PPL 3/25

Reagan Shirk

March 25, 2020

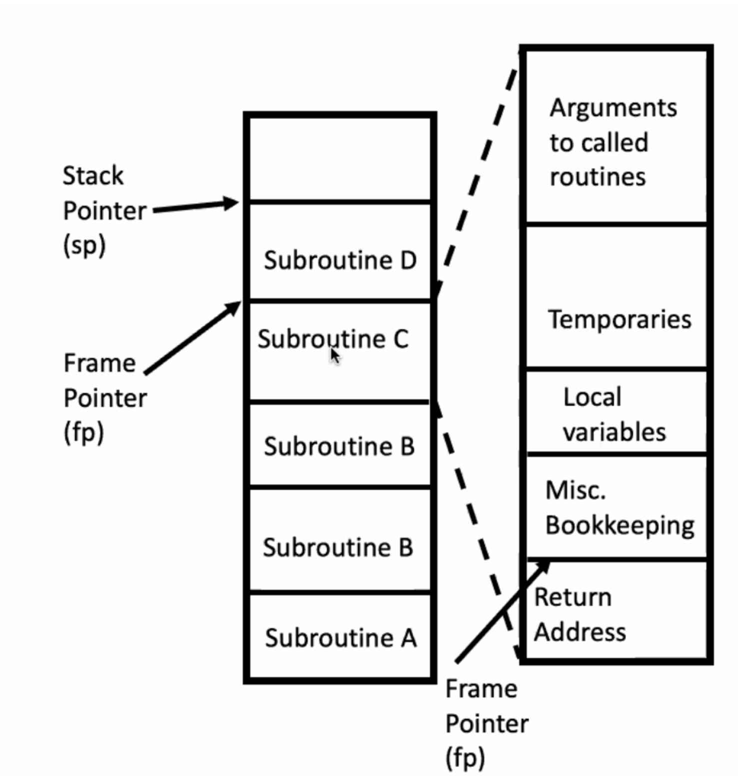
Stack Allocation

- Local variables are created when a subroutine is called and it is destroyed when it ends
- There's a global variable in the third assignment that does…something? with memory allocation
- Each subroutine call creates a new instance of each local variable
- Not all languages do this (or used to do this)
  - Fortran added it in 1990
  - Made it so that you couldn't have recursion
  - Made it so that there was no destinction between global and stack variables (subroutines still called in LIFO order though)
- Constants defined in subroutines can be statically stored
- When a constant is really a constant and doesn't depend on anything, you can store it….somewhere? The slides say you can allocate memory for a single instance of a constant and allow all of the calls of a subroutine to use it, but I didn't hear specifically say where that memory is allocated. The stack I guess lol
- Some languages initialize variables at runtime because it could depend on other variables. This means:
  - Constants are allowcated on the stack
- There are *true* constants and constants that require intilization (that makes more sense with everything else he just said)

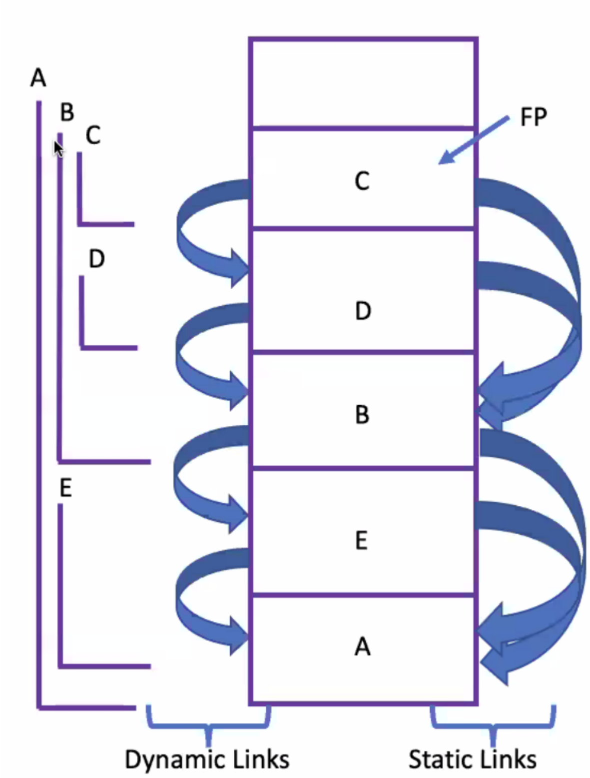
Stack-based Allocation

- Apparently different than Stack Allocation
- I love how he tells us that he's moving his mouse, as if we can't see it happening
- Each call of a subroutine assigns memory from the stack for the different variables and constants used in the subroutine
- Good to keep temporary and local variables separate
  - Temporary variables are determined by the compiler but I didn't really catch what they are (hopefully I'm not supposed to already know)
- The memory of a subroutine is allocated in a frame or an activation record
- The frame also has allocated memory for temporary variables (this is where separating temporary and local variables comes in I think)
- It's more convienent to allocated memory for function parameters at the top of the frame/activation record
  - You can access the parameters of the function by simple…something? I think he said by a simple offset
- The actual arguments of the subroutine are pushed onto the stack
- The layout of the memory is heavily dependent on the language and the implementation
- The actual address of the stack frame isn't able to be determined at compile time, but the offsets within the frame are

- The frame pointer knows where shit is, and you can access things externally using it. For example, if you have a variable “count” with an offset of 10, then you can access “count” with fp + 10
- The offset of variables can be used to both load and store instruction variants
- The offset is computed statistically by the compiler with datatype information (i.e. a char is 1 byte, etc)



- A static link is a pointer to the stack frame which corresponds to the lexically surrounding subroutine
  - relevant in programming languages that actually support nested declarations (nothing we’ve worked with before)
- A dynamic link is a pointer to a stack frame that makes the call to the current subroutine
- In the graphic below, we can see examples of lexically surrounding subroutines
  - A is a lexically surrounding subroutine for E because E is declared directly inside of A
  - A is *not* a lexically surrounding subroutine for C or D. This is because B needs to be active in order for C and D to run, so B is the lexically surrounding subroutine for C and D.



Maintaining the Static Link

- Wow I zoned out hardcore. I don't even know how, my phone is off and I wasn't looking at anything on my computer. Oops. Back to the topic at hand
- Languages with nested subroutines require a little extra work to maintain the static link
- Most work falls on the caller side (there is a caller and a callee)
- The work depends on the nesting of the callee
  - If the callee is nested inside the caller, the caller passes itself as the static link of the callee
  - If the callee is $k \geq 0$ scopes outward of the caller, there are two possibilities:
    * Scopes surrounding the callee also surround the caller
    * The caller dereferences on its own static link k times and passes the result as the static link of the callee

Calling Sequence

- Only fifteen more minutes… will I survive?
- There's kind of a grid structure here, you have the prologue of the caller and callee and the epilogue of the caller and callee
- Caller Prologue:
  - Save registers
  - Compute argument values and move them onto the stack or the registers
  - If the language supports nested subroutines, you have to compute the static link and pass it as a hidden object
    * He said it's normally some time of attribute but I didn't hear what kind
  - Use a special instruction to jump to the address start of a subroutine

    * This includes saving the return address in the stack or in a register

- Callee Prologue:
  - This allocates a frame by decrementing some constant offset from the stack pointer
  - It also saves the old frame pointer into the stack and updates to the newly allocated frame
- Caller Epilogue
  - Nine more minutes…
  - This moves the returned values to wherever they need to be
  - This also restores the pending registers
- Callee Epilogue
  - If there is a return value, move it to a register or a specific address given by the caller
  - Restore the register values from the caller
  - Jump to the return address

Heap Based Allocation

- The heap is a program segment that the program can grab memory from during its execution
- It requires allocation and deallocation of memory
- Dynamic allocation is generally a performance killer
  - This is because you have to jump from user mode to kernel mode which is slow
- Oops I dozed off