More Fun with

Global and Local Variables

under the Hood

- In which we find out where to find information about **global variables** in a **relocatable object file**,
- which **global variables** have **global** or **local linker symbols**,
- how they get complicated in **concurrent programming**,
- which variables other **threads** can access,
- and how **global** and **local** variables can access **heap** memory

Anne Huang
CS 361
netID: ahuang27

Motivation and Organization

This magazine covers selections from topics like linking, memory, and concurrency, but regroups them by whether they are relevant to **global variables**, **local static variables**, or **local automatic variables**.

First we'll discuss the things that are unique about dealing with **global variables** in systems programming, and then we'll talk about what we know about those same topics in relation to **local variables**.

This way, when you read about the contrasts in the ways different kinds of variables are handled in concurrent programming, you can refresh yourself about their other contrasts, side by side.

Finally, we'll review what we know about global and stack memory in relation to traversing them to find pointers into the heap.

Topics

GLOBAL VARIABLE

What are they?

- Global variables are not declared inside any functions (CS: APP 956)
- By convention, you can often find a global variable at the top of your C program
  - if you wanted to manipulate which variables it shows up before or after in your **symbol table**, you could move it around, above or below functions or other global variables. More on symbol tables in a couple pages…

  - Lives in global memory
  - Information about it in the **.data** section of a **relocatable object file**, if initialized
  - Information about it in the **.bss** section of a **relocatable object file**, if not initialized

What's a **relocatable object file**?

You know how when you're writing a program, what you're writing is basically in English?

After you save and exit your .c file, where you have written all your source code, that's still just text. You can directly open it up in Notepad and still read it.

When you input into the command line

gcc -o [and then the name of your .c file]

you'll get back a .o file. That's your relocatable object file! It's now in binary.

You still can't run it yet though. You'll have to turn it into an executable object file first. More on that later. (CS: APP 655-7)

Relocatable object files have **sections**

The ones we're interested in here are the **.data** section, where you'll find information about global variables that are initialized, and the .**bss** section, where you'll find information about global variables that are not initialized. (CS:APP 659)

Example of a Global Variable

```
int x = 0 ;
// ^ Often you would put a global variable here – this would go
// in .bss because it is declared as being equal to 0.

int main(int argc, char** argv)
{
  //…
  return 0;
}
```

Source: Taylor, 8/28/2015 lecture

So what's a **linker**?

The **linker** will take different .o files and turn them into a single executable object (the one you'll be able to run). To do that, every symbol in that object will need to mean one thing. (CS:APP 654-7)

The **linker** will associate different types of symbols with different types of global variables.

- **Nonstatic global variables**
  - If a global variable in **C** doesn't have a **static** label, it's kind of like declaring a global variable **public** in Java. If a global variable in **C** is **not static**, that means other files can access it.
  - **Nonstatic global variables** in **C** have **global linker symbols** – each variable is associated with a **symbol** that needs to be associated with a definition for your program to run
    - [**Connect to What You Know**: If you've used Netbeans for Java, you might have gotten warnings about your symbols when you referenced variables you hadn't declared or defined anywhere.]
- **Static global variables**
  - labeled "static" in C – only the file that defines them can access them
  - For example, the Elevator struct and the Passenger struct in main.c of HW5 are static – so we can't directly access them from the hw5.c file
  - have **local linker symbols**
  - Don't get these mixed up with (nonstatic) local variables – those live on the stack and the linker does not care about them!

(CS: APP 660)

Example

```
int sum  = 1 ;       // a global symbol, would go in .data.
static int val = 4; // a local symbol.

int main(int argc, char** argv)
{
    // …
  return 0;
}
```

Source: Taylor, 8/28/2015 lecture

If you use the **readelf** command in Linux with the name of your relocatable object file, you'll see a table that includes the names of your global variables – that's called the **symbol table!** Each row is called a **symbol table entry** and contains information about that symbol.

If you see a **3** next to the name of a variable in a symbol symbol entry, that means it's in the **.data** section – **3** is the index of that section in the relocatable object file.

(CS:APP 662)

EXAMPLE OF A SYMBOL TABLE ENTRY

```
Num:  Value   Size   Type     Bind     Ot  Ndx   Name
   8:        0     4   OBJECT GLOBAL 0     3    y
```

Source: CS:APP 662

Check your understanding

Let's say you're reverse engineering, and you know that the **symbol table entry** for a particular variable needs to have a 3 in the index column (as you see in the example above). What should you do when you declare that variable in your source code to make sure you replicate that symbol table entry?

Hint

3 - that means the .data section. What does that tell us?

Solution

Well, the .data section has your initialized global variables. So you need to declare that variable outside of any function and make sure you initialize it with a nonzero value right when you declare it.

SYMBOL TABLE ENTRY FOR A COMMON VARIABLE

By contrast, the **symbol table entry** for global variables that are not initialized have the value COMMON, or COM for short, in the index column of its **symbol table entry**.

Example of a COMMON Variable

```
/*generate.c*/

int *x ;

void generate()
{
    //…
    return 0;
}
```

Example of a Symbol Table Entry for a COMMON Variable

```
Num:  Value  Size   Type     Bind     Ot  Ndx  Name
   9:      4     4  OBJECT GLOBAL 0   COM  x
```

How can I remember what **COMMON** in the symbol table entry tells me about the variable?

Good question.

Since a global variable is not particular to any function, a reference to it in any function in any file will mean the same thing, in common.

These are kind of like **COMMON** blocks in Fortran (K & R 73).

Wait, but what if I've declared multiple global variables with the same name?

If you initialize more than one of them, you'll get an error. (Those are both called **strong** symbols, and you can't have more than one of those with the same name.)

If you've initialized only one of them, the linker will pick the one you initialized. (The one you initialized is **strong**; the one you didn't initialize is **weak**.)

If none of them are initialized, the linker will arbitrarily pick one of them. (They're all **weak**.)

(CS:APP 664)

Read further in CS: APP if you want to learn more about other symbols.

GLOBAL VARIABLES in CONCURRENT PROGRAMMING

CONCURRENT PROGRAMMING is a situation in which you'll have to pay particular attention to **global variables**.

What's concurrent programming?

You know how often, every line of code you write executes sequentially, one at a time, going from the top of your page to the bottom?

By contrast, in **concurrent programming**, you can have different parts of your code executing at the same time.

Source: http://randu.org/tutorials/threads/

A **peer thread** is another part of your program that runs at the same time within the same **address space** – so it shares your global variables.

An **address space** is a bunch of addresses that are all nonnegative integers, and they are ordered.

Source: CS:APP 778

If you write a function and make it your **thread routine**, it will run at the same time as other functions in your same program, and it can write to the same global variables that your other functions are writing to.

What do I need to get started with using different threads?

First you need to declare a **pthread_t**.

pthread_t, which is defined in the **sys/types.h** header file, is the ID of a thread (similar to the way pid_t is the type for the ID of a process, but creating a peer thread is different from creating a child process because a child process gets its own new address space, whereas a peer thread does not – read further in CS: APP if you want to learn more about the contrasts between threads and processes).

You use the **pthread_create** function to create a new thread. The arguments are:

1. A pointer to the **pthread_t**
2. Leave the 2nd argument NULL for now
3. A pointer to the function that will be your **thread routine**
4. The argument to be passed into the function that serves as your thread routine (if you want your function to have more than one argument, you'll have to pass in a pointer to a struct that contains those arguments

Sources: CS:APP 727, 949-50;
http://pubs.opengroup.org/onlinepubs/009696699/basedefs/sys/types.h.html

Sharing Global Variables across Peer Threads

- In **concurrent programming**, all **peer threads** share the same instance of a global variable.

  You don't always know what order statements in different threads are executing relative to one another. So if we're not careful, we're not always sure what the value of a specific global variable is at a specific point in time. We're not always sure which thread has manipulated it in what order.

Below are examples of functions that are **not thread-safe.** Here, different threads can share the local variable **counter**. If these functions are running at the same time, they can all write to the value of **counter**. Depending on what order they execute in, we can end up with different results for the variable **counter**.

```
unsigned int counter = 0; // Let's review: This would go
                          // in .bss because it is a global
                          // variable declared as equal to
                          // 0.
void bar1(void){
    counter++;
}
```

```
void bar2(void){
    counter*=10;
}
```

Sources: CS: APP 979-80; Taylor, 12/2/2015 lecture

In the example below, we have a **race condition**, in which the result of the global variable **counter** may be different depending on which thread executes first.

```
int counter = 1; // both the original thread and the tid1
                 // thread we create below can access this
                 // Let's review: this would be a global
                 // symbol. This would go in .data
                 // because we initialize it here, outside of
                 // of any function.
int main(int argc, char** argv){
  counter +=10;

  pthread_t tid1;
  pthread_create(&tid1, NULL, threadRoutine, NULL);
  pthread_join(tid, NULL); // This function causes us to
                           // wait for the thread passed
                           // in as the first argument to
                           // stop.
  return 0;
}
 void threadRoutine(void *vargp){
  counter *= 10;
}
```

Sources: Taylor, 8/26, 8/28, 11/18, 12/2/2015 lectures; CS:APP 951, 955, 984

EXTERNAL VARIABLE

What are they?

- This is when your file references variables that it does not define itself – their definitions are in other files

- These have **global symbols**

Their Symbol Table Entry Index

- Instead of seeing a numeric index next to the name of this variable in a symbol table, you'll see **UNDEF** (CS: APP 660-1)

LOCAL VARIABLE

What are they?

- You can find local variables declared in functions.

There are two kinds of local variables:

LOCAL STATIC VARIABLE

- Has the word "**static**" in the beginning of the declaration
- These actually don't live on the stack! These would have a **local linker symbol**
- You'll find these in .data / .bss too! (CS:APP 660)
- In concurrent programming, all peer threads share that same instance

Hey! That sounds familiar! Does this have anything to do with the variables I label **static** in Java?

There are some similarities. In **Java**, when a variable is static in a class, the value of that variable is shared across all instances of that class.

So then what's the difference between a global nonstatic variable and a local **static** variable in **C**?

Good question! Not all functions can access the local static variable. For example, if it is declared in the thread routine, only the function that is that thread routine can access it (CS: APP 995).

LOCAL AUTOMATIC VARIABLE

- no "**static**"
  - Lives on the stack!
  - As we said before, this would not have a linker symbol
- different instances in every thread (CS: APP 956)

Can one thread access a variable on the stack of another thread?

If you declare a local variable inside a function, but then you declare a global variable outside any functions and point that at the local variable in your function, other peer threads will be able to access it.

PUTTING IT ALL TOGETHER:

Accessing a Local Automatic Variable with a Pointer into the Stack

Example

```
int *x; // Let's review: this would be COMMON because it
        // is not initialized. It has a global linker symbol.

int main(int argc, char** argv)
{
  int *y = 5;  // Let's review: this would not be in either
               // .data or .bss because it is a local variable

  x = y;       // → Other threads would be able to access
               //     the stack variable y because it is being
               //     pointed to by the stack variable x
  return 0;
}
```

Source: Taylor, 8/26/2015 lecture

Peer Threads Sharing a Local Static Variable

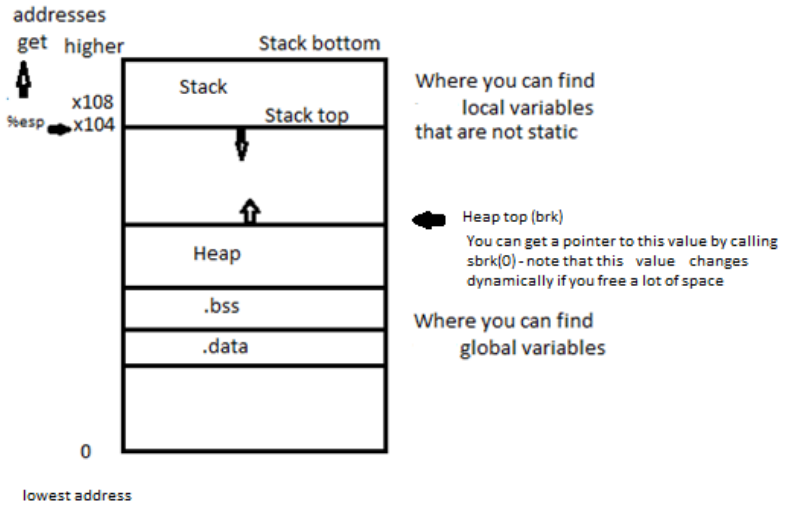
Example

```
int sum; // both the original thread and the tid1 thread
          // we create below will be able to access this
          // Let's review: The symbol table entry for this
          // variable would say that it is COMMON
int main(int argc, char** argv){
  int res = 2; // Even though the variable res is on the
  sum = res; // stack, another thread will be able to
             // access it because the global variable sum
             // references it.
  pthread_t tid1;
  pthread_create(&tid1, NULL, threadRoutine, NULL);
  return 0;
}
 void threadRoutine(void *vargp){
  int counter = 3; // this is a local variable other threads
                   // can't reference
                   // Let's review: this variable would
                   // not be in either .data or .bss
  static int val = 4;  // Side Note: This would have a local
                       // linker symbol.
                       // Because this is a local static
                       // variable, other threads can access
                       // it. This is not thread-safe. To learn
                       // more about how to fix this, read
                       // further in CS: APP.
}
```

Sources: Taylor, 8/26, 8/28, 11/18, 12/2/2015 lectures, CS:APP 955

In the previous pages, we saw examples of getting pointers to the stack in global memory. Now let's take a look at another way in which one section of memory can point at another.

MEMORY DIAGRAM



Note that the stack grows down (addresses get smaller), while the heap grows up (addresses get bigger).

(CS:APP 173, 220, 680, 811, 813, 815)

WHAT'S THE HEAP?

What happens if you don't know when you declare a variable how much space you will end up needing?

That's what you can use **malloc** for. The memory that is allocated as a result of a **malloc call** lives in the **heap**.

LOOKING FOR ROOT NODES THAT POINT INTO THE HEAP

A block of memory in the heap is only accessible to the user if it is part of a path from a root node from outside the heap, that points into the heap.

If you assign the return value of a malloc call to a global variable, then that global variable is a root node that points into the heap.

If you assign the return value of a malloc call to a local variable, then that stack variable is a root node that points into the heap.

(CS:APP 817, 839)

References:

Bryant, Randal, and David O'Hallaron. *Computer Systems: A Programmer's Perspective*. 2nd ed. Boston: Prentice Hall, 2011.

Kernighan, Brian, and Dennis Ritchie. *The C. Programming Language*. 2nd ed. Upper Saddle River, NJ: Prentice Hall, 1988.

http://pubs.opengroup.org/onlinepubs/009696699/basedefs/sys/types.h.html

http://randu.org/tutorials/threads/

Taylor, Cynthia. Course lecture 2: Compiling and linking. University of Illinois at Chicago, Aug. 26, 2015.

---. Course lecture 3: Symbols. University of Illinois at Chicago, Aug. 28, 2015.

---. Course lecture 35: Sharing and threads. University of Illinois at Chicago, Nov. 18, 2015.

---. Course lecture 40: Other threading issues. University of Illinois at Chicago, Dec. 2, 2015.