Computer Architecture
Machine Representation of Programs: Procedures 1

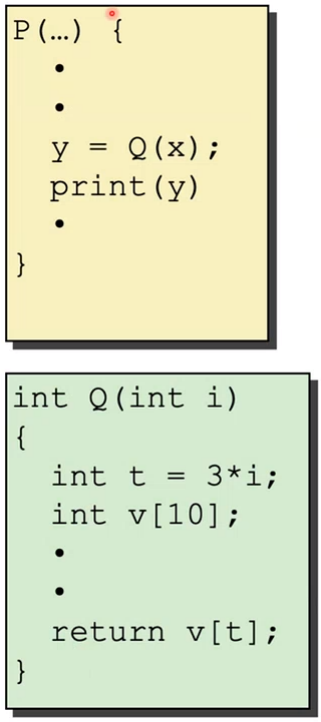
Cain Susko

Queen's University
School of Computing

February 16, 2022

Procedures

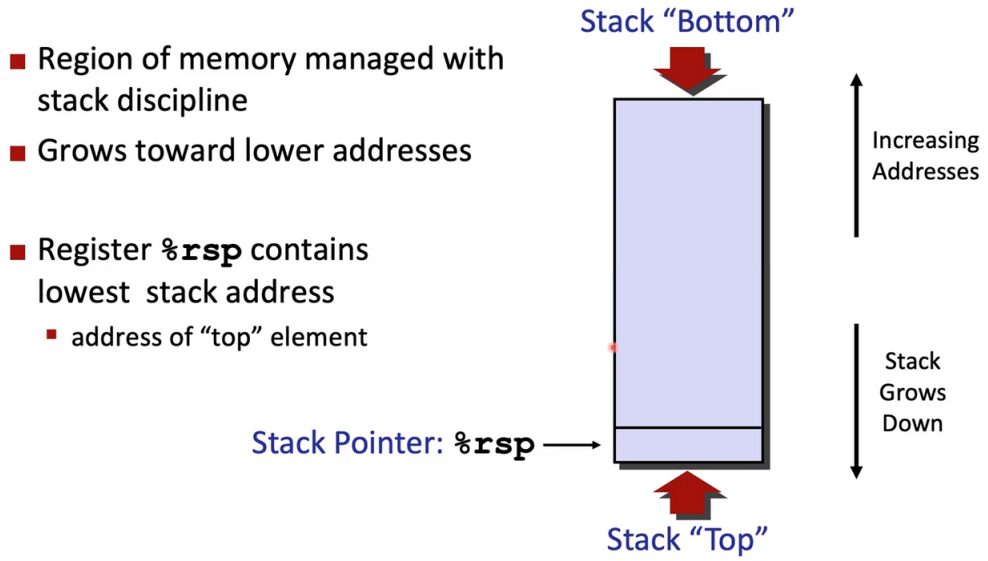
a procedure is a list of instructions that allow a computer/CPU to do various complex tasks. They are also known as functions which alludes to their manifestation; code!



There can be many procedures that a computer can run, thus there is also a passing of control between procedures. This passing of control also tends to pass data (ie. Input parameter, return variable)

Stack Structure

The stack structure is dependent on the architecture and operating system being used. The x84-64 stack is defined as:

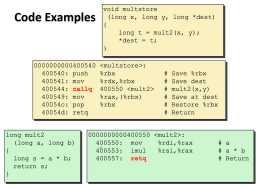


With the stack we can read and write to it using pust and pop:

`pushq Src` Fetch operand at `Src`. Decrement `%rsp` by 8. Write operand to address given by `%rsp`

`popq dest` Read value at address given by `%rsp`, incrament `%rsp` by 8 (because we are using a 64 bit architecture) and store the value at `dest` (Note: *dest* must be a register)

Whithin the context of procedures, the stack within the architecture allows for passing of control between procedures. For example:



Procedure Control Flow

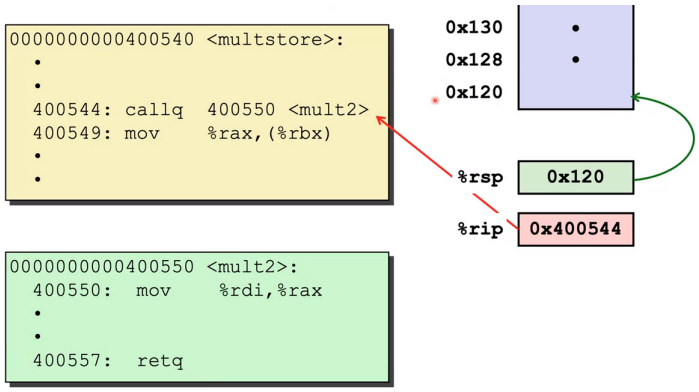
We use the stack to support procedure *call* and *return*.

`call label` pushes return address onto stack and jumps to label.

- Return Address: address of next instruction right after call.

`ret` pop (returns) address from stack and jumps to address.

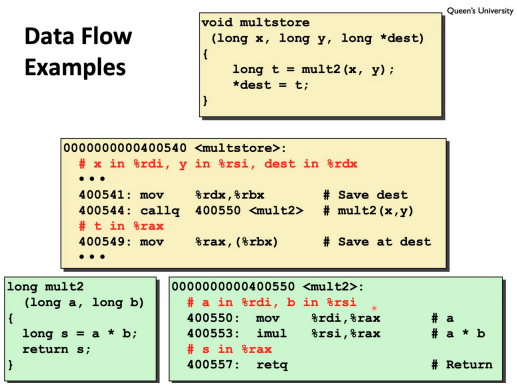
Below is an example of a stack control flow in x86-64 architecture:



- first, our pointer on the stack $\%rsp$ and the program counter $\%rip$. The next step is to move to a new procedure.
- $\%rip$ is now pointing to the first part of the green procedure and $\%rsp$ pushed the address of the previous place onto the stack
- as $\%rip$ moves along the new function it reaches the return call, where it then gets the value from the stack by popping the top of the stack into the program counter.

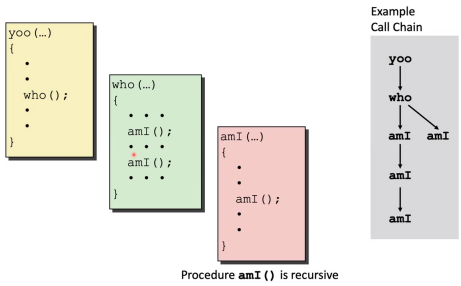
Procedure Data Flow

Data flow is achieved by using registers for the first 6 arguments of a procedure and 1 for the return value. The stack holds all remaining arguments.
An example of Data Flow with less than or equal to 6 arguments:



Managing Local Data

Most modern Languages support recursion or are 'reentrant'. Becuase of this, the languages need a place to store the multiple concurrent instances of a procedure.



Within the stack, the state for a given procedure is needed only for a limited time: from when it is called to when it is returned. Additionally, the callee returns before the caller does. These are a part of whats known as stack discipline.
Finally, the stack is allocated in *frames* where each frame is a state for a single procedure instantiation.

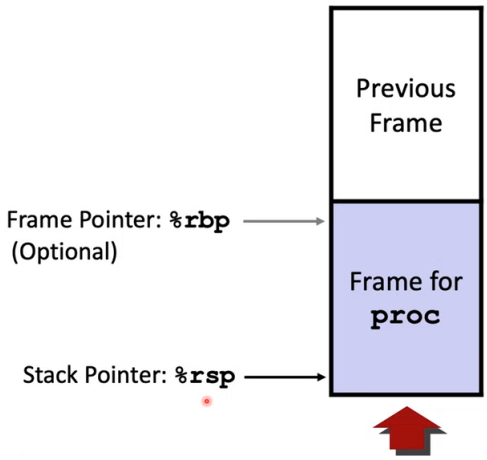
Stack Frame Each stack frame contains space that we need for temporary storage for local variables.

- return information
- local storage
- temporary space

The computer must also manage these frames (obviously).

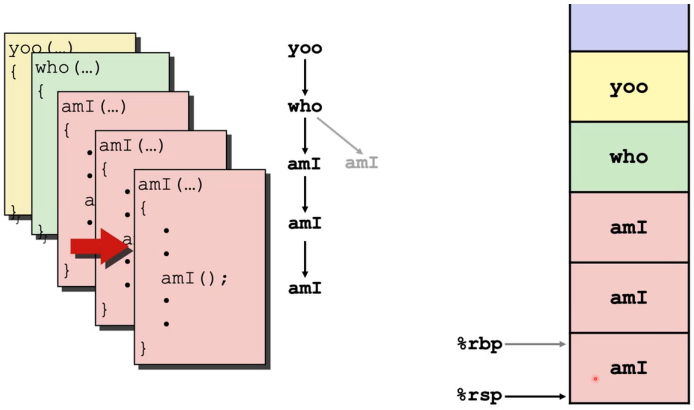
- space is allocated when entering a procedure. Manifests as setup code with a push by `call` instruction
- space is deallocated on return. Manifests as finishing code which includes a pop by `ret` instruction

With reference to the previous representation of the stack, frames are stored using an (optional) frame pointer: commonly, `%rbp`.

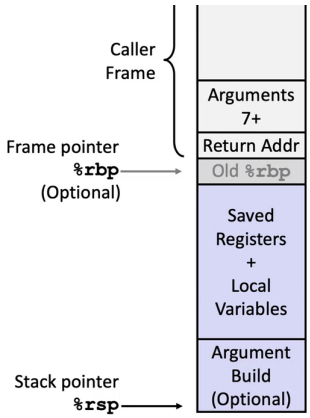


Note: all variables for `proc` are within the frame, between `%rbp & %rsp`

For many functions being called or cany instances of the same function being called, the stack and it's frames would look like so:



x86 Linux Within the x86-64 architecture on a linux system, a stack is structured like so:



Where:

- the current stack frame contains:
  - Argument Build - parameters for function that is about to be called.
  - Local Varibles - if unable to keep in registers
  - Saved Register Context

  - Old Frame Pointer - optional

- the caller stack frame contains:
  - Return Address - pushed by `call` instruction.
  - Arguments for this Call

An example of this local data management is that of the `incr` function, which incraments a variable.

```
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

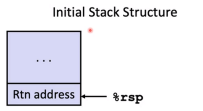
```
incr:
  movq    (%rdi), %rax
  addq    %rax, %rsi
  movq    %rsi, (%rdi)
  ret
```

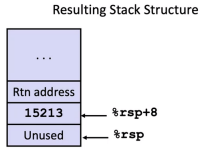
Register	Use(s)
%rdi	Argument p
%rsi	Argument val, y
%rax	x, Return value

The implementation of this function is:

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

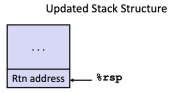
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```





And as the program progresses, the stack and it's frames are updated as needed.

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```




```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Register	Use(s)
%rax	Return value