MODULE IV

Storage Organization -
Stack Allocation of Space -
Access to Nonlocal Data on the Stack -
Heap Management -
Introduction to Garbage Collection

RUNTIME ENVIRONMENT

By **runtime**, we mean a program in execution. **Runtime environment** is a state of the target machine, which may include software libraries, **environment** variables, etc., to provide services to the processes running in the system.

I. Storage Organization

- When the target program executes then it runs in its own logical address space in which the value of each program has a location.
- The logical address space is shared among the compiler, operating system and target machine for management and organization. The operating system is used to map the logical address into physical address which is usually spread throughout the memory.

The run-time representation of an object program in the logical address space consists of data and program areas as shown in Fig. 5.1

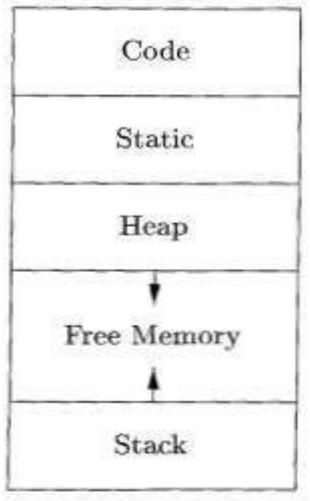


Figure 7.1: Typical subdivision of run-time memory into code and data areas

Storage needed for a name is determined from its type.

- Runtime storage comes into blocks, where a byte is used to show the smallest unit of addressable memory. Using the four bytes a machine word can form. Object of multibyte is stored in consecutive bytes and gives the first byte address.
- Run-time storage can be subdivide to hold the different components of an executing program:

1. Generated executable code
2. Static data objects
3. Dynamic data-object- heap
4. Automatic data objects- stack

Two areas, *Stack* and *Heap*, are at the opposite ends of the remainder of the address space. These areas are dynamic; their size can change as the program executes. Stack to support call/return policy for procedures.Heap to store data that can outlive a call to a procedure. . The heap is used to manage allocate and deallocate data.

Static Versus Dynamic Storage Allocation

The layout and allocation of data to memory locations in the run-time environment are key issues in storage management. The two terms *static* and *dynamic* distinguish between compile time and run time, respectively. We say that a storage-allocation decision is

Static:- if it can be made by the compiler looking only at the text of the program, not at what the program does when it executes.
Dynamic:- if it can be decided only while the program is running.

Compilers use following two strategies for dynamic storage allocation:

Stack storage. Names local to a procedure are allocated space on a stack. stack supports the normal call/return policy for procedures.

Heap storage. Data that may outlive the call to the procedure that created it is usually allocated on a "heap" of reusable storage.The heap is an area of virtual memory that allows objects or other data elements to obtain storage when they are created and to return that storage when they are invalidated.

II. Stack allocation of space

1 Activation Trees

2 Activation Records

3 Calling Sequences

4 Variable-Length Data on the Stack

Each time a procedure is called, space for its local variables is pushed onto a stack, and when the procedure terminates, that space is popped off the stack.

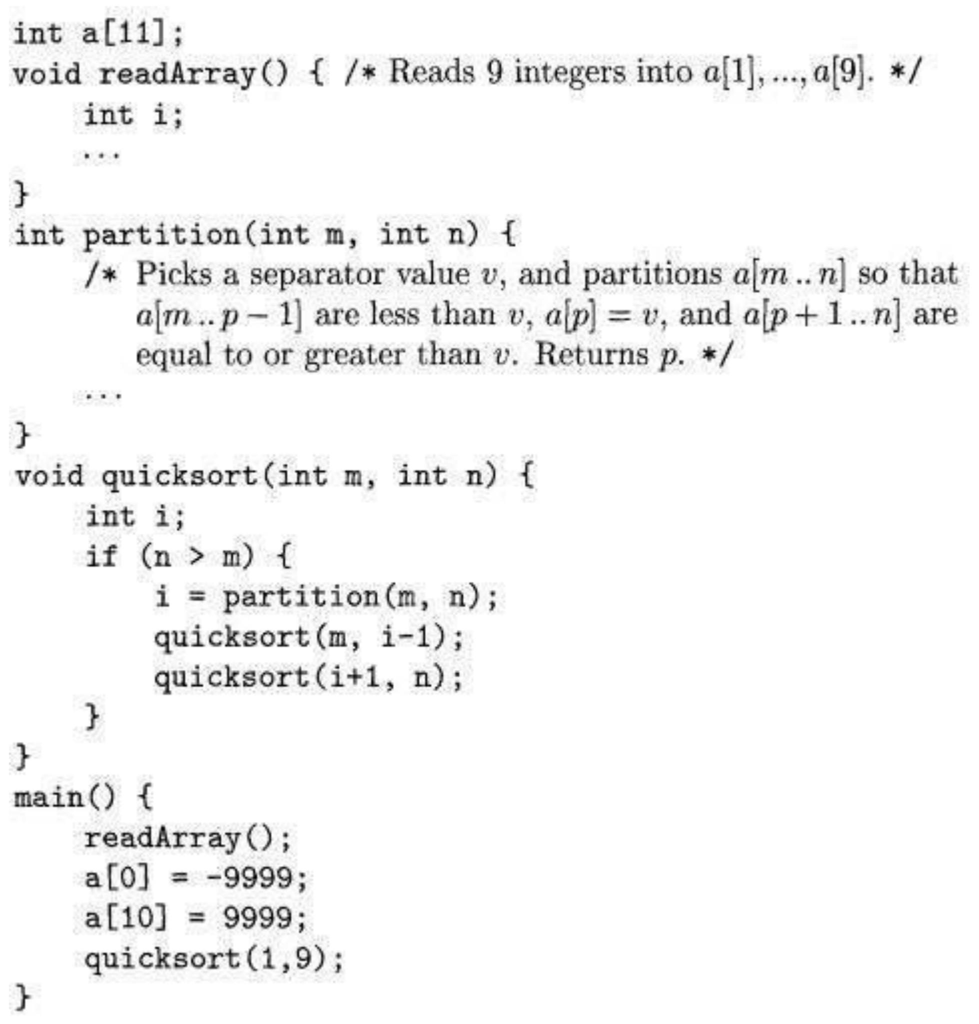
1 Activation Trees

Stack allocation is a valid allocation for procedures since procedure calls are nested

Example: quicksort algorithm

The main function has three tasks. It calls *readArray,* sets the sentinels, and then calls *quicksort* on the entire data array.

Procedure activations are nested in time. If an activation of procedure *p* calls procedure *q*, then that activation of *q* must end before the activation of *p* can end.



```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p-1] are less than v, a[p] = v, and a[p+1..n] are
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

Figure 7.2: Sketch of a quicksort program

Represent the activations of procedures during the running of an entire program by a tree, called an activation tree. Each node corresponds to one activation, and the root is the activation of the "main" procedure that initiates execution of the program. At a node for an activation of procedure p, the children correspond to activations of the procedures called by this activation of p.

```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```

Figure 7.3: Possible activations for the program of Fig. 7.2

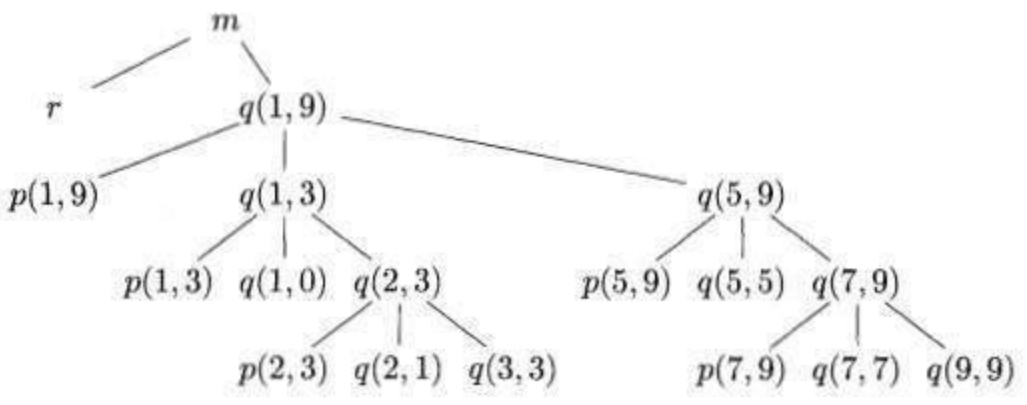


Figure 7.4: Activation tree representing calls during an execution of *quicksort*

2 Activation Records

a. Procedure calls and returns are usually managed by a run-time stack called the control stack.
b. Each live activation has an activation record (sometimes called a frame)
c. The root of activation tree is at the bottom of the stack
d. The current execution path specifies the content of the stack with the last
e. Activation has record in the top of the stack.

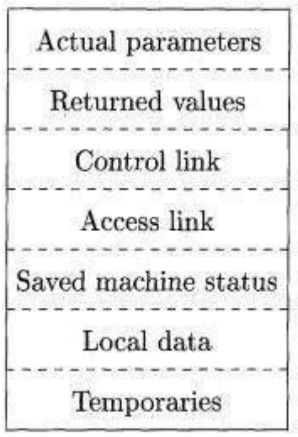


Figure 7.5: A general activation record

An activation record is used to store information about the status of the machine, such as the value of the program counter and machine registers, when a procedure call occurs. When control returns from the call, the activation of the calling procedure can be restarted after restoring the values of relevant registers and setting the program counter to the point immediately after the call. Data objects whose lifetimes are contained in that of an activation can be allocated on the stack along with other information associated with the activation.

An activation record contains all the necessary information required to call a procedure. An activation record may contain the following units (depending upon the source language used).

Temporaries	Stores temporary and intermediate values of an expression.
Local Data	Stores local data of the called procedure.
Machine Status	Stores machine status such as Registers, Program Counter etc., before the procedure is called.
Control Link	Stores the address of activation record of the caller procedure.
Access Link	Stores the information of data which is outside the local scope.
Actual Parameters	Stores actual parameters, i.e., parameters which are used to send input to the called procedure.
Return Value	Stores return values.

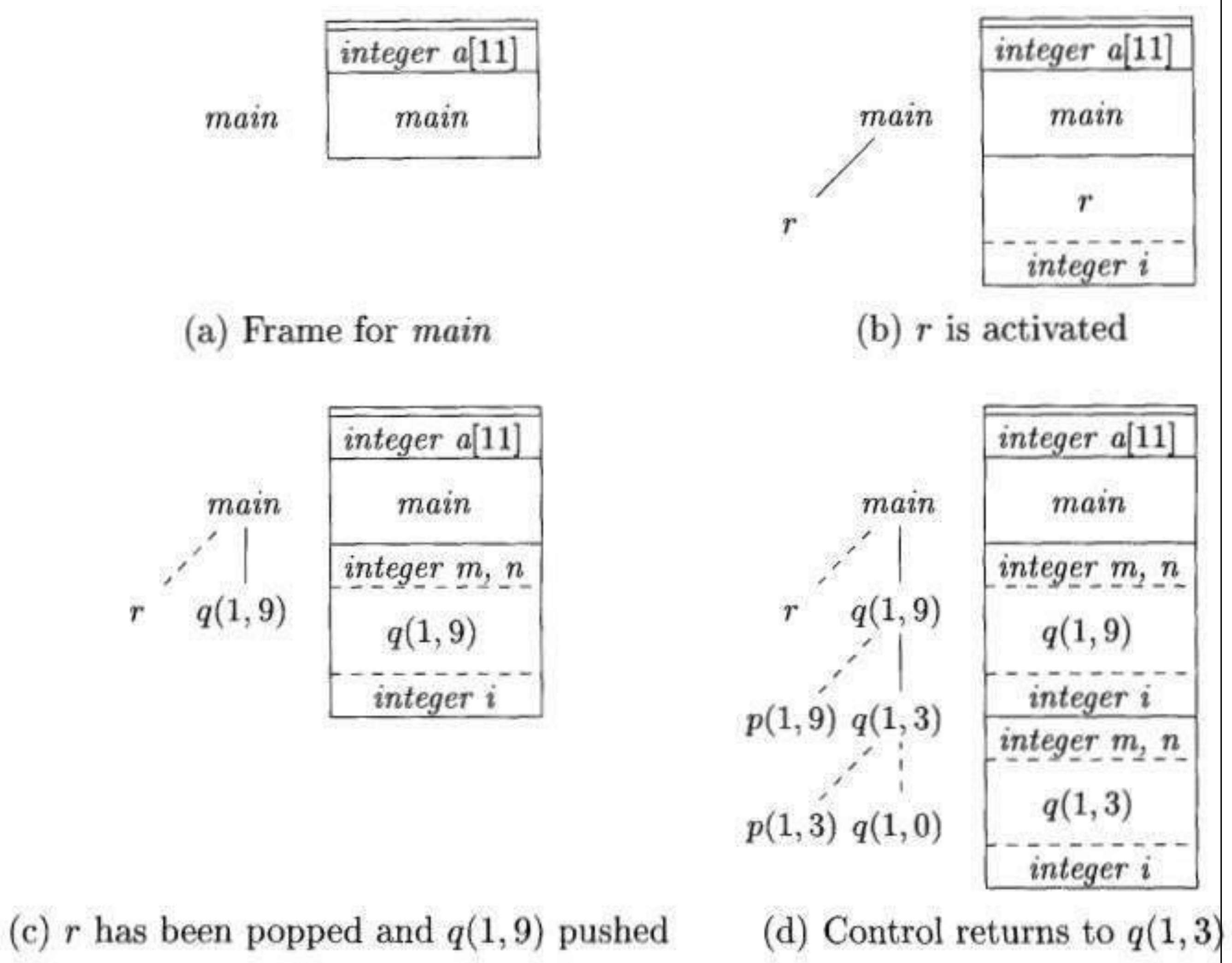
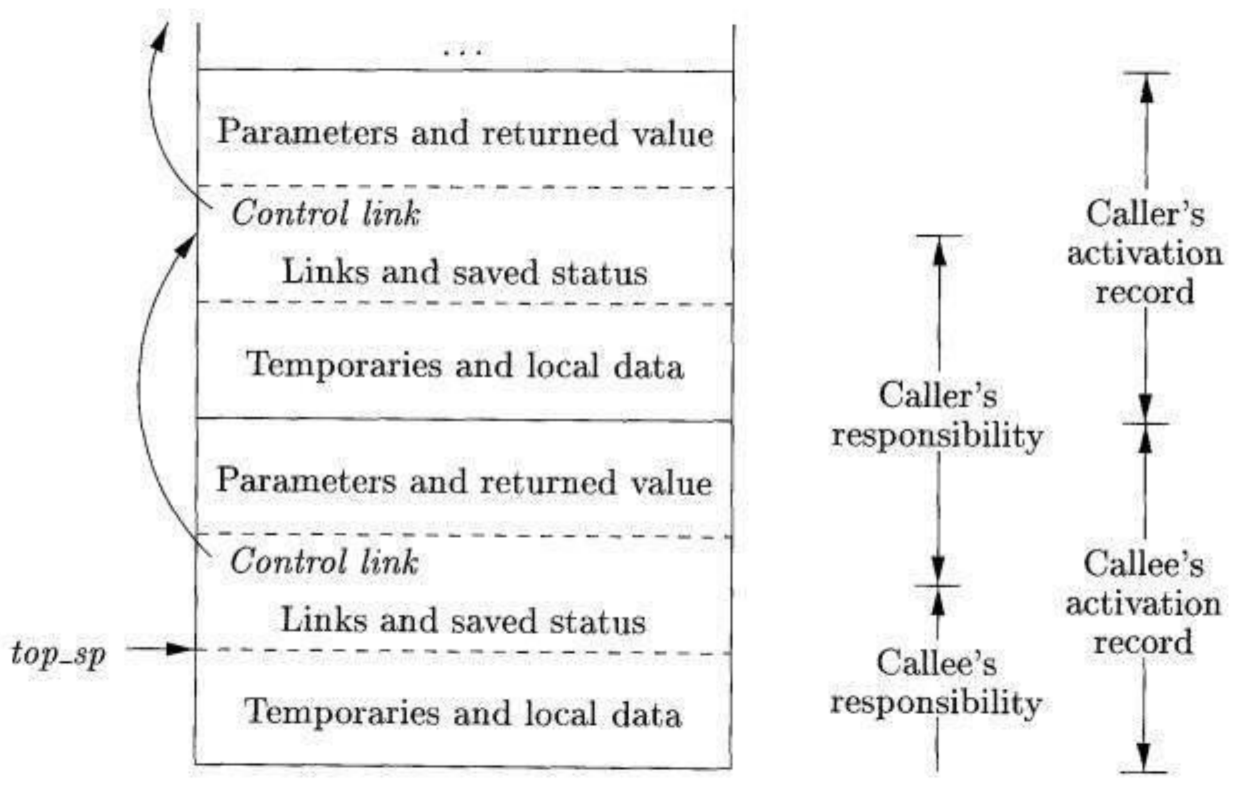


Figure 7.6: Downward-growing stack of activation records

3 Calling Sequences

Designing calling sequences and the layout of activation records, the following

1. Values communicated between caller and callee are generally placed at the beginning of callee's activation record
2. Fixed-length items: are generally placed at the middle. such items typically include the control link, the access link, and the machine status fields.
3. Items whose size may not be known early enough: are placed at the end of activation record
4. We must locate the top-of-stack pointer judiciously: a common approach is to have it point to the end of fixed length fields in the activation record.



...
Parameters and returned value
Control link
Links and saved status
Temporaries and local data
Parameters and returned value
Control link
Links and saved status
top_sp
Temporaries and local data
Caller's responsibility
Callee's responsibility
Caller's activation record
Callee's activation record

Figure 7.7: Division of tasks between caller and callee

A register topsp points to the end of the machine-status field in the current top activation record. This position within the callee's activation record is known to the caller, so the caller can be made responsible for setting topsp before control is passed to the callee. The calling sequence and its division between caller and callee is as follows:

1. The caller evaluates the actual parameters.

The caller stores a return address and the old value of *topsp* into the callee's activation record. The caller then increments *topsp* to the position shown in Fig. 7.7. That is, *topsp* is moved past the caller's local data and temporaries and the callee's parameters and status fields.

The callee saves the register values and other status information.

The callee initializes its local data and begins execution.

A suitable, corresponding **return sequence** is:

1. The callee places the return value next to the parameters, as in Fig. 7.5.
2. Using information in the machine-status field, the callee restores *topsp* and other registers, and then branches to the return address that the caller placed in the status field.

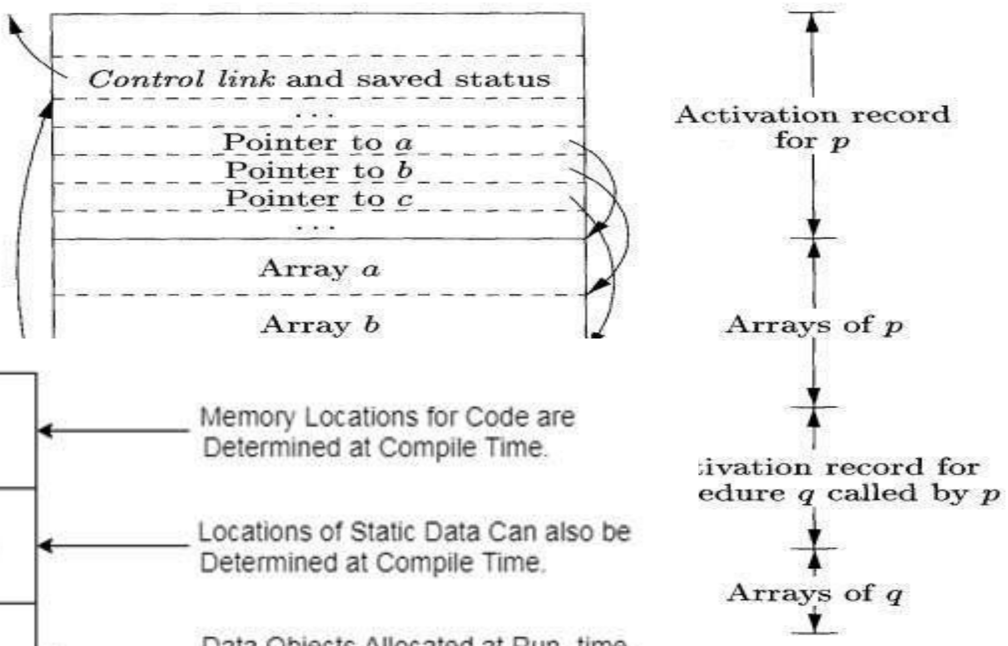
3. Although *topsp* has been decremented, the caller knows where the return value is, relative to the current value of *topsp;* the caller therefore may use that value.

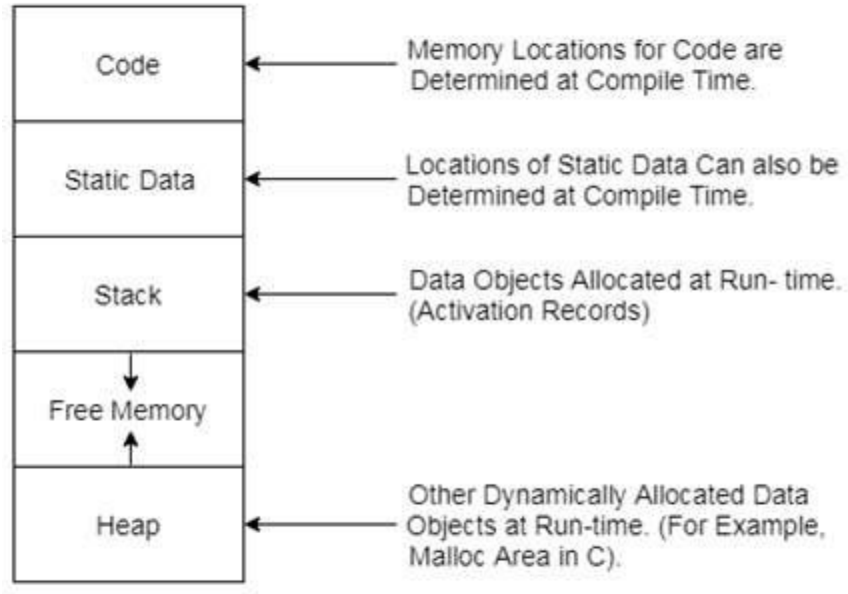
4. Variable-Length Data on the Stack

The run-time memory-management system must deal frequently with the allocation of space for objects the sizes of which are not known at compile time, but which are local to a procedure and thus may be allocated on the stack.

it is possible to allocate objects, arrays, or other structures of unknown size on the stack. The reason to prefer placing objects on the stack if possible is that we avoid the expense of garbage collecting their space. Note that the stack can be used only for an object if it is local to a procedure and becomes inaccessible when the procedure returns.

A common strategy for allocating variable-length arrays (i.e., arrays whose size depends on the value of one or more parameters of the called procedure) is shown in Fig. 7.8. The same scheme works for objects of any type if they are local to the procedure called and have a size that depends on the parameters of the call.





Code
Static Data
Stack
Free Memory
Heap
Memory Locations for Code are Determined at Compile Time.
Locations of Static Data Can also be Determined at Compile Time.
Data Objects Allocated at Run- time. (Activation Records)
Other Dynamically Allocated Data Objects at Run-time. (For Example, Malloc Area in C).

ted arrays

d by p. The activation record for q
cated beyond that . Access to the

III. Access to Non Local data on the stack

Consider how procedures access their data. Especially im-portant is the mechanism for finding data used within a procedure *p* but that does not belong to *p*

1 Data Access Without Nested Procedures

Names are either local to the procedure in question or are declared globally.

1. For global names the address is known statically at compile time providing there is only one source file. If multiple source files, the linker knows. In either case no reference to the activation record is needed; the addresses are know prior to execution.
2. For names local to the current procedure, the address needed is in the AR at a known-at-compile-time constant offset from the sp. In the case of variable size arrays, the constant offset refers to a pointer to the actual storage.

2 Issues With Nested Procedures

Access becomes far more complicated when a language allows procedure dec-larations to be nested .The reason is that knowing at compile time that the declaration of *p* is immediately nested within *q* does not tell us the relative positions of their activation records at run time. In fact, since either *p* or *q* or both may be recursive, there may be several activation records of *p* and/or *q* on the stack.

Finding the declaration that applies to a nonlocal name *x in a* nested pro-cedure *p* is a static decision; it can be done by an extension of the static-scope rule for blocks. Suppose *x* is declared in the enclosing procedure *q*. Finding the relevant activation of *q* from an activation of *p* is a dynamic decision; it re-quires additional run-time information about activations. One possible solution is to use access links.

3. A Language With Nested Procedure Declarations

In various languages with nested procedures, one of the most influential is ML.

ML is a *functional language*, meaning that variables, once declared and initialized, are not changed. There are only a few exceptions, such as the array, whose elements can be changed by special function calls.

• Variables are defined, and have their unchangeable values initialized,

v a l (name) = (expression)

• Functions are defined using the syntax:

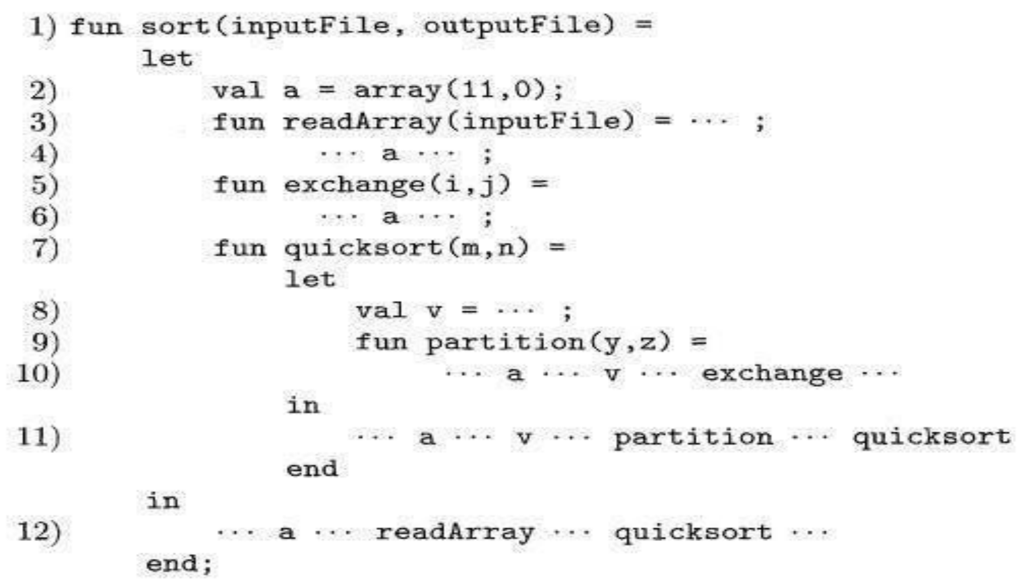
fun (name) ((arguments)) = (body)

• For function bodies, use let-statements of the form:

let (list of definitions) in (statements) end The definitions are normally v a l or fun statements. The scope of each such definition consists of all following definitions, up to the in, and all the statements up to the end. Most importantly, function definitions can be nested. For example, the body of a function p can contain a let-statement that includes the definition of another (nested) function q. Similarly, q can have function definitions within its own body, leading to arbitrarily deep nesting of function

4. Nesting Depth

Nesting depth is 1 to procedures that are not nested within any other procedure. For example, all C functions are at nesting depth 1. However, if a procedure *p* is defined immediately within a procedure at nesting depth *i*, then give *p* the nesting depth *i*



```
 1) fun sort(inputFile, outputFile) =
        let
 2)         val a = array(11,0);
 3)         fun readArray(inputFile) = ··· ;
 4)                 ··· a ··· ;
 5)         fun exchange(i,j) =
 6)                 ··· a ··· ;
 7)         fun quicksort(m,n) =
                let
 8)                 val v = ··· ;
 9)                 fun partition(y,z) =
10)                         ··· a ··· v ··· exchange ···
                in
11)                 ··· a ··· v ··· partition ··· quicksort
                end
        in
12)         ··· a ··· readArray ··· quicksort ···
        end;
```

Figure 7.10: A version of quicksort, in ML style, using nested functions

5. Access Links

A direct implementation of the normal static scope rule for nested functions is obtained by adding a pointer called the *access link* to each activation record. If procedure *p* is nested immediately within procedure *q* in the source code, then the access link in any activation of *p* points to the most recent activation of *q*. Note that the nesting depth of *q* must be exactly one less than the nesting depth of p. Access links form a chain from the activation record at the top of the stack to a sequence of activations at progressively lower nesting depths.

Figure 7.11 shows a sequence of stacks that might result from execution of the function *sort* of Fig. 7.10. In Fig. 7.11(a), we see the situation after *sort* has called *readArray* to load input into the array *a* and then called *quicksort(l,* 9) to sort the array. The access link from *quicksort(l,* 9) points to the activation record for *sort*, not because *sort* called *quicksort* but because *sort* is the most closely nested function surrounding *quicksort* in the program.

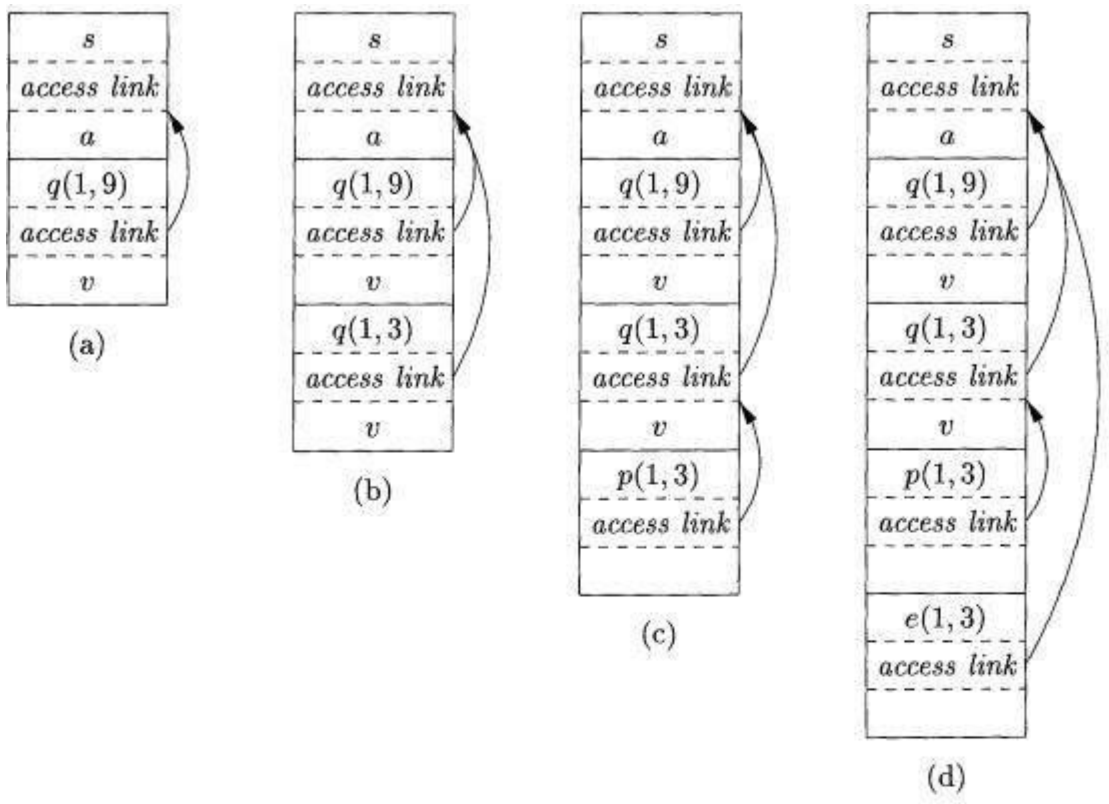


Figure 7.11: Access links for finding nonlocal data

see a recursive call to *quicksort(l,* 3), followed by a call to *partition*, which calls *exchange.* Notice that *quicksort(l,* 3)'s access link points to *sort*, for the same reason that *quicksort(l,* 9)'s does.

6. Manipulating Access Links

The harder case is when the call is to a procedure-parameter; in that case, the particular procedure being called is not known until run time, and the nesting depth of the called procedure may differ in different executions of the call. consider situation when a procedure *q* calls procedure *p*, explicitly. There are three cases:

1. Procedure *p* is at a higher nesting depth than *q*. Then p must be defined immediately within *q*, or the call by *q* would not be at a position that is within the scope of the procedure name *p*. Thus, the nesting depth of *p* is exactly one greater than that of *q*, and the access link from *p* must lead to *q*. It is a simple matter for the calling sequence to include a step that places in the access link for *p* a pointer to the activation record of *q*.

2. The call is recursive, that is, *p* = *q*.Then the access link for the new activation record is the same as that of the activation record below it.

3. The nesting depth np of p is less than the nesting depth nq of q. In order for the call within q to be in the scope of name p, procedure q must be nested within some procedure r, while p is a procedure defined immediately within r. The top activation record for r can therefore be found by following the chain of access links, starting in the activation record for q, for nq — np + 1 hops. Then, the access link for p must go to this activation of r.

7. Access Links for Procedure Parameters

When a procedure *p* is passed to another procedure *q* as a parameter, and *q* then calls its parameter (and therefore calls *p* in this activation of *q)*, it is possible that *q* does not know the context in which *p* appears in the program. If so, it is impossible for *q* to know how to set the access link for *p*. The solution to this is, when procedures are used as parameters, the caller needs to pass, along with the name of the procedure-parameter, the proper access link for that parameter. The caller always knows the link, since if *p* is passed by procedure *r* as an actual parameter, then *p* must be a name accessible to r, and therefore, r can determine the access link for *p* exactly as if *p* were being called by *r* directly.

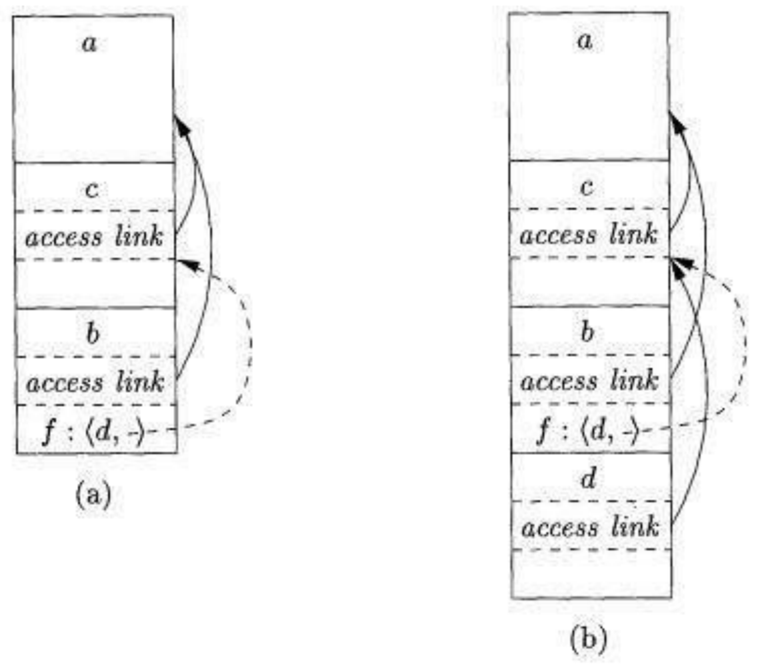


Figure 7.13: Actual parameters carry their access link with them

8. Displays

One problem with the access-link approach to nonlocal data is that if the nesting depth gets large, we may have to follow long chains of links to reach the data we need. A more efficient implementation uses an auxiliary array *d*, called the *display*, which consists of one pointer for each nesting depth. We arrange that, at all times, *d[i]* is a pointer to the highest activation record on the stack for any procedure at nesting depth *i*. Examples of a display are shown in Fig. 7.14.

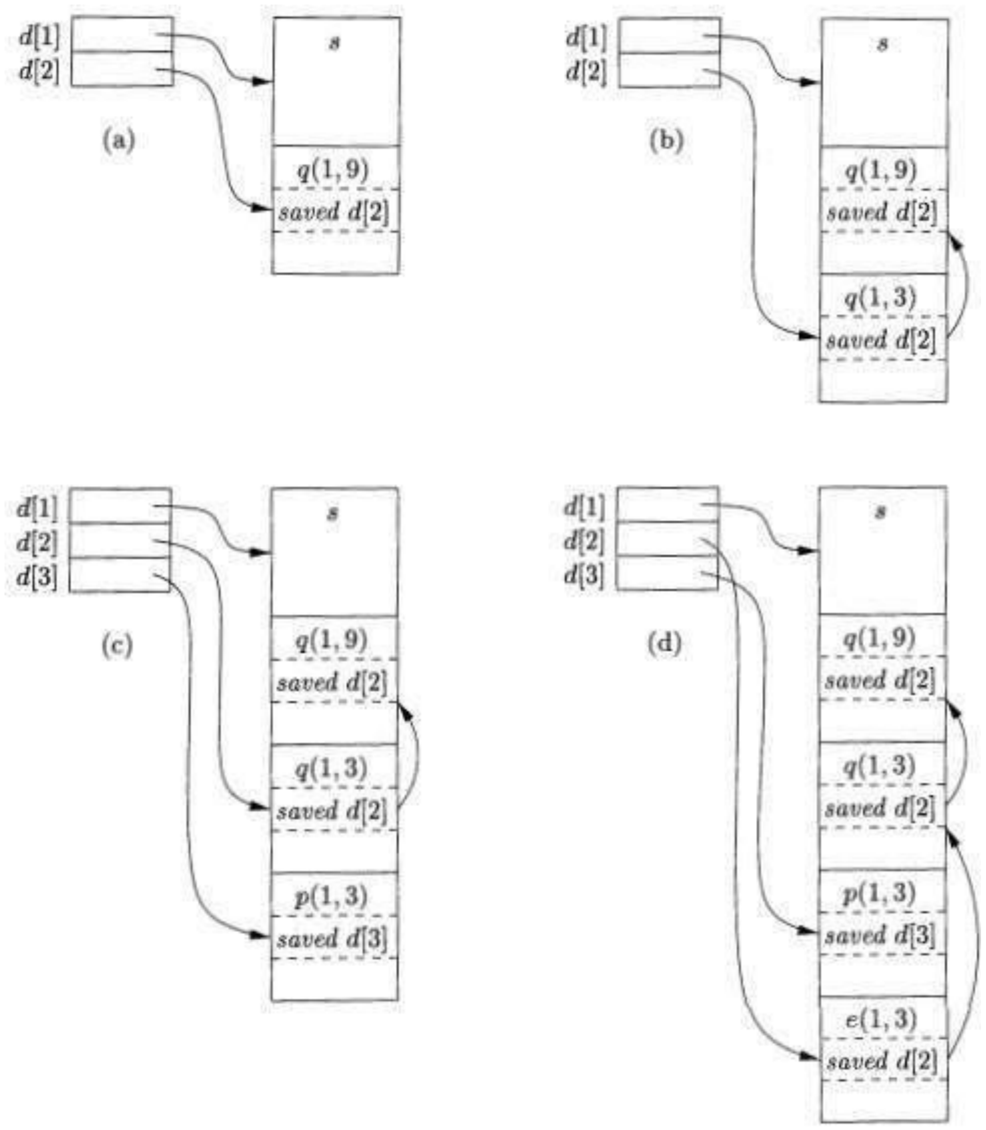


Figure 7.14: Maintaining the display

In order to maintain the display correctly, we need to save previous values of display entries in new activation records.

IV. Heap Management

The heap is the portion of the store that is used for data that lives indefinitely, or until the program explicitly deletes it.

1 The Memory Manager

2 The Memory Hierarchy of a Computer

3 Locality in Programs

4 Reducing Fragmentation

5 Manual Deallocation Requests

1 The Memory Manager

It performs two basic functions:

• **Allocation**. When a program requests memory for a variable or object,[3] the memory manager produces a chunk of contiguous heap memory of the requested size. If possible, it satisfies an allocation request using free space in the heap; if no chunk of the needed size is available, it seeks to increase the heap storage space by getting consecutive bytes of virtual memory from the operating system. If space is exhausted, the memory manager passes that information back to the application program.

• **Deallocation**. The memory manager returns deallocated space to the pool of free space, so it can reuse the space to satisfy other allocation requests. Memory managers typically do not return memory to the operating sys-tem, even if the program's heap usage drops.

Thus, the memory manager must be prepared to service, in any order, allo-cation and deallocation requests of any size, ranging from one byte to as large as the program's entire address space.

Here are the properties we desire of memory managers:

• **Space Efficiency**. A memory manager should minimize the total heap space needed by a program. Larger programs to run in a fixed virtual address space..

• **Program Efficiency.** A memory manager should make good use of the memory subsystem to allow programs to run faster.

• **Low Overhead**. Because memory allocations and deallocations are fre-quent operations in many programs, it is important that these operations be as efficient as possible. That is, we wish to minimize the *overhead*

2. The Memory Hierarchy of a Computer

The efficiency of a program is determined not just by the number of instructions executed, but also by how long it takes to execute each of these instructions. The time taken to execute an instruction can vary significantly, since the time taken to access different parts of memory can vary from nanoseconds to milliseconds. Data-intensive programs can therefore benefit significantly from optimizations that make good use of the memory subsystem.

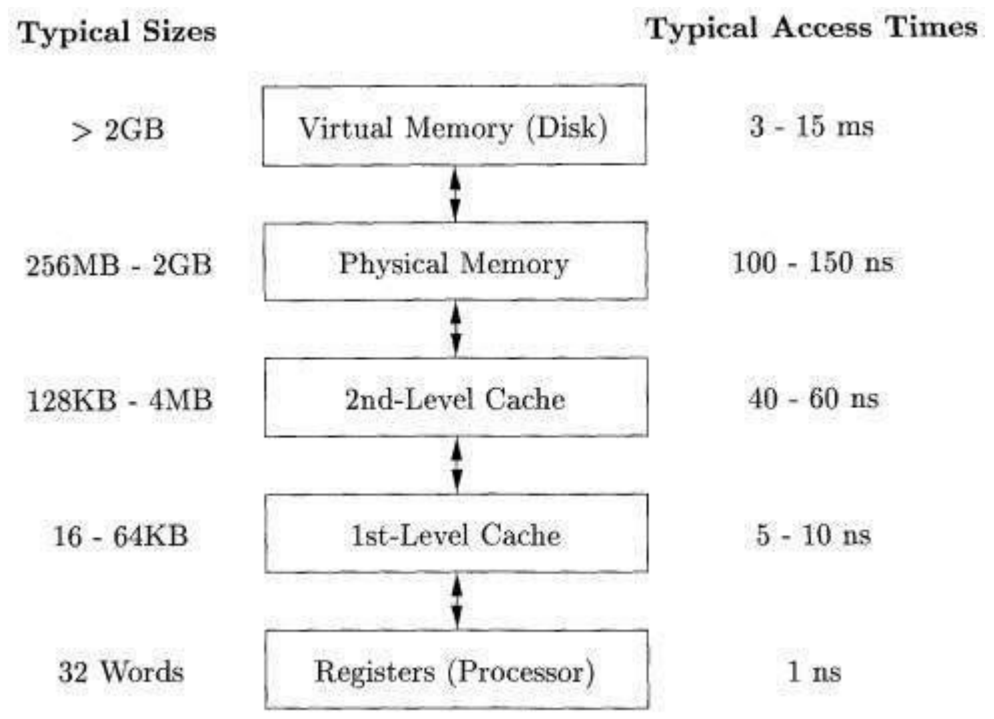


Figure 7.16: Typical Memory Hierarchy Configurations

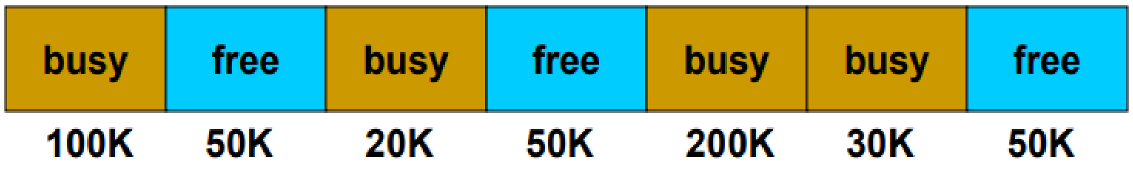
3. Locality in Programs

Most programs exhibit a high degree of locality; that is, they spend most of their time executing a relatively small fraction of the code and touching only a small fraction of the data. We say that a program has temporal locality if the memory locations it accesses are likely to be accessed again within a short period of time. We say that a program has *spatial locality* if memory locations close to the location accessed are likely also to be accessed within a short period of time.

Programs spend 90% of their time executing 10% of the code. Programs often contain many instructions that are never executed. Programs built with components and libraries use only a small fraction of the provided functionality.

The typical program spends most of its time executing innermost loops and tight recursive cycles in a program. By placing the most common instructions and data in the fast-but-small storage, while leaving the rest in the slow-but-large storage. Average memory-access time of a program can be lowered significantly.

4. Reducing Fragmentation



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

With each deallocation request, the freed chunks of memory are added back to the pool of free space. We coalesce contiguous holes into larger holes, as the holes can only get smaller otherwise. If we are not careful, the memory may end up getting fragmented, consisting of large numbers of small, noncontiguous holes. It is then possible that no hole is large enough to satisfy a future request, even though there may be sufficient aggregate free space.

Best - Fit and Next - Fit Object Placement

We reduce fragmentation by controlling how the memory manager places new objects in the heap. It has been found empirically that a good strategy for minimizing fragmentation for real life programs is to allocate the requested memory in the smallest available hole that is large enough. This *best-fit* algorithm tends to spare the large holes to satisfy subsequent, larger requests. An alternative, called *first-fit*, where an object is placed in the first (lowest-address) hole in which it fits, takes less time to place objects, but has been found inferior to best-fit in overall performance.

To implement best-fit placement more efficiently, we can separate free space into *bins*, according to their sizes.Binning makes it easy to find the best-fit chunk.

M a n a g i n g and Coalescing Free Space

When an object is deallocated manually, the memory manager must make its chunk free, so it can be allocated again. In some circumstances, it may also be possible to combine *(coalesce)* that chunk with adjacent chunks of the heap, to form a larger chunk. There is an advantage to doing so, since we can always use a large chunk to do the work of small chunks of equal total size, but many small chunks cannot hold one large object, as the combined chunk could.

Automatic garbage collection can eliminate fragmentation altogether if it moves all the allocated objects to contiguous storage.

5. Manual Deallocation Requests

In manual memory management, where the programmer must explicitly arrange for the deallocation of data, as in C and C + + . Ideally, any storage that will no longer be accessed should be deleted.

Problems with Manual Deallocation

1. Memory leaks ‰

   Failing to delete data that

   cannot be referenced ‰

   Important in long running or

   nonstop programs „

2. Dangling

   pointer

   derefere

   ncing ‰

   Referenc

   ing

   deleted

   data „

Both are serious and hard to debug

V. Garbage Collection

1. Reclamation of chunks of storage holding objects that can no longer be accessed by a program „
2. 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 newpointers

„

Thus, any memory location can be (theoretically) accessed at any time and hencecannot be considered inaccessible.