CS270 - Systems Programming

Mr.  Raoul  Finkel
Dr.  Ray  Goldstein
Prof.  Ralph  —

www.cs.uky.edu/~raphael/courses/CS270.html
lru225@uky.edu
0: standard input
1: standard output
2: standard error

```c
char *buf;
buf = malloc(count);

fd = open("/etc/hosts", O_RDONLY);
count = 100;
size = read(fd, where, count);

size = write(fd, where, count);

success = close(fd);

perror("/etc/hosts");
```

free(buf)
foo232.netlab.uky.edu

```
% prog -f --g=foo /etc/hosts ...
  ↑      ↑
what to flags
run     parameters
```

% ls -l
% man ls

---

software tools

---

use (client) spec.
implementation
x86 64 and 64

abstraction / reality physical

efficiency → layout in memory
potential bugs
hardware limits

C int ?

\(2^{31} - 1\)

C float ?

\(\text{le20} + 3.14 - \text{le20} \rightarrow 0.0\)

0.1

Bugs: memory referencing
array ref out of bounds.
Invalid pointers
bad alloc/dealloc
language assistance
Java
JavaScript
Perl
Python

tools

gdb
valgrind
...

Binary representation

bit: 0 or 1

numbers: base 10 100,0
base 2
base 16 (hex)

255.3,0 = 110.42

byte: 0000 \rightarrow \text{1111}
0 0 0 \rightarrow \text{FF FF}
0 \rightarrow 255

base 2
base 16
base 10
unsigned integers in n bits:

\[ 0 \rightarrow 2^n - 1 \]

\[ n = 8 : \ 0 \rightarrow 255 \]
\[ n = 32 : \ 0 \rightarrow 4095 \]

\[
\text{Signed integers in n bits:} \\
-2^{n-1} \rightarrow 2^{n-1} - 1
\]

\[ n = 8 : \ -128 \ldots 127 \]
\[ n = 32 : \ -2048 \ldots 2047 \]

\[ \text{Signed integers 2's complement} \]

[Diagram of 8-bit representation]

To negate a number:
1) flip all bits
2) add 1
\begin{align*}
\text{4 bit examples } (-8 \ldots 7) \\
\text{negate:} & \quad 0000 & \quad 0_{10} \\
& \quad 1111 + 1 \\
& \quad \underline{0000} \\
\text{negate:} & \quad 0110 & \quad 6_{10} \\
& \quad 1001 + 1 \\
& \quad \underline{1010} \\
\text{negate:} & \quad 0101 & \quad -6_{10} \\
& \quad 0101 + 1 \\
& \quad \underline{0110} \\
\text{negate:} & \quad 1000 & \quad 6_{10} \\
& \quad 0111 + 1 \\
& \quad \underline{1000} \\
\text{negate:} & \quad 0111 & \quad -8_{10} \\
& \quad 0111 + 1 \\
& \quad \underline{1000} \\
\end{align*}
Given 32 bits, \(-2^{31} \ldots 2^{31}\)

\[2^{31} = 2^0 \cdot 2^0 \cdot 2^0 \cdot 2^1\]

\[2^0 \quad 1K \quad \sim 1000\]

\[2^{20} \quad 1M \quad \sim 1,000,000\]

\[2^{30} \quad 1G \quad \sim 1,000,000,000\]

\[2^{40} \quad 1T\]

\[4893_{10} > 2^{12}\]

<table>
<thead>
<tr>
<th>Type</th>
<th>bytes 32</th>
<th>bytes 64</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>long long</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>
Byte ordering

int: 4 B = 32 b = 8 hex digits

19 of 8743,0 = 01 234567 1b
first
big-endian

U
last
U
first
little-endian

x/4 b

x/1 x w

Memory organization

addresses are in bytes.

program that is running = process.

process point of view, memory is one long array.

<table>
<thead>
<tr>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
</tr>
</tbody>
</table>

every process has its own memory.
sharing is possible, usually memories are disjoint
physical memory

process memory (virtual)

4 0
0 → 4KB ← page

B 1 2 3 4 5
heap

inistructions data

data formats

integers: 64 bit
32
16
8
String:
array of bytes,
each byte represents a character,
followed by a 0 byte.

```c
#define STRLEN 200
char myString[200];
char *myString = malloc(length+1);
```

character? \(\text{ASCII, EBCDIC, FIELDDATA}\)

Unicode
+ large, 16 bit representation

\[2^{16} = 2^8 \cdot 2^8 = 64K\]
compact representation UTF-8

- need longer memory areas

byte ordering? never a worry.
Boolean Algebra

George Boole

ops: \[ \begin{array}{c|cc}
& 0 & 1 \\
\text{and} & 0 & 0 \\
& 1 & 0 \\
\text{or} & 0 & 1 \\
& 1 & 1 \\
\end{array} \]

not \[ \begin{array}{c|cc}
& 0 & 1 \\
0 & 0 & 1 \\
1 & 1 & 1 \\
\end{array} \]

\[ \begin{array}{c|cc}
& 0 & 1 \\
\text{xor} & 0 & 1 \\
& 1 & 0 \\
\end{array} \]

\[ \begin{array}{c|cc}
& 0 & 1 \\
\text{equiv} & 0 & 1 \\
& 1 & 0 \\
\end{array} \]

\[ \begin{array}{c|cc}
& 0 & 1 \\
\text{shift} & 0 & 1 \\
\end{array} \]

6 << 1 "6 left shift by 1"

\[ \begin{array}{c}
in: 0110_2 \\
out: 1100_2 \\
j << 33
\end{array} \]
j \gg 2
in 0110_2
out 0001_2

signed or unsigned?

int j
unsigned int j
extend sign bit
shift in 0 bits

Compilation

gcc [options]

-o filename output file name
don't compile, just preprocessing
-E compile to assembler
S filename.s
-c compile, don't link
filename.o
-g add extra debugging info
-Om turn on optimization
tools for inspecting programs

`objdump -d`  disassemble
`gdb`          show location, type of identifiers
`nm`           
`strings`      
`od`           
`ltd`          
`dispy`        graphical dissembler

→

Machine architecture
  registers
  instructions
microarchitecture
  caches
  microcode

basics
  x86-64
  x86-32
  ARM

program counter `pc`
other registers
condition codes (1 bit each)
memory
to read an int:

```c
unsigned int myInt;
read(fd, &myInt, sizeOf(myInt))
printf("%d", myInt);
```
Architectures

machine code / assembler code

x86 : 32-bit code 1985

-m32

x86-64 : 64-bit code 2003

registers

Instructions

\[ \text{mov} \quad \text{src, dest} \]

\[ \text{movl} \quad 4\text{bytes} \]

\[ \text{movb} \quad 1\text{byte} \]

\[ \text{movw} \quad 2\text{bytes} \]

operands:

1) immediate : integer constant

\$0 \times 3

\$32

\$a 0 \times 20

only \text{ Ax src}

2) register : \text{Ror bp}

3) memory address : \text{(R or Rax)} 8\text{bytes}

either \text{src or dest, not both}
4) displacement  \( s(\%eax) \)
src or dest, not both
\[ -0 \times 18(\%rbp) \]

5) general  \( s(\%eax, \%ebp, 4) \)
\[ M(\%cr1, \%cr2, a) \]
\[ \left[ \%cr1 + a \cdot \%cr2 + M \right] = \text{address} \]
src or dest, not both.

---

Examples:
Say:
\( \%edx \)
\[ f000 \]
\( \%eax \)
\[ f000 \rightarrow \]
\( 0000 f000 \)
\[ \%ecx \]
\[ 00001000 \]

0\( \times 8(\%edx) \)
address: f008
(\( \%edx, \%ecx \))
address: f100
(\( \%edx, \%ecx, 4 \))
f400
0\( \times 8(\%edx, 2) \)
le08 0l
On Windows:

- Install Xming
- on putty:

General addressing:

\[ M(\text{reg}, \text{reg}, n) \]

\[ i \quad \uparrow \quad \uparrow \quad \uparrow \]

1, 2, 4, 8

purposes:

1) Compute an address

\[ p = 4 \times [i] \]

\[ p = x + i \]

\( i \) in register \%eax

\( x \) starts at a place pointed to by \%ebx

\( x \) is an array of ints

\[ \text{lea} (\%ebx, \%eax, 4), \%rcx \]

\( p \) stored in \%rcx

2) Compute arithmetic

\[ x \times x \times x \]

\[ j \times 12 \quad j \text{ in } \%eax \]

\[ \text{lea} (\%eax, \%eax, 2), \%eax \]

\[ \text{call } \%eax \]

\[ \text{call } $2, \%eax \]
Arithmetic operations

2-operand:

```
addq src, dest  
sub           
imul          
sal <- 0      
sar s ->      
shr o ->     
and           
or
```

left shift

```
dest = dest << src
```

right shift

```
dest = dest >> src
```

1-operand

```
inc dest     
dec dest     
neg dest     
not dest
```

```
d = d + 1

```

```
d = d - 1

```

```
d = -dest (arith)

```

```
d = ~dest (bitwise)

```

Condition codes: set by arithmetic instructions.

Low Boolean values

```
CF Carry flag
ZF Zero flag
SF Sign flag
OF Overflow flag (for signed integers)
```
\[ t = a + b \]

sets CF if carry out
ZF if \( t = 0 \)
SF if \( t < 0 \)
OF if 2's compl. overflow

\[(a > 0 \land b > 0 \land t < 0) \lor \]
\[(a < 0 \land b < 0 \land t > 0)\]

Other ways to set flags

\texttt{cmp } b, a : a - b \texttt{ discard result set flags.}

\[ \texttt{test } b, a : a \& b \texttt{ discard result set flags.} \]

\texttt{sete}
\[ \texttt{ne} \rightarrow \texttt{ZF} \]
\[ \texttt{ls} \rightarrow \texttt{SF} \]
\[ \texttt{ns} \rightarrow \texttt{SF} \]
\[ \texttt{ge} \rightarrow \texttt{signed arithmetic} \]
\[ \texttt{gt} \rightarrow \texttt{SF, OF, ZF} \]
\[ \texttt{le} \rightarrow \texttt{unsigned arithmetic} \]
\[ \texttt{a} \rightarrow \texttt{above / below} \]
Use of flags:
- Jump instructions
  - jmp
  - je
  - js
  - gs
  - ga
  - unconditional

Conditional move
- cmov
- src, dest

In loops
- do-while
- for
- while

Procedures
- Recursion → individual copies of local data for each instance
  ⇒ separate data area (frames)
  ⇒ frames are on a stack.

![](A>B>C⇒A)
What is in a frame?

1) extra parameters, last first
2) return address

\[ 	ext{fop} \Rightarrow \text{local vars} \Rightarrow \text{ret. address} \]

\( \text{fop} \) : points to most recent stack elt.

\[ \text{params} \Rightarrow \text{ret. addr.} \]

\[ \text{call} \Rightarrow \text{temp. calc. results} \Rightarrow \text{saved registers} \]

\( \text{r} \Rightarrow \text{rbp} \Rightarrow \text{par 19} \Rightarrow \text{par 20} \Rightarrow \text{bottom} \]

\[ \text{A} \]

\[ \text{C} \]

\[ \text{push} : \text{place new element on stack} \]
\[ \text{pop} : \text{take element from stack} \]

Timeline:

\[ \text{call} \Rightarrow \text{A} \Rightarrow \text{B} \Rightarrow \text{return} \]

“linkage”
volatile registers: B may use freely
non-volatile registers: B may not use freely,
A may assume no change

A's call linkage:
push volatile registers that it needs.

B's call linkage:
push non-volatile registers that it needs.

B's return linkage:
pop those non-volatile registers.

A's return linkage:
pop those volatile registers.

which one?
special: \$orip, \$orsp
non-volatile: \$orb, \$orsi, \$ordi
volatile: \$orbx, \$orbp, \$or12, \ldots \$or15
all the rest

when are there local variables?
local array
local variable, especially if pointed to
& or a struct.
Data types

Integer

- Signed vs. unsigned: mostly no difference.
- Instructions depend on size

<table>
<thead>
<tr>
<th></th>
<th>Intel</th>
<th>C</th>
<th>assembler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>byte</td>
<td>char</td>
<td>b</td>
</tr>
<tr>
<td>2B</td>
<td>word</td>
<td>short</td>
<td>w</td>
</tr>
<tr>
<td>4B</td>
<td>doubleword</td>
<td>int</td>
<td>l</td>
</tr>
<tr>
<td>8B</td>
<td>quadword</td>
<td>long</td>
<td></td>
</tr>
</tbody>
</table>

Floating point

- In reg, in memory

<table>
<thead>
<tr>
<th></th>
<th>Intel</th>
<th>C</th>
<th>assembler</th>
</tr>
</thead>
<tbody>
<tr>
<td>4B</td>
<td>float</td>
<td>float</td>
<td>s</td>
</tr>
<tr>
<td>8B</td>
<td>double</td>
<td>double</td>
<td>l</td>
</tr>
<tr>
<td>16B</td>
<td>extended</td>
<td>long double</td>
<td>t</td>
</tr>
</tbody>
</table>

Arrays

- In memory

Declaration: `int A[1];`

<table>
<thead>
<tr>
<th>T</th>
<th>name</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>str[12];</td>
<td>12B</td>
</tr>
<tr>
<td>int</td>
<td>val[15];</td>
<td>20B QK/16</td>
</tr>
<tr>
<td>double</td>
<td>a[23];</td>
<td>24B</td>
</tr>
<tr>
<td>char</td>
<td>x-p[23];</td>
<td>24B</td>
</tr>
</tbody>
</table>

Lengths:

- 12B
- 20B QK/16
- 24B
- 24B
\texttt{int \ val [5] = \{1, 2, 3, 4, 5\};}

<table>
<thead>
<tr>
<th>expression</th>
<th>type</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{val[0]}</td>
<td>\texttt{int}</td>
<td>\texttt{1}</td>
</tr>
<tr>
<td>\texttt{val}</td>
<td>\texttt{int*}</td>
<td>\texttt{x = where \ val \ begins}</td>
</tr>
<tr>
<td>\texttt{val + 1}</td>
<td>\texttt{int*}</td>
<td>\texttt{x + 4}</td>
</tr>
<tr>
<td>\texttt{&amp;val[2]}</td>
<td>\texttt{int*}</td>
<td>\texttt{x + 8}</td>
</tr>
<tr>
<td>\texttt{val[4]}</td>
<td>\texttt{int}</td>
<td>\texttt{?}</td>
</tr>
<tr>
<td>\texttt{&amp;val[5]}</td>
<td>\texttt{int*}</td>
<td>\texttt{x + 20}</td>
</tr>
<tr>
<td>\texttt{* (val + 1)}</td>
<td>\texttt{int}</td>
<td>\texttt{2}</td>
</tr>
<tr>
<td>\texttt{val + i}</td>
<td>\texttt{int*}</td>
<td>\texttt{x + 4 \cdot i}</td>
</tr>
</tbody>
</table>

\#define \texttt{ZLEN} 5

typedef \texttt{int} \texttt{myArrayType [ZLEN];}

\texttt{myArrayType cmu = \{21, 5, 2, 1, 3\};}
\texttt{mit = \{20, 2, 1, 3, 9\};}
\texttt{uky = \{9, 4, 7, 2, 0\};}

\texttt{V} \hspace{2cm} 2013 \hspace{2cm} 2013 \hspace{2cm} 2013
\begin{array}{l|lll}
\hline
\texttt{cmu} & \texttt{mit} & \texttt{uky} \\
\hline
\end{array}
uky is in \( \text{mov} \ ax:ax0000h, 90rdh, 4 \), \( rdx \)
\[ \text{return} \] \( \text{uky} \) \[ \text{mov} \ ox:0 (9orsp, 9ordh, 4), 90eax \]
\[ \text{return} \] \( \text{uky} \) \[ \text{mov} \ ox:20 (9orsp, 9ordh, 4), 90eax \]

```
-27c

utt
```

nested arrays

my ArrayType pgh[4] =

1.5.f, 2.0, 67,
.27, 11, 82 \[73, pgh[1], 37 \]
```

\[152057\]

row-major order

\[\text{int} \ pgh[4][5]\]
\[
\text{return } \text{pgh} [\text{index}] \\
\downarrow \\
p_{\text{ax}} \\
\text{lea } (p_{\text{ax}}, p_{\text{ax}}, 4), p_{\text{ax}} \quad \# a = \text{index} \\
\text{lea pgh } (p_{\text{ax}}, 4), p_{\text{ax}} \quad \# a = 3_{\text{0 index}} + \text{pg}{\text{h}}
\]

\[
T \quad a_{\text{my Array}[R][C]} \\
\text{where is } a_{\text{my Array}[i][j]}? \\
\text{myArray } + \text{TiKi } + \text{Tj} \\
\quad + \text{TIC } + \text{Tj } = \text{T (iC } + \text{j)} \\
\]

\[\text{to zero an array} \]
\[\text{bzero (ptr to array, number of bytes)}\]

\[
\text{structs} \\
\text{struct rec } \exists \\
\quad \text{int y [8];} \\
\quad \text{int i;} \\
\quad \text{struct rec } \ast \text{n;} \\
\]
code | address | type
---|---|---
foo | x | struct rec
foo.y | x | int *
foo.i | x+12 | int
foo.m | x+16 | struct rec *
foo.y[1] | x+4 | int

linked lists are built from structs.

```
struct s1 {
    char c;
    int y[2];
    double x;
    char d;
} *p;
```

<table>
<thead>
<tr>
<th>size(8)</th>
<th>offset</th>
<th>pad</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>3B</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4B</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>1</td>
<td>24</td>
<td>7B</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
</tr>
</tbody>
</table>

data must be aligned (on most architectures)
if data uses \( n \) bytes, align (usually) to
an address divisible by \( n \).

\( n=4 \): divisible by 4: address ends \( 00 \)
[8 bytes data on our machine: div. by 8]
[16 B div. by 8]
to align: add padding as necessary between fields.
entire struct must be aligned to the requirement
of its largest field. (for arrays of structs)
Unions

like a struct, but
all fields start at offset 0.
purpose:
1) save space
2) reinterpret data.
3) detect byte ordering.
example in notes p. 32.

Buffer over-flow

buffer is an array used to store data.
usually for I/O.
problem: library functions don't all check
for buffer-size limits.

gets() → fgets
strcpy() → strncpy
strcat() → strncat
scanf() : scanf("...\000...", ~
fscanf()
sscanf()

Internet worm: 1988
Hundreds of buffer overflow examples.
Mitigating vulnerability
- overallocated arrays
- better library routines
- randomized stack offsets
- non-executable segments (stack especially)
  - canaries

randomize text segment locations
(Kaiser)

Malware
- worm: self-duplicating, stand-alone program
- virus: "" attached to a program

Memory layout in Linux in virtual space

* : a.out file

libraries:
- text (read, execute)
- pad (no access)
- constants (read)
- data (bss) real, write
Linking

To combine the results of 1 or more compilations, along with libraries.

```
#include "proj.h"
```

Benefits

decompose work
experts can program commonly used functions.
changes to one file do not require recompiling others.
linker can select only necessary parts of library files.

source file compiles to linksto

```
proj.h

a.c
b.c
d.c

a.o
b.o
d.o

proj

library

1

relocatable object files

object file

library

```
Symbols in C programs

void swap();

- Exported global identifier

extern int myGlobal;

- Imported global identifier

int myGlobal

- Local, exported

static int myLocal;

- Local, not exported

swap(&myGlobal, &myLocal);

Method

Compiler uses a data structure to keep track of all identifiers.
Symbol Table:
- Type, location, global/local flag

Compiler outputs global symbols to object file.
- All imported globals are marked "unresolved"

Linker resolves unresolved references by connecting them across object files.
- Compiler complains if still unresolved.
- Multiply resolved.
linker refers to libraries to finish resolution. Result is executable if all identifiers resolve. It is a standard name for output: a.out

Shared object file
name ends in .so
libraries
shared at run time by multiple processes
Windows calls it a DLL
relocation