Notes for CSC 2/454, Nov. 18 and 23, 2020

==

Stack management

Recall allocation strategies: static, stack, heap

Maintaining the Run-Time stack
- Contents of a stack frame
  - bookkeeping: return PC (dynamic link), saved registers,
    static link, (rarely) alignment or interrupt mask information
  - arguments and return value(s)
  - local variables
  - temporaries
- Maintenance of stack is responsibility of "calling sequence" and subroutine "prologue" and "epilogue".
  - space is saved by doing as much work as possible in the prologue and epilogue
  - time ***may*** be saved by doing work in the caller instead, where more information may be known. E.g., there may be fewer registers ***in use*** at the point of call than are used ***somewhere*** in the callee.
  - common strategy is to divide registers into "caller-saves" and "callee-saves" sets.
    - Caller uses the "callee-saves" registers first; "caller-saves" registers if necessary.
    - Callee uses the "caller-saves" registers first; "callee-saves" registers if necessary.

Local variables, parameters, and temporaries are assigned fixed ***offsets*** from the frame pointer or stack pointer at compile time

Variable-length locals and parameters are handled with descriptors (dope vectors). The descriptors are at known offsets. For locals, they are accompanied by a pointer to space higher up in the frame. For value arguments, the pointer points down in the frame.

Stack layout varies significantly from machine to machine and, to some degree, from compiler to compiler.

Many compilers access everything relative to the stack pointer when they can, so the frame pointer can be used for something else. This is not possible w/ variable-sized data in the frame.

Typical modern compiler aims to minimize memory accesses and to rely on simple instructions:
- no special instructions other than `jsr` or `call`
- most arguments passed through registers
  (but space reserved on stack)
- often skip frame pointer
- relatively stable sp (arg build area)
- simple leaf routines make no use of memory at all

Older compilers often used the stack more, and leveraged complex, special-purpose instructions:
- special subroutine-calling instructions to save and update the frame pointer, save registers, branch, and allocate space for the frame all in one or two instructions.
- special `push` and `pop` operations to load/store and update `sp` in one instruction
- (usually) all arguments passed on the stack
- (usually) real frame pointer
- (usually) sp moves up and down as arguments are pushed and popped.
  Convenient for function calls embedded in argument lists.
  No longer done this way on x86, however -- x86-64, esp., makes more use of (now more numerous) registers.

Case study

PLP 4e presents LLVM on ARM-32 (e.g., iPhone) and GCC on x86 (32 & 64)

GCC on x86-64

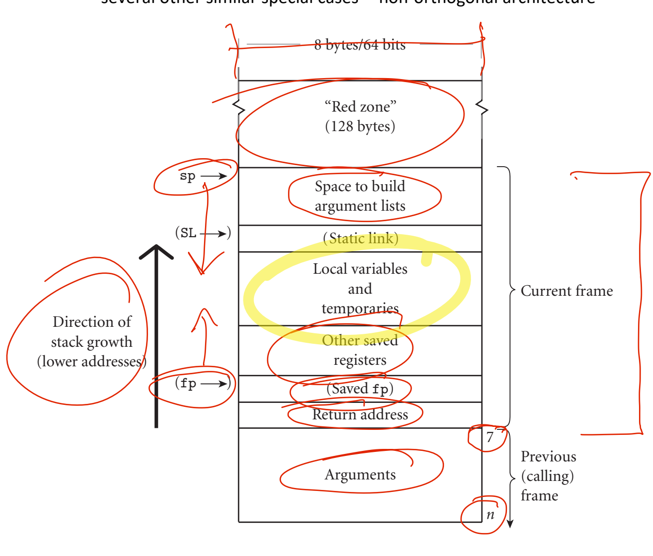
register usage
- 16 64-bit integer registers, 16 128-bit FP/SSE registers
  (various other legacy registers that are not commonly used)
- naming of registers is complicated, due to evolution of the ISA over the years

`rsp`	stack pointer; callee-saved
`rbp`	frame pointer (if used); callee-saved
`rdi, rsi, rdx, rcx, r8, and r9` (in that order)	first 6 integer arguments; caller-saved
rbx, r12, r13, r14, r15	callee-saves temporaries
rax, r10, r11	caller-saves temporaries

static link (if needed) is passed in `r10`.

`rax` and (if needed) `rdx` are used to return function value

`rax` and `rdx` are over-written by division operation
several other similar special cases -- non-orthogonal architecture



Note: In previous incarnations of x86 ABI, SP points to last used location.
On some machines/OSes, it points to first ***unused*** location. ***Beware!***

Actual calling sequence

Caller
1) saves caller-saves registers into temporary locations in current frame, if necessary
2) puts args into registers and (if necessary) the build area at the top of the current frame
3) puts static link in `r10` (skipped for C, or for level-0 callees)
4) executes call

In prologue, Callee
1) pushes `fp` (decrementing `sp` by 8)
2) copies `sp` into `fp`, creating new `fp`
3) pushes callee-saves regs, if necessary
4) subtracts rest of frame size from `sp`

In epilogue, Callee
1) sets return value, if any
2) restores callee-saved regs, if any
3) copies `fp` into `sp`, deallocating frame
4) pops `fp` off stack
5) returns

Steps 3) and 4) can be combined into a one-byte '`leave`' instruction. It's never been entirely clear to me why compilers sometimes generate it and sometimes don't -- perhaps details of timing on particular processor implementations.

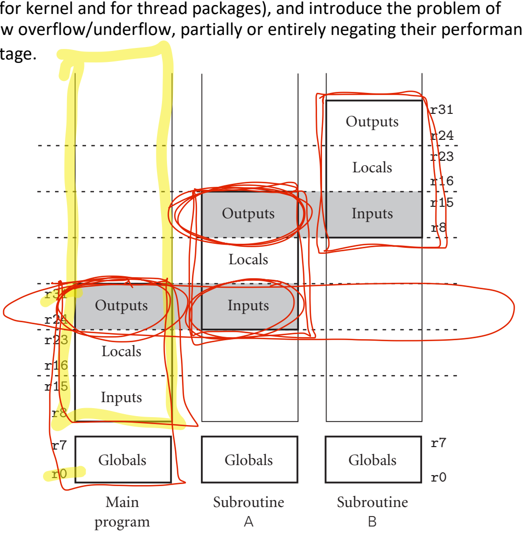
After call, Caller
1) moves return value from register to wherever it's needed (if appropriate)
2) restores caller-saves registers lazily over time, as their values are needed

Many parts of sequence can be elided in special cases.
In particular
- many routines get by w/out `fp`
- red zone lets small leaf routines avoid updating `sp`

Register windows

The Berkeley RISC, and its offspring, the SPARC, use hardware-implemented ***register windows*** in an attempt to reduce the amount of register saving and restoring, and the number of register-register moves.

Unfortunately, these also dramatically complicate context switching (both for kernel and for thread packages), and introduce the problem of window overflow/underflow, partially or entirely negating their performance advantage.



The Itanium (x64) also has register windows, of variable size.

Access to non-local variables via static links

Each frame points to the frame of the (correct instance of) the routine inside which it was declared. In the absence of formal subroutines, "correct" means closest to the top of the stack.

You access a variable in a scope *k* levels out by following *k* static links and then using the known offset within the frame thus found.

You set up static links as follows:
- case 1: callee is nested (directly) inside you
  callee's static link is pointer to your frame
- case 2: callee is *k* scopes out (*k* may be 0)
  callee's static link is found by indirecting off your own static link *k* times

Procedures as parameters:
When you form the closure, you figure out a static link just as if you were going to call the routine directly; the closure consists of the routine's address and the static link.