Representing Information (2)

Outline

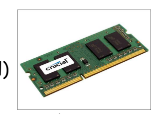
- Byte ordering
- Representing Strings and Code
- Encodings
  - Unsigned and two's complement
- Suggested reading
  - Chap 2.1.3, 2.1.4, 2.1.5, 2.2

Virtual Memory

- The memory introduced in previous slides
  - is only an conceptual object and
  - does not exist actually
- It provides the program with what appears to be a monolithic byte array
- It is a conceptual image presented to the machine-level program

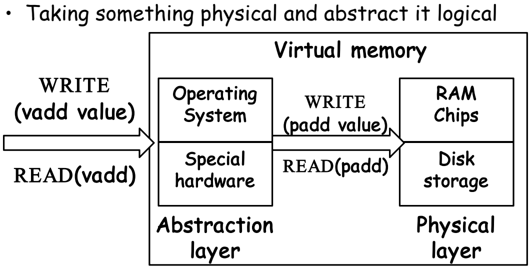
Virtual Memory

- The actual implementation uses a combination of
  - Hardware
  - Software
- Hardware
  - random-access memory (RAM) (physical)
  - disk storage (physical)
  - special hardware (performing the abstraction)
- Software
  - and operating system software (abstraction)



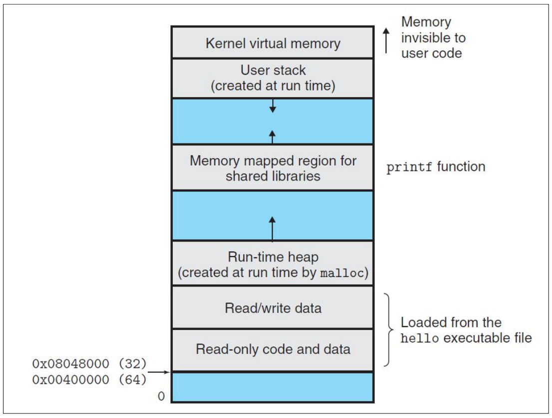
Way to the Abstraction

- Taking something physical and abstract it logical



Subdivide Virtual Memory into More Manageable Units

- One task of
  - a compiler and
  - the run-time system
- To store the different *program objects*
  - Program data
  - Instructions
  - Control information



Byte Ordering

- How should a large object be stored in memory?
- For program objects that span multiple bytes
  - What will be the address of the object?
  - How will we order the bytes in memory?
- A multi-byte object is stored as
  - a contiguous sequence of bytes
  - with the address of the object given by the smallest address of the bytes used

Byte Ordering

- Little Endian
  - Least significant byte has lowest address
  - Intel
- Big Endian
  - Least significant byte has highest address
  - Sun, IBM
- Bi-Endian
  - Machines can be configured to operate as either little- or big-endian
  - Many recent microprocessors

Big Endian (0x1234567)

		0x100	0x101	0x102	0x103		
		01	23	45	67		

Little Endian (0x1234567)

		0x100	0x101	0x102	0x103		
		67	45	23	01		

How to Access an Object

- The actual machine-level program generated by C compiler
  - simply treats each program object as a block of bytes
- The value of a pointer in C
  - is the virtual address of the first byte of the above block of storage

How to Access an Object

- The C compiler
  - Associates *type* information with each pointer
  - Generates different machine-level code to access the pointed value
    - stored at the location designated by the pointer depending on the *type* of that value
- The actual machine-level program generated by C compiler
  - has no information about data *types*

Code to Print Byte Representation

```
typedef unsigned char *byte_pointer;

void show_bytes(byte_pointer start, int len)
{
   int i;
  for (i = 0; i < len; i++)
   printf("0x%p\t0x%.2x\n",
              start+i, start[i]);
   printf("\n");
}
```


Code to Print Byte Representation

```
void show_int(int x) {
   show_bytes((byte_pointer) &x, sizeof(int));
}

void show_float(float x) {
   show_bytes((byte_pointer) &x, sizeof(float));
}

void show_pointer(void *x) {
   show_bytes((byte_pointer) &x, sizeof(void *));
}
```


Features in C

- typedef
  - Giving a name of type
  - Syntax is exactly like that of declaring a variable
- printf
  - Format string: %d, %c, %x, %f, %p
- sizeof
  - sizeof(T) returns the number of bytes required to store an object of type T
  - One step toward writing code that is portable across different machine types

Features in C

- Pointers and arrays
  - start is declared as a pointer
  - It is referenced as an array start[i]
- Pointer creation and dereferencing
  - Address of operator &
  - &x
- Type casting
  - (byte_pointer) &x

Code to Print Byte Representation

```
void test_show_bytes(int val) {
   int ival = val;
   float fval = (float) ival;
   int *pval = &ival;
   show_int(ival);
   show_float(fval);
   show_pointer(pval);
}
```

Example

- Linux 32: Intel IA32 processor running Linux
- Windows: Intel IA32 processor running Windows
- Sun: Sun Microsystems SPARC processor running Solaris
- Linux 64: Intel x86-64 processor running Linux

- With argument 12345 which is 0x3039

Machine	Value	Type	Bytes (hex)
Linux 32	12,345	`int`	39 30 00 00
Windows	12,345	`int`	39 30 00 00
Sun	12,345	`int`	00 00 30 39
Linux 64	12,345	`int`	39 30 00 00
Linux 32	12,345.0	`float`	00 e4 40 46
Windows	12,345.0	`float`	00 e4 40 46
Sun	12,345.0	`float`	46 40 e4 00
Linux 64	12,345.0	`float`	00 e4 40 46
Linux 32	`&ival`	`int *`	e4 f9 ff bf
Windows	`&ival`	`int *`	b4 cc 22 00
Sun	`&ival`	`int *`	ef ff fa 0c
Linux 64	`&ival`	`int *`	b8 11 e5 ff ff 7f 00 00

Representing Codes

```
int sum(int x, int y) {
    return x + y;
}
```

Linux 32: 55 89 e5 8b 45 0c 03 45 08 c9 c3
Windows: 55 89 e5 8b 45 0c 03 45 08 5d c3
Sun: 81 c3 e0 08 90 02 00 09
Linux 64: 55 48 89 e5 89 7d fc 89 75 f8 03 45 fc c9 c3

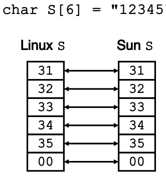
Byte Ordering Becomes Visible

- Circumvent the normal type system
  - Casting
  - Reference an object according to a different data type from which it was created
  - Strongly discouraged for most application programming
  - Quite useful and even necessary for system-level programming
- Disassembler
  - 80483bd: 01 05 64 94 04 08->add %eax, 0x8049464
- Communicate between different machines

Representing Strings

- Strings in C
  - Represented by array of characters
  - Each character encoded in ASCII format
  - String should be null-terminated
    Final character = 0
  - \a \b \f \n \r \t \v
  - \\ \? \' \" \000 \xhh

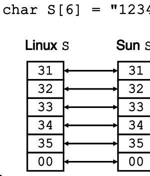
`char S[6] = "12345";`



Representing Strings

- Compatibility
  - Byte ordering not an issue
    Data are single byte quantities
  - Text files generally platform independent
    Except for different conventions of line termination character!

`char S[6] = "12345";`



Representing Strings

```
/* strlen: return length of string s */
int strlen(char *s)
{
  char *p = s ;

  while (*p != '\0')
    p++ ;
  return p-s ;
}
<string.h>
```


Representing Strings

```
/* trim: remove trailing blanks, tabs, newlines */
int trim(char s[])
{
  int n;

  for (n = strlen(s)-1; n >= 0; n--)
     if ( s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
           break;
  s[n+1] = '\0';
  return n
}
```


Unsigned Representation

- Binary (physical)
  - Bit vector $[x_{w-1}, x_{w-2}, x_{w-3}, \ldots x_0]$
- Binary to Unsigned (logical)

$$B2U(X) \quad = \quad \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's Complement

- Binary (physical)
  - Bit vector $[x_{w-1}, x_{w-2}, x_{w-3}, \ldots x_0]$
- Binary to Signed (logical)

$$B2T(X) \quad = \quad -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Sign Bit

- 2's complement

From Two's Complement to Binary

- If nonnegative
  - Nothing changes
- If negative, its binary representation is
  - $1x_{w-2} \cdots x_1 x_0$
  - Its value is x
  - Assume its absolute value is y = -x
- The binary representation of y is
  - $0y_{w-2} \cdots y_1 y_0$

Two's Complement

- Binary (physical)
  - Bit vector $[x_{w-1}, x_{w-2}, x_{w-3}, \ldots x_0]$
- Binary to Signed (logical)

$$B2T(X) \quad = \quad -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Sign Bit

- 2's complement

From Two's Complement to Binary

$$x = -2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i = -y = -\sum_{i=0}^{w-2} y_i 2^i$$

$$\sum_{i=0}^{w-2} x_i 2^i = 2^{w-1} - \sum_{i=0}^{w-2} y_i 2^i = \sum_{i=0}^{w-2} (1-y_i) 2^i + 1$$

$$2^{w-1} = \sum_{i=0}^{w-2} 2^i + 1 \qquad x_{w-1} = 1 \qquad y_{w-1} = 0$$

31

Two's Complement

$$\sum_{i=0}^{w-1} x_i 2^i = \sum_{i=0}^{w-1} (1-y_i) 2^i + 1$$

- What does it mean?
  - Computing the negation of x into binary with w-bits
  - Complementing the result
  - Adding 1

32

From a Number to Two's Complement

- -5
  - 5
  - 0101 (binary for 5)
  - 1010 (after complement)
  - 1011 (add 1)

33

Two's Complement Encoding Examples

Binary/Hexadecimal Representation for -12345

Binary:	0011 0000 0011 1001 (12345)				
Hex:	3	0	3	9	
Binary:	1100 1111 1100 0110 (after complement)				
Hex:	C	F	C	6	
Binary:	1100 1111 1100 0111 (add 1)				
Hex:	C	F	C	7	

34

Numeric Range

- Unsigned Values
  - Umin=0
  - Umax=2^w-1
- Two's Complement Values
  - Tmin = -2^{w-1}
  - Tmax = 2^{w-1}-1

35

Numeric Range

- Relationship
  - | *TMin*| = *TMax* + 1
  - *Umax* = 2* *TMax* + 1
  - -1 has the same bit representation as *Umax*,
    - a string of all 1s
  - Numeric value 0 is represented as
    - a string of all 0s in both representations

36

Alternative representations of signed numbers

- Ones' Complement:
  - The most significant bit has weight $2^{w-1}-1$
- Sign-Magnitude
  - The most significant bit is a sign bit
    - that determines whether the remaining bits should be given negative or positive weight

X	B2U(X)	B2T(X)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

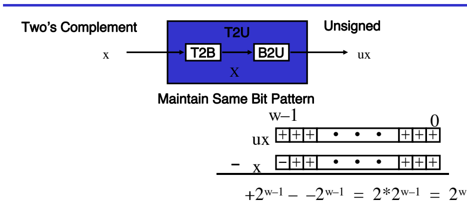
Mappings between logical and physical

- Uniqueness
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding

Mappings between logical and physical

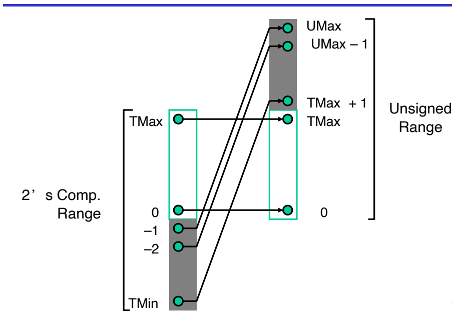
- Invert Mappings (from logical to physical)
  - $U2B(x) = B2U^{-1}(x)$
    - Bit pattern for unsigned integer
  - $T2B(x) = B2T^{-1}(x)$
    - Bit pattern for two's comp integer

Relation Between 2's Comp. & Unsigned



$$ux = \begin{cases} x & x \geq 0 \\ x + 2^w & x < 0 \end{cases}$$

Conversion between two Representations



Casting among Integral Data Type

43

Outline

- Conversions
  - Signed vs. unsigned
  - Long vs. short
- Suggested reading
  - Chap 2.2

44

Integral data type in C

- Signed type (for integer numbers)
  - char, short [int], int, long [int]
- Unsigned type (for nonnegative numbers)
  - unsigned char, unsigned short [int], unsigned [int], unsigned long [int]
- Java has no unsigned data type
  - Using byte to replace the char

45

Typical Ranges for C integral data types 32

C declaration	Typical 32-bit	
	minimum	maximum
char unsigned char	-128 0	127 255
short [int] unsigned short	-32,768 0	32,767 65,535
int unsigned [int]	-2,147,483,648 0	2,147,483,647 4,294,967,295
long [int] unsigned long	-2,147,483,648 0	2,147,483,647 4,294,967,295
int32_t uint32_t	-2,147,483,648 0	2,147,483,647 4,294,967,295
int64_t uint64_t	-9,223,372,036,854,775,800 0	9,223,372,036,854,775,800 18,446,744,073,709,551,615

Typical Ranges for C integral data types 64

C declaration	Typical 32-bit	
	minimum	maximum
char unsigned char	-128 0	127 255
short [int] unsigned short	-32,768 0	32,767 65,535
int unsigned [int]	-2,147,483,648 0	2,147,483,647 4,294,967,295
long [int] unsigned long	-9,223,372,036,854,775,800 0	9,223,372,036,854,775,800 18,446,744,073,709,551,615
int32_t uint32_t	-2,147,483,648 0	2,147,483,647 4,294,967,295
int64_t uint64_t	-9,223,372,036,854,775,800 0	9,223,372,036,854,775,800 18,446,744,073,709,551,615

Guaranteed Ranges for C integral data types

C declaration	Typical 32-bit	
	minimum	maximum
char unsigned char	-127 0	127 255
short [int] unsigned short	-32,767 0	32,767 65,535
int unsigned [int]	-32,767 0	32,767 65,535
long [int] unsigned long	-2,147,483,648 0	2,147,483,647 4,294,967,295
int32_t uint32_t	-2,147,483,648 0	2,147,483,647 4,294,967,295
int64_t uint64_t	-9,223,372,036,854,775,800 0	9,223,372,036,854,775,800 18,446,744,073,709,551,615

Casting among Signed and Unsigned in C

- C Allows a variable of one type to be interpreted as other data type
  - Type conversion (implicitly)
  - Type casting (explicitly)

Signed vs. Unsigned in C

- Casting
  - Explicit casting between signed & unsigned
  - same as U2T and T2U
    - int tx, ty;
    - unsigned ux, uy;
    - tx = (int) ux;
    - uy = (unsigned) ty;

Signed vs. Unsigned in C

- Conversion
  - Implicit casting also occurs via assignments and procedure calls
    - int tx, ty;
    - unsigned ux, uy;
    - tx = ux;
    - uy = ty;

Casting from Signed to Unsigned

```
short int          x =  12345;
unsigned short int ux = (unsigned short) x;
short int          y  = -12345;
unsigned short int uy = (unsigned short) y;
```

- Resulting Value
  - No change in bit representation
  - Nonnegative values unchanged
    - *ux* = 12345
  - Negative values change into (large) positive values
    - *uy* = 53191

Weight	12,345		−12,345		53,191	
	Bit	Value	Bit	Value	Bit	Value
1	1	1	1	1	1	1
2	0	0	1	2	1	2
4	0	0	1	4	1	4
8	1	8	0	0	0	0
16	1	16	0	0	0	0
32	1	32	0	0	0	0
64	0	0	1	64	1	64
128	0	0	1	128	1	128
256	0	0	1	256	1	256
512	0	0	1	512	1	512
1,024	0	0	1	1,024	1	1,024
2,048	0	0	1	2,048	1	2,048
4,096	1	4096	0	0	0	0
8,192	1	8192	0	0	0	0
16,384	0	0	1	16,384	1	16,384
±32,768	0	0	1	−32,768	1	32,768
Total		12,345		−12,345		53,191

back

Unsigned Constants in C

- By default
  - constants are considered to be signed integers
- Unsigned if have "U" as suffix
  - 0U, 4294967259U

Casting Convention

- Expression Evaluation
  - If mix unsigned and signed in single expression
    - signed values implicitly cast to unsigned
  - Including comparison operations <, >, ==, <=, >=
  - Examples for *W* = 32

55

Casting Convention

Constant1	Constant2	Relation	Evaluation
0	0U	==	unsigned
-1	0	<	signed
-1	0U	>	unsigned
2147483647	-2147483648	>	signed
2147483647U	-2147483648	<	unsigned
-1	-2	>	signed
(unsigned)-1	-2	>	unsigned

56

From short to long

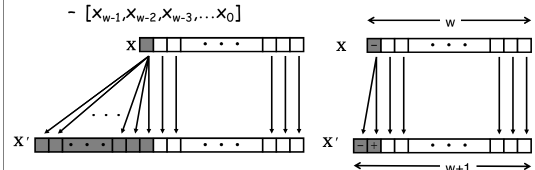
```
short int x =  12345;
int ix = (int) x;
short int y = -12345;
int iy = (int) y;
```

- We need to expand the data size
- Casting among unsigned types is normal
- Casing among signed types is trick

57

Expanding the Bit Representation

- Zero extension
  - Add leading 0s to the representation
- Sign extension
  - $[x_{w-1}, x_{w-2}, x_{w-3}, \ldots x_0]$



58

From short to long

```
short int sx = 12345; : 0x0309
int x = (int) sx;      : 0x00000309
short int sy =-12345;: 0xcfc7
int y = (int) sy;      : 0xffffcfc7
usigned ux = sy;       : 0xffffcfc7
// (usigned)(int)sy
```

59

From short to long

```
int fun1(unsigned word) {
     return (int) ((word << 24) >> 24);
}
int fun2(unsigned word) {
     return ((int) word << 24) >> 24;
}
```

w	fun1(w)	fun2(w)
0x00000076	00000076	00000076
0x87654321	00000021	00000021
0x000000C9	000000C9	FFFFFFC9
0xEDCBA987	00000087	FFFFFF87

Describe in words the useful computation each of these functions performs.

60

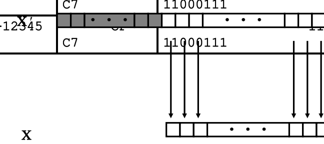
From long to short

```
int       x  = 53191;
short int sx = x;
int       y  = -12345;
Short int sy = y;
```

- We need to truncate the data size
- *Casing from long to short is trick*

Truncating Numbers

	Decimal	Hex	Binary
x	53191	00 00 CF C7	00000000 00000000 11001111 11000111
sx	-12345	CF C7	11001111 11000111
y	-12345	FF FF CF C7	11111111 11111111 11001111 11000111
sy	-12345	CF C7	11001111 11000111



Truncating Numbers

- Unsigned Truncating

$$B2U_w([x_w, x_{w-1}, \cdots, x_0]) mod 2^k = B2U_k([x_k, x_{k-1}, \cdots, x_0])$$

- Signed Truncating

$$B2T_k([x_k, x_{k-1}, \cdots, x_0]) = B2T_k(B2U_w([x_w, x_{w-1}, \cdots, x_0]) mod 2^k)$$

Advice on Signed vs. Unsigned

Non-intuitive Features

```
float sum_elements ( float a[], unsigned length ) {
   int i ;
   float result = 0;
   for ( i = 0; i <= length – 1; i++)
        result += a[i] ;
   return result;
}
```


Advice on Signed vs. Unsigned

```
/* Prototype for library function strlen */
size_t strlen(const char *s); /*size_t is usigned */

/* Determine whether string s is longer than string t */
/* WARNING: This function is buggy */
int strlonger(char *s, char *t) {
   return strlen(s) - strlen(t) > 0;
}
```


Advice on Signed vs. Unsigned

```
/*
 * Illustration of code vulnerability similar to that found in
 * FreeBSD's implementation of getpeername()
 */

/* Declaration of library function memcpy */
void *memcpy(void *dest, void *src, size_t n);

```

Advice on Signed vs. Unsigned

```
/* Kernel memory region holding user-accessible data  */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer  */
int copy_from_kernel(void  *user_dest, int maxlen) {
      /* Byte count len is minimum of buffer size and maxlen  */
      int len = KSIZE < maxlen ? KSIZE : maxlen;
      memcpy(user_dest, kbuf,  len);
      return len;
}
```

Invoking copy_from_kernel() with a negative maxlen

Advice on Signed vs. Unsigned

- Usigned values are very useful
  - Collections of bits
    - Bit vectors
    - Masks
  - Addresses
  - Multiprecision Arithmetic
    - Numbers are represented by arrays of words