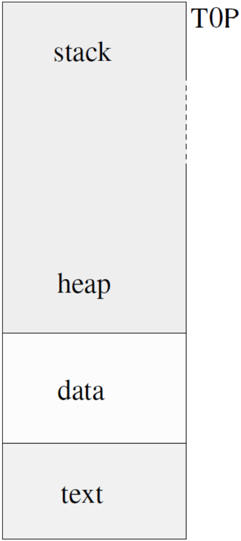
# The Stack and Functions

# The Stack

- Up until now we have mostly ignored the stack.
- Instead we have relied on the data/bss segment.
- This has been sufficient so far, but this will change.
- We are now in a position where our programs will start being complex enough that we cannot rely entirely on fixed size allocations.



# The stack in a Familiar Context

- Consider the following(intentionally inefficient)recursive C++ function. Assuming we don't know that

$$fib(n) = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n\sqrt{5}} \tag{1}$$

```
// assuming fib(1)=1
int fib(int n)
{
   if(n<2)
      return 1;
   int fibM1=fib(n-1);
   int fibM2=fib(n-2);
   return fibM1+fibM2;
}
```

- If $N$ is too large our program will experience a stack overflow.

# The Stack in a Familiar Context

- But why?

# The Stack in a Familiar Context

- But why?
- There are two causes (though in a sense they are the same)
  - The more obvious one is each execution of the *fib* will push at least two integers onto the stack. One for each of `FibM1`, and `FibM2`. Now the space complexity of this naive fibincci algorithm is $O(n)$. We can try and roughly calculate $fib(10\,000\,000)$ we need $2 * 4\text{bytes} * 10\,000\,000 = 76.29\text{mb}$ just to store the max required number of `FibM1`s, and `FibM2`s.
    - Most Linux distros have a stack between $8mb$ and $16mb$

# The Stack in a Familiar Context

- But why?
- There are two causes (though in a sense they are the same)
  - The more obvious one is each execution of the *fib* will push at least two integers onto the stack. One for each of `FibM1`, and `FibM2`. Now the space complexity of this naive fibincci algorithm is $O(n)$. We can try and roughly calculate $fib(10\,000\,000)$ we need $2 * 4\text{bytes} * 10\,000\,000 = 76.29\text{mb}$ just to store the max required number of `FibM1s`, and `FibM2s`.
    - ★ Most Linux distros have a stack between $8mb$ and $16mb$
- What if we implement our recursive function to somehow not push any local variable data onto the stack?
  - Even if this was possible there is one stack item we would have to push.

# The Stack in a Familiar Context

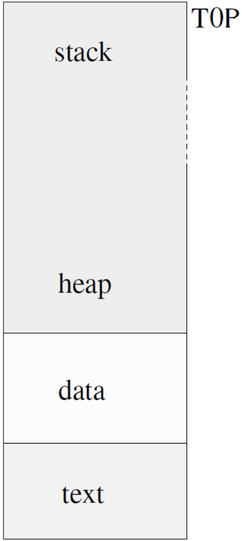
- But why?
- There are two causes (though in a sense they are the same)
  - The more obvious one is each execution of the *fib* will push at least two integers onto the stack. One for each of `FibM1`, and `FibM2`. Now the space complexity of this naive fibincci algorithm is $O(n)$. We can try and roughly calculate $fib(10\,000\,000)$ we need $2 * 4\text{bytes}*10\,000\,000 = 76.29\text{mb}$ just to store the max required number of `FibM1s`, and `FibM2s`.
    - Most Linux distros have a stack between $8mb$ and $16mb$
- What if we implement our recursive function to somehow not push any local variable data onto the stack?
  - Even if this was possible there is one stack item we would have to push.
  - The return address.

# The Stack

- Now that we have seen how it can break, lets us use the stack.
- The first point to recall is that the stack starts at :0x7fffffffffff. Unless there is some stack randomization in place.
- The second point is that the *rsp* register stores the stack pointer.
- If nothing has been added to the stack *rsp* =0x7fffffffffff.
- We can interact with the stack in a number of ways, simplest of which are the `push` and `pop` instructions.

# The Stack

- The `push` instruction decrements the `rsp` register and stores the value being pushed at this address
- The `pop` instruction places the value at the top of the stack into its operand and increments `rsp`
- With the x86-64 instructions you should push and pop 8 bytes at a time
    - It is also possible to push and pop 2 bytes (word) at a time
    - Direct 4 byte push and pops is not enabled in 64-bit mode



# A Stack Example

Example with start of the stack at $0x7ffffffff$

```
;A
mov  rax,74
push rax
;B
inc  rax
push rax
;C
inc  rax
push rax
;D
pop rax
;E
pop rax
;F
```

| | | | | | | |
|---|---|---|---|---|---|---|
| 0x7ffffffffff0 | x | 74 | 74 | 74 | 74 | 74 |
| 0x7fffffffffe8 | x | x | 75 | 75 | 75 | x |
| 0x7fffffffffe0 | x | x | x | 76 | x | x |
| . | x | x | x | x | x | x |
| . | x | x | x | x | x | x |
| . | x | x | x | x | x | x |
| | A | B | C | D | E | F |

In GDB

- You can use x/1dg $rsp to test the stack content.
- You can use p/x $rsp to test the stack pointer itself.
- Just remember that stack randomization effects the initial stack pointer.

# The Stack

- Stack space is often reserved for local variables by subtracting the size needed from the stack pointer (`rsp`).
- Then an offset is used to refer to the variables.

```
sub rsp,      16  ;subtract 16 bytes
mov qword [rsp+8], 123 ;set our first qword variable to 12
mov qword [rsp],   24  ;set our second qword variable to 2
```

# The stack

| | | | | |
|---|---|---|---|---|
| 0x7fffffffffff | | | | |
| 0x7ffffffffff0 | x | | 123 | 123 |
| 0x7fffffffffe8 | x | | | 24 |
| 0x7fffffffffe0 | x | x | x | x |
| . | x | x | x | x |
| . | x | x | x | x |
| . | x | x | x | x |
| | x | x | x | x |
| | | sub rsp, 16 | mov [rsp+8],123 | mov [rsp], 24 |

# The Stack

- Remembering offsets for a large number of variables can become a burden. Instead use the equ pseudo-op.

```
first   equ 8
second  equ 0
mov qword [rsp+first],123
mov qword [rsp+second],24
```

# The Stack

- Remembering offsets for a large number of variables can become a burden. Instead use the equ pseudo-op.

```
first   equ 8
second  equ 0
mov qword [rsp+first],123
mov qword [rsp+second],24
```

- How do we delete the variables after use?

# The Stack

- Remembering offsets for a large number of variables can become a burden. Instead use the equ pseudo-op.

```
first   equ 8
second  equ 0
mov qword [rsp+first],123
mov qword [rsp+second],24
```

- How do we delete the variables after use?
  - We just move the stack pointer back.

    ```
    add rsp,16
    ```

# Functions

- Assuming we have loaded our parameters (will be explained shortly)
- You can call a function using

```
    call    my_function
```

- `my_function` should be an appropriate address/label in the code segment
- The function's return value will be in `rax` or `xmm0`
- The effect of a function call is much like

```
    push    next_instruction
    jmp     my_function
next_instruction:
```

# The Return Instruction

- You can return to the location a function was called from using

```
ret
```

- The effect of the return instruction (`ret`) is to pop an address off the stack and branch to it
- We could get much the same effect using

```
pop   rdi
jmp   rdi
```

# Function Parameters

- How do we get our parameters to our function?
- On 32 bit Linux all parameters were pushed onto the stack

# Function Parameters

- How do we get our parameters to our function?
- On 32 bit Linux all parameters were pushed onto the stack
- On x86-64 there are 8 more registers, so some parameters are passed in registers.

# Function Parameters

- How do we get our parameters to our function?
- On 32 bit Linux all parameters were pushed onto the stack
- On x86-64 there are 8 more registers, so some parameters are passed in registers.
- Linux and Mac OS/X pass integer and address parameters 1 through 6 in `rdi`, `rsi`, `rdx`, `rcx`, `r8` and `r9`
  - The remaining integer and address parameters are pushed onto the stack
  - The first 8 floating point parameters are passed in registers `xmm0` - `xmm7`

# Function Parameters

- How do we get our parameters to our function?
- On 32 bit Linux all parameters were pushed onto the stack
- On x86-64 there are 8 more registers, so some parameters are passed in registers.
- Linux and Mac OS/X pass integer and address parameters 1 through 6 in `rdi`, `rsi`, `rdx`, `rcx`, `r8` and `r9`
  - The remaining integer and address parameters are pushed onto the stack
  - The first 8 floating point parameters are passed in registers `xmm0` - `xmm7`
- Windows uses registers `rcx`, `rdx`, `r8` and `r9` for the first 4 integer and address parameters
  - The remaining integer and address parameters are pushed onto the stack. (there is also a set amount of stack padding)
  - Windows uses `xmm0` - `xmm3`

# Function Parameters

- How do we get our parameters to our function?
- On 32 bit Linux all parameters were pushed onto the stack
- On x86-64 there are 8 more registers, so some parameters are passed in registers.
- Linux and Mac OS/X pass integer and address parameters 1 through 6 in `rdi`, `rsi`, `rdx`, `rcx`, `r8` and `r9`
  - The remaining integer and address parameters are pushed onto the stack
  - The first 8 floating point parameters are passed in registers `xmm0` - `xmm7`
- Windows uses registers `rcx`, `rdx`, `r8` and `r9` for the first 4 integer and address parameters
  - The remaining integer and address parameters are pushed onto the stack. (there is also a set amount of stack padding)
  - Windows uses `xmm0` - `xmm3`
- In all cases pushed parameters are pushed in reverse order

# Function Parameters (2)

- Functions like `printf` having a variable number of parameters must place the number of floating point parameters in `rax`

| Register | Usage | Preserved across function calls |
|---|---|---|
| `rax` | temporary register; with variable arguments passes information about the number of vector registers used; $1^{\text{st}}$ return register | No |
| `rbx` | callee-saved register; optionally used as base pointer | Yes |
| `rcx` | used to pass $4^{\text{th}}$ integer argument to functions | No |
| `rdx` | used to pass $3^{\text{rd}}$ argument to functions; $2^{\text{nd}}$ return register | No |
| `rsp` | stack pointer | Yes |
| `rbp` | callee-saved register; optionally used as frame pointer | Yes |
| `rsi` | used to pass $2^{\text{nd}}$ argument to functions | No |
| `rdi` | used to pass $1^{\text{st}}$ argument to functions | No |
| `r8` | used to pass $5^{\text{th}}$ argument to functions | No |
| `r9` | used to pass $6^{\text{th}}$ argument to functions | No |
| `r10` | temporary register, used for passing a function's static chain pointer | No |
| `r11` | temporary register | No |
| `r12-r15` | callee-saved registers | Yes |

# Simple Function

Simple function that returns the larger of two longs.

```
; long max(long a, long b)
max:
   mov rax, rdi    ; move parm1 to rax
   cmp rax, rsi    ; compare rax to parm2
   cmovl rax, rsi ; if parm2 > rax then move parm 2 to rax
   ret
```

# Simple Function

Calling the simple function

```
mov rdi, 123 ; load parm1
mov rsi, 742 ; load parm2
call max
```

# Real World Functions

- Both Linux and Windows require the maintenance of the stack on 16 byte boundaries during the main part of functions.

# Real World Functions

- Both Linux and Windows require the maintenance of the stack on 16 byte boundaries during the main part of functions.
  - The reason behind this requirement is to make it possible for local variables (on the stack) to be on 16 byte boundaries,
  - a requirement for some SSE(Streaming SIMD Extensions) and AVX(Advanced Vector Extension) instructions.

# Real World Functions

- Both Linux and Windows require the maintenance of the stack on 16 byte boundaries during the main part of functions.
  - The reason behind this requirement is to make it possible for local variables (on the stack) to be on 16 byte boundaries,
  - a requirement for some SSE(Streaming SIMD Extensions) and AVX(Advanced Vector Extension) instructions.
- Conforming functions generally start with "`push rbp`" which re-establishes the 16 byte bounding temporarily **botched by the function call**
  - Remember, the `call` operation pushes a 8 byte value onto the stack (the return address)

# Real World Functions

- Both Linux and Windows require the maintenance of the stack on 16 byte boundaries during the main part of functions.
  - The reason behind this requirement is to make it possible for local variables (on the stack) to be on 16 byte boundaries,
  - a requirement for some SSE(Streaming SIMD Extensions) and AVX(Advanced Vector Extension) instructions.
- Conforming functions generally start with "`push rbp`" which re-establishes the 16 byte bounding temporarily **botched by the function call**
  - Remember, the `call` operation pushes a 8 byte value onto the stack (the return address)
- Following that, conforming functions subtract multiples of 16 from `rsp` to allocate stack space or push pairs of 8 byte values
  - Even if this means over allocation

# Hello world, at last

```
        section .data
msg:    db      "Hello World!",0x0a,0

        section .text
        global  main
        extern  printf
main:
        push    rbp
        mov     rbp, rsp     ; will explain shortly
        mov     rdi, msg     ; parameter 1 for printf
        mov     rax, 0       ; 0 floating point parameters
        call    printf
        mov     rax, 0       ; return 0
        mov     rsp, rbp     ; will explain shortly (NIB)
        pop     rbp
        ret
```

# Stack frames

- Stack frames are used by the gdb debugger to trace backwards through the stack to inspect calls made in a process
- If we start and end each function like:

```
push    rbp
mov     rbp, rsp

...
mov rsp, rbp
pop rbp
ret
```

- We are in effect constructing a link list of all of the stack frames.
- All non-leaf functions must have the stack frame set up and destruction to conform to the ABI and be properly c/c++ compatible.

# Stack frames

| | In base function: | In Function L1: | In Function L2: | In Function L3: | In Function L4: |
|---|---|---|---|---|---|
| rbp | 0x0 | StackPointer(0) | StackPointer(1) | StackPointer(2) | StackPointer(3) |
| rsp | StackPointer(0) | StackPointer(1) | StackPointer(2) | StackPointer(3) | StackPointer(4) |
| | | | | | |
| 0x7ffffffff0 | x | Ret Adr to Base | Ret Adr to Base | Ret Adr to Base | Ret Adr to Base |
| 0x7fffffffe8 | x | 0x0 | 0x0 | 0x0 | 0x0 |
| 0x7fffffffe0 | x | x | Ret Adr to L1 | Ret Adr to L1 | Ret Adr to L1 |
| 0x7fffffffd8 | x | x | StackPointer(0) | StackPointer(0) | StackPointer(0) |
| 0x7fffffffd0 | x | x | x | Ret Adr to L2 | Ret Adr to L2 |
| 0x7fffffffc8 | x | x | x | StackPointer(1) | StackPointer(1) |
| 0x7fffffffc0 | x | x | x | x | Ret Adr to L3 |
| 0x7fffffffb8 | x | x | x | x | StackPointer(2) |
| 0x7fffffffb0 | x | x | x | x | x |

*assuming no local varibles are stored on the stack. If the local variables were stored where would they be?

# Stack frames

- If you require space for local variables you simple need to subtract an amount from the stack pointer `rsp`.

# Stack frames

- If you require space for local variables you simple need to subtract an amount from the stack pointer `rsp`.
  - The subtraction should always maintain the 16 byte boundary.
  - For example, say we wish to have a single quadword as a local variable.

# Stack frames

- If you require space for local variables you simple need to subtract an amount from the stack pointer `rsp`.
  - The subtraction should always maintain the 16 byte boundary.
  - For example, say we wish to have a single quadword as a local variable.
    - We only **have** to subtract 8 bytes.
    - But we **should** maintain the 16 byte boundary.

```
push    rbp
mov     rbp, rsp
sub     rsp,16
```

# Stack frames

- If you require space for local variables you simple need to subtract an amount from the stack pointer `rsp`.
  - The subtraction should always maintain the 16 byte boundary.
  - For example, say we wish to have a single quadword as a local variable.
    - We only **have** to subtract 8 bytes.
    - But we **should** maintain the 16 byte boundary.

    ```
    push    rbp
    mov     rbp, rsp
    sub     rsp,16
    ```

- if we allocate local variables we **must** use

  ```
  mov rsp, rbp
  pop rbp
  ret
  ```

  Just popping will not work.

# Stack frames

- If you prefer you can utilize

  ```
  leave
  ret
  ```

  instead of

  ```
  mov rsp, rbp
  pop rbp
  ret
  ```

# print max example

```
main:
   push rbp
   mov  rbp, rsp
;   print_max ( 100, 200 );
   mov  rdi, 100    ; first parameter
   mov  rsi, 200    ; second parameter
   call print_max
   mov  rax, 0     ; to return 0
   leave
   ret
```

# print max example

```
; void print_max ( long a, long b )
; {
a   equ  0
b   equ  8
max equ 16
print_max:
   push rbp
   mov  rbp, rsp
   sub  rsp, 32          ; leave space for a, b and max
   mov  [rsp+a], rdi     ; save a
   mov  [rsp+b], rsi     ; save b
   mov  [rsp+max], rdi ; max = a;
   cmp  rsi, rdi         ; if ( b > max ) max = b
   jng  skip
   mov  [rsp+max], rsi
```

# print max example

```
skip:
segment .data
   fmt db   "max(%ld,%ld) = %ld",0xa,0
segment .text
   mov  rdi, fmt        ; address of format string
   mov  rsi, [rsp+a]    ; first  %ld
   mov  rdx, [rsp+b]    ; second %ld
   mov  rcx, [rsp+max]  ; third  %ld
   mov  rax, 0          ; zero floating point param
   call printf
   leave
   ret
```

# Accessing Local Variables

- Can you see the potential issue with the way we are accessing our local variables?

# Accessing Local Variables

- Can you see the potential issue with the way we are accessing our local variables?
  - If anything is pushed onto the stack our offsets become incorrect.

# Accessing Local Variables

- Can you see the potential issue with the way we are accessing our local variables?
  - If anything is pushed onto the stack our offsets become incorrect.
- Solution?

# Accessing Local Variables

- Can you see the potential issue with the way we are accessing our local variables?
  - If anything is pushed onto the stack our offsets become incorrect.
- Solution?
  - Simply use `rbp` as the base address. Just remember that since rbp holds the value of rsp before the subtraction you need to use subtract from rbp and not add (think about what the offset values should be).
  - or just don't use push and pop other than for the stack frame setup and destruction

# Recursive Functions

- Recursive algorithms serve as a good example for why we need stack based storage.
- Often times we can get away with utilizing registers that are preserved across function calls.
- However consider the case where we a have a recursive algorithm. On the first level we decide to use `r15` to store a value. But now on the second level `r15` is already in use...

# Recursive Functions

- Consider again the following recursive Fibonacci function

```
// assuming fib(1)=1
long fib(long n)
{
   if(n<2)
      return 1;
   return fib(n-1)+fib(n-2);
}
```

# Recursive Functions

- ```
  fib:  push rbp
        mov rbp, rsp
        sub rsp, 16
  N   equ 0
  nM1 equ 8
        mov rax, 1           ;base case return value
        cmp rdi, 2
        jl .end              ;first parameter<2 (base case)
        dec rdi              ;recall rdi is the first parameter (n)
        mov [rsp+N], rdi     ;save N-1
        call fib
        mov [rsp+nM1], rax
        mov rdi, [rsp+N]     ;load N-1
        dec rdi
        call fib
        add rax, [rsp+nM1]
        .end
        leave
        ret
  ```
  ```

# Function Implementation: Correct Practice

- If a function is a **non-leaf** function you **must** set up and destroy a stack frame.
- If you utilize a register in your function that should be preserved across function calls (like `r15`) you **must** restore it to its original value.
- Example

```
sub rsp, 16
mov [rsp], r15
mov [rsp+8], r14
....
....
mov r14, [rsp+8]
mov r15, [rsp]
add rsp, 16
```

# Function Implementation: Correct Practice

- or if you make use of no other stack based memory.

```
push r15
push r14
....
....
pop r14
pop r15
```