AMD64 Calling Conventions for Linux / Mac OSX

CSE 378 – Fall 2010, Section – Week 2

CALLING CONVENTIONS

- Calling conventions are a scheme for how functions receive parameters from their caller and how they return a result.
- Adhering to calling conventions ensures that your functions won't step on each other's data when using the same registers.
- Calling conventions allow us to implement recursive functions and call functions which we cannot see the implementations of.
- Certain registers need to have their contents preserved by the caller if the caller wants to ensure that the values in those registers are saved across the function call.
- Other registers need to have their contents saved by the callee (the function being called) before using them.
- Here's a table summarizing what each register is used for and who is responsible for saving its contents:

Register	Use	Saved By
%rax	Returning a value from a function	Caller
%rbx	Optionally used as a base pointer	Callee
%rcx	Used to pass the 4th argument to a function	Caller
%rdx	Used to pass the 3rd argument to a function & optionally to return a second value	Caller
%rsp	Stack pointer	
%rbp	Frame pointer	Callee
%rsi	Used to pass the 2nd argument to a function	Caller
%rdi	Used to pass the 1st argument to a function	Caller
%r8	Used to pass 5th argument to a function	Caller
%r9	Used to pass 6th argument to a function	Caller
%r10	Temporary register also used to pass a function's static chain pointer	Caller
%r11	Temporary register	Caller
%r12	Temporary register	Callee
%r13	Temporary register	Callee
%r14	Temporary register	Callee
%r15	Temporary register	Callee

- Registers are saved by spilling them to the current function's stack frame. This requires allocating space by adjusting the stack pointer and copying the register's contents to the space. An example:

```
subq $8, %rsp
movq %rsi, (%rsp)
```

- Registers are restored by reversing the saving process. E.g.:

```
movq (%rsp), %rsi
addq $8, %rsp
```

THE STACK

- Used for storing static global data and variables local to a function
- In essence a given function's stack frame looks like:

Position	Contents	Frame
8n+16(%rbp)	memory argument eightbyte n	Previous
	...	
16(%rbp)	memory argument eightbyte 0	
8(%rbp)	return address	Current
0(%rbp)	previous %rbp value	
-8(%rbp)	unspecified	
	...	
0(%rsp)	variable size	
-128(%rsp)	red zone	

- The stack must be aligned on a 16 byte boundary when a function is called, so %rsp must be divisible by 16 immediately preceding any 'call' instruction.
- The area up to 128 bytes below the stack pointer is called the red zone. It may be used for temporary storage but can be destroyed by any called function.
- Since leaf functions do not call any other functions, modern compilers (e.g. gcc) often choose not to allocate a stack frame for those functions (keeping %rsp == %rbp) and spill the necessary registers directly to the red zone.

IMPLEMENTING FUNCTIONS

- Each function contains a prologue at the beginning and an epilogue at the end.
- The prologue sets-up the stack frame for the function by saving the base pointer to the stack and moving the base pointer to the top of the stack. Example:

```
pushq %rbp
movq %rsp, %rbp
```

- The epilogue cleans up the stack frame and restores the stack and base pointers to the pre-call values and jumps to the saved return address Example:

```
leave
ret
```

- The caller is responsible for setting-up the arguments for the callee in the appropriate order (%rdi, %rsi, %rdx, %rcx, %r8, %r9)
- The callee function returns its value via %rax.