Operating Systems
ECE344
Understanding Program Address Space

Ashvin Goel
ECE
University of Toronto
Overview

- Address space of a program
- Review of program data
- Understanding program stack, function call and return
An address space is the set of (virtual) memory regions accessible to a program:

- Text region – the program code (usually read only)
- Data, heap regions – global/static, heap/dynamic variables
- Stack regions – used for function and system calls

Each program has its own address space.
**Review of Program Data**

- **Type of variable**: `char` and `char *` with `var`.
- **Variable**: `var`.
- **Address of a variable**: is a memory location.
- **Pointer stores memory address**.
- **Address of a variable**: is a memory location.
- **Pointer dereference returns value at pointer address**.

```
cchar       var;
cchar * p  =  &var;
c

*p == var
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>150</td>
</tr>
<tr>
<td>5</td>
<td>34</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>
int a = 5;
int main(int argc)
{
    int b=10, c=a;
    c = f(b+c);
    return c;
}
int f(int x)
{
    int y = x;
    return y;
}
Function Call and Return

int a = 5;
int main(int argc)
{
    int b=10, c=a;
    c = f(b+c);
    return c;
}

int f(int x)
{
    int y = x;
    return y;
}
Summary

- Address space of a program contains program code and data
  - Program code is located in text region
  - Program globals or static variables are located in data region
- Function data is located in the stack region
  - Stack is composed of activation frames
    - One per active function call
    - A function call pushes an activation frame on the stack
    - A function return pops the current activation frame
  - Activation frames are linked using fp register
  - Function data is accessed using either fp or sp register
Why do we need two pointers, the frame pointer and the stack pointer, to locate data on the stack?

You see the following instructions at the beginning of an x86 function: `push %rbp; mov %rsp, %rbp`. Similarly, you see the following instructions at the end of an x86 instruction: `mov %rbp, %rsp; pop %rbp`. Can you explain how these instructions modify: a) the two pointers above? b) the activation frame?
Why do we need two pointers, the frame pointer and the stack pointer, to locate data on the stack?

- The frame pointer register is used to track the data in the current activation frame, and to link the activation frames of the active functions. The stack pointer register is used to track the top of the stack. Typically, all parameters, local variables and return values are accessed using the frame pointer. This pointer is only updated on function call and return. The stack pointer can be updated at any time within a function. For example, when a new local variable is allocated within a new scope, the stack pointer is updated. This pointer is also updated by the call and the return instructions.

- Optimizing compilers allow accessing the stack using only the stack pointer, so that the frame pointer register can be used as a general-purpose register. However, this complicates debugging a program since the stack pointer can move during the function call, and so the different variables that are indexed using the stack pointer may not appear to have correct values. This is why debugging versions of programs always use both the frame pointer and the stack pointer, while optimized programs may only use the stack pointer.
You see the following instructions at the beginning of an x86 function: `push %rbp; mov %rsp, %rbp`. Similarly, you see the following instructions at the end of an x86 instruction: `mov %rbp, %rsp; pop %rbp`. Can you explain how these instructions modify:

a) the two pointers above?

b) the activation frame?

In the assembly shown, rbp is the frame pointer register. When a caller function (e.g., main) needs to invoke the callee function (e.g., f function), it will first push the arguments of f in the activation frame of f, and then invoke the “call” assembly instruction. This instruction pushes the return address on the stack, and at this point, the stack pointer points to the return address. Now the callee function (e.g., f) will start running. See next two slide.
At the beginning of function f, you will see assembly code shown below:

1. push %rbp

In C:

    sp--;            // decrement stack pointer, so stack pointer
    // points to “prev fp” location
    *sp = fp;        // save the current fp register value (of main)
    // at the current stack location, this fills
    // the “prev fp” location in the
    // activation frame of f function

2. mov %rsp, %rbp

In C:

    fp = sp;         // the frame pointer points to the stack
    // pointer at the start of the function.
    // after this, the fp is not changed until the
    // function returns, but sp can be changed,  
    // e.g., when a local variable is allocated on
    // the stack.
Think Time Answers (End of Function)

At the end of function f, you will see assembly code shown below. This code does the reverse of the code shown at the beginning of the function:

1. mov %rbp, %rsp

   In C:
   
   sp = fp;  // set stack pointer to frame pointer

2. pop %rbp

   In C:
   
   fp = *sp;  // frame pointer is assigned value of prev fp
   sp++;  // stack pointer is incremented, so it
   // points to return address

The two instructions above (mov, pop) are the same as the “leaveq” instruction. Next, the callee function invokes the “ret” instruction, which pops the return address from the stack and jumps back to the caller using this address. The caller then pops the return value, and the arguments it had pushed on the stack for the callee. At this point, the stack pointer points to the location before the callee (f) was invoked.