

CPSC 213

Introduction to Computer Systems

Unit 1e

Procedures and the Stack

Reading

Companion

• 2.8

Textbook

• *Procedures, Out-of-Bounds Memory References and Buffer Overflows*
• 3.7, 3.12

Local Variables of a Procedure

```
public class A {  
    public static void b () {  
        int l0 = 0;  
        int l1 = 1;  
    }  
}
```

```
void b () {  
    int l0 = 0;  
    int l1 = 1;  
}
```

```
public class Foo {  
    static void foo () {  
        A.b ();  
    }  
}
```

Java

Can l0 and l1 be allocated statically (i.e., by the compiler)?

- [A] Yes
- [B] Yes, but only by eliminating recursion
- [C] Yes, but more than just recursion must be eliminated
- [D] No, no change to the language can make this possible

Dynamic Allocation of Locals

```
void b () {  
    int l0 = 0;  
    int l1 = 1;  
}
```

Lifetime of a local

- starts when procedure is called and ends when procedure returns
- allocation and deallocation are implicitly part of procedure call
- Should we allocate locals from the heap?
 - the heap is where Java new and C malloc allocate dynamic storage
 - could we use the heap for locals?
 - [A] Yes
 - [B] Yes, but it would be less efficient to do so
 - [C] No

Procedure Storage Needs

frame

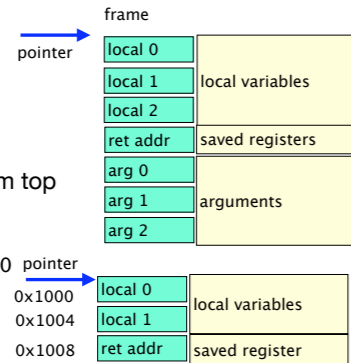
- local variables
- saved registers
 - return address
- arguments

access through offsets from top

- just like structs with base

simple example

- two local vars
- saved return address



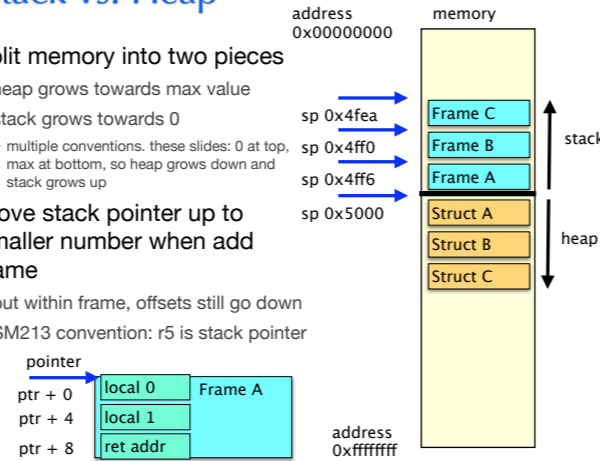
Stack vs. Heap

split memory into two pieces

- heap grows towards max value
- stack grows towards 0
 - multiple conventions. these slides: 0 at top, max at bottom, so heap grows down and stack grows up

move stack pointer up to smaller number when add frame

- but within frame, offsets still go down
- SM213 convention: r5 is stack pointer



Runtime Stack and Activation Frames

Runtime Stack

- like the heap, but optimized for procedures
- one per thread
- grows "up" from higher addresses to lower ones

Activation Frame

- an "object" that stores variables in procedure's local scope
 - local variables and formal arguments of the procedure
 - temporary values such as saved registers (e.g., return address) and link to previous frame
- size and relative position of variables within frame is known statically

Stack pointer

- register reserved to point to activation frame of current procedure
- SM213 convention: r5
- accessing locals and args static offset from r5, the stack pointer (sp)
 - locals are accessed exactly like instance variables; r5 is pointer to containing "object"

Compiling a Procedure Call / Return

Procedure Prologue

- code generated by compiler to execute just before procedure starts
- allocates activation frame and changes stack pointer
 - subtract frame size from the stack pointer r5
- saves register values into frame as needed; save r6 always

Procedure Epilogue

- code generated by compiler to execute just before a procedure returns
- restores saved register values
- deallocates activation frame and restore stack pointer
 - add frame size to stack pointer r5

Snippet 8: Caller vs. Callee

```
foo: deca r5 # sp -= 4 for ra  
     st r6, (r5) # *sp = ra  
     gpc $6, r6 # r6 = pc  
     j b # goto b 0  
  
ld (r5), r6 # ra = *sp  
inca r5 # sp += 4 to discard ra  
j (r6) # return  
  
b: deca r5 # sp -= 4 for ra  
   st r6, (r5) # *sp = ra  
   deca r5 # sp -= 4 for l1  
   deca r5 # sp -= 4 for l0  
  
ld $0, r0 # r0 = 0  
st r0, 0x0(r5) # l0 = 0  
ld $0x1, r0 # r0 = 1  
st r0, 0x4(r5) # l1 = 1  
  
inca r5 # sp += 4 to discard l0  
inca r5 # sp += 4 to discard l1  
ld (r5), r6 # ra = *sp  
inca r5 # sp += 4 to discard ra  
j (r6) # return
```

- 1 allocate frame save r6
- 2 call b()
- 3 save r6 and allocate frame
- 4 body
- 5 deallocate frame return
- 6 restore r6 deallocate frame return

Optimized Procedure Call / Return

Eliminate Save/Restore r6 For Leaf Procedures

- only need to save/restore r6 if procedure calls another procedure
- otherwise r6 is untouched, no need to save to stack
- can determine statically
- Procedure Prologue
 - code generated by compiler to execute just before procedure starts
 - allocates activation frame and changes stack pointer
 - subtract frame size from the stack pointer r5
 - saves registers into frame as needed; saves r6 **only if procedure is not a leaf**

Procedure Epilogue

- code generated by compiler to execute just before a procedure returns
- restores any saved register values
- deallocates activation frame and restore stack pointer
 - add frame size to stack pointer r5

Snippet 8: Optimized Leaf Procedure

```
foo: deca r5 # sp -= 4 for ra  
     st r6, (r5) # *sp = ra  
     gpc $6, r6 # r6 = pc  
     j b # goto b 0  
  
ld (r5), r6 # ra = *sp  
inca r5 # sp += 4 to discard ra  
j (r6) # return  
  
b: deca r5 # sp -= 4 for ra  
   st r6, (r5) # *sp = ra  
   deca r5 # sp -= 4 for l1  
   deca r5 # sp -= 4 for l0  
  
ld $0, r0 # r0 = 0  
st r0, 0x0(r5) # l0 = 0  
ld $0x1, r0 # r0 = 1  
st r0, 0x4(r5) # l1 = 1  
  
inca r5 # sp += 4 to discard l0  
inca r5 # sp += 4 to discard l1  
ld (r5), r6 # ra = *sp  
inca r5 # sp += 4 to discard ra  
j (r6) # return
```

- 1 allocate frame save r6
- 2 call b()
- 3 save r6 and allocate frame
- 4 body
- 5 deallocate frame return
- 6 restore r6 deallocate frame return

Arguments and Return Value

return value

- SM213 convention: in register r0

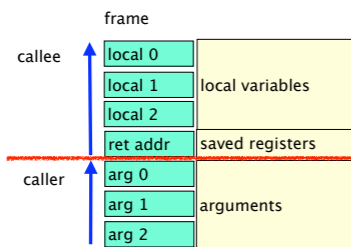
arguments

- in registers or on stack
- if on stack, must be passed in from caller

Procedure Storage Needs

allocate/deallocate stack frame for callee is done by combination of caller and callee

- caller: save/restore registers r0-r3
 - if need their current values after call
- caller: arguments
 - if passed on stack
- callee: locals
- callee: save/restore registers r4-r7
 - if will change them during call
 - incl return address (if not leaf) r6
 - (r5 should only be used for stack)



Creating the stack

Every thread starts with a hidden procedure

- its name is start (or sometimes something like crt0)

The start procedure

- allocates memory for stack
- initializes the stack pointer
- calls main() (or whatever the thread's first procedure is)

For example in Snippet 8

- the "main" procedure is "foo"
- we'll statically allocate stack at addresses 0x1000-0x1024 to keep simulation simple

```
.pos 0x100  
start: ld $0x1028, r5 # base of stack  
       gpc $6, r6 # r6 = pc  
       j foo # goto foo ()  
       halt
```

```
.pos 0x1000  
stack: .long 0x00000000  
       .long 0x00000000  
       ...
```

Snippet 9

```
public class A {  
    static int add (int a, int b) {  
        return a+b;  
    }  
}
```

```
public class foo {  
    static int s;  
    static void foo () {  
        s = add (1,2);  
    }  
}
```

Java

```
int add (int a, int b) {  
    return a+b;  
}
```

C

Formal arguments

- act as local variables for called procedure
- supplied values by caller

Actual arguments

- values supplied by caller
- bound to formal arguments for call

Arguments in Registers (S9-args-regs.s)

```
.pos 0x200  
foo:  
     deca r5 # sp -= 4  
     st r6, (r5) # save r6 to stack  
     ld $0x1, r0 # arg0 (r0) = 1  
     ld $0x2, r1 # arg1 (r1) = 2  
     gpc $6, r6 # r6 = pc  
     j add # goto add ()  
     ld $s, r1 # r1 = address of s  
     st r0, (r1) # s = add (1,2)  
     ld 0x0(r5), r6 # restore r6 from stack  
     inca r5 # sp += 4  
     j 0x0(r6) # return  
  
.pos 0x300  
add:  
     add r1, r0 # return (r0) = a (r0) + b (r1)  
     j 0x0(r6) # return
```

Arguments on Stack (S9-args-stack.s)

```

.pos 0x200
foo:
    decra r5          # sp=-4
    st -r6(r5)       # save r6 to stack
    ld $0x2, r0      # r0 = 2
    decra r5          # sp=-4
    st r0, (r5)      # save arg1 on stack
    ld $0x1, r0      # r0 = 1
    decra r5          # sp=-4
    st r0, (r5)      # save arg0 on stack
    gpc $6, r6       # r6 = pc
    j add            # goto add ()
    inca r5          # discard arg0 from stack
    inca r5          # discard arg1 from stack
    ld $s, r1        # r1 = address of s
    st r0, (r1)      # s = add (1,2)
    ld (r5), r6      # restore r6 from stack
    inca r5          # sp+=4
    j (r6)           # return

.pos 0x300
add:
    ld 0x0(r5), r0   # r0 = arg0
    ld 0x4(r5), r1   # r1 = arg1
    add -1, r0       # return (r0) = a(r0) + b(r1)
    j 0x0(r6)       # return
    
```

Question

```

void foo () {
    // r5 = 2000
    one ();
}

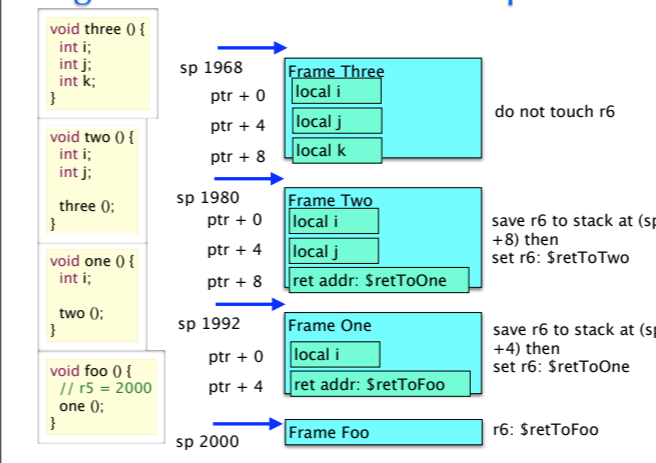
void one () {
    int i;
    two ();
}

void two () {
    int i;
    int j;
    three ();
}

void three () {
    int i;
    int j;
    int k;
}
    
```

- What is the value of r5 when executing in the procedure three() (in decimal)
- [A] 1964
 - [B] 2032
 - [C] 1968
 - [D] None of the above
 - [E] I don't know

Diagram of Stack for this Example



Stack Summary

- stack is managed by code that the compiler generates
 - stack pointer (sp) is current top of stack (stored in r5)
 - grows from bottom up towards 0
 - push (allocate) by decreasing sp value, pop (deallocate) by increasing sp value
- accessing information from stack
 - callee accesses local variables, saved registers, arguments as static offsets from base of stack pointer (r5)
- stack frame for procedure created by mix of caller and callee work
 - caller setup
 - arguments passed through stack: allocates room for them and save them to stack
 - sets up new value of r6 return address (to next instruction in this procedure, after the jump)
 - saves registers r0-r3 to stack if expect to use values after call
 - jump to callee code
 - callee setup (prologue)
 - unless leaf procedure, allocates room for old value of r6 and saves it to stack
 - save r4, r7 to stack if they will be overwritten
 - allocates space on stack for local variables
 - callee teardown (epilogue)
 - ensure return value in r0
 - deallocates stack frame space for locals
 - unless leaf procedure, restores old r6 and deallocates that space on stack
 - if previously saved, restore old r4/r7 and deallocate that space on stack
 - jump back to return address (location stored in r6)
 - caller teardown
 - deallocates stack frame space for arguments
 - restore r0-r3 if previously saved to stack, deallocate that space
 - use return value (if any) in r0

Variables: a Summary

- global variables
 - address known statically
- reference variables
 - variable stores address of value (usually allocated dynamically)
- arrays
 - elements, named by index (e.g. a[i])
 - address of element is base + index * size of element
 - base and index can be static or dynamic; size of element is static
- instance variables
 - offset to variable from start of object/struct known statically
 - address usually dynamic
- locals and arguments
 - offset to variable from start of activation frame known statically
 - address of stack frame is dynamic

Buffer Overflows

Security Vulnerability in Buffer Overflow

Find the bug in this program

```

void printPrefix (char* str) {
    char buf[10];
    char *bp = buf;

    // copy str up to "." input buf
    while (*str!='.')
        *(bp++) = *(str++);
    *bp = 0;

    // read string from standard input
    void getInput (char* b) {
        char* bc = b;
        int n;
        while ((n=fread(bc,1,1000,stdin))>0)
            bc+=n;
    }

    int main (int argc, char** argv) {
        char input[1000];
        puts ("Starting.");
        getInput (input);
        printPrefix (input);
        puts ("Done.");
    }
    
```

Possible array (buffer) overflow

How the Vulnerability is Created

- The "buffer" overflow bug
 - if the position of the first '.' in str is more than 10 bytes from the beginning of str, this loop will write portions of str into memory beyond the end of buf
- Giving an attacker control
 - the size and value of str are inputs to this program
- if an attacker can provide the input, she can cause the bug to occur and can determine what values are written into memory beyond the end of buf

the ugly

- buf is located on the stack
 - so the attacker now has the ability to write to portion of the stack below buf
 - the return address is stored on the stack below buf
- ```

void printPrefix (char* str) {
 char buf[10];
 char *bp = buf;

 // copy str up to "." input buf
 while (*str!='.')
 *(bp++) = *(str++);
 *bp = 0;
}

```
- The Stack when printPrefix is running
- |                |
|----------------|
| buf [0..9]     |
| other stuff    |
| return address |

# Mounting the Attack

- Goal of the attack
  - exploit input-based buffer overflow bug
  - to inject code into program (the virus/worm) and cause this code to execute
  - the worm then loads additional code onto compromised machine
- The approach
  - attack a standard program for which the attacker has the code
  - scan the code looking for bugs that contain this vulnerability
  - reverse-engineer the bug to determine what input triggers it
  - create an attack and send it
- The attack input string has three parts
  - a portion that writes memory up to the return address
  - a new value of the return address
  - the worm code itself that is stored at this address