CS 24000 - Programming In C

Week 16: Review

Zhiyuan Li
Department of Computer Science
Purdue University, USA

This has been quite a journey

- Congratulation to all students who have worked hard in honest ways!
- After this course, continue to read and write many programs
  - Participate in projects with professors
  - Participate in projects that benefit community
  - Participate in innovations and commercializations

What have we learned in this course?

- We not only studied the syntax and C programming language, but more importantly,
- We studied the semantics of the C language, through which
- We learned several fundamental concepts in the program execution model for imperative programming languages

The Concept of Address Space

- Processes are execution instances of programs
- Each process has its own (virtual) address space
- The range of the address space is determined by the length of memory address supported by the machine model
  - E.g. 32 bit vs. 64 bit

Physical Memory

- Not every virtual address is mapped to a physical memory location
- The operating system allocates physical memory to each process
  - From part of the secondary storage (called swap device)
- It is prohibitively expensive, and therefore impractical to map the entire virtual address space to physical memory
  - We learned how to use some commands to examine the memory allocated to the process

- Both virtual address space and physical address space are partitioned into sections by functionality
  - Some are read only
  - Some are executable
  - Trying to access sections that violate the protection status will cause run time exceptions
- Each section is partitioned into pages to allow efficient memory management

- Efficient memory operations are made possible by the memory hierarchy
  - Faster memory devices are used as “caches” for the slower memory devices
  - E.g. The DRAM main memory is used as a “cache” for the swap device
  - There are faster memories, e.g. SRAM, that are used as “caches” for the main memory

- Processes have separate address spaces
- The contents in the parent's address space are physically shared by the child until a page is written by either process (copy-on-write)
  - This has a significant impact on evaluating the cost of forking a process
- If we want different processes to write to the same physical page, or some write to and some read from the same physical page
  - We need to allocate shared memory (in the unit of pages)

- The same shared physical memory page may be attached to different virtual addresses in different processes.
- The addresses can be assigned by the operating system

- Concurrent writes (or mixed writes and reads) by different processes require synchronization
- We have learned the concept of semaphores for synchronization
- The physical implementation of semaphores require placing them in the shared memory
- Hence, each semaphore (as a structure) has its address

The Concept of Pointers and Addresses

- In C and C++, the concept of pointers and addresses are tightly related.
- Pointer arithmetic allows C programs to step through the address space in a highly flexible way
- This makes C language a convenient one for low level system implementation
- However, for large software engineering projects that do not involve so much machine level details, pointer arithmetic is full of potential danger of misuse and prone to errors

- We have learned the relationship between different ways to address memory locations
  - Pointer increment/decrement
    - In strides that depend on the type of the data (primitive or structured) that are pointed to
  - Base of arrays
    - Dimensions of arrays
  - Indexing arrays
  - The & operation to take the address of anything that has an "l-value"

L-value

- An expression is said to have an l-value if it is allocated a memory address
  - We have learned a large variety of expressions that have l-values, e.g.
    - Array elements
    - Fields of struct
    - Simple scalars (of various arithmetic types or pointers)
      - Pointer variables can have their own addresses to
    - Let p be a scalar variable, &p is its address
    - &p itself will not have an l-value, because we do not know (or have) a unique memory address that stores &p
    - Hence & (&p) will generate a compiler error

- Let p be a pointer variable
- Does the expression p++ have l-value?
  - No. p++ has a r-value that is the current value of p.
  - It also has a side effect that is to increment p by a stride dependent on the type of the data structure p points to
  - But p++ has no l-value, just like p+1 has no l-value

The pointer dereference operation

- The * operation (as pointer dereference) can be applied to any pointer expression
- The result of pointer arithmetic (p + offset) is a pointer expression
- Many pointer expressions do not have l-values
- But * (e), for valid e, always has an l-value
  - Because it refers to a memory location!
- Therefore & (* (e)) is perfectly legal.
  - E.g. q = & (* (p + 1))

Segmentation fault

- Just because we have a pointer that is declared to point to a certain type of data does not mean the memory has been allocated to the (potential) data that it can point to
- Forgetting this, we will likely have segmentation fault
- A simple example

```
int *a;
*a = 0;
```

- → Segmentation fault
- Also, `int * a =0;` initialize `a` to null instead of initializing `*a` to 0.

Organization of the virtual address space

- We have learned the common partitioning of the virtual address space
  - Text (i.e. the code region)
  - Static (to store constants, including strings, and static variables)
  - Stack (to store function locals, including formal parameters and compiler-generated locals, and to save registers)
  - The stack frames can be viewed as a linked list
    - When debugging, we can follow such a linked list to examine the value changes through a call chain
  - Heap (malloc(), free())

Address of a function

- The instructions generated for a function are stored in the text region of the address space
  - They have addresses
  - When a function is called, the control flow transits to the beginning address of the called function
- Which function to call by function invocation can be decided as late as run time
  - This is done by passing the address of one function A (i.e. the function pointer) to another function B
- Unlike in OO languages, C does not have a way to check data types at run time,
  - which restricts the formal parameters in the use of function pointers
  - The types and number of the parameters must be uniform among all functions that might be A passed to B

- Program statements involving function pointers are often difficult to understand
  - The execution flow must be analyzed carefully

```
int cmp(int val, int* limit) { return val > *limit; }

int bar( int (*f) (int, int*), int* base, int c) {
   if (f(c, base)) {
         return bar(f, base, c-1) +
                  bar(f, base, c-2);
   }
   return 1;
}

int main(){
  int std = 2;
  int x = bar(cmp, &std, 5);
  printf("%d", x);
  return 0;
}
```

- In computer organization and compiler courses, we will learn that referencing consecutive addresses is much more efficient than referencing addresses in long strides
- Because referencing consecutive addresses exploits spatial data locality
  - Processor hardware fetch data from slower memory to faster memory in a unit that contains a multiple of consecutive “words”
    - From memory to register in the unit of a single word
    - From main memory to caches in the unit of multiple words
  - The memory is “byte-addressable”, but memory allocation is word-aligned

Data padding

- That is why, by default, structures are padded so that they are word aligned.
- Aggressive, but not yet commonly adopted, compiler techniques would reorganize the data structure to better exploit spatial locality
  - A sequence of memory operations may be applied to the same field of different structures
  - Hence, we are visiting nonconsecutive memory addresses, spoiling spatial locality

- In the next (and final) lecture, we continue to review important concepts and their relationships
- To address some students' concern that multiple choice questions may be prone to typos, the final exam will take the format of "short answers"
- See next example.

Example of a “final exam” question

- What is the 2’s complement representation of decimal number -14?

So, in the final exam, we will not use the format shown next:

Quiz 12 #1

- What is 2's complement representation (with 16 bits) for decimal number -14?
- (a) 1000 0000 0000 1110
- (b) 0000 0000 0000 1110
- (c) 0000 0000 0001 0100
- (d) 1111 1111 1111 0010

- Answer (d)

Quiz 12 #2

- What does the following program print? (assume 32 bit machine)
    - #include <stdio.h>
    - int main() {
    - int x;
    - char a = 0xaa, b = 0x11;
    - a = b+a;
    - x = a;
    - printf ("%x", x);
    - }
- (a) 00000067 (b) 000000bb (c) ffffffbb (d) 100000bb

- Answer (c)

Quiz 12 #3

- What is the result of the following operation?

  1 & 3 || 2

(a) It will have compiler error, because &3 is illegal operation

(b) 1

(c) 11, because 1 & 3 is "01 & 11" which equals 01 in binary, and "01 || 10" is 11 in binary, decimal 2 is binary 10

(d) The value is undefined

- Answer (b) Boolean expression will evaluate to 0 or 1. The || result is 1 (true) if either operand is nonzero