Recursion - implementation

Revisit method calling:

Machine level implementation in the presence of *recursion*

CSCU9A2 Recursion 2

Recursion: Coping with *return addresses, parameters, local variables and results*

- When a method is called:
  - *Actual parameter values* must be transferred
  - The *return address* must be remembered
  - Storage is needed for local variables
  - A result might be transferred back (non-`void` method)
- We looked at a scheme for the Brookshear machine:
  - Fixed memory locations were used, associated with the method
- Note:
  - This is (like) how the earliest programming languages worked
  - It does not work *in general* – in particular for *recursion*
  - More advanced CPUs have more powerful instructions

CSCU9A2 Recursion 2

Summary of the earlier scheme...

- Here is how a (`void`) method call with one parameter might compile:

```
Addr  Instr
...
...   Compile param and store to [7E]
30    MOV 36 -> R0       Note return addr
32    MOV R0 -> [7F]     Save in known loc
34    JMPEQ 80, R0       Jump to method
36    ...
...
7E    00                 Reserved for parameter
7F    00                 Reserved for return addr
80    ...                Start of method
...
90    MOV [7F] -> R0     Retrieve return addr
92    MOV R0 -> [95]     Modify JMPEQ addr operand
94    JMPEQ 00, R0       Return jump
...
```

call

method

Could use 7D, 7C, etc for local variables

Could use, say, R1 to return a result

CSCU9A2 Recursion 2

Why this doesn't work

- In general:
  - Recursion: A method may call itself
    - either *directly* or *indirectly*
  - So *at any one time* there may be *several* return addresses to remember, and *several* (sets of) parameters
- Methods need to be "reentrant" (re-entrant)
  - It must be possible to start a new call *without damaging a previous unfinished call*
  - Each ***separate run time call*** must have ***its own*** parameter and local variable storage...
  - ...and return address
  - But OK to use, say, a register for the *return result* (- the calling code must save it immediately)
- Example on following slide...

CSCU9A2 Recursion 2

Example: memory needed for each *call*

- From the tutorial:

```
main...   display(4);

private void display(int n)
{
  if (n>0)
  {
     System.out.println(n);
     display(n-1);
     System.out.println(n);
  }
  else
     System.out.println(n);
}
```

- Every call needs its own return address
- Every execution needs its own `n` – so preserved for after the recursive call

CSCU9A2 Recursion 2

How to solve the problem

- Observation:
  - Method calling and returning happens in a very ordered way
  - All the method calls *within* a method body complete *before* the method itself returns: e.g

```
Step 1: paintScreen                                     (allocate storage: paintScreen)
Step 2: paintScreen → drawWindow                        (allocate storage: drawWindow)
Step 3: paintScreen → drawWindow → fillRect             (allocate storage / dealloc storage: fillRect)
Step 4: paintScreen → drawWindow
Step 5: paintScreen → drawWindow → drawRect             (allocate storage / dealloc storage: drawRect)
Step 6: paintScreen → drawWindow                        (dealloc storage: drawWindow)
Step 7: paintScreen
Step 8: paintScreen → drawWindow                        (allocate storage: drawWindow)
Step 9: paintScreen → drawWindow → fillRect             (allocate storage / dealloc storage: fillRect)
Step 10:paintScreen → drawWindow
Step 11:paintScreen → drawWindow → drawRect             (allocate storage / dealloc storage: drawRect)
Step 12:paintScreen → drawWindow                        (dealloc storage: drawWindow)
Step 13:paintScreen                                     (dealloc storage: paintScreen)
```

> *Storage*: *Space for parameters, local variables and return address*

CSCU9A2 Recursion 2

- The same with **display**:

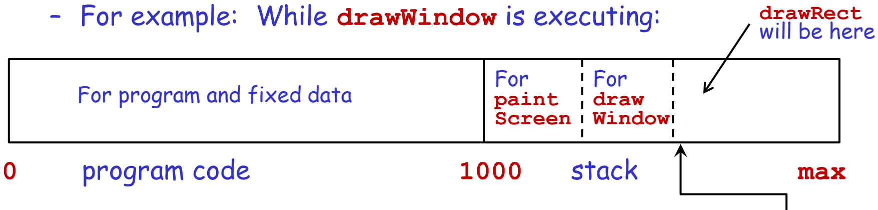
```
Step 1: display(2)
Step 2: display(2) → display(1)
Step 3: display(2) → display(1) → display(0)
Step 4: display(2) → display(1)
Step 5: display(2)
```

storage storage storage

- This matches what computer scientists call a "stack"
  - A *last-in-first-out* information management strategy (LIFO)
  - e.g. a restaurant plate stack, bangles on a wrist
- The scheme:
  - At method call: new storage block required
  - Executing body: use memory locations in *most recent block*
  - Return: use return address in *most recent block* & *discard the block*

Outline of solution using a stack

- A (large) block of memory (RAM) is allocated to hold the "call stack" or "run time stack" or just "the stack"*
  - Each method call has a "stack frame" (or "activation record")
  - To hold *all* parameters, local variables and return addresses
  - For example: While **drawWindow** is executing:



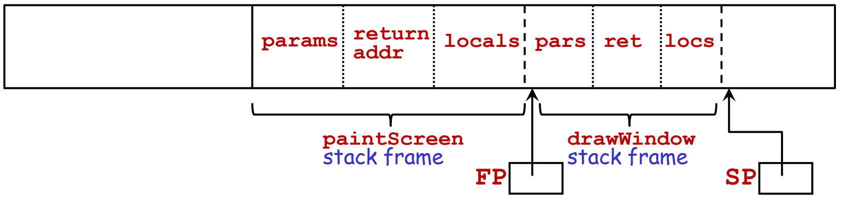
- One register (possibly a special one, like the PC) holds the address of the *next* "free" memory location
  - The "stack pointer" or SP
  - It "points" because it holds a memory address
  - It points where the *next call* should allocate memory

SP

* See Wikipedia: Call stack

Optional reading: more detail (intricate!)

- The section of the stack for a method call is usually called a "stack frame" (or "activation record")
  - Contains parameter values, local variables and return addr



- Often another register, the "frame pointer" FP, points at the "bottom" of the *current* stack frame
  - The addresses of parameters and variables are calculated *relative* to this pointer
  - It is where the SP has to be reset to when this method returns
  - The previous FP may be saved with the parameters - for restoration when this method returns

CSCU9A2 Recursion 2

9

Optional reading (cont): Call/return: possible mechanisms using a call stack

- Code to *call a method* (and post-process after return):
  - Copy SP to Rn (will be new FP)
  - Evaluate each actual param, and "push" onto the stack
  - Push the current FP
  - Copy Rn to FP (FP ready for new call)
  - Push the return address
  - Jump to the method start address
  - (After return) "Pop" the FP – to restore current frame
  - Subtract enough from SP to "discard" parameters
- Stack "push" means:
  - Save data where SP points, then increment SP
- Stack "pop" means:
  - Decrement SP, then fetch data from where SP points

Often one machine instr in modern CPUs

CSCU9A2 Recursion 2

10

- At the start of method body:
  - Push initial values for local variables onto stack
- Throughout method body:
  - Access parameters and local variables *relative* to the FP (next slide)
- Code for return from method:
  - Subtract enough from SP to "discard" local variables
  - Pop return address from the stack
  - Jump to return address

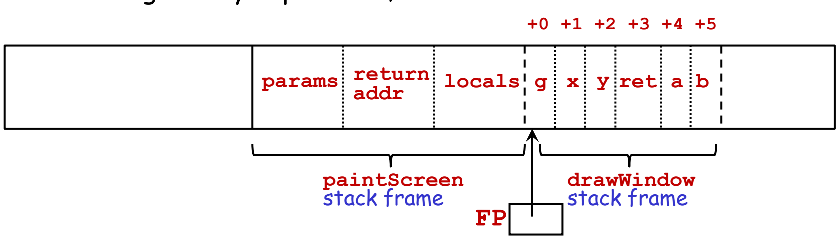
  Often one machine instr in modern CPUs
- The run time stack is very important:
  - Modern CPUs (since, say, c1970!) have special support:
  - SP register
  - Push and Pop instructions
  - Jump to / Return from subroutine instructions

CSCU9A2 Recursion 2

11

Optional reading (cont): One final problem

- With this scheme, ***method bodies can no longer have fixed memory addresses for accessing parameters and variables!***
- Fortunately, each is in a known location *relative* to the FP: eg, assuming one byte per item, as in Brookshear:



- So, addresses must be *calculated*: E.g. fetch from/store to **y**:
  - Load FP to Rn
  - Add +2 to Rn — Address of **y** is now in Rn
  - MOV [Rn] -> Rm — New Brookshear instructions:
  - MOV Rm -> [Rn] — "Indirect addressing"

CSCU9A2 Recursion 2

12

Optional reading (cont): Addressing modes in machine language

- "Addressing modes" determine the kinds of *operands* that may appear in machine instructions
  - See Wikipedia: Addressing mode
- The original Brookshear machine has:
  - Register: e.g. MOV 35 -> R1
  - Immediate/literal: e.g. MOV 35 -> R1
  - Absolute/direct: e.g. MOV [80] -> R1
- Now we have seen: (not in original Brookshear)
  - Register indirect: e.g. MOV [R1] -> R2
- And also usually available is: (not in original Brookshear)
  - "Base + offset"/indexed: e.g. MOV [R1]+2 -> R2
  - Which could be used to simplify the previous slide

CSCU9A2 Recursion 2

13

Summary

- To make recursion possible, programming languages use a *run time stack* in the RAM
  - One *stack frame* is allocated per method call...
  - ...containing parameter values, local variables, a return address, and necessary administration information
  - The stack frames are allocated on a last-in-first-out basis
- This scheme is used for *all method calls* – not specifically recursive methods
- The compiler generates method call/return code to manage the stack frames
- The scheme given here is quite realistic
  - Although precise details will vary with hardware and programming language

CSCU9A2 Recursion 2

14

End of lecture

CSCU9A2 Recursion 2