Function calls in assembly

1/13/25

Administrivia

- HW 1 (ASCII art in assembly) due Wed night
- Lots of extra credit:
  - Another candidate in the next two days
    - Lunch at 12:15 on Tuesday, Oak Room
    - Research talk at 4:15pm, SMC A201 (reception at 3:45)
    - Teaching demonstration at 9:30 on Wednesday (A206)
  - Next Monday: MLK convocation at 11am

Recall: Loading and storing integers

- To store an int from a register to memory:

```
sw    reg, address       #"store word"
```

- To load an int from memory to a register:

```
lw    reg, address       #"load word"
```

- For both, address is

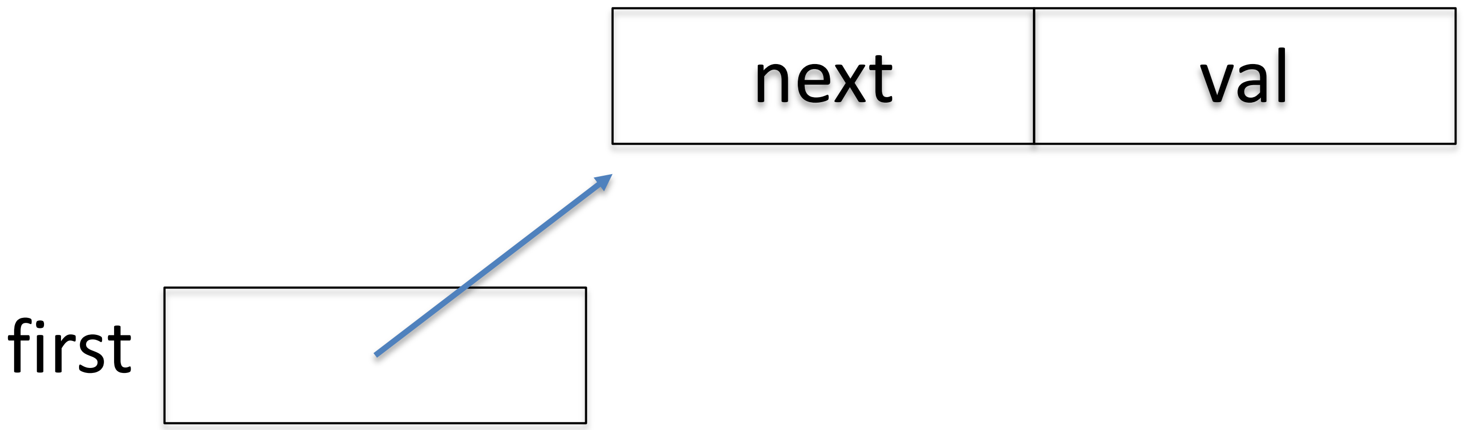
```
(register)             #use register value
imm(register)          #use imm + register value
```

i.e. an integer

What about storing objects in memory?

- Assembly (and C) lack true classes
- Can store structs (basically classes w/o methods)

Recall: Linked lists



Parts of a function call

1. Place parameters where function can get them
2. Transfer control
3. Acquire needed storage and save registers
4. Perform the task
5. Place return value where calling program can get it
6. Restore registers and free storage
7. Return control to point of origin

1. Place parameters where function can get them
5. Place return value where calling program can get it

- Power of convention:
  - Put function arguments into \$a0, \$a1, ...
  - Put return value into \$v0

2. Transfer control
7. Return control to point of origin

- Program counter: Register containing address of next instruction to execute

2. Transfer control
7. Return control to point of origin

- Program counter: Register containing address of next instruction to execute

- jal instruction “jump and link”
  - Changes PC and stores its old value in register $ra

- jr instruction changes PC to value of a register

Print and increment function

middle_man: Repackaging print (aka print_and_increment)

```
print:  addi   $v0, $zero, 1
        syscall
        addi   $v0, $a0, 1
        jr     $ra

middle_man:
        jal    print
        jr     $ra
```

Why doesn't middle_man (which claims to print its argument and return it + 1) work?

```
print:  addi    $v0, $zero, 1
        syscall
        addi    $v0, $a0, 1
        jr      $ra

middle_man:
        jal     print
        jr      $ra
```

A. middle_man incorrectly passes arguments
B. middle_man incorrectly calls print
C. middle_man doesn't return correctly
D. middle_man doesn't pass out the correct return value
E. Not exactly one of the above

Why doesn’t middle_man (which claims to print its argument and return it + 1) work?

```
print:  addi    $v0, $zero, 1
        syscall
        addi    $v0, $a0, 1
        jr      $ra

middle_man:
        jal     print
        jr      $ra
```

A. middle_man incorrectly passes arguments
B. middle_man incorrectly calls print
C. middle_man doesn’t return correctly
D. middle_man doesn’t pass out the correct return value
E. Not exactly one of the above

Register conventions

- Functions preserve the contents of the s registers ($s0, $s1, ...)

- When a function call is made, all other registers may change value

Register conventions

- Functions preserve the contents of the s registers ($s0, $s1, ...)
  - Save them to memory at beginning of function
  - Restore them from memory before returning
- When a function call is made, all other registers may change value

Register conventions

Only true if you make them so

- Functions preserve the contents of registers ($s0, $s1, ...)
  - Save them to memory at beginning of function
  - Restore them from memory before returning
- When a function call is made, all other registers may change value

Approach 1: Save to .data segment

```
          .data
funcRegs: .space 8

          .text
func:     la   $t0, funcRegs
          sw   $ra, ($t0)
          sw   $s0, 4($t0)
          ...
          #function body
          ...
          la   $t0, funcRegs
          lw   $ra, ($t0)
          lw   $s0, 4($t0)
          jr   $ra
```

Approach 1: Save to .data segment

```
          .data
funcRegs: .space 8

          .text
func:     la   $t0, funcRegs
          sw   $ra, ($t0)
          sw   $s0, 4($t0)
          ...
          #function body
          ...
          la   $t0, funcRegs
          lw   $ra, ($t0)
          lw   $s0, 4($t0)
          jr   $ra
```

When doesn't this work?

A. Does not scale beyond a few functions

B. func cannot be recursive

C. func cannot be compiled without knowing the context of calls to it

D. More than one of the above

E. This works in all cases

Approach 1: Save to .data segment

```
        .data
funcRegs: .space 8

        .text
func:   la   $t0, funcRegs
        sw   $ra, ($t0)
        sw   $s0, 4($t0)
        …
        #function body
        …
        la   $t0, funcRegs
        lw   $ra, ($t0)
        lw   $s0, 4($t0)
        jr   $ra
```

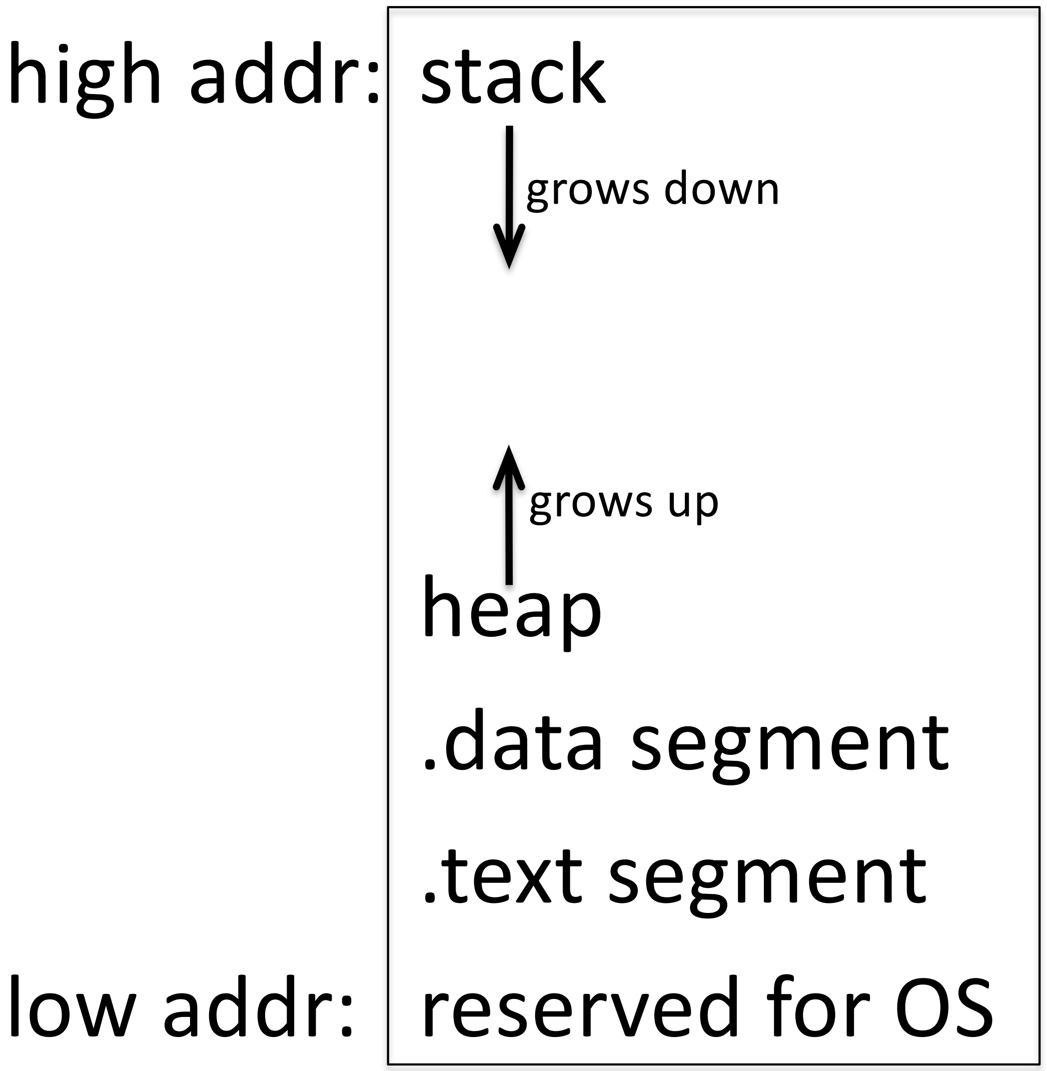
When doesn't this work?

A. Does not scale beyond a few functions
B. <u>func cannot be recursive</u>
C. func cannot be compiled without knowing the context of calls to it
D. More than one of the above
E. This works in all cases

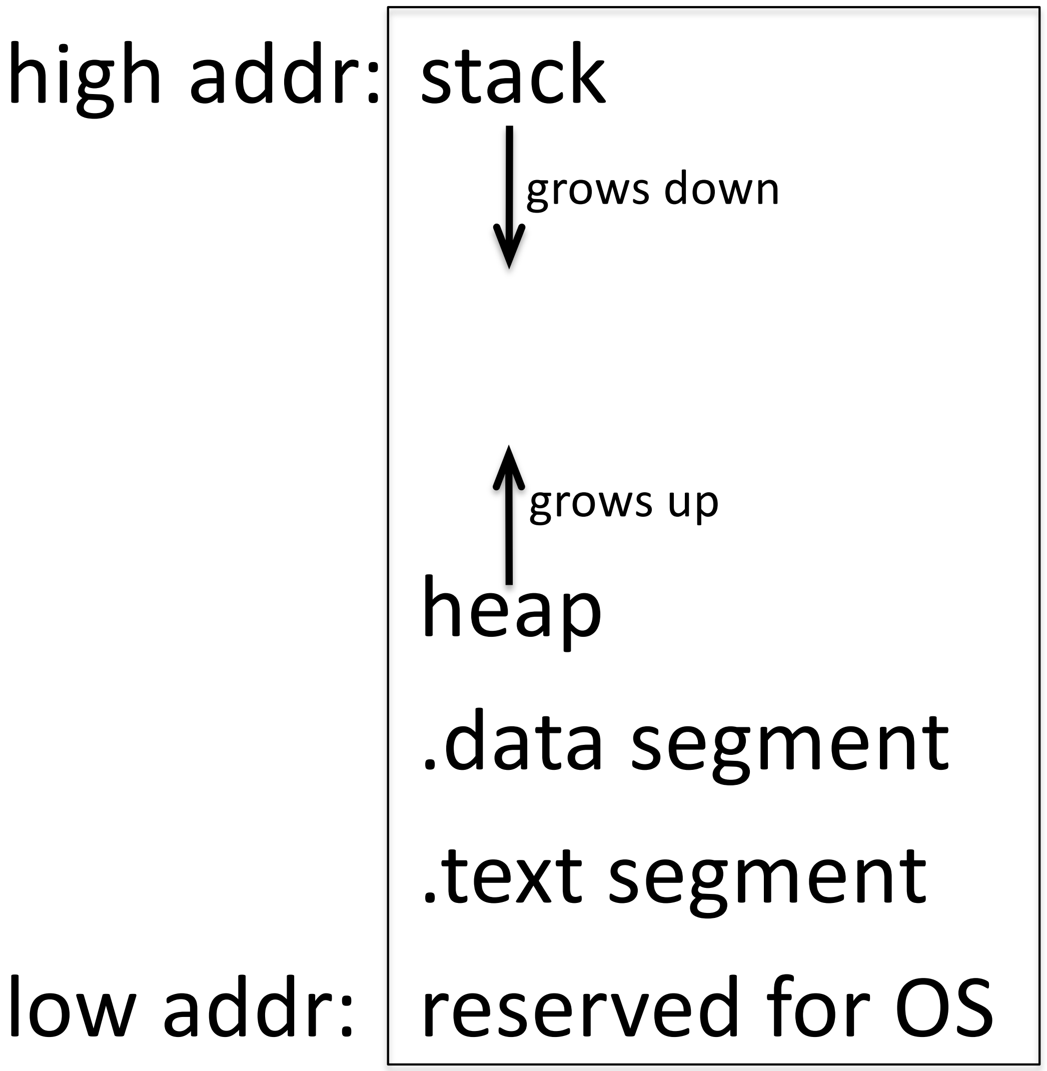
Acquire needed storage and save registers

- Issue: Making a function call overwrites $ra, imperiling the calling function’s ability to return

Approach 2: Save to the stack



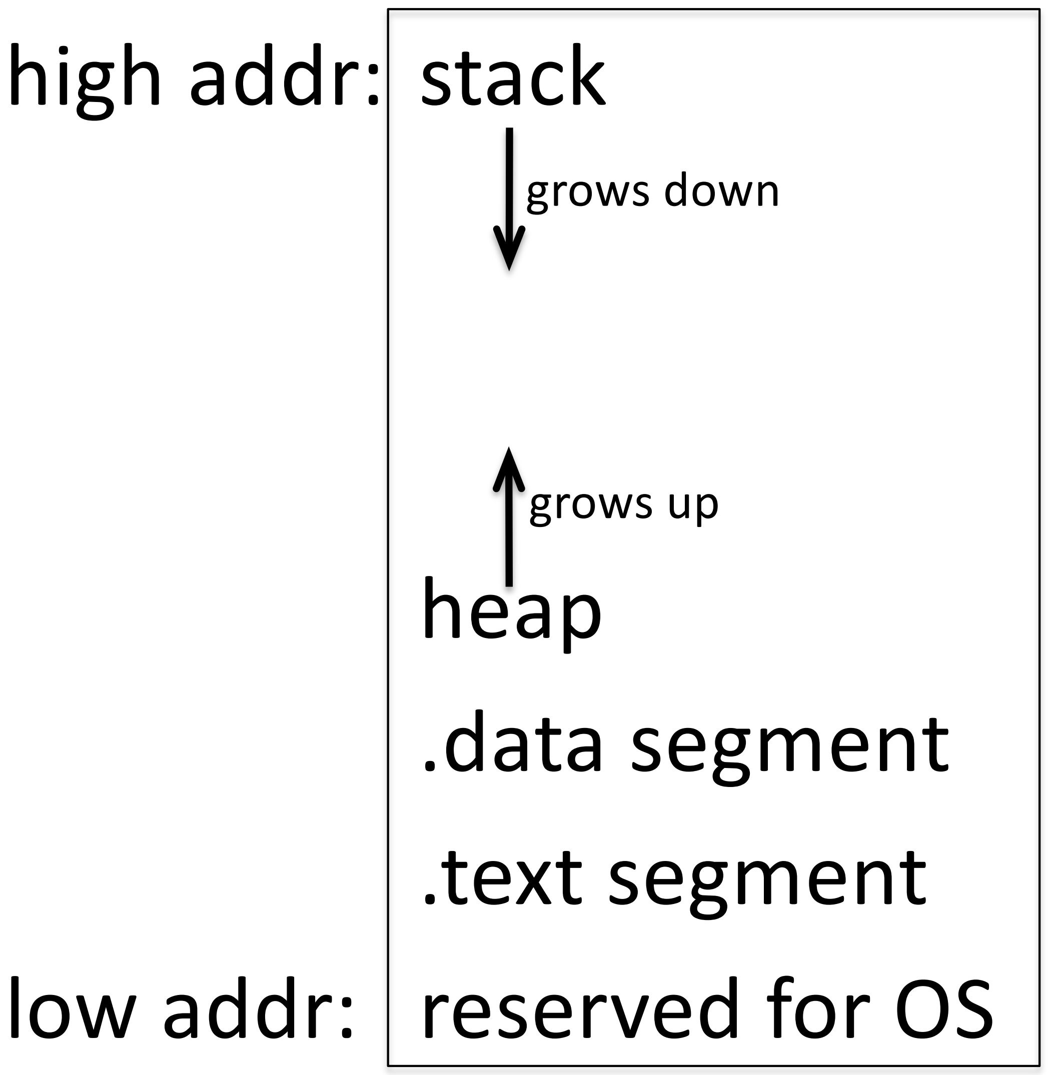
Approach 2: Save to the stack



Stack composed of “activation records” or “stack frames”, each with the local variables and saved registers for one function call

Bottom of the stack is stored in $sp (stack pointer)

Approach 2: Save to the stack



Stack composed of "activation records" or "stack frames", each with the local variables and saved registers for one function call

Bottom of the stack is stored in \$sp (stack pointer)

To reserve another frame:
    \$sp = \$sp – (frame size)

To free the frame:
    \$sp = \$sp + (frame size)

Parts of a function call

1. Place parameters where function can get them
2. Transfer control
3. Acquire needed storage and save registers
4. Perform the task
5. Place return value where calling program can get it
6. Restore registers and free storage
7. Return control to point of origin