Assembly Functions

Systems Programming

JOHANNES KEPLER
UNIVERSITÄT LINZ

Michael Sonntag
Institute of Networks and Security

INSTITUTE
OF NETWORKS
AND SECURITY

What is a „function“?

- **Name**
  - Symbol representing the address where the function begins
- **Parameters**
  - Input; arbitrary number and type should be supported
  - Sometimes also output or input+output parameters desirable
- **Local variables**
    - Often called “automatic variables”
  - Private data storage
  - Not accessible from outside
  - “Thrown away” when function ends
- **Global variables**
  - Public data storage
  - Accessible from inside and outside

What is a „function“?

- **Return address**
  - Invisible “parameter”
  - Tells program where to resume executing after function completed
  - In most progr. languages, return address is handled automatically
    - Even in most assembly languages, e.g. X86-*
- **Return value**
  - Transfer result back to caller
  - Most programming languages allow only a **single** return value
    - Most also require that it fits in a **single register** (or use one of the circumventions listed below themselves to emulate this)
  - More than one/larger return value needed?
    1. Use pointers to data (i.e. memory addresses of data)
    2. In the function change/set (parameter) values that a pointers point to
    3. Read values in caller after function returns
    - Pointer to data as direct return value
      - Careful: Who reserved the memory, who frees it?
  - Can be “extended” through in+out parameters

Stack

- Where do we store return addresses, parameters, local variables…?
  - Sometime also large return values (structures)
- In a special memory area called the “Stack”
- We don’t know how “deep” functions will be called (recursion!), so we cannot set an upper limit for the stack size
  - Practically, most OS **do** set a limit → if reached, the program is terminated (to prevent e.g. runaway recursion). But until then only as much memory is provided as is actually used!
- Dangerous from the security point of view
  - Attackers can manipulate it: arbitrary data at arbitrary position (within the stack)
  - Might be executed (but see modern precautions!)

Stack

- Region at the “top” addresses of memory (of current process)
  - Stack is separate for each process
  - Multiple threads: separate stack for each thread
    - Obviously not at the very top any more, as threads share memory, so stack size limits are more stringent (relocating a stack is impossible!)
    - On 64 Bit computers the logical address space is very large, so this is less a problem
- Stack **grows down** in memory (from high towards low addresses)
- Limited in size: Everything larger than a few kB should be put on Heap
- Used to **implement functions**
  - Save state of the caller
  - Pass parameters to the callee
  - Store return address
  - Store local variables
  - Store large return data (reserved by caller)

Stack

- **RSP register** always points to the "top" of the stack
  - =**Lowest** used address
    - For a quadword (=8 bytes) it is the "first" byte (=lowest address of the 8)
- Push data (**`push`**)
  - "Decrement" RSP register
    - How much? As many bytes as the new data item is long!
  - Write data item at new top
- Pop data (**`pop`**)
  - Read data from top (and store it in some place, typ. some register)
  - "Increment" RSP register
    - How much? As many bytes as we remove – This need **NOT** be the same as was used on push!
- RSP can also be modified by **`SUB/ADD`**, e.g. for creating/destroying local variables (remember: stack grows down, so **`SUB`** **creates** space!)
  - Then the "new content" is **NOT** initialized (=old values)!

Calling conventions

- How to call a functions:
  - □ How do we pass parameters? Any metadata (=type) for them?
    - ● Which registers? Or stack only?
    - ● Object-oriented languages: Where do we put the “this” pointer?
  - □ What about local variables?
    - ● And who cleans up after them?
  - □ What about the return value (“A” register, stack…)?
  - □ Which register may be used in the function (=who saves them)?
  - □ Return address (x86 → hardware restrictions!), frames…
- This is a „calling convention“
  - □ „Convention“ because there are few technical restrictions
  - □ Every programming language (or programmer) can decide on his own what to do/how to do it
  - □ You can even mix them in a single program
  - □ BUT: However a function is programmed, this function must be called in an exactly matching way!

Calling conventions: cdecl

- C declaration → Original C programming language
  - All parameters are passed on the stack
  - Stored in reverse order (last parameter is pushed first)
  - Return value in A (=EAX) register
  - Registers A, C, D (=EAX, ECX, EDX) are caller saved, rest is callee saved
  - EBP register used for frame pointer
  - Caller cleans up stack
  - Linux modification: Stack must be 16-Byte aligned on function call
  - Used in Linux-x86-**32** Bit
  - Typical C declaration: `void _cdecl funct();`

Specify the calling convention to use for this function

Calling conventions: Others

- Pascal: Pascal programming language
  - Similar to cdecl, but parameters are pushed on the stack in normal order, which prevents functions with variable count of parameters
  - Callee has to clean up stack
  - Used by Windows 16 Bit (Windows 3.x)
- Stdcall: Similar to Pascal
  - Parameters are pushed in reverse order, like in cdecl
  - Standard calling convention for Windows 32 Bit
- Microsoft fastcall: `__fastcall`
  - First two argument are passed in ECX and EDX, rest on stack in reverse order
  - Windows 32 Bit (depending on compiler; used for optimization)

Calling conventions: Microsoft X64

- RCX, RDX, R8, R9 are used for the first four parameters, the rest are pushed on the stack in reverse order
  - Smaller values are right-justified in registers (=lower bits)
- Return value is in RAX (smaller values do **not** set upper bits to 0!)
- RAX, RCX, RDX, R8, R9, R10, R11 are caller-saved
- RBX, RBP, RDI, RSI, RSP, R12, R13, R14, R15 are callee-saved
- Caller must always reserve 32 Byte of space on stack (shadow space) for the first four parameters (even if less used!)
  - Note that only space is provided, the parameter values are **not** written there by the caller – they are **only** in the registers!
- Stack pointer must be aligned to 16 Bytes
- Used on Windows 64 Bit

Calling conv.: SystemV AMD64 ABI

- SystemV (old Unix version; 1983), AMD64 (“original” 64 Bit version of IA32), ABI (“Application Binary Interface” ≈ calling convention+…)
- First six parameters are passed in RDI, RSI, RDX, RCX, R8, and R9, the rest are pushed on the stack in reverse order
    - Fewer parameters? Function can use them freely (caller-saved!)
  - Linux Kernel calls: RCX replaced by R10; max. 6 parameters; no stack ever used (except as memory for where pointers point to)
- Return value is store in RAX (+ potentially RDX for long values)
- RAX, RCX, RDX, RSI, RDI, R1-R11 are caller-saved
- RBP, RBX, R12-R15 are callee-saved

Matz, Hubička, Jaeger, Mitchell (Eds.), "System V Application Binary Interface: AMD64 Architecture Processor Supplement“, http://chamilo2.grenet.fr/inp/courses/ENSIMAG3MM1LDB/document/doc_abi_ia64.pdf

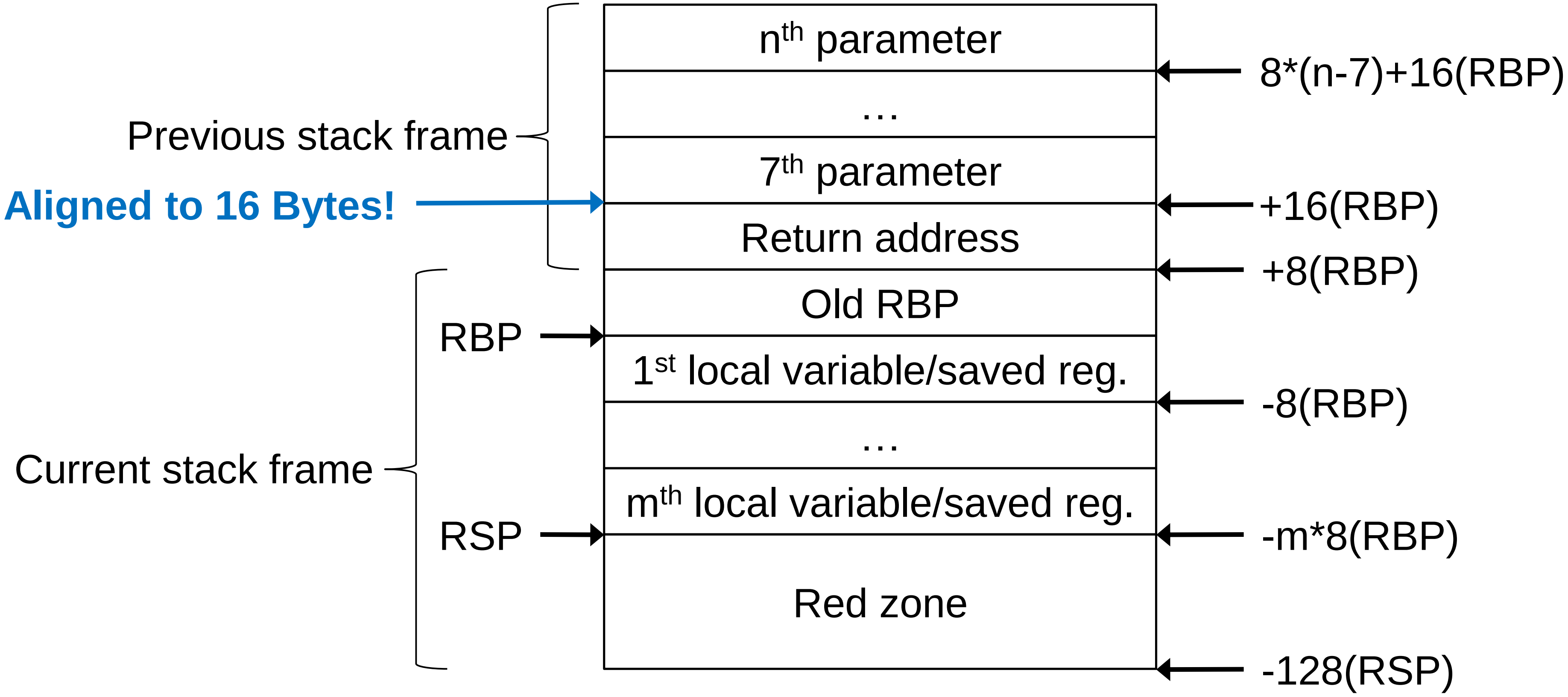
Calling conv.: SystemV AMD64 ABI

- Stack pointer must be aligned to 16 Bytes (=end of parameters)
- 128 Bytes below RSP are guaranteed to exist and can be used by function for e.g. local variables without RSP adjustment (“Red zone”)
  - Not used by signals and interrupt handlers
  - Will be overwritten by function calls → useful for “leaf” functions
    - “Leaf” function = function that doesn’t call **any** other function
  - Optimization purpose: Use without adjusting RSP
- Used on **Linux 64 Bit**, MacOS…

Matz, Hubička, Jaeger, Mitchell (Eds.), "System V Application Binary Interface: AMD64 Architecture Processor Supplement“, http://chamilo2.grenet.fr/inp/courses/ENSIMAG3MM1LDB/document/doc_abi_ia64.pdf

Stack frame

- How does the stack look like when a function is called?
  - □ For each function (without optimizations used!) that is called, a “frame” is reserved



Calling a function – Caller

- Caller adjusts stack pointer so it will end up at 16-Byte alignment after the next 3 bullets

↕ **Can be exchanged**

- Caller saves (typically on stack) all caller-save registers - **if needed**
- Caller pushes parameters N to 7 on stack in reverse order
- Caller puts parameters 1 to 6 in the appropriate registers
- Caller executes the **`call ...`** instruction
  - Which pushes address of the next instruction (=RIP) on the stack
    - This is the “return address”, where execution resumes after the function returns
  - The RIP register is modified to contain the address specified in the call instruction – which is the start of the function

Calling a function – Callee - Prologue

- Callee saves the base pointer on the stack
  - “Trace back” to the previous function; used e.g. by debuggers
  - **`push %rbp`**
- Callee copies the stack pointer into the base pointer
  - **`movq %rsp,%rbp`**
- Callee subtracts amount of bytes needed for local variables from RSP
  - **`subq $???,%rsp`**
    - If less than 128 needed (with registers!), the red zone can be used
- Callee pushes all callee-save registers to be used on the stack

- Now the actual function begins
  - Parameters: RBP+16 (7th parameter) and up
  - Local variables: RBP-8 (1st local variable) and down
  - Callee stores return value in RAX (if function and not procedure)

Calling a function – Callee - Epilogue

- Callee restores all callee-saved registers
- Callee resets the stack to remove the local variables
  - Should be done even if the red zone was used
  - Sometimes used to restore stack in case of exceptions
    - Scanning the code for the epilogue to correctly unwind it
  - **`movq %rbp,%rsp`**
- Callee restores the old frame pointer
  - **`popq %rbp`**
- Caller returns to calling program
  - **`ret`**

Calling a function – Back at Caller

- All local variables have been destroyed
  - □ **Never** attempt to return a pointer to a local variable!
- Future stack pushes will overwrite the values
  - □ Some might still be accessible, as the red zone of the current stack frame might cover them (partially)
  - □ Red zone: Purely an optimization
    - ● Need not set RBP to RSP and need not subtract from RSP
    - ● RSP is used as base for all parameters and local variables
    - ● RBP is no longer needed and can be used as normal register
    - ● Can be used for temporary data and local variables
    - ● **Careful! Further function calls start from RSP and NOT from the end of the red zone, so calling a function will destroy data in it!**
- Return value is in RAX (plus potentially RDX)
- Caller has to remove parameters 7 to N from stack
  - □ **addq $**(N-6)*8**,%rsp** or N-6 times **popq %**<some register>
- Caller restores any saved caller-save registers – **if done**
- Caller re-adjusts stack if any adjustment for alignment was made (or ignores this and just wastes the space until it is cleaned up when it returns itself and resets the stack)

↕ **Can be exchanged**

JOHANNES KEPLER
UNIVERSITÄT LINZ

INSTITUTE
OF NETWORKS
AND SECURITY

https://www.ins.jku.at

THANK YOU FOR YOUR ATTENTION!

Slides by: Michael Sonntag
michael.sonntag@ins.jku.at
+43 (732) 2468 - 4137
S3 235 (Science park 3, 2nd floor)