Computer Architecture (Practical Class)
Assembly: Stack and Procedure Calls

Luís Nogueira

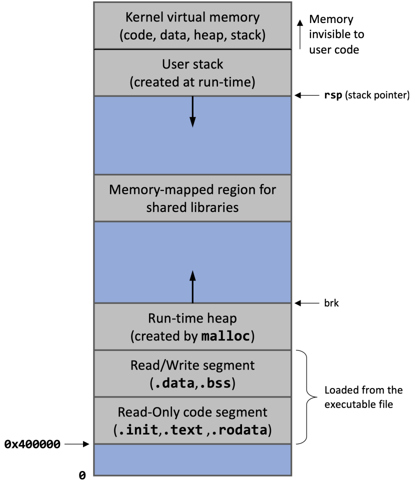
Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

lmn@isep.ipp.pt

2022/2023

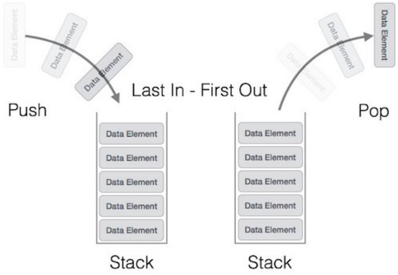
x86-64 Linux memory layout

- Memory viewed as array of bytes
- The range of virtual addresses that is available to a program is called its *virtual address space*
- Different regions have different purposes
- Heap and stack expand and contract dynamically at run time
- Unlike the code (text) and data areas, which are fixed in size once the program begins executing

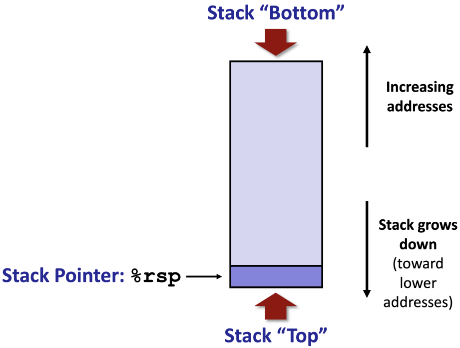


A generic stack in software

- A structure of type LIFO (Last In/First Out)
- A new element can only be added to the top, becoming the new top
- Existing elements can only be removed from the top



- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%rsp` indicates lowest allocated position on the stack (i.e., address of top element)



The PUSH and POP instructions

`pushq source`

- Decrements `%rsp` by 8 bytes
- Writes `source` at address given by `%rsp`

`popq destination`

- Reads the value at address given by `%rsp` and writes it to `destination` (usually a register)
- Increments `%rsp` by 8 bytes

Important notes

- PUSH and POP accept 16- and 64-bit operands
- Therefore, only the `w` and `q` variants can be used
- The RSP register is updated by 2 or 8 bytes, respectively

Illustration of stack operation

Initially

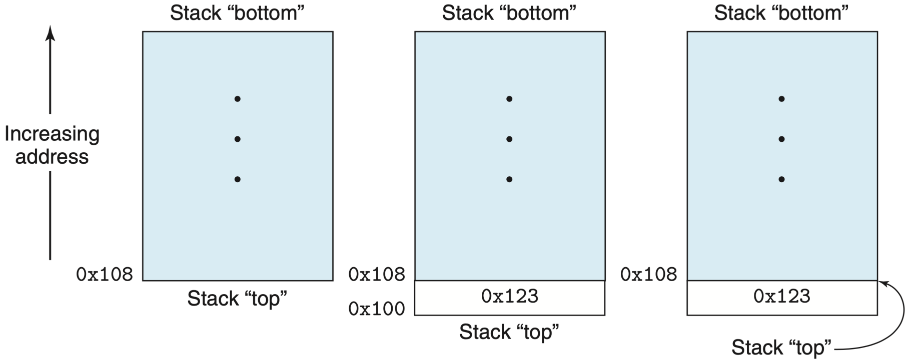
%rax	0x123
%rdx	0
%rsp	0x108

`pushq %rax`

%rax	0x123
%rdx	0
%rsp	0x100

`popq %rdx`

%rax	0x123
%rdx	0x123
%rsp	0x108



We can easily store in, and retrieve values from, memory without the need to explicitly handle memory addressing

PUSH/POP Examples

```
.section .data
x:
        .int -125

.section .text
.global test_stack
test_stack:
    movl   $24420, %ecx
    movw   $350, %dx
    pushq  %rcx
    pushw  %dx
    pushq  x(%rip)            # sets the 4 most significant bytes to 0
    leaq   x(%rip), %rdi
    pushq  %rdi
    popq   %rsi
    popq   %rax               # the value of x is in %eax
    popw   %dx
    popq   %rcx
    ret
```

Why is the stack so important?

- x86-64 programs require the use of the stack to:
  - Support procedure calls and return
  - Temporary data storage
  - Store local variables (next practical class)
  - Store parameters of a procedure about to call (next practical class)

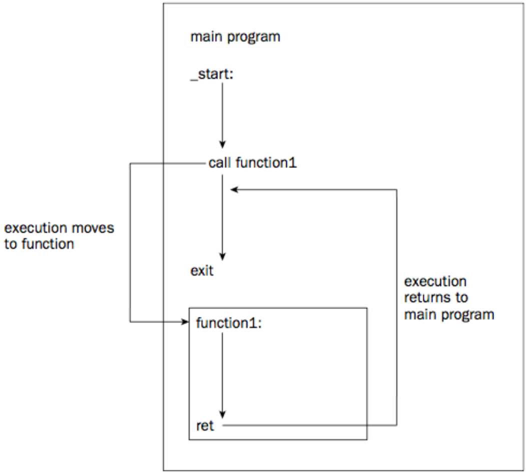
- The stack is allocated in `frames` (more on this on the next pratical class)
  - State for a single procedure instantiation
  - Stack pointer `%rsp` indicates stack top
  - Frame pointer `%rbp` indicates start of current frame

- Identified by a label
- Invoked through the `call` instruction
- End with the `ret` instruction
- The return value must be stored in the `%rax` register (or parts of it) before the execution of the `ret` instruction

Procedure Call Example

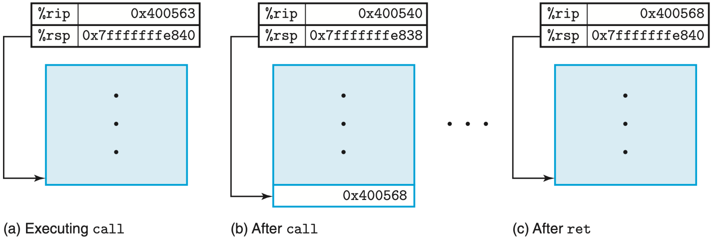
```
procedure2:
        ...
        ret

procedure1:
        ...
        call procedure2
        ...
```

Procedure calls: Execution flow



Transfer of Control

- The `call` instruction has only one parameter
  - The label that identifies the procedure to be called (the label is converted into the corresponding memory address)
- The `ret` instruction allows the execution flow to continue in the instruction immediately after the procedure call
- However, the `ret` instruction does not have any argument
- How does the `ret` instruction retrieve the correct return address?



Procedure call: **call** *label*

- Push return address on stack (address of the instruction following the call)
- Jump (*jmp*) to *label* (sets `%rip` to the corresponding address)

Procedure return: **ret**

- Pop address from stack
- Jump (*jmp*) to address
- It is up to the programmer to ensure that `%rsp` is pointing to the address pushed by *call*, prior to issuing a RET instruction

- When an x86-64 procedure requires storage beyond what it can hold in registers, it allocates space on the stack
- It is the most adequate place to save the current value of a register for later restauration

Save and Restore a Register

```
...
pushq %rdx          # save current value of %rdx

# divide %edx:%eax by %ecx
movl $0, %edx       # dividend
movl $1000, %eax    # dividend
movl $500, %ecx     # divisor
divl %ecx           # unsigned division
...                 # remainder on %edx

popq %rdx           # restore old value %rdx
...
```

Saving and restoring register state (2/2)

- Whenever a procedure p1 calls another procedure p2, it is possible that p2 uses some or all of the registers being used in p1

Save and Restore a Register

```
p1:
   ...
   movq $10, %rcx
   ...
   call p2          # can overwrite %rcx
   cmpq %rax, %rcx  # there is no guarantee on the current content of %ecx
   ...
```

- Therefore, there is no guarantee that those registers keep their contents after calling *p2*

- Although only one procedure can be active at a given time, we must make sure that when one procedure (the caller) calls another (the callee), the callee does not overwrite some register value that the caller planned to use later

Register saving conventions in x86-64 Linux

For efficiency sake, there is a convention that determines which registers should be saved prior and restored after a call

"Caller Save"

- Caller saves the current contents of a set of registers prior to call
- After the procedure return, it restores those registers to their old values

"Callee Save"

- Calle saves the current contents of another set of registers before using them
- Before returning, it restores those registers to their old values

Important note

It is particularly important to follow this convention every time a C function is invoked from your Assembly code (or any other function that was not written by you)

Caller-saved registers

Caller is responsible for their management

- If it uses them after the call
- Therefore, they can be modified by the callee

%rax

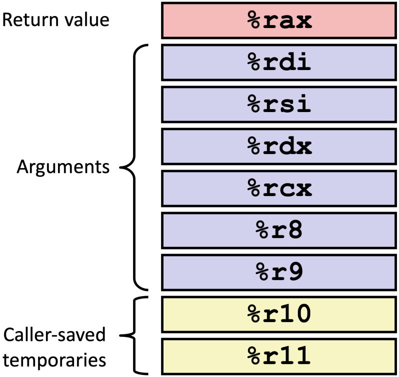
- Return value

%rdi, ..., %r9

- Arguments of function about to call
- More on this on the next pratical class

%r10, %r11

- Temporaries



Callee must preserve their value

- By either not changing it at all
- By pushing the original value on the stack, altering it, and then popping the old value from the stack before returning

%rbx, %r12, ..., %r15

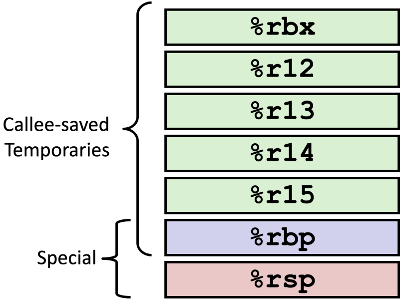
- Temporaries

%rbp

- May be used as a frame pointer
- More on this on the next pratical class

%rsp

- Stack pointer
- Restored to original value upon exit from procedure



- Assume that `a`, `b` and `c` are global variables of type *long*

Function f1

```
long f1(){
        return a + b + f2();
}
```

Function f2

```
long f2(){
        return c + 1;
}
```

- How can you implement an equivalent code in Assembly?

Function f1

```
f1:
  # f1 is called by other functions
  # (e.g., main in C)
  # save %rbx on stack
  pushq %rbx

  movq a(%rip), %rdx
  movq b(%rip), %rbx

  # caller is responsible for %rdx
  # save only those that are used
  # after the call
  # saves %rdx on stack
  pushq %rdx
  call f2
  # restore %rdx
  popq %rdx

  addq %rdx, %rbx
  addq %rbx, %rax

  # restore %rbx before returning
  popq %rbx
  ret
```

Function f2

```
f2:
  # callee is responsible for %rbx
  # save only those that are used
  # save %rbx on stack
  pushq %rbx

  movq c(%rip), %rbx
  incq %rbx
  movq %rbx, %rax

  # restore %rbx before returning
  popq %rbx
  ret
```

Procedures in Assembly: Summary

In caller, before invoking another procedure with `call`:

- Save to stack the current contents of caller-saved registers, if they are intended to be used after the return of the callee procedure with their values unchanged

In callee, before changing the contents of callee-saved registers:

- Save to stack the current contents of those registers

In callee, before returning with `ret`:

- Restore the old values of used callee-saved registers
- Place in `%rax` the intended return value
- Ensure that the value at the top of the stack is the return address pushed by `call`

In caller, after the return of the callee:

- Use, if necessary, the return value of the callee procedure in `%rax`
- Restore the old values of used caller-saved registers

Practice

- Implement, in Assembly, a function `int power()` where a variable `base` is raised to a fixed `exponent`. Test it by calling your function from a C program.
- Implement another function in Assembly, `int sum_powers()`, that using the previous function, traverses an array `int nums[10]` and returns the sum of the powers of each of its elements, as a base, raised always to the same exponent
- Consider that the variables `base`, `exponent`, and the array `nums` are declared in C. Iteratively, populate the `base` variable with each of the elements of `nums`

main.c

```c
#include <stdio.h>
#include "power.h"

int nums[]={1,2,3,4,5,6,7,8,9,10};
int size = sizeof(nums)/sizeof(nums[0]);
int base;
unsigned int exponent = 2;

int main(){

  int res,sum;

  base = 10;
  res = power();
  printf("base: %d and exponent: %u = %d\n",base,exponent,res);

  sum = sum_powers();
  printf("\nSum of powers of %u = %d\n",exponent,sum);

  return 0;
}
```

Practice: Assembly source - `int power()`

power.S

```
        .global power
power:
    movl base(%rip), %edi         # base on %edi
    movl exponent(%rip), %ecx     # exponent on %ecx
    movl $1, %eax                 # initial result in %eax

power_loop_start:
    cmpl $0, %ecx                 # if exponenent is 0, go to end_power
    je end_power
    imull %edi, %eax              # multiplies current result by base
    decl %ecx                     # decreases the exponent
    jmp power_loop_start          # jumps to next exponent

end_power:
    ret
```

sum_powers.S

```
        .global sum_powers
sum_powers:
        leaq nums(%rip), %rdi    # address of nums on %rdi
        movl size(%rip), %ecx    # number of elements in nums
        movl $0, %edx            # tmp = 0
        cmpq $0, %rcx
        jle end_sum
traverse_vec:
        movl (%rdi), %esi        # %esi = num[i]
        movl %esi, base(%rip)    # base = %esi

        pushq %rdi               # caller saves %rdi before call
        pushq %rcx               # caller saves %rcx before call
        pushq %rdx               # caller saves %rdx before call
        call power
        popq %rdx                # caller restores %rdx after power return
        popq %rcx                # caller restores %rcx after power return
        popq %rdi                # caller restores %rdi after power return

        addl %eax, %edx          # add return of power to tmp
        addq $4, %rdi            # point to next element in nums
        loop traverse_vec
end_sum:
        movl %edx,%eax           # final result in %eax
        ret
```