6.1 STORAGE ORGANIZATION

In compiler writing, the target program being executed runs in its own logical address space. This logical address space is managed and organized with the close coordination of the compiler, the operating system, and the target machine. A run-time object program in logical address space is typically divided into code and data areas, as shown

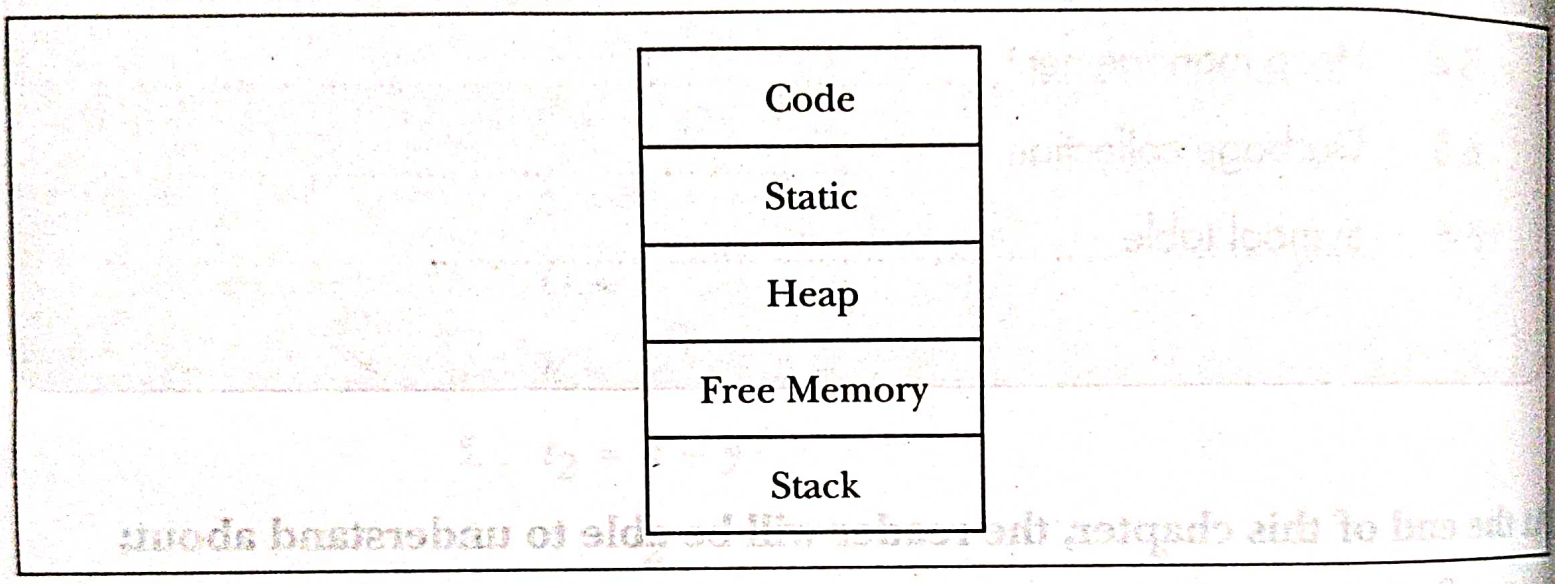


FIGURE 6.1: Sub-division of run-time memory into code and data areas

The addressing type used in the target machine, (such as Register addressing, byte addressing, etc.), influences the storage allocation and layout for the data objects. The allocation can be one of two types

1. **Aligned** - Here, the variables, such as integers etc are placed at addresses divisible by 4. But this can lead to improper or inefficient utilization of storage in cases where the required space is not exactly divisible by 4.
2. **Packed** - used when sufficient space is not available; variables are placed in contiguous blocks, end-to-end. It may be required to execute some additional instructions at run time to place the packed data so that it can be operated as if it were aligned properly.

The division of run-time memory as depicted above can be described as follows:

- *Code* – This area is used to place the executable target code, as the size of the generated code is fixed at compile time.

- *Static* – the size of some program data objects may also be known at compile time. These objects are placed in the Static area.
- *Stack and Heap* – these areas are placed at the two ends of the unused space, and grows towards each other. Stack area is used to store Activation Records generated during procedure calls, and is basically used for temporary or short-lived data, while the Heap area is used to allocate and deallocate arbitrary, dynamic chunks of storage.

Of these areas, the *Code* and *Static* areas are statically determined at compile time, which allows the addresses of the objects to be compiled into the target code itself; the *Stack* and *Heap* areas are dynamic, and helps in improved or maximized space utilization.

6.1.1 Static Versus Dynamic Storage Allocation

Static	Dynamic
Is also known as compile-time allocation	Is known as run-time allocation.
The decision is made based on the contents of the program; doesn't depend on the output.	Can be decided only when the program is running.
Ex: Code and Static area	Ex: Stack and Heap areas

Sometimes, Compilers allocate dynamic space using a combination of both the Stack and the Heap areas:

- **Stack** - used for local and short-term variables.
- **Heap** - used for non - local and long-term variables; memory is allocated wher the objects are created and deallocated when they are nullified.

Garbage allocation enables the reuse of space allocated to useless data elements b detecting such elements and returning their space explicitly.

6.2 STACK ALLOCATION OF SPACE

Stack area is used to allocate space to the local variables whenever a procedure is called; this space is popped off the stack when the procedure terminates. This arrangement has 2 advantages:

- It allows space to be **shared** by procedures whose durations are non-overlapping.
- Allows us to compile code for a procedure keeping the relative addresses of the non-local variables always the same.

2.1 Activation Trees

ctivation Trees are used to efficiently describe the nesting of procedure calls to make e stack allocation feasible. The nesting of procedure calls can be illustrated using e following example:

xample 6.1

onsider a sorting program which reads nine integers into an array 'a' and sorts them ing the recursive Quicksort algorithm.

```
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.. pp-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 );
main ( ) {
readArray ( );
a[0] = -9999;
a[10] = 9999;
quicksort ( 1, 9 );
}
```

FIGURE 6.2: A typical Quick sort program

The main function performs three tasks:

- call readArray
- set the end points
- call quicksort on the input array

One possible sequence of calls for the execution of the program can be depicted as follows

```
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 6.3: Possible flow of the program of fig. 6.2

Nesting of procedure activations which means that if an activation of a procedure p calls procedure q, then that activation of q must end before the activation of p can end. We have 3 cases here:

Case 1

The activation of q terminates normally; control returns just after the point of p at which the call to q was made.

Case 2

The activation of q aborts, and cannot continue; then p aborts simultaneously with q.

Case 3

The activation of q terminates and q cannot handle the exception condition. In this case, there are 2 cases:

Case 3.1

If p **can handle** the exception, the activation of q terminates and the activation of p continues, need not necessarily be from the calling point.

Case 3.2

If p cannot handle the exception, the activation of p terminates along with q, and the exception is passed to some open activation which can handle it.

An **activation tree** is used to represent the activations of procedures during the execution of the entire program.

In the tree

- **Node**- represents the activation of a procedure.
- **Root**- activation of the "main" procedure.
- **Children nodes**- activations of procedures called by the parent procedure.

The activations are shown in their order of occurrence from left to right.

6.2.2 Activation Records

Activation Record (AR) is a memory block used for information management for single execution of a procedure. AR is also called a *frame,* are used to store information about the status of a machine when a procedure call occurs. This status can include information such as the value of the program counters and the machine registers etc. ARs allow the control of flow of the program: when the control returns from the called procedure, the calling procedure is activated by restoring the values of the relevant registers and the program counter to the point immediately after the call. AR keeps track of flow of procedures.

A run-time stack, called the *control stack,* is used to manage procedure calls and returns. In this stack, the root of the activation trees is at the bottom, and the activation of the procedure where the control currently resides is at the top.

Also, from the top to the bottom, the activation of procedures is arranged in the reverse order from the currently active procedure to the root.

The ARs are typically drawn upside down, i.e., the bottom of the stack is shown higher than the top, and can be depicted as follows: (read from bottom to top)

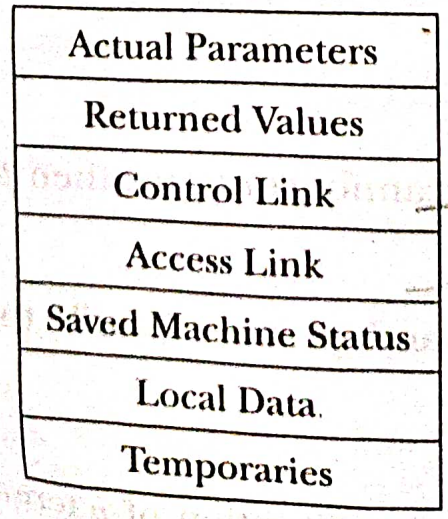


FIGURE 6.4: A general Activation Record

1. *Temporaries* hold temporary values, such as the result of a Mathematical calculation, a buffer value or so on.
2. *Local Data* belongs to the procedure where the control is currently located.
3. A *Saved Machine Status* provides information about the state of a machine just before the point where the procedure is called.
4. An *Access Link* is used to locate remotely available data. This field is optional.
5. *Control Link* points to the activation record of the procedure which called it, i.e., the caller. This field is optional. This link is also known as dynamic link.
6. *Return Value* holds any values returned by the called procedure. These values can also be placed in a register depending on the requirements.
7. The *Actual Parameters* used by the calling procedure are stored here.

AR size is determined when a procedure is called.

6.2.3 Calling Sequences

Calling Sequence is a code that allocates an AR on the stack and enters information into its fields. It is the sequence in which procedures are called by the main procedure as well as by the other procedures.

Return Sequence is a code used to restore the state of the machine so the calling procedure can continue its execution after the call.

The code in a calling sequence is divided between the calling procedure ("caller") and the called procedure (the "callee"). The portion of calling sequence assigned to the caller is generated separately each time a procedure is called; however, the portion assigned to the callee is generated only once. Hence for efficiency, it is better to put as much of the calling sequence code in the callee as possible.

Some important principles while designing calling sequences are:

1. For ease of communication and efficient data accessing, the values communicated between caller and callee are usually placed at the beginning of the callee's activation record.
2. The fixed-length items, such as control link, access link, machine status etc., are normally placed in the middle.
3. Variable sized data items are placed at the end of the AR. Such items include dynamic variables such as dynamic arrays, temporaries and temporary variables etc.
4. The positioning of the top-of-stack [TOP] pointer should be done such that it is easy to access not only the fixed-length records but also the variable length fields. Typically the TOP pointer is positioned to point to the end of the fixed-length fields in the AR.

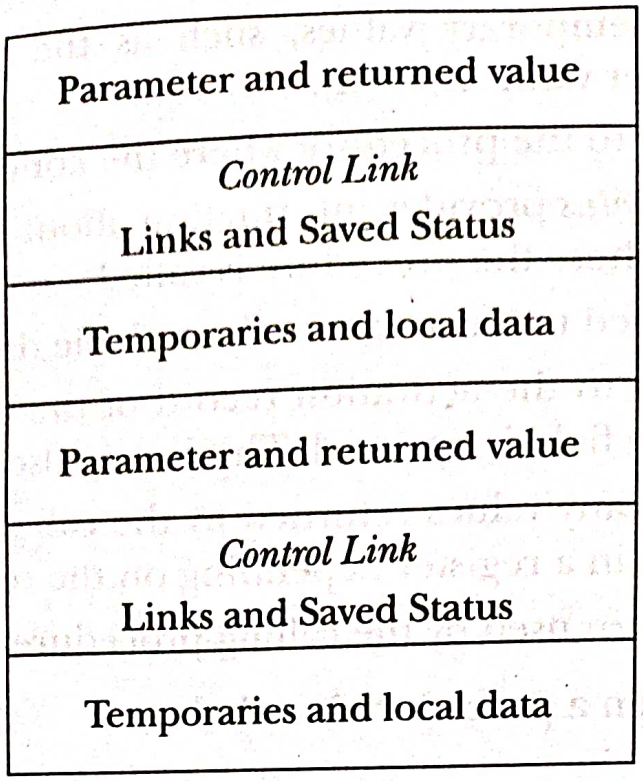


FIGURE 6.5: Division of tasks between caller and callee

A typical *calling sequence* can be described as follows:

- Evaluation of actual parameters by the caller.
- Caller stores the return address and the old value of the TOP pointer into the callee's AR.
- Register values and other status info are stored by the callee.
- Initialization of locally available data.

The corresponding *return sequence* is as follows:

- Return values are placed next to the parameters.
- Callee restores the value of the TOP pointer and other registers, and branches to the return address specified in the status field.
- Caller uses the return value placed in the registers.

6.2.4 Variable-Length Data on the Stack

Stacks are usually used to allocate space to fixed size variable, but can also be used for variables whose size is not known at compile time. Stack usage minimizes and avoids the cost of garbage collecting the unused space.

Space allocation for variable-length arrays is done as follows:

- The arrays being used by the particular procedure are placed after the AR of that procedure, at fixed offsets from the TOP pointer; a pointer to the beginning of each array is stored in the AR.

- If any procedure 'q' is called by this procedure, then the AR for the q is placed after the calling procedure's arrays, and the arrays of q are placed beyond it.

The stack is accessed using 2 pointers: *top* and *top_sp*; top points to actual top of the stack, and top_sp points to the end of the machine status field, and used to find local, fixed-length fields of the top AR.

6.3 ACCESS TO NON-LOCAL DATA ON THE STACK

Nonlocal data is a parameter or data value that is used or required within a procedure, but does not physically belongs to that procedure. There are various situations or conditions of access of nonlocal data. These include:

6.3.1 Data Access Without Nested Procedures

Generally, a variable has a scope of existence or use only within the function or procedure where it is defined. After being declared once globally, if the same variable is declared elsewhere, then this new definition overrides the global definition and is used as its value.

For non-nested procedures, variables are accessed as follows:

1. Global variables are declared statically as their values and locations are known or fixed at compile time.
2. Other variables (local) are declared locally at the top of the stack, using the top_sp pointer.

Static allocation allows parameter passing by reference which is much more efficient than passing by values.

6.3.2 Issues with Nested Procedures

1. Access is complicated when nested procedures are used.
2. The nesting does not specify the (relative) positions of the nested and the nesting ARs; this makes it difficult to determine the scope of reference of the variable being accessed.

6.3.3 A Language with Nested Procedure Declarations – ML

Properties of ML include:

- The variables declared in ML are unchangeable once they are initialized; this means that ML is a Functional Language.
- Variable declaration is done using the statement:

val <name> = <exp>

6.4 HEAP MANAGEMENT

As previously discussed, Heap is the unused memory space available for allocation dynamically. It is used for data that lives indefinitely, and is freed only explicitly. The existence of such data is independent of the procedure that created it.

6.4.1 The Memory Manager

Memory manager is used to keep account of the free space available in the heap area. *Functions* include:

- **Allocation:** this refers to the act of allocating or providing the requested amount of memory space to the application program. This memory is usually allocated in contiguous blocks. If the requested amount of space is not available, the memory manager either tries to use the free space from the virtual memory or informs the application program about the unavailability.
- **Deallocation:** this refers to returning the unused free space back to the pool of free space, for later reuse.

Properties of an efficient memory manager include:

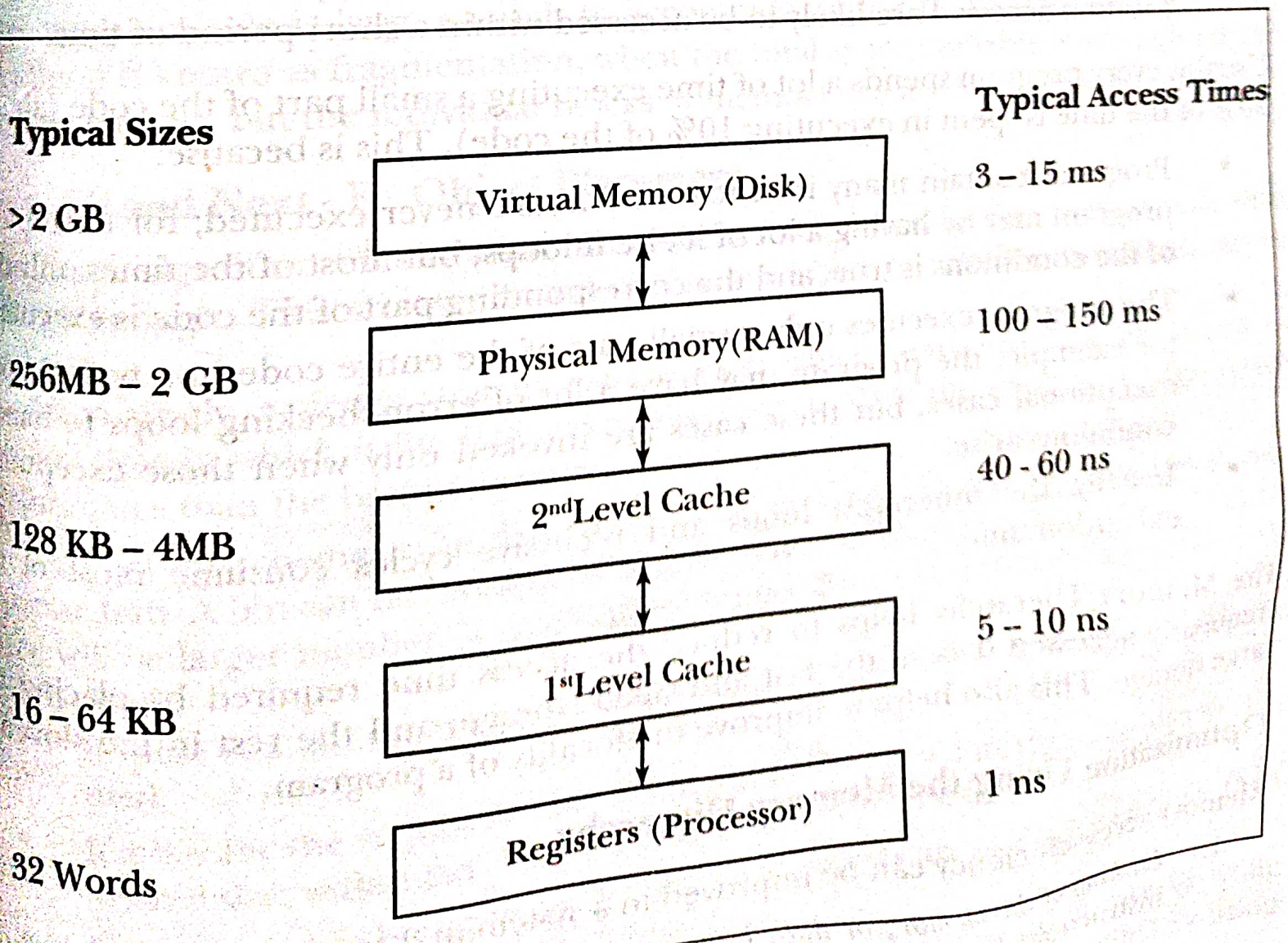
- **Space efficiency:** this means it should minimize the total heap space required by a program.

- **Program efficiency:** this requirements mandates that the memory manager make proper and good use of the memory system to increase the speed of performance of the programs. This is closely related to the concept of "locality" or placement of the object.
- **Low overhead:** the frequently performed memory allocation and deallocation processes should be performed as quickly and as efficiently as possible, reducing the execution time for memory operations.

4.2 The Memory Hierarchy of a Computer

any system, there is always a tradeoff between the memory space available and the ne required to access that space: the more the space available, the more time it takes access a record from that particular space. Likewise for smaller memory regions too; s impossible to get a storage that is both large and fast.

is problem is solved by designing the storage in the form of a hierarchy known as e *memory hierarchy*. This is a collection or arrangement of a series of storage elements, anged such that the smaller, faster ones are placed nearer to the processor than the ger slower ones. This can be understood by the figure below:



1.6 A SIMPLE CODE GENERATOR

compute science, *'Code Generation'* is defined as the process in which a compiler ode generator' converts some internal representation of source code into a form (ie achine code) that can be executed by a machine. The main job of code generator to convert an intermediate representation into a linear sequence of machine structions. The generator may try to use faster instructions, use fewer instructions, ploit available registers, and avoid redundant computations. Code generation (CG) ase includes:-

1. Instruction selection
2. Instruction scheduling
3. Register allocation

imple code generator (SCG) algorithm generates code for a single block. It tries to id generation of unnecessary loads and stores by considering each three-address truction and keeps track of what values are in which registers. Major issues during G lies in deciding the usage of registers. The following are the four principal ways of gisters usage:-

1. Some or all of the operands of an operation must be in registers in order to perform the operation
2. Registers are used to hold temporary results of a subexpression.
3. Registers are used to hold *'global values'* [global values are those values that are computed in one basic block and used in other block]
4. Registers are useful in run-time storage management.

me registers are reserved for global variables and some for stack management. ence registers available in the machine are used. The machine instructions are of e form:-

- LD reg,mem
- ST mem,reg
- OP reg, reg, reg

CG uses descriptors to keep track of register contents and address for names:-

1. Register Descriptor
   - It keeps track of the current content in each register.
   - Register descriptor is consulted whenever a new register is required.
   - Let us assume that, initially register descriptor shows that all registers a empty. Each register will hold the value of zero/ more names at any give time as the code generation for the block progresses.

2. Address Descriptor
   - It is used to keep track of the locations where the current value of the nam can be found at runtime.
   - The location can be a register, a stack location or a memory address.