11 *Run-Time Storage Organization*

The evolution of programming language design has led to the creation of increasingly sophisticated methods of run-time storage organization. Originally, all data was global, with a lifetime than spanned the entire program. Correspondingly, all storage allocation was *static*. A data object or instruction sequence was placed at a fixed address for the entire execution of a program.

Algol 60 and succeeding languages introduced local variables accessible only during the execution of a subprogram. This feature led to *stack allocation*. When a procedure was called, space for its local variables (its frame) was pushed on a run-time stack. Upon return, the space was popped. Only procedures actually executing were allocated space, and recursive procedures, which require multiple frames, were handled cleanly and naturally.

Lisp and later languages, including C, C++ and Java, popularized dynamically allocated data that could be created or freed at any time during execution.

Dynamic data led to *heap allocation*, which allows space to be allocated and freed at any time and in any order during program execution. With dynamic allocation, the number and size of data objects need not be fixed in advance. Rather, each program execution can "customize" its memory allocation.

All memory allocation techniques utilize the notion of a *data area*. A data area is a block of storage known by the compiler to have uniform storage allocation requirements. That is, all objects in a data area share the same data allocation policy. The global variables of a program can comprise a data area. Space for all variables is allocated when execution of a program begins, and variables remain allocated until execution terminates. Similarly, a block of data allocated by a call to `new` or `malloc` forms a single data area.

We'll begin our study of memory allocation with static allocation in Section 11.1. In Section 11.2 we'll investigate stack-based memory allocation. Finally, in Section 11.3, we'll consider heap storage.

11.1 Static Allocation

In the earliest programming languages, including all assembly languages, as well as Cobol and Fortran, all storage allocation was static. Space for data objects was allocated in a fixed location for the lifetime of a program. Use of static allocation is feasible only when the number and size of all objects to be allocated is known at compile-time. This allocation approach, of course, makes storage allocation almost trivial, but it can also be quite wasteful of space. As a result, programmers must sometimes *overlay* variables. Thus, in Fortran, the `equivalence` statement is commonly used to reduce storage needs by forcing two variables to share the same memory locations. (The C/C++ `union` can do this too.) Overlaying can lead to subtle programming errors, because assignment to one variable implicitly changes the value of another. Overlaying also reduces program readability.

In more modern languages, static allocation is used both for global variables that are fixed in size and accessible throughout program execution and for program literals (that is, constants) that need to be fixed throughout execution. Static allocation is used for `static` and `extern` variables in C/C++ and for `static` fields in Java classes. Static allocation is also routinely used for program code, since fixed run-time addresses are used in branch and call instructions. Also, since flow of control within a program is very hard to predict, it difficult to know which instructions will be needed next. Accordingly, if code is statically allocated, any execution order can be accommodated. Java allows classes to be dynamically loaded or compiled; but once program code is made executable, it is static.

Conceptually, we can bind static objects to absolute addresses. Thus if we generate an assembly language translation of a program, a global variable or program statement can be given a symbolic label that denotes a fixed memory address. It is often preferable to address a static data object as a pair (DataArea, Offset). Offset is fixed at compile-time, but the address of DataArea can be deferred to link- or run-time. In Fortran, for example, DataArea can be the start of one of many `common` blocks. In C, DataArea can be the start of a block of storage for the variables local to a module (a ".c" file). In Java, DataArea can be the start of a block of storage for the static fields of a class. Typically these addresses are bound when the program is linked. Address binding must be deferred until link-time or run-time because subroutines and classes may be compiled independently, making it impossible for the compiler to know about all the data areas in a program.

Alternatively, the address of DataArea can be loaded into a register (the **global pointer**), which allows a static data item to be addressed as (Register, Offset). This addressing form is available on almost every machine. The advantage of addressing a piece of static data with a global pointer is that we can load or store a global value in one instruction. Since global addresses occupy 32 (or 64) bits, they normally "don't fit" in a single instruction. Instead, lacking a global pointer, global addresses must be formed in several steps, first loading the high-order bits, then masking in the remain low-order bits.

11.2 Stack Allocation

Almost all modern programming languages allow recursive procedures, functions or methods, which require dynamic allocation. Each recursive call requires the allocation of a new copy of a routine's local variables; thus the number of data objects required during program execution is not known at compile-time. To implement recursion, all the data space required for a routine (a procedure, function or method) is treated as a data area that, because of the special way it is handled, is called a *frame* or *activation record*.

Local data, held within a frame, is accessible only while a subprogram is active. In mainstream languages like C, C++ and Java, subprograms must return in a stack-like manner; the most recently called subprogram will be the first to return. This means a frame may be pushed onto a *run-time stack* when a routine is called (activated). When the routine returns, the frame is popped from the stack, freeing the routine's local data. To see how stack allocation works, consider the C subprogram shown in Figure 11.1.

```
p(int a) {
     double b;
     double c[10];
     b = c[a] * 2.51;
}
```

Figure 11.1 A Simple Subprogram

Procedure p requires space for the parameter a as well as the local variables b and c. It also needs space for control information, such as the return address (a procedure may be called from many different places). As the procedure is compiled, the space requirements of the procedure are recorded (in the procedure's symbol table). In particular, the *offset* of each data item relative to the beginning of the frame is stored in the symbol table. The total amount of space needed, and thus the size of the frame, is also recorded. The memory requirements for individual variables (and hence an entire frame) is machine-dependent. Different architectures may assume different sizes for primitive values like integers or addresses.

In our example, assume p's control information requires 8 bytes (this size is usually the same for all routines). Assume parameter a requires 4 bytes, local variable b requires 8 bytes, and local array c requires 80 bytes. Because many machines require that word and doubleword data be *aligned*, it is common practice to pad a frame (if necessary) so that its size is a multiple of 4 or 8 bytes. This guarantees a useful invariant—at all times the top of the stack is properly aligned. Figure 11.2 shows `p`'s frame.

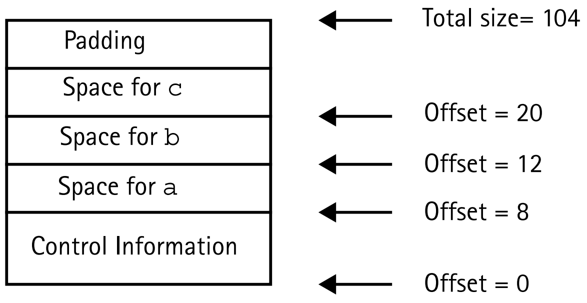

Figure 11.2 Frame for Procedure `P`

Within `p`, each local data object is addressed by its offset relative to the start of the frame. This offset is a fixed constant, determined at compile-time. Because we normally store the start of the frame in a register, each piece of data can be addressed as a (Register, Offset) pair, which is a standard addressing mode in almost all computer architectures. For example, if register R points to the beginning of `p`'s frame, variable `b` can be addressed as (R,12), with 12 actually being added to the contents of R at run-time, as memory addresses are evaluated.

Normally, the literal `2.51` of procedure `p` is not stored in `p`'s frame because the values of local data that are stored in a frame disappear with it at the end of a call. If `2.51` were stored in the frame, its value would have to be initialized before each call. It is easier and more efficient to allocate literals in a static area, often called a **literal pool** or **constant pool**. Java uses a constant pool to store literals, type, method and interface information as well as class and field names.

11.2.1 Accessing Frames at Run-Time

At any time during execution there can be many frames on the stack. This is because when a procedure `A` calls a procedure `B`, a frame for `B`'s local variables is pushed on the stack, covering `A`'s frame. `A`'s frame can't be popped off because `A` will resume execution after `B` returns. In the case of recursive routines there can be hundreds or even thousands of frames on the stack. All frames but the topmost represent suspended subroutines, waiting for a call to return.

The topmost frame is *active*, and it is important to be able to access it directly. Since the frame is at the top of the stack, the stack top register could be used to access it. The run-time stack may also be used to hold data other than frames, like temporary or return values too large to fit within a register (arrays, structs, strings, etc.)

It is therefore unwise to require that the currently active frame always be at *exactly* the top of the stack. Instead a distinct register, often called the *frame pointer*, is used to access the current frame. This allows local variables to be accessed directly as offset + frame pointer, using the indexed addressing mode found on all modern machines.

As an example, consider the following recursive function that computes factorials.

```
int fact(int n) {
    if (n > 1)
          return n * fact(n-1);
    else  return 1;
}
```

The run-time stack corresponding to the call `fact(3)` is shown in Figure 11.3 at the point where the call of `fact(1)` is about to return. In our example we show a slot for the function's return value at the very beginning of the frame. This means that upon return, the return value is conveniently placed on the stack, just beyond the end of the caller's frame. As an optimization, many compilers try to return scalar values in specially designated registers. This helps to eliminate unnecessary loads and stores. For function values too large to fit in a register (e.g., a `struct`), the stack is the natural choice.

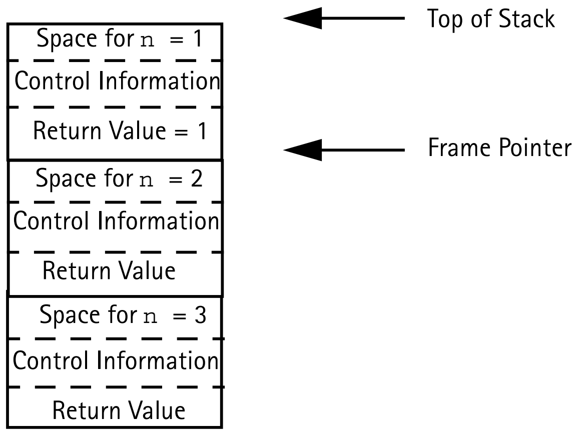

Figure 11.3 Run-time Stack for a Call of `fact(3)`

When a subroutine returns, its frame must be popped from the stack and the frame pointer must be reset to point to the caller's frame. In simple cases this can be done by adjusting the frame pointer by the size of the current frame. Because the stack may contain more than just frames (e.g., function return values or registers saved across calls), it is common practice to save the caller's frame pointer as part of the callee's control information. Thus each frame points to the preceding frame on the stack. This pointer is often called a *dynamic link* because it links a frame to its dynamic (run-time) predecessor. The run-time stack corresponding to a call of `fact(3)`, with dynamic links included, is shown in Figure 11.4.

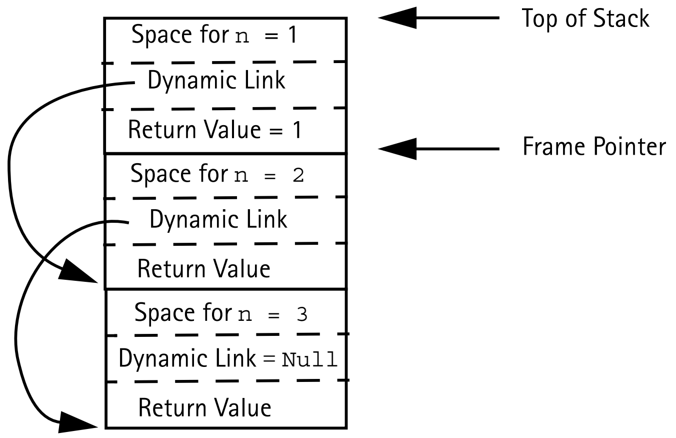

Figure 11.4 Run-time Stack for a Call of `fact(3)` with Dynamic Links

11.2.2 Handling Classes and Objects

C, C++ and Java do not allow procedures or methods to nest. That is, a procedure may not be declared within another procedure. This simplifies run-time data access—all variables are either global or local to the currently executing procedure.

Global variables are statically allocated. Local variables are part of a single frame, accessed through the frame pointer.

Languages often need to support simultaneous access to variables in multiple scopes. Java and C++, for example, allows classes to have member functions that have direct access to all instance variables. Consider the following Java class.

```
class k {
      int a;
      int sum(){
         int b;
         return a+b;
}     }
```

Each object that is an instance of class `k` contains a member function `sum`. Only one translation of `sum` is created; it is shared by all instances of `k`. When `sum` executes it requires two pointers to access local and object-level data. Local data, as usual, resides in a frame on the run-time stack. Data values for a particular instance of `k` are accessed through an object pointer (called the `this` pointer in Java and C++). When `obj.sum()` is called, it is given an extra implicit parameter that a pointer to `obj`. This is illustrated in Figure 11.5. When `a+b` is computed, `b`, a local variable, is accessed directly through the frame pointer. `a`, a member of object `obj`, is accessed indirectly through the object pointer that is stored in the frame (as all parameters to a method are).

C++ and Java also allow **inheritance** via **subclassing**. That is, a new class can extend an existing class, adding new fields and adding or redefining methods. A

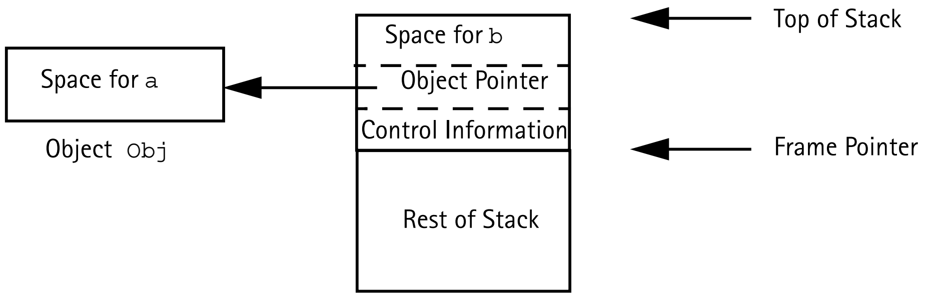

Figure 11.5 Accessing Local and Member Data in Java

subclass `D`, of class `C`, maybe be used in contexts expecting an object of class `C` (e.g., in method calls). This is supported rather easily—objects of class `D` always contain a class `C` object within them. That is, if `C` has a field `F` within it, so does `D`. The fields `D` declares are merely appended at the end of the allocations for `C`. As a result, access to fields of `C` within a class `D` object works perfectly. In Java, class `Object` is often used as a placeholder when an object of unknown type is expected. This works because all objects are subclasses of `Object`.

Of course, the converse cannot be allowed. A class `C` object may not be used where a class `D` object is expected, since `D`'s fields aren't present in `C`.

11.2.3 Handling Multiple Scopes

Many languages, including Ada, Pascal, Algol and Modula-3 allow procedure declarations to nest. The latest releases of Java allow classes to nest (see Exercise 6). Procedure nesting can be very useful, allowing a subroutine to directly access another routine's locals and parameters. However, run-time data structures are complicated because multiple frames, corresponding to nested procedure declarations, may need to be accessed. To see the problem, assume that routines *can* nest in Java or C, and consider the following code fragment

```
int p(int a){
     int q(int b){
        if (b < 0)
              q(-b);
        else  return a+b;
     }
     return q(-10);
}
```

When `q` executes, it can access not only its own frame, but also that of `p`, in which it is nested. If the depth of nesting is unlimited, so is the number of frames that must be made accessible. In practice, the level of procedure nesting actually seen is modest—usually no greater than two or three.

Two approaches are commonly used to support access to multiple frames. One approach generalizes the idea of dynamic links introduced earlier. Along with a dynamic link, we'll also include a *static link* in the frame's control information area. The static link will point to the frame of the procedure that statically encloses the current procedure. If a procedure is not nested within any other procedure, its static link is `null`. This approach is illustrated in Figure 11.6.

As usual, dynamic links always point to the next frame down in the stack. Static links always point down, but they may skip past many frames. They always point to the most recent frame of the routine that statically encloses the current

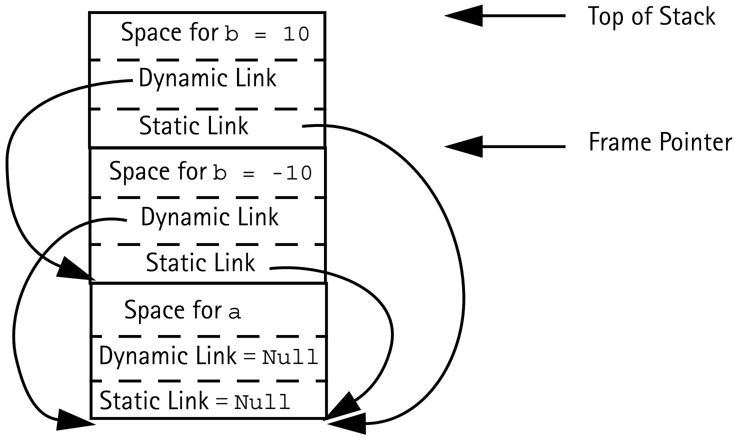

Figure 11.6 An Example of Static Links

routine. Thus in our example, the static links of both of q's frames point to p, since it is p that encloses q's definition. In evaluating the expression a+b that q returns, b, being local to q, is accessed directly through the frame pointer. Variable a is local to p, but also visible to q because q nests within p. a is accessed by extracting q's static link, then using that address (plus the appropriate offset) to access a. (See Exercise 18.)

An alternative to using static links to access frames of enclosing routines is the use of a *display.* A display generalizes our use of a frame pointer. Rather than maintaining a single register, we maintain a *set of registers* which comprise the display. If procedure definitions nest *n* deep (this can be easily determined by examining a program's AST), we will need *n+1* display registers. Each procedure definition is tagged with a nesting level. Procedures not nested within any other routine are at level 0. Procedures nested within only one routine are at level 1, etc. Frames for routines at level 0 are always accessed using display register D0. Those at level 1 are always accessed using register D1, etc. Thus whenever a procedure `r` is executing, we have direct access to `r`'s frame plus the frames of all routines that enclose `r`. Each of these routines must be at a different nesting level, and hence will use a different display register. Consider Figure 11.7, which illustrates our earlier example, now using display registers rather than static links.

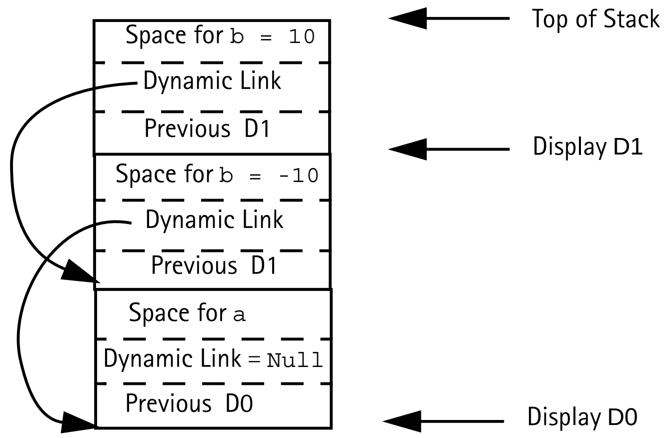

Figure 11.7 An Example of Display Registers

Since `q` is at nesting level 1, its frame is pointed to by D1. All of `q`'s local variables, including `b`, are at a fixed offset relative to D1. Similarly, since `p` is at nesting level 0, its frame and local variables are accessed via D0. Note that each frame's control information area contains a slot for the previous value of the frame's display register. This value is saved when a call begins and restored when the call ends. The dynamic link is still needed, because the previous display values doesn't always point to the caller's frame.

Not all compiler writers agree on whether static links or displays are better to use. Displays allow direct access to all frames, and thus make access to all visible variables very efficient. However, if nesting is deep, several valuable registers may need to be reserved. Static links are very flexible, allowing unlimited nesting of procedures. However, access to non-local procedure variables can be slowed by the need to extract and follow static links.

Fortunately, the code generated using the two techniques can be improved. Static links are just address-valued expressions computed to access variables (much like address calculations involving pointer variables). A careful compiler can notice that an expression is being needlessly recomputed, and reuse a previous computation, often directly from a register. Similarly, a display can be allocated statically in memory. If a particular display value is used frequently, a register allocator will place the display value in a register to avoid repeated loads and stores (just as it would for any other heavily used variable).

11.2.4 Block-Level Allocation

Java, C and C++, as well as most other programming languages, allow declaration of local variables within blocks as well as within procedures. Often a block will contain only one or two variables, along with statements that use them. Do we allocate an entire frame for each such block?

We could, by considering a block with local variables to be an in-line procedure without parameters, to be allocated its own frame. This would necessitate a display or static links, even in Java or C, because blocks can nest. Further, execution of a block would become more costly, since frames need to be pushed and popped, display registers or static links updated, and so forth.

To avoid this overhead, it is possible to use frames only for true procedures, even if blocks within a procedure have local declarations. This technique is called *procedure-level frame allocation*, as contrasted with *block-level frame allocation*, which allocates a frame for each block that has local declarations.

The central idea of procedure-level frame allocation is that the relative location of variables in individual blocks within a procedure can be computed and fixed at compile-time. This works because blocks are entered and exited in a strictly textual order. Consider, the following procedure

```
void p(int a) {
  int b;
  if (a > 0)
        {float c, d;
           // Body of block 1 }
  else  {int e[10];
           // Body of block 2 }
}
```

Parameter `a` and local variable `b` are visible throughout the procedure. However the `then` and `else` parts of the if statement are mutually exclusive. Thus variables in block 1 and block 2 can *overlay* each other. That is, `c` and `d` are allocated just beyond `b` as is the array `e`. Because variables in both blocks can't ever be accessed at the same time, this overlaying is safe. The layout of the frame is illustrated in Figure 11.8.

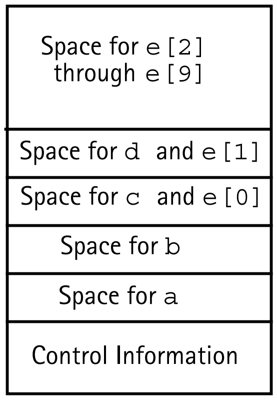

Figure 11.8 An Example of a Procedure-Level Frame

Offsets for variables within a block are assigned just after the last variable in the enclosing scope within the procedure. Thus both c and e[] are placed after b because both block 1 and block 2 are enclosed by the block comprising p's body. As blocks are compiled a "high water mark" is maintained that represents the maximum offset used by any local variable. This high water mark determines the size of the overall frame. Thus a[9] occupies the maximum offset within the frame, so its location determines the size of p's frame.

The process of assigning local variables procedure-level offsets is sometimes done using **scope flattening**. That is, local declarations are mapped to equivalent procedure-level declarations. This process is particularly effective if procedure-level register allocation is part of the compilation process (see Section 15.3).

11.2.5 More About Frames

ClosuresIn C it is possible to create a pointer to a function. Since a function's frame is created only when it is called, a function pointer is implemented as the function's entry point address. In C++, pointers to member functions of a class are allowed. When the pointer is used, a particular instance of the class must be provided by the user program. That is, two pointers are needed, one to the function itself and a second pointer to the class instance in which it resides. This second pointer allows the member function to access correctly local data belonging to the class.

Other languages, particularly functional languages like Lisp, Scheme, and ML, are much more general in their treatment of functions. Functions are *first-class objects*. They can be stored in variables and data structures, created during execution, and returned as function results.

Run-time creation and manipulation of functions can be extremely useful. For example, it is sometimes the case that computation of `f(x)` takes a significant amount of time. Once `f(x)` is known, it is a common optimization, called **memoizing**, to table the pair `(x,f(x))` so that subsequent calls to `f` with argument `x` can use the known value of `f(x)` rather than recompute it. In ML it is possible to write a function `memo` that takes a function `f` and an argument `arg`. `memo` computes `f(arg)` *and* also returns a "smarter" version of `f` that has the value of `f(arg)` "built into" it. This smarter version of `f` can be used instead of `f` in all subsequent computations.

```
fun memo(fct,parm)= let val ans = fct(parm) in
 (ans, fn x=> if x=parm then ans else fct(x))end;
```

When the version of `fct` returned by `memo` is called, it will access to the values of `parm`, `fct` and `ans`, which are used in its definition. After `memo` returns, its frame must be preserved since that frame contains `parm`, `fct` and `ans` within it.

In general when a function is created or manipulated, we must maintain a pair of pointers. One is to the machine instructions that implement the function, and the other is to the frame (or frames) that represent the function's execution environment. This pair of pointers is called a **closure**. Note also that when functions are first-class objects, a frame corresponding to a call may be accessed *after* the call terminates. This means frames can't always be allocated on the run-time stack. Instead, they are allocated in the heap and garbage-collected, just like user-created data. This appears to be inefficient, but Appel [1987] has shown that in some circumstances heap allocation of frames can be faster than stack allocation.

Cactus StacksMany programming languages allow the concurrent execution of more than one computation in the same program. Units of concurrent execution are sometimes called tasks, or processes or threads. In some cases a new system-level process is created (as in the case of `fork` in C). Because a great deal of operating system overhead is involved, such processes are called *heavy-weight processes*. A less expensive alternative is to execute several threads of control in a single system-level process. Because much less "state" is involved, computations that execute concurrently in a single system process are called *light-weight processes*.

A good example of light-weight processes are *threads* in Java. As illustrated below, a Java program may initiate several calls to member functions that will execute simultaneously.

```
public static void main (String args[]) {
        new AudioThread("Audio").start();
        new VideoThread("Video").start();
}
```

Here two instances of `Thread` subclasses are started, and each executes concurrently with the other. One thread might implement the audio portion of an application, while the other implements the video portion.

Since each thread can initiate its own sequence of calls (and possibly start more threads), all the resulting frames can't be pushed on a single run-time stack (the exact order in which threads execute is unpredictable). Instead, each thread gets its own stack segment in which frames it creates may be pushed. This stack structure is sometimes called a *cactus stack*, since it is reminiscent of the saguaro cactus, which sends out arms from the main trunk and from other arms. It is important that the thread handler be designed so that segments are properly deallocated when their thread is terminated. Since Java guarantees that all temporaries and locals are contained with a method's frame, stack management is limited to proper allocation and deallocation of frames.

A Detailed Frame LayoutThe layout of a frame is usually standardized for a particular architecture. This is necessary to support calls to subroutines translated by different compilers. Since languages and compilers vary in the features they support, the frame layout chosen as the standard must be very general and inclusive. As an example, consider Figure 11.9, which illustrates the frame layout used by the MIPS architecture.

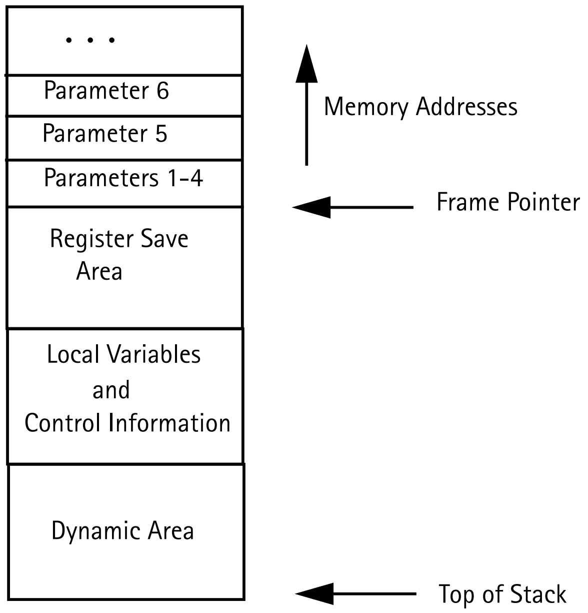

Figure 11.9 Layout for MIPS R3000

By convention, the first four parameters, if they are scalar, are passed in registers. Additional parameters, as well as non-scalar by-value parameters, are passed through the stack. The slots for parameters 1-4 can be used to save parameter registers when a call is made from within a procedure. The register save area is used at two different times. Registers are commonly partitioned in **caller-saved** registers (which a caller is responsible for) and **callee-saved** registers (which a subprogram is responsible for). When execution of a subroutine begins, callee-saved registers used by the subroutine are saved in the register save area. When a call is made from within the subroutine, caller-saved registers that are in use are saved in the register save area. At different call sites different registers may be in use. The register save area must be big enough to handle all calls within a particular subroutine. Often a fixed size register save area, big enough to accommodate all caller-saved and callee-saved registers, is used. This may waste a bit of space, but only registers actually in use are saved.

The local variables and control information area contains space for all local variables. It also contains space for the return address register, and the value of the caller's frame pointer. The value of a static link or display register may be saved here if they are needed. The stack top may be reset, upon return, by adding the size of the parameter area to the frame pointer. (Note that on the MIPS, as well as on many other computers, the stack grows downward, from high to low addresses).

The details of subroutine calls are explored more thoroughly in Section 13.2 (at the bytecode level) and Section 15.1 (at the machine code level).

Because the Java Virtual Machine is designed to run on a wide variety of architectures, the exact details of its run-time frame layout are unspecified. A particular implementation (such as the JVM running on a MIPS processor), chooses a particular layout, similar to that shown in Figure 11.9.

Some languages allow the size of a frame to be expanded during execution. In C, for example, `alloca` allocates space on demand on the stack. Space is pushed beyond the end of the frame. Upon return, this space is automatically freed when the frame is popped.

Some languages allow the creation of *dynamic arrays* whose bounds are set at run-time when a frame is pushed (e.g., `int data[max(a,b)]`). At the start of a subprogram's execution, array bounds are evaluated and necessary space is pushed in the dynamic area of the frame.

C and C++ allow subroutines like `printf` and `scanf` to have a variable number of arguments. The MIPS frame design supports such routines, since parameter values are placed, in order, just above the frame pointer.

Non-scalar return values can be handled by treating the return value as the "0-th parameter." As an optimization, calls to functions that return a non-scalar result sometimes pass an address as the 0-th parameter. This represents a place where the return value can be stored prior to return. Otherwise, the return value is left on the stack by the function.

11.3 Heap Management

The most flexible storage allocation mechanism is *heap allocation*. Any number of data objects can be allocated and freed at any time and in any order. A storage pool, usually called a *heap*, is used. Heap allocation is enormously popular—it is difficult to imagine a non-trivial Java or C program that does not use `new` or `malloc`.

Heap allocation and deallocation is far more complicated than is the case for static or stack allocation. Complex mechanisms may be needed to satisfy a request for space. Indeed, in some cases, all of the heap (many megabytes) may need to be examined. It takes great care to make heap management fast and efficient.

11.3.1 Allocation Mechanisms

A request for heap space may be *explicit* or *implicit*. An explicit request involves a call to a routine like `new` or `malloc`, with a request for a specific number of bytes. An explicit pointer or reference to the newly allocated space is returned (or a `null` pointer if the request could not be honored).

Some languages allow the creation of data objects of unknown size. Assume that in C++, as in Java, the + operator is overloaded to represent string catenation. That is, the expression `Str1 + Str2` creates a new string representing the catenation of strings `Str1` and `Str2`. There is no compile-time bound on the sizes of `Str1` and `Str2`, so heap space must be allocated to hold the newly created string.

Whether allocation is explicit or implicit, a *heap allocator* is needed. This routine takes a size parameter and examines unused heap space to find free space that satisfies the request. A *heap block* is returned. This block will be big enough to satisfy the space request, but it may well be bigger. Allocated heap blocks are almost always single- or double-word aligned to avoid alignment problems in heap-allocated arrays or class instances. Heaps blocks contain a header field (usually a word) that contains the size of the block as well as auxiliary bookkeeping information. (The size information is necessary to properly "recycle" the block if it is later deallocated.) A minimum heap block size (commonly 16 bytes) is usually imposed to simplify bookkeeping and guarantee alignment.

The complexity of heap allocation depends in large measure on how *deallocation* is done. Initially, the heap is one large block of unallocated memory. Memory requests can be satisfied by simply modifying an “end of heap” pointer, very much as a stack is pushed by modifying a stack pointer. Things get more involved when previously allocated heap objects are deallocated and reused. Some deallocation techniques *compact* the heap, moving all “in use” objects to one end of the heap. This means unused heap space is always contiguous, making allocation (via a heap pointer) almost trivial.

Some heap deallocation algorithms have the useful property that their speed depends not on the total number of heap objects allocated, but rather only on those objects still in use. If most heap objects “die” soon after their allocation (and this does seem to often be the case), deallocation of these objects is essentially free.

Unfortunately, many deallocation techniques *do not* perform compaction. Deallocated objects must be stored for future reuse. The most common approach is to create a *free space list*. A free space list is a linked (or doubly-linked) list that contains all the heap blocks known not to be in use. Initially it contains one immense block representing the entire heap. As heap blocks are allocated, this block shrinks. When heap blocks are returned, they are appended to the free space list.

The most common way of maintaining the free space list is to append blocks to the head of the list as they are deallocated. This simplifies deallocation a bit, but makes *coalescing* of free space difficult.

It often happens that two blocks, physically adjacent in the heap, are eventually deallocated. If we can recognize that the two blocks are now both free and adjacent, they can be coalesced into one larger free block. One large block is preferable to two smaller blocks, since the combined block can satisfy requests too large for either of the individual blocks.

The *boundary tags* approach (Knuth 1973) allows us to identify and coalesce adjacent free heap blocks. Each heap block, whether allocated or on the free space list, contains a tag word on both of its ends. This tag word contains a flag indicating "free" or "in use" and the size of the block. When a block is freed, the boundary tags of its neighbors are checked. If either or both neighbors are marked as free, they are unlinked from the free space list and coalesced with the current free block.

A free space list may also be kept in *address order*; that is, sorted in order of increasing heap addresses. Boundary tags are no longer needed to identify adjacent free blocks, though maintenance of the list in sorted order is now more expensive.

When a request for *n* bytes of heap space is received, the heap allocator must search the free space list for a block of sufficient size. But how much of the free space list is to be searched? (It may contain many thousands of blocks.) What if no free block exactly matches the current request? There are many approaches that might be used. We'll consider briefly a few of the most widely used techniques.

Best FitThe free space list is searched, perhaps exhaustively, for the free block that matches most closely the requested size. This minimizes wasted heap space, though it may create tiny fragments too small to be used very often. If the free space list is very long, a best fit search may be quite slow. Segregated free space lists (see below) may be preferable.

First FitThe first free heap block of sufficient size is used. Unused space within the block is split off and linked as a smaller free space block. This approach is fast, but may “clutter” the beginning of the free space list with a number of blocks too small to satisfy most requests.

Next FitThis is a variant of first fit in which succeeding searches of the free space list begin at the position where the last search ended, rather than at the head of the list. The idea is to “cycle through” the entire free space list rather than always revisiting free blocks at the head of the list. This approach reduces *fragmentation* (in which blocks are split into small, difficult to use pieces). However, it also reduces *locality* (how densely packed active heap objects are). If we allocate heaps objects that are widely distributed throughout the heap, we may increase cache misses and page faults, significantly impacting performance.

Segregated Free Space ListsThere is no reason why we must have only *one* free space list. An alternative is to have several, indexed by the size of the free blocks they contain. Experiments have shown that programs frequently request only a few “magic sizes.” If we divide the heap into segments, each holding only one size, we can maintain individual free space lists for each segment. Because heap object size is fixed, no headers are needed.

A variant of this approach is to maintain lists of objects of special "strategic" sizes (16, 32, 64, 128, etc.) When a request for size *s* is received, a block of the smallest size ≤ *s* is selected (with excess size unused within the allocated block).

Another variant is to maintain a number of free space lists, each containing a range of sizes. When a request for size *s* is received, the free space list covering *s*'s range is searched using a best fit strategy.

Fixed-Size SubheapsRather than linking free objects onto lists according to their size, we can divide the heap into a number of *subheaps*, each allocating objects of a single fixed size. We can then use a *bitmap* to track an object's allocation status. That is, each object is mapped to a single bit in a large array. A 1 indicates the object is in use; a 0 indicated it is free. We need no explicit headers or free-space lists. Moreover, since all objects are of the same size, *any* object whose status bit is 0 may be allocated. We do have the problem though that subheaps may be used unevenly, with one nearly exhausted while another is lightly-used.

11.3.2 Deallocation Mechanisms

Allocating heap space is fairly straightforward. Requests for space are satisfied by adjusting an end-of-heap pointer, or by searching a free space list. But how do we deallocate heap memory no longer in use? Sometimes we may never need to deallocate! If heaps objects are allocated infrequently or are very long-lived, deallocation is unnecessary. We simply fill heap space with "in use" objects.

Unfortunately, many—perhaps most—programs cannot simply ignore deallocation. Experience shows that many programs allocate huge numbers of short-lived heap objects. If we "pollute" the heap with large numbers of **dead** objects (that are no longer accessible), locality can be severely impacted, with active objects spread throughout a large address range. Long-lived or continuously running programs can also be plagued by **memory leaks**, in which dead heap objects slowly accumulate until a program's memory needs exceed system limits.

User-controlled DeallocationDeallocation can be manual or automatic. Manual deallocation involves explicit programmer-initiated calls to routines like `free(p)` or `delete(p)`. Pointer `p` identifies the heap object to be freed. The object's size is stored in its header. The object may be merged with adjacent unused heap objects (if boundary tags or an address-ordered free space list is used). It is then added to a free-space list for subsequent reallocation.

It is the programmer's responsibility to free unneeded heap space by executing deallocation commands. The heap manager merely keeps track of freed space and makes it available for later reuse. The really hard decision—when space should be freed—is shifted to the programmer, possibly leading to catastrophic *dangling pointer* errors. Consider the following C program fragment

```
q = p = malloc(1000);
free(p);
/* code containing a number of malloc's */
q[100] = 1234;
```

After `p` is freed, `q` is a dangling pointer. That is, `q` points to heap space that is no longer considered allocated. Calls to `malloc` may reassign the space pointed to by `q`. Assignment through `q` is illegal, but this error is almost never detected. Such an assignment may change data that is now part of another heap object, leading to very subtle errors. It may even change a header field or a free-space link, causing the heap allocator itself to fail.

11.3.3 Automatic Garbage Collection

The alternative to manual deallocation of heap space is automatic deallocation, commonly called *garbage collection*. Compiler-generated code and support subroutines track pointer usage. When a heap object is no longer pointed to (that is, when it is *garbage*), the object is automatically deallocated (collected) and made available for subsequent reuse.

Garbage collection techniques vary greatly in their speed, effectiveness and complexity. We shall consider briefly some of the most important approaches. For a more thorough discussion see [Wilson 96] or [JL 96].

Reference CountingOne of the oldest and simplest garbage collection techniques is *reference counting*. A field is added to the header of each heap object. This field, the object's *reference count*, records how many references (pointers) to the heap object exist. When an object's reference count reaches zero, it is garbage and may be added to the free-space list for future reuse.

The reference count field must be updated whenever a reference is created, copied, or destroyed. When a procedure returns, the reference counts for all objects pointed to by local variables must be decremented. Similarly, when a reference count reaches zero and an object is collected, all pointers in the collected object must also be followed and corresponding reference counts decremented.

As shown in Figure 11.10, reference counting has particular difficulty with *circular structures*. If pointer `P` is set to null, the object's reference count is reduced to 1. Now both objects have a non-zero count, but neither is accessible through any external pointer. That is, the two objects are garbage, but won't be recognized as such.

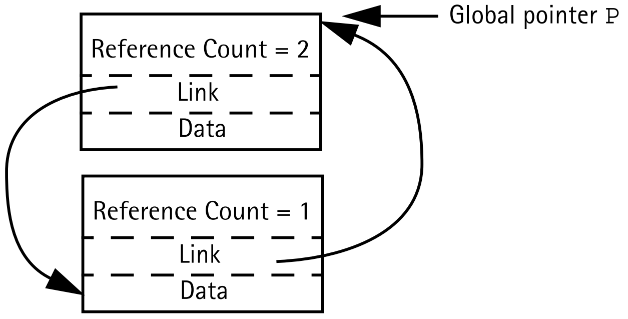

Figure 11.10 An Example of Cyclic Structures

If circular structures are rare, this deficiency won't be much of a problem. If they are common, then an auxiliary technique, like mark-sweep collection, will be needed to collect garbage that reference counting misses.

An important aspect of reference counting is that it is *incremental*. That is, whenever a pointer is manipulated, a small amount of work is done to support garbage collection. This is both an advantage and a disadvantage. It is an advantage in that the cost of garbage collection is smoothly distributed throughout a computation. A program doesn't need to stop when heap space grows short and do a lengthy collection. This can be crucial when fast real-time response is required. (We don't want the controls of an aircraft to suddenly "freeze" for a second or two while a program collects garbage!)

The incremental nature of reference counting can also be a disadvantage when garbage collection isn't really needed. If we have a complex data structure in which pointers are frequently updated, but in which few objects ever are discarded, reference counting always adjusts counts that rarely, if ever, go to zero.

How big should a reference count field be? Interestingly, experience has shown that it doesn't need to be particularly large. Often only a few bits suffice. The idea here is that if a count ever reaches the maximum representable count (perhaps 7 or 15 or 31), we "lock" the count at that value. Objects with a locked reference count won't ever be detected as garbage by their counts, but they can be collected using other techniques when circular structures are collected.

In summary, reference counting is a simple technique whose incremental nature is sometimes useful. Because of its inability to handle circular structures and its significant per-pointer operation cost, other garbage collection techniques, as described below, are often more attractive alternatives.

Mark-Sweep CollectionRather than incrementally collecting garbage as pointers are manipulated, we can take a batch approach. We do nothing until heap space is nearly exhausted. Then we execute a *marking phase* which aims to identify all live (non-garbage) heap objects.

Starting with global pointers and pointers in stack frames, we mark reachable heap objects (perhaps setting a bit in the object's header). Pointers in marked heap objects are also followed, until all live heap objects are marked.

After the marking phase, we know that any object not marked is garbage that may be freed. We then *sweep* through the heap, collecting all unmarked objects and returning them to the free space list for later reuse. During the sweep phase we also clear all marks from heap objects found to be still in use.

Mark-sweep garbage collection is illustrated in Figure 11.11. Objects 1 and 3 are marked because they are pointed to by global pointers. Object 5 is marked because it is pointed to by object 3, which is marked. Shaded objects are not marked and will be added to the free-space list.

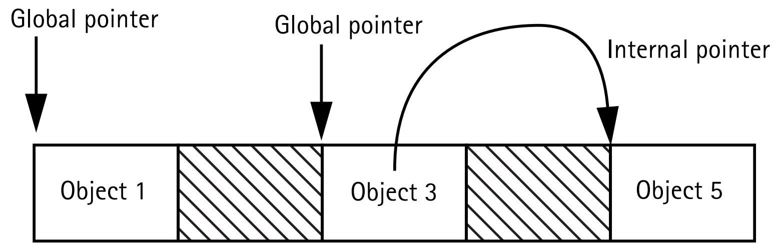

Figure 11.11 Mark-Sweep Garbage Collection

In any mark-sweep collector, it is vital that we mark *all* accessible heap objects. If we miss a pointer, we may fail to mark a live heap object and later incorrectly free it. Finding all pointers is not too difficult in languages like Lisp and Scheme that have very uniform data structures, but it is a bit tricky in languages like Java, C and C++, that have pointers mixed with other types within data structures, implicit pointers to temporaries, and so forth. Considerable information about data structures and frames must be available at run-time for this purpose. In cases where we can't be sure if a value is a pointer or not, we may need to do conservative garbage collection (see below).

Mark-sweep garbage collection also has the problem that *all* heap objects must be swept. This can be costly if most objects are dead. Other collection schemes, like copying collectors, examine *only* live objects.

After the sweep phase, live heap objects are distributed throughout the heap space. This can lead to poor locality. If live objects span many memory pages, paging overhead may be increased. Cache locality may be degraded too.

We can add a *compaction phase* to mark-sweep garbage collection. After live objects are identified, they are placed together at one end of the heap. This involves another tracing phase in which global, local and internal heap pointers are found and adjusted to reflect the object's new location. Pointers are adjusted by the total size of all garbage objects between the start of the heap and the current object. This is illustrated in Figure 11.12.

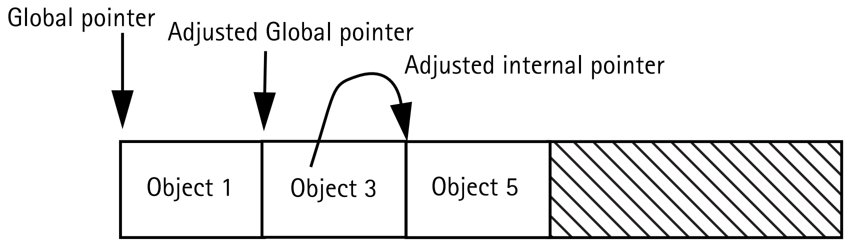

Figure 11.12 Mark-Sweep Garbage Collection with Compaction

Compaction is attractive because all garbage objects are merged together into one large block of free heap space. Fragments are no longer a problem. Moreover, heap allocation is greatly simplified. An "end of heap" pointer is maintained. Whenever a heap request is received, the end of heap pointer is adjusted, making heap allocation no more complex than stack allocation.

However, because pointers must be adjusted, compaction may not be suitable for languages like C and C++, in which it is difficult to unambiguously identify pointers.

Copying CollectorsCompaction provides many valuable benefits. Heap allocation is simple end efficient. There is no fragmentation problem, and because live objects are adjacent, paging and cache behavior is improved. An entire family of garbage collection techniques, called *copying collectors* have been designed to integrate copying with recognition of live heap objects. These copying collectors are very popular and are widely used, especially with functional languages like ML.

We'll describe a simple copying collector that uses *semispaces*. We start with the heap divided into two halves—the *from* and *to spaces*. Initially, we allocate heap requests from the from space, using a simple "end of heap" pointer. When the from space is exhausted, we stop and do garbage collection.

Actually, though we *don't* collect garbage. What we do is collect live heap objects—garbage is never touched. As was the case for mark-sweep collectors, we trace through global and local pointers, finding live objects. As each object is found, it is moved from its current position in the from space to the next available position in the to space. The pointer is updated to reflect the object's new location. A "forwarding pointer" is left in the object's old location in case there are multiple pointers to the same object (we want only one object with all original pointers properly updated to the new location).

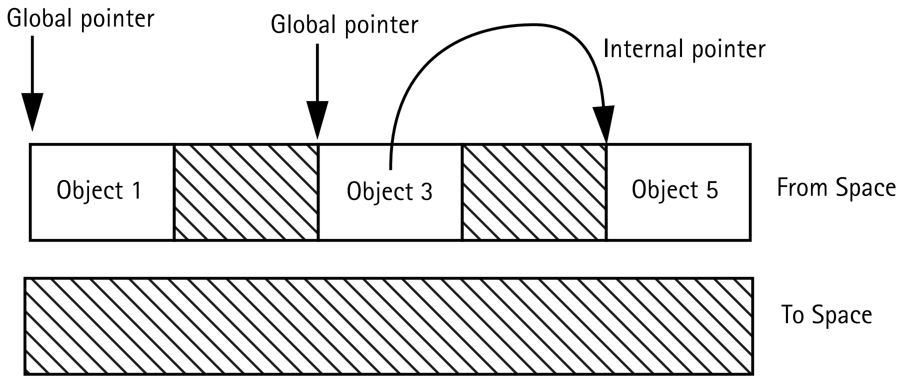

Figure 11.13 Copying Garbage Collection (a)

This is illustrated in Figure 11.13. The from space is completely filled. We trace global and local pointers, moving live objects to the to space and updating pointers. This is illustrated in Figure 11.14. (Dashed arrows are forwarding pointers). We have yet to handle pointers internal to copied heap objects. All copied heap objects are traversed. Objects referenced are copied and internal pointers are updated. Finally, the to and from spaces are interchanged, and heap allocation resumes just beyond the last copied object. This is illustrated in Figure 11.15.

The biggest advantage of copying collectors is their speed. Only live objects are copied; deallocation of dead objects is essentially free. In fact, garbage collection can be made, on average, as fast as you wish—simply make the heap bigger. As the heap gets bigger, the time between collections increases, reducing the number of times a live object must be copied. In the limit, objects are never copied, so garbage collection becomes free!

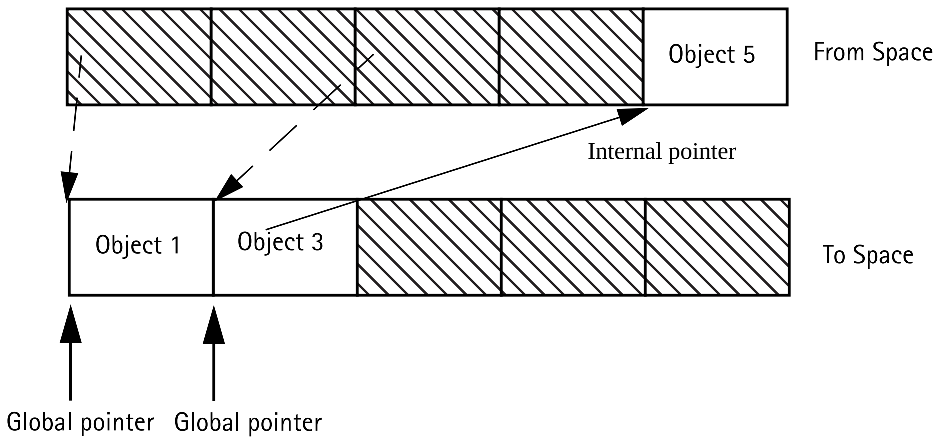

Figure 11.14 Copying Garbage Collection (b)

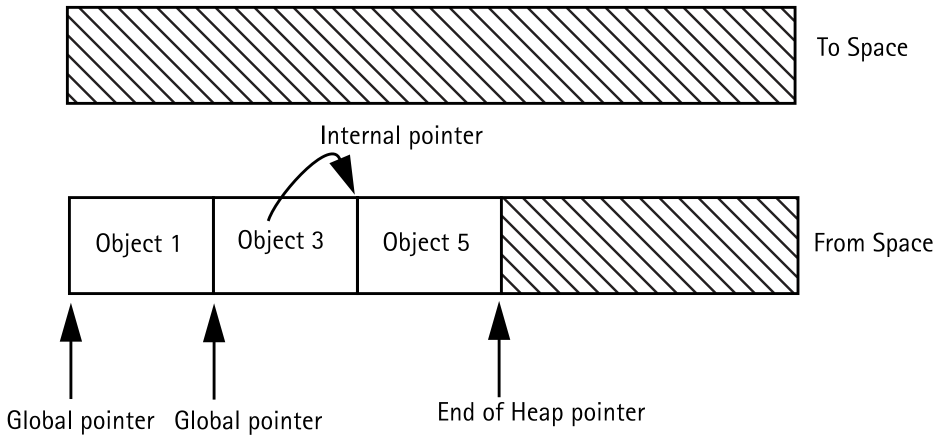

Figure 11.15 Copying Garbage Collection (c)

Of course, we can't increase the size of heap memory to infinity. In fact, we don't want to make the heap so large that paging is required, since swapping pages to disk is dreadfully slow. If we can make the heap large enough that the lifetime of most heap objects is less than the time between collections, then deallocation of short-lived objects will appear to be free, though longer-lived objects will still exact a cost.

Aren't copying collectors terribly wasteful of space? After all, at most only half of the heap space is actually used. The reason for this apparent inefficiency is that *any* garbage collector that does compaction must have an area to copy live objects to. Since in the worst case *all* heap objects could be live, the target area must be as large as the heap itself. To avoid copying objects more than once, copying collectors reserve a to space as big as the from space. This is essentially a space-time trade-off, making such collectors very fast at the expense of possibly wasted space.

If we have reason to believe that the time between garbage collections will be greater than the average lifetime of most heaps objects, we can improve our use of heap space. Assume that 50% or more of the heap will be garbage when the collector is called. We can then divide the heap into 3 segments, which we'll call A, B and C. Initially, A and B will be used as the from space, utilizing 2/3 of the heap. When we copy live objects, we'll copy them into segment C, which will be big enough if half or more of the heap objects are garbage. Then we treat C and A as the from space, using B as the to space for the next collection. If we are unlucky and more than 1/2 the heap contains live objects, we can still get by. Excess objects are copied onto an auxiliary data space (perhaps the stack), then copied into A after all live objects in A have been moved. This slows collection down, but only rarely (if our estimate of 50% garbage per collection is sound). Of course, this idea generalizes to more than 3 segments. Thus if 2/3 of the heap were garbage (on average), we could use 3 of 4 segments as from space and the last segment as to space.

Generational TechniquesThe great strength of copying collectors is that they do no work for objects that are born and die between collections. However, not all heaps objects are so short-lived. In fact, some heap objects are very long-lived. For example, many programs create a dynamic data structure at their start, and utilize that structure throughout the program. Copying collectors handle long-lived objects poorly. They are repeatedly traced and moved between semispaces without any real benefit.

Generational garbage collection techniques [Unger 1984] were developed to better handle objects with varying lifetimes. The heap is divided into two or more *generations*, each with its own to and from space. New objects are allocated in the youngest generation, which is collected most frequently. If an object survives across one or more collections of the youngest generation, it is “promoted” to the next older generation, which is collected less often. Objects that survive one or more collections of this generation are then moved to the next older generation. This continues until very long-lived objects reach the oldest generation, which is collected very infrequently (perhaps even never).

The advantage of this approach is that long-lived objects are “filtered out,” greatly reducing the cost of repeatedly processing them. Of course, some long-lived objects will die and these will be caught when their generation is eventually collected.

An unfortunate complication of generational techniques is that although we collect older generations infrequently, we must still trace their pointers in case they reference an object in a newer generation. If we don’t do this, we may mistake a live object for a dead one. When an object is promoted to an older generation, we can check to see if it contains a pointer into a younger generation. If it does, we record its address so that we can trace and update its pointer. We must also detect when an existing pointer inside an object is changed. Sometimes we can do this by checking "dirty bits" on heap pages to see which have been updated. We then trace all objects on a page that is dirty. Otherwise, whenever we assign to a pointer that already has a value, we record the address of the pointer that is changed. This information then allows us to only trace those objects in older generations that might point to younger objects.

Experience shows that a carefully designed generational garbage collectors can be very effective. They focus on objects most likely to become garbage, and spend little overhead on long-lived objects. Generational garbage collectors are widely used in practice.

Conservative Garbage CollectionThe garbage collection techniques we've studied all require that we identify pointers to heap objects accurately. In strongly typed languages like Java or ML, this can be done. We can table the addresses of all global pointers. We can include a code value in a frame (or use the return address stored in a frame) to determine the routine a frame corresponds to. This allows us to then determine what offsets in the frame contain pointers. When heap objects are allocated, we can include a type code in the object's header, again allowing us to identify pointers internal to the object.

Languages like C and C++ are weakly typed, and this makes identification of pointers much harder. Pointers may be type-cast into integers and then back into pointers. Pointer arithmetic allows pointers into the middle of an object. Pointers in frames and heap objects need not be initialized, and may contain random values. Pointers may overlay integers in unions, making the current type a dynamic property.

As a result of these complications, C and C++ have the reputation of being incompatible with garbage collection. Surprisingly, this belief is false. Using *conservative garbage collection*, C and C++ programs *can* be garbage collected.

The basic idea is simple—if we can't be sure whether a value is a pointer or not, we'll be conservative and assume it is a pointer. If what we think is a pointer isn't, we may retain an object that's really dead, but we'll find all valid pointers, and never incorrectly collect a live object. We may mistake an integer (or a floating value, or even a string) as an pointer, so compaction in any form *can't* be done. However, mark-sweep collection will work.

Garbage collectors that work with ordinary C programs have been developed [BW 1988]. User programs need not be modified. They simply are linked to different library routines, so that `malloc` and `free` properly support the garbage collector. When new heap space is required, dead heap objects may be automatically collected, rather than relying entirely on explicit `free` commands (though `free`s are allowed; they sometimes simplify or speed heap reuse).

With garbage collection available, C programmers need not worry about explicit heap management. This reduces programming effort and eliminates errors in which objects are prematurely freed, or perhaps never freed. In fact, experiments have shown [Zorn 93] that conservative garbage collection is very competitive in performance with application-specific manual heap management.

Exercises

1. Show the frame layout corresponding to the following C function:

```
int f(int a, char *b){
    char c;
    double d[10];
    float e;
    ...
}
```

Assume control information requires 3 words and that `f`'s return value is left on the stack. Be sure to show the offset of each local variable in the frame and be sure to provide for proper alignment (`int`s and `float`s on word boundaries and `double`s on doubleword boundaries).

2. Show the sequence of frames, with dynamic links, on the stack when `r(3)` is executed assuming we start execution (as usual) with a call to `main()`.

```
r(flag){
    printf("Here !!!\n");
```

```
}
q(flag){
    p(flag+1);
}
p(int flag){
    switch(flag){
      case 1: q(flag);
      case 2: q(flag);
      case 3: r(flag);
}
main(){
    p(1);
}
```

3. Consider the following C-like program that allows subprograms to nest. Show the sequence of frames, with static links, on the stack when `r(16)` is executed assuming we start execution (as usual) with a call to `main()`. Explain how the values of `a`, `b` and `c` are accessed in `r`'s `print` statement.

```
p(int a){
   q(int b){
      r(int c){
         print(a+b+c);
      }
      r(b+3);
```

```
        }
        s(int d){
            q(d+2);
        }
        s(a+1);
    }
    main(){
         p(10);
    }
```

4. Reconsider the C-like program shown in Exercise 3, this time assuming display registers are used to access frames (rather than static links). Explain how the values of a, b and c are accessed in r's print statement.
5. Consider the following C function. Show the content and structure of f's frame. Explain how the offsets of f's local variables are determined.

```
    int f(int a, int b[]){
        int i = 0, sum = 0;
        while (i < 100){
            int val = b[i]+a;
            if (b[i]>b[i+1]) {
                int swap = b[i];
                b[i] = b[i+1];
                b[i+1] = swap;
            } else {
```

```
            int avg = (b[i]+b[i+1])/2;
            b[i] = b[i+1] = avg; }
         sum += val;
         i++;
      }
      return sum;
   }
```

6. Although the first release of Java did not allow classes to nest, subsequent releases did. This introduced problems of nested access to objects, similar to those found when subprograms are allowed to nest. Consider the following Java class definition.

```
class Test {
   class Local {
        int b;
        int v(){return a+b;}
        Local(int val){b=val;}
   }
   int a = 456;
   void m(){
      Local temp = new Local(123);
      int c = temp.v();
   }
}
```

Note that method `v()` of class `Local` has access to field `a` of class `Test` as well as field `b` of class `Local`. However, when `temp.v()` is called it is given a direct reference only to `temp`. Suggest a variant of static links that can be used to implement nested classes so that access to all visible objects is provided.

7. Assume we organize a heap using reference counts. What operations must be done when a pointer to a heap object is assigned? What operations must be done when a scope is opened and closed?

8. Some languages, including C and C++, contain an operation that creates a pointer to a data object. That is, `p = &x` takes the address of object `x`, whose type is `t`, and assigns it to `p`, whose type is `t*`.

   How is management of the run-time stack complicated if it is possible to create pointers to arbitrary data objects in frames? What restrictions on the creation and copying of pointers to data objects suffice to guarantee the integrity of the run-time stack?

9. Consider a heap allocation strategy we shall term *worst fit*. Unlike best fit, which allocates a heap request from the free space block that is closest to the requested size, worst fit allocates a heap request from the largest available free space block. What are the advantages and disadvantages of worst fit as compared with the best fit, first fit, and next fit heap allocation strategies?

10. The performance of complex algorithms is often evaluated by simulating their behavior. Create a program that simulates a random sequence of heap allocations and deallocations. Use it to compare the average number of iterations

that the best fit, first fit, and next fit heap allocation techniques require to find and allocate space for a heap object.

11. In a strongly typed language like Java all variables and fields have a fixed type known at compile-time. What run-time data structures are needed in Java to implement the mark phase of a mark-sweep garbage collector in which all accessible ("live") heap objects are marked?

12. The second phase of a mark-sweep garbage collector is the sweep phase, in which all unmarked heap objects are returned to the free-space list.

Detail the actions needed to step through the heap, examining each object and identifying those that have been not been marked (and hence are garbage).

13. In a language like C or C++ (without unions), the marking phase of a mark-sweep garbage collector is complicated by the fact that pointers to active heap objects may reference data within an object rather than the object itself. For example, the sole pointer to an array may be to an internal element, or the sole pointer to a class object may be a pointer to one of the object's fields.

How must your solution to Exercise 11 be modified if pointers to data within an object are allowed?

14. One of the attractive aspects of conservative garbage collection is its simplicity. We need not store detailed information on what global, local and heap variables are pointers. Rather, any word that *might* be a heap pointer is treated as if *is* a pointer.

What criteria would you use to decide if a given word in memory is possibly a pointer? How would you adapt your answer to Exercise 13 to handle what appear to be pointers to data within a heap object?

15. One of the most attractive aspects of copying garbage collectors is that collecting garbage actually costs nothing since only live data objects are identified and moved. Assuming that the total amount of heap space live at any point is constant, show that the average cost of garbage collection (per heap object allocated) can be made arbitrarily cheap by simply by increasing the memory size allocated to the heap.

16. Copying garbage collection can be improved by identifying long-lived heap objects and allocating them in an area of the heap that is not collected.

What compile-time analyses can be done to identify heap objects that will be long-lived? At run-time, how can we efficiently estimate the "age" of a heap object (so that long-lived heap objects can be specially treated)?

17. An unattractive aspect of both mark-sweep and copying garbage collects is that they are batch-oriented. That is, they assume that periodically a computation can be stopped while garbage is identified and collected. In interactive or real-time programs, pauses can be quite undesirable. An attractive alternative is *concurrent garbage collection* in which a garbage collection process runs concurrently with a program.

Consider both mark-sweep and copying garbage collectors. What phases of each can be run concurrently while a program is executing (that is, while the program is changing pointers and allocating heap objects)? What changes to garbage collection algorithms can facilitate concurrent garbage collection?

18. Assume we are compiling a language like Pascal or Ada that allows nested procedures. Further, assume we are in procedure `P`, at nesting level m, and wish to access variable `v` declared in Procedure `Q`, which is at nesting level n. If `v` is at offset of in Q's frame, and if static links are always stored at offset sl in a frame, what expression must be evaluated to access `v` from procedure `P` using static links?