Computer Systems
Lecture 13

Variables

Dr Lito Michala, Dr José Cano
School of Computing Science
University of Glasgow
Spring 2020

Outline

- Retrospective

- Variables
  - Static variables
  - Local variables
  - Dynamic variables

- The call stack

- Recursive factorial

Retrospective

- What is a computer program?
- What is a variable?

What is a computer program?

- A beginner's view
  - The computer runs programs
  - A program is lines of code (Python, C, whatever)

- The strange program shows how wrong that is!

- A more sophisticated view
  - The lines of source code are input to the assembler (compiler) which generates the initial value of the machine language
  - When a program is booted, initial machine language is stored in memory
  - The computer executes the machine language instructions in memory; the original assembly language code (labels and all) no longer exists

- Essential concepts
  - Source code and object code
  - Compile time and run time

What is a variable?

- Beginner's view

  - A variable is a box with a name that holds a value

  - An expression can use the value in the box; an assignment can modify the value in the box

  - In assembly language, you define a variable with a data statement

- A more sophisticated view

  - Variables are distinct from variable names: many variables may have the same name

  - A variable has a scope in a program: a region where it corresponds to a particular box

  - Variables do not correspond to data statements: they are created and destroyed dynamically as a program runs

  - Initialising a variable is not the same as assigning a value to it

Review of procedures: Call (jal), return (jump)

- To call a procedure dowork: jal R13,dowork[R0]
    - The address of the instruction after the jal is placed in R13
    - The program jumps to the effective address, and the procedure starts executing

- To return when the procedure has finished: jump 0[R13]
    - The effective address is 0 + the address of the instruction after the jal
    - The program jumps there and the main program resumes

Review: Basic calls with jal

- The simplest kind of procedure
  - Call it with jal R13,procname[R0]
  - It returns by executing jump 0[R13]

Review: Activation records, a.k.a. stack frames

- There is a call stack or execution stack that maintains complete information about all procedure calls and returns

- Every "activation" of a procedure pushes a stack frame

- When a procedure returns, its stack frame is popped (removed) from the stack

- R14 contains the address of the current (top) stack frame

- The stack frame contains
  - A pointer to the previous stack frame (required to make the pop work)
  - The return address (saved value of R13)
  - The saved registers (so the procedure can use the registers without destroying information)
  - Local variables (the procedure can have some memory of its own to use)

11

Review: Sequence of stack operations
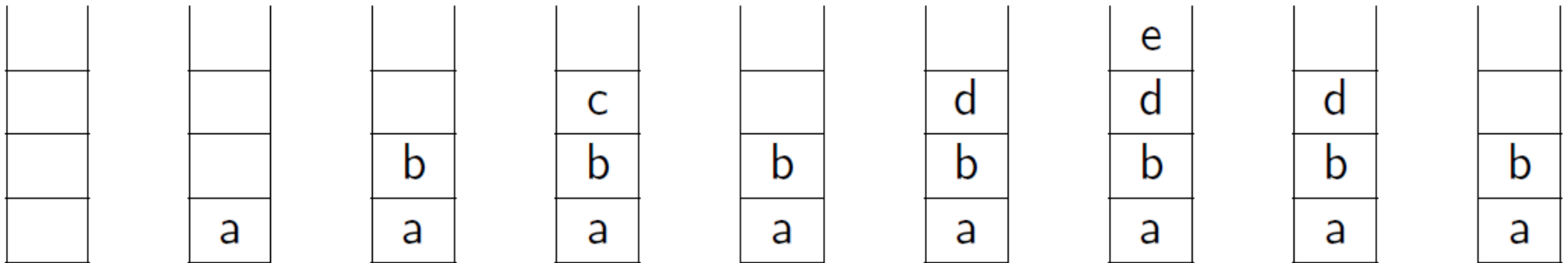
- empty; push a; push b, push c, pop returns c, push d, push e, pop returns e, pop returns d, …

Outline

- Retrospective

- Variables
    – Static variables
    – Local variables
    – Dynamic variables

- The call stack

- Recursive factorial

Access to variables

- Depending on the programming language, there are several different ways that variables can be allocated

- For each of these, there is a corresponding way to access the variable in memory

- Three key issues
  - The lifetime of a variable: when it is created, when it is destroyed
  - The scope of a variable: which parts of the source program are able to access the variable
  - The location of a variable: what its address in memory is

- The compiler generates the correct object code to access each variable

Three classes of variable

- **Static variables**: Sometimes called global variables
    - Visible through the entire program

- **Local variables**: Sometimes called automatic variables
    - Visible only in a local procedure

- **Dynamic variables**: Sometimes called heap variables
    - Used in object oriented and functional languages

Static variables

- The lifetime of a static variable is the entire execution of a program
  - When the program is launched, its static variables are created
  - They continue to exist, and to retain their values, until the program is terminated

- In C, you can declare a variable to be static

- In Pascal, all global variables are static
  - i.e. all variables that aren't defined locally

- So far, we have been using static variables

Combining static variables with code

- The simple way we have been defining variables makes them static

```
    load R1,x[R0]          ; R1 := x

    ...
    trap R0,R0,R0          ; terminate


; Static variables
x data 0
n data 100
```

- These variables exist for the entire program execution
  – There is one variable x, and one variable n

Disadvantages of combining variables and code

- The executable code cannot be shared.
  - Suppose two users want to run the program
  - Each needs to have a copy of the entire object, which contains both the instructions and the data
  - That means the instructions are duplicated in memory
  - This is inefficient use of memory

- To avoid the duplication of instructions, we need to separate the data from the code

- Modern operating systems organise information into segments
  - A code segment is read-only, and can be shared
  - A data segment is read/write, and cannot be shared

Local variables

- Local variables are defined in a function, procedure, method, or in a begin...end block, or a {...} block

- A local variable has one name, but there may be many instances of it if the function is recursive

- Therefore local variables cannot be stored in the static data segment

- They are kept in stack frames

- The compiler (or assembler) works out the address of each local variable relative to the address of the stack frame

- The variables are accessed using the stack frame register

Accessing local variables

load R1,x[R14]   ; access local variable x; R14 points to frame

- The compiler (or the programmer) works out the exact format of the stack frame

- Each local variable has a dedicated spot in the stack frame, and its address (relative to the frame) is used in the load instruction

Dynamic variables

- A dynamic variable is created explicitly
  - e.g. using new in Java

- It is not limited to use in just one function

- The lifetime of a dynamic variable does not need to follow the order that stack frames are pushed or popped

- So dynamic variables can't be kept in the static data segment, and they can't be kept on the stack

The Heap

- Languages that support dynamic variables (Lisp, Scheme, Haskell, Java) have a region of memory called the heap

- The heap typically contains a very large number of very small objects

- The heap contains a free space list, a data structure that points to all the free words of memory

- The heap is maintained by the language "runtime system", not by the operating system

- When you do a new, a (small) amount of memory is allocated from the heap and a pointer (address) to the object is returned

- When the object is no longer required, the memory used to hold it is linked back into the free space list

Outline

- Retrospective

- Variables
  - Static variables
  - Local variables
  - Dynamic variables

- The call stack

- Recursive factorial

The call stack

- Each procedure call pushes information on the stack

- The information needed by the procedure is in the stack frame (also called activation record)

- Each procedure return pops information o the stack

- A register is permanently used as the stack pointer
  - For each computer architecture, there is a standard register chosen to be the stack pointer
  - In Sigma16, R14 is the stack pointer
  - When you call, you push a new stack frame and increase R14
  - As a procedure runs, it access its data via R14
  - When you return, you set R14 to the stack frame below

Simplest stack: return addresses

return address
return address
return address
return address

- Just save the return address on the stack

Saved registers

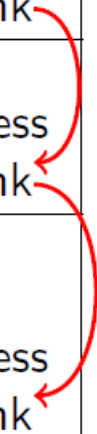```
       save R4
       save R3
       save R2
       save R1
    return address
       save R1
    return address
       save R3
       save R2
       save R1
    return address
```

- Save the registers the procedure needs to use on the stack, and restore them before returning
  - This way the procedure won't crash the caller

# Dynamic links

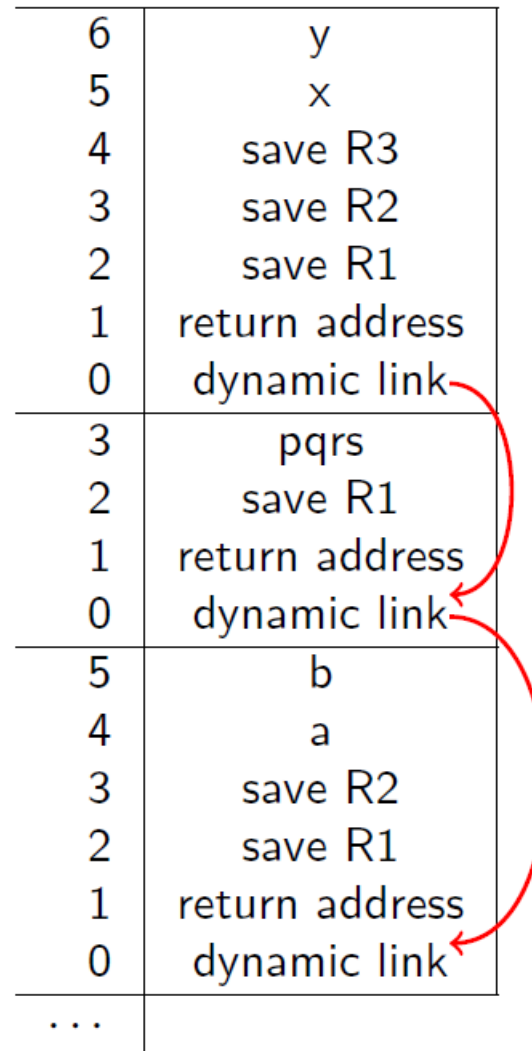| | |
|---|---|
| 4 | save R3 |
| 3 | save R2 |
| 2 | save R1 |
| 1 | return address |
| 0 | dynamic link |
| 2 | save R1 |
| 1 | return address |
| 0 | dynamic link |
| 3 | save R2 |
| 2 | save R1 |
| 1 | return address |
| 0 | dynamic link |
| . . . | |

- **Problem**: since each activation record can have a different size, how do we pop the top one o the stack?

- **Simplest solution**: each activation record contains a pointer (called dynamic link) to the one below

# Local variables



| | |
|---|---|
| 6 | y |
| 5 | x |
| 4 | save R3 |
| 3 | save R2 |
| 2 | save R1 |
| 1 | return address |
| 0 | dynamic link |
| 3 | pqrs |
| 2 | save R1 |
| 1 | return address |
| 0 | dynamic link |
| 5 | b |
| 4 | a |
| 3 | save R2 |
| 2 | save R1 |
| 1 | return address |
| 0 | dynamic link |
| . . . | |
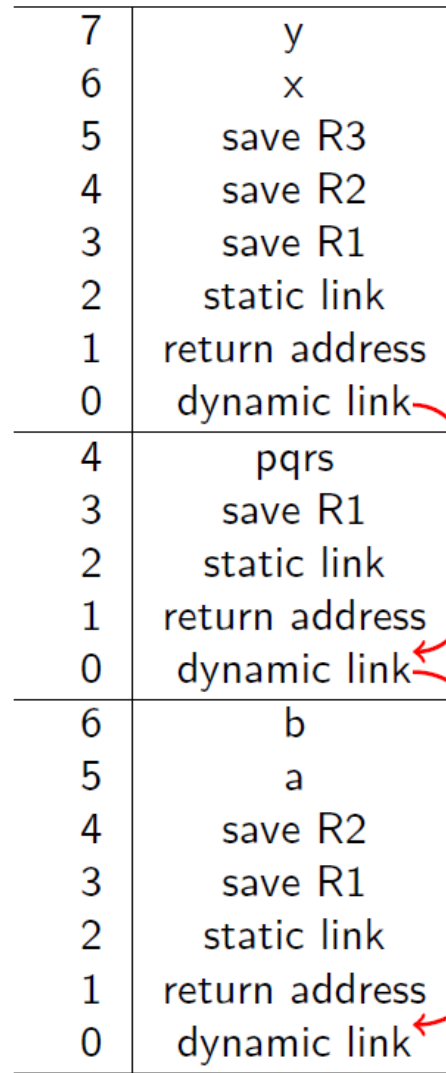
- The procedure keeps its local variables on the stack

# Static links for scoped variables

| | |
|---|---|
| 7 | y |
| 6 | x |
| 5 | save R3 |
| 4 | save R2 |
| 3 | save R1 |
| 2 | static link |
| 1 | return address |
| 0 | dynamic link |
| 4 | pqrs |
| 3 | save R1 |
| 2 | static link |
| 1 | return address |
| 0 | dynamic link |
| 6 | b |
| 5 | a |
| 4 | save R2 |
| 3 | save R1 |
| 2 | static link |
| 1 | return address |
| 0 | dynamic link |

# Accessing a word in the stack frame

- Work out a "map" showing the format of a stack frame

- Describe this in comments (similar to the register usage comments)

- Suppose local variable, say "avocado", is kept at position 7 in the stack frame

- To access the variable:
    - load R1,7[R14]                    ; R1 := avacado
    - store R1,7[R14]                  ; R1 := avacado
    - Also, we can dene the symbol "avacodo" to be 7, and write:
    - load R1,avacado[R14]          ; R1 := avacado
    - store R1,avacado[R14]        ; R1 := avacado

- These are called local variables because every call to a procedure has its own private copy

# Example from factorial program (see below)

- These comments document the structure of a stack frame for the program

```
; Structure of stack frame for fact function
;       6[R14]   origin of next frame
;       5[R14]   save R4
;       4[R14]   save R3
;       3[R14]   save R2
;       2[R14]   save R1 (parameter n)
;       1[R14]   return address
;       0[R14]   pointer to previous stack frame
```

Outline

- Retrospective

- Variables

  - Static variables

  - Local variables

  - Dynamic variables

- The call stack

- **Recursive factorial**

Recursive factorial

- In the Sigma16 examples, there is a program called factorial

- This program illustrates the full stack frame technique

- It uses recursion, i.e. a function that calls itself

- **Note**: the best way actually to compute a factorial is with a simple loop, not with recursion

- But recursion is an important technique, and it's better to study it with a simple example (like factorial) rather than a complicated "real world" example

About the factorial program

- Comments are used to identify the program, describe the algorithm, and document the data structures

- Blank lines and full-line comments organise the program into small sections

- The caller just uses jal to call the function

- The function is responsible for building the stack frame, saving and restoring registers

- The technique of using the stack for functions is general, and can be used for large scale programs

Statement of problem, and register usage

```
;-------------------------------------------------------------
; factorial.asm.txt
;-------------------------------------------------------------

; This program for the Sigma16 architecture uses a recursive function
; to compute x! (factorial of x), where x is defined as a static
; variable.

; The algorithm uses a recursive definition of factorial:
;    if n <=1
;       then factorial n = 1
;       else factorial n = n * factorial (n-1)

; Register usage
;    R15 is reserved by architecture for special instructions
;    R14 is stack pointer
;    R13 is return address
;    R2, R3, R4 are temporaries used by factorial function
;    R1 is function parameter and result
;    R0 is reserved by architecture for constant 0
```

Format of main program stack frame

```
;-----------------------------------------------------------------
; Main program

; The main program computes result := factorial x and terminates.

; Structure of stack frame for main program
;    1[R14]    origin of next frame
;    0[R14]    pointer to previous stack frame = nil
```

Main program initialisation

```
; Initialise stack
    lea    R14,stack[R0]        ; initialise stack pointer
    store  R0,0[R14]            ; previous frame pointer := nil
```

Main program calls factorial

```
; Call the function to compute factorial x
    load    R1,x[R0]            ; function parameter := x
    store   R14,1[R14]          ; point to current frame
    lea     R14,1[R14]          ; push stack frame
    jal     R13,factorial[R0]   ; R1 := factorial x
```

Main program finishes

```
; Save result and terminate
     store   R1,result[R0]          ; result := factorial x
     trap    R0,R0,R0               ; terminate
```

39

Description of factorial function

```
;-----------------------------------------------------------------
factorial
; Function that computes n!
;    Input parameter n is passed in R1
;    Result is returned in R1
```

Format of stack frame for factorial

```
; Structure of stack frame for fact function
;    6[R14]    origin of next frame
;    5[R14]    save R4
;    4[R14]    save R3
;    3[R14]    save R2
;    2[R14]    save R1 (parameter n)
;    1[R14]    return address
;    0[R14]    pointer to previous stack frame
```

Factorial: build stack frame

```
; Create stack frame
      store   R13,1[R14]          ; save return address
      store   R1,2[R14]           ; save R1
      store   R2,3[R14]           ; save R2
      store   R3,4[R14]           ; save R3
      store   R4,5[R14]           ; save R4
```

Factorial: check for base or recursion case

```
; Initialise
    lea     R2,1[R0]                ; R2 := 1

; Determine whether we have base case or recursion case
    cmpgt   R10,R1,R2               ; R10 := n>1
    jumpt   R10,recursion[R0]    ; if n>1 then go to recursion case
```

Factorial: base case

```
; Base case.  n<=1 so the result is 1
    lea    R1,1[R0]              ; factorial n = 1
    jump   return[R0]            ; go to end of function
```

Factorial: recursion case

```
; Recursion case.  n>1 so factorial n = n * factorial (n-1)
recursion
     sub     R1,R1,R2              ; function paramemter := n-1

; Call function to compute factorial (n-1)
     store   R14,6[R14]            ; point to current frame
     lea     R14,6[R14]            ; push stack frame
     jal     R13,factorial[R0]     ; R1 := factorial (n-1)
     load    R2,2[R14]             ; R2 := saved R1 = n
     mul     R1,R2,R1              ; R1 := n * fact (n-1)
```

Factorial: restore registers and return

```
; Restore registers and return; R1 contains result
return
        load    R2,3[R14]           ; restore R2
        load    R3,4[R14]           ; restore R3
        load    R4,5[R14]           ; restore R4
        load    R13,1[R14]          ; restore return address
        load    R14,0[R14]          ; pop stack frame
        jump    0[R13]              ; return
```
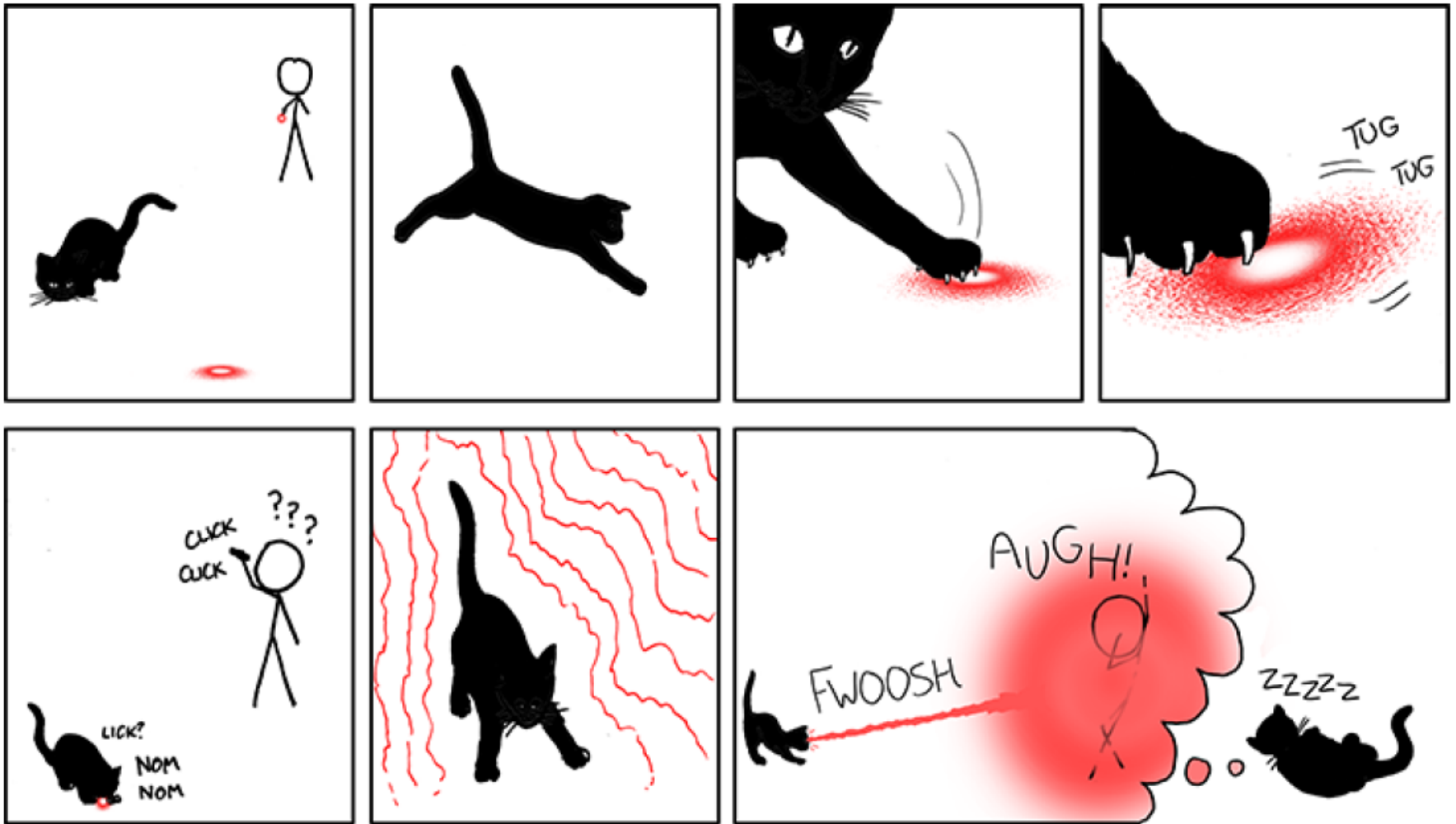
Static data area

```
;-------------------------------------------------------------
; Static data segment

x        data    5
result   data    0
stack    data    0    ; stack extends from here on...
```

Summary

- Variables defined with data statement are static
  - Each static variable must have a unique name
  - Static variables exist through entire execution of program

- Variables defined in a procedure are local
  - Different procedures can use the same name for different variables
  - Local variables are kept in the stack frame
  - Call (push stack frame); return (pop stack frame)
  - R14 points to current stack frame; local variables are accessed using R14

laser_pointer

https://xkcd.com/729/