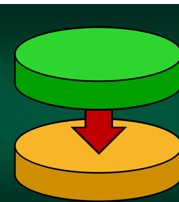
Part 7

Subroutines & Operating Systems

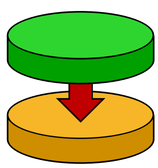
The System Stack



Pile of… Data

The System Stack

- The processor maintains a stack in memory
- It is used to allow *subroutines* which are similar to the function you are used to programming in Java



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
- A fixed location pointer (S0) defines the bottom of the stack
- A *stack pointer* (SP) gives the location of the top of the stack

Approaches

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

Size of the Stack

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

Subroutine Call Basics

Organizing Your Program

Subroutine Call

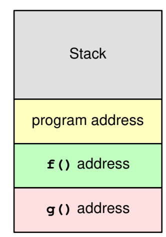
- The stack is essential for subroutines to work
- How?
  - used to save the return addresses for call instructions
  - backup and restore registers
  - pass data between subroutines *(we won't get to this)*

When you call a subroutine…

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

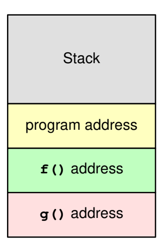
Nesting is Possible

- Subroutines can call other subroutines
- Here: the program calls f() which then calls g(), 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



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)



Passing Parameters

- Useful subroutines…
  - need you to be able to pass data into them
  - and be able to read data from it
- One of the easiest ways to accomplish this is by using the processors registers
- *Incredibly efficient!*

Passing Through Registers

- However… processor might not have enough registers for each parameter
- *Other forms of passing data have to be used*

x64 Subroutines

Organizing Your Programs … with Intel

Instruction: Call

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

Instruction: Call

Usually a label (which is an address)

```
CALL address
```


Instruction: Return

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

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!

Instruction: Return

No arguments!

```
RET
```


Subroutine Example

```
_start:
    mov  $4, %rax
    mov  $12, %rbx
    call AddIt
    add  $1, %rbx
    ...
AddIt:
   add %rax, %rbx
   ret
```


Saving Registers & Lost Data

Avoiding horrible side-effects

Saving Registers & Lost Data

- Each subroutine will use the registers as it needs
- So, when a sub is called, *it may modify the caller's registers*
- Some processors have few registers – so its very likely
- This can lead to hard-to-fix bugs if caution is not used – e.g. loop counter gets changed

Two Solutions

- Caller saves values
  - caller saves all their registers to memory before making the subroutine call
  - after, it restores the values before continuing
  - not recursion friendly – it pushes all of them!
- Subroutine saves the values
  - push registers (it will change) onto the stack
  - before it returns, it pops (and restores) the old values off the stack

Saving Registers… How nice! :-)

```
DoSomething:
   push %rax
   push %rbx
   push %rcx

   ...

   pop %rcx
   pop %rbx
   pop %rax
   ret
```

Save registers

Your code

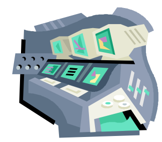
Restore them.
Note the reverse order

Operating Systems

The master software

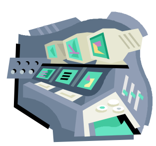
What is an operating system?

- 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



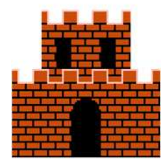
Execution Modes

- *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



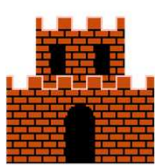
Vector Tables

- 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



Vector Tables

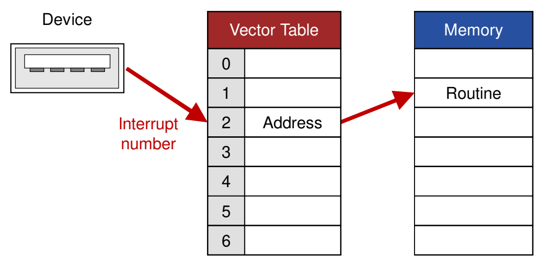
- But how does this happen?
- The processor can be *interrupted* – alerted – that something must be handled
- It then runs a special program that handles the event



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

How It Works

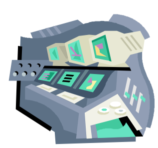


The Processor Actions

1. Backup the register file
2. Execute *Interrupt Service Routine (ISR)*
3. Once completed: restore the original executing program & register file

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



Interact with Applications

How do WE talk to the OS

Interact with Applications

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

Interact with Applications

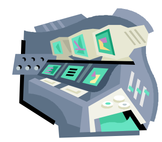
- Software can interrupt itself with a specific number
- This interrupt is *designated specifically for software*
- The operating system then handles the software's request

Application Program Interface

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

The Changing Interrupts…

- In the 32-bit Intel x86 the program can invoke any interrupt number
- … which turned out not to be particularly safe
- So, the Intel x64 hardcoded the interrupt number



Instruction: syscall (64-bit)

```
SYSCALL
```

Calls interrupt number reserved for programs needing attention

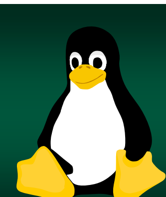
Returning from an Interrupt

```
IRET
```

Interrupt Return
(You will never use this unless you write an operating system)

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

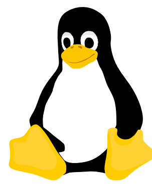


Linux System Calls

How software and hardware "talk"

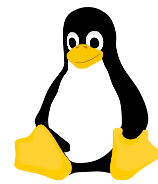
Interrupts on the Linux

- 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

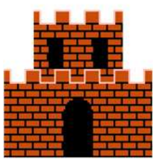


How It Works

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



How to Call Linux – 64 bit



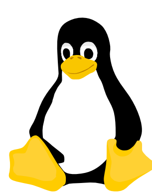
- The rax register must contain *system call number*
- This number indicates what you asking the OS to do
- There are only 329 total calls in the entire 64-bit UNIX operating system!

How to Call Linux – 64 bit

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

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!



Some Linux 64 Calls

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

Linux 64: Sys Write

```
mov   $1, %rax
mov   $1, %rdi
mov   $address, %rsi
mov   $length, %rdx
syscall
```

Linux command for WRITE

1 = Screen

Call Linux

Linux 64: Sys Read

```
mov   $0, %rax
mov   $0, %rdi
mov   $address, %rsi
mov   $maxBytes, %rdx
syscall
```

Linux command for READ

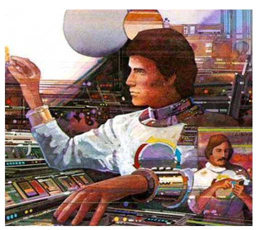
0 = Keyboard

Maximum number of bytes to read

Call Linux

Null-Terminated Strings

- Notice that UNIX requires that you specify the number of bytes before you write
- So, you need to count the bytes every time



Null-Terminated Strings

- That's why C-style null-terminated strings are used
- The programming language (or library) counts the bytes for you!