

Activation Records

- function/procedure calls involve:
 - supply environment for temporary memory, local vars;
 - pass information to the new environment – the parameters
 - return information from the routine – the PC
- need a data structure to maintain this, and a calling protocol
- structure dynamically constructed when a function is called and destroyed upon return
- this lecture: stack layout, calling protocol, and the static link

- Collection of locals, params, PC, temps, etc., maintained in an *activation record* or *stack frame*
- Typically functions are entered and exited in a last-in-first-out order
 - => stack is a suitable structure
 - stack operations not plain push/pop - batch of locals on enter exit... not necessarily of fixed size
 - => keep track of pointers of caller and callee record

What should go in frame

- incoming parameters
 - actually, this is from previous frame
- local variables
 - some may be in registers
- temporary results – working stack
- return address (when making a call to another function) – and other ‘administrative’ info
- outgoing parameters

Layout of frame (almost)

- Frame Pointer points to record of caller function
- Stack Pointer points to top of current record
- access to fields on stack is through fixed offset from FP
 - locals, etc. have negative offset (stack grows towards lower addresses)
 - parameters have positive offsets
- fixed offset from SP? – problem size of saved regs and temps not known until later.

where do arguments/results go?

- what if there are many? what if they are big? – put in memory and pass a reference.
 - e.g. array $a[n]$ – want to be able to access $a[i]$
 - solves issue of dangling reference
- programming language semantics: parameter-passing by value (need to put in memory), by reference (compiler should generate code to pass address and appropriate pointer dereferences)

Registers

- “some local vars & params in regs” – which ones, and how/when is that safe?
- we want to keep local vars in registers whenever possible. who will guarantee that they are not destroyed?
 - caller-save
 - callee-save
 - not necessarily hardware enforced – more a convention

Parameter passing

- keep parameters 1-k in registers, others on stack (e.g., k=4)
- but if f calls g and g calls h, g will have to copy r1-rk as well.
- again – no copy on leaf proc., dead vars, intra-proc allocation...
- convention: allocate params 1-n in outgoing params, but don't copy 1-k; copy params whose address is taken, or if needed (done by callee).

Return address

- on CALL put automatically in a register
- on non-leaf proc have to save it

in Frame

- too many locals and temps
- vars too large
- var passed by reference
- need address arithmetic
- var needed in nested function (so nested func would know where to look...)
- reg needed for specific purpose, e.g. parameter passing

- var *escapes* if needs to go to frame... when can we know this?

Calling protocol

- on call
 - set up parameters
 - save caller-saved regs
- on enter:
 - allocate new frame
 - save old FP, set new FP
 - save callee-saved regs (eager vs lazy?)
- on exit:
 - restore callee-saved regs
 - reset SP, FP
 - jump to PC
 - free up frame

Typical Calling Sequence

1. Call sequence (done by the caller f before entering g) (assume FP and SP in regs)

- f puts arguments a1,...,an onto the stack (or in registers)
- f puts function g's static link onto the stack (or in a register)
- f puts the return address of this call to g onto the stack (or in a register)
- f puts current FP onto the stack (i.e., control link, optional)
- Jump to g's code

2. Entry sequence (the first thing done after entering the callee g)

- move SP to FP
- decrement SP by the frame size of g (stack grows downwards!!!)
- (optional: save callee-save registers if any)

3. Return sequence (the callee g exits and returns back to f)

- put the return result into a designated register
- (optional: restore callee-save registers if any)
- fetch the return address to a register (if in register, do nothing)
- fetch the saved FP of f back to the FP register
- increment SP by the frame size of g (pop off the activation of g)
- Return back to f

Tiger Specifics (also true for many other modern compilers)

- return address is put in a designated register
- only maintain SP at runtime (FP is a "virtual" reg. = SP - framesize) (when implementing Tiger, frame-size of each function is a compile-time constant)
- *Must maintain a separate FP and SP if (1) the frame size of a function may vary (2) the frames are not always contiguous (e.g., linked list)*

Static Link

- pointer to frame which is *statically*, i.e. *lexically*, enclosing the given one
- needed with nested functions, to access vars declared in outer functions
 - not needed if everything defined at top level => 2 scopes, local and global
- procedures have depth known at compile time – static depth

How to determine lexically enclosing scope?

- keep static link in record on stack
- keep array of most recently entered procedure at level *i* (display)
- lambda lifting – pass all variables which will be needed in nested scopes as additional parameters!
- one objective to minimize memory traffic

- we'll use static link or access link
- may need multiple memory accesses to get to variable – depending on nesting depth
- compiler should generate code to access offset within correct frame – i.e. multiple access and pointer deref.
- var in local scope accessed with offset from FP
- vars in outer scope accessed with offsets from SL
- => should provide semantic attribute *level*

Higher-order functions

- Need to keep act. record (local vars, static link) after the function returns
 - permit nested function and functions returned as results
 - problem – function records need to persist after function has returned
 - stack not a good structure in this case