Run-time Systems
Code Genaration

Run-time Support

- The target program interacts with system resources.
- There is a need to manage memory when a program is running
  - This memory management must connect to the data objects of programs
  - Programs request for memory blocks and release memory blocks
  - Passing parameters to functions needs attention
- Other resources such as printers, file systems, etc., also need to be accessed

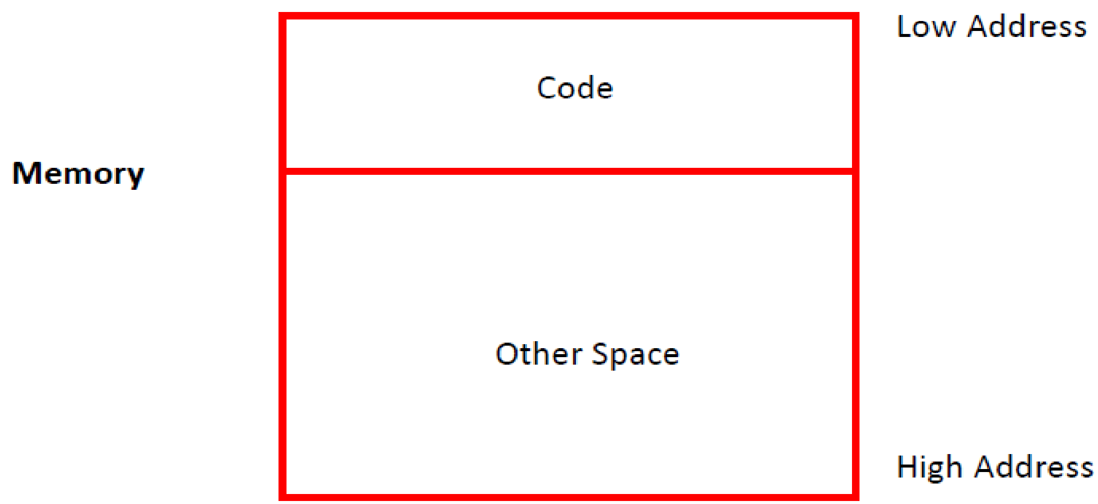
Runtime Support

- Execution of a program is initially under the control of the operating system
- When a program is invoked:
  - The OS allocates space for the program
  - The code is loaded into part of the space
  - The OS jumps to the entry point (i.e., “main”)

Management of Run-time Resources

- The compiler is not only responsible for generating code but also handling the associated data
- Compiler needs to decide what the layout of data is going to be and then generate code that correctly manipulates the data
  - References data from within code
  - Code and layout of data needs to be designed together

- Storage Organization



Procedure Activations

- Two goals in code generation:
  - Correctness
  - Speed
- Fast as well as correct – Difficult

- Two assumptions of Activation:
  - Execution is sequential; control moves from one point in a program to another in a well-defined order
  - When a procedure is called, control always returns to the point immediately after the call

Procedure Activations

- An invocation of procedure **P** is an *activation of* ***P***

- The *lifetime of an activation of* ***P*** *is*
    - All the steps to execute **P**
    - Including all the steps in procedures which **P** calls

- The *lifetime of a variable x is the portion of execution in which x is defined*

- Note that
    - Lifetime is a dynamic (run-time) concept
    - Scope is a static concept

Procedure Activations

- Observation
  - When **P** calls **Q**, then **Q** returns before **P** returns

- Lifetimes of procedure activations are properly nested
- Activation lifetimes can be depicted as a tree

```
class Main {
        int g() { return 1; };
        int f()  { return g(); };
        int main() {{ g(); f(); }};
}
```

Procedure Activations

```
class main {
        int g() : { return1; };
        int f(int x) { if (x == 0) then return g(); else

                return 1+ f(x - 1); };
        int main()  { f(3); };
}
```

Nesting of activations

- The activation tree may be different for every program input
- Since activations are properly nested, a stack can track currently active procedures

Activation Records

- Information needed to manage one procedure activation is called an *activation record (AR) or frame*
- If procedure **F** calls **G**, then **G**'s activation record contains a mix of info about **F** and **G**.
- **F** is "suspended" until **G** completes, at which point **F** resumes
- **G**'s AR contains information needed to
  - Complete execution of **G**
  - Resume execution of **F**

Activation Records

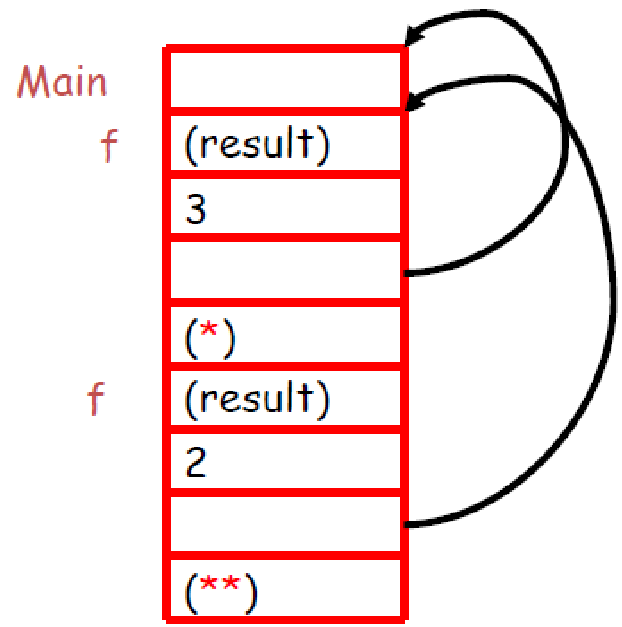
- Space for **G**'s return value
- Actual parameters
- Pointer to the previous activation record
  - The *control link; points to AR of caller of* **G**
- Machine status prior to calling **G**
  - Contents of registers & program counter
  - Local variables
- Other temporary values

Activation Records

```
class main {
        int g() : { return1; };
        int f(int x) { if (x == 0) then return g();
                       else return 1+ f(x - 1);(**) };
        int main()  {f(3); (*)};
}
```

- AR:

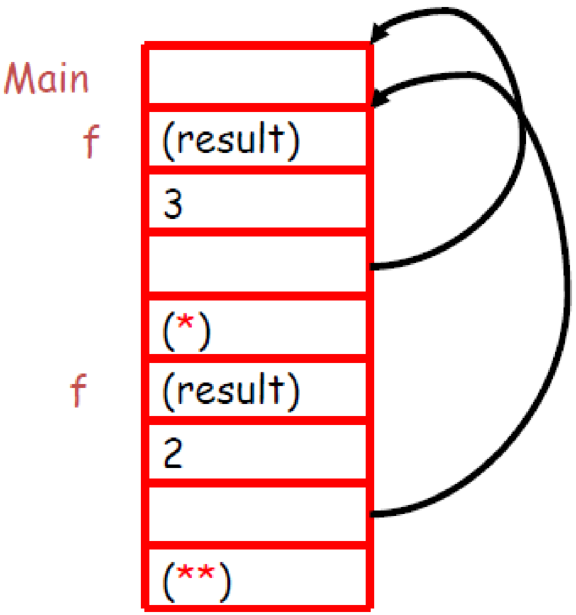
result
argument
control link
return address



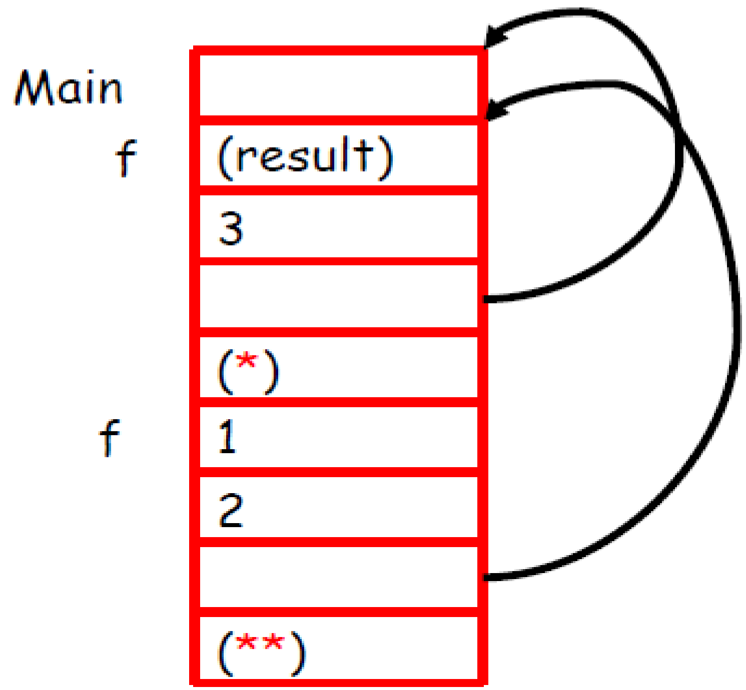
- **Main** has no argument or local variables and its result is never used; its AR is uninteresting
- (*) and (**) are return addresses of the invocations of **f**

Activation Records

- The advantage of placing the return value 1st in a frame is that the caller can find it at a fixed offset from its own frame

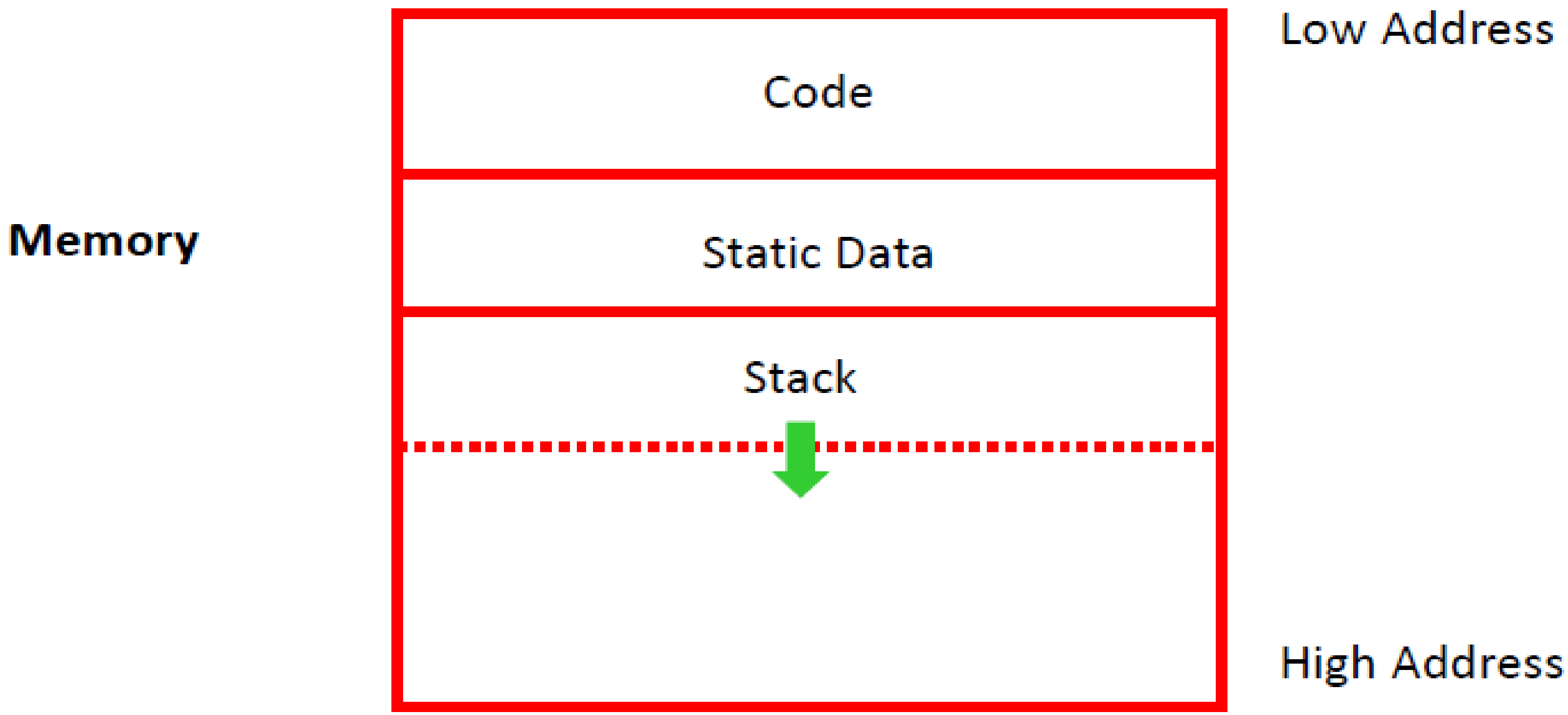


Stack state after the call to the 2nd invocation of **f** returns



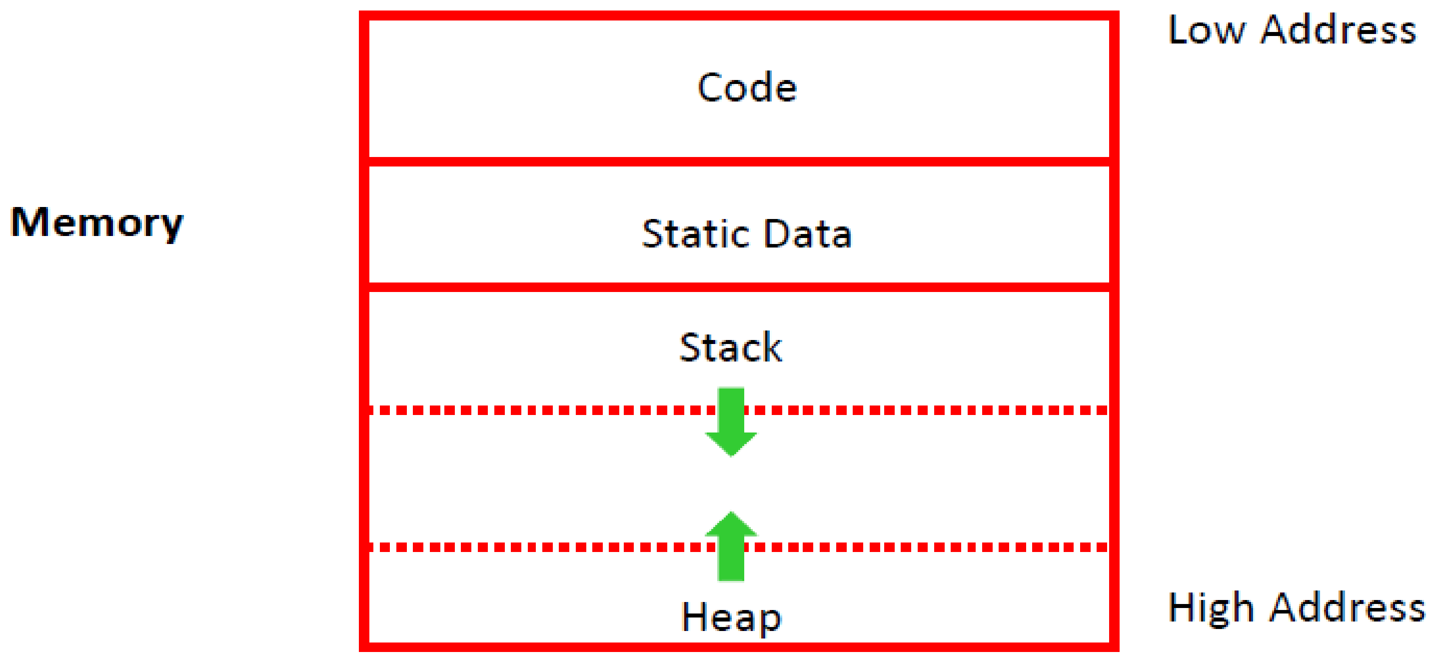
Global Data

- All references to a global variable point to the same object
  - Can't store a global in an activation record
- Globals are assigned a fixed address statically



Heap

- A value that outlives the procedure that creates it cannot be kept in AR
    - int foo() { malloc (Bar) }
    - The Bar value must survive deallocation of foo's AR
- A *heap* is generally used to store dynamically allocated data



Heap Memory Management

- Heap is used for allocating space for objects created at run time
  - For example: nodes of dynamic data structures such as linked lists and trees
- Dynamic memory allocation and deallocation based on the requirements of the program
  - *malloc()* and *free()* in C programs
  - *new()* and *delete()* in C++ programs
  - *new()* and garbage collection in Java programs
- Allocation and deallocation may be *completely manual* (C/C++), *semi-automatic* (Java), or *fully automatic* (Lisp)

Memory Manager

- Manages heap memory by implementing mechanisms for allocation and deallocation, both manual and automatic
- Goals
  - Space efficiency: minimize fragmentation
  - Program efficiency: take advantage of locality of objects in memory and make the program run faster
  - Low overhead: allocation and deallocation must be efficient
- Heap is maintained either as a doubly linked list or as bins of free memory chunks (more on this later)

Allocation and Deallocation

- In the beginning, the heap is one large and contiguous block of memory
- As allocation requests are satisfied, chunks are cut off from this block and given to the program
- As deallocations are made, chunks are returned to the heap and are free to be allocated again (*holes*)
- After a number of allocations and deallocations, memory is fragmented and not contiguous
- Allocation from a fragmented heap may be made either in a *first-fit* or *best-fit* manner
- After a deallocation, we try to *coalesce* contiguous holes and make a bigger hole (free chunk)

Heap Fragmentation

busy	free	busy	free	busy	busy	free
100K	50K	20K	50K	200K	30K	50K

- To begin with the whole heap is a single chunk of size 500K bytes
- After a few allocations and deallocations, there are holes
- In the above picture, it is not possible to allocate 100K or 150K even though total free memory is 150K

First-Fit and Best-Fit Allocation Policies

- The *first-fit* strategy picks the **first** available chunk that satisfies the allocation request
- The *best-fit* strategy searches and picks the smallest (**best**) possible chunk that satisfies the allocation request
- Both of them chop off a block of the required size from the chosen chunk, and return it to the program
- The rest of the chosen chunk remains in the heap

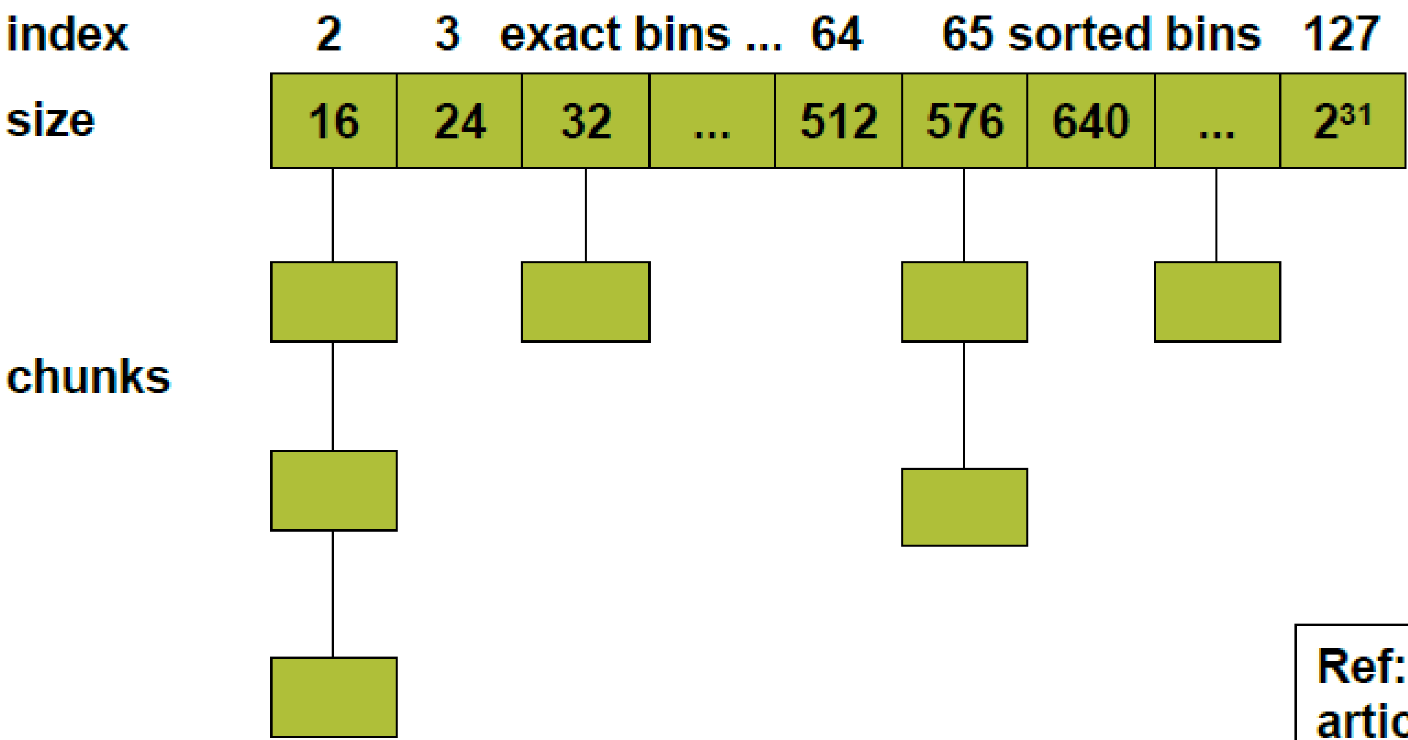
Next-Fit Allocation Policy

- Best-fit strategy has been shown to reduce fragmentation in practice, better than first-fit strategy
- *Next-fit* strategy tries to allocate the object in the chunk that has last been split
  - Tends to improve speed of allocation
  - Tends to improve spatial locality since objects allocated at about the same time tend to have similar reference patterns and life times (cache behaviour may be better)

Bin-Based Heap

- Free space organized into *bins* according to their sizes (**Lea Memory Manager in GCC)**
  - Many more bins for smaller sizes, because there are many more small objects
  - A bin for every multiple of 8-byte chunks from 16 bytes to 512 bytes
  - Then approximately logarithmically (double previous size)
  - Within each "small size bin", chunks are all of the same size
  - In others, they are ordered by size
  - The last chunk in the last bin is the *wilderness chunk*, which gets us a chunk by going to the operating system

Bin-Based Heap



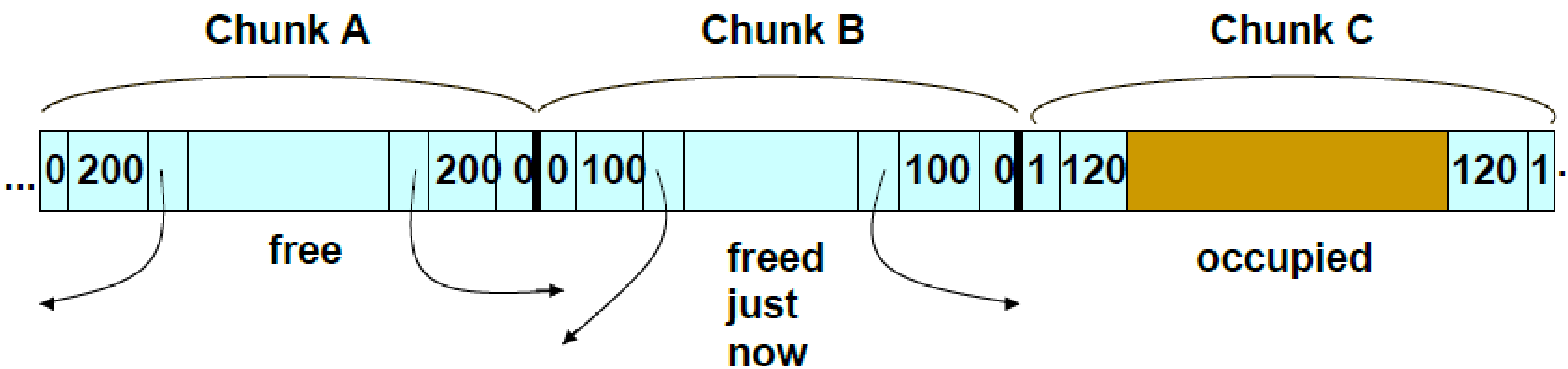
Ref: From Lea's article on memory manager in GCC

Managing and Coalescing Free Space

- Should coalesce adjacent chunks and reduce fragmentation
  - Many small chunks together cannot hold one large object
  - In the Lea memory manager, no coalescing in the exact size bins, only in the sorted bins
  - *Boundary tags* (free/used bit and chunk size) at each end of a chunk (for both used and free chunks)
  - A *doubly linked list* of free chunks

Boundary Tags and Doubly Linked Lists

3 adjacent chunks. Chunk B has just been deallocated and returned to the free list. Chunks A and B can be merged, and this is done just before inserting it into the linked list. The merged chunk AB may have to be placed in a different bin.



Problems with Manual Deallocation

- Memory leaks
  - Failing to delete data that cannot be referenced
  - Important in long running or nonstop programs
- Dangling pointer dereferencing
  - Referencing deleted data
- Both are serious and hard to debug

Garbage Collection

- Reclamation of chunks of storage holding objects that can no longer be accessed by a program
- GC should be able to determine types of objects
  - Then, size and pointer fields of objects can be determined by the GC
  - Languages in which types of objects can be determined at compile time or run-time are type safe
    - Java is type safe
    - C and C++ are not type safe because they permit type casting, which creates new pointers
    - Thus, any memory location can be (theoretically) accessed at any time and hence cannot be considered inaccessible

Reachability of Objects

- The *root set* is all the data that can be accessed (reached) directly by a program without having to dereference any pointer
- Recursively, any object whose reference is stored in a field of a member of the root set is also reachable
- New objects are introduced through object allocations and add to the set of reachable objects
- Parameter passing and assignments can propagate reachability
- Assignments and ends of procedures can terminate reachability

Reachability of Objects

- Similarly, an object that becomes *unreachable* can cause more objects to become unreachable
- A garbage collector periodically finds all unreachable objects by one of the two methods
  - Catch the transitions as reachable objects become unreachable
  - Or, periodically locate all reachable objects and infer that all *other* objects are unreachable

Reference Counting GC

- This is an approximation to the first approach mentioned before
- We maintain a count of the references to an object, as the mutator (program) performs actions that may change the reachability set
- When the count becomes zero, the object becomes unreachable
- Reference count requires an extra field in the object and is maintained as below

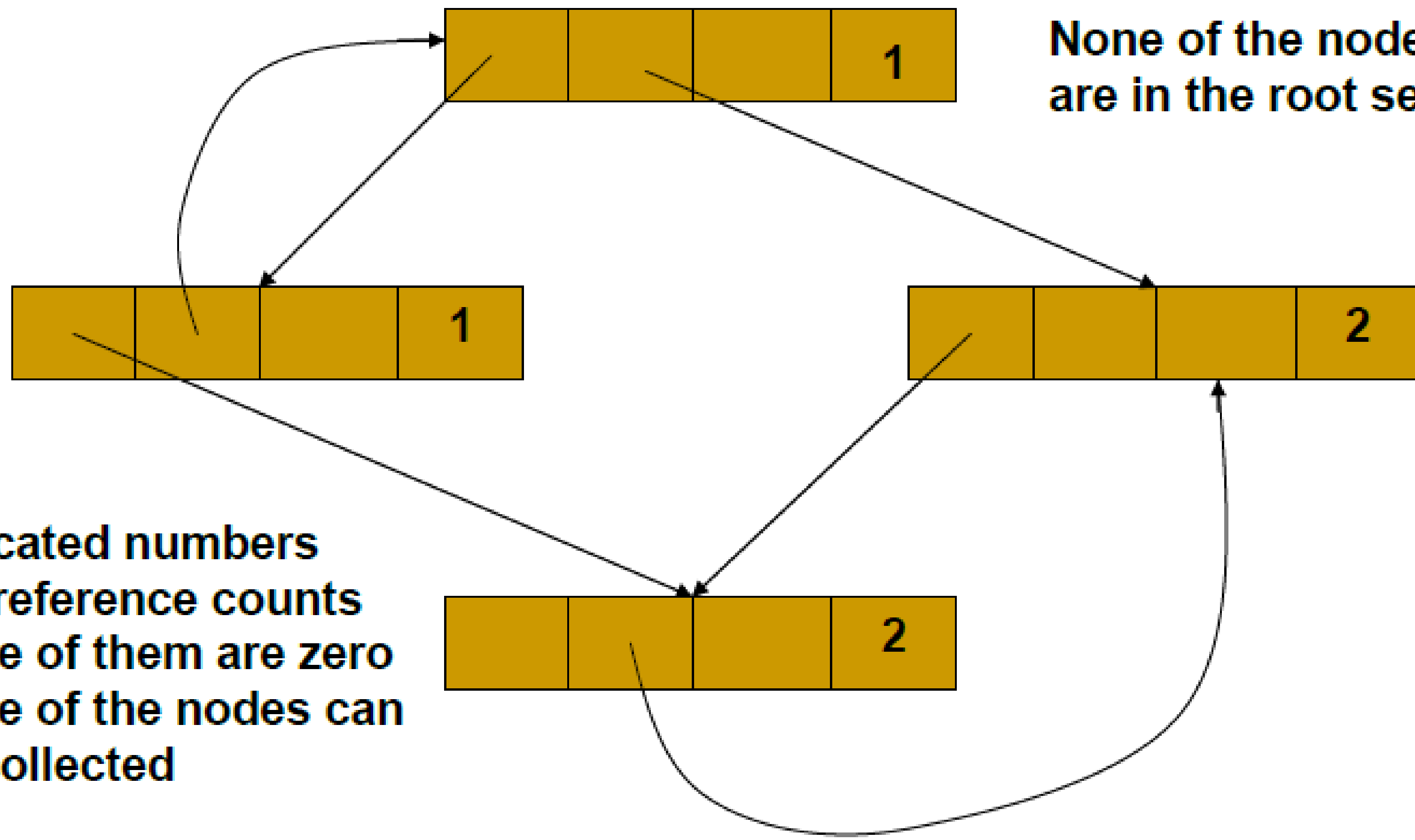
Maintaining Reference Counts

- *New object allocation.* ref_count=1 for the new object
- *Parameter passing.* ref_count++ for each object passed into a procedure
- *Reference assignments.* For u:=v, where u and v are references, ref_count++ for the object *v, and ref_count-- for the object *u
- *Procedure returns.* ref_count-- for each object pointed to by the local variables
- *Transitive loss of reachability.* Whenever ref_count of an object becomes zero, we must also decrement the ref_count of each object pointed to by a reference within the object

Reference Counting GC: Pros and Cons

- High overhead due to reference maintenance
- Cannot collect unreachable cyclic data structures (ex: circularly linked lists), since the reference counts never become zero
- Garbage collection is incremental
  - overheads are distributed to the mutator's operations and are spread out throughout the life time of the mutator
- Garbage is collected immediately and hence space usage is low
- Useful for real-time and interactive applications, where long and sudden pauses are unacceptable

Unreachable Cyclic Data Structure



None of the nodes are in the root set

Indicated numbers are reference counts
None of them are zero
None of the nodes can be collected

Mark-and-Sweep GC

- Memory recycling steps
  - Program runs and requests memory allocations
  - GC traces and finds reachable objects
  - GC reclaims storage from unreachable objects
- Two Phases
  - Marking reachable objects
  - Sweeping to reclaim storage
- Can reclaim unreachable objects in cyclic data structures
- Stop-the-world algorithm

Mark-and-Sweep GC : Mark

```
/* marking phase */
1. Start scanning from root set, mark all reachable objects
   (set reached-bit = 1), place them on the list Unscanned
2. while (Unscanned ≠ Φ) do
   { object o = delete(Unscanned);
     for (each object o1 referenced in o) do
         { if (reached-bit(o1) == 0)
               { reached-bit(o1) = 1; place o1 on Unscanned;}
         }
   }
```

Mark-and-Sweep GC : Sweep

- /* Sweeping phase, each object in the heap is inspected only once */

```
3. Free = Φ;
   for (each object o in the heap) do
     { if (reached-bit(o) == 0)   add(Free, o);
       else reached-bit(o) = 0;
     }
```

Code Generation: Stack Machines

- Only storage is a stack
- An instruction $r = F(a_1,\ldots a_n)$:
  - Pops **n** operands from the stack
  - Computes the operation **F** using the operands
  - Pushes the result **r** on the stack

- Consider two instructions:
  - push i - push integer i on the stack
  - add - add two integers

Stack Machines

- A program:
  - `push 7`
  - `push 5`
  - `Add`

- Stack machines provide a simple machine model
  - Simple compiler
  - Inefficient

- Location of the operands/result is not explicitly stated
  - Always the top of the stack

Stack Machines

- Stack machine Vs. register machine
  - **add** instead of **add r_1, r_2, r_3**

- There is an intermediate point between a pure stack machine and a pure register machine
- An *n-register stack machine*
  - Conceptually, keep the top *n locations of the pure stack machine's stack in registers*

- 1-register stack machine
  - The register is called the *accumulator*

Stack Machines

- In a pure stack machine
  - An **add** does **3** memory operations: Two reads and one write
- In a 1-register stack machine the **add** does
  - acc ← acc + top_of_stack
- In general, for an operation $\mathbf{op(e_1,\ldots,e_n)}$
  - $\mathbf{e_1,\ldots,e_n}$ are subexpressions
- For each $\mathbf{e_i}$ $\mathbf{(0 < i < n)}$
  - Compute $\mathbf{e_i}$
  - Push result on the stack
- Pop **n-1** values from the stack, compute **op**
- Store result in the accumulator

Stack Machines

***Operations for the stack machine with accumulator*: 3 + (7 + 5)**

Code	**Acc**	**Stack**
acc ← 3	3	<init>
push acc	3	3, <init>
acc ← 7	7	3, <init>
push acc	7	7, 3, <init>
acc ← 5	5	7, 3, <init>
acc ← acc + top_of_stack	12	7, 3, <init>
pop	12	3, <init>
acc ← acc + top_of_stack	15	3, <init>
pop	15	<init>

Code Generation

- Code that can be executed on a real machine
  - The MIPS processor
- We will simulate a stack machine using MIPS instructions and registers
  - The accumulator is kept in MIPS register $a0
  - The stack is kept in memory
    - The stack grows towards lower addresses in MIPS
  - Address of the next location on stack is kept in register $sp
    - Top of the stack is at address $sp + 4

- MIPS uses RISC processor model
- 32 general purpose registers (32 bits each)
- We use $sp, $a0 and $t1 (a temporary register)

Code Generation

- lw reg1 offset(reg2)
  - Load 32-bit word from address reg_2 + offset into reg_1
- add reg_1 reg_2 reg_3
  - $reg_1 \leftarrow reg_2 + reg_3$
- sw reg1 offset(reg_2)
  - Store 32-bit word in reg_1 at address reg_2 + offset
- addiu reg_1 reg_2 imm
  - $reg_1 \leftarrow reg_2 + imm$
  - "u" means overflow is not checked
- li reg imm
  - $reg \leftarrow imm$

Code Generation

- Stack-machine code for **7 + 5** in MIPS

acc ← 7	`li`	`$a0`	`7`	
push acc	`sw`	`$a0`	`0($sp)`	
	`addiu`	`$sp`	`$sp -4`	
acc ← 5	`li`	`$a0`	`5`	
acc ← acc + top_of_stack	`lw`	`$t1`	`4($sp)`	
	`add`	`$a0`	`$a0`	`$t1`
pop	`addiu`	`$sp`	`$sp`	`4`

Code Generation

- A language with integers and integer operations
- $P \rightarrow D; P \mid D$
- $D \rightarrow \text{def id(ARGS)} = E;$
- $\text{ARGS} \rightarrow \text{id, ARGS} \mid \text{id}$

- A program consists of a list of declarations

- A declaration is a function definition.
- The function takes a list of identifiers as arguments.
- The function body is an expression.

- $E \rightarrow \text{int} \mid \text{id} \mid \text{if } E_1 = E_2 \text{ then } E_3 \text{ else } E_4$
  $\mid E_1 + E_2 \mid E_1 - E_2 \mid \text{id}(E_1, \ldots, E_n)$

- Expressions are integers, identifiers, if-then-else with a predicate which allows the equality test, sums and differences of expressions and function calls.

- The first function definition in the list is the entry point, that is the *main* routine.

Code Generation

- This language may be used to define the fibonacci function:

```
def fib(x) = if x = 1 then 0 else
             if x = 2 then 1 else
                fib(x - 1) + fib(x – 2)
```

- To generate code for this language, we generate MIPS code for each expression **e** that:
  - Computes the value of **e** in **$a0**
  - Preserves **$sp** and the contents of the stack
- We define a code generation function **cgen(e)** whose result is the code generated for **e**

Code Generation

- **cgen(e)** is going to work by cases.
- The code to evaluate a constant simply copies it into the accumulator:
  - `cgen(i) = li $a0 i`

- *This preserves the stack, as required*

- cgen(e_1 + e_2) =

```
cgen(e1)
sw $a0 0($sp)
addiu $sp $sp -4
cgen(e2)
lw $t1 4($sp)
add $a0 $t1 $a0
addiu $sp $sp 4
```

- The code for + is a template with "holes" for code for evaluating e_1 and e_2
- Stack machine code generation is recursive
- Code generation for expressions can be done as a recursive-descent of the AST

Code Generation

- MIPS instruction: `sub` reg_1 reg_2 reg_3
  - Implements $reg_1 \leftarrow reg_2 - reg_3$

- cgen(e_1 - e_2) =

```
cgen(e1)
sw $a0 0($sp)
addiu $sp $sp - 4
cgen(e2)
lw $t1 4($sp)
sub $a0 $t1 $a0
addiu $sp $sp 4
```

Code Generation

- MIPS instruction: `beq` reg_1 reg_2 `label`
  - Branch to label if $reg_1 = reg_2$
- MIPS instruction: `b label`
  - Unconditional branch to label

- cgen(if e_1 = e_2 then e_3 else e_4) =

```
cgen(e1)
sw $a0 0($sp)
addiu $sp $sp - 4
cgen(e2)
lw $t1 4($sp)
addiu $sp $sp 4
beq $a0 $t1 True_branch

False_branch:
    cgen(e4)
    b end_if
True_branch:
    cgen(e3)
End_if:
```

Code Generation – Function Calls

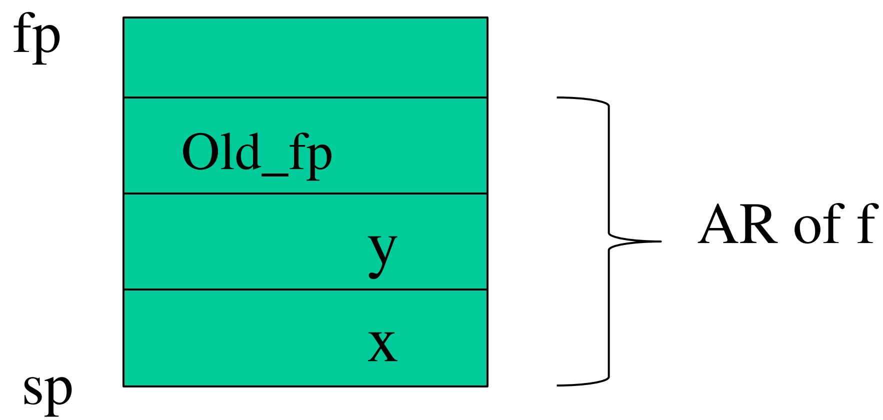
- Code for function calls and function definitions depends on the layout of the AR

- A very simple AR suffices for this language:
    - The result is always in the accumulator
    - No need to store the result in the AR
    - The activation record holds actual parameters
        - For $f(x_1,\ldots,x_n)$ push $x_n,\ldots,x_1$ on the stack
        - These are the only variables in this language
    - The stack discipline guarantees that on function exit \$sp is the same as it was on function entry

Code Generation – Function Calls

- A pointer to the current activation is useful
    - This pointer lives in register $fp (frame pointer)
- So, for this language, an AR with the caller's frame pointer, the actual parameters, and the return address suffices

- Consider a call to f(x,y), the AR is:



Code Generation – Function Calls

- The calling sequence is the instructions (of both caller and callee) to set up a function invocation

- New MIPS instruction: jal label
  - Jump to label, save address of next instruction in $ra
  - To be used in Caller

- New MIPS instruction: jr reg
  - Jump to address in register reg
  - To be used in Callee

Code Generation – Function Calls

Code in Caller

```
cgen(f(e1,…,en)) =
     sw $fp 0($sp)
     addiu $sp $sp - 4
     cgen(en)
     sw $a0 0($sp)
     addiu $sp $sp - 4
     …
     cgen(e1)
     sw $a0 0($sp)
     addiu $sp $sp - 4
     jal f_entry
```

- The caller saves its value of the frame pointer
- Then it saves the actual parameters in reverse order
- Finally the caller saves the return address in register $ra
- The AR so far is 4*n+4 bytes long

Code Generation – Function Calls

Code in Callee

```
cgen(def f(x1,…,xn)= e) =
F_entry:
     move $fp $sp
     sw $ra 0($sp)
     addiu $sp $sp - 4
     cgen(e)
     lw $ra 4($sp)
     addiu $sp $sp z
     lw $fp 0($sp)
     jr $ra
```

- The frame pointer points to the top, not bottom of the frame
- The callee pops the return address, the actual arguments and the saved value of the frame pointer
- $z = 4*n + 8$

Code Generation – Function Calls

- The "variables" of a function are just its parameters
  - They are all in the AR
  - Pushed by the caller

- Problem: Because the stack grows when intermediate results are saved, the variables are not at a fixed offset from $**sp**
- Solution: use a frame pointer
  - Always points to the return address on the stack

- Let x_i be the i^{th} ($i = 1,\ldots,n$) actual parameter of the function for which code is being generated
  - `cgen(x`$_i$`) = lw $a0 z($fp)` ($z = 4*i$)

Code Generation

- In production compilers:
  - Emphasis is on keeping values in registers
    - Especially the current stack frame
  - Intermediate results are laid out in the AR, not pushed and popped from the stack
    - The code generator must assign a location in the AR for each temporary

Code Generation – Handling Temporaries

- Let NT(e) = Number of temporaries needed to evaluate e
- NT(e_1 + e_2)
    - Needs at least as many temporaries as NT(e_1)
    - Needs at least as many temporaries as NT(e_2) + 1
- Space used for temporaries in e_1 can be reused for temporaries in e_2

- NT(e_1 + e_2) = max(NT(e_1), 1 + NT(e_2))
- NT(if e_1 = e_2 then e_3 else e_4) = max(NT(e_1),1 + NT(e_2), NT(e_3), NT(e_4))
- NT(id(e_1,…,e_n)) = max(NT(e_1),…,NT(e_n))
- NT(int / id) = 0

Code Generation

- def fib(x) = if x = 1 then 0 else
  if x = 2 then 1 else
  fib(x - 1) + fib(x – 2)

2 Temporary variables required

- For a function definition $f(x_1, \ldots, x_n) = e$ the AR has 2 + n + NT(e) elements
- Return address
- Frame pointer
- n arguments
- NT(e) locations for intermediate results

Old_fp
X_n
…
X_1
Return Address
Temp NT(e)
…
Temp 1

Code Generation

- Code generation must know how many temporaries are in use at each point
- Add a new argument to code generation
  - The position of the next available temporary
- The temporary area is used like a small, fixed-size stack

```
cgen(e1 + e2) =
     cgen(e1)
     sw $a0 0($sp)
     addiu $sp $sp - 4
     cgen(e2)
     lw $t1 4($sp)
     add $a0 $t1 $a0
     addiu $sp $sp 4
```

$\Rightarrow$

```
cgen(e1 + e2, nt) =
     cgen(e1, nt)
     sw $a0 nt($fp)
     cgen(e2, nt + 4)
     lw $t1 nt($fp)
     add $a0 $t1 $a0
```