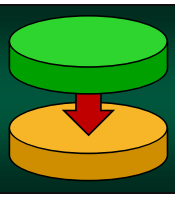
Subroutines & Operating Systems

Part 8

1



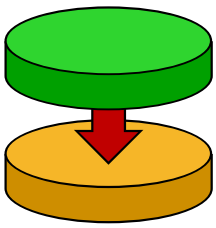
The System Stack

Pile of… Data

2

The System Stack

- The processor maintains a stack in memory
- It allows *subroutines*
  - analogous to the "functions" you use in Java and other third-generation languages
  - but, much more simple



3

Examples of Stacks

- Page-visited "back button" history in a web browser
- Undo sequence in a text editor
- Deck of cards in Windows Solitaire

4

Implementing in Memory

- On a processor, the stack stores integers
  - size of the integer the bit-size of the system
  - 64-bit system → 64-bit integer
- Stacks is stored in memory
  - A fixed location pointer (S0) defines the bottom of the stack
  - A *stack pointer* (SP) gives the location of the top of the stack

5

Approaches

- Growing upwards
  - Bottom Pointer (S0) is the *lowest* address in the stack buffer
  - stack grows towards *higher* addresses
- Grow downwards
  - Bottom Pointer (S0) is the *highest* address in the stack buffer
  - stack grows towards *lower* addresses

6

Size of the Stack

- As an abstract data structure…
  - stacks are assumed to be infinitely deep
  - so, an arbitrary amount of data can be stored
- However…
  - stacks are implemented using memory buffers
  - which are finite in size
- If the data exceeds the allocated space, a *stack overflow* error occurs

7

Subroutine Call Basics

Organizing Your Program

8

Subroutine Call

- The stack is essential for subroutines to work
- How?
  - used to save the return addresses for call instructions
  - backup and restore registers
  - pass data between subroutines

9

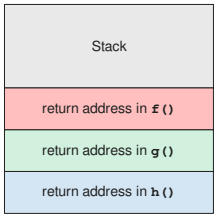
When you call a subroutine…

1. Processor pushes the instruction pointer (IP) – an address – on the stack
2. IP is set to the address of the subroutine
3. Subroutine executes and ends with a "return" instruction
4. Processor pops & restores the original IP
5. Execution continues after the initial call

10

Nesting is Possible

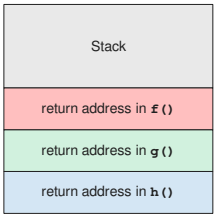
- Subroutines can call other subroutines
- `f()` calls `g()` which then calls `h()`, etc…
- The stack stores the return addresses of the callers
- Just like the "history button" in your web browser, you can store many return addresses



11

Nesting is Possible

- Each time a subroutine completes, the processor pops the top of the stack
- …then returns to the *caller*
- This allows normal function calls and recursion (a powerful tool)



12

x64 Subroutines

Organizing Your Programs … with Intel

13

Instruction: Call

- The *Call Instruction* transfers control to a subroutine
- Other processors call it different names such as JSR (Jump Subroutine)
- The stack is used to save the current IP

14

Instruction: Call

Usually, a label (which is an address)

```
CALL address
```


15

Instruction: Return

- The Return Instruction is used mark the end of subroutine
- When the instruction is executed…
  - the old instruction pointer is read from the system stack
  - the current instruction pointer is updated – restoring execution after the initial call

16

Instruction: Return

- Do not forget this!
- If you do…
  - execution will simply continue, in memory, until a return instruction is encountered
  - often is can run past the end of your program
  - …and run data!

17

Instruction: Return

No arguments!

```
RET
```


18

Subroutine Example

```
Main:
    mov  rcx, 4
    mov  rbx, 12
    call AddIt
    add  rbx, 1
    ...
AddIt:
    add  rbx, rcx
    ret
```


19

Saving Registers & Lost Data

Making subroutines clean

20

Saving Registers & Lost Data

- Each subroutine will use the registers as it needs
- So, when a subroutine is called, *it may modify the caller's registers*

21

Subroutine Example

```
StingersUp:
    .ascii "Stingers Up!\n\0"

 ...
Main:
    mov  rdx, 1947
    call Hornet
    call PrintInteger
```

We will create a subroutine called Hornet to print this

22

```
Main:
    mov  rdx, 1947
    call Hornet
    call PrintInteger
    ...

Hornet:
    lea  rdx, StingersUp
    call PrintString
    ret
```

We want to print 1947 after "Stingers Up!"

Hornet is called before PrintInteger

But, rdx is changed here

23

Subroutine Wrong Output

```
Stingers Up!
4202698
```

That's not 1947. The subroutine changed rdx to an address.

24

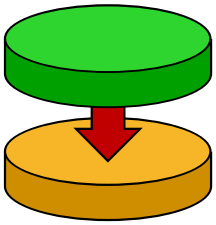
Saving Registers & Lost Data

- Some processors have few registers – this is very likely
- This can lead to hard-to-fix bugs if caution is not used – *e.g. subroutine changes the callers loop counter*

25

Use the Stack!

- Subroutine saves registers it will change
- How? Registers are pushed onto the state at the beginning of the subroutine
- Before it returns, it pops (and restores) the old values



26

Saving Registers… How nice! :-)

```
DoSomething:
    push rax
    push rbx
    push rcx

    ...

    pop rcx
    pop rbx
    pop rax
    ret
```

Backup registers

Your code

Restore them.
Note the reverse order

27

Fixed Subroutine

```
Hornet:
    push rdx

    lea  rdx, StingersUp
    call PrintString

    pop rdx
    ret
```

Push the value on the stack
`stack.push(rdx)`

Restore the value
`rdx = stack.pop()`

28

Subroutine Correct Output

```
Stingers Up!
1947
```

The subroutine worked perfectly

29

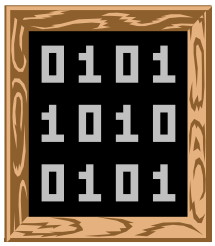
Stack Frames

The Jenga of data!

30

Stack Frames

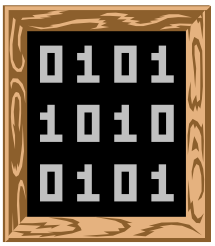
- *Stack Frames* is an compiler approach where subroutine arguments are passed using the system stack
- The stack is also used to store local variables



31

Why is this needed?

1. Need to support **any** number of parameters – *even if it exceeds the available number of registers*
2. Need support local variables



32

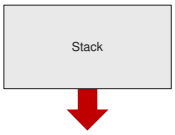
Stack Frame Contents

- Contains all the information needed by subroutine
- Includes:
  - calling program's return address
  - input parameters to the subroutine
  - the subroutine's local variables
  - space to backup the caller's register file

33

Nesting is Possible

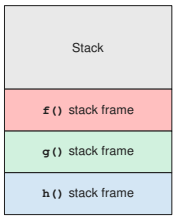
- Stack is LIFO (last in first out), so subroutines can call subroutines
- This approach allows recursion and all the features found in high-level programming languages



34

Nesting is Possible

- For example, subroutine `f()` calls `g()` which then calls `h()`
- Since the stack is used, the stack frames follow the LIFO behavior



35

How it Starts Up

- Caller
  - pushes the subroutine's arguments onto the stack
  - caller calls the subroutine
- Subroutine then…
  - uses the stack to backup registers
  - and *"carve"* out local variables

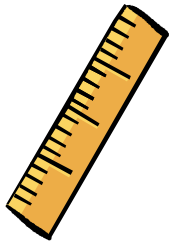
36

How it Finishes

- Subroutine…
  - restores the original register values
  - removes the local variables from the stack
  - calls the processor "return" instruction
- Caller, then…
  - removes its arguments from the stack
  - handles the result – which can be passed either in a register or on the stack

37

Stack Frame Size Varies

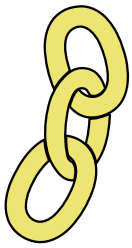


- The number of input arguments and local variables varies from subroutine to subroutine
- The arrangement of data within the stack frame also varies from compiler to compiler
- Stack frames is a *concept* – and it is used with various differences

38

What About Different Object Files?

- Programs are often created from multiple object files
- These can be created by different compilers and linked separately –
- So, how do we make sure that these are all compatible?



39

Calling Convention

- A *calling convention* is defined by a programming system (e.g. a language) to define *how* data will be passed
- In particular, it defines the structure of the stack frame and how data is returned

40

Calling convention

- For example:
  - Is the first argument pushed first? Or last?
  - Is the result in a register? Or the stack?
- If all subroutines follow the same format
  - caller can use the same format for each
  - subroutines can also be created separately and linked together

41

Compatibility

- If two different compilers use the same calling convention, the resulting object files will be compatible
- This means, large applications can be created in different programming languages

42

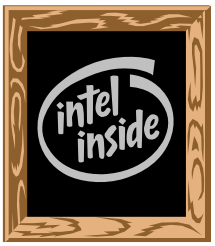
Stack Frames on the x64

Pretty much how its done on all processors

43

Stack Frames on the x64

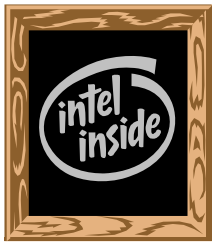
- Stack frames on the x64 are accomplished pretty much the same way as other processors
- How it is done in real life is not simple – and is one of the hardest concepts to understand



44

Stack Frames on the x64

- On the x64, we will use the Base Pointer (RBP) to access elements in the stack frame
- This is a pointer register
- We will use it as an "anchor" in our stack frame



45

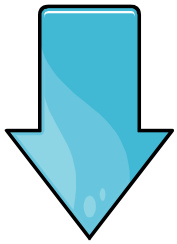
Stack Frames on the x64

- As we build the stack frame, we will set RBP to fixed address in the stack frame
- Our parameters and local variables will be accessed by looking at memory *relative* to the RBP
- So, we will look *x* many bytes above and below the "anchor"

46

Stack on the x64

- The stack base on the x64 is stored in high memory and grows downwards towards 0
- So, as the size of the stack increases, the stack pointer (RSP) will decrease in value



47

Stack on the x64

- On a 64-bit system, it will decrease by increments of 8 bytes
- So, each of our values (local variables and parameters) will be offsets of 8

48

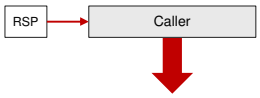
Stack Frame Start Steps

1. Caller pushes arguments
2. Call the subroutine
3. Backup (push) the Base Pointer
4. Set the Base Pointer
5. "Carve" Local Variables
6. Backup (push) local variables

49

Structure of a Stack Frame

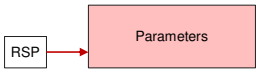
- In this example, the stack is growing downward
- The stack pointer (RSP) is always on the bottom of the stack frame



50

1. Push Parameters

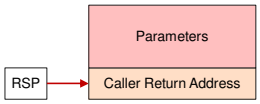
- Caller starts by pushing each of the parameters onto the stack
- Order parameters are pushed is defined in the *calling convention*



51

2. Call the subroutine

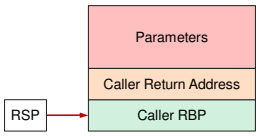
- The caller then uses the *Call Instruction* to pass control to the subroutine
- The processor pushes the IP (instruction pointer) on the stack
- Subroutine now runs



52

3. Backup the Old Base Pointer

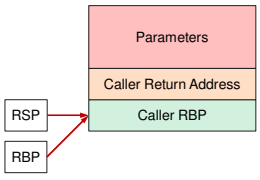
- We need to set the Base Pointer (RBP)
- So, the old version needs to be saved (so it can be restored)
- Old RBP is pushed



53

4. Set the Base Pointer

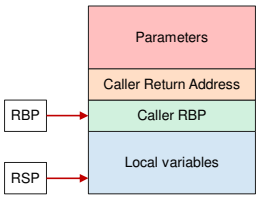
- Then, it sets the Base Pointer (RBP) to the current stack pointer (RSP) address
- RBP is an "anchor" that we will use



54

5. "Carve" Local Variables

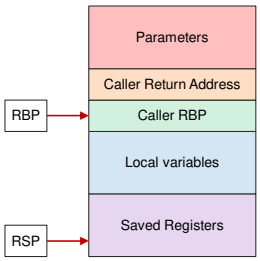
- Subroutine now creates local variables on the stack
- Their initial values can be simply pushed



55

6. Backup Registers

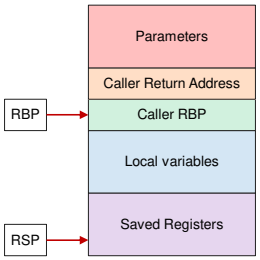
- Finally, the subroutine saves all the registers (that will change) on the stack
- It will restore them at the end



56

The Completed Stack Frame

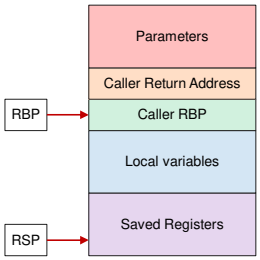
- The Stack Frame contains all the information a subroutine needs
- We just need to be able to access the data programmatically



57

Parameters & Local Variables

- Now, RBP is set to an address between the parameters and the local variables
- We can use *offsets* from RBP to access each



58

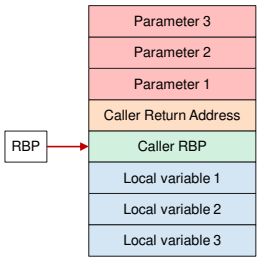
Compiler Friendly… Not Human

- The offset values have to count the bytes from the RBP register
- These are not easy to read since both parameters and local variables are just offsets

59

Size of Stack Values – 64 bit

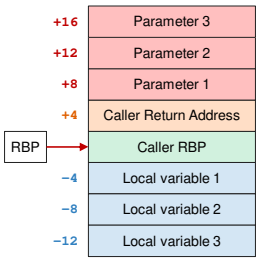
- x64 stack grows downward, so…
- Local variables will have a negative offset
- Parameters will have a positive offset



60

Size of Stack Values – 32 bit

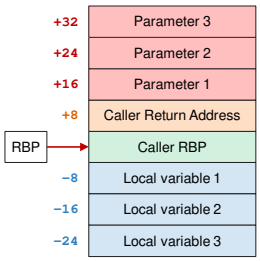
- On a 32-bit system, each word is 4 bytes
- So, each value on the stack is 4 bytes
- Offsets increase and decrease by 4



61

Size of Stack Values – 64 bit

- On a 64-bit system, each word is 8 bytes
- So, each value on the stack is 8 bytes
- Offsets increase and decrease by 8



62

Caller Example

```
push   42
push   rax
call   Subroutine
```

1. Push parameters (16 bytes added)
2. Call subroutine

63

Subroutine: Setup Example

```
push   rbp
mov    rbp, rsp
push   1
push   2
push   rax
push   rbx
```

3. Backup RBP
3. Set new RBP
4. Local variables
5. Backup registers

64

Subroutine Data Example

```
mov   rax, [rbp + 16]
add   rax, [rbp + 24]

mov   [rbp - 8], rax
```

Offset from pointer
Parameter 1
Parameter 2
Local variable 1

65

Stack Frame Ending Steps

1. Restore registers (pop)
2. Restore the stack pointer (set it to the base pointer)
3. Restore the old base pointer
4. Return and delete parameters

66

Stack Frame Return

- The Return can also be used to clean up the caller's stack items
- You can specify the number of bytes to pop (and discard) after the return
- Alternatively, the caller can clean up the stack

67

Stack Frame Return

```
RET byteCount
```

Bytes to discard after return

68

Subroutine: Ending Example

```
pop   rbx
pop   rax

mov   rsp, rbp
pop   rbp
ret   16
```

1. Restore registers (pop in reverse order)
2. Set RSP to RBP
Effectively deletes all local variables
3. Restore RBP
4 Return after the stack is restored

69

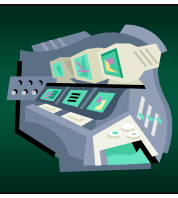
```
MySub:
    push  rbp
    mov   rbp, rsp
    push  1
    push  2
    push  rax
    push  rbx

    ...

    pop   rbx
    pop   rax
    mov   rsp, rbp
    pop   rbp
    ret   16
```

Setup base pointer
Local variables (with initial values)
backup registers
Restore registers
Restore base pointer

70



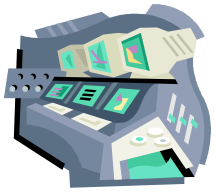
Operating Systems

The master software

71

What is an operating system?

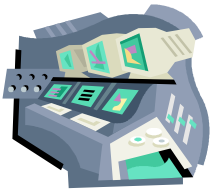
- The operating system is simply a series of programs
- These programs, however, run with special privileges which are needed by the OS
- Processors support two modes for executing programs



72

Execution Modes

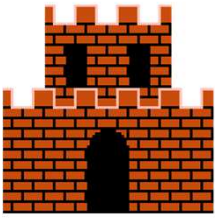
- *Privileged (supervisor) mode*
  - can run special instructions
  - can talk to all the hardware
  - etc…
- *User mode*
  - can only execute certain instructions
  - can't talk to all the hardware



73

Vector Tables

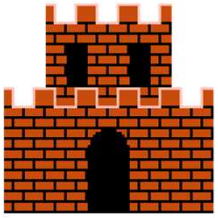
- Programs (and hardware) often need to talk to the operating system
- Examples:
  - software needs talk to the OS
  - USB port notifies the OS that a device was plugged in



74

Vector Tables

- But how does this happen?
- The processor can be *interrupted* – alerted – that something must be handled
- It then runs a special program that handles the event



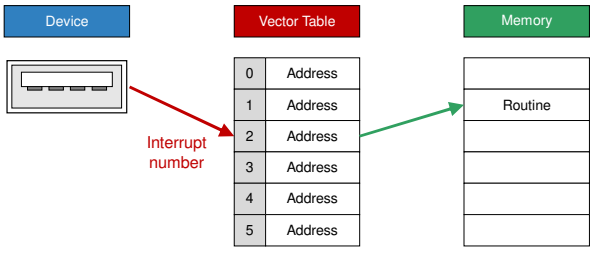
75

Vector Table

- During an interrupt, the device sends the processor an *interrupt number*
- The processor looks up the number in the *vector table*
- Table contains the address of *Interrupt Service Routine (ISR)* to execute

76

How It Works



77

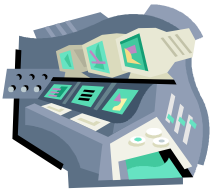
What Happens…

1. Device interrupts the processor
2. Current program state is saved
   - register file
   - instruction pointer
3. Processor executes ISR using the Vector Table address
4. Current program state is restored

78

The Kernal

- All these Interrupt Service Routines belong to the *kernal* – the core of the operating system
- Vast majority of the operating system is hidden from the end user



79

Interact with Applications

How do WE talk to the OS

80

Interact with Applications

- Software also needs to talk to the operating system
- For example:
  - draw a button
  - print a document
  - close this program
  - etc…

81

Interact with Applications

- Software can interrupt itself with a specific number
- This interrupt is *designated specifically for software*
- The operating system then handles the software's request

82

Application Program Interface

- Programs "talk" to the OS using *Application Program Interface (API)*
- Application → Operating System → IO
- Benefits:
  - makes applications faster and smaller
  - also makes the system more secure since apps do not directly talk to IO

83

Instruction: syscall (64-bit)

```
SYSCALL
```

Calls interrupt number reserved for programs needing attention

84

Subroutine vs. Interrupt

Subroutine	Interrupt
Executes code	Executes code
Returns when complete	Returns when complete
Called by the application	Executed by the processor
Part of the application	Handles events for the OS

85

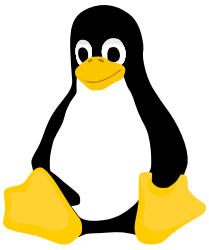
Linux System Calls

How software and hardware "talk"

86

Interrupts on the Linux

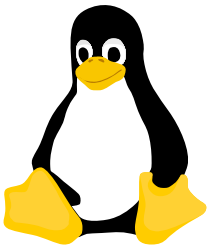
- Linux, like other operating systems communicate with applications using *interrupts*
- Applications do not know where (in memory) to contact the kernal – so they ask the processor to do it



87

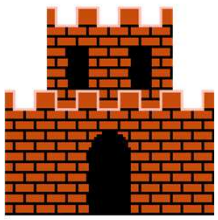
How It Works

1. Fill the registers
2. Interrupt using syscall (or INT 0x80 if on 32-bit)
3. Any results will be stored in the registers



88

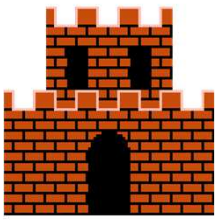
How to Call Linux – 64 bit



- The **rax** register must contain the *system call number*
- This number indicates what you asking the OS to do
- There are only 329 total calls in the entire 64-bit UNIX operating system!

89

How to Call Linux – 64 bit

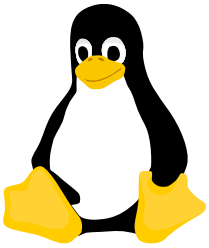


- Different registers are used to hold data
- The order is also quite odd: `rdi`, `rsi`, `rdx`, `r10`, `r8`

90

Kernals are Simple!

- Linux only has **1** write and **1** read system call
- The location, number of bytes, and device only change *"write x many bytes from address y to device z"*
- So, writing to the screen, a file, a port, etc…use the same call!



91

Some Linux 64 Calls

System Call	rax	rdi	rsi	rdx
read	0	file descriptor	address	max bytes
write	1	file descriptor	address	count
open	2	address	flags	mode
close	3	file descriptor		
get pid	39			
exit	60	error code		

92

Linux 64: Sys Write

```
mov   rax, 1
mov   rdi, 1
lea   rsi, address
mov   rdx, length
syscall
```

Linux command for WRITE

1 = Screen

Call Linux

93

Linux 64: Sys Read

```
mov   rax, 0
mov   rdi, 0
lea   rsi, address
mov   rdx, maxBytes
syscall
```

Linux command for READ

0 = Keyboard

Maximum number of bytes to read

Call Linux

94

Write Example

```
SacState:
   .ascii "Stinger's up!\n"    #\n counts as 1 character
...
   mov  rax, 1                 #1 = write
   mov  rdi, 1                 #1 = screen
   lea  rsi, SacState
   mov  rdx, 14                #14 bytes
   syscall
```


95