CSE401: “The Stack”

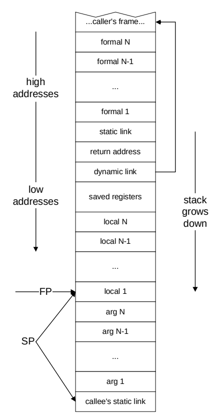
David Notkin
Autumn 2000

Stack frame layout

- Need space for
  - Formals
  - Locals
  - Dynamic link
  - Static link
  - Other run-time data (e.g., return address, saved registers)
- Assign dedicated registers to support access to stack frames
  - Frame pointer (FP): ptr to beginning of stack frame (fixed)
  - Stack pointer (SP): ptr to end of stack (can move)

Key property

- All data in stack frame is at a *fixed, statically computed* offset from the FP
- This makes it easy to generate fast code to access the data in the stack frame
  - And even lexically enclosing stack frames
- Can compute these offsets solely from the symbol tables
  - Based also on the chosen layout approach



Calling conventions

- Need to define responsibilities of caller and callee
  - To make sure the stack frame is properly set up and torn down
- Some things can only be done by the caller
- Other things can only be done by the callee
- Some can be done by either
- So, we need a protocol

PL/0 calling sequence

- Caller
  - Evaluates actual arguments, push them on the stack
    - Order?
    - Alternative: First k arguments in registers
  - Pushes static link of callee on stack
    - Or in register? Before or after stack arguments?
  - Executes call instruction
    - Return address stored in register by hardware
- Callee
  - Saves return address on stack
  - Saves caller’s frame pointer (dynamic link) on stack
  - Saves any other registers that might be needed by caller
  - Allocates space for locals, other data
    - `sp := sp - size_of_locals - other_data`
    - Locals stored in what order?
  - Sets up new frame pointer (`fp := sp`)
  - Starts executing callee’s code

PL/0 return sequence

- Callee
  - Deallocates space for local, other data
    - `sp := sp + size_of_locals + other_data`
  - Restores caller's frame pointer from stack
  - Restores return address from stack
  - Executes return instruction
- Caller
  - Deallocates space for callee's static link, args
    - `sp := fp`
  - Continues execution in caller after call

PL/0 storage allocation

- How and when it is decided how big a stack frame will be?
  - It's necessary that the frame always be the same size for every invocation of a given procedure
- Also, how and when is it decided exactly where in a stack frame specific data will be?
  - Some pieces are decided a priori (such as the return address)
  - Others must be decided during compile-time, such as local variables (since the number and size can't be known beforehand)
- This is all done during the storage allocation phase

PL/0 storage allocation

```
void SymTabScope::allocateSpace() {
    _localsSize  = 0;
    _formalsSize = 0;

    for (int i = 0; i < _symbols->length(); i++) {
        _symbols->fetch(i)->allocateSpace(this);
    }

    for (int j = 0; j < _children->length(); j++) {
      _children->fetch(j)->allocateSpace();
    }
}
```



```
int SymTabScope::allocateFormal(int size) {
    int offset = _formalsSize;
    _formalsSize += size;
    return offset;
}

int SymTabScope::allocateLocal(int size) {
    int offset = _localsSize;
    _localsSize += size;
    return offset;
}

void VarSTE::allocateSpace(SymTabScope* s) {
    int size = _type->size();
    _offset = s->allocateLocal(size);
}
```


Accessing locals

- If a local is in the same stack frame then
  - `t := *(fp + local_offset)`
- If in lexically-enclosing stack frame
  - ```
    t := *(fp + static_link_offset)
    t := *(t + local_offset)
    ```
- If farther away
  - ```
    t := *(fp + static_link_offset)
    t := *(t + static_link_offset)
    …
    t := *(t + local_offset)
    ```


At compile-time…

- …need to calculate
  - Difference in nesting depth of use and definition
  - Offset of local in defining stack frame

Accessing callee procedures

similar to accessing locals

- If calling procedure declared in same stack frame then
  - ```
    static_link := fp
    call p
    ```
- If calling procedure in lexically-enclosing stack frame
  - ```
    static_link := *(fp + static_link_offset)
    call p
    ```
- If farther away
  - ```
    t := *(fp + static_link_offset)
    t := *(t  + static_link_offset)
    …
    static_link := *(t + static_link_offset)
    call p
    ```


Can apply to this example on your own

```
module M;
  var x:int;
  proc P(y:int);
    proc Q(y:int);
      begin R(x+y);end Q;
    proc R(z:int);
      begin P(x+y+z);end R;
    begin Q(x+y);end P;
begin
  x := 1;;
  P(2);
end M.
```


Parameter passing

- When passing arguments, need to support right semantics
- One issue: when is the argument expression evaluated?
  - Before call?
  - If and when needed by callee?
- Another issue: what happens if formal is assigned to in callee?
  - Is this visible to the caller? If so, when?
  - What happens with aliasing among arguments and lexically visible variables?
- Different choices lead to different representations for passed arguments and different code to access formals

Some parameter passing modes

- call-by-value
- call-by-sharing
- call-by-reference
- call-by-value-result
- call-by-name
- call-by-need
- …

Call-by-value

- If formal is assigned, doesn't affect caller's value
- Implement by passing copy of argument value
  - Trivial for scalars
  - Inefficient for aggregates

```
var a : int;
proc foo(x:int,y:int);
begin
  x := x + 1;
  y := y + a;
end foo;

a := 2;
foo(a,a);
output := a;
```


Call-by-sharing

- If implicitly reference aggregate data via pointer (e.g., Java, Lisp, Smalltalk, ML, …) then call-by-sharing is call-by-value applied to implicit pointer
  - "call-by-pointer-value"
  - Efficient, even for big aggregates
  - Assignments of formal to a different aggregate don't affect caller (e.g., `f := x`)
  - Updates to contents of aggregate visible to caller immediately (e.g., `f[i] := x`)
  - Aliasing/sharing relationships are preserved
```

Call-by-reference

- If formal is assigned, actual value is changed in caller
  - Change occurs immediately
  - Assumes actual is an lvalue
- Implement by passing pointer to actual
  - Efficient for big data structures
  - References to formal must do extra dereference

```
var a : int;
proc foo(x:int,y:int);
begin
  x := x + 1;
  y := y + a;
end foo;

a := 2;
foo(a,a);
output := a;
```


Big immutable data

for example, a constant string

- Expensive to pass by-value
- Can implement as call-by-reference
  - Since you can't assign to the data, you don't care

Call-by-value-result

- If formal is assigned, final value is copied back to caller when callee returns
  - "copy-in, copy-out"
- Implement as call-by-value with assignment back when procedure returns
  - More efficient for

```
var a : int;
proc foo(x:int,y:int);
begin
  x := x + 1;
  y := y + a;
end foo;

a := 2;
foo(a,a);
output := a;
```


Call-by-result

```
var a : int;
proc foo(x:int,y:int);
begin
  x := x + 1;
  y := y + a;
end foo;

a := 2;
foo(a,a);
output := a;
```


Ada: in, out, in out

- Programmer selects intent
- Compiler decides what mechanism is more efficient
- Program's meaning "shouldn't" depend on which is chosen

Call-by-name, call-by-need

- Variations on lazy evaluation
  - Only evaluate argument expression if and when needed by callee
- Supports very cool programming tricks
- Hard to implement efficiently in traditional compilers
  - Thunks
- Largely incompatible with side-effects
  - So more common in purely functional languages like Haskell and Miranda