Runtime Issues

Semantics of executing code

- Basic issues
  - Parameter passing
  - Data representation
  - Memory management
  - Calling conventions

- Related issues
  - Runtime checks
  - Object-oriented code

Parameter Passing

What are the semantics of a procedure call?

- Caller: How is argument passed?
- Callee: How are formals accessed?

Issues:

- Comprehension
- Safety
- Efficiency

Terminology:

- l-value / r-value
- argument / formal

Call-by-value

Evaluate r-value of argument

Formal is bound to a copy of the argument

- Formal is a “new” variable

Caller copies argument

- fast for data that fits in registers
- slow for other data

Callee accesses formal directly

- fast for fixed-sized data
- slow for variable-sized data

Used in C, Pascal

Call-by-reference

Evaluate l-value of argument

Formal is bound to l-value

- Formal is the same as the argument

Caller passes pointer to argument

- fast, but argument must reside in memory

Callee must do dereference the pointer

- slow

Used in some Fortran implementations

Provided by Pascal and Modula-3

Call-by-reference in C

```
foo (int *x, int *y, int *z)
{
   *x = *x + 1;
   *y = *y + 1;
   *z = 3;
}

main ()
{
   int a = 2, b;
   foo (&a, &a, &b);
   printf (“%d\n”, a);
   printf (“%d\n”, b);
}
```

Call-by-result

Used to return more than one value

Formal is copied back to argument **when callee returns**

- Formal is a “new” variable

Caller passes pointer to argument

Callee copies final value of formal into argument

Provided by Ada

Call-by-value-result

Combination of call-by-value and call-by-result

- argument is copied into formal
- formal is copied back to argument **when callee returns**
- “copy-in, copy-out”, “copy-restore” (book)

Caller passes l-value of argument

Callee copies argument into formal

Callee copies final value for formal into argument

Used in some Fortran implementations

Call-by-value-result in C

```
foo (int *x, int *y, int *z)
{
   int tx = *x, ty = *y, ty = *z;
   tx = tx + 1;
   ty = ty + 1;
   tz = 3;
   *x = tx; *y = ty; *z = tz;
}
main ()
{
   int a = 2, b;
   foo (&a, &a, &b);
   printf (“%d\n”, a);
   printf (“%d\n”, b);
}
```

Data Representation

Determined by type of data

- scalar data based on machine representation
- aggregates group these together

Integer: use architectural representation
(2,4, and/or 8 bytes of memory, maybe aligned)

Bool: e.g., 0 or 1, 1 byte or word

Char: 1-2 bytes or word

Pointer: use architectural representation
(2,4, or 8 bytes, maybe two words if segmented machine)

Records

Concatenate layout of fields, respecting alignment restrictions

```
r: record
     b: bool;
     i: int;
     m: record
          b: bool;
          c: char;
        end;
     j: int;
   end;
```

Arrays

Repeat layout of element type, respecting alignment

```
a: array [5] of record
                  i: int;
                  c: char;
                end;
```

Array length? dope vector

```
procedure test(x:array[] of int);
   return x[100];
end test;
```

Multi-dimensional Arrays

Recursively apply layout rule to subarray first

```
a: array [3] of array[5] of record
                                i: int;
                                c: char;
                             end;
```

Leads to **row-major** layout

Alternative: **column-major**

Strings

String ≈ array of chars

- can use array layout rule to layout strings

How to determine length of string at run-time?

- Pascal: strings have statically-determined length
- C: special terminating character
- High-level languages: explicit length field

Storage Allocation

Where do we allocate space for each variable and data structure?

Key issue: what is the **lifetime** of a variable/data structure?

- whole execution of program (global variables)
  $\Rightarrow$ **static** allocation
- execution of a procedure activation (arguments, local variables)
  $\Rightarrow$ **stack** allocation
- variable (dynamically-allocated data)
  $\Rightarrow$ **heap** allocation

UNIX Memory Map

Code area

- read-only machine instruction area
- shared across processes running same program

Static data area

- place for read/write variables at fixed location in memory
- can be initialized, or cleared

Heap

- place for dynamically allocation/freed data
- can expand upwards through `sbrk` system call

Stack

- place for stack-allocated/freed data
- expands/contracts downwards automatically

Static Allocation

Statically-allocate variables/data structures with global lifetime

- global variables
- compile-time constant strings, arrays, etc.
- `static` local variables in C, all locals in Fortran

Machine code (text segment)

Compiler uses symbolic address

Linker determines exact address

Stack Allocation

Each procedure's data gets its own **activation record**

An activation record is typically allocated as a **stack frame**.

LIFO (last-in, first-out) discipline matches nesting of procedure calls

Fast to allocate & deallocate stack frame

Good memory locality

Problems with Stack Allocation

Nested functions that escape

```
procedure foo(x:int):proctype(int):int;
var z; z := x + 3;
   procedure bar(y:int):int;
   begin
      return z + y;
   end bar;
begin
   return bar;
end foo;
```

More Problems With Stack Allocation

Addresses of locals escape

```
procedure foo(x:int):&int;
var y:int;
begin
   y := x * 2;
   return &y;
end foo;
```

Stack Frame Layout

- Arguments not passed in registers
- Local variables
- Temporary values
- Dynamic link (pointer to calling stack frame)
- Static link (pointer to lexically-enclosing stack frame)
- Machine state (saved registers, possibly return address)

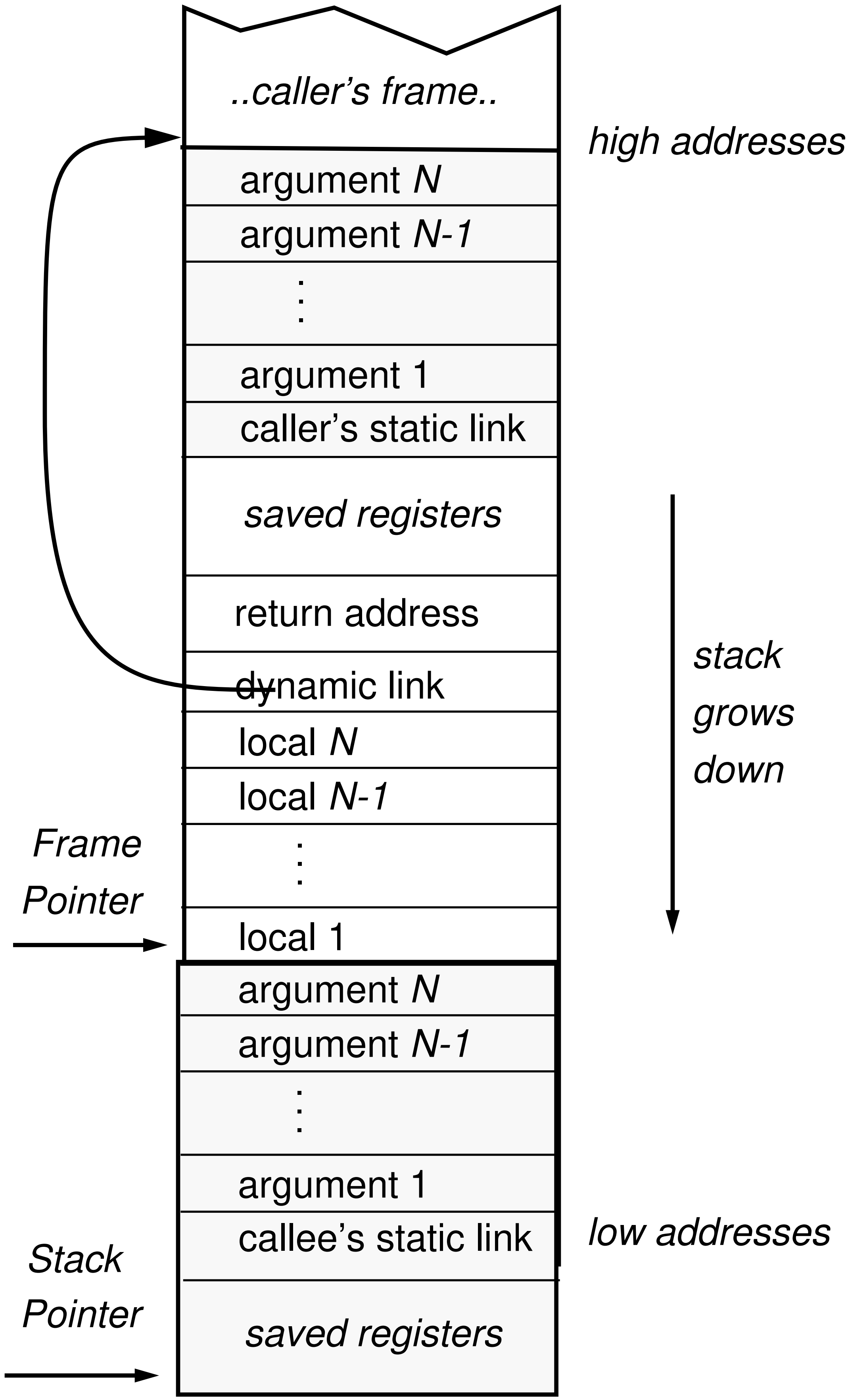
Assign dedicated register(s) to support access to stack frames

- stack pointer (SP): ptr to end of stack

or

- frame pointer (FP): ptr to beginning of stack frame (fixed)
- stack pointer (SP): ptr to end of stack (can move)

PL/0 Stack Layout



Static Linkage

Necessary for lexical scoping of procedures

```
Module M;
var x:int;
procedure P(y:int);
   procedure R(z:int); begin P(x+y+z); end R;
   procedure Q(y:int); begin R(x+y); end Q;
begin
   Q(x+y);
end P;

begin
   x := 1; P(2);
end M.
```

Static Linkage (more)

If in same stack frame:

```
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 + static_link_offset)
t := *(t + local_offset)
```

Dynamic Scoping

Free variables bindings are determined at runtime

```
proc foo (): int;
begin
  return x;
end foo;
```

- harder to understand: free variables are implicit parameters
- used in APL, older Lisp dialects

PL/0 Symbol Table

```
proc foo (f1 : int, f2 : array[5] of bool);
var l1 : array[3] of int;
var l2 : bool;
var l3 : int;
begin ... end foo;
```

...			
proc	“foo”	int, array [5] of bool:void	...
...			

			offset	a formal?
var	“f1”	integer	0	yes
var	“f2”	array[5] of bool	4	yes
var	“l1”	array[3] of int	0	no
var	“l2”	bool	12	no
var	“l3”	int	16	no

PL/0 Symbol Table (more)

```
void SymTabScope::allocateSpace() {
   ...
   foreach sym
      sym->allocateSpace(this);
   foreach child scope
      child->allocateSpace();
}


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

void othersSTE::allocateSpace(SymTabScope* s) {}
```

PL/0 Symbol Table (yet more)

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

Heap Allocation

Heap-allocate data structures with unknown lifetime

- `new/malloc` to allocate space
- implicit allocation (Lisp)

Heap-allocate activation records of first-class functions

More expensive than stack

Heap Reclamation

- Explicit `free` (C)
  - dangling references
  - storage leaks
- Automatic **garbage collection**
  - System deallocates unreachable memory
  - Provided by Lisp, Modula-3, Java

Calling Conventions

Composition of the stack frame:

- How are parameters passed?
- Where is the return value saved?
- What else is on the stack?

Separation of responsibilities between caller and callee
in setting up, tearing down stack frame

Only caller can do some things

Only callee can do other things

Some things could be done by both

PL/0 Calling Sequence

Caller:

- evaluates actual arguments, pushes them on stack
- pushes static link of callee on stack
- saves registers
- 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
- allocates space for locals, other data
- sets up new frame pointer
- starts running code...

PL/0 Return Sequence

Callee:

- deallocates space for locals, other data
  e.g. `sp := sp + size_of_locals + other_data`
- restores caller's frame pointer from stack
- restores return address from stack
- executes return instruction

Caller:

- restores registers
- deallocates space for callee's static link, args
  e.g. `sp := fp`
- continues execution...