September 26, 2001

- Introductions
  - **CS 395 Introduction to computer systems**
  - **WF 10:30-12**
  - **Peter Dinda, office hours: M2-4 or appt**
  - **Dong Lu, office hours: F3-5 or appt**
- Description
  - **Three goals**
    - ***Learn about hierarchy of abstractions that make up a computer system so that you can flesh them in later***
      - **This course is "what everyone should know"**
    - ***Demystify the machine and the tools***
      - **Go deep**
    - ***Bring you up to speed in doing systems programming on Unix***
  - **Elements**
    - ***Soup to Nuts – Physics to Distributed Systems***
    - ***Strong emphasis on***
      - **Data representations (integer and floating point)**
      - **The machine model (instruction set architecture)**
        - And understanding compiler-generated code for it
      - **Memory systems and memory management**
      - **Linking**
      - **Exceptions**
      - **Unix systems programming**
    - ***Less emphasis on***
      - **Physics to logic**
      - **Networking programming**
      - **Concurrency**
      - **Distributed systems**
    - ***Elements***
      - **C programming on Linux**
      - **De-compiling code and understanding assembly (but it's not an assembly language programming course)**
      - **Understanding a bit of how the machine works (the P3), but it's not a computer org class**
    - ***Learn by doing***
      - **Labs that will make you hack**
      - **Homeworks that will also make you hack**
  - **Where it fits into curriculum**
    - ***Intended as a sophomore class***
    - ***May become a regular class***

May become a required class
After 211 or 311, before OS, Networking, Compilers, etc.
A lot is riding on what you think of this

IT'S A BETA TEST

It's been taught several times at CMU

But on a semester system
This is the first time in a quarter system
We're going to cover networking programming, concurrency, and performance optimization/measurement only at a high level and leave the rest for OS, Networking, Compilers, and later courses

New textbook due out in 2002 (you'll get a free copy)

It's a quite unique textbook.
Although we won't go through all of it, you might find yourself reading the parts we skip.

Although we may screw things up, your grade will not be effected by the beta nature of the course
Feedback is helpful to us and to the authors
We're going to evaluate the course for use here permanently and we're going to evaluate the book to help with the next edition.

Mechanics

Course web site:

http://www.cs.nwu.edu/~pdinda/ics-f01

Communication

Email list will be started
IRC server for chat
Newgroup
All info will be on the web site for this

Textbook and C book

Have Dong hand out Textbooks at end of class – if can't make it, get copies from Olga
Textbook is also available from the web page
Limited copies – if you're not in yet, please don't take a paper copy

Machine access

You will need TLAB(125) windows and linux accounts and a cardkey for 125. See Pam for the accounts and Olga for the cardkey
TLAB-10 through 15 will be running linux dedicated, so you can do remote login

It's OK to work on your own machines or with cygwin or whatever, but everything will be graded in the TLAB environment. It's your responsibility to be sure that it runs there.

Grading

10 % Homeworks
50 % Labs
20 % midterm
20 % final
Note that Homeworks may be more in-depth than the amount of grade assigned to them. Please consider this when you are allocating time. I wanted completeness
Note that the book has lots of problems with the answers in the back.

Syllabus, TLAB, Physics, Datalab handouts

Syllabus may change a bit, especially toward the end

Survey next time to see if we need to do an "introduction to programming on unix"

But the web page has lots of material on this already

The Great Realities according to your textbook

Information is bits in context

A program must interpret the bits for it to make sense
Understanding these interpretations is important to writing good programs and debugging.
Ints are not Integers and floats are not reals

- **Is $x^2 \geq 0$?**
  - Float's: Yes!
  - Int's:
    - » **65535 * 65535 --> -131071 (On most machines)**
    - » **65535L * 65535 --> 4292836225 (On Alpha)**
- **Is (x + y) + z = x + (y + z)?**
  - Unsigned & Signed Int's: Yes!
  - Float's:
    - » **(1e10 + -1e10) + 3.14 --> 3.14**
    - » **1e10 + (-1e10 + 3.14) --> 0.0**

It's important to understand these unusual properties

Programs are translated by other programs into different forms

Fx program – parallel fortran
Compiled to regular fortran (77)

F77 translated to C
Gcc C preprocessor(cpp) handles macros
Gcc compilers (cc1) converts to assembly language (.s)
Assembler (as) converts to object code (.o)
Linker (ld) combines many object codes and libraries to create an executable

It pays to understand how compilation systems work

Linking problems
Security holes
Performance optimization
Compiler bugs
It pays to be able to look at the assembly output and understand what the compiler has done.

Let's you try different approaches for performance critical code

Processors read and interpret instructions stored in memory

Once you understand what your program is doing at the machine level, you can understand what parts of the system architecture and microarchitecture it is using and when and you can use this to spot bottlenecks and understand it's performance
Your book gives an example of how "hello world" exercises the system architecture of a typical Pentium iii machine.

Caches matter

Caches match big and slow kinds of memory with small and fast kinds of memory
You don't have to know about caches to write a correct program
You probably need to think about caches when you write a FAST correct program

Storage forms a hierarchy

On-chip registers to local disk
Think of each layer as a cache on the layer below
Your book talks about the network (and distributed systems) as the lowest layer on the hierarchy. This is debatable.

The Operating System Manages the hardware

Protection and abstraction

Processes, Threads, Virtual memory, I/O abstraction

The kernel, interrupts, and System calls

Systems communicate with each other using Networks

TCP/IP is a virtual network

Berkeley Sockets as the standard way of programming network applications

Another View: The Abstraction Hierarchy

As computer scientists, we like to raise the level of abstraction – Ideally, we create interfaces with simple syntax, semantics, and timing on top of complex things

For the most part, we can just happily use our highest level abstractions when we program

Sometimes, though, we need to strip away an abstraction to understand what is happening beneath

Debugging

Performance

Security (buffer overflow exploits)

Research to develop better abstractions

Abstractions are sort of like layers on an onion, but not quite. Sometimes and abstraction can depend on several abstractions below it. And sometimes on abstractions more than one layer below it.

Show hardware graph

Show software graph

Indicate where we will be going

Physics up to ISA and micro and system in one day

Then a week on data representations – making using of logic

Then a couple of weeks on the ISA –assembler and the compiler

Then a couple of weeks on memory

A week on exceptions

A week on aspect of the unix virtual machine

Two+ weeks on other elements

Concurrency in programs

Network programming

…

What does this mean for a web server

Limits

Let's say it's written in C

Single threaded, no concurrency, one request at a time

And it uses sockets

Compiler toolchain path

Gcc httpd.c –o httpd

As before, except now we start in C

Dynamically link with sockets and C shared libraries

Execution

Type "httpd" into shell

Shell calls fork, which triggers an exception

OS figures out that the exception means "fork

OS creates new process with its own virtual environment – it's a clone of the shell process

process calls "exec", which triggers an exception, which the OS catches. OS cleans up the virtual enviroment, loads the first page of httpd

OS figures out that you want to run "exec"

Loads image (really first page) into memory and transfers control to its entry point

Entry point function loads shared libraries. They are probably already in memory, so the OS just maps them into httpd's address space

Entry point calls main (stack discipline)

Main runs of the end of the page, triggering an exception. OS sees it's a missing page and loads it from the httpd file

…

httpd creates, binds, and listens and accepts on a socket. Each call is an exception that the OS has to figure out.

Now the process "blocks" because there is no client. By blocks, we mean the OS does not run it.

A network packet comes in and causes an exception

The OS looks at it, passes it to network stack.

Network stack notices it's the beginning of a TCP three-way handshake and send back appropriate

response (OS+device drivers hide the details of using the NIC)
Third packet arrives, network stack establishes TCP connection, passes it to OS
OS notices connection matches httpd's outstanding accept request, unblocks httpd
Eventually, httpd runs again, accept finishes, now it has a file descriptor for the open connection
At any point, extra packets arrive, get handled by network stack and added to a buffer of received data for the connection
Httpd reads on fd, causes exception, OS notices it has data available, passes data back to httpd, copying it up to app
Httpd returns from read, processes request, sends write, causes exception, etc.
Notice that data is pushed across memory bus, i/o bus, and NIC.

Make sure to get handouts and book

September 28

Survey handout

How to represent a bit

A bit as a range of voltage levels
"1" threshold
"0" threshold

Transistor device

MOSFET – CMOS technology
Show symbol, and complementary device

Transistor down

Metal, Oxide, Semiconductor -> Conductor, insulator, semiconductor
Semiconductor

Silicon ~ Sand
Grow giant single crystals of pure silicon and cut them up to form the base on which to work
Resistive behavior easily changed by addition of impurities "doping"

Also changes its quantum behavior
Electric fields affect resistence

Interesting stuff happens when we put differently doped materials, and metal and oxide together

This is how transistors are made
Electric fields can affect resistence

MOSFETs are easy transistors to think about

Photolithography

Transistors (and chips) are built up by layers, like a stack of pancakes.

Think of each layer as a photograph

Each layer generally "deposits" a single kind of material

The negative (or mask) determines where it is deposited

Picture of how a MOSFET is built up in layers is in your handout

Small – about the size of a fingernail

But 10s to hundreds on each slab

Then they get cut apart, each is tested, if it passes, it gets mounted in a carrier and sold

One reason why they are small is defects

The other reason is that smaller=faster

Speed of light – one ns is about a foot

100 GHz -> 3 mm

Plus, electricity moves considerably slower than light

Currently, photolithography can put about 100 million transistors on a single chip

P4 ~ 42 million transistors

Regular structures can be much denser

Moore's Law

Resolution of the mask keeps cranking up

Doubling of the number of transistors ever year

It's been happening since the '60s

First microprocessor 4004 had 2300 transistors

Likely to continue for the next 10 years.

Transistor

Transfer characteristic

Nonlinear operation "off, linear, on"

Think of it as an electrically controlled switch for our purposes

Regular -> push the switch closed

Complementary -> pull the switch closed

Now we have bits = voltage, and we have electrically controlled switches

Combinational Logic

Truth tables

~A, A&B, A|B, ~(A&B), A xor B A nxor B

not, and, or (either or both), not-and, xor different, nxor – same

Implementing these using transistors

- **Invertor**
- **Nand**

More complex logic can be built from these primitives

- ***All you really need is nand***
- ***A bit adder***
  - **Xor as almost an add**
  - **But two bits in need at least two bits out, right?**
  - **What about carry?**
  - **Carry = a and b**
  - **This is a half-adder**
  - **A full adder takes three bits in (x,y,carryin) and produces two bits out (sum, carry)**
    - Half-add a and b giving s1 + c1
    - Half-add s1 and cin giving sum and c2
    - Carry is c1 or c2
  - **Then we can arrange full-adders to make adders for arbitrary length data**
- ***Logic can get quite complex. We sometimes talk about a "cone of logic" that drives some output***
- ***Note that there is no memory***

Memory

- ***Asynchronous***
  - **DRAM cell**
    - Leakage + refresh
  - **SRAM cell**
- ***Synchronous***

These combinational circuits work asynchronously.

- ***What if output 1 arrives before output 2?***

The notion of a clock

- ***All outputs must be ready at certain intervals***
- ***The clock edge places them into a synchronous memory***
- ***Latch, flip flop (won't talk about these)***
- ***REGISTER (usually not visible to the programmer) holds the STATE of the machine***

Putting it together

- ***Inputs + state => combinational logic cone => outputs + next state***
- ***Clock edge releases the hounds, inputs arrive***
- ***Slowest path through the logic cone determines the speed of the chip***

Microprocessor

A chip like this designed to interpret some of the inputs as a simple language
We'll talk more about microarch in a week or so

Bytes
- **8 bit quantity**
- **Darn near universal**
- **Sufficient to capture alphabet of western languages**

Machine Word Size
- **Integer number of bytes**
- **Ordinary size of integer data on the machine**
  - *What it's good at working on*
  - *32 bit word size on IA32*
  - *64 bit on itanium, alpha, sparc64*
- **Binary, Hexadecimal representation**
  - *0000 0001 0011 0100 1000 1001 1111 0011*
    - **0x013489f3**
  - *1101 1110 1010 1101 1011 1110 1110 1111*
    - **0xdeadbeef**
- **Can have multiples and fractions of words**
  - *Always integer number of bytes*

C data types and how they map into bytes

October 3

Mechanics
- **Info on the web page**
- **Reading: 2.4-2.5**
- **Homework out at end of class**
- **Course newsgroups cs.cs395-ics.discussion, cs.cs395-ics.announce**
- **IRC Server**
  - *Dualsword, password is CSAPP*

Who would like an intro to tools evening session?
- **Maybe Thursday at 6pm in classroom**

Finish up logic
- **Bit-wise Logic is a "Boolean Algebra"**
- **Integers are a ring**
- **Boolean algebras and rings have similar, but not identical properties – the book will tell you more**
- **We've talked about Not, And, Or, nand, not, nor, xor, nxor**
  - *Conversions*
    - **DeMorgan's Laws – convert between ands and ors**

- Xor from or, not and and
- *Ultimately, you only need a nand*

Logic on a machine operates in parallel across the bits of a word

Logic in C

- *Int a, b; // say ints are words are 32 bits*
- *When you say A&B, you get a 32 parallel ands*
- *Generalize to bit vectors, both < 32 bits and >32 bits*
- *Bit vectors = sets*
  - Or = union
  - And = intersection
  - Not = completment
  - Xor = symmetric difference – what isn't in both sets
- *& | ~ are different from && || and !*
  - Treat the whole word as a giant bit
  - All zero = false
  - Any one = true
  - latter return 0 or 1
- *Shift operators << >>*
  - Arithmetic versus logical
- *Xor swaps*
  - X^=y;
  - Y^=x'

X^=y; Words as the unit of operation of machine

Determine size of a c type: sizeof(type)

Usually an int is the size of the word

Memory as an array of bytes ultimately

But also an array of machine words

Addresses are byte addresses

Endianness

- *Little endian – LSB has lowest address*
- *Big endian – MSB has lowest address*
- *Intel is little-endian*

Showbytes code in your text to print a region of memory in hex

All C data types end up being a sequence of bytes in memory

- *Interpretation of those bytes is up to the processor and to the language, compiler, and program*
- *Int short char float double pointers*
- *Pointers*
  - Machine + OS contrive to create virtual address space for programs

0..2^n-1-1 bytes

compiler makes pointers n bits (n usually multiple of 8)

ALL POINTERS ARE THE SAME

Often, sizeof(int)=sizeof(char*), but not always, common mistake

C strings:

Char = byte

Sequence of bytes ending with a null (0) byte

Programs in memory

Sequence of bytes that the processor can interpret as a sequence of instructons

Math and logic operations

Memory operations (combined with ML in intel)

Branches

Intel encoding is variable length instructions

RISC machines usually have fixed length instructions

Exceptions

Integers

Lowercase -> bits, uppercase->quantity

Let's use 3 bit examples to make things a little easier

Unsigned versus signed

Unsigned B = -b_n-1*2^n-1 + sum(0..n-2,b_I*2^I)

Unsigned goes from –2^n-1 to 2^n-1-1 => -4 to 3

Signed goes from 0..2^n-1 => 0..7

Signed A = sum(0..n-1, a_I*2^I)

MSB indicates sign1=>negative

But notice wacky representation

0=000, -1=111 –2=110 –3=101 –4=100

What is the complement?

~x+1

This signed number convention is called 2's complement arithmetic

All operations are identical, from logic pov, to unsigned math

No extra logic

Only interpretation

Other possibility

One's complement

Complement = ~x

Has two zeros

Sign+magnitude

How big are the ints/shorts/etc on my machine…

Compile-time constants given to you by the compiler

K&R – look for limits.h

How do I get specific size items

Make #ifdef typedefs that adjust according to the architecture

On many platforms, use #include <stdint.h> and then int32_t, etc

Casting

Explicit Casts

From smaller unsigned to larger unsigned

Zero-fill extra bytes

Same number

From larger to smaller unsigned

Like zeroing out upper bytes

Potentially different number

From smaller to larger signed

SIGN EXTEND

Same number(!)

From larger to smaller signed

Like zeroing out upper bytes

Likely to be a different number!

Even sign may change!

From unsigned to signed

BIT REPRESENTATION STAYS THE SAME

Postive numbers and zero OK

Small Negative number becomes large positive number

Large negative number becomes small positive number

IMPLICIT CASTS

Signed arith unsigned => cast to unsigned!

Including for comparisons <, > ,etc.

-1 > 0 => no

-1 > 0U => yes!

Call to function with signed or unsigned with args that are opposite => implicit cast

Importants of gcc –Wall

Casting order

Short x = -1;

Unsigned int x= (unsigned)(int)x;

Unsigned int x = (unsigned)(unsigned short)x;

When to use unsigned

Don't use it just because you think the number is going to be positive

Loop index from I=0, I<n;I++

Someone will change the loop body later on

Modular math

Extended precision (arbitrary length fixed point math codes)

When you really really need an extra bit of precision

Integer Math

Integer math on computers, like logic, has a theoretical foundation

The theory of groups: modular arithmetic->Abelian group

Your textbook has more to say on this

Again, 3 bit examples

Unsigned addition

Draw it just like grade school math

U +v => (u+v) modulo 2^n

U+v is output if u+v < 2^n

U+v –2^n if >=

Real sum => range 0 to 2^n+1 => 0 to 2^n

The carry out is thrown away

It's thrown into a condition bit, which we'll talk about later, but from the C POV, it's thrown away

U+v > 2^n is known as integer overflow

How do we check for overflow in C?

S=u+v; is s < u ?

Or, equivalently, is s<v ?

Signed addition with 2's complement numbers

Exactly the same operation as with unsigned numbers, but the interpretation of the inputs and the results are now different

Overflows are now more complicated

S= u+v

Sum must be the range –2^w-1 to 2^w-1-1 else overflow

If >2^w-1-1 then wrapps aroundto –2^w-1

If <-2^w-1 then wraps around to +2^w-1-1

How to check for overflow

U,v<0 and s>0 => overflow

U,v>=0 and s<0 => overflow

Subtraction

Addition with 2's complement: u-v = u+ ~v+1

October 5

Mechanics

Should have been receiving mail messages from me (ICS: …)

Reading 3, 3.1-3.5, 5.7

IRC moved to grayling

Lab session… next Tuesday, 6-7:30.

Finish up integer math

Comparisons

u>v ?

one approach: u-v and check if positive

all the overflow cases need to be sanity checked

Split into three tests

U<0, v>=0 ? => false

u>=0, v<0 ? => true

else (u-v)>0 (will not overflow)

At assembly level can use the carry bit

Multiplication

Unsigned

Need 2n bits as output

Range is 0 to (2^n-1)^2

Signed

Range is -2^(n-1)*(2^(n-1)-1) to(-2^(n-1))^2

Unsigned mult in C

Top n bits of product are thrown away!

Really u*v modulo 2^n

Signed mult in C

Same deal! Same operation!

Note that sign can once again flip!

Think of this as cast to unsigned, do mult, then cast back to signed

When we do multiplications, we are assuming that the values will actually fit within the output

Ie, for 32 bit machines, that the input values really only use 16 bits

Casting games

Say you want to multiply two shorts and assign to an int.

If you multiply first and then cast, you'll get only 16 bits

So cast first, then multiply

At the machine level, the whole product is usually generated, often put into two registers

Thus, in assembly code, it is easy to deal with it

Arbitrary precision math packages

Using shifts to do unsigned multiplies by powers of 2

X<<a => x*2^a

Compilers will make this optimization for you in most cases

If you use shifts on signed numbers, results can be surprising

3 bit notion: 011 << 1 => 110

3 turns into –2

Using shifts to do unsigned divides by powers of 2

X>>a => floor(x/2^a)

Also a compiler optimization

Note that floor(x/2^a) is wrong direction of round for negative x

Instead want ceil(x/2^a)

Compute as floor((x+2^a-1)/2^a)

X+(1<<a)-1)>>k

Division

X / y = floor(x/y) – dividend if x, y > 0

X%y = remainder if x,y >0

How this is done is beyond topic of the course.

Floating Point Numbers

IEEE 754 standard, mid ‘80s

Used to many different standards

There still are, but almost all processors do IEEE by default

Developed by numerical analysts, hard to make really fast

Has kind of stymied moving research results in number representations and computer math into new processors

Log-based math

Interval math

…

Basic idea

Fractional binary numbers

Shift so that everything is 1.x and keep track of the shift amount

What does this mean…

You can only represent numbers that look like (int)x/2^n – ie, a subset of the rational numbers

Repeating bit patterns

1/3

1/10

Representation

Sign bit (1 = negative) s

Exponent E (exp

)Significand (mantissa) M (mant)

Normal range is 1.0 to less than 2.0

Float = sign bit 8 bit exp, 23 frac bits = 32

"single precision"

Double = sign, 11 exp, 52 frac = 64

"double precision"

Normalized Number Representational

Exp != 0 or 111111111

exp is biased…

Actual exponent E = exp – bias

Single => bias is 127, exp=1..254, E=-126 to 127

Double => bias = 1023, exp=1..2046, E=-1022 to 1023

Bias is generally 2^(m-1)-1 where m is number of exp bits

Mantissa has implied leading 1

Only the bits after the binary point are stored (free bit!)

Min=1.0 (0000)

Max=2.0-epsilon (1111111)

Example

399.0

399 = 0x18f hex

0001 1000 1111 binary

1.10001111 x 2^8

float.

M=1.10001111, mant = 10001111 (+23-8 bits of zero)

E=8, exp=E+ bias = 8+127 = 135 = 10000111

Sign = 0

Denormalized Number representation

Exp=0000000

E = -bias + 1 (-127 +1 = -126)

Signicand assumed to be 0.xxxxxxxx (x=bits in word)

Very very small numbers – closer to zero than the closest normalized numbers

Precision decreases as get smaller

Gradual underflow

If significand is all 0 => "true zero" (note +/-)

Special Representations

Exp=1111111111

Frac=0000000 => INF

+/- infinity

if an operation overflows, the number becomes infinite
1.0/0.0 => +inf
1.0/-0.0 => -inf

frac!=00000 => NAN
sqrt(-1) = NAN
inf-inf = NAN
…

Number line

NAN separate
-inf … - normalized… -denorm –0 +0 +denorm +norms +inf

Special properties of the encoding

FP Zero == Integer 0 (all bits zero)
Unsigned compare almost works with fp numbers

Operations

Rounding

FPU has much higher internal precision than the number representaitions support
FPU first computed "exact" result, then reduces it to the desired precision.

Overflow if exp too large
Rounding to make fit into fractional part

Rounding

Zero => lob off the extra bits in the frac
Round down (towards –inf)
Result is no greater than actual result
Round up (towards +inf)
Result is no smaller than actual result
Round to nearest even (default)
Tiebreaker -> round so that least sig digit is even
Statistically unbiased
Generally the default rounding mode is all you have unless you dive down to assembly

Multiplication is pretty easy

S =s1 xor s2
M = m1 * m2
E = e1*e2
Normaize

Shift right so that M < 2, incrementing E each time
E out of range => overflow
Round M to fit precision

Or give denormed result

BUT

NOT ASSOCIATIVE DUE TO OVERFLOW, ROUNDING
NOT DISTRIBUTIVE
A(B+C) != A*B +A*C*
NOT MONTONIC IN PRESENCE OF INF AND NAN

Addition

Align mantissa
Shift mantissa with smaller exp right until same exp
Do sign mag add of mantissas giving output mantissa, sign. Exponent is exp of larger input
Normalize and round like in multiplication
BUT

NOT ASSOCIATIVE DUE TO OVERFLOW + ROUNDING
CAN GET INFINITIES + NANS
INF AND NANS DON'T HAVE ADDITIVE INVERSES
NOT MONOTONIC IN PRESENSCE OF INF AND NAN

FP in C

Float, double

Casts between ints, floats and foubles CHANGES BITS

FP type to int type => get truncated fractional part. Meaningless if out of range, but usually get saturating math
Int type->fp type => perfect if enough bits in matissa. Otherwise will round according to rounding mode.

Questions

Int x; float f; double d;

X==(int)(float)x ? NO
X==(int)(double)x ? YES
F==(float(double) f ? YES
D==(float)d NO
F==--(-f) ? YES
2/3 == 2/3.0 ? NO
d < 0.0 => ((2*d) < 0.0) ? YES
d > f => -f < -d YES
d*d >= 0.0 YES
(d+f)-d == f NO

Intel IA32 FP

8087(paired with 8086 or 8088) was first CPU to implement IEEE

Merged into one chip since 486.

Hardware

Separate FPU from integer unit

Add, multiply, divide

Sometimes others (CORDIC for trig

Extended precision "long double" => 80 bits

Can sometimes be a source of trouble because the registers are 80 bits and data is converted on read/write

FPU is a demented stack machine – programmed very very differently from the integer part of the processor

We will talk a bit more about this after we have done integer assembly

October 10

Mechanics

Handout HW2

Reading still 3, 3.1-3.5, 5.7, but add 3.6

Floating point rehash

Representation

Sign bit (1 = negative) s

Exponent E (exp

)Significand (mantissa) M (mant)

Normal range is 1.0 to less than 2.0

Float = sign bit 8 bit exp, 23 frac bits = 32

"single precision"

Double = sign, 11 exp, 52 frac = 64

"double precision"

Normalized Number Representational

Exp != 0 or 111111111

exp is biased…

Actual exponent E = exp – bias

Single => bias is 127, exp=1..254, E=-126 to 127

Double => bias = 1023, exp=1..2046, E=-1022 to 1023

Bias is generally 2^(m-1)-1 where m is number of exp bits

Mantissa has implied leading 1

Only the bits after the binary point are stored (free bit!)

Min=1.0 (0000)

Max=2.0-epsilon (1111111)

Example

399.0

399 = 0x18f hex

0001 1000 1111 binary
1.10001111 x 2^8

float.

M=1.10001111, mant = 10001111 (+23-8 bits of zero)
E=8, exp=E+ bias = 8+127 = 135 = 10000111
Sign = 0

Denormalized Number representation

Exp=0000000
E = -bias + 1 (-127 +1 = -126)
Signicand assumed to be 0.xxxxxxxx (x=bits in word)
Very very small numbers – closer to zero than the closest normalized numbers

Precision decreases as get smaller
Gradual underflow

If significand is all 0 => "true zero" (note +/-)

Special Representations

Exp=1111111111

Frac=0000000 => INF

+/- infinity
if an operation overflows, the number becomes infinite
1.0/0.0 => +inf
1.0/-0.0 => -inf

frac!=00000 => NAN

sqrt(-1) = NAN
inf-inf = NAN
…

Number line

NAN separate
-inf … - normalized… -denorm –0 +0 +denorm +norms +inf

Special properties of the encoding

FP Zero == Integer 0 (all bits zero)
Unsigned compare almost works with fp numbers

Operations

Rounding

FPU has much higher internal precision than the number representaitions support
FPU first computed "exact" result, then reduces it to the desired precision.

Overflow if exp too large
Rounding to make fit into fractional part

Rounding

Zero => lob off the extra bits in the frac
Round down (towards –inf)

Result is no greater than actual result

Round up (towards +inf)

Result is no smaller than actual result

Round to nearest even (default)

Tiebreaker -> round so that least sig digit is even

Statistically unbiased

Generally the default rounding mode is all you have unless you dive down to assembly

Multiplication is pretty easy

S =s1 xor s2

M = m1 * m2

E = e1*e2

Normaize

Shift right so that M < 2, incrementing E each time

E out of range => overflow

Round M to fit precision

Or give denormed result

BUT

NOT ASSOCIATIVE DUE TO OVERFLOW, ROUNDING

NOT DISTRIBUTIVE

A(B+C) != A*B +A*C*

NOT MONTONIC IN PRESENCE OF INF AND NAN

Addition

Align mantissa

Shift mantissa with smaller exp right until same exp

Do sign mag add of mantissas giving output mantissa, sign. Exponent is exp of larger input

Normalize and round like in multiplication

BUT

NOT ASSOCIATIVE DUE TO OVERFLOW + ROUNDING

CAN GET INFINITIES + NANS

INF AND NANS DON'T HAVE ADDITIVE INVERSES

NOT MONOTONIC IN PRESENSCE OF INF AND NAN

FP in C

Float, double

Casts between ints, floats and foubles CHANGES BITS

FP type to int type => get truncated fractional part. Meaningless if out of range, but usually get saturating math

Int type->fp type => perfect if enough bits in matissa. Otherwise will round according to rounding mode.

Questions

Int x; float f; double d;

X==(int)(float)x ? NO
X==(int)(double)x ? YES
F==(float(double) f ? YES
D==(float)d NO
F==--(-f) ? YES
2/3 == 2/3.0 ? NO
d < 0.0 => ((2*d) < 0.0) ? YES
d > f => -f < -d YES
d*d >= 0.0 YES
(d+f)-d == f NO

Intel IA32 FP

8087(paired with 8086 or 8088) was first CPU to implement IEEE

Merged into one chip since 486.

Hardware

Separate FPU from integer unit
Add, multiply, divide
Sometimes others (CORDIC for trig
Extended precision "long double" => 80 bits

Can sometimes be a source of trouble because the registers are 80 bits and data is converted on read/write

FPU is a demented stack machine – programmed very very differently from the integer part of the processor

We will talk a bit more about this after we have done integer assembly

Machine-level programming

The model: Instruction set architecture

Hardware software intferface

Model versus reality ISA means hardware can change underneath

Language + programmer visible state + I/O = ISA

Defines a primitive language – how bits are to be interpreted as instructions that do input, output, and change the state of the machine

IA32 is a "CISC" ISA

Idea: raise level of abstraction closer to the language to make programmers more productive

Other archs are "RISC" ISAs

Idea: lower level of abstraction to make hardware faster and compiler has easier time of it anyway.

Some new archs are basically just the logic cone!

Reconfigurable machines
Custom-computing machines
FPGAs

RISC is a very nice idea. Was thought that CISC can't be made fast, but intel (and IBM! With 370) managed to do it.

Lots of money
Silicon is 1st order effect
Not "insanely" CISC – Lisp machine or VAX

State of the machine (programmer visible)

Memory in all its forms

Language: instructions

Instructions

Just another interpretation of bits
In most ISAs, instructions = words
In Intel (IA32), instructions can be 1 to several bytes long

Data flow… Take 1-2 items from memory, apply some logical or arithmetic operation to them, and then write the result back to memory

Control flow… decide which instruction will be executed next

Implicit control flow -> the next instruction in memory
Also the fastest

State: Memory

Registers

Fastest form of memory – programmer visible because compilers are very good at scheduling them
Scratchpad memory

Main memory

Most of the rest of the memory hierarchy
That there is a main memory is a part of the ISA. How it is structured is not.
Programmer can ignore how it works and not worry about correctness, but knowing how it works on a specific

implementation can make it possible for her to optimize performance

I/O: special memory locations that other devices than CPU can read and write

We'll talk about this more later when we look at I/O in Unix

IA32

Dates back to 8-bit archs in the mid 1970s (~8K transistors)

Then 16 bit in late 70s, early 80s (first PCs) (30-150K transistors)

32 bit with the 386 in 1985 (300K transistors).

P4 (42M transistors)

Essentially, IA32 = 386

What do the transistors do

Essentially optimize your code as it is running
Scheduling
Parallelism, out of order execution, speculative execution
LOTS OF CACHE MEMORY

Compare IA64 = Itanium (10 M transistors)

IA64 very different
VLIW machine
Program it not with individual instruction but with packets that the compiler/user guarantees can be done in parallel
Lots of debate whether this is a disaster for intel or not
Not clear whether compilers can do this and not clear if hardware will scale
Floating point perf if 1st implmenetations great
Int really terrible
Ultra-slow IA32 compatibility mode

Switch to overheads at this point class 05 ~slide 7

The model

CPU / memory, addresses, data, instructions
CPU

EIP => program counter, adx of next inst
Registers
Condition codes – "extra outputs" from instructions – ie add's carry out

Generally use CCs to decide branche

Memory

Byte addressable array
Per-thread stack abstraction

Used to support procedure calls

October 12-October 24 were slides

October 25, 2001

- **Mechanics**
  - **HW2 in**
  - **HW3 will be available later**
  - **Lab session: Monday 8-9:30 in the lab**
  - **Midterm: Tuesday, 11/6, 6-7:30, CS classroom – makeups before or after, will cover up to but not including linking**
  - **Reading for next time: 6.5-6.7**
- **The memory hierarchy**
  - **Register, L1, L2 Cache, main memory, disk**
  - **Different technologies**
    - ***registers – latches or similar***
    - ***Cache -> typically SRAM***
    - ***Main memory -> DRAM***
  - **Register**
    - ***Flip-flop***
    - ***Works at clock rate***
    - ***Usually arranged in to register file***
    - ***Typically can read 2 values and write another simultaneously. Often can do more than that.***
    - ***Typically, a few hundred bytes***
    - ***Block size: word size – 4 or 8 bytes***
    - ***Paces processor performance 2x every year***
  - **Cache: SRAM cell**
    - ***Six transistors (as per handout)***
    - ***Bit constantly circulates – auto-refreshing***
    - ***Mutual feedback keeps value from degrading***
    - ***Fast: ~5ns***
    - ***Expensive: $100/MB***
    - ***Typically 32K to a few MB***
    - ***Block size: 32 bytes or so***
    - ***Performance paces processors, but remains $$***
  - **Main Memory: DRAM cell**
    - ***One transistor, one cap***
    - ***Needs to be refreshed externally***
    - ***Charge quickly leaks away***
    - ***Slow: 60ns***
    - ***Cheap $1/MB***
    - ***Maybe 512 MB or a GB***
    - ***Block size: 8 K or so***

Density grows 4X per year, but performance remains flat

Disk

Non-volatile
Ultra-slow: 10ms or so
Ultra-cheap: $0.02/MB
100 GB common
Mechanical
Density grows even quicker, but performance very flat

Processor<->memory gap
Reg<->cache BS~=8 bytes
Cache<->memory BS ~=32 bytes
Mem<->disk BS~=8 KB (virtual memory)
Generally cheaper per byte to access a lot of memory than a little.

A memory chip as an array of bits

SRAM (typicall)

N x m sram:
A linear array of n m bit words (blocks)
Array drives sense amplifiers at bottom that give bits and/or allow us to write
Address decode selects appropriate row
Read and/or write

DRAM

N x m dram:
N blocks, arranged in a square (sqrt(n)^2)
*Bottom is a sqrt(N)*m bit row buffer and a multiplexor*
Each block contains m bits
Address given in two parts
Supply row address, do RAS, copies row to buffer
This is slow
Supply column address, do CAS, m bits in that column are supplied
This is fast
Fast access: RAS, multiple CAS
Writing: RAS, CAS+write (multiple) and the copy entire row back into the array.

SIMM/DIMM

Linear array of DRAMS, each m bits wide

8 DRAMs x 8 bits each => 64 bits wide

Parity "ECC", SECDED and higher level error correcting codes on memory systems

Disk

Ferromagnetic coatings

Think of cassette tape

Domains – minimagnets that can be made to point in one direction or another

Areal Density – bits/area

Areal density has become phenomenal and continues to grow faster than in any other memory technology

Platters, cylinder, head, sector

Arm, HSAs, and heads – flight!

Low end disks: 5400 RPM, typical: 7200 rpm

High end: 15000 RPM

Typically variable number of sectors per track

Sector markers

Standards for treating the disk as a linear array of logical disk blocks or sectors

EIDE / ATA

SCSI

Access a sector at a time

Select appropriate head (fast)

Kick the arm into motion (SLOW)

Wait for it to settle, then read sector marker

Repeat until on the right track (SLOW)

SEEK TIME

Now wait until the sector comes around (slow)

ROTATIONAL LATENCY

Read bits and send to controller

Controller writes them into memory

TRANSFER TIME

The System's Buses

CPU

Reg file, ALU

Bus Interface

I/O Bridge

Main memory

I/O Bus

Disk controller

Disk

CPU/Memory

- **Movl A, %eax**
  - ***Generate address***
  - ***Put address on bus***
  - ***Memory responds with data***
  - ***Bus interface puts data in register***
- **Movl %eax, A**
  - ***Generate address***
  - ***Put address on bus***
  - ***Put data on the bus***

Memory/Disk

- **Initiation from CPU -> bus->io bridge/io bus/controller**
- **Controller writes memory**
- **Controller produces interrupt**

Locality

- **Temporal**
- **Spatial**
- **Consider instruction fetch**
  - ***Temp – happens one after the other***
  - ***Spatial – typically in order***
- **Consider sum of 1d array**
  - ***Temp: loop iteration***
  - ***Spatial: sequential***
- **Consider sum of a 2d array**
  - ***Order of the loops matters***
  - ***Columns in inner loop -> better spatial locality***

Caches exploit locality

- **Put a small amount of faster, more expensive memory in from of a larger, slower, and cheaper memory**
- **Caches keeps copies of data in the larger memory**
- **Managed so that cache "usually" has the data that we will want next**
- **Caches are managed in units of cache blocks**
  - ***For processor<->main memory, 32 bytes is typical***
  - ***L1, L2 caches managed by hardware***
  - ***Memory as cache of disk blocks managed by software***
- **Cache hit, cache miss**
- **Recently used data is likely to be in the cache -> exploit temporal locality**

- **Cache blocks are big and contain nearby data -> exploit spatial locality**
- **Cache hit rate**
- **Average access time**
  - ***Hitrate*speed of cache + missrate*speed of memory + overhead***

Next time…

- **Types of caches and how to write code that that makes them behave well**

October 31, 2001

Mechanics

- **HW3 out**
  - ***Problem 3 bug:***
  - ***Do for direct mapped, fullyassoc, and 4-way set associative***
- **HW1 solutions on web**
  - ***Bug in solutions for problem 4 in floating point***
  - ***Gave a cross-the-board point boost***
- **HW2 solutions on the web on Friday to allow for late handinds**
- **Bomblab in tonight**
  - ***Snapshot of mail spool file tonight at midnight for first grading***
  - ***If late, please send mail to that effect and then send mail again when you are done and want to be graded***
- **Bufbomblab out**
- **Reading: chapter 7**
- **Midterm: Tuesday, 6-7:30, classroom, one 8.5x11 sheet OK. Look at homeworks**

The system architecture

Disk

- **Ferromagnetic coatings**
  - ***Think of cassette tape***
  - ***Domains – minimagnets that can be made to point in one direction or another***
    - **Want to make domain as small as possible, but still distinguishable from agacent domains**
  - ***Reading using induction – coil+head+differences***
  - ***Areal Density – bits/area***

Areal density has become phenomenal and continues to grow faster than in any other memory technology
Platters, cylinder, head, sector
Arm, HSAs, and heads – flight!
Low end disks: 5400 RPM, typical: 7200 rpm
High end: 15000 RPM
Typically variable number of sectors per track
Sector markers
Standards for treating the disk as a linear array of logical disk blocks or sectors

- ***EIDE / ATA***
  - **Pretty simple, grew out of PC MFM disks**
  - **Really rightly coupled to storage, and particularly hard disks**
  - **Cheap**
  - **Slower disks**
  - **Two devices plus controller**
  - **Runs on wire**
- ***SCSI***
  - **Complex and powerful**
  - **General purpose – scanners, etc.**
  - **Fastest disks and hardware**
  - **Runs on wire and on fiber**
  - **Six devices plus controller plus subdevices (logical units)**

Access a sector at a time

- ***Select appropriate head (fast)***
- ***Kick the arm into motion (SLOW)***
- ***Wait for it to settle, then read sector marker***
- ***Repeat until on the right track (SLOW)***
  - **SEEK TIME**
  - **~10 ms**
- ***Now wait until the sector comes around (slow)***
  - **ROTATIONAL LATENCY**
  - **7200 RPM,**
- ***Read bits and send to controller***
- ***Controller writes them into memory***
  - **TRANSFER TIME**
  - **Tseek+trot+ttransfer**

Disk in the tlab machines

- ***Lab machines: IBM Deskstar 45 GXP (DTLA-307045)***
- ***EIDE/ATA***
- ***11 gbits/in^2***

3 platters
6 heads
7200 rpm => 120/s => 8.3 ms for one revolution
27,724 cylinders => 166,344 tracks
512 byte sector => 88 million sectors
Variable densisty encoding
avg #sectors per track => 530 sectors per track
*avg bytes/sec off disk = 530*512 / 8ms = 33 MB/s*
max is about 56 MB/s
max from cache is about 100 MB/s (rarely achieved)
t_seek= 8.5 ms (15 ms from first to last cylinder,1.2 track to track)
t_rot = 8.3/2 = > 4.2 ms
t_xfer = 8ms/530 = 15 us
time to read a sector (512 bytes):
t_seek+t_rot+t_xfer
8.5+4.2+0.015 = 12.7 ms
~40 KB/s
*time to read the average track (530*512 = 271K)*
8.5+0+8.3 = 16.8 ms
16 MB/s
*time to read the average cylinder (530*512*6 = 1.6 MB)*
8.5+0+6*8.3 = 58.3 ms
28 MB/s
*time to read 10 ajacent cylinders (530*512*6*10=160 MB)*
8.5 +0+9*(1.2+6*8.3) = 468 ms
~33 MB/s

RAID

Basic idea: spread data across the disks so that sequential sectors (or larger blocks in, say, a file) spread to different disks
Then, a large request gets spread out to all the disks, which work in parallel on it, increasing the bandwidth to the data
Speedup is rarely linear
Striping
Higher levels of RAID add redundancy using error correcting codes (erasure codes)
Code words also striped across the disk
If a disk fails, the raid controller can still reconstruct the data from the other disks

When you plug in a new disk, the raid controller reconstructs the data that was on the failed disk rebuilding your redundancy
Hot Plug

Lab machines: IBM Deskstar 45 GXP (DTLA-307045)
11 gbits/in^2
3 platters
6 heads
7200 rpm => 120/s => 8.3 ms for one revolution
27,724 cylinders => 166,344 tracks
512 byte sector => 88 million sectors
avg #sectors per track => 530 sectors per track
avg bytes/sec off disk = 530*512 / 8ms = 33 MB/s

Locality and caches

Temporal

If we accessed it now, we're likely to access it again in the near future
So, caches prefer data that we've accessed recently

Spatial

If we accessed it now, we're likely to access things near it in the memory map in the near future
So, caches prefer data that's near data we've accessed before

Caches usually fetch big chunks around the addresses that we ask for

Consider instruction fetch

Spatial – typically in order
Temporal - loops

Consider sum of 1d array

Temp: loop iteration, access to sum var
Spatial: sequential

Consider sum of a 2d array

Order of the loops matters
Columns in inner loop -> better spatial locality

Caches for main memory (others similar!)

Put a small amount of faster, more expensive SRAM in from of a larger, slower, and cheaper DRAM

An access is a memory address read or write

The memory address is used to find a word in the cache (if it's there)

- **Details are in managing this memory**
- **General rule is that the management rules are derived from looking at lots of memory reference traces taken from programs that we want our processor to be fast on**
  - ***SPEC benchmarks***
- **Newer ideas: Make memory references explicit ahead of time**
  - ***Prefetch instruction***
  - ***Pipelined load in DSPs***
  - ***Predict next access***
  - ***Predict next value!***
- **High level concerns:**
- **Caches are managed in units of cache blocks**
  - ***For processor<->main memory, 32 bytes is typical***
  - ***L1, L2 caches managed by hardware***
  - ***Memory as cache of disk blocks managed by software***
- **Cache hit, cache miss**
- **Cache misses**
  - ***Capacity***
  - ***Conflict***
  - ***Tension between these two kinds of misses***
- **Working set of a program/loop**
- **Average access time**
  - ***Hitrate*speed of cache + missrate*speed of memory + overhead***
  - ***0.95*10ns + 0.05*60ns = 12.5 ns***
  - ***Making the hit rate even a little bit better makes a big difference***

Cache management basics

- **Replacement policy – who is the victim**
  - ***Replace the one that will next be accessed furthest in the future***
  - ***LRU and its approximations***
- **Write policy**
  - ***Write-through***
  - ***Write-invalidate***
  - ***Write-back***
  - ***Write-allocate***
- **Instructions or data or unified**

Cache structure

Linear array of cache lines grouped into sets
Cache line contains cache block, tag, valid bit
B=2^b bytes in the block
Line contains B byte block, t bit tag, and valid and dirty bit
S=2^s sets in the cache
E lines per set
Cache size=B*S*E
Correspond to partition of the address into t | s | b

Cache access (read)

Chop address into t, s and b
Select set indexed by s bits
Scan within that set looking for tag t bits
If match, and is valid, select data item from block using b bits plus data item length
If not match or match and not valid, return fail, and now read goes to main memory
But the read will be for the whole block associated with the address
When read comes back, select oldest line in the set and replace it
If the replaced line is dirty, write it back to memory first
Note that we are really just hashing on the address. You can think of this as a hash table with buckets or trees
Trees more typical – log lookup within the set
In hardware, the tree is n log n size, and usually bigger
Why not use the higher order bits? Think about what a hash needs to do – it needs to randomly smear out agencent things

Direct-mapped caches

Each set is a single line
Cheap, less effective.
Lots of conflict misses
Common layout for L2 cache, which are big
Easy to do in hardware
Note how collisions can happen easily
Memory copy code, for example

One way to get around this is to be very cognizant of where objects are allocated in memory that are likely to be used together in loop nests.

Essentially, you can manage the cache yourself by placing objects carefully and structuring loop nests well

Blocked matrix multiply

n-way associative caches

Each set is of size n

More expensive, more effective

Fewer conflict misses

Common for L1 caches, which are small

Harder to do in hardware

Easier to resolve collisions

Fully associative caches

There is only one set

Expensive, very effective

No conflict misses

Rarely done in hardware

Commonly done in software

Disk cache often does this

Buffer cache for disk in the OS

Virtual memory in the OS

Network filesystem caches

Web caches

Basic principle behind all caches

Exploit spatial and temporal locality

Hash on the address (or key) to select a subset of the data in the cache then search within that subset for the actual address – or other ways to do this

Associativity tradeoff between hit rate and time to do search.

Degree of associativity largely determined by disparity in performance between the two components that cache is impedance matching.

Memory mountain characterization

Communication in a parallel or distributed system

Memory to memory copys tend to dominate so you try to eliminate them.

- **But some you often can't – gather of data in application space into application buffer for handoff to send**
- **Parallel arrays and strided reference patterns for this gather**
- **Memory mountain characterizes in terms of temporal locality by manipulating the working set size and spatial locality by manipulating the stride**

Other interesting points

- **Processor memory gap means that if you're going to main memory anyway, you have more time to compute addresses – can store addresses in clever ways**
  - ***Inspector / executor***

November 2

Mechanics

- **Buffer lab delayed until next week**
- **Midterm: T 6-7:30, classroom, one sheet**

Hi-level view of linking

- **Combine .o files and .a files from different sources to produce an executable**
- **The loader can then load that executable into memory and start it running**
- **Works at a pretty low level on on symbols**
  - ***Symbol = functions and global variables***
  - ***Symbol definition: named blob of data with a rudimentary type***
  - ***Symbol reference: a name with a rudimentary type***
- **Does two pretty simple things (devil is in the details)**
  - ***Resolve references to external symbols***
  - ***Relocate symbols***
- **Dynamic linking**
  - ***Delay part or all of the linking step to compile time***
  - ***Lets us update pieces of the software without relinking***
  - ***Lets us keep only one copy of shared software in memory (more on this latter in VM)***

Why?

- **Separate compilation**

- **Modularity**
- **Libraries**
- **Shared libraries**

Toolchain

- **Compiler driver -> cpp -> cc1 -> as -> ld**
  - ***As generates a relocatable object file (.o)***
    - **Can be patched to work at any address**
    - **Has external references that must be resolved**
  - ***Ld takes multiple relocatable object files and libraries (.a files, which basically contain lots of .o files) and generates an executable in which all references are interna***

Example

- **a.c:**

```
#include <stdio.h>
int foo();
int x=5;
int main() {
    printf("%d\n",foo());
}
```

- **b.c**

```
extern int x; int foo() {… return x; }
```

The ELF format for .os

- **Slides**
- **All .os start from zero**
- **Executable and shared library format is the same – different magic numbers and special symbols**
  - ***Entry point***
  - ***All references are internal and resolved***

Symbol table

- **Definitions**
  - ***Strong versus weak definitions***
    - **Strong = functions+initialized global variables**
    - **Weak = uninitialized global variables**
  - ***a.o:***
    - **Function main, size 50 initial value 0, is in .text at offset 30 and is strong**
    - **Global variable x, size4, initial value 5, is in .data at offset 55 and is strong**
  - ***b.o:***
    - **Function foo, size 30, initial value 0, is in .text at offset 3 and is strong**
- **Declarations (external references)**
  - ***a.o***

Function foo, size unknown, initial value unknown, … is an external ref
Function printf, size unknown, initial value unknown, is an external reference

b.o

Global variable x, size 4, initial value unknown, is and external reference

Symbols in C++

Carry namespace and typeid information encoded in the name

Name-mangling

You write void Packet::Serialize(const int fd) const

Symbol is like

Serialize__C6Packeti

Symbol resolution

Read each .o file's symbol table into memory.

Look for multiple definitions

Multiple definitions for strong symbols are not allowed

If you have them, fail

Strong symbol definition takes precedence over weak symbol definitions

If you only have weak symbol defs, pick one.

Note nondterminism?
What if a had int y; and b and float y

Now you've got one definition for each symbol

Now look through all the external references and match them to your definitions

Unresolved external references -> fail

No unresolved -> done

Now you have a mapping from each symbol name to the symbol location (.o file, section, which offset)

Symbol relocation

Assembler has to generate code whether it knows the ultimate location of the symbol or not

Jumps and calls

References to external global variables

So, whenever it doesn't know the ultimate location, it makes a note of it in.relo.text or .relo.data

Note indicates where the reference is, in terms of the segment and the segment offse., which symbol it is using, and how it is using that symbol

Linker needs to know the addressing mode

Easy one: absolute

Location represents the ultimate absolute address of the symbol

Harder one: PC relative

Location represents the offset from the instruction pointer when it is executing this instruction.

When we merge all the .o files, we will have to move symbols from their "initial" locations to their final locations in the merged ELF file's sections

Each section is also assigned a location where it will be placed when the program is loaded
We patch to get to the locations that the symbols will have at load time.

Once the new locations are determined, we need to patch all of the references that we noted in the .relo segments so that they point (either directly or indirectly) to the appropriate things.

Final fixup

Write the appropriate entry point into the elf file

_start

Loading

The exec family of OS calls
Create virtual address space from 0 to 2^n-1 (more on this later)
Split address space into kernel and user space
Map kernel (more on this later)
Create a read-only memory segment at "bottom" of memory and "copy" ELF's .init, .text, and .rodata into it.
Create a read-write segment above that and "copy" .data and .bss into it.
Set OS concept of "brk" (end of heap) above read/write segment
Set %esp (top of stack) to top of memory
The _start
_start->…->cstartup (RTL)->main

Libraries

.a files
Basically, a convenient concatentation of .o files
Can change one .o file and then replace it in the .a file

Linker looks inside of .a files in strange ways (symbol resolution)
Load the library
Scan through the .o files in it to fix external references
Keep scanning until the set of external refs doesn't change any more.
Then throw out the .o files and keep going
Library interdependencies

Dynamic linking

Basic idea: partially link the executable and have the loader fully link it when it is run
Lets us change modules without recompiling and re-linking
Lets us reuse modules that are already in memory
Compilation with -fPIC
Link step (-shared)
Don't copy data into executable, just copy enough info so that you can do relocation and resolution at run-time.
.interp section
loader runs the dll loader here
DLL loader
Creates a segment between heap and stack for each shared object.
Relocates the shared object to that position
Relocates references to symbols in those libraries
PIC
Make it possible for each process to map the library into a different segment location without being updated by the dll loader
Basically, do external refs through a global offset table whose position is known
Lazy update of the table and the functions they point to so that calls are basically just one extra indirection

November 7, 2001

Mechanics

hw2 back
exploit lab out
reading for today: 8,8.1-8.4
reading for next time: 8.5-8.8

- **control flow: order in which instructions are executed**
  - **flow from I1 to I2 to I3 ….**
  - **high level movement of eip (successive instructions)**
  - **branches and jumps and call/return cause eip to change drastically**
- **Exceptional control flow**
  - **but eip can also change in response to external and internal events not captured by program state**
  - **Events**
    - ***Network packet arrives***
    - ***Disk read finishes***
    - ***user hit ctrl-c***
    - ***division by zero***
    - ***timer***
  - **Exceptional control flow exist at all levels of abstraction**
    - ***Application***
      - **Handling program errors**
    - ***Langauge:***
      - **C++ or java exceptions: try/catch**
      - **setjmp longjmp**
    - ***operating system***
      - **signals**
      - **process context switch**
      - **error returns**
  - **General idea**
    - ***process chugging along, get to instruction Icurr***
    - ***event happens, next eip is in an exception handler***
    - ***exception handler finishes execution***
    - ***control passes back to either Icurr or Inext depending on type of exception (or it may not return at all)***
- **Exceptions at the operating system level**
  - **Hardware and software**
  - **Hardware defines the types of exceptions that are possible. OS supplies a handler for each type**
  - **Exception table**
    - ***maps from the exception number to a pointer to the exception handler for that exception***
    - ***Processor detects when exception occurs and jumps through the table to the exception handler***

When we jump through the table, the machine is put into supervisor/kernel mode (as opposed to user mode)

Protection model in i386 is actually way more powerful and cooler than what we describe here, but the market has seen fit not to use it.

exception handler runs code to fix the problem or otherwise address it, and then "returns"

Memory map

Kernel mapped into user space, free space at bottom for exception table. Special hardware register points to start of table.

Now running in supervisor mode, the kernel code and data can be accessed.

Types of exceptions

Fault

Synchronous

examples:

divide by zero, general protection fault

page fault

Abort

Synchronous

Typically a hardware error or a serious software problem

Handler does not return

example:

parity check

machine check

double fault

Trap

Intentional exception

Special INT instruction

For system calls

Synchronoous

Interrupt

Caused by external event (signal from I/O device)

A wire going to the processor. voltage change->interrupt asserted

Asynchronous

Draw processor/memory/io/controller, interrupt controller

returns to next instruction

Examples:

I/O event: network packet, disk read, keyboard, serial, TIMER

Reset button

CTRL-ALT-DEL

maskable versus unmaskable

Process abstraction

Imagine that we can have as many of these address spaces in the system as we want (tell you how later in VM)

Imagine that they are independent

Imagine that we can select between them pretty quickly.

Can use exceptional flow control to switch between them.

Exception handler can select ANOTHER address space and then return control to IT.

Exception handler must change the whole execution context

Registers, including eip

Address space

Code, stack, heap, …

OS-structures

open files, environment vars, ..

This execution context is called a process

Operating system lets you have many of them

Each process "thinks" it's the only program running on the system

private address space

logical control flow

to the process, it just looks like certain instructions take a really long time to finish.

Not allowed to touch kernel space, though.

Has to request that the kernel do things through a trap.

Kernel can then "vet" what the process wants it to do.

Exception handler changing context (and switchting to another process is called a CONTEXT SWITCH

Possible point of confusion… context switch versus call to exception handler

exception handler operates in context of the current process, but it is in kernel mode so it can touch the kernel stuff

Call to exception handler is cheap

Context switch is expensive

When do context switches happen?

Timer interrupt handler

~100 times per second on linux

Context switch if there is another process ready to run and the current process's quantum is done. quantum typically >10ms

System call

read from disk slow, so OS initiates the read and then context switches to another process

I/O interript handler

read from disk done. So OS can now switch back to the process that called read.

Scheduler

Actually, you can think that when the handlers are done, they call the scheduler. Scheduler decides whether a context switch is now appropriate and which process to switch to.

Scheduling outside scope of class

Process control in programs

Error handling for unix systems calls

negative return value indicates error has occurred.

error code stored in the global variable errno, translate to string using sterrror, print unsing perror

Typical schemes

error => exit

book uses wrappers for the system calls that basically just print the error string and then exit.

Translate error code into app error code and pass back for higher levels to deal with.

translate error into C++ or java exception and pass up the call chain

Processes are arranged as a tree

Each process has a pid and a ppid

getpid, getppid

pstree –a

ps auxww

The root of the tree is the init process, which the kernel starts after it has booted.

Process state from programmers perspective

Running or ready to run

stopped due to signal (suspended)

terminated

by signal

return or exit

Process progress

- sleep(secs) / usleep(usecs) / pause()
- system calls that context switch away from the process
  - *read/write/lseek/open/close/select…*
- Signals

Process environment

- argc and argv[] and envp[] passed in to main
- getenv to get environment variables
- setenv to set environment variables
  - *only for itself and its children*

Process execution returns a value to its parent when it terminates

- via return from main or by exit(rc) call

Processes are created by cloning parent processes

- The fork() call
- rc=fork();
  - *<0 => error*
  - *=0 => am child process*
  - *>0 => pid of child process*
- RETURNS TWICE when sucessful
- Child process has a new address space that is separate from, but a duplicate of the parent.
- Child process executes concurrently with the parent process
- Child but shares open files (file descriptors) of the parent)

The parent must explicitly request the child's return code

- waitpid(pid,&status,options)
- normal case – stall until child with pid terminates, get return code back in status
- Can set to be non-blocking
- Also, regular wait() call to wait for any one
- status encodes the return value from the child AND the circumstances of how it was terminated. Macros in book discuss how to get each component book
- Zombies: processes that have terminated but their parent process hasn't called wait on them yet. Kernel needs to keep the return value around until it can give it to a parent.

A process can run a new program in place of the program it currently contains

- exec family of functions
- execve(char *filename, *argv[], char *envp[]
  - *argv=>multilevel array*
- argv[argc]=0;
- stack layout on after call
  - *env strings*
  - *arg strings*
  - *env points [0] is lowest in memory*
  - *argv pointers [0] is lowest in memory*
  - *envp, argv, argc (high to low address)*
  - *stack from from main*

Putting it together -> shell

- normal state – stalled in read waiting for user, other processes running
- command typed -> interrupts -> keystrokes -> read returns -> fgets
- When complete command line, does parse
  - *example: ls *.c*
- Shell expands wildcards
  - *ls foo.c bar.c*
- shell forks
  - *parent waits on child*
  - *child execs ls with foo.c and bar.c and environment*
  - *when ls finishes, parent gets its return code.*
- shell goes back to reading
- ls *.c &
  - *same except don't call wait right away*
  - *instead, ask kernel to send you a SIGCHILD when any of your children die. Then you just do waitpids to find out which one*
  - *More on signals on Friday.*

Aside: virtual machines

An operating system that handles all exceptions in such a way that it appears to its "applications" that they are running on the raw hardware.

The "applications" can then be entire operating systems with their applications – or more virtual machines.

Old idea from the '70s that's hot once again

IBM mainframe VM OS

Typical: run VM, under VM run MVS for batch/transaction processing, also run a bunch of interactive CMSes (ie, mainframe DOS). Run development VM and its children, run development OSes

Current IBM idea

run multiple copies of linux on mainframe hardware – a whole virtual cluster.

As your traffic grows, you migrate some of the virtual linux machines to real hardware to make them fast.

VMWare

Run linux on windows, run windows on linux, …

November 9, 2001

Mechanics

Expect to have midterms back next Wednesday

Bomblab performance back today – good job

HW 4 and Malloc lab shifted back (new syllabus for info)

Reading for next time: 10, 10.1-10.6

Exceptions as underlying mechanism that OS can use to create the process abstraction

Beyond the process abstraction – threads

We'll only talk briefly about this here and then a little bit more at the tail end of the course when we consider Concurrency within an application

Weak area of the book is that it concentrates on threads as concurrency, ignoring other approaches that are also commonly used. We'll try to touch on them all

Much more on concurrency and its issues in an operating system class

Kernel Threads

So far: single thread of execution in a process

Context switch means we go to a completely different address space, completely different

register set, and a completely different kernel context (open files)

Thread context switch – we only change the register sets

Multiple threads of execution in a process => multiple stacks

Have to be careful stacks don't collide

The threads communicate through their shared address space – this also means that access to shared region has to be very carefully controlled

Hard to get right

One of the reasons why there are non-thread approaches to concurrency

Two threads, each doing I++;

I=0 at start

movl (location of i), %eax;

incl %eax

movl %eax, (location of i)

What is value of I after both execute function?

Believe it or not, we can also build thread context switches entirely at user level

Signals

Exceptions pushed up to the application level

Can be many different signals – it's an OS abstraction

Intended to look similar to an interrupt

Process provides signal handler instead of kernel

When kernel delivers the signal, it calls the handler out of the blue – control suddenly jumps there.

Sending a signal

kill(process id\|process group ,signal number)

Or from from command line using the kill command

What's a process group?

All the processes that have the same process group number

getpgid();

Processes inherit process group numbers from parents

Processes can change their or other process's pgids

setpgid(pid,pgid)

Shell uses this to group processes into jobs (especially background jobs) so that it's easy to stop/kill/etc jobs

What are some signals?

SIGHUP – user disconnected
SIGINT – ctrl-c
SIGILL – illegal instruction
SIGFPE – floating point exception
SIGBUS – trying to access memory that doesn't exist
SIGSEGV – trying to access memory you don't have rights to
SIGABRT – Abort!
SIGKILL – kill immediately
SIGQUIT. SIGTERM, – please quit gracefully
SIGSTOP , SIGCONT
SIGCHLD – a child stopped or died
SIGALRM – user timer went off
SIGWINCH – window size changed
SIGPWR – power just died (sent due to a UPS)
SIGIO – I/O now possible on some file you asked me to keep an eye on.
SIGURG – urgent data on some network connection
SIGUSR1, SIGUSR2 – user signals

How do we react to signals by default?

Terminate
Terminate and dump core
postmortem debugging
gdb myprocesss corefile
Stop until get SIGCONT
Ignore it

How do we change how we react?

signal and sigaction
oldhandler* =signal(signalnum,our handler*)
void handler(int num);
Special handlers
SIG_IGN
SIG_DFL
signal(SIGINT,myctrlchandler)
signal(SIGPIPE,mypipehandler);
signal(SIGSEGV,myrepairhandler);
Can't do it with all signals – some are unmaskable
SIGKILL will always terminate. You can't override it.
SIGSTOP will always stop

How do we get periodic signals

alarm(seconds_between_ticks);
signal(SIGALRM,myalarmhandler);

Kind of like a timer interrupt, eh?

Combine SIGALRM, setjmp/longjump(later), SIGIO and you have the building blocks to implement threads at user-level within your process!

Lots of packages that do this

signal semantics are a bit of a mess (different on different unix implementations)

interaction with system calls
posix sigaction call
interaction with threads

Setjmp/longjmp

Non-local jumps
tag where you are in the stack (perhaps one of many), and then later can return instantly to that place.

setjmp(jmpbuffer);
la la la
longjmp(jmpbuffer);

Perhaps even a signal handler can do a longjmp in response to a signal

SIGPIPE in web server => cleanup connection, longjmp back to accept
SIGALRM => figure out current location in current thread, setjmp that context, then longjmp to saved context of another thread

November 14, 2001

Mechanics

reading

today: 10-10.5
next time: 10.7-10.8

exploit lab extra time to Monday
malloc lab out this Friday
exams back

distributions

props

disk, simm, processor

Playing around with the tools

ps auxww
pstree –a loop
strace a process to see system calls
gdb foo core
kill3
The /proc filesystem

- **The /dev devices**

Virtual memory

- **So far we've made a identity mapping from addresses our program generates to addresses that we send to the memory system**
  - ***Physical addressing***
- **And we assumed that all the data was kept in main memory**
- **We talked a little about multiple address spaces and how when we switch from process to process, we need to "copy" in the new process's memory**
  - ***Remember that the process thinks that it has memory from 0 to 2^n-1***
  - ***But EACH process thinks this.***
  - ***And what happens if the machine doesn't really have 2^n bytes of memory????***
- **How can we be smarter than copying while still maintaining protection?**
  - ***Address space abstraction is what we want. The idea of copying is just a possible implementation.***
- **Virtual addressing**
  - ***Basic idea: pass addresses generated by program through a function or table that maps them to actual physical addresses.***
  - ***Now our process can have its 0..2^n address space but the actual data can be at arbitrary memory locations***
- **But I still don't have enough memory!**
  - ***Translation can fail => you're trying to access parts of your address space that you don't have physical memory for***
    - **Not as much of a problem as you might think – remember that the address space of a linux process has lots of giant holes in it.**
    - **Can either completely fail when we try to access memory in a "hole", or the operating system can allocate memory for us and update the translation table.**
  - ***Translation can be to memory that is currently on disk (CACHING!)***
    - **Translation fails, OS brings memory in from disk and places it SOMEWHERE in physical memory, updates translation table, and tries again.**
    - **All the issues of caching exist here too**

miss rate, hit rate, average access time
cache structure (usually fully associative because disk is so slow)
replacement policy, write policy, etc.

Disk is called "backing store"

Usually you think of caching as being over a swap partion or paging file, but we can also think of a regular FILE as being the backing store for pages of memory

But isn't this really really slow – after all, you're adding a translation to each memory reference

Page granularity

Entries are contiguous blocks of memory, not one byte
Page size on Intel: 4K
Like blocksize in a cache – big benefit if there is spatial locality in the references

Translation table -> page table

Page Table Base Register
Page tables are stored in main memory
But the pages that they are on are marked as accessible only by the kernel

Address is

virtual page number | page offset
do virtual page number -> physical page number, concat with page offset
send to memory

Hardware cache the translations

Translation lookaside buffer (TLB)

Generally very small and very associative

But won't these page tables themselves be big?

multi-level page tables (just like multi-level arrays)

Split VP number into VP number for 1st table, 2nd table, …

On some hardware: giant pages

What's the path?

common case

generate virtual address in program
mapping is in the TLB and permissions are appropriate
generate physical address
goto main memory

uncommon case

generate address in the program
mapping is in TLB, but permissions wrong: raise protection fault and let OS figure out what to do.

mapping is not in the TLB
(on intel) hardware looks through page tables in main memory to find translation, updates TLB
translation found and permissions OK: update TLB, translate and go
translatikon found and permissions not OK: raise a protection fault
translation not found: raise a page fault
page fault and protection fault handlers takes care of situation
permissions are really bad (there are some tricks that are played here – copy on write) => send process signal
permissions are only appear to be bad (copy on write, mark writable, allocate) => update memory, PT, and restart instruction
entry is missing => should we allocate page? If so, do it and restart. If not, send process a signal.
entry is invalid => page fault => Is the page on disk? If so, bring it back into memory and restart instruction
Note, scheduler gets involved here.
(on some other machines) – if it's not in TLB, simply generate a page fault and let the OS figure out what to do (normally just runs through page tables and does the above itself)

But what about multiple programs?

each process has its own page table.
Page table is part of the OS context of the process
Each entry has permissions associated with it
valid, dirty, read write exec (user), read write exec (kernel)
Only the kernel can change the translation table, so the kernel can protect one process from another
And itself from the process!

What are some useful things to do here

aid to multiprogramming / protection
keep processes separate from each other
keep them from touching the kernel except through system calls

aid to linking and loading

linker sees simple linear address space model… process is always 0..2^n-1

loader is easy -> make regions of the executable file be backing store for their appropriate regions in the address space (remember that ELF is really basically a memory image). Then the OS will "page-in" the executable as it runs – demand paging

Sharing

Can map a physical page into MULTIPLE processes' address spaces. That page is then shared between them and can be used for communication

Even read only pages -> map the libc shared library file into everyone's address space

Malloc

Memory allocators can treat the heap as being physically continguous memory

November 16, 2001

Mechanics

Exams back

HW3 back

Reading: 10.9-10.12

malloc lab out

exploit lab due Monday – note comments about what your readme should contain

Summarize VM so far…

OS wants to provide the following abstractions

Each process has a 0..2^n-1 address space

Simplifies linking/loading

Simplifies dynamic memory allocation within the program

Address space may have holes

Not all addresses have data

Each process's address space can be larger than the amount of physical memory in the machine

"extra" data seamlessly stored on the disk

Physical memory is thus a cache over actual memory on the disk

There are multiple processes, each with a separate address space

Processes can not touch each other's memory unless permission is explicitly provided.

Some parts of the address spaces can be shared

shared memory for shared libraries, communication

A process's address space (virtual address space) is split up into pages (virtual pages)

4K pages on the intel

Each address now consists of a virtual page number and an offset within the page

The machine's memory (physical address space) is also split up into pages (frames or physical pages)

For each process, the OS maintains a page table, which maps from a virtual page number to a physical page number

virtual page number is index into table

content of table is

valid bit (ie, in memory?)

Does that virtual page number currently map to any physical page?

An invalid page may simply be on disk – the OS has auxillary structures to figure this out.

permissions (read/write/execute for user and kernel mode)

physical page number (if valid)

often a disk block if it does not

The table is itself stored in memory and may be pageable.

OS tells hardware where the page table is via a special register (page table base register) accessible only in kernel mode

This means that switching from one table to another (ie, on a context switch) is just a register load

each memory reference is translated by the hardware through the page table

If the PT entry is invalid or the permissions are not sufficient for the kind of access, the hardware raises an exception and the OS becomes involved

invalid + really on disk? load and resume process.

invalid+not really on disk+on stack? allocate

bad permissions? sends the process a signal

plus other tricks like COW

The hardware makes this acceptably fast by having a cache on translations, a translation lookaside buffer (TLB) so that it doesn't always have to go to the page tables.

Multilevel page tables

Since virtual address space has lots of "holes" having one large table for it can be inefficient
Instead, create a table of tables. Or table of tables of tables.
Split virtual page number part of address into sections corresponding to each table

For example, 32 bit address, 4K page => 20 bits of virtual page number => 2^20 entry table
Split into 2 10 bit regions. => 2^10 entry table, each entry is a pointer to another 2^10 entry table.
1st table is indexed by 1st 10 bits of address. Thus, each entry represents 2^22 bytes in the address space (2^12 byte page * 2^10 pages)
if 1st level entry is "blank", then translation stops. Thus we can chop up holes into 1024 page chunks and only have one page table entry for each chunk

Virtual memory on Intel + Linux (Slides)

November 21, 2001 (SLIDES)

Mechanics

HW4 delayed until after thanksgiving
Delay on grading, HW4, etc due to, ironically, a security breech this weekend
ssh compensation attack patch has a kind of stack overflow bug itself, machine cracked. 2.5 days to clean out from the mess so far.
Note that reading is all of chapter 10
Responsible for more sophisticated allocators, garbage collection
Reading for next time: 12-12.4

SLIDES

November 28, 2001

Essentially, this covers the unix systems programming in a nutshell handout and the book's material

Mechanics

Exploit labs back
HW4 out
Final exam: December 11, 9 am, cs class room

Reading: Today 12-12.4, next time: 12.5 + unix systems programming handout and sockets programming handout.

Revisit mmap

mapping chunks of files into the address space

The file abstraction

A stream of bytes without interpretation

The namespace for files

single rooted hierarchy

maps from a pathname to an inode number

mount points

links and symlinks

filename versus pathname

absolute path versus relative path

the current diretory

may be multiple pathnames for a file

The partition and inode number

Each file has only one partition+inode number

Flat

inode has detailed info about the file

int stat(pathname,struct stat *s)

inode properties

owner, group

permissions for owner, group, all

Stdio versus the Unix interface

File descriptors and what they mean to the OS

Inode table

cached inode data from the disk + reference counts

The OS's open file table

inode, refcount, position, type of access

The process's file descriptor table

pointers to open file table entires

On fork, get copy of this table + open file table counts are increased.

The open/read/write/lseek/close interface

Unix error handling -> return –1 and set errno

fd = open(pathname, flags)

O_RDONLY, O_RDWR, O_WRONLY

fd is a handle to the open file

implicit position, initially set to zero.

count = read(fd,buf,len)

may read fewer than the requested number of bytes, returns –1 on error. returns 0 on end-of-file
increments position by the count.
Blocks until there is data (can change this)

count = write(fd,buf,len)

may write fewer than requested number, returns –1 on errors
increments position by count
blocks but can be set to non-blocking

position=lseek(fd,offset,whence);

Seeks to a new position offset from current position, beginning of file, or end of file.

close(fd)

Non-blocking I/O

open with O_NONBLOCK
or use ioctl to set to non-blocking
"blocking" functions now return "EAGAIN"

Signal-driven I/O

fcntl(fd,F_SETSIG)
OS sends you a SIGIO signal whenever there is data available on the fd or it can be written
It's up to your signal handler to figure out which fd caused the signal and why

Select and poll

What if you have a bunch of file descriptors and you want to read from them as data becomes available? ie, you have multiple I/O things happening in arbitrary order and you must handle them.

could do a thread per fd
could do non-blocking I/O and just keeps scaning over the fds.

int num=select(int maxfd, fd_set *read, fd_set *write, fd_set *except, timeval *timeout).
blocks the process until one or more fds become available for read, write, have exceptions, or a timeout occurs, or (in some systems) a signal is delivered.
Event-driven programming based on select is common

Beyond files

Unix attempts to map many things into the filesystem model and lets you access them using the same read/write/etc interface

But the abstraction breaks down in different ways for different devices

/proc filesystem

/proc/processid/mem – the virtual address space of the process

/proc/… various locations in the kernel and device drivers, presented in nice ways

Non-unix filesystems

mounted on the namespace

"virtual filesystem" drivers create "virtual inode" layer on top of underlying filesystem

remote filesystems: nfs, afs, alex

/dev devices

communication with device drivers

/dev/hda – your hard drive

/dev/hda1 – a parition on the hd

/dev/st0 – the first scsi tape drive

/dev/dsp – the sound card

```
fd=open("/dev/dsp",O_WRONLY);
write(fd,data,len)
whamo, you're playing sound.
```

ttys (terminals) and pttys (pseudoterminals)

Your process automatically inherited your open ptty when it was forked
fd 0 -> stdin
fd 1 -> stdout
fd 2 -> stderr

pipes

anonymous (unnamed) files that support oneway communication

fifos (named pipes)

named files, mkfifo call, otherwise like pipes

unix domain sockets

named meeting points for establishing connections between processes using sockets interface.

ioctl – the kitchen sink of device-specific stuff

fd is /dev/dsp fd

ioctl(fd,…) might be used to set the the type of data, or to adjust the mixer, etc.

Essentially, this covers the sockets in a nutshell handout and the reading in the book

Mechanics

- **Reading: 11-11.4, sockets in a nutshell handout, systems programming in a nutshell, concurrency handout TBA**

The network stack as another hierarchy of abstractions

- **"stack" here means "layered architecture"**
- **Each layer may also have several components**
- **Internet protocol stack model and our presentation**
  - ***physical – wires, signaling***
    - **copper, optical Ethernet; atm; modem line**
    - **bit encodings**
  - ***data link – MAC, LAN stuff (Ethernet)***
    - **Ethernet collision control**
    - **Ethernet switch and hub auto-configuration**
    - **Learning bridges**
  - ***network – routing***
    - **IP**
  - ***transport – end-to-end communication***
    - **TCP**
  - ***application***
    - **HTTP**
- **Also OSI stack model**

Abstractly, a network is a graph

- **nodes are either hosts or "routers"**

Packet switching

- **Slice up messages into little chunks. Stamp their destination address on them, and inject them into the network**
- **Each router gets the chunk a little bit closer to its destination**

A network (ours and beyond)

- **Ethernet network: TLAB, 125, (100 mbit switches)**
  - ***NIC in machine***
  - ***switch -> full connectivity given permutation***
  - ***hub -> only one NIC speaks at a time***
- **2^{nd}+3^{rd} floors**
  - ***Ethernet network: 100 mbit port -> 100 mbit switches -> gigabit links to router***
  - ***Ethernet network: 10 mbit port -> 10 mbit hub -> 100 mbit switch -> gigabit***
- **Networks of networks**

the birl router – ties these networks and the outside world together
Outside connections: 2 gigabit links to campus network
campus network peers with other networks
NON ETHERNET NETWORKS
ATM-based
modems
DSL

Point of IP: virtual network on top of these networks of networks

Think of a network as being a node as far as IP routing is concerned
But IP is also end-to-end, so application doesn't have to know the details. Can just use IP.

IP is UNRELIABLE DATAGRAMS

may not arrive
may not be corrupted
may not be reordered
Visible at application layer as the UDP protocol (UDP is a thin layer on top of IP)
No connections

TCP implements RELIABLE BYTE STREAMS on top of IP

bidirectional
A byte pushed in one end of a TCP connection will eventually emerge at the other end uncorrupted and in the order in which it was pushed in.
Connection oriented

Application-level protocols such as HTTP, FTP, database protocols, etc. typically implemented on top of TCP

Streaming media typically done on UDP

Programming

Berkeley socket interface

windows has it to, although it's been subsetted and then extended so it's a little different

Core idea:

"opening" and "closing" a connection is different from a file
What's the name? There are two principles now

Clients and servers

Active open

Passive open

Associations:

(client ip, client port, proto, server ip, server port)

opening means basically to fill in these parameters on both client and server side

Opening is a little inverted:

create a socket (gives you a file descriptor for a socket with proto)

give it a name (ip + port)

Connect the other end

use the fd just like a file

except no seeking

reliability

Client:

int fd = socket(AF_INET, SOCK_STREAM,…)

connect(fd, sockaddr *address_of_server, int len);

Local side is automatically bound to some available port and INADDR_ANY

sockaddr_in

address family (AF_INET)

ip address of host (network byte order)

port in network byte order

read/write

close(fd);

Server

int fd = socket(AF_INET,SOCK_STREAM,…)

bind(fd,sockaddr *address_on_this_machine,intlen);

listen(fd,5);

int fd2 = accept(fd,…)

read/write fd2

accept more connections on fd!

close(fd2)

December 5, 2001

Essentially, this covers the reading and the concurrency handout (the handout in more detail)

Mechanics

Malloclab and HW4 due Friday

Final: Tuesday, December 11, 9-10:30, covers linking to end of class, non-cumulative, one 8.5x11 paper sheet allowed, calculator recommended

- **Reading for next time: 12.6-12.9, handout TBA**

Key reasons for concurrency in programs

- **Need to respond to events that arrive in an unknown order**
  - ***ssh***
- **Program logically decomposes into tasks that can run in parallel**
  - ***word processor***
  - ***connections to a web server***
- **Performance: exploiting multiprocessors and distributed environments**

Concurrency and communication typically go together

- **Communicating tasks**
- **Two high-level models**
  - ***message passing (send/receive)***
    - **messages ordered in the communication channel**
    - **messages generally buffered, but buffers are not infinitely large**
  - ***shared memory***
    - **synchronization primitives needed to control access**
  - ***Each has theoretical basis***
  - ***Each includes data transfer and synchronization***
  - ***Neither lets you avoid critical issues like deadlock***

Concurrency/Communcation and its goals

- **Model**
  - ***Application consists of tasks that must be executed to completion***
  - ***Tasks communicate with each other***
- **Goals**
  - ***One task blocking should not make other tasks block***
  - ***Every task should make progress***
  - ***Tasks should be treated fairly (notion of fairness depends on application – ie, priorities)***
  - ***Concurrency and communication must avoid deadlock, livelock, and data corruption/bugs***
  - ***Minimize overhead***
  - ***Extensibility to multiprocessors and distributed environments***

Subtlety of concurrency/communication bugs – the software engineering side of choosing a concurrency model

Bugs can now show up under specific timing conditions that may be difficult to reproduce

"Heisenbugs"

Often painful to debug

software is deployed, deadlocks…

No core file, no one at customer site to connect to program with gdb and see what's going on

Little information about the chain of events that what lead to the deadlock (logs, perhaps)

More synchronization than is "Sufficient" (defensive programming) *may* make the program less likely to contain such bugs, but it will also probably make it slower… where is the right tradeoff?

The benefits and costs of using pre-emptive multi-tasking instead of cooperative multitasking

pre-emption – less code

cooperative – exact control over when context switches happen

Concurrency Approaches with application to a web server

Web server socket pseudocode

Want to accept new connections as they come in

Want to handle existing connections as they become ready for reading/writing

Will ignore synchronization in this…

Processes

Fork-based web server

1. **startup**
2. **accept connection**
3. **fork new process**
4. child does read/write/close

close fd in parent

5. non-blocking waitpids for outstanding children

6. goto 2

Writing logs – perhaps in parent before step 6

Or have children do it, but they must lock the file - flock

Or, shared semaphore among all children

Creation overhead for processes – Process pools

mechanism for passing fd to a process

Threads

OS schedules thread contexts

thread context = registers + stack

Thread executes in context of a process, sharing address space, open files, other OS-level

pthread interface

int pthread_create(pthread_t *tid, pthread_attr_t *attributes, void * (*f)(void*), void *arg)

int pthread_detach(tid)

int pthread_exit(void *threadret)

joins all threads if called from main

int pthread_cancel(tid)

int pthread_join(tid, void **return)

Semaphores

int sem_init(sem_t *sem, int pshared, int val);

int sem_wait(sem_t*) – (P) (atomic: while (sem<=0) {}; sem--;

int sem_post(sem_t*) - V: atomic sem++

Web server with threads

1. startup

2. accept connection

3. spawn thread + detach to handle connection

thread function does read/write/close

4. goto 2

Synchronization – log file

Creation overhead for threads - Thread pools

Started talking about select-based I/O here. Contents in next lecture.

December 7, 2001

Mechanics

HW4 due at exam

Lab due at midnight tonight, handin as described

Exam: 12/11, Tuesday, 9am here, sheet of paper, calculator

Today's goal: finish up select and signal-driven I/O, talk at a high-level about protocols like http, talk at a high level about distributed systems

Cooperative versus pre-emptive approaches to concurrency

cooperative approaches put total control in your hands, but require that you implement state maintenance, and scheduling. Also has very low overhead.

Big advantage: when your task is running, it executes just like a purely sequential program – easier to debug, easy to incorporate other code, etc.

Big disadvantage: you personally have to insure liveness – that all tasks make progress.

pre-emptive approaches tend to make state maintenance easier, and take care of scheduling for you, but you no longer have as much control.

Higher overheads

Big advantage: liveness is taken care of by the thread or process scheduler.

Big disadvantage: synchronization – your task is no longer equivalent to a sequential program

Task state and context switches

In thread-based or process-based concurrency, task state can live in stack and registers. It is automatically saved and restored through the process of context switches

Not completely correct – you can have state outside of these in which case you need to manage it yourself

In cooperative schemes like select, it is your responsibility to do save/restore of state.

Harder to get right, but potentially much faster

Applies also to partially pre-emptive, partially cooperative approaches like signal-driven I/O.

Select-based I/O: an example of cooperative multitasking

The connection list: file descriptors, current state (how much read, how much written, what phase of the protocol, etc)

Have OS wait on multiple events

setup

put acceptfd on read list

create fd lists for read, write, exception

call select

scan output fd lists for active fds

if accept, do accept and put new fd in read list

if fd, do read/write/close as necessary

This is the same as we discussed in I/O

No preemption!

However, must explicitly maintain the state of each fd from select call to select call

What if something blocks or takes a long time? Uh Oh. Programmer's responsibility to see that this doesn't happen.

Signal-driven I/O: an interesting in-between case – preemption and cooperation.

- **Have OS send a signal to you whenever an fd is readable/writable**
- **setup**
- **install signal handler for sigio**
- **fcntl(acceptfd,F_SETFL,O_NONBLOCK);**
- **fnctl(acceptfd,F_SETSIG,0);**
- **goto sleep**
- **signal handler wakes up, sees what fd is active**
- **if acceptfd, do accept, add connectionfd to list, do fcntls on it, return**
  - ***if other, do I/O, possibly close, return***
- **Signal handler is sequential, so can just access log-file directly**
  - ***Note that main line of program CAN get pre-empted and thus there should be careful synchronization between it and the signal handler***
  - ***Note that the handler has to maintain the state of each of its fds so that when it is called again it knows where to pick up from***
  - ***What if signal handler blocks or takes a long time? Uh oh. It's the programer's responsibility to see that this doesn't happen.***

HTTP protocol

- **Text-based**
- **Request/response (Like RPC, but with only a few methods, GET, PUT, etc)**
- **Request**
  - ***GET filename HTTP/version***
  - ***Attributes***
  - ***empty line***
  - ***GET / HTTP/1.0***
- **Response**
  - ***HTTP/version return_code text_status***
  - ***Attributes***
  - ***Content-Length: length of following data in bytes***
  - ***Content-Type: MIME type of content***
  - ***HTTP/1.1 200 OK***
  - ***Date: Fri, 07 Dec 2001 15:39:18 GMT***
  - ***Server: Apache/1.3.12 (Unix) (Red Hat/Linux)***
  - ***Last-Modified: date***

Content-Length: 12543
Content-Type: text/html
empty line
html document

Client and server like the echo server in the book
CGI is basically just running a program to generate the response
Book has example of an iterative web server with CGI

Distributed and parallel systems (most of this will be discussion of the handout materials)

Why?

Resource requirements
Reliability
Performance

Algorithms

Distributed algorithms must work in the face of errors, failed connections, etc.

Consensus algorithm example (voting) I have a database that's replicated across many sites. The databases are loosely consitent. Write an algorithm so that if I can communicate with any m of the n databases, I can reconstruct the actual value of a record.
Given a distributed database, make sure that all updates to a record are perceived by all users within a fixed amount of time

DNS

Parallel algorithms try to go to a new level of performance.

Sorting takes O(nlogn) on a single machine. How much faster can it be on a parallel machine with infinite processors
Work complexity – the number of operations performed in total
Depth complexity – the length of the longest dependent chain of operations.
Sorting

Sequential: O(nlogn) work (depth irrelevant)
Parallel: O(nlogn) work, O(logn) depth – what this means is that given an infinite number of processors, you complete as sort in O(logn) time while not doing any more work than the sequential version

For lots of tasks, depth complexity is much smaller than work complexity, so there is tremendous potential for speedup.

Processes and communication (sockets or message passing) are the assembly language of higher layers – very primitive

Unfortunately, abstractions have not raised much beyond that, at least from the perspective of the typical developer. Lots of good research results, few products.

Higher-level communication abstractions

- ***Message Passing***
- ***Collective Communication***
- ***Remote Procedure Call (RPC) and friends***
- ***Disributed Shared Memory (DSM)***

Languages – mostly from parallel computing, not that much work has been done on languages for distributed computing.

- ***automatic parallelization and vectorization***
- ***Explicitly parallel languages***
  - **array-oriented,**
  - **collection-oriented**
- ***Interface definition languages (IDLs)***

Consensus and Consistency

- ***Consensus problem occurs at all levels of a system***
  - **Data shared in some way among multiple tasks**
  - **Each task has a view of the data**
  - **How to make these views consistent with each other?**
- ***Consistency models***
  - **Observe the updates to the shared data that some other task is making. What is the order in which they are seen**
  - **Sequential consistency**
    - Seen in the order they are issued.
    - Corresponds to SOME interleaved execution of the tasks
    - Very difficult and costly to achieve in practice
  - **Release consistency**
    - Total order between releases (or barriers)
    - No ordering guaranteed for updates happening between two barriers
  - **Even more "relaxed" forms of consistency.**
  - **Virtual Synchrony**

Hope the class has been enjoyable. Don't forget to do the CTECs and please give me any feedback you'd like. Recall that we want to determine whether to adopt this class for sophomores.