COMP124

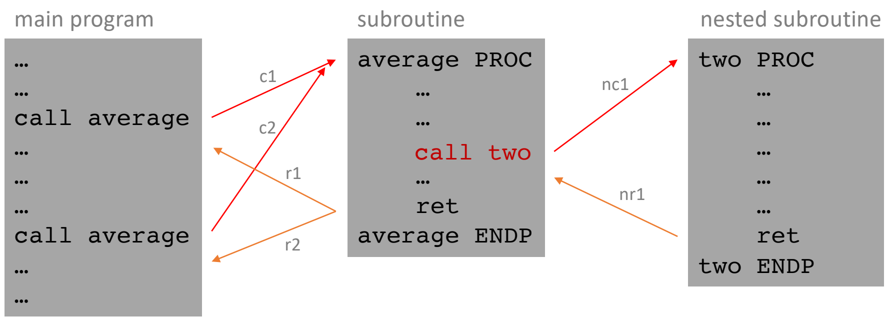
Computer Systems

06 | Stacks | Parameters

Dr Stuart Thomason

Nested Calls

- We could use a special register in the CPU to store the return address
  - When we need to return, just copy that register into the instruction pointer
  - But then how would we handle nested calls?
- Most reasonably complex programs use multiple subroutines that could call each other in a nested manner
- As soon as the second call happens, the return address for the first call is overwritten



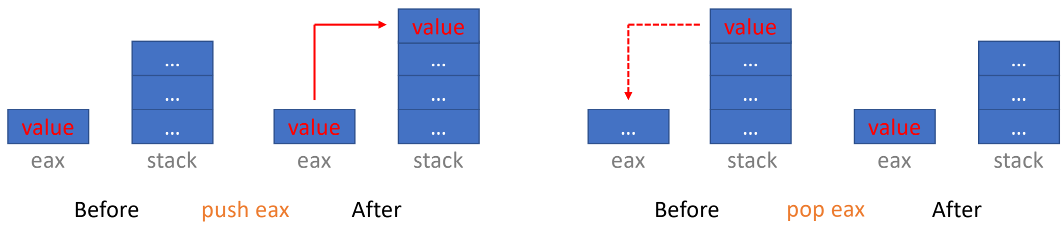
Storing Return Address

- We need a way to store an arbitrary number of nested return addresses
  - Can't use CPU registers because we can't predict how many we'll need
  - Deeper levels of nesting will need more storage
  - So we must store them in main memory

- The stack is an obvious choice of data structure for this storage
  - Uses main memory (RAM) of the current process
  - Treated separately and accessed directly through CPU registers

The Stack

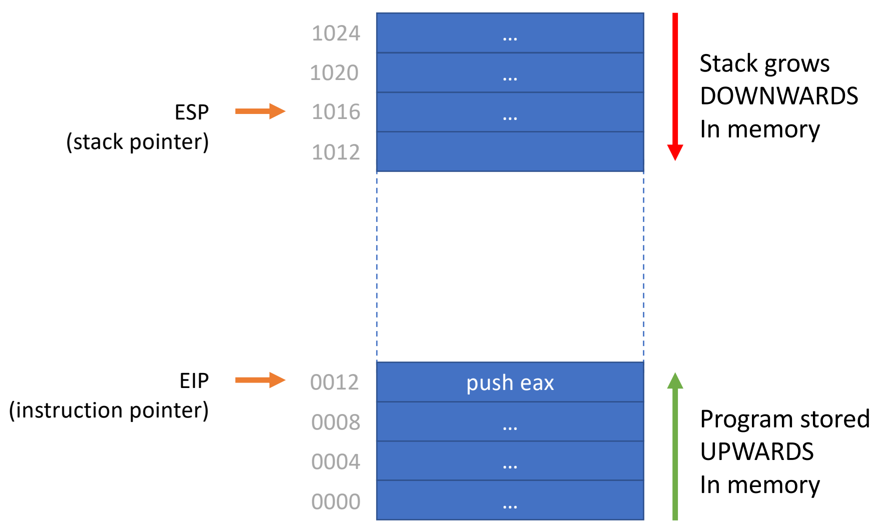
- Stacks exist in high level programming as a useful data structure
  - Stores and retrieves values in a specific order
  - LIFO – last in, first out
  - For comparison, a queue is FIFO (first in, first out)
- Every stack has two operations (push and pop)
- Imagine this is like putting and taking cards from a pile on a table



Machine Level Stacks

- The Intel x86 architecture uses main memory for the stack but accesses it via a register
  - ESP – stack pointer (32-bit name)
  - Always points to memory address of top item on the stack
- Important: the stack grows downwards in memory
- The push instruction…
  - Decrements ESP so it points to next free area of memory on the stack
  - Writes the data item to that address (ie. to the top of the stack)
- The pop instruction…
  - Moves the data addressed by ESP into the given register
  - Increments ESP by the correct amount to remove the item from the stack (but note that the data stays in memory until overwritten)
- The programmer must take care to tidy the stack and ensure items are removed when no longer needed (in general, pop whatever you pushed)

Memory Layout



Call and Return

- Now we can define exactly what the subroutine call and return mechanism looks like

- The call instruction…
  - Takes current value of instruction pointer and pushes it onto the stack
  - Puts the address of subroutine into instruction pointer
  - Next instruction to be fetched will be the first instruction in the subroutine

- The ret instruction…
  - Pops top item off stack and places it into the instruction pointer
  - Next instruction to be fetched will be the instruction after original call instruction

- Nested subroutines will return properly because the top-most return address (on stack) will always be from the most recent call instruction

Manipulating the Stack Pointer

- We can change the value of ESP directly from our code
  - Must be careful to keep track of all the pushes and pops so they match up
  - Can quickly clean stack of multiple bytes if we don't need the data via pops
  - We often do this after pushing parameters for a subroutine call (see later)

- To take 8 bytes off the stack (remember it grows downwards)

```
add esp, 8
```

- Or to reserve some space on the stack for 256 bytes

```
sub esp, 256
```

Subroutine Parameters

- Most subroutines require parameters (but occasionally they don't)
- In many cases the parameters for a subroutine will be simple data (eg. numbers)

- Consider a subroutine that takes two integer parameters and returns the biggest
- In a high level language this is easy to specify with appropriate syntax

```
int bigg(int first, int second) { … }
```

- But in assembly language we don't have the luxury of any syntax
- We need to make some decisions
  - Where do we store the parameters?
  - How do we pass the parameters?
  - Where do we put the return value?

Pass by Value

- A simple subroutine like this uses pass by value (parameters are copied into registers)

```
bigg PROC
     cmp eax, ebx
     jl snd
     ret
snd: mov eax, ebx
     ret
bigg ENDP

…
mov eax, first
mov ebx, second
call bigg
mov max, eax
…
```

- This depends on the caller and the callee agreeing on which registers to use

Pass by Reference

- Consider a subroutine that swaps two variables (exchanges values in memory)
- Knowing the value of each variable isn't enough (and making a copy wouldn't work)
- We need to know their memory addresses so we can make the swap

```
swap PROC
    mov temp, [eax]
    mov [eax], [ebx]
    mov [ebx], temp
    ret
swap ENDP

…
lea eax, first
lea ebx, second
call swap
…
```

- The caller and the callee still need to agree on which registers to use

Stacking Parameters

- Note that Intel x86 has an instruction that can swap values inside two registers

```
xchg eax, ebx
```

- One operand can be a memory label, but this is slow due to locking (see concurrency lectures)

- For subroutines with a few parameters, using registers is fine
  - But there are only a few registers (might already be in use for other data)
  - Caller and callee need to agree on which registers to use
- A better way to pass parameters is to push them to the stack
  - Caller pushes parameters before making the call
  - Callee pops (or accesses) parameters and does what it needs to do
  - Caller tidies up stack after subroutine returns
  - Both caller and callee still need to agree on the order of parameters and stack use

Parameter Stack Example

- Consider a subroutine to work out the area of a square

```
area PROC
    pop ebx
    pop eax
    mov temp, eax
    pop eax
    mul temp
    push ebx
    ret
area ENDP

…
push width
push height
call area
mov result, eax
…
```

Calling Convention

- The caller and callee need to agree a calling convention
- In the previous example…
  - Caller places two items onto the stack
  - Callee removes those items during execution
  - Callee places return result into accumulator

- Notice the extra pop (at the start) and push (at the end) inside the subroutine
  - The call instruction pushes the return address to the stack
  - So this will be on top of the parameters
  - Subroutine needs to remove it and keep it safe so it can access parameters below
  - Then put it back on the stack before it returns
  - If you forget to save the return address, you will get undefined results (probably a crash, because the return will go to a random memory address)

Intel x86 Calling Conventions

- The Intel x86 (32-bit) architecture defines four calling conventions
  - cdecl – Pushes parameters on stack in reverse order (right to left); caller cleans stack
  - fastcall – First two parameters in registers, rest reversed on stack; callee cleans stack
  - stdcall – Pushes parameters on stack in reverse order; callee cleans stack
  - thiscall – First parameter in ECX, rest reversed on stack; callee cleans stack

- The previous example (area subroutine) used the stdcall convention
  - Parameters were pushed (order didn't matter)
  - Subroutine removed them and cleaned stack before it returned

- In future lectures we will call external C libraries to help with input and output
  - Library routines expect us to use the cdecl convention (C declaration)
  - External code is a 'black box' so we don't know which registers it uses or modifies