In addition the network addresses can be of variable length
- ▶ So if we had a /16 address, it could be referring to over 65,000 hosts, but need only one table entry
- ▶ This entry would give the line to that subnet

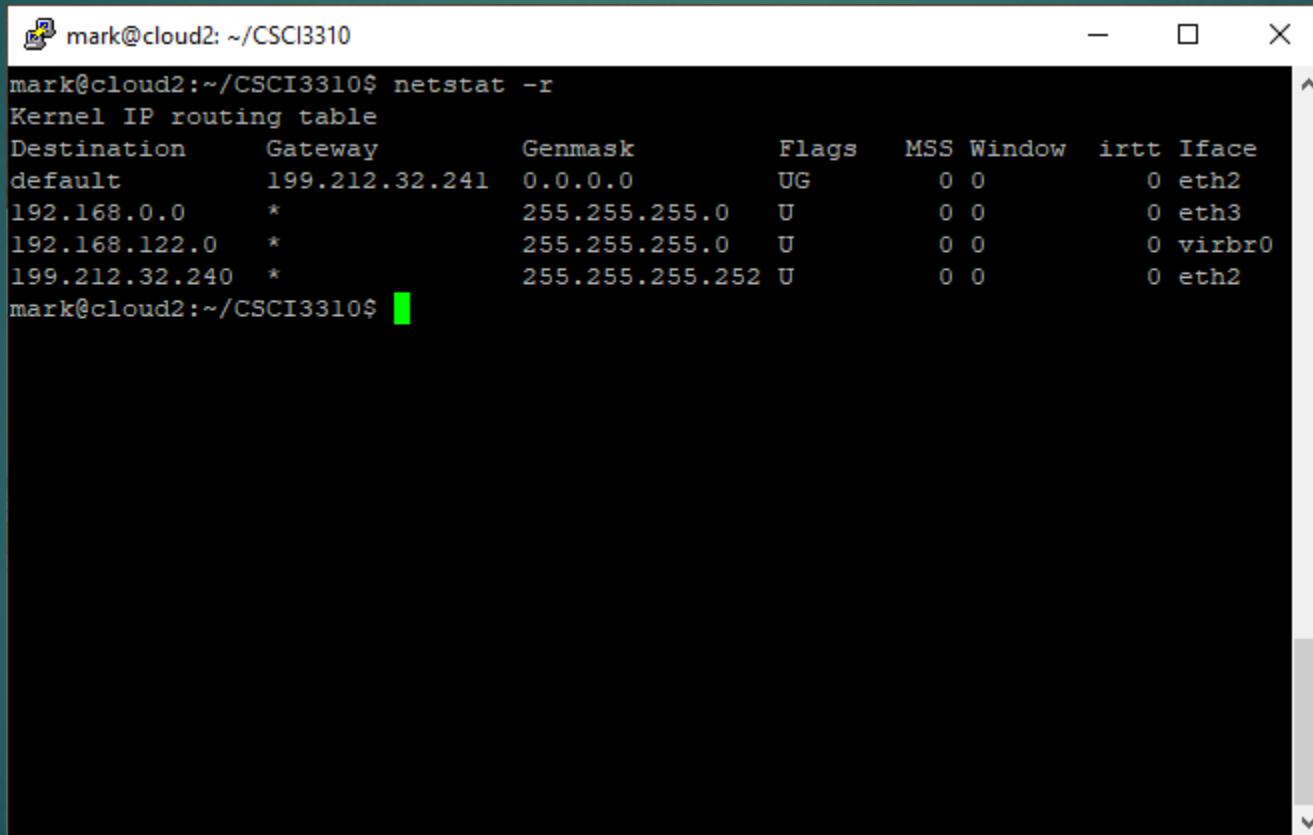
# IPv4 Addresses

- ▶ This nicely fits into our router hierarchy as well
- ▶ Our /16 address points to a single router, but at this router there could be multiple /24 addresses, each of which goes to a separate router
- ▶ We can put the division anywhere that we like
- ▶ Not all the network addresses in a router need to be the same length, there can be different numbers of bits
- ▶ They can even overlap, with the longest network address taking precedence in routing decisions

# IPv4 Addresses

- ▶ The next slide shows the routing table for the computer from the ifconfig output
- ▶ Note that this computer has two ethernet interfaces, it serves as the gateway for my laboratory
- ▶ The eth2 interface is for the outside Internet, and the eth3 interface is for the computers in the lab
- ▶ The destination field is a network address, the genmask field gives the number of bits in this address
- ▶ The interface field specifies the interface the packets are to go out on

# Routing Table



mark@cloud2: ~/CSCI3310\$ netstat -r  
Kernel IP routing table  
Destination Gateway Genmask Flags MSS Window irtt Iface  
default 199.212.32.241 0.0.0.0 UG 0 0 0 eth2  
192.168.0.0 \* 255.255.255.0 U 0 0 0 eth3  
192.168.122.0 \* 255.255.255.0 U 0 0 0 virbr0  
199.212.32.240 \* 255.255.255.252 U 0 0 0 eth2  
mark@cloud2:~/CSCI3310\$

# IPv4 Addresses

- ▶ The default entry in this table specifies where the packets that don't fit the other entries go
- ▶ All the computers in my laboratory, except the gateway one aren't directly on the Internet
- ▶ And the trick here is what saved IPv4
- ▶ It was originally thought that all computers would need to be directly connected to the Internet
- ▶ With home networks the concern was that we would soon run out of Internet addresses, which was the main reason for going to IPv6

# IPv4 Addresses

- ▶ We solved this problem by taking 3 ranges of IP addresses and making them local addresses, they cannot be used on the Internet at large
- ▶ These ranges are:
  - 10.0.0.0 – 10.255.255.255/8
  - 172.16.0.0 – 172.31.255.255/12
  - 192.168.0.0 – 192.168.255.255/16
- ▶ We can assign these addresses within the home, laboratory, or campus, and they won't conflict with other sites

# IPv4 Addresses

- ▶ I can have as many computers as I want at home, and I only need one external IP address!
- ▶ But, how do these internal computers access the Internet?
- ▶ We use **NAT** (Network Address Translation) for this
- ▶ When the internal computer wants to reach an Internet site it uses the IP address and port number of the service it wants
- ▶ The packet is sent to the gateway to send it to the Internet, but the internal computers IP address can't be used as the source IP address, it's not legal on the Internet at larger

# IPv4 Addresses

- ▶ To solve this problem, the gateway puts its IP address in the packet and assigns it an unused port on the gateway
- ▶ The gateway then saves this port number, and the mapping back to the internal computer's IP address and port
- ▶ When the destination responds, the gateway examines the port number and then determines the correct internal IP address and port to send it to
- ▶ Note: this will fail when we run out of free port numbers, this limits the number of connections to 10,000 to 20,000
- ▶ Okay for a house, maybe not okay for a university

# IPv4 Addresses

- ▶ There's one problem left to solve
- ▶ I want to run a server on one of my internal computers, but it's not visible to the outside world
- ▶ This is handled through **port mapping**
- ▶ Take an unused port number on the gateway and use it for the external connection
- ▶ When a request comes for that port, the request is forwarded to the internal computer, the port number can be changed in the mapping

# IPv4 Addresses

- ▶ Most home routers include both NAT and port mapping, nothing special needs to be done
- ▶ This was at first viewed as a horrible hack by many network people, but it has a number of advantages
- ▶ It gave us lots of time to implement IPv6
- ▶ It keeps a lot of traffic off of higher level networks
- ▶ The packets for your home printer don't go out on the Internet, they stay within your home network
- ▶ Your home computers also don't see all the garbage packets on the Internet

# IPv4 Addresses

- ▶ One big advantage is it increases the security of the home network
- ▶ External agents can't directly access computers inside your home network
- ▶ People aren't good at setting up security, the gateway can handle a lot of this
- ▶ An old hacker trick is to check each of the ports on an IP address to see which ones are open and running an insecure service
- ▶ The home router really isn't running any services, and with port mapping hackers can't easily determine if its an insecure service

# ARP

- ▶ When IP sends a packet it knows the IP address, but it will pass it off to ethernet to actually deliver the packet along the wire
- ▶ Ethernet knows nothing about IP addresses, so we need some way of mapping IP addresses to ethernet addresses
- ▶ The Address Resolution Protocol, **ARP**, is used for this purpose
- ▶ When a computer sends a message it sends an ARP packet out onto the Ethernet, this is a protocol at the same level as IP
- ▶ This packet is broadcast to all the computers on the ethernet link
- ▶ It contains the destination IP address, along with the IP address of the current computer

# ARP

- ▶ The computer with the requested IP address will then respond, the packet it sends back will have its ethernet address
- ▶ To save broadcasts, the ARP response is saved in a cache
- ▶ Most computers will watch the ARP requests/responses go by and add that information to their caches as well
- ▶ Periodically entries are removed from the cache in case a computer is disconnected or its IP address changes
- ▶ The arp program can be used to print the current contents of the cache, see the next slide

# ARP

```
mark@cloud2: ~
mark@cloud2:~$ arp -v
Address          HWtype  HWAddress          Flags Mask       Iface
sr7-s.con.dc-uoit.net  ether   00:1a:a0:c9:99:41  C           eth3
192.168.0.178    ether   00:14:6c:53:f8:b4  C           eth3
sr5-s.con.dc-uoit.net      (incomplete)          eth3
192.168.0.105     ether   00:21:b7:a0:2d:15  C           eth3
192.168.0.154      (incomplete)          eth3
192.168.0.179      (incomplete)          eth3
docker2          ether   00:30:67:71:20:3b  C           eth3
199.212.32.241    ether   00:16:ca:4b:00:4b  C           eth2
docker          ether   00:30:67:71:1b:1e  C           eth3
Entries: 9      Skipped: 0      Found: 9
mark@cloud2:~$
```

# Packet Sniffing

- ▶ **Packet sniffing** is the process of capturing data link packets that appear at a network interface
- ▶ In the case of a switched ethernet data link this would only be packets sent to that interface, plus any broadcast packets
- ▶ In the case of non-switched ethernet and WiFi this would be all packets on the data link segment, regardless of who they are addressed to
- ▶ That is, all the packets that appear on a given data link will be captured and they are in the form of data link packets

# Packet Sniffing

- ▶ Why would I want to do this? There are two main legitimate reasons
- ▶ First, it is used in network testing and debugging
- ▶ Example: a faulty switch, capture the packets going into the switch and coming out of it
- ▶ Are the packets being switched correctly? Is the switch modifying the packets? Are a large number of packets being dropped?
- ▶ On a single data link, check to see if there are packets that shouldn't be on that link
- ▶ That was the case here when I first arrived, there was a lot of garbage on the university network

# Packet Sniffing

- ▶ Second, security and threat detection
- ▶ Are there packets on the data link that shouldn't be there?
- ▶ Is this the sign of an attack, is someone probing the network looking for weak spots
- ▶ There are real-time tools that monitor this type of thing
- ▶ If an attack is occurring determine where it is coming from and how it can be stopped
- ▶ Example: filter out packets coming from a particular source IP address, inform network operators about the problem

# Packet Sniffing

- ▶ There are a number of packages for packet sniffing
- ▶ A well known open source one is tcpdump and libpcap
- ▶ This package has been around for decades and is one of the main work horses in the area
- ▶ It has also defined a standard file format for packet captures called PCAP, example packet captures can be downloaded from the Internet
- ▶ This package was originally designed for Unix and is now widely available on Linux, there are also ports to Windows
- ▶ Packages like Wireshark and Nmap are more modern versions

# libpcap

- ▶ Will start examining libpcap and work our way up to the IP level
- ▶ Libpcap can either capture live copy off of an existing network connection, or it can read a pre-captured file, the only difference is in how the capture device is opened
- ▶ Libpcap will deliver data link level packets, one packet at a time
- ▶ This is in the format of an array of bytes, no interpretation of the packets, that's our job
- ▶ So, the array of bytes will be a data link packet
- ▶ In the case of ethernet it will start with an ethernet header, in the case of WiFi it will start with a WiFi header, they are different

# libpcap

- ▶ Our first job is to process the data link header, stick to ethernet
- ▶ Recall an ethernet header has 6 bytes of destination address, 6 bytes of source address and two bytes of type
- ▶ The header is always 14 bytes long, so it is tempting to just skip over it, but we should examine the type field to make sure its an IP packet, there are other possibilities
- ▶ If its an IP packet, skip over 14 bytes to find the start of the IP header
- ▶ An IP header is variable length, so we need to examine the header to find where the enclosed packet starts

# libpcap

- ▶ The IP header will tell us the type of transport layer service that is in the packet
- ▶ We will typically be looking for TCP or UDP
- ▶ Skip over the IP header and then start processing the TCP or UDP header
- ▶ That is the general outline of processing a capture, go down one more level before looking at actual code
- ▶ Libpcap will go into a loop capturing packets, when a new packet is found it will call a callback procedure, we specify this procedure before entering the loop

# libpcap

- ▶ There are many ways of structuring the callback procedure
- ▶ I will go through one architecture that works quite well
- ▶ The initial callback procedure processes the data link packets, in our case ethernet packets
- ▶ It will check the type of packet, and will call another procedure based on the type
- ▶ In our case we are only looking for IP packets, so we need to write a procedure to handle them
- ▶ The IP procedure will then determine the type of transport protocol and call the TCP or UDP procedure

# libpcap

- ▶ The basic idea is to have a separate procedure for each protocol
- ▶ This procedure will collect and print the information for that protocol and then determine the next procedure to call
- ▶ This separates the protocols into their own procedures, easier to concentrate on one thing at a time
- ▶ Debug one layer of the protocol stack at a time, make sure that the data passed up the protocol stack is valid
- ▶ It is also easy to add another protocol, just look at the type field and call the appropriate procedure

# libpcap

- ▶ Now to look at the code, more information can be found at  
<https://www.tcpdump.org/>
- ▶ The first thing we need to do is open a source of packets
- ▶ In the case of processing a PCAP file, we use a statement of the form:

```
handle = pcap_open_offline(argv[1], error_buffer);
```

- ▶ The first parameter is the name of the file and the second parameter is char array that contains any error message, this is a common approach in this library
- ▶ A pointer to a pcap\_t structure is returned, this is used in subsequent libpcap calls

# libpcap

- ▶ If we are capturing from a network interface we use the following calls:

```
device = pcap_lookupdev(error_buffer);
```

```
handle = pcap_open_live(device, BUFSIZ, 0, packet_timeout,  
error_buffer);
```

- ▶ The first call finds the name of the first interface device, this is a char array
- ▶ The second call opens this device for capture, the second parameter is the size of the internal capture buffer, BUFSIZ is defined in pcap.h

# libpcap

- ▶ The third parameter is a flag indicating the use of promiscuous mode
- ▶ In this mode all the packets on the data link are captured, otherwise only the packet for the network interface are captured
- ▶ On switched ethernet this has no impact, on WiFi it would capture everything
- ▶ The third parameter is a timeout in msec, if a packet isn't seen during this time period the capture will terminate
- ▶ This procedure also returns a pointer to a pcap\_t structure

# libpcap

- ▶ Once we have a handle we can enter the capture loop:  
`pcap_loop(handle, packet_count, packetCallback, NULL);`
- ▶ The first parameter is the handle, the second parameter is the number of packets to capture, if 0 is used it will process packets until the end of file, or the capture process is interrupted
- ▶ The third parameter is the procedure to call for each packet, this is the callback procedure
- ▶ The fourth parameter is a string that will be passed to the callback procedure each time it is called

# libpcap

- ▶ The next slide shows the callback procedure from laboratory nine
- ▶ The first parameter is the string passed as the last parameter to `pcap_loop()`, we aren't using it
- ▶ The second parameter is a structure with information on the packet capture, the most important field in this structure is the time when the packet was captured
- ▶ The third parameter is a pointer to an array of bytes, the packet that was captured
- ▶ The first thing this procedure does is print the time of capture

# libpcap

```
void packetCallback(u_char *args, const struct pcap_pkthdr *header,
                    const u_char *packet) {
    struct ether_header *eptr;
    short type;
    printf("time: %s", ctime((const time_t*) &header->ts.tv_sec));
    eptr = (struct ether_header *) packet;
    type = ntohs(eptr->ether_type);
    if(type == ETHERTYPE_IP) {
        printf("IP packet\n");
        processIP(packet+14);
    }
    if(type == ETHERTYPE_ARP) {
        printf("arp packet\n");
    }
    printf("\n");
}
```

# libpcap

- ▶ Now we need to process the ethernet frame
- ▶ There are several ways that we could do this, one approach is to examine the frame one byte at a time and build up its fields
- ▶ The other way is to cast the third parameter to a pointer to a structure that describes the frame structure
- ▶ We could describe this structure ourselves, or we could use the one declared in the /usr/include/netinet directory
- ▶ I'm using the second approach, eptr is a pointer to an ether\_header structure, I use this structure to decode the frame

# libpcap

- ▶ The type field is extracted from this structure, converted to the host byte order, and then we can determine the type of packet that is inside the ethernet frame
- ▶ We are only interested in IP packets, so when we find one we call the processIP() procedure with the contents of the ethernet frame
- ▶ We also print a message if we find an ARP packet, but we don't process it
- ▶ The procedure for processing an IP packet is quite similar, and again we use the structures that are declared in /usr/include/netinet

# libpcap

```
void processIP(const u_char *packet) {
    struct iphdr *ip;
    u_char *payload;
    char *addr;
    unsigned int len;
    /*
     * cast the bytes to an IP packet header
     */
    ip = (struct iphdr*) packet;
    /*
     * check that we have an IPv4 packet
     */
    if(ip->version != 4) {
        printf("not version 4\n");
        return;
    }
```

# libpcap

- ▶ This procedure checks the version of the IP packet, we are only interested in IPv4
- ▶ It prints some information about the packet, pay attention to how the IP addresses are printed
- ▶ It then determines if the enclosed packet is a TCP or UDP packet
- ▶ At this point we would call the appropriate procedure to process these packets

# libpcap

```
/*
 * compute the header length and the location
 * of the TCP or UDP packet
 */
len = ip->ihl*4;
printf("header length: %d\n", len);
payload = (unsigned char*)packet+len;

if(ip->protocol == IPPROTO_TCP) {
    printf("TCP packet\n");
    // call the TCP procedure here
}
if(ip->protocol == IPPROTO_UDP) {
    printf("UDP packet\n");
    // call the UDP procedure here
}
```

# libpcap

```
/*
 * print the source and destination addresses
 */
addr = (char*) &(ip->saddr);
printf(" source: %hu.%hu.%hu.%hu\n",addr[0], addr[1],addr[2],
addr[3]);
addr = (char*) &(ip->daddr);
printf(" destination: %hu.%hu.%hu.%hu\n",addr[0],
addr[1],addr[2], addr[3]);
```

# Summary

- ▶ Examined some of the issues in the network layer
- ▶ Examined some of the common routing algorithms used at the network level
- ▶ Examined the structure of the IPv4 protocol, IPv6 is similar
- ▶ Looked at how the Internet is structure as a hierarchy of routers
- ▶ Examined packet capture and one the libraries that can be used for it



# CSCI 3310

# Network Programming

# Part Three

MARK GREEN  
ONTARIO TECH

# Introduction

- ▶ So far we have used TCP, a stream interface to networks that is reliable
- ▶ Networks are not byte streams, they are packets of bytes
- ▶ In TCP a packet is sent over the network, and the sender waits for an acknowledgement from the receiver that it got there okay
- ▶ If it doesn't get the acknowledgement it sends the packet again
- ▶ It also makes sure that the packets are received in the correct order
- ▶ This can be time consuming, it can take seconds or longer to correctly transmit a packet

# Data Grams

- ▶ TCP is what we want for many applications, such as file transfer
- ▶ In other applications timeliness is far more important
- ▶ Think of a network game, if you are fighting an enemy, you want to know their status immediately, you don't want to wait for a reliable transmission
- ▶ There will be a constant stream of packets with the enemy's state, if one is lost it doesn't matter
- ▶ You don't want to wait for a retransmission, you would rather have the most recent status, particularly if the enemy is moving

# Data Grams

- ▶ This is the idea behind data grams and UDP
- ▶ This is a lighter weight protocol, it is faster, but it is not reliable
- ▶ This is the protocol used when there is a constant stream of status updates
- ▶ You aren't interested in what happened in the past, you want to know the current situation
- ▶ A number of the network management applications use this approach as well

# UDP

- ▶ With TCP we first establish a connection and then send information over this connection
- ▶ UDP doesn't use connections, each time we send a message we specify the network address and port
- ▶ We can't use read and write with UDP, instead we use `sendto()` and `recvfrom()`
- ▶ The first three parameters to these functions are the same as `read()` and `write()`, while the last two parameters are the same as `connect()` and `accept()`
- ▶ The fourth parameter to these functions is a flag that is 0 in most cases

# UDP

- ▶ As an example we will redo the echo server as a UDP server, this is example4.tar on Canvas
- ▶ The start of the server is similar to the TCP one, we need to establish the address and port number, create the socket and bind the socket
- ▶ The main difference is in getaddrinfo:

```
hints.ai_socktype = SOCK_DGRAM;  
hints.ai_flags = AI_PASSIVE | AI_ADDRCONFIG;  
if((rc = getaddrinfo(NULL, "4321", &hints, &addr))) {
```

# UDP

- ▶ With UDP we don't need to call listen or accept, since there are no connections
- ▶ The body of our server is:

```
while(1) {  
    rc = recvfrom(sock, buffer, 512, 0,  
                  (struct sockaddr*) &address, &addrLength);  
    sendto(sock, buffer, rc, 0,  
           (const struct sockaddr*) &address, addrLength);  
}
```

# UDP

- ▶ This is a smaller server than TCP
- ▶ The client is equally simple, we make the same change to the input to getaddrinfo(), and we don't need to call connect()
- ▶ The code for the main loop in the client is shown on the next slide
- ▶ The only complicated part is checking for end of file from the user
- ▶ The rest is a straight forward send() and receive() from the server
- ▶ Again this is shorter than the TCP version

# UDP

```
while(1) {
    ret = fgets(buffer, 512, stdin);
    /*
     * check for end of file
     */
    if(ret == NULL)
        break;
    sendto(sock, buffer, strlen(buffer), 0,
           (const struct sockaddr*) addrinfo, addr->ai_addrlen);
    recvfrom(sock, buffer, 512, 0, NULL, 0);
    printf("%s",buffer);
}
```

# UDP

- ▶ But there are some serious problems here, just look at the client code
- ▶ First, the `recvfrom()` call accepts input from anywhere
- ▶ On the first `sendto()` a port will be assigned to our client, but anything sent to that port we will receive
- ▶ This is okay for the server, since its job is to echo everything it receives
- ▶ But, we could have a rogue application sending things to our client
- ▶ It probably doesn't matter in this case, but it could be important in others

# UDP

- ▶ There is a more serious issue, UDP isn't reliable, a packet could be dropped
- ▶ If the client and server are running on the same computer this won't occur, but as soon as the network is involved we could have problems
- ▶ If the echo from the server is lost our client will hang in the `recvfrom()` call
- ▶ The packet is lost, there is nothing to receive, so the call will never return
- ▶ This is a major problem

# UDP

- ▶ We can solve the first problem by using connect()
- ▶ Yes, we can connect() a UDP socket, but this is different from TCP, since it doesn't build a connection between the two processes
- ▶ This is called a connected UDP socket
- ▶ Exactly what connect() does varies between OS's
- ▶ At the bare minimum it informs the kernel that we are only interested in that address, so we can't receive data grams from other addresses
- ▶ On some systems it will check to see if the address is reachable and the port has a process, but you can't rely on this

# UDP

- ▶ With a connected socket we can use read and write again, since we know what's on the other end of the socket
- ▶ We can still use sento and recvfrom, but in this case the address must be NULL and the address length must be zero
- ▶ Unlike TCP sockets, connect() can be called multiple times with a UDP socket
- ▶ This allows us to connect with multiple peers, we can even disconnect
- ▶ This is clearly useful for clients, but can be used with servers
- ▶ After the initial recvfrom() the server can use connect() to interact with only one client

# UDP

- ▶ We can still use select() with UDP sockets, this solves the second problem with our client
- ▶ We can use select() to determine if the socket is ready to read or the terminal is ready
- ▶ This prevents us from blocking in the recvfrom() system call
- ▶ Note that this isn't a problem in the server
- ▶ It waits for any client to connect and then immediately replies with the line it receives

# Summary

- ▶ Examined the basics of UDP
- ▶ Examined a simple UDP server and client
- ▶ Discussed some of the problems that can occur with UDP
- ▶ Will come back to UDP later once we have covered some of the theory

# CSCI 3310

# Transport Layer

MARK GREEN  
ONTARIO TECH

# Introduction

- ▶ The transport layer provides services to applications, this is the part of the network that applications deal with
- ▶ Provides a clean reliable interface to networking, hides all the nasty details so programmers don't need to worry about them
- ▶ Start by examining some of the general issues at the transport level
- ▶ Then turn to TCP and UDP, which are the most popular protocols in this layer

# Reliability

- ▶ In general the network layer provides unreliable connectionless delivery of packets
- ▶ This is a reasonably good match to something like UDP, but for connection oriented protocols, like TCP we need more
- ▶ TCP guarantees a reliable byte stream between applications
- ▶ All bytes must be delivered correctly and in order
- ▶ Clearly this can't be done with straight IP, we need some way of detecting lost packets and packets that arrive out of order
- ▶ Note, from the application point of view we are sending bytes, but for the network layer they must be packaged as packets

# Reliability

- ▶ The standard technique for detecting lost packets is to use acknowledgments, we've seen this before at the lower levels
- ▶ There is a major difference here, transport layer packets go from host to host, previous techniques were just on the data link, there is a different time scale
- ▶ On the data link, for example WiFi, we can send a packet and then wait for the acknowledgment, the round trip time is little more than twice the time required to send the packet
- ▶ This basically halves the bandwidth
- ▶ But for hosts on the other side of the world the transmission time can be a good fraction of a second or longer

# Reliability

- ▶ Waiting for an acknowledgment for each packet will slow communications down by several orders of magnitude
- ▶ The solution is to send a number of packets without waiting for acknowledgements, they will come along later
- ▶ This introduces a buffering problem, if a packet is lost we need to send it again
- ▶ This means at the transport layer we need to keep the packet in a buffer until we receive the acknowledgement
- ▶ We need to do the same on the receiving side, packets may come in out of order, we need to assemble them correctly before passing them up to the application

# Reliability

- ▶ How much buffering do we need?
- ▶ That depends upon network speed and the number of errors
- ▶ The operating system will need to buffer all the connections, so if we are not careful we could use up a lot of memory
- ▶ We would like to put each packet in a separate buffer, easy to release the buffer when we are finished with it
- ▶ How big does each of these buffers need to be?
- ▶ We could say the maximum packet size, but many of the packets will be much smaller, this is a waste of memory

# Reliability

- ▶ We could have variable size buffers, but this would require a memory allocation for each buffer
- ▶ For fixed sized buffers, we allocate them once, and then keep a list of free buffers
- ▶ Maybe we could have several pools of different sizes
- ▶ Once we run out of buffers, at either end, the whole communications process stops
- ▶ Need to be careful that we don't create a deadlock
- ▶ Protocols must be designed to avoid this

# Checksums

- ▶ We need to detect packets with errors, so we can discard them
- ▶ We do this by adding checksums to our transport layer packets
- ▶ But, aren't we already doing this at the lower layers?
- ▶ While the lower layer checksums are helpful, they are not sufficient
- ▶ At the data link layer they only work over a single wire, at the network level they work over multiple wires
- ▶ But, routers recalculate checksums, since they can modify packet headers
- ▶ If a router is faulty, the checksum could be wrong

# Checksums

- ▶ This is why we add checksums to the transport layer packets
- ▶ An extra check to make sure nothing happened between the sending host and the receiving host
- ▶ Warning, checksums will not catch all of the errors, but it will get many of them
- ▶ A checksum will detect a single bit error
- ▶ But, if there are two bits in error, they can fool the checksum

# Multiplexing

- ▶ At the network layer packets are delivered from one host to another
- ▶ At the transport layer packets are delivered between applications
- ▶ The networking software needs to determine which application a packet needs to be sent to
- ▶ This is called multiplexing and it's usually performed using port numbers, or a similar way of identifying the application endpoint for the packet

# UDP

- ▶ UDP is an unreliable packet oriented service
- ▶ The main thing it adds over top of IP is multiplexing
- ▶ IP delivers packets between hosts, UDP adds the ability to deliver packets between applications
- ▶ It does this by adding port numbers to its header, with the port numbers identifying the source and destination applications
- ▶ The UDP header is shown on the next slide, the only two required fields in the header are the destination port and the packet length
- ▶ The source port is optional, it should be set to zero if not used

# UDP Packet Header

| UDP datagram header |       |             |   |   |   |   |   |   |   |   |   |    |    |    |    |    |                  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|---------------------|-------|-------------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|------------------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Offsets             | Octet | 0           |   |   |   |   |   |   | 1 |   |   |    |    |    |    | 2  |                  |    |    |    |    |    | 3  |    |    |    |    |    |    |    |    |    |    |
| Octet               | Bit   | 0           | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15               | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 0                   | 0     | Source port |   |   |   |   |   |   |   |   |   |    |    |    |    |    | Destination port |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 4                   | 32    | Length      |   |   |   |   |   |   |   |   |   |    |    |    |    |    | Checksum         |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

# UDP

- ▶ If a source port isn't included there is no way to get a reply, so it is used on almost all UDP packets
- ▶ The checksum is also optional, but it's almost always included
- ▶ This provides an end-to-end check of the validity of the packet
- ▶ The checksum is performed over a pseudo header which includes some of the information from the IP headers, shown on the next slide
- ▶ While this violates layering, since it involves data from two layers, it provide an extra guarantee that the packet is valid
- ▶ In particular that it has reached the correct IP address

# UDP Pseudo Header

# UDP

- ▶ UDP can be used on its own, particularly over reliable network connections, as in the lab
- ▶ It is used for time sensitive information, packets that must be delivered quickly, but not necessarily reliably
- ▶ Since it just adds multiplexing to IP it is quite often used as the basis for other protocols
- ▶ Easy to build another protocol on top of it, there is little to get in the way
- ▶ A good way to experiment with new network protocols

# TCP

- ▶ TCP is a reliable byte oriented protocol, it transfers streams of bytes from one application to another
- ▶ TCP has no notion of packets, while TCP traffic is packaged as packets at the network level, at the transport level packets don't exist
- ▶ This makes acknowledgements more difficult, we must acknowledge at the byte level and not the packet level
- ▶ Thus the sequence numbers and acknowledgements must be in terms of bytes and not packets

# TCP

- ▶ The TCP header is shown on the next slide
- ▶ As expected there is a source and destination port
- ▶ The data offset, also called the header length, is the size of the header in 32 bit words
- ▶ TCP headers have options, and in this case they are actually used and quite common
- ▶ The sequence number is incremented on each byte and indicates how much the transmitter has sent
- ▶ The acknowledgement specifies the next byte+1 expected, it acknowledges the receipt of all the bytes up to that point

# TCP Header

# TCP

- ▶ The checksum is over the TCP pseudo header, shown in the next slide, which is similar to the UDP pseudo header and plays the same role
- ▶ The window size gives the amount of buffer space the receiver still has available
- ▶ This is included in acknowledgements so the transmitter knows how much data it can send without blocking or dropped packets
- ▶ In the fourth word there are a number of flags that deal with how packets are processed
- ▶ The CWR and ECE flags are used in flow control

# TCP Pseudo Header

TCP pseudo-header for checksum computation (IPv4)

| Bit offset | 0–3                    | 4–7      | 8–15             | 16–31          |  |  |
|------------|------------------------|----------|------------------|----------------|--|--|
| 0          | Source address         |          |                  |                |  |  |
| 32         | Destination address    |          |                  |                |  |  |
| 64         | Zeros                  | Protocol | TCP length       |                |  |  |
| 96         | Source port            |          | Destination port |                |  |  |
| 128        | Sequence number        |          |                  |                |  |  |
| 160        | Acknowledgement number |          |                  |                |  |  |
| 192        | Data offset            | Reserved | Flags            | Window         |  |  |
| 224        | Checksum               |          |                  | Urgent pointer |  |  |
| 256        | Options (optional)     |          |                  |                |  |  |
| 256/288+   | Data                   |          |                  |                |  |  |

# TCP

- ▶ The URG field is set if there is urgent data
- ▶ In this case the urgent pointer is the offset from the current sequence number where the urgent data can be found
- ▶ This was to provide signalling information to the application and can be processed out of order
- ▶ Example: the user has pressed the ctrl-C key to interrupt a process
- ▶ While this is an interesting idea, it really isn't used that much in practice
- ▶ If the ACK flag is 1, the acknowledgment field is valid, otherwise it is to be ignored

# TCP

- ▶ Why would we do this?
- ▶ If the traffic is largely one way, the sender may have nothing to acknowledge, so there is no need to interpret the field
- ▶ The PSH indicates pushed data, the receiver should send the data to the application as soon as possible
- ▶ Normally TCP would buffer data before sending it to the application, which is more efficient
- ▶ This is used when trying to do something close to real-time over TCP
- ▶ Another example is a terminal program, you want keystrokes echoed immediately, not after a screen full of characters have been accumulated

# TCP

- ▶ The RST flag is used to reset a connection, this terminates the connection in an abnormal way
- ▶ This could occur due to an application crash, or when a host refuses to accept a connection
- ▶ This could also be sent if there is a major error somewhere in the network, too many bad packets
- ▶ The SYN flag is used in establishing connections
- ▶ The FIN flag indicates that an application is finished, it will not send more data and is releasing the connection

# TCP

- ▶ There are a number of options that are used with TCP
- ▶ The MSS or maximum segment size option specifies the largest packet size a host is willing to accept
- ▶ By default this is a payload size of 536 bytes
- ▶ The connection is more efficient if this size is increased particularly over high speed links
- ▶ The window size field is 16 bits, this is the available buffer space, so the maximum available buffer space is 64K
- ▶ This was okay in the early days with slower networks, but with high speed networks larger buffers and transmissions are more efficient

# TCP

- ▶ The window scale option allows the two hosts to negotiate a scale factor for the window size
- ▶ This is the number of bits that the window size is shifted to the left
- ▶ The maximum value that can be used is 14
- ▶ In this case the window size is no longer in units of bytes, but chunks of bytes
- ▶ The timestamp option is sent by the sender and echoed by the receiver on each packet
- ▶ This is used to estimate the round trip time on the connection, which is used in setting retransmit timers

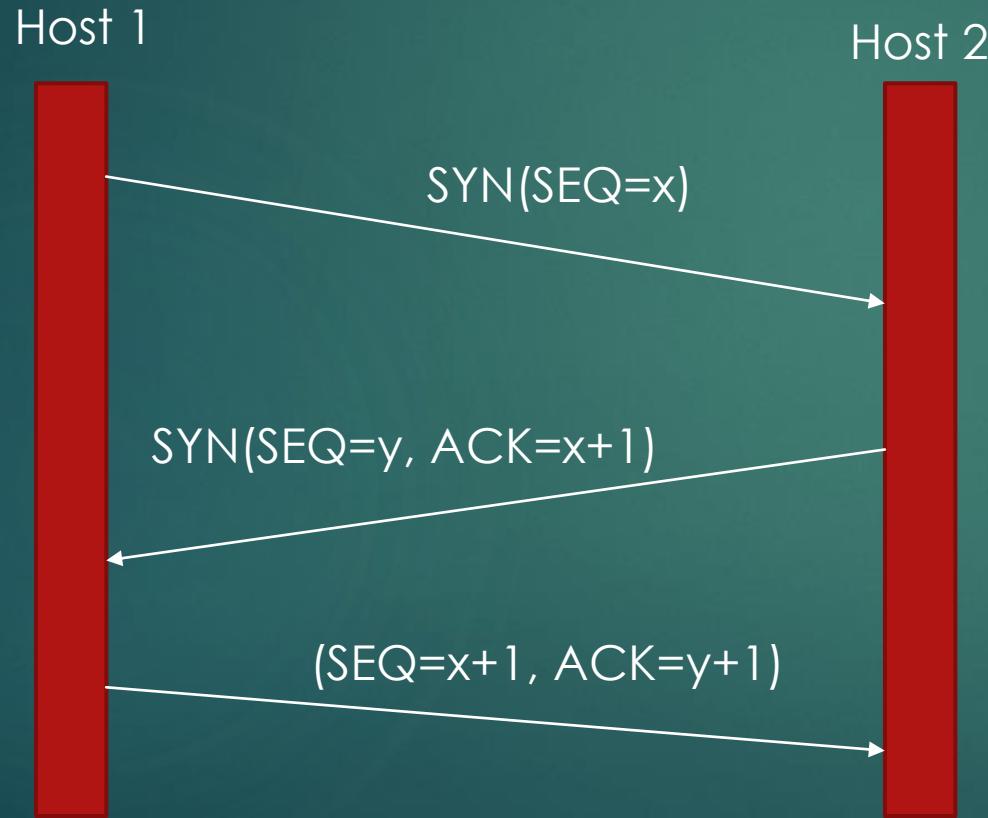
# TCP Connection Management

- ▶ TCP is a connection-oriented transport, so we must deal with connection management
- ▶ The first thing is setting up a connection
- ▶ This is accomplished by one of the hosts sending a packet with the SYN flag set to the other host, the maximum packet size option is usually included in this packet
- ▶ If the other host denies the connection it will send back a packet with the RST flag set
- ▶ If it does accept the connection, it sends back a packet with both the SYN and ACK flags set

# TCP Connection Management

- ▶ Finally the original host sends back a packet with the ACK flag set
- ▶ This is known as a **three way handshake**
- ▶ Both hosts will create new sequence numbers for the connection, each connection has its own sequence of numbers
- ▶ The sequence is not started at 0, to avoid the possibility that there are old packets floating around with the same IP addresses and port numbers, there are also hacks that take advantage of this
- ▶ The host has some algorithm for selecting the starting sequence number, and in general both hosts will be different

# TCP Connection Management



# TCP Connection Management

- ▶ Timers are used in case one of the packets is lost, in this case the connection sequence starts over again
- ▶ Connection release is similar
- ▶ Recall, a socket can be viewed as two separate communications channels, one direction can be closed and the other direction kept open
- ▶ When a process is finished sending data it sends a packet with the FIN flag set
- ▶ In response it will get a packet with the ACK flag set, at this point it can shut down that half of the socket

# TCP Connection Management

- ▶ When the other host is finished it will send a packet with the FIN flag set, and the first host will send back a packet with the ACK flag set
- ▶ Both processes will wait for a short period of time and the connection will be shut down, this is to handle any late packets that are floating around
- ▶ This normally takes four packets, but the second host can send a FIN and ACK in the same packet reducing the packet count by 1
- ▶ If the ACK isn't received after an appropriate delay, the host takes down the connection
- ▶ This avoids the problem of lost ACKs keeping the connection open infinitely long

# Buffer Management

- ▶ TCP must buffer packets on either side of the communications in case a packet is lost or they arrive out of order
- ▶ The sending side knows how much buffer space it has and won't accept data for transmission if there is no buffer space
- ▶ The write call in the application will block
- ▶ The sending side doesn't automatically know how much buffer space the receiver has, if it overruns its buffers packets will be lost
- ▶ To deal with this the sending side sends back its available buffer space in the ACK packet, this is the window size field

# Buffer Management

- ▶ When the connection starts the sender assumes the receiver has buffer space for the maximum size packet they negotiated
- ▶ After that on each ACK the receiver informs the sender of available buffer space
- ▶ The sender knows how much data it can send
- ▶ If the buffer space ever reaches zero, the sender must stop, they are blocked at this point
- ▶ The receiver will then wait for the application to consume some of the data
- ▶ At this point the receiver will send an ACK with the amount of buffer space and the sender can continue

# Buffer Management

- ▶ As it stands there are a couple of problems
- ▶ What happens if the ACK gets lost after buffer space has been cleared? Does the sender block forever
- ▶ This is a potential deadlock
- ▶ This is solved by the sender using a window probe, this is a one byte packet that is basically checking on the buffer size
- ▶ In this case the sender should respond immediately with an ACK, it will inform the sender how much buffer space it has
- ▶ The sender can periodically send window probes until buffer space becomes available

# Buffer Management

- ▶ Since this is a byte oriented protocol, the sender can write one byte at a time and the reader can read one byte at a time
- ▶ This can lead to a waste of bandwidth, a packet with one byte of data is 41 bytes long, 40 bytes of header plus one byte of data, the data link layer could add more pad bytes, plus an ACK packet could contain no data
- ▶ There are several techniques for dealing with this
- ▶ One easy approach is for the receiver to wait until it has data for the sender and add the ACK flag to this packet
- ▶ Clearly there needs to be a timeout on this otherwise we will again get into a deadlock situation

# Buffer Management

- ▶ Now to deal with sending one byte at a time
- ▶ One solution to this is **Nagle's algorithm**
- ▶ At the start, we let the sender send a single byte, but the sender then buffers until an ACK is received from the receiver
- ▶ It will then send the buffered data and again wait for an ACK
- ▶ The key idea is to wait until there is enough data buffered to make it worthwhile to send a packet
- ▶ Doesn't work well with interactive applications, will delay the response from the sender
- ▶ The TCP\_NODELAY socket option is used to turn this off

# Buffer Management

- ▶ The flip side of the coin is **silly window syndrome**
- ▶ If the receiving application removes 1 byte from a full buffer, the receiver can send an ACK with a window size of 1
- ▶ The sender can then send one byte, since that's the amount of buffer space
- ▶ If the receiving process has this one byte read is a loop, we end up with a lot of packets with very little data transmitted
- ▶ The solution is to wait until the buffer is half empty, or has enough space for the maximum packet size before sending the ACK

# Summary

- ▶ Examined the general issues at the transport layer
- ▶ Examined the UDP protocol, not much to add on top of IP
- ▶ Examined the TCP protocol:
  - ▶ Connections management
  - ▶ Buffer management

# CSCI 3210

# Application Layer

MARK GREEN  
FACULTY OF SCIENCE  
ONTARIO TECH

# Introduction

- ▶ There are many application layer services, anyone can write one
- ▶ Will examine two of the more important ones from a network management point of view:
  - ▶ DNS
  - ▶ DHCP
- ▶ Both of these are used in network configuration

# DNS

- ▶ Domain Name System (DNS) is a crucial part of the Internet
- ▶ IP only works in terms of IP addresses, either 32 bit or 128 bit
- ▶ These are not good for human consumption, there needs to be a better way of naming hosts on the Internet
- ▶ A domain name based system has been developed, which are human readable names
- ▶ The DNS converts these names into IP addresses

# DNS

- ▶ Start by looking at an example of a domain name:  
graphics.science.uoit.ca
- ▶ Domain names are read from right to left and consist of names separated by decimal points
- ▶ The high level domain in this example is ca, this is the domain for Canada
- ▶ The next level down is uoit, which is one of the domain names for the university
- ▶ Note that a single institution can have multiple domain names

# DNS

- ▶ The third level name is science, which identifies our faculty
- ▶ Finally graphics identifies my research lab within the faculty of science
- ▶ Note how the domain names are hierarchical, start with the top level domain and work down to a specific site
- ▶ The domain name system is organized based on this hierarchical structure
- ▶ The individual names consist of letters, digits and hyphens, and must be less than 63 characters long
- ▶ The case of the letters doesn't matter

# DNS

- ▶ The DNS is a distributed database implemented by a large number of servers, called name servers
- ▶ Again, this is organized hierarchically
- ▶ For the time being we will assume that the DNS just contains the mapping from domain names to IP addresses
- ▶ The software on a computer that interacts with the DNS is called a resolver, we have been using getaddrinfo as our interface to the resolver
- ▶ You give the resolver a domain name, and it returns the IP address for it

# DNS

- ▶ So how does the resolver work?
- ▶ There are too many computers on the Internet for one name server to have all the names, the database would be too large and searching would be too slow
- ▶ Instead queries are handled hierarchically
- ▶ There are a small number of root name servers that know all of the high level domains
- ▶ A request for `graphics.science.uoit.ca` would start at a root name server, which would return the name server for the `ca` domain

# DNS

- ▶ The request is then passed to the ca domain and it would look up the name server for uoit
- ▶ The uoit name server would then lookup the name server for science, which could finally return the IP address for graphics
- ▶ The science name server is maintained locally in the Faculty of Science, it contains all the information about the computers in the faculty
- ▶ The uoit name server is maintained at the university level, it doesn't know about the computers in the faculty, it only knows its name server

# DNS

- ▶ Similarly the ca name server knows about uoit and ontariotechu, but not about anything inside the university
- ▶ There are two ways this could be implemented:
  - ▶ The root name server could handle querying all the lower level name servers and return the IP address
  - ▶ The local name server starts at ca, gets the name server for the next level and works its way down
- ▶ Clearly the first approach would place too much of a load on the root name servers

# DNS

- ▶ There are a very large number of name server queries on the Internet every second, there needs to be an efficient way of handling them
- ▶ First UDP is used as the underlying protocol, this saves a large number of packet exchanges
- ▶ Second, there are multiple name servers at each level, referred to as primary and secondary name servers
- ▶ The primary name server is usually the authoritative name server, this is the one maintained by the system manager
- ▶ It is the one that is the authority on the domain

# DNS

- ▶ The secondary name servers have a copy of the database
- ▶ The primary name server periodically sends them database updates
- ▶ The query load can be divided between the primary and secondary name servers
- ▶ Another optimization is that name servers cache results, so they rarely start their search at the root name server
- ▶ To handle updates to the Internet, each cached entry has a time to live, after that time the entry is deleted
- ▶ This time is typically on the order of one day

# DNS

- ▶ What does the DNS store?
- ▶ It is a sequence of records, usually stored in a binary form, but easily converted to text:

Domain\_name time\_to\_live class type value

- ▶ They are basically sorted by domain name
- ▶ The time to live is in seconds, used to delete old records
- ▶ The class field is normally IN, which stands for Internet, DNS has been expanded to store information on other domains
- ▶ The type is the type of data stored, and value is its value

# DNS

- ▶ The common record types are:

| Type  | Meaning        | Value                        |
|-------|----------------|------------------------------|
| A     | IP4 address    | 32 bit number                |
| AAAA  | IP6 address    | 128 bit number               |
| MX    | Mail exchange  | Address of mail server, etc. |
| CNAME | Canonical name | Domain name                  |
| PTR   | Pointer        | Alias for IP address         |

# DNS

- ▶ The PTR field assists with reverse DNS, given an IP address find the domain associate with it
- ▶ A domain can have many records associated with it
- ▶ Typically there will be at least two IP addresses associated with it, one for IP4 and one for IP6
- ▶ There is usually an MX entry if the domain accepts email
- ▶ This entry can also be used for blacklisting email servers
- ▶ There is quite often an entry that provides a description of the domain and the system manager

# DNS Tools

- ▶ There are quite a few tools that have been developed over the years for examining DNS
- ▶ One of the early ones is nslookup, it is available on most operating systems, including windows
- ▶ This isn't the easiest of the tools to use
- ▶ More modern tools are dig and host, which are specific to Linux
- ▶ The next few slides show some of the output from dig

# DNS – dig uoit.ca any

```
mark@MSI: ~/network/example3
;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;uoit.ca.          IN      ANY

;; ANSWER SECTION:
uoit.ca.        85476   IN      MX      10 bsf2.dc-uoit.net.
uoit.ca.        85476   IN      MX      30 smtpix.dc-uoit.net.
uoit.ca.        85476   IN      MX      10 bsf3.dc-uoit.net.
uoit.ca.        85476   IN      TXT    "v=spf1 a:bsf2.dc-uoit.net a:bsf3.dc-uoit.net a:smtpo0.dc-uoit.
net a:smtpo1.dc-uoit.net a:smtpo-z.dc-uoit.net -all"
uoit.ca.        85476   IN      TXT    "adobe-idp-site-verification=68771dde9e8a95b4735f432a184f5c7c86
ec67a06e1d7ec6feaebb522bd0b01b"
uoit.ca.        85476   IN      TXT    "spf2.0/pr"
uoit.ca.        85476   IN      SOA    dnsa.dc-uoit.net. teamadmin.dc-uoit.net. 2021030901 10800 3600
60 86400
uoit.ca.        85458   IN      A       205.211.180.149
uoit.ca.        85458   IN      A       205.211.180.148
uoit.ca.        38939   IN      NS     ns1.d-zone.ca.
uoit.ca.        38939   IN      NS     dnsa1.dc-uoit.net.
uoit.ca.        38939   IN      NS     ns2.d-zone.ca.
uoit.ca.        38939   IN      NS     dnsa0.dc-uoit.net.

;; AUTHORITY SECTION:
uoit.ca.        38939   IN      NS     dnsa1.dc-uoit.net.
uoit.ca.        38939   IN      NS     ns2.d-zone.ca.
uoit.ca.        38939   IN      NS     ns1.d-zone.ca.
uoit.ca.        38939   IN      NS     dnsa0.dc-uoit.net.
```

# DNS – dig science.uoit.ca any

```
mark@MSI: ~/network/example3
;; QUESTION SECTION:
;science.uoit.ca.          IN      ANY

;; ANSWER SECTION:
science.uoit.ca.        2836    IN      SOA     dnsa.dc-uoit.net. richard\drake.uoit.ca. 2020100600 7200 900 6
04800 3600
science.uoit.ca.        2836    IN      A       205.211.180.148
science.uoit.ca.        2836    IN      A       205.211.180.149
science.uoit.ca.        2836    IN      SPF     "v=spf1 mx -all"
science.uoit.ca.        2836    IN      TXT    "v=spf1 mx -all"
science.uoit.ca.        2836    IN      MX      10 mail.science.uoit.ca.
science.uoit.ca.        2836    IN      NS      dnsa1.dc-uoit.net.
science.uoit.ca.        2836    IN      NS      dnsa0.dc-uoit.net.

;; AUTHORITY SECTION:
science.uoit.ca.        2836    IN      NS      dnsa1.dc-uoit.net.
science.uoit.ca.        2836    IN      NS      dnsa0.dc-uoit.net.

;; ADDITIONAL SECTION:
dnsa0.dc-uoit.net.      57119   IN      A       205.211.180.197
dnsa1.dc-uoit.net.      57119   IN      A       205.211.181.197

;; Query time: 622 msec
;; SERVER: 192.168.219.1#53(192.168.219.1)
;; WHEN: Tue Mar 16 16:09:54 EDT 2021
;; MSG SIZE  rcvd: 317

mark@MSI:~/network/example3$
```

# DHCP

- ▶ In order to communicate a computer must have an IP address
- ▶ An IP address can manually be assigned to a computer, along with the gateway, mask and DNS servers
- ▶ This requires some knowledge of the network and can be a pain to administer
- ▶ This won't work for most home networks
- ▶ This is solved by Dynamic Host Configuration Protocol or DHCP
- ▶ This is a server on a network that is used to assign IP addresses to computers using a standard protocol – over UDP

# DHCP

- ▶ When a computer boots without an IP address it broadcasts a DHCP discover message on its local network segment
- ▶ A DHCP server will respond with an IP address and the information required to configure the computer to use the Internet
- ▶ The IP address will have a lease associated with it, the length of time that the computer can use that address
- ▶ When the lease expires the computer must renew the lease
- ▶ This way the DHCP server can harvest IP addresses that are no longer in use

# DHCP

- ▶ On the server side the system manager will give the DHCP server a range of IP addresses to allocate
- ▶ Some addresses on the network may remain statically assigned
- ▶ This is typically the case for servers, or printers
- ▶ There are several ways these addresses can be allocated
- ▶ In a relatively static environment the DHCP server will attempt to assign the same IP address to a computer each time it boots
- ▶ This is the case with home networks

# DHCP

- ▶ In larger organization this typically doesn't happen
- ▶ The university has a fixed pool of IP addresses, it doesn't have enough to assign one to each student
- ▶ Your home router is both a DHCP client and server
- ▶ When it starts it will request an IP address from your ISP, it will probably get the same address each time that it starts
- ▶ It will also receive configuration information from the ISP, such as the DNS to use and potentially some network parameters

# DHCP

- ▶ Your home router than serves as a DHCP server for your home network
- ▶ This is all automatic, you just plug in the router and it works
- ▶ Some home routers give you some control over DHCP
- ▶ They allow you to assign static addresses to some devices, and control the range of IP addresses available for automatic allocation
- ▶ Static addresses are often required for port forwarding

# Summary

- ▶ Covered the two main application layer protocols that are used for network configuration
- ▶ These protocols have evolved over the years to provide automatic configuration for standard network usage
- ▶ What allows you to plug in a home router and have it work right out of the box

# CSCI 3310

# Introduction to

# Security

MARK GREEN  
FACULTY OF SCIENCE  
ONTARIO TECH

# Introduction

- ▶ There are two key principles behind security
- ▶ First, very few systems are designed with security in mind
- ▶ Security is often an after thought, usually after some disaster has occurred
- ▶ The only well known system that I know that was designed with security in mind was Blackberry Messenger
- ▶ Even military systems are often designed without security in mind
- ▶ There is a rush to get things done, security doesn't seem like an important feature, until after it is needed

# Introduction

- ▶ Second, most security problems are human related
- ▶ You can have the very best technology, but if the people in the system aren't careful, the security technology doesn't matter
- ▶ This was well known before computer systems
- ▶ The modern phishing attacks are good examples of this
- ▶ You fool someone on the inside into giving up important information that allows you to crack the system

# Analysis

- ▶ Start with an analysis of the security situation:
  - ▶ What can be attacked?
  - ▶ Who is the attacker?
- ▶ Once you've answered these questions you can come up with a security solution
- ▶ This solution may or may not be technical, or it could be a combination of the two

# What?

- ▶ The first thing is to identify what attackers might be interested in
- ▶ Now the obvious thing is information, they are after information that is stored on computer systems
- ▶ We hear about this every day in the media
- ▶ Note, very few security hacks are actually reported, most organizations keep them secret
- ▶ What we hear is a poor indication of what may actually be happening in the real world
- ▶ Still, information theft is probably the most important

# What?

- ▶ Theft of computer and communications is a crime in most countries
- ▶ This dates from before PCs, where people would hack into systems to steal computer time
- ▶ This still occurs, but now the interest is in using these computers to attack other computers
- ▶ Only a dumb hacker uses there own computer to attack other computers, this is far too easy to trace
- ▶ You want to launch your attacks from someone else's computer, so it's harder to trace you

# What?

- ▶ What kind of attack would this be?
- ▶ The most common one is a denial of service attack, prevent other people from accessing a resource
- ▶ This usually takes the form of overloading a server with garbage messages
- ▶ It is so busy processing the garbage that it doesn't have time to process real request
- ▶ In the extreme the service will be take down
- ▶ In a lot of cases these are politically motivated

# What?

- ▶ In a new form of this is ransomware attacks
- ▶ The attacker penetrates a system, encodes all the critical data and charges for the key to decrypt it
- ▶ This is becoming more common, and the motivation is money
- ▶ These attacks involve gaining access to a computer system
- ▶ This is what we normally think of, but there are a whole range of attacks where the attacker doesn't need to gain access to the system

# What?

- ▶ These types of attacks involve authentication
- ▶ When a bank receives a request to transfer money from one account to another how does it know that this is an authentic request?
- ▶ How do we determine if a request is real
- ▶ This isn't a new type of attack, it doesn't need a computer
- ▶ A well known example of this is the fake invoice request, where a criminal sends a fake invoice to a company hoping it will be paid

# What?

- ▶ This can happen with electronic transactions as well
- ▶ This is the authentication of documents or transactions
- ▶ Another form is the authentication of a person or a computer, the source of information
- ▶ In a face-to-face transaction you have confidence in who the other person is
- ▶ But, in an electronic transaction, how do we really know?
- ▶ A recent example of this is deep fakes
- ▶ Placing an innocent person at the scene of a crime

# What?

- ▶ Without proper authentication a customer could claim they never ordered a service after they received it
- ▶ How can the supplier counter this claim?
- ▶ This is one of the uses of blockchain technology, track the trail of a request in a way that can't be forged or tampered with
- ▶ Another example is identity theft, where someone obtains a loan or service in your name
- ▶ This could come from stealing information from a website

# Who?

- ▶ There are three main groups of hackers
  - ▶ Countries
  - ▶ Professionals
  - ▶ Script kiddies
- ▶ Countries have been in this business for thousands of years
- ▶ They have been spying on each other as a way of achieving a military advantage
- ▶ Sometimes they are interested in commercial advantages
- ▶ Can have significant resources

# Who?

- ▶ This is probably the most difficult to defend against
- ▶ Some countries have special training centers for hackers, viewed as part of the military
- ▶ For this group, resources are not a problem, quite often they will over value information
- ▶ There can be an ideological twist to this that makes it hard to defend against
- ▶ Example, fake information distributed over social media channels

# Who?

- ▶ The professional hacker is more recent
- ▶ This probably started with industrial espionage, spying on other companies to find trade or marketing secrets
- ▶ This is probably the main group of hackers now, they are in it for the money
- ▶ In a way this is also the easiest group to defend against
- ▶ They want to make a profit off of their hack, they aren't doing this for their country, some political belief or the fun of it

# Who?

- ▶ The way to deal with this group is to make it too expensive to steal the data
- ▶ It costs a hacker to break into a system, it takes time and computing resources
- ▶ They intend to sell or profit in some way from the information they steal
- ▶ If the value of the information is less than the cost of obtaining it, they are losing money
- ▶ Why websites advertise that they don't store credit card data, they become unattractive to hackers

# Who?

- ▶ The last category has been around for many decades as well, they are basically doing it for fun, the thrill, etc.
- ▶ They are called script kiddies because they are typically younger and just copy scripts off of websites, they usually don't know how they work
- ▶ This group usually doesn't cause much damage, but they still need to be dealt with
- ▶ Some security people view them as a waste of time, but they could be identifying a weakness in the system

# Analysis

- ▶ Now that we know the what and the who, we can determine what needs to be protected, and the type of protection required
- ▶ Identify the data that must be protected and for how long
- ▶ Some data is public information, we don't need to take any steps to protect it, or information that won't be a problem if its released to the public
- ▶ Other information has a finite lifetime, after that it is worthless
- ▶ Company financial information must be kept secret until it is officially released, same with the federal budget
- ▶ Before release this data is very valuable, after release worthless

# Analysis

- ▶ Once the lifetime has been determined, we know how to protect it
- ▶ If the lifetime of the data is two weeks, and we encrypt it with a technique that requires three weeks to break, we are safe
- ▶ This works well for data with a short lifetime, say several years
- ▶ Over that time span we can be confident of the time required to break codes, the technology doesn't change that fast
- ▶ For other data, it must be kept secure for 30 to 50 years, this is often the case with government and military data

# Analysis

- ▶ In the past this data was kept on paper and stored in a very secure location, thick walls, armed guards, etc.
- ▶ Now this data is stored electronically on computer systems, it can be stolen, how do we protect it when it's out in the wild
- ▶ The answer has been cryptography, but many unbreakable codes have been broken within a decade of their proposal
- ▶ How do we know whether a code will still be secure 50 years from now?
- ▶ Technology can evolve a lot in that time

# Analysis

- ▶ Who have legitimate access to the data and for what purpose?
- ▶ This can be a difficult one to deal with
- ▶ Ideally you want as few people to have access as possible
- ▶ Diplomatic and military organizations have sophisticated security classifications, but sometimes they have problems
- ▶ Consider medical records, who has access to the medical data on a particular patient?
- ▶ We want to restrict this to the people who are currently treating the patient and they may only have access to a subset of the data

# Analysis

- ▶ For each piece of data we will end up with:
  - ▶ The lifetime of the data
  - ▶ Who can access it and for what purpose
  - ▶ When and how it should be destroyed
- ▶ The last one is interesting, for paper document they can be shredded, but what about electronic ones?
- ▶ Easy to create a copy of an electronic document, need to keep track of these copies
- ▶ Central data stores can simplify this process

# Analysis

- ▶ How do you destroy electronic documents?
- ▶ You can't just delete the file, the disk blocks will just be added to the free list, can examine the free list to get the data
- ▶ Some OS's save deleted files in the trash bin, easy to recover
- ▶ Non destructive techniques involve zeroing or reformatting the disk
- ▶ The most reliable technique is to physically destroy the disk drive
- ▶ A common technique is to drill holes through the disk drive, if the holes a large enough it can't be reconstructed

# Analysis

- ▶ Stored data is the easiest one to deal with
- ▶ Next is unauthorized access to equipment, this is a version of authentication
- ▶ How do we identify the person trying to access the system?
- ▶ The historical technique is to use the user name and password, this is currently the most common approach
- ▶ The user name is viewed as common knowledge, but the password is secret
- ▶ There is an assortment of problems with this approach

# Analysis

- ▶ How do we store passwords? If they are stored as plain text anyone with access to the system could potentially determine everyone's password
- ▶ Most systems store an encoded version of the password, they never store the plain text
- ▶ Example of only storing the information that is necessary
- ▶ How do we communicate the password to the system?
- ▶ The password will be sent as plain text, the network connection must be secure, or anyone can read it

# Analysis

- ▶ Many passwords are easy to crack, there is a small number of them that are quite common
- ▶ Hackers use a dictionary attack, a file of common passwords and just try all of them, this can be surprisingly effective
- ▶ Many systems force passwords to be difficult to guess, expand the range of characters that must be used, make sure they aren't in a dictionary
- ▶ Another approach is to force users to change their passwords frequently, this just annoys users and probably results in weaker passwords

# Analysis

- ▶ A better approach is to use tokens
- ▶ A token is a large binary string that is sent between systems to identify a user
- ▶ Replaces user name and password
- ▶ Some can be as short as 256 bits, but they are usually much longer
- ▶ In some systems the token is static, but other systems use dynamic tokens that change regularly, in that case if the hacker copies the token it's useless
- ▶ Tokens are computer generated, little to no user involvement

# Analysis

- ▶ Token systems assume that the user starts with a secure system, they have already been authenticated to that system
- ▶ This could be a personal device, or a corporate server
- ▶ The token is generated on the secure device and sent to the system the user wants to connect to
- ▶ The system at the other end knows how to validate the token, this is usually an algorithmic process that involves several pieces of secret data

# Analysis

- ▶ There are two main advantages to this approach:
  - ▶ They are much more secure than passwords
  - ▶ There are no user names, can't determine who is talking to who
- ▶ Biometrics have also been used
  - ▶ They can be quite secure, but you need to have physical access to the user, they are difficult to use over a network
  - ▶ Some of the biometric systems can be fooled, and sometimes they fail to recognize the person

# Analysis

- ▶ At this point we know the following:
  - ▶ Who can access the system
  - ▶ What they can access
  - ▶ How they are authenticated
- ▶ Now we get to the problem of validating the information that is flowing over the network
- ▶ First of all, do we let requests or information come into our system from a network?

# Analysis

- ▶ We may have no choice in this, if we are a bank or an ecommerce site we need to be able to accept network transactions
- ▶ The base level for this is the user has already authenticated and is using a “secure” channel for communications
- ▶ This is the approach used for ecommerce sites and commercial banking
- ▶ The transactions of are of low value, so not a profitable target
- ▶ Some systems require separate validation for transaction over a certain amount, usually a text message

# Analysis

- ▶ For larger transactions they must be digitally signed
- ▶ A way to verify if each transaction is correct, related to cryptography
- ▶ Unfortunately, many of the digital signing techniques have been broken, they are easier to break than cryptographic codes
- ▶ There are two techniques that are secure
- ▶ The first is the use of blockchain technology
- ▶ It is not only used for crypto currencies, but for any transaction where you need a secure record

# Analysis

- ▶ The other technique is to use tokens
- ▶ For non-public transaction, you can restrict the networks that you will accept transactions from
- ▶ Example, a private banking network for large transactions
- ▶ At this point we've determined:
  - ▶ The transaction that will be accepted
  - ▶ How they will be validated
  - ▶ Where we will accept them from

# Summary

- ▶ Basic understanding of security and how we can go about doing a security analysis of a system
- ▶ The analysis should be done before the system is designed, and definitely before it is implemented
- ▶ Now move on to how security can be implemented

# CSCI 3310

# Cryptography

MARK GREEN  
FACULTY OF SCIENCE  
ONTARIO TECH

# Introduction

- ▶ Cryptography dates back over 2000 years, it isn't a new idea
- ▶ Many ciphers have been developed, and most of them have been broken
- ▶ Some have nice theoretical backgrounds, but are vulnerable in practice
- ▶ Quite often gives people a false sense of security, I've encoded the information so no one can read it
- ▶ Ciphers are an important part of the security system, but they are only part of it

# Introduction

- ▶ In WWII the Germans thought their Enigma code was unbreakable
- ▶ But, the British cracked it in the early part of the war, the Germans never found out about it
- ▶ This was a key part of their defeat
- ▶ The same thing happened with the Japanese codes
- ▶ Both countries thought they were sending secret messages, but they were all being read by the other side
- ▶ Never be too confident of your ciphers

# Introduction

- ▶ Be careful what you encrypt, the more information that is encrypted the easier it is to break the code
- ▶ Definitely don't encrypt public information, or things that a hacker could guess
- ▶ Again in WWII, the Germans encrypted their weather reports to their submarines
- ▶ Weather is public information, assisted with breaking the code
- ▶ The weather was also useful in detecting the position of the German submarines

# Basics

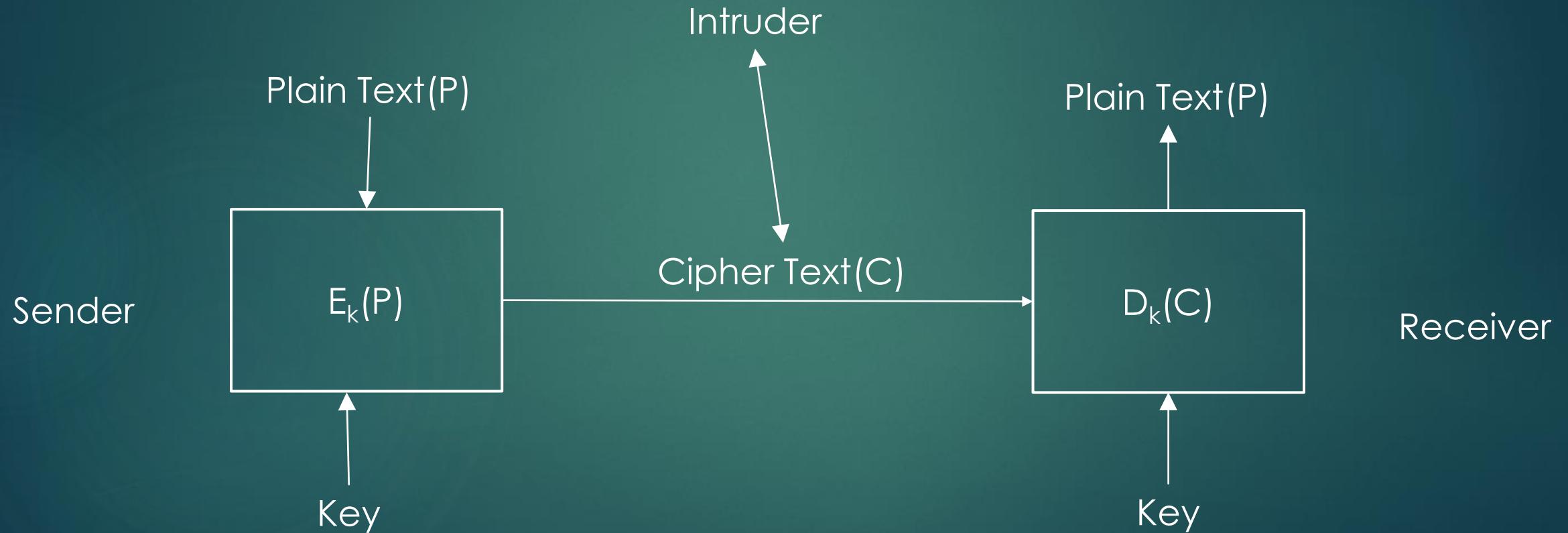
- ▶ Start with some terminology and basic techniques
- ▶ The unencoded text is call plaintext, while the encoded text is called ciphertext
- ▶ There is an algorithm for encrypting and decrypting messages, it is assumed that this algorithm is well known
- ▶ The mistake the Germans made was thinking their Enigma algorithm was secret, but it was based on a commercial product, so it really wasn't secret
- ▶ A key parameterizes the algorithm, this is the part that is kept secret

# Basics

- ▶ It is the key that must be kept secret, there must be a large number of possible keys, too hard to guess the key
- ▶ Breaking the code is basically determining the key
- ▶ Next slide shows the situation
- ▶ The plain text is  $P$ , the cipher text is  $C$
- ▶ The function that encrypts the message is  $E_k$ , where  $k$  is the key
- ▶ The function that decrypts the message is  $D_k$
- ▶ We have the following:

$$D_k(E_k(P)) = P$$

# Basics



# Basics

- ▶ We also assume that there is an intruder that is monitoring the communications
- ▶ If our key is secret the intruder cannot decrypt the message, so it looks like the communications is safe
- ▶ But, is it?
- ▶ If the intruder is only interested in reading the messages, yes it is safe
- ▶ But, what else could the intruder do?
- ▶ It can save the messages and inject them back into the message stream, called an injection attack

# Basics

- ▶ If the messages were bank transactions, this would cause transactions to occur that shouldn't
- ▶ The intruder could watch what the receiver does when it receives a message
- ▶ Over time the intruder could develop a mapping between the encoded message and the receiver's actions
- ▶ The could be used to cause the receiver the perform actions the sender didn't intend
- ▶ The receiver could also determine what different parts of the messages meant

# Basics

- ▶ I don't need to know your password, I only need to know the encrypted form of your password that you send to other systems
- ▶ We need to make sure that saved messages are useless
- ▶ This is a problem if our encryption algorithm always produces the same encryption when we give it the same plain text
- ▶ There are several ways around this problem
- ▶ The first is to make sure that the messages are fresh, that is the message changes with time, so we can detect old messages
- ▶ The easiest way to do this is to add a time stamp to each message

# Basics

- ▶ If the receiver gets a message that is much older than the transmission time it can drop the message
- ▶ This could be an intruder, but it could also be due to a network error
- ▶ Another approach is to add some extra information to the message that really isn't required
- ▶ For example, yesterdays baseball scores, or some other random data
- ▶ This makes it more difficult for the intruder to correlate messages with the resulting actions, the message will be different each time

# Basic Ciphers

- ▶ Most ciphers are constructed from a small number of basic building blocks, there are only 3 of them
- ▶ Two are well known ciphers that have been used for centuries
- ▶ The first is the substitution cipher
- ▶ This replaces a letter by another letter in the alphabet
- ▶ The key is the mapping for each of the letters
- ▶ In the case of English this would be a 26 letter key, if only one case is used
- ▶ At first this look pretty safe, there are  $26!$  possible keys and an intruder may need to try all of them to find the correct key

# Basic Ciphers

- ▶ This is the first example of where the theory leads us to an incorrect result
- ▶ We don't need to try all of the keys, in fact we can break the code much faster
- ▶ The letters in English have different frequencies, how often they occur in normal text
- ▶ These statistics are well known, and this occurs in other languages as well
- ▶ This is all the information a hacker needs to break a substitution code

# Basic Ciphers

- ▶ The hacker accumulates a reasonable amount of encrypted text and starts computing letter frequencies
- ▶ The hacker then compares these letter frequencies to the known English letter frequencies and matches the encrypted letters to the known English letter frequencies
- ▶ This process converges to the correct key quite quickly
- ▶ One way to avoid this is to change the key frequently, that way the hacker won't have enough encrypted text to determine the key
- ▶ Both parties now need to know the key sequence, this could be a lot of information, called a code book

# Basic Ciphers

- ▶ The other ancient cipher is the permutation cipher
- ▶ In this approach instead of changing the letters, the order of the letters is changed
- ▶ A fixed length block of plain text is permuted to produce a fixed length block of cipher text
- ▶ The message is encrypted one block at a time
- ▶ The key in this case is the permutation that is applied to a block of text
- ▶ The size of the block is determined by the length of the key

# Basic Ciphers

- ▶ The longer the key the more secure the cipher appears to be
- ▶ If there are more possible keys, it takes longer to check all of the possible keys
- ▶ Again a statistical approach can be used, but it needs to be a bit more sophisticated
- ▶ Instead of examining individual letter frequencies, the hacker works with pairs and triplets of letter frequencies, known as digraphs and trigraphs
- ▶ Again these frequencies are well known

# Basic Ciphers

- ▶ The hacker needs a larger sample of encrypted text, but can still use the same basic approach to determine the key
- ▶ With electronic communications a program can be written to do this analysis and it can quite quickly determine the keys
- ▶ The third basic cipher cannot be broken, this is the XOR cipher
- ▶ Consider the exclusive or (XOR) operation
- ▶ Interesting property: XOR is its own inverse:

$$(X \text{ XOR } 1) \text{ XOR } 1 = X$$

$$(X \text{ XOR } 0) \text{ XOR } 0 = X$$

# Basic Ciphers

- ▶ Start by converting the message into a bit string, for electronic communications that has already been done for us
- ▶ For the key have another bit string that is at least as long as the message
- ▶ Now perform a bit wise XOR between the message and the key
- ▶ The resulting message will look like random garbage
- ▶ The receiver then performs the same bit wise XOR operation with the same key to retrieve the message
- ▶ This can be implemented very easily and efficiently

# Basic Ciphers

- ▶ Now consider a hacker, they start guessing keys and applying them to the encrypted messages
- ▶ They examine the result for something that makes sense, if they get a result that is a proper English sentence they assume they have found the correct key
- ▶ With the XOR cipher, many keys will produce reasonable results, just not the message that was sent
- ▶ Even if the hacker guesses the correct key, they won't know, since many keys produce reasonable results

# Basic Ciphers

- ▶ This looks like we've solved the problem, we can end the lecture now
- ▶ Except there is one major problem, the key must be at least as long as the message to be sent, and it appears to be a random sequence of bits, hard to memorize
- ▶ How do we securely transfer the key to the receiver?
- ▶ This is exactly the same problem as securely transmitting the message to the receiver
- ▶ We really haven't solved the problem, we've just replaced one problem by an equally hard problem

# Real World Ciphers

- ▶ There is both a technical and human side to real world ciphers, they are equally important
- ▶ The human side revolves around trust
- ▶ Very few people understand the detailed mechanics of modern ciphers, must be able to trust the people who develop them
- ▶ We are interested in ciphers that become standards, so they are widely implemented and used
- ▶ Ecommerce sites would not work well if there wasn't a standard cipher

# Real World Ciphers

- ▶ Consider something like https, the secure version of http
- ▶ All the web browsers and servers must agree on the cipher they are using, otherwise they won't be able to communicate
- ▶ We need to have one standard that everyone trusts and uses in their implementation of https
- ▶ What happens when the cipher is broken (this has happened in the past)?
- ▶ We need to replace the cipher in all the programs that use it
- ▶ In the past this has taken over a decade

# Real World Ciphers

- ▶ How do we know that there isn't a back door into the cipher?
- ▶ Something that would allow a government or criminal to easily spy on our messages
- ▶ Cipher development must occur in an open environment, where the whole process is visible
- ▶ Many ciphers have been developed by companies and individuals, they are not widely used for this reason
- ▶ Without an open process, you need to trust everyone involved in the cipher development

# Real World Ciphers

- ▶ Currently the cipher standards are developed by NIST, national institute of standards in the US
- ▶ This is done using an open competition, the process takes a number of years
- ▶ NIST announces a competition for a new standard, and the basic things that it is interested in
- ▶ They also develop a set of test cases and benchmarks for the candidate ciphers
- ▶ They then open the competition to anyone in the world

# Real World Ciphers

- ▶ Each entry must provide an open description of the cipher and an open source implementation
- ▶ The competition then goes into multiple rounds, where each of the candidate ciphers is evaluated
- ▶ This is done in an open conference where anyone can participate
- ▶ All the candidate ciphers are listed on the NIST website
- ▶ On each round a number of ciphers are eliminated
- ▶ The final round has about 6 ciphers, and the final one is selected from that group

# Real World Ciphers

- ▶ This process was used about 25 years ago to select the current standard, AES
- ▶ The process is being repeated now, since all the current ciphers are vulnerable to attack by quantum computers
- ▶ This competition will produce a cipher standard that can't be broken by quantum computers
- ▶ It is currently in the final round, with the hope of producing a new standard within the next year or so

# Symmetric Key Ciphers

- ▶ The most secure standard ciphers at this time are symmetric key ciphers
- ▶ In this case there is a single key that is used to encrypt and decrypt plain text
- ▶ The key sizes are measured in bits
- ▶ These are block based ciphers, where one block of text is encrypted at a time
- ▶ The block is usually the same size as the key, but this isn't always the case

# Symmetric Key Ciphers

- ▶ All of these ciphers are based on the three basic techniques that we've seen: substitution, permutation and XOR
- ▶ They use all three techniques in different combinations, divided into stages
- ▶ Each stage is driven by part of the key and usually consists of all three techniques, sometimes working on only a part of the plain text
- ▶ Modern ciphers have around 20 stages, even though they are based on simple techniques, the result is quite complex

# Symmetric Key Ciphers

- ▶ Substitution, permutation and XOR can all be efficiently implemented in hardware
- ▶ Implemented at the byte level
- ▶ Substitution is table lookup, only need a 256 byte table
- ▶ Permutation is the permutation of the bits within a byte, basically a switch
- ▶ These operations can be performed very quickly in hardware
- ▶ Don't want the cipher to be a bottleneck in our systems, if it's too slow people won't use it

# Symmetric Key Ciphers

- ▶ The first popular symmetric key cipher was DES, developed by IBM
- ▶ This was a standard for a period of time
- ▶ This cipher was controversial, since it's believed that NSA was involved in its design
- ▶ The standard version seemed to have a back door that allowed the NSA to spy on messages
- ▶ The standard version used a 56 bit key, and it has been broken
- ▶ There are more secure versions of the cipher, but they aren't widely used now

# Symmetric Key Ciphers

- ▶ The current standard is AES (Advanced Encryption Standard)
- ▶ This was developed by a competition launched by NIST and became a standard in November 2001
- ▶ The AES standard supports key lengths of 128, 192 and 256 bits, and the block lengths match the key lengths
- ▶ The 192 bit version is rarely used
- ▶ This standard is widely implemented in many programming languages
- ▶ Intel chips have instructions that support AES, so it can be quite efficient

# Symmetric Key Ciphers

- ▶ On its own AES will always produce the same cipher text given the same plain text as input
- ▶ This occurs at the block level, which isn't a particularly long sequence of bytes
- ▶ Thus, it is usually combined with another technique that breaks this up
- ▶ One of the standard way is Cipher Block Chaining (CBC), so you will often hear the cipher as AES-CBC, sometimes with the bit length included

# Symmetric Key Ciphers

- ▶ CBC is based on an initialization vector and the XOR operation
- ▶ The initialization vector is chosen randomly and is XOR'ed with the first block of plain text before it is encrypted
- ▶ After that each block of plain text is XOR'ed with the previous block's encrypted text before it is encrypted
- ▶ Therefore, the encrypted version of a block of plain text will be different each time it is encrypted
- ▶ The initialization vector can be stored with the key, or it can be transmitted as the first part of the message

# Symmetric Key Ciphers

- ▶ AES-CBC is viewed as secure and can't be broken by most traditional code breaking techniques
- ▶ There are several approaches that have claimed to be able to simplify the code breaking task
- ▶ There are several statistical techniques that can reduce the time required to discover the key
- ▶ There are also techniques that are based on monitoring the electrical power consumption of a device while encrypting
- ▶ Recall: a 0 is represented by 0 volts, and a 1 by positive voltage

# Symmetric Key Ciphers

- ▶ By clever power monitoring you can determine the motion of bits and their values
- ▶ A CPU chip emits low energy electro-magnetic waves, these waves change based on the instruction being executed
- ▶ This can be used to reconstruct the code that is being executed, which will then give the key
- ▶ You need access to the hardware in order to do this
- ▶ But, this shows the length that people will go to in order to break a code

# Symmetric Key Ciphers

- ▶ It is believed that AES can be cracked by a quantum computer
- ▶ This has been known for quite a long time, but has only been a concern recently
- ▶ The rapid advance of quantum computing hardware has raised this as a possibility
- ▶ Some researchers have predicted that within a decade quantum computers will be able to break AES
- ▶ It may take a bit longer than that, but it's coming
- ▶ NIST is now having a competition to develop a quantum safe cipher

# Symmetric Key Ciphers

- ▶ The main problem with symmetric key ciphers is that the key must be kept secret, but shared with all of the parties involved
- ▶ There must be some way of communicating the keys
- ▶ They can't be sent as plain text, they would no longer be secret
- ▶ They need to be encrypted, but now how do we send the key for this new encryption?
- ▶ Another approach is to distribute the key on a USB key, but what happens if that is stolen?
- ▶ This is a major weakness in the approach

# Symmetric Key Ciphers

- ▶ Quantum computing comes to the rescue here
- ▶ There are quantum ciphers that use quantum properties
- ▶ These ciphers can detect if someone is listening to the message and can counter their spying
- ▶ The problem is they are quite slow, cannot be practically used to encrypt a whole message
- ▶ But, they can be used to encrypt a key, called Quantum Key Distribution (QKD)
- ▶ There are commercial QKD devices, they are still on the expensive side

# Public Key Ciphers

- ▶ The main problem with symmetric key ciphers is the need for a secret key
- ▶ If a large number of parties need to communicate they all need to have the secret key
- ▶ But, now all of these parties can decode messages, not only the ones sent to them
- ▶ With many copies of the key there is also a greater chance of it being stolen
- ▶ This is the motivation behind public key ciphers, we make the keys public

# Public Key Ciphers

- ▶ For this to work we need to have two keys
- ▶ One key is public, and will be well known
- ▶ The other key is private, only one party will have it
- ▶ We can also have that the encryption and decryption algorithms are different
- ▶ Consider two parties, Alice and Bob (the standard names used in these types of discussions) that want to communicate
- ▶ To send a message to Alice, Bob encrypts the message using Alice's public key and then sends the message

# Public Key Ciphers

- ▶ This message can only be decrypted using Alice's private key
- ▶ When Alice receives the message, she decrypts it with her private key
- ▶ To send a message to Bob, Alice uses Bob's public key to encrypt the message, which only Bob can decrypt
- ▶ Alice and Bob can put their public keys on their websites, for example, so anyone can send messages to them
- ▶ This solves the problem of distributing keys, the private keys are kept in one place

# Public Key Ciphers

- ▶ One of the best known public key ciphers is RSA, which is based on the difficulty of factoring large numbers
- ▶ It determines its keys in the following way:
  - ▶ Choose two large prime numbers,  $p$  and  $q$ , at least 1024 bits
  - ▶ Compute  $n = p \times q$ , and  $z = (p-1) \times (q-1)$
  - ▶ Choose a number relatively prime to  $z$ , call it  $d$
  - ▶ Find  $e$  such that  $e \times d = 1 \text{ mod } z$
- ▶ The keys are  $d$  and  $e$

# Public Key Ciphers

- ▶ RSA is a block cipher, we divide the message into blocks of  $2^k$  bits where k is the largest integer with  $2^k < n$
- ▶ For a message block P, the cipher text, C is computed in the following way:

$$C = P^e \bmod n$$

- ▶ The message can then be decrypted in the following way:

$$P = C^d \bmod n$$

- ▶ The public information is  $(e, n)$  and the private information is  $(d, n)$

# Public Key Ciphers

- ▶ This cipher is clearly more expensive than something like AES
- ▶ Thus, RSA is usually used to transmit keys for symmetric key ciphers instead of encrypting an entire message
- ▶ RSA also suffers from the problem that each time a plain text message is encrypted it will produce the same cipher text
- ▶ Again this can be solved by a technique like CBC that modifies the plain text before it is encrypted
- ▶ There are other public key ciphers, but RSA has remained the most popular one

# Digital Signatures

- ▶ There are several versions of digital signatures, depending upon what we want to accomplish
- ▶ Many of them are based on public key cryptography
- ▶ If we just need to know whether A signed a document, one way is for A to encrypt the document with their private key
- ▶ Anyone who gets the document can then use A's public key to decrypt the document
- ▶ If A didn't sign the document, this decryption won't work, the result will be garbage
- ▶ So the document can't be counterfeited or forged

# Digital Signatures

- ▶ Normally with a public key cipher we have  
$$D(E(P)) = P$$
- ▶ We also have  
$$E(D(P)) = P$$
- ▶ The RSA algorithm has this property
- ▶ With this we can make the message more secure
- ▶ In the previous approach anyone with A's public key can decrypt the message, so it isn't secret
- ▶ Now, at A's end the message can be decrypted with A's private key and encrypted with B's public key, now only B can read the message

# Digital Signatures

- ▶ At this point we send  $E_B(D_A(P))$  over the network
- ▶ In this case the decryption is done using A's private key
- ▶ Now at the other end B applies  $D_B$  with his private key
- ▶ This give  $D_A(P)$ , which B saves as proof of receiving the message
- ▶ B then applies  $E_A$ , using A' public key and retrieves the original message
- ▶ In this case we are using RSA, which we know is slow, but it does protect the message

# Message Digests

- ▶ What happens if we don't care whether other people can read the message?
- ▶ Example, distributing software, want to make sure that the receivers can use the software, but don't want it modified during transmission
- ▶ We want to protect the message from being tampered with
- ▶ This is the purpose of a message digest, a bit string that we add to the message that is a hash of the message contents
- ▶ The receiver can verify that the message agrees with its message digest, by rehashing it at their end

# Message Digests

- ▶ A message digest MD must satisfy the following properties:
  1. Given P, it's easy to compute  $MD(P)$
  2. Given just  $MD(P)$  it's effectively impossible to determine P
  3. Given P, no one can find a  $P'$  that satisfies  $MD(P') = MD(P)$
  4. A change to the input of even 1 bit produces a very different output
- ▶ Since  $MD(P)$  is much shorter than P, we can use a public key cipher like RSA to protect it

# Message Digests

- ▶ There have been a number of standards proposed for message digests
- ▶ The first one was MD-5
- ▶ This standard is no longer considered to be secure, but it is still widely used
- ▶ Why?
- ▶ It is used in the open source community to check that source code has been delivered correctly
- ▶ If an error occurs over the network the MD-5 signature will change and it can be detected

# Message Digests

- ▶ It takes considerable resources to break MD-5, so for less secure applications it is still valuable
- ▶ There is a family of standards called SHA (Secure Hash Algorithm), developed by NIST
- ▶ The first version, SHA-1 was the replacement for MD-5, but is no longer considered to be completely secure
- ▶ SHA-1 produces a 160 bit message digest
- ▶ Its current secure replacement is the SHA-2 family of message digests
- ▶ This is a number of related algorithms that differ in the number of bits in the digest

# Message Digests

- ▶ The common versions of SHA-2 are SHA-256 and SHA-512
- ▶ There are other versions that use a reduced number of bits
- ▶ Both SHA-1 and SHA-2 are considered by be legal signatures in a number of counties
- ▶ SHA-3 came out of a competition by NIST that started in 2006 and became a standard in 2015
- ▶ MD-5, SHA-1 and SHA-2 are all based on similar approaches
- ▶ It was thought that in the future there could be an attack that would work on all of them

# Message Digests

- ▶ The competition of SHA-3 stated that the algorithm must use a different approach for additional safety
- ▶ SHA-3 is now beginning to appear in cryptographic software packages and will probably become the dominate message digest
- ▶ Hardware acceleration for SHA-3 is starting to appear on some CPUs
- ▶ How can message digests be compromised?
- ▶ If the digest is  $L$  bits long, finding the message that corresponds to a given digest can be done in  $2^L$  evaluations using brute force
- ▶ This can be reduced to  $2^{L/2}$  using several different techniques

# Message Digests

- ▶ Clearly, the longer the digest the more secure it is
- ▶ There is some thought that these standards can also be broken by quantum computers
- ▶ The effectiveness is questioned by some researchers
- ▶ It has been suggested that Grover's algorithm could be used for this
- ▶ In theory that would reduce the time to crack the digest by a factor of 2
- ▶ This can be remedied by just increasing the size of the digest by a factor of 2

# Message Digests

- ▶ We can further secure a message digest by encrypting it
- ▶ Since they are short, public key ciphers are not a problem
- ▶ Without encryption, someone could change the document and then recompute the message digest
- ▶ With encryption the public key can be used to decrypt the message digest
- ▶ The message digest for the document can be computed, and if they don't match we know that the document has been modified
- ▶ Anyone tampering with the document will not know the private key used to encrypt the message digest

# Public Key Infrastructure

- ▶ There is also a problem with distributing public keys
- ▶ How do I know the public key of another person?
- ▶ Say A wants to communicate with B, but doesn't know B's public key
- ▶ B could put their public key on their website, so A can read it
- ▶ Now A needs to issue a http request to get the key
- ▶ An intruder can intercept this request, instead of returning B's public key, they return their own public key
- ▶ Now the intruder can read all of the messages

# Public Key Infrastructure

- ▶ Our intruder is smart, he reads all the messages and then sends them to B, using B's public key
- ▶ B then thinks he is talking to A, but his responses go through the intruder, who is faking A
- ▶ The messages go back and forth between A and B, but they are going through the intruder, who is reading all of the messages
- ▶ This is called a man in the middle attack, and is a common hacker technique
- ▶ If B is A's bank a considerable amount of damage could occur

# Public Key Infrastructure

- ▶ How do we get around this problem?
- ▶ The basis of the solution is a certificate, which binds your identity to your public key
- ▶ There is one certificate for each of your public keys, typically a system will have approximately 100 certificates
- ▶ How do we know if a certificate is valid?
- ▶ The certificate is signed by a certificate authority (CA), a trusted third party
- ▶ They sign it with their private key, which can then be verified using their public key

# Public Key Infrastructure

- ▶ This requires an infrastructure for issuing certificates
- ▶ The initial idea was to have a small number of root CAs, which would then have a number of regional authorities underneath them
- ▶ This would form a hierarchy of CAs
- ▶ It would basically follow a structure similar to DNS
- ▶ In practice this hasn't happened, the structure is more chaotic
- ▶ Some companies discovered that there was money to be made, so they set up their own root CA
- ▶ Others thought that this should all be like open source

# Public Key Infrastructure

- ▶ Now we have a wide range of entities issuing certificates, some more trustworthy than others
- ▶ There are standard formats for certificates and software for creating them
- ▶ Most browsers now ship with a set of standard certificates, this expands as you interact with various services, these certificates are for the CAs, so you have their public keys
- ▶ Most users never see this, the idea is to make it largely automatic at the browser level
- ▶ Browsers provide options for viewing your certificates, this is central storage on your system that all browsers share

# Public Key Infrastructure

- ▶ The certificates are used to set up secure communications, why they are important to browsers and web servers
- ▶ When websites register their domain a certificate for them is usually issued, the registration must be renewed, usually yearly or the certificate will expire
- ▶ They are also used to sign software, particularly for mobile applications
- ▶ Example, IOS applications must obtain certificates from Apple
- ▶ What happens if a certificate is compromised? How do we deal with this?

# Public Key Infrastructure

- ▶ The original idea was that certificates would have relatively short lifetime, maybe a year at most
- ▶ When they expired they would need to be renewed
- ▶ Reasoning:
  - ▶ A compromised certificate would have a limited lifetime
  - ▶ We can make money by renewing certificates
- ▶ The commercial reason still exists, but there are a number of certificates with very long lifetimes, mainly the CAs themselves

# Public Key Infrastructure

- ▶ Certificates can also be revoked if they have been compromised
- ▶ This is a complicated process, it's not clear that it works very well
- ▶ The idea is that the issuing CA will produce lists of the certificates that have been revoked and distribute them to the sites that verify certificates
- ▶ With the initial hierarchical system this seemed reasonable, but with the way things have evolved its not clear how practical this now is

# Summary

- ▶ A quick overview of cryptography, basic terminology and techniques
- ▶ Examined some of the standard ciphers
- ▶ Examined the problem of key management

# CSCI 3130

# Network Security

MARK GREEN  
FACULTY OF SCIENCE  
ONTARIO TECH

# Introduction

- ▶ Examine some of the issues in network security
- ▶ Some of the tools that can be used to make networks more secure
- ▶ A large topic, just cover some of the main points

# Firewalls

- ▶ Our main tool in network security is the firewall
- ▶ A simple idea that becomes exceedingly complicated in implementation
- ▶ A firewall sits between the Internet and a local computer or a local area network
- ▶ The single point in or out of the network
- ▶ The basic idea behind a firewall is:
  - ▶ Block malicious network traffic from entering the local network
  - ▶ Prevent secrets from leaving the local network

# Firewalls

- ▶ While it sound easy, it can be quite difficult to implement
- ▶ All the information flowing in and out of the LAN must go through the firewall
- ▶ Thus, it must be very efficient, don't want it to be a network bottleneck
- ▶ This restricts the amount of work it can do on each packet
- ▶ The simplest form of firewall just does port blocking
- ▶ It examines incoming TCP and UDP packets to see which ports they are directed to
- ▶ If the port is blocked the packet is dropped

# Firewalls

- ▶ The is the simplest type of firewall to implement and manage, and is quite efficient
- ▶ Most home routers have this type of firewall, it is preconfigured to block most ports
- ▶ A web interface can be used to open particular ports
- ▶ Since the router is already doing NAT and port mapping adding firewall functionality is quite easy
- ▶ The user interface is quite simple and reasonably easy to understand
- ▶ This is a good tool and should be used on all systems

# Firewalls

- ▶ More sophisticated firewalls are based on a set of rules that examine packet contents
- ▶ In most cases its just header information they examine, but more sophisticated ones also examine the data
- ▶ They can roughly be divided into two groups:
  - ▶ Stateless
  - ▶ State based
- ▶ The state based firewalls track connections and have rules that can span multiple packets, the stateless one operate on one packet at a time

# Firewalls

- ▶ While state based firewalls can detect many types of malicious attacks, there are problems with this approach
- ▶ The rules tend to get complex, and the notation for describing them is not user friendly
- ▶ Thus, it becomes difficult to configure the firewall and determine exactly what it is doing
- ▶ This often leads to configuration mistakes by the system administrator
- ▶ If the rules are incorrect, malicious traffic can easily get through it, and it can be hard to fix

# DNS

- ▶ DNS was set up before network security was viewed as a big issue
- ▶ The original version of DNS had several major security holes
- ▶ First, anyone can deploy a DNS server, it is a distributed system with no central control
- ▶ A rogue DNS can have incorrect or malicious information
- ▶ When you ask for the IP address for a domain, it redirects you to a different domain
- ▶ This can be used to spoof websites, user thinks they are interacting with their bank, but are instead interacting with a malicious website

# DNS

- ▶ With the original version of DNS there was no way of detecting rogue DNS servers
- ▶ Second, is a problem called cache poisoning
- ▶ In this attack a DNS server claiming to be an authoritative server sends new cache records to a secondary server or resolver
- ▶ These records will have a long time to live and will produce incorrect results
- ▶ Occasionally the local resolver cache will become polluted and you need to reboot your system

# DNS Security

- ▶ There are several things that have been done to improve DNS security
- ▶ The first is DNSSEC, this is a standard that signs the records sent by the DNS server
- ▶ Either SHA-1 or SHA-2 is used as a message digest for the record, this is then encrypted with an RSA private key
- ▶ The public key is then made available from the DNS server
- ▶ Most DNS servers now use DNSSEC and most resolvers can handle it
- ▶ It took about 10 years for it to be widely available

# DNS Security

- ▶ The second problem is cache poisoning, where secondary servers get incorrect data
- ▶ There is now a secure protocol for transferring cache updates to the secondary servers
- ▶ This is now widely used
- ▶ With these two changes DNS appears to be largely secure

# WiFi

- ▶ WiFi can be a major security hole, since it is a broadcast based technology
- ▶ Anyone within range of an access point receives all of the packets handled by that access point
- ▶ This make it very easy for people to spy on WiFi connections, in addition WiFi can have a range of 100 meters or more, so the person spying may not be visible
- ▶ The early versions of WiFi had no security, which soon became a problem
- ▶ This is still the case with some free WiFi services

# WiFi

- ▶ The early attempts at WiFi security were not very successful
- ▶ The encryption techniques were too weak and easy to crack
- ▶ The current version works reasonably well, but still has some holes
- ▶ There are two approaches that can be used to connect to a WiFi access point
- ▶ In an enterprise based system you need to have an ID and password to access WiFi
- ▶ This is the case with the university system
- ▶ We already have user names and passwords, so this is easy to add

# WiFi

- ▶ The other approach is to use a password or a key
- ▶ This is the approach used in home WiFi systems
- ▶ In either case the access point will produce a unique session key for the connection
- ▶ Since each connection has a different key, users can't easily spy on each other
- ▶ Some public WiFi systems have no security, this is the case when there is no account or password
- ▶ You should be very careful using these systems

# WiFi

- ▶ WiFi is subject to man in the middle attacks, particularly in public areas
- ▶ All that's required is a laptop with two WiFi connections, the second one is a USB one
- ▶ One connection spoofs an access point, while the other one connects to the real access point
- ▶ When a system connects to the spoofed access point they are given a session key for that access point
- ▶ The hacker can then read everything the system is sending before forwarding it to the real access point

# IPsec

- ▶ IPsec is used to add security at the IP level
- ▶ This is a fairly complicated standard, so won't go into the details
- ▶ It is implemented on many systems, but is usually not installed or configured by default
- ▶ IPsec requires both kernel modifications and programs at the user level
- ▶ The kernel modifications are used to modify the IP packets
- ▶ The user level program are used for key negotiations

# IPsec

- ▶ IPsec runs in two modes
- ▶ In transport mode only the payload is encrypted
- ▶ An authentication header can be added that prevents the IP addresses from being changed
- ▶ This mode cannot be used with NAT
- ▶ In tunnelling mode the whole IP packet is encrypted and then enclosed in another IP packet
- ▶ This can be routed through NAT, since only the outer IP packet is changed

# VPN

- ▶ There is a wide range of technologies that fall under this name, they offer various levels of protection
- ▶ Before the Internet many large companies had private networks
- ▶ They would rent lines from telecommunications companies to connect their sites
- ▶ All the protocols and routing were handled completely within the company and there was no public access to the network, very secure
- ▶ Private networks are very expensive and when the Internet came along many companies wanted to transition to the Internet

# VPN

- ▶ Problem: the Internet is not secure, they wanted to same protection as their private networks
- ▶ Wanted a virtual private network, or VPN
- ▶ VPN works best within an enterprise, know all the systems that need to be connected
- ▶ Some of these systems could be remote workers, but you know who they are
- ▶ In this case the network has a known topology, you know all the systems that need to be connected
- ▶ Easy to identify the end to end connections

# VPN

- ▶ There are two main ways of doing an enterprise VPN
- ▶ If the same service provider handles all of the Internet traffic they can set up a VPN at the router level
- ▶ Example, the router can examine the source and destination IP addresses to see if they both belong to the VPN
- ▶ Only local traffic is allowed along the VPN
- ▶ Encryption can be added to this for extra security
- ▶ The other approach is at the software level using some form of tunnelling protocol, such as IPsec

# VPN

- ▶ Depending upon the tunnelling protocol the packets may or may not be encrypted
- ▶ There are a number of commercial services that also go under the VPN name, but they don't provide the end to end security that the enterprise ones do
- ▶ For commercial services they don't know which sites you want to connect to
- ▶ If a site isn't on their VPN, which most aren't, you won't have end to end security
- ▶ You will only have security to the VPN service servers

# VPN

- ▶ Why would you want to use a commercial VPN?
  - ▶ It secures public WiFi, if you are a frequent user of public WiFi systems this greatly increases your security
  - ▶ You can access websites that are normally blocked, as long as the VPN server can reach them, you can reach them
  - ▶ If you want to hide your identity when interacting with certain servers
- ▶ Note: a commercial VPN will not completely protect you when interaction with Amazon or your bank

# TLS

- ▶ IPsec works at the network layer adding to IP, in this case applications don't need to be aware of it, no extra programming is required
- ▶ TLS works at the transport layer, usually over top of TCP
- ▶ Application need to be aware that they are using TLS, but it provides more security than IPsec
- ▶ TLS evolved from SSL, one of the first protocols for security over TCP
- ▶ The initial versions of SSL were developed by Netscape to allow Internet commerce

# TLS

- ▶ There are problems with all the versions of SSL and it should not be used
- ▶ TLS was designed to be a replacement for SSL
- ▶ Version 1.0 and 1.1 of TLS also had security problems and are no longer recommended for use
- ▶ Version 1.2 was introduced in 2008 and is the most common version in use now
- ▶ Virtually all web servers and browsers support this version
- ▶ Over 99% of web sites use it for security

# TLS

- ▶ The most recent version of TLS is 1.3 and it appeared in 2018
- ▶ It has added security and is slowly replacing version 1.2
- ▶ Approximately 50% of web sites accept version 1.3
- ▶ TLS is a connection oriented protocol, the client must request a TLS connection
- ▶ TLS has an elaborate protocol for establishing a connection
- ▶ The client starts by sending the server a message specifying the version they want to use, the cipher suits that they support and a random number

# TLS

- ▶ The server responds with the TLS version it will use, one of the ciphers from the client's list and a random number
- ▶ It also sends a certificate that identifies the server
- ▶ The client can use the server's public key to verify that the certificate is valid
- ▶ In some cases the client will also send a certificate to the server, so the server can authenticate the client
- ▶ At this point the client and server will agree on a session key that will be used for the current session
- ▶ There are several techniques for this

# TLS

- ▶ Once a connection has been established the client and server communicate in a secure way
- ▶ This is the protocol that is used by https for secure web sessions
- ▶ There are a number of ways of attacking TLS
- ▶ The most common is protocol downgrade, where the client and server are forced to use a previous version of the protocol that has security holes
- ▶ I suspect it is also possible to use a man in the middle attack with the current version of TLS

# Honeypots

- ▶ This technique pre-dates computers, from the world of espionage and counter espionage
- ▶ A honeypot is a system designed to attract hackers, but will do no damage to the organization
- ▶ It looks like a real system with real resources that are worth stealing, but in fact has no useful information
- ▶ They usually run in a virtual machine so they are easy to monitor and control
- ▶ It doesn't matter if the VM is compromised, can always spin up a new VM

# Honeypots

- ▶ There are a number of reasons for wanting to use a honeypot
- ▶ One use is to trap hackers, if they spend enough time on the honeypot you can sometimes trace where they came from and identify the hacker
- ▶ They can be used for research to determine the technique that hackers are using to break into systems
- ▶ A sniffer is used to capture all the packets, which are then analyzed, including the packet contents, to determine what that hacker is doing
- ▶ The hacker can't detect the sniffer, so they don't know they are being watched

# Honeypots

- ▶ Honeypots can be used to distract hackers from the real systems
- ▶ They think they have found the real system and are getting lots of “useful” information, they don’t have time to attack the real systems
- ▶ One of the most interesting uses is misinformation
- ▶ The honeypot is full of information that is wrong, it is fake data
- ▶ A hacker trying to steal data thinks they have found the data they are looking for and stops hacking the organization
- ▶ For example fake credit card information, when this information appears on the Internet, can determine who stole it, hurts their reputation

# Honeypots

- ▶ Misinformation is quite often used by government and military organizations
- ▶ Hackers think they have found the plans for a secret new weapon
- ▶ Their country builds it, when they test it for the first time it explodes
- ▶ Even worse they could construct and deploy a weapon that the enemy could control on the battle field
- ▶ There are a number of countries that have fallen for this
- ▶ One of the main problems with honeypots is that they require a lot of work to administer

# Man in the Middle Attacks

- ▶ These are probably the hardest attacks to defend against, since they can happen far away from your organization
- ▶ Most of the security techniques can be cracked in this way, there are few that are immune
- ▶ This brings up the difference between protection and detection
- ▶ Example, quantum key distribution can't prevent the attack, but it can detect it
- ▶ Once the attack has been detected you can take defensive actions, track down where the leak is

# Man in the Middle Attacks

- ▶ Where can these attacks occur?
- ▶ We've already seen that WiFi is vulnerable, avoid public WiFi
- ▶ Ethernet can also be vulnerable if a hacker has access to the physical cables, they can then install a repeater in the cables that performs the attack
- ▶ Physical access can be performed via a boyfriend attack
- ▶ If all the links between routers are buried fibre, that won't be a source
- ▶ That leaves a hacked router as the source of the problem, which should be relatively easy to find

# Summary

- ▶ A high level overview of network security
- ▶ Described some of the tools that can be used to make a network secure
- ▶ Some of the techniques, such as honeypots, that can be used to defeat hackers
- ▶ This is a very large topic, have skipped all the details
- ▶ Develop an awareness of what could go wrong

# CSCI 3310

# Operating Systems

# Security

MARK GREEN  
FACULTY OF SCIENCE  
ONTARIO TECH

# Introduction

- ▶ A quick survey of operating systems security
- ▶ Look at some of the major topics
- ▶ Not go too deeply into the details
- ▶ Again we need to be careful, people trying to break the system will go to great lengths
- ▶ There are some interesting ways of getting around security

# OS Architecture

- ▶ Few operating systems are designed with security in mind
- ▶ There are some experimental systems, but none of the popular ones have given it a lot of thought
- ▶ One of the problems with existing systems is they are far too large with far too many features
- ▶ Their kernels are far too large, most of the code really doesn't need to be there
- ▶ The larger the kernel is the more opportunity for hacking, the more places that things can go wrong

# OS Architecture

- ▶ Parts of the system can interact in unintended ways to open up a security hole
- ▶ The large number of security patches that come out for all of the major operating systems are a indication of this
- ▶ A kernel that is 10x smaller will be more secure
- ▶ Example: networking doesn't need to be in the kernel, it can all be done in user space
- ▶ Only the device drivers need to be in the kernel
- ▶ Most of the original networking software was done in user space

# Resource Access

- ▶ One of the main OS functions is to ensure that users don't access things that they shouldn't
- ▶ This includes things like memory, files, hardware devices, etc.
- ▶ Memory has a lot of hardware protection, so that isn't a major concern, but other resources are
- ▶ For personal computers there is typically only one user, unless the computer has been hacked
- ▶ It may be shared by several people, but only one at a time
- ▶ This isn't the case with servers, where there may be multiple concurrent users and numerous services running

# Resource Access

- ▶ The key question is: what resources can a user (or a server) access and what can they do with them
- ▶ Each object in the system will have a set of operations that can be performed on it
- ▶ Example: a file can have read, write and execute operations
- ▶ A domain is a pair:
  - ▶ An object with a set of operations
  - ▶ A set of rights to use the individual operations
- ▶ Each file could have multiple domains, depending upon access rights for different users

# Resource Access

- ▶ This leads to a matrix where the columns of the matrix are the objects and the row are the domains
- ▶ When it comes to users they could be members of many domains, usually given by a user ID and a group ID
- ▶ Clearly this matrix will be very large, one column for each file
- ▶ It could also be quite sparse
- ▶ While this is an interesting way to think about resource access, it doesn't work in practice
- ▶ Where would we store the matrix?

# Resource Access

- ▶ One way to organize this is with an Access Control List (ACL)
- ▶ Each object has a list of the access rights for different groups of users
- ▶ A good example of this is the access rights on Linux files
- ▶ Each Inode has the access list for the file
- ▶ The alternative to this is capabilities
- ▶ In this case each user (or process) has a list of its domains, the objects and the access rights that it has to those objects
- ▶ This is more difficult to implement, but can give finer grained control

# Security Models

- ▶ Over the years many security models have been developed
- ▶ Mathematically they are quite nice, and you can prove interesting things
- ▶ There has also been attempts to prove that particular operating systems are secure
- ▶ While proofs can be developed, they turn out to be not too helpful
- ▶ It has been shown that for these security models, there are always ways that they can leak
- ▶ One way is by monitoring process behavior

# Authentication

- ▶ We've already discussed some of this under network security
- ▶ User authentication is typically done through some kind of login procedure
- ▶ The typical way is with a user name and passwords
- ▶ Problem: many people choose weak passwords, so this can be compromised
- ▶ One time passwords are passwords that can only be used once, each time the user logs in the password changes
- ▶ These passwords are usually generated algorithmically

# Authentication

- ▶ In most cases users can't deal with one time passwords, unless they are based on a simple function
- ▶ In most cases an external device is used to generate the password for the user
- ▶ In addition to passwords there can be challenges
- ▶ The system has a collection of questions that it can ask the user along with the answers
- ▶ One of these questions is chosen at random and the user needs to give the correct answer

# Authentication

- ▶ Biometrics are becoming a common authentication technique
- ▶ In this technique some biological property of the user is measured and compared to a stored results
- ▶ Finger prints and face recognition are the two most common techniques, both cases require special equipment
- ▶ There are techniques for faking finger prints, so this is not completely secure
- ▶ Face recognition can give false negatives, so a back up technique is usually required
- ▶ There are also racial issues with face recognition

# OS Attacks

- ▶ There are quite a few ways of attacking an operating system
- ▶ Attacking the kernel directly is quite difficult, but not impossible
- ▶ Start by assuming the attacker wants root access and will do this through a SETUID program
- ▶ The classic way of doing this is through buffer overflow
- ▶ In this type of attack the program allocates a buffer to contain input from the user and then uses a function like `gets(buffer)` to read the input
- ▶ Unfortunately, `gets` is passed a buffer, but not the length of the buffer and reads a line of input into the buffer

# OS Attacks

- ▶ If the input line is longer than the buffer, memory outside of the buffer will be overwritten
- ▶ The question is: what is this memory?
- ▶ The buffer is usually an array on the stack
- ▶ Remember that stacks grow down through memory, and the location of the buffer will be the lowest address in the allocated block on the stack
- ▶ When the buffer overflows it overwrites locations above it on the stack, one of these locations just happens to be the return address for the procedure

# OS Attacks

- ▶ The attacker places machine code in the input, and overwrites the return address to jump to this code
- ▶ This code will typically call an exec procedure that will start a new shell, which will have root permissions, this is called a shellcode attack
- ▶ Alternatively the attacker can place code in the heap by taking advantage of procedures like strcpy, this is called heap spraying
- ▶ Most modern operating systems prevent the execution of code in the stack and heap to defeat this type of attack

# OS Attacks

- ▶ If the attacker can't inject code into the program, he is just going to have to use existing code
- ▶ There are many variations of this type of attack
- ▶ One of the procedure is libc is called system
- ▶ The procedure has a single parameter, a text string and starts a new shell passing this string to the shell for execution
- ▶ The trick is to get the program to call system with a string that you enter
- ▶ Again this can be done with buffer overflow

# OS Attacks

- ▶ The attacker's input places the command to be executed on the stack and overwrites the return address to call system
- ▶ A more sophisticated version of this involves shared libraries
- ▶ On Linux there will be a table in memory that points to each of the procedures in the shared library
- ▶ This can be used to find the location of the system procedure in libc, or change the procedures that are called
- ▶ This can be used to disable procedures that check for appropriate permissions before performing an operation

# OS Attacks

- ▶ Address Space Layout Randomization (ASLR) can be used to reduce this type of attack
- ▶ Each time a program is loaded into memory the location of procedures is randomly changed
- ▶ This makes it hard for the attacker to find the location of system() or a similar procedure
- ▶ The main problem with this is its easy to put libraries at random locations, particularly with dynamic loading, but the procedures in the libraries will still be at the same relative location
- ▶ The process just isn't random enough

# OS Attacks

- ▶ Injection attacks occur when input entered by the user can end up being executed
- ▶ The main place that this occurs is in the system() procedure, but there are other places
- ▶ You should never use any user input as part of something that is executed
- ▶ If an attacker is just after information there is another set of techniques that can be used
- ▶ One is to enter an input value that is out of range in hopes that it's used as an array index

# OS Attacks

- ▶ You should always check that user input is within range
- ▶ Some operating systems have been a bit careless with memory allocation to processes
- ▶ When a new block of memory is requested, the previous contents have not been erased
- ▶ The process can then look through the memory for the other process in the hope of finding something interesting
- ▶ Memory should always be cleared to zero before it is given to another process, this is an operating systems bug

# OS Attacks

- ▶ Linux has a potential memory management bug that may have been exploited
- ▶ To make switching between user and supervisor mode more efficient, Linux has included the kernel memory space in the page table for every process
- ▶ In this way the page tables don't need to be switched on a system call
- ▶ These pages should be marked as not accessible to the user program, but there is some thought that there may be a way around this

# Malware

- ▶ A general term that includes virus, worm and rootkit, the major difference is how aggressive they are
- ▶ Virus is the least aggressive of the group and needs another program to live on
- ▶ A virus doesn't appear on its own
- ▶ The virus code is added to the code of another program and is executed whenever that program is executed
- ▶ One approach is to add the virus to the end of the executable, which is then modified to jump to the start of the virus when the program is run

# Malware

- ▶ The regular program is then run after the virus, the user typically doesn't know that the virus has run
- ▶ There are numerous things a virus can do
- ▶ It can insert a bot into the system that can be controlled by the attacker, this is typically used for a DDoS attack on another system
- ▶ It can search for bank accounts and other personal information
- ▶ one of the most common attacks now is ransomware
- ▶ The virus encrypts the disk and then demands money for the key to decrypt it

# Malware

- ▶ Ransomware is easy to deal with, if you plan ahead
- ▶ The key is to keep programs and data on separate file systems
- ▶ The data file systems are backed up regularly, even if the virus has placed itself on that file system, as long as programs aren't executed from that file system it doesn't matter
- ▶ To get rid of the ransomware you need to reinstall all the programs and then restore the data file systems
- ▶ This only works if you have planned ahead, security cannot be an after thought

# Malware

- ▶ Worms are more aggressive, they can do the same things that viruses do, but they can also replicate themselves
- ▶ Worms look for network connections and they try to propagate themselves to other computers on the network
- ▶ They will also try to get on removable storage, with the hope of being moved to another computer
- ▶ Rootkits are by far the worst, while viruses and worms tend to be a single file, rootkits are multiple files and attempt to hide themselves so they can't be found or removed
- ▶ This makes them very difficult to deal with

# Malware

- ▶ One type of rootkit modifies the boot record so it starts instead of the operating system
- ▶ It then starts a virtual machine and runs the real OS inside the virtual machine
- ▶ It can then monitor all traffic to I/O devices, including the network
- ▶ The only way to detect this is by noticing a difference in performance
- ▶ One of the most famous rootkits is the Sony rootkit from 2005, this was added to the music CDs sold by Sony

# Malware

- ▶ Sony was concerned about people trying to steal their music
- ▶ They added an autorun.inf file to the start of their music CDs
- ▶ A regular CD player would ignore this file, but on a Windows computer it would run automatically when the CD was inserted
- ▶ The CD contained a 12Mbyte rootkit, a CD normally has around 640Mbytes of data, so this wasn't noticed
- ▶ This rootkit replaced the CD driver with one that only allowed the CD to be played by Sony's media player
- ▶ It also looked for about 200 media copy programs and deleted them

# Malware

- ▶ All its files started with \$sys\$ and it modified the OS so those files would never be displayed, essentially hiding all its files
- ▶ It also made changes to the Windows registry to hide it further
- ▶ It was discovered by a security expert not long after it was deployed
- ▶ Sony initially denied its existence, but was forced to admit it
- ▶ The first fix Sony released only removed part of the rootkit
- ▶ It was then forced to issue a second fix, that not only removed the rootkit, but many other system files, it also left a backdoor into the system
- ▶ It left a large number of computers unusable

# Malware

- ▶ There were court cases, Sony was sued, etc.
- ▶ This gave rootkits a bad name, some hackers won't use them as a result
- ▶ How does malware get on your computer?
- ▶ You put it there, you are tricked into installing it on your system
- ▶ Malware quite often rides on pirated software, or “free” software that you download from the Internet
- ▶ You can download it from websites by mistake, thinking that you need it to look at some content

# Malware

- ▶ One of the main sources used to be browser extensions
- ▶ They would claim to do something useful, but often contained a virus or worm
- ▶ Most modern browsers either don't allow extensions, or will not install them automatically
- ▶ Anti-virus software can help with this and has dramatically reduced the number of virus and worm attacks
- ▶ Most operating systems are now more secure and make it more difficult to infect a system

# SolarWinds Exploit

- ▶ One of the most famous recent attacks, did a lot of damage
- ▶ An example of a supply chain attack, this is a sophisticated attack, since it is now harder to use viruses and worms
- ▶ In this approach the attacker looks for the weakest point in the software development supply chain and exploits it
- ▶ The developer assembles parts of an application from several sources, it could buy a library from another developer
- ▶ May not know the contents of all the software, particularly a library distributed as binary

# SolarWinds Exploit

- ▶ SolarWinds produces a product that is used for network monitoring
- ▶ It is used by a large number of government and industry organizations, including those involved in security
- ▶ The attackers inserted a remote access tool into updates to the SolarWinds software that gave them a back door into the systems
- ▶ About a week or so after compromise additional software was installed on the system that allowed the attackers to spy on it
- ▶ Since the SolarWinds software was trusted it had higher privileges than all of the malware inherited

# SolarWinds Exploit

- ▶ How did this happen?
- ▶ All the details are not completely known, but the hackers got access to Microsoft Office 365 accounts for SolarWinds
- ▶ This probably came from a phishing attack where someone accidentally gave out a password
- ▶ From there they infected the SolarWinds build system
- ▶ Once this had been done, they could include their software in the SolarWinds software that was distributed
- ▶ This was usually in the form of updates to existing customers

# SolarWinds Exploit

- ▶ This was a very well planned attack and took well over a year to execute
- ▶ They were very patient and let things evolve slowly, waiting a month or two between steps in the attack
- ▶ Systems that were compromised didn't immediate react, waited several weeks before they joined the network
- ▶ By the time it was detected it had infected well over 10,000 systems
- ▶ The attackers only used a small percentage of the systems they compromised, they were after high value targets

# Summary

- ▶ Examined some of the issues in operating systems security
- ▶ Outlined some of the types of attacks, there are many others
- ▶ Many of the simpler attacks are now well defended
- ▶ Attacks are getting more sophisticated, as illustrated by the SolarWinds exploit