# ISA 563: Fundamentals of Systems Programming

Pointers and Memory Management

Feb. 5, 2013

# Overview: System Memory

- Memory stores many kinds of data
  - Process sections
  - Kernel/userspace
  - Dynamic memory management
- Memory is managed jointly:
  - The operating system's virtual memory system
  - The C library memory management code
- **Pointers** allow explicit manipulation of variable addresses

# The Process Address Space

- A process represents a program in execution
- Processes have an address space: a way of labeling all the different kind of memory, data, and variables used by the program or the OS to manage the program

# Reasons to Use Pointers

- Copy-by-value is expensive for large data types
- Dynamic memory allocation
- Data structure link management
- A form of polymorphism via ***function pointers***

# What is a Pointer?

- Essentially an address
- Variables
  - Name
  - Address
  - Type
  - Value
- Function identifiers are essentially addresses

# Example: Simple Pointer

```
int score = 100;
int *score_ptr;

score_ptr = &score;

fprintf(stdout,
        "addressof(score) = %p\n",
        score_ptr);
```

# Properties of 'score'

```
int score = 100;
int *score_ptr = &score;
```

```
score
&score
sizeof(score)
score_ptr
&score_ptr
*score_ptr
sizeof(score_ptr)
```

# Reading and Writing from/to Pointers

- `ptr = value;  // update the address held by ptr`
- `*ptr = value; // update the variable pointed to by ptr`

- `fprintf(stdout, “ptr = %p\n”, ptr);`
- `fprintf(stdout, “val = %d\n”, *ptr)`

# Pointers to Complex Data Types

```
struct node {
    double value;
    struct node *next;
};

struct node head;
head.value = 12.34;

struct node *node_ptr = &head;

// pointer access to struct fields
printf("\nhead.value = %g\n", head.value);
printf("node_ptr->value = %g\n", node_ptr->value);
```

# Demo

varptr.c

# Argument Passing

- Two ways to pass arguments to functions:
  - Call by value
    - Argument value is copied
    - Changes to argument does not affect the original
  - Call by reference
    - A reference (pointer) to the variable is passed
    - Passed variable can be changed through the pointer
    - The pointer itself, again, is passed by value

# Demo

arg_passing.c

# Argument Passing for Large Data Types

- Call by value:
  - Large data type has to be copied for the call
  - Large data type has to be copied back to caller
- Call by reference
  - A reference to large data type is passed
  - Data is modified through the reference (pointer)

# Demo

large_args.c

# Pointers and Arrays

- Strong relationship between pointers and arrays in C

```
int a[10];
int *p = &a[0];
```

  - Close correspondence between indexing and pointer arithmetic
    - a[i] == *(p+i)
    - a[i] <=> *(a+i)
    - a + i is the i-th element of a
    - p = a; // can also be used instead of p=&a[0];

# Pointers and Arrays (Cont'd)

- Although very close, there are some differences:
  - Array name is not variable, a pointer is.
    - p = a;  // legal
    - p++;  // legal
    - a = p;  // illegal
    - a++;  // illegal
  - sizeof gives the size of all the elements for array, and gives the size of the pointer for pointers
    - int a[10]; // sizeof(a) == 10 * sizeof(int)
    - p = a;  // sizeof(p) == sizeof(int *)

# Demo

var_array.c

# Demo

qsort.c

# Memory Management APIs

# malloc(3), calloc(3), realloc(3)

- malloc is a C library call that ask the C library memory magement code to allocate or apportion a section of user space memory for your process
- calloc is similar but clears this memory for you
- realloc re-sizes already-allocated chunks. (Can also do malloc, free, etc.)

# Releasing Memory

- Use free(3)
- Avoid double-free error
  - Set pointer to NULL immediately after a call to free

```
char *x = (char *) malloc(10);

if ( x == NULL ) {
   fprintf(stderr, “malloc failed\n”);
   exit(-1);
}

memset(x, 'A', 10);
free(x);
x = NULL;
```

# memset(3)

- Write a value into a chunk of memory

- Arguments
  - `void *;  -- pointer to a chunk`
  - `int;     -- char to write into the chunk`
  - `size_t;  -- number of bytes to write`

# memcpy(3) / memove(3)

- Copy one chunk to another
- Arguments:
  - `void *;  -- pointer to source chunk`
  - `void *;  -- pointer to destination chunk`
  - `size_t;  -- number of bytes to transfer`
- Use memove if you suspect src and dst overlap. Supposedly, memcpy is faster, but src and dst cannot overlap.

# strcpy(3) / strncpy(c)

- Like memcpy, but treats '\0' as end of string
- CAUTION: use strncpy instead of strcpy

- Arguments (strncpy)
  - `char *dest;  // pointer to destination`
  - `char *src;  // pointer to source`
  - `size_t n;  // number of chars to copy at MOST. If src is longer than or equal to what dest can hold, no automatic NULL terminator. If less, remaining destination NULL-filled.`

# Misc.

- bzero
- strncat, strncmp,
- strdup
- strerror
- strlen
- strstr
- strtok
- ...