# CSE 307: Principles of Programming Languages

## Runtime Environments

R. Sekar

# Topics

# Components of Runtime Environment (RTE)

Static area: allocated at load/startup time.

- Examples: global/static variables and load-time constants.

Stack area: for execution-time data that obeys a last-in first-out lifetime rule.

- Examples: nested declarations and temporaries.

Heap: a dynamically allocated area for "fully dynamic" data, i.e. data that does not obey a LIFO rule.

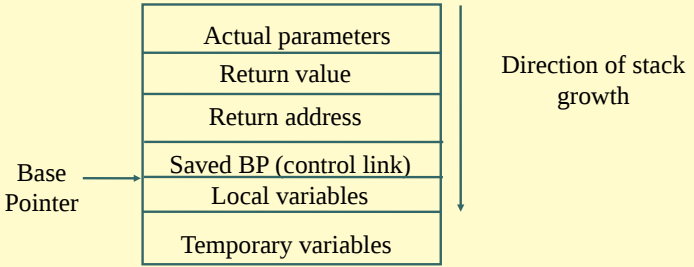
- Examples: objects in Java, lists in OCaml.

# Languages and Environments

- Languages differ on where activation records must go in the environment:
  - (Old) Fortran is static: all data, including activation records, are statically allocated.
    - Each function has only one activation record — no recursion!
- Functional languages (Scheme, ML) and some OO languages (Smalltalk) are heap-oriented:
  - almost all data, including AR, allocated dynamically.
- Most languages are in between: data can go anywhere
  - ARs go on the stack.

# Procedures and the environment

- An Activation Record (AR) is created for each invocation of a procedure
- Structure of AR:



# Access to Local Variables

- Local variables are allocated at a fixed offset on the stack
  - Accessed using this constant offset from BP
    - Example: to load a local variable at offset 8 into the EBX register (x86 architecture)

      mov 0x8(%ebp),%ebx

- Example:

```
{int x; int y;
   { int z; }
   { int w; }
}
```

# Steps involved in a procedure call

- Caller
  - Save registers
  - Evaluate actual parameters, push on the stack
    - Push l-values for CBR, r-values in the case of CBV
  - Allocate space for return value on stack (unless return is through a register)
  - Call: Save return address, jump to the beginning of called function
- Callee
  - Save BP (control link field in AR)
  - Move SP to BP
  - Allocate storage for locals and temporaries (Decrement SP)
  - Local variables accessed as [BP-k], parameters using [BP+l]

# Steps in return

- Callee
  - Copy return value into its location on AR
  - Increment SP to deallocate locals/temporaries
  - Restore BP from Control link
  - Jump to return address on stack
- Caller
  - Copy return values and parameters
  - Pop parameters from stack
  - Restore saved registers

# Example (C):

```
int x;
void p(int y){
   int i = x;
   char c; ...
}
void q (int a){
   int x;
   p(1);
}
main(){
   q(2);
   return 0;
}
```

# Non-local variable access

- Requires that the environment be able to identify frames representing enclosing scopes.
- Using the control link results in dynamic scope (and also kills the fixed-offset property).
- If procedures can't be nested (C), the enclosing scope is always locatable:
  - it is global/static (accessed directly)
- If procedures can be nested (Ada, Pascal), to maintain lexical scope a new link must be added to each frame:
  - access link, pointing to the activation of the defining environment of each procedure.

# Access Link vs Control Link

- Control Link is a reference to the AR of the caller
- Access link is a reference to the AR of the surrounding scope
- **Dynamic Scoping:** When an identifier is not found in the AR of the current function, use *control link* to get to the caller's AR and look up the name there
  - If not found, follow the caller's control link, and then its caller's control link and so on
- **Static Scoping:** When an identifier is not found in the AR of the current function, use *access link* to get to AR for the surrounding scope and look up the name there
  - If it is not found there, keep walking through the access links until the name is found.
- **Note:** Except for top-level functions, access links correspond to function scopes, so they cannot be determined statically
  - They need to be "passed in" like a parameter

# Access Link Vs Control Link: Example

```
int q(int x) {
  int p(int y) {
    if (y==0)
      return x+y;
    else {
      int x = 2*p(y-1);
      return x;
    }
  }

  return p(3);
}
```

- If p used its caller's BP to access x, then it can end up accessing the variable x defined within p
  - This would be dynamic scoping.
  - To get static scoping, this access should use q's BP
- *Access link:* Have q pass a link to its BP explicitly.
  - Calls to self: pass access link without change.
  - Calls to immediately nested functions: pass your BP
  - Calls to outer functions: Follow your access link to find the right access link to pass
  - Other calls: these will be invalid (like `goto` to an inner block)

# Supporting Closures

- *Closures* are needed for
  - Call-by-name and lazy evaluation
  - Returning dynamically constructed functions containing references to variables in surrounding scope
- Variables inside closures may be accessed long after the functions defining them have returned
  - Need to “copy” variable values into the closure, or
  - Not free the AR of functions when they return,
    - i.e., allocate ARs on heap and garbage collect them

# Implementation of Exception Handling

- Exception handling can be implemented by adding “markers” to ARs to indicate the points in program where exception handlers are available.
- In C++, entering a try-block at runtime would cause such a marker to be put on the stack
- When exception arises, the RTE gets control and searches down from stack top for a marker.
- Exception then "handed" to the catch statement of this try-block that matches the exception
- If no matching catch statement is present, search for a marker is continued further down the stack, and the whole process is repeated.

# Heap management

- Issues
  - No LIFO property, so management is difficult
  - Fragmentation
  - Locality
- Models
  - Explicit allocation and free (C, C++)
  - Explicit allocation, automatic free (Java)
  - Automatic allocation and free (OCAML)

# Allocation

- A variable is stored in memory at a location corresponding to the variable.
- Constants do not need to be stored in memory.
- Environment stores the binding between variable names and the corresponding locations in memory.
- The process of setting up this binding is known as storage allocation.

# Static Allocation

- static allocation
  - Allocation performed at compile time.
  - Compiler translates all names to corresponding location in the code generated by it.
  - Examples:
    - all variables in original FORTRAN
    - all global and static variables in C/C++/Java

# Stack Allocation

- Needed in any language that supports the notion of local variables for procedures.
- Also called “automatic allocation”, but this is somewhat of a misnomer now.
- Examples: all local variables in C/C++/Java procedures and blocks.
- Implementation:
  - Compiler translates all names to relative offsets from a location called the “base pointer” or “frame pointer”.
  - The value of this pointer varies will, in general, be different for different procedure invocations

# Stack Allocation (Continued)

- The pointer refers to the base of the “activation record” (AR) for an invocation of a procedure.
- The AR holds such information as parameter values, local variables, return address, etc.

```
int fact(int n){
   if n=0 then 1
   else{
      int rv = n*fact(n-1);
      return rv;
   }
}
main(){
   fact(5);
}
```

# Stack Allocation (Continued)

- An activation record is created on the stack for each a call to function.
- The start of activation record is pointed to by a register called BP.
- On the first call to fact, BP is decremented to point to new activation record, n is initialized to 5, rv is pushed but not initialized.
- New activation record is created for the next recursive call and so on.
- When n becomes 0, stack is unrolled where successive rv's are assigned the value of n at that stage and the rv of previous stage.

# Heap Management

- Issues
  - No LIFO property, so management is difficult
  - Fragmentation
  - Locality
- Models
  - explicit allocation, free
    - e.g., malloc/free in C, new/delete in C++
  - explicit allocation, automatic free
    - e.g., Java
  - automatic allocation, automatic free
    - e.g., Lisp, OCAML, Python, JavaScript

# Fragmentation

Internal fragmentation: When asked for $x$ bytes, allocator returns $y > x$ bytes

- $y - x$ represents internal fragmentation

External fragmentation: When (small) free blocks of memory occur in between (i.e., external to) allocated blocks

- the memory manager may have a total of $M \gg N$ bytes of free memory available, but no contiguous block larger enough to satisfy a request of size $N$.

# Approaches for Reducing Fragmentation

- Use blocks of single size (early LISP)
  - Limits data-structures to use less efficient implementations.
- Use bins of fixed sizes, e.g., $2^n$ for $n = 0, 1, 2, \ldots$
  - When you run out of blocks of a certain size, break up a block of next available size
  - Eliminates external fragmentation, but increases internal fragmentation
- Maintain bins as LIFO lists to increase locality
- malloc implementations (Doug Lea)
  - For small blocks, use bins of size 8k bytes, $0 < k < 64$
  - For larger blocks, use bins of sizes $2^n$ for $n > 9$

# Coalescing

- What if a program allocates many 8 byte chunks, frees them all and then requests lots of 16 byte chunks?
  - Need to coalesce 8-byte chunks into 16-byte chunks
  - Requires additional information to be maintained
    - for allocated blocks: where does the current block end, and whether the next block is free

# Explicit Vs Automatic Management

- Explicit memory management can be more efficient, but takes a lot of programmer effort
- Programmers often ignore memory management early in coding, and try to add it later on
  - But this is very hard, if not impossible
- Result:
  - Majority of bugs in production code is due to memory management errors
    - Memory leaks
    - Null pointer or uninitalized pointer access
    - Access through dangling pointers

# Managing Manual Deallocation

- How to avoid errors due to manual deallocation of memory
  - Never free memory!!!
  - Use a convention of object ownership (owner responsible for freeing objects)âĂŔ
    - Tends to reduce errors, but still requires a careful design from the beginning. (Cannot ignore memory deallocation concerns initially and add it later.)âĂŔ
  - Smart data structures, e.g., reference counting objects
  - Region-based allocation
    - When a collection of objects having equal life time are allocated
    - Example: Apache web server's handling of memory allocations while serving a HTTP request

# Garbage Collection

- Garbage collection aims to avoid problems associated with manual deallocation
  - Identify and collect garbage automatically
- What is garbage?
  - Unreachable memory
- Automatic garbage collection techniques have been developed over a long time
  - Since the days of LISP (1960s)

# Garbage Collection Techniques

- Reference Counting
  - Works if there are no cyclic structures
- Mark-and-sweep
- Generational collectors
- Issues
  - Overhead (memory and space)âĂŔ
  - Pause-time
  - Locality

# Reference Counting

- Each heap block maintains a count of the number of pointers referencing it.
- Each pointer assignment increments/decrements this count
- Deallocation of a pointer variable decrements this count
- When reference count becomes zero, the block can be freed

# Reference Counting (Continued)

Disadvantages:

- Does not work with cyclic structures
- May impact locality
- Increases cost of each pointer update operation

Advantages:

- Overhead is predictable, fixed
- Garbage is collected immediately, so more efficient use of space

# Mark-and-Sweep

- Mark every allocated heap block as “unreachable”
- Start from registers, local and global variables
- Do a depth-first search, following the pointers
  - Mark each heap block visited as “reachable”
- At the end of the sweep phase, reclaim all heap blocks still marked as unreachable

# Garbage Collection Issues

- Memory fragmentation
  - Memory pages may become sparsely populated
  - Performance will be hit due to excessive virtual memory usage and page faults
  - Can be a problem with explicit memory management as well
    - But if a programmer is willing to put in the effort, the problem can be managed by freeing memory as soon as possible
- Solution:
  - Compacting GC
    - Copy live structures so that they are contiguous
  - Copying GC

# Copying Garbage Collection

- Instead of doing a sweep, simply copy over all reachable heap blocks into a new area
- After the copying phase, all original blocks can be freed
- Now, memory is compacted, so paging performance will be much better
- Needs up to twice the memory of compacting collector, but can be much faster
  - Reachable memory is often a small fraction of total memory

# Generational Garbage Collection

- Take advantage of the fact that most objects are short-lived
- Exploit this fact to perform GC faster
- Idea:
  - Divide heap into generations
  - If all references go from younger to older generation (as most do), can collect youngest generation without scanning regions occupied by other generations
  - Need to track references from older to younger generation to make this work in all cases

# Garbage collection in Java

- Generational GC for young objects
- “Tenured” objects stored in a second region
  - Use mark-and-sweep with compacting
- Makes use of multiple processors if available
- References

  http://java.sun.com/javase/technologies/hotspot/gc/gc_tuning_6.html

  http://www.ibm.com/developerworks/java/library/j-jtp11253/

# GC for C/C++: Conservative Garbage Collection

- Cannot distinguish between pointers and nonpointers
  - Need “conservative garbage collection”
- The idea: if something “looks” like a pointer, assume that it may be one!
  - Problem: works for finding reachable objects, but cannot modify a value without being sure
    - Copying and compaction are ruled out!
- Reasonable GC implementations are available, but they do have some drawbacks
  - Unpredictable performance
  - Can break some programs that modify pointer values before storing them in memory