CS354: Machine Organization and Programming

Lecture 4
Friday the September 11th 2015

Section 2
Instructor: Leo Arulraj

Class Announcements

Student Note Takers needed.

Students who are looking for an easy way to earn some extra money should read this email. The McBurney Center is recruiting a paid notetaker for your CS/ECE354 class. You'll receive a stipend of about $30 per credit for notes provided for the entire duration and scope of the class. No extra time outside of class is required, except for a short orientation for new notetakers. Detailed instructions will be on the Notetaker Information Form you'll get from the McBurney student as soon as you are hired.

If interested, make copies of sample notes from the last lecture and email or submit them to me as soon as possible. Make sure you include your name, phone number and email address with your sample notes. If your notes are selected, you will be contacted directly by the student who needs the notetaker.

Lecture overview

1. Arrays vs. Pointers
2. Function Pointers
3. Structs and Unions
4. malloc & free functions
5. Static vs. Dynamic Memory Allocation

Swap two variables

```
void swap(int a, int b){
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

Swap two variables using pointers

```c
void swap(int *px, int *py){
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}
```

Example C Program on swap using pointers

Arrays vs. Pointers

Arrays and Pointers are often used interchangeably

Example:

```
int ar[100];  /* an array of 100 integers */
int *arptr = ar;
arptr[4] = 10; //sets the 5th element to 10
```

Arrays vs. Pointers

- And, we could now change the value of the 7th element of the array to 1000 with

  `*(arptr+6) = 1000;`

- We can even do the same thing with

  `*(ar+6) = 1000;  /* 7th item is at offset of 6 from the element at index=0 */`

Arrays vs. Pointers

- Stated a little more formally,
  a[i] is the same as ***(a+i)**
  and **&a[i]** is the same as **a+i**

- However, a pointer **is** a variable, but an array
  name **is not** a variable. So,
  arptr = arr is legal,
  but **arr = arptr** and **arr++** **are not** legal.

- Pointer can be used in place of an array.
  Array can not be used as a pointer in all
  scenarios.

Pointers increment with sizeof(type)

```
int ar[5]={0,6,-1,15,102};
int *ap = ar;
printf("ptr ap = %0x val *ap= %d
\n",ap, *ap);
ap+=1;
printf("ptr ap = %0x val *ap= %d
\n",ap, *ap);
```

Output:

ptr ap = a81b0d60 val *ap= 0

ptr ap = a81b0d64 val *ap= 6

0	6	-1	15	102

in memory

0	ar
6
-1
15
102

$$\text{addr of } ar[i] = ar + 4*i$$

→ assumes 4-byte ints and byte addressable

→ probably assumes that ar is assigned an

Example C Program on pointer arithmetic

Multiple Indirection

- **Consider this:**

  ```
  int    a =  3;
  int   *b = &a;
  int  **c = &b;        //char** argv used in main func
  int ***d = &c;
  ```

- **The values of these pointers equate to each other like this:**

  ```
  *d ==   c;
  **d ==  *c ==  b;
  ***d == **c == *b == a == 3;
  ```

Pointers and Const

The following two declarations specify the int variable as const.

- const int *ptr_a;

- int const *ptr_a;

- So, *ptr_a = 10 is not allowed

The following one however specifies the pointer as const.

- int *const ptr_b;

- So, ptr_a ++ is not allowed

Example C Program on const pointers

Function Pointers 1

Syntax: return_type (*POINTER_NAME)(arg_type arg1, …)

Useful to pass callbacks to other functions.

Practically useful example: *Sorting an array using qsort function in stdlib.h library*

Qsort function takes in a pointer to a comparison function as last argument

void **qsort** *(void *array, size_t count, size_t size, comparison_fn_t compare)*

Function Pointers 2

Let us define **our comparison function** as:

```
int compare_doubles (const void *a, const void *b) {
    const double *da = (const double *) a;
    const double *db = (const double *) b;
    return (*da > *db) - (*da < *db);
}
```

This returns:

-1 if *a < *b
 0 if *a == *b
+1 if *a>*b

Function Pointers 3

```
int main(){
  double a[10];
  qsort (a, 10, sizeof (double), compare_doubles);
  return 0;
}
```

Example program using function pointers to sort

Structures

Structures are a derived type that collect a set of variables under one type

For example,

```
struct line {
    int a, b, c;  /* line is  ax + by = c  */
};

struct line diagonal;
diagonal.a = 1;
diagonal.b = 1;
diagonal.c = 0;
```

The . (period) is an operator on a structure, to access the correct member of the structure.

Operations on Structures

- Copy it

- Assign to it (as a whole unit)

- Get its address (with the & operator)

- Access a member variable (using . operator)

- **CANNOT compare two structures even if they are of the same type.**

The -> operator

- We often have a pointer to a structure and want to access its members and it can be done with:

(*ptr).member

[parantheses needed because unary * is of lower precedence than . operator.]

- Convenient Alternative:

ptr->member

- The dot(.) and **->** operators are left to right associative and have highest precedence. So, use parentheses when needed.

Example C program on

. and -> operators

Unions

Very Similar to Structures but:

- Memory allocated for a union variable is the maximum needed by any of its members.

- Only one member used in each union variable.

Unions

Example:

union job {
 char name[32];
 float salary;
 int worker_no;
};

struct job1 {
  char name[32];
  float salary;
  int worker_no;
};

name

32 bytes

Fig: Memory allocation in case of union

name + **salary** + **worker_no**

32 bytes   4 bytes   4 bytes

Fig: Memory allocation in case of structure

Example C Program on the Storage sizes needed for a Structure and a Union

malloc – Basic Memory Allocation

void * **malloc** *(size_t size)*   *[from stdlib.h]*

- returns a pointer to a newly allocated block *size* bytes long, or

- a null pointer if the block could not be allocated.

Example usage:

struct foo *ptr;

ptr = (struct foo *) malloc (sizeof (struct foo));

if (ptr == 0) abort ();

memset (ptr, 0, sizeof (struct foo)); //initialize to 0

calloc –Allocating cleared space

*void * **calloc** *(size_t count, size_t eltsize)   [from stdlib.h]*

- Allocates a block long enough to contain a vector of *count* elements, each of size *eltsize*.

- Its contents are cleared to zero before calloc returns

- Aside: use man pages for quick information about system functions. Eg. Type "man 3 free" in shell

- Aside: 3 is for section name. Section 3 contains C Library Functions. use "man man" to know more.

free –Allocating cleared space

void **free** *(void *ptr)*   *[from stdlib.h]*

- When you no longer need a block that you got with malloc or calloc, use the function free to make the block available to be allocated again

- The free function deallocates the block of memory pointed at by *ptr*.

- If you forget to call free, not the end of the world because all of the program's space is given back to the system when the process terminates.

Dynamic Memory Allocation

1. Dynamic memory allocation is done from the heap.

2. Allocators are used to manage memory

3. Heap is maintained as a collection of various-sized blocks.

4. A block is a contiguous chunk of virtual memory that is either allocated or free.

5. More details on allocators in a future lecture.

Memory Allocation

- *What* is in memory?

- *When* is it in memory?
  And, when is or was the memory space allocated?

- *Where* within memory is it?

What and *Where* are

- program code (machine code)

- global variables (data)

- stack

- heap

Each can be thought of as residing in its own, separate section of memory. These sections are often identified as segments.

When is the memory space allocated?

- static
  The compiler knows details of the allocation, so causes the assembler to allocate the space. This implies that the memory image created by the assembler contains the memory space.

- dynamic
  The allocation of memory space occurs *while the program executes*.

Program Code

The program's source code is compiled and then it is assembled to produce machine code (memory image).

To run the program, the memory image is placed (copied) into memory.

Therefore, the code is already in memory, and classified as a static allocation.

```
int x, y, z;   /* global variables */
int main() {
}
```

Both the compiler, and therefore the assembler, know exactly how much memory space is needed.

When the program (memory image) is placed into memory, the allocation of memory for the integers has been completed.

It is therefore classified as a static allocation.

```
#define MAX 4
int array[MAX];   /* global */
int main() {
}
```

```
char *stringptr;

stringptr = (char *)
  malloc(bytes_needed + 1);
```
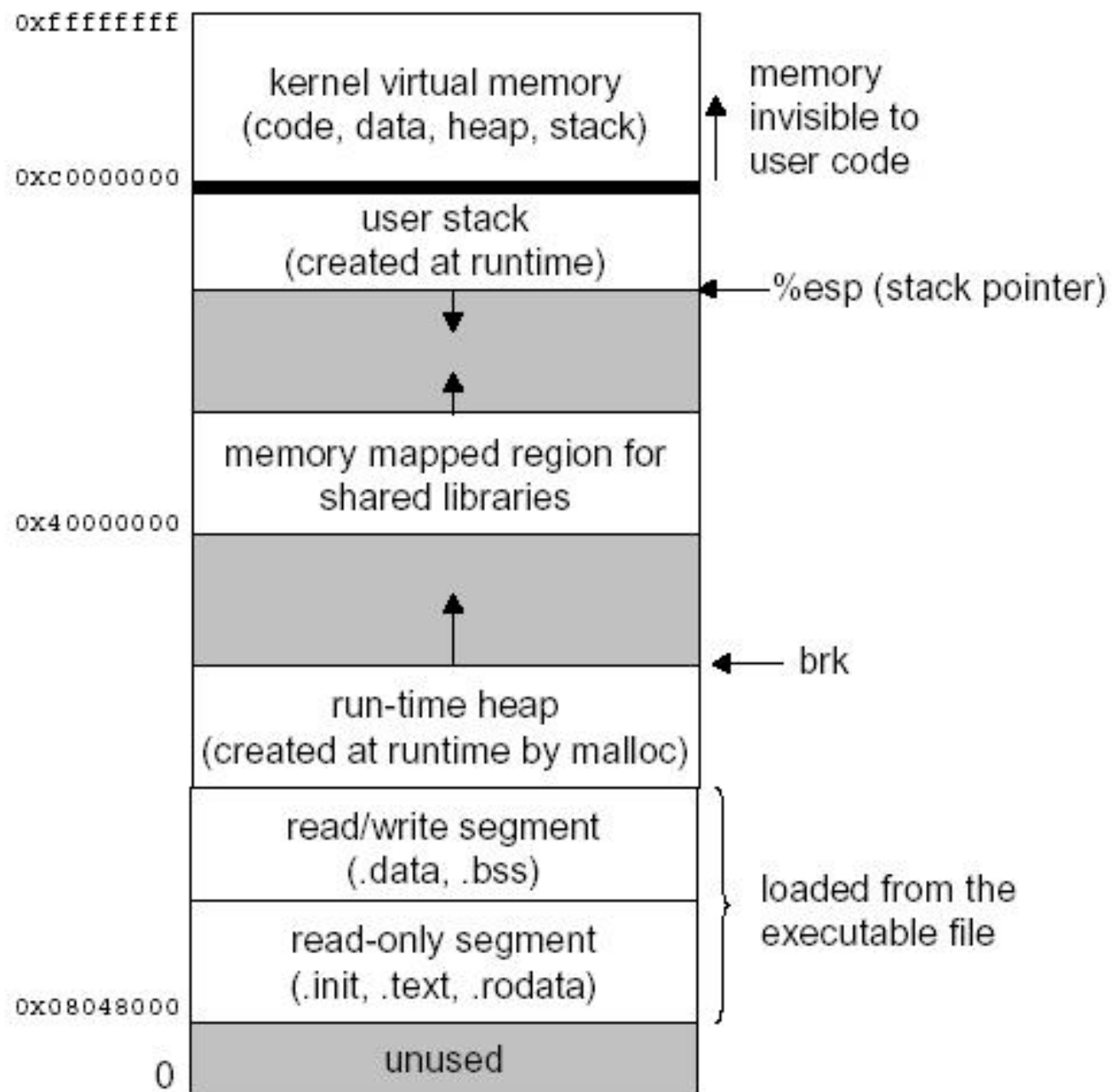
```
int fcn(int y) {
}

int main() {
  int x, y;
  y = 20;
  x = fcn(y);
}
```

Summary

- **Static**
  - program code
  - global variables

- **Dynamic**
  - anything that goes onto the stack, such as parameters and return address
  - memory allocated as the program is running, such as allocation done with `malloc()`

```
0xffffffff         ┌──────────────────────────┐
                   │  kernel virtual memory   │        memory
                   │  (code, data, heap, stack)│  ↑    invisible to
0xc0000000         ├──────────────────────────┤        user code
                   │      user stack          │
                   │   (created at runtime)    │
                   ├──────────────────────────┤  ◄─── %esp (stack pointer)
                   │            ↓             │
                   │            ↑             │
                   ├──────────────────────────┤
                   │ memory mapped region for │
                   │     shared libraries     │
0x40000000         ├──────────────────────────┤
                   │            ↑             │
                   ├──────────────────────────┤  ◄─── brk
                   │      run-time heap       │
                   │ (created at runtime by malloc)│
                   ├──────────────────────────┤ ┐
                   │   read/write segment     │ │
                   │      (.data, .bss)       │ │  loaded from the
                   ├──────────────────────────┤ │  executable file
                   │   read-only segment      │ │
                   │  (.init, .text, .rodata) │ │
0x08048000         ├──────────────────────────┤ ┘
                   │         unused           │
       0           └──────────────────────────┘
```

size command in linux

GNU size command lists the section sizes---and the total size---for each of the object in its argument list

For example:

size command output for endian.c program

text	data	bss	dec	hex	filename
1263	492	16	1771	6eb	endian

size command in linux

text: (readonly) Has code and constant data.

data: (readwrite) The data area contains global and static variables used by the program that are explicitly initialized with a non-zero (or non-NULL) value.

bss: (Block Started by Symbol) The BSS segment contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.