U-IV

Runtime Environments.

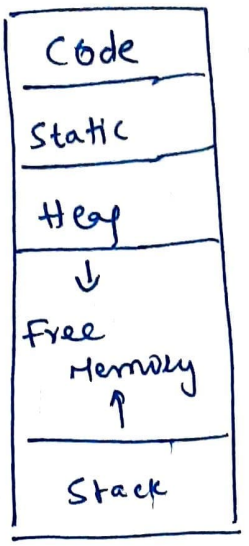
A Compiler must implement the abstractions in a source program. The Abstractions includes concepts like names, scopes, bindings, datatypes, operators, procedures, parameters, flow of control statements.

To implement all these abstraction Compiler will cooperate with O.S and System software by creating and managing Run-time-Environments in which target programs Execute. Run time Environments deals with issues like storage locations, mechanisms to access variables, linkage b/w procedures, etc.

Storage Organization.

~~From~~ compiler Writer Concides, executing program runs in its own logical address space. The logical address space is shared b/w compiler, O.S and target machine. The O.S maps this logical Address into physical Address (Actual Address).

A compiler like subdevides the memory as follows



⇒ Compiler places executable target code, in code area, which is low end of memory.

⇒ program data objects like constants, data generated by compiler, such as info to support garbage collection are placed in static area.

→ Stack and heap are dynamic their size can be changed as program executes. These areas grows towards each other as needed.

→ Stack stores normally procedures calls and returns, activation records.

→ Heap also contains data of procedures. Heap memory is reusable.

Stack Allocation Of Space.

Most compilers ~~use~~ stores procedures, functions, or methods on Run-time memory called as stack. Each time a procedure is called The local variables allocates memory on stack. And when procedure terminates The variables are popped out.

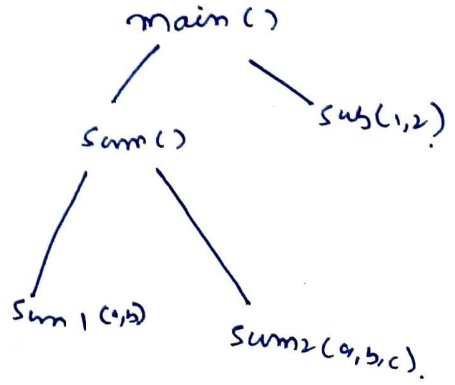
This method allows space to be shared by procedure calls, whose duration do not overlap in time.

Activation Trees.

Stack Allocation is used when activations of procedures can be nested. Activation of procedures during running of an entire program can be represented by a Tree, known as Activation Tree. Each None corresponds to one Activation. The root of Activation Tree is main, which initiates the execution of a program. If a node of procedure P has children, Then these corresponds to Activations of procedures called by P. Activations are shown as they are called, from left to right. Left child must finish before its right activation can begin.

Concider following exa code.

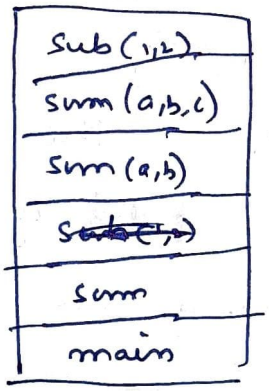
```
main()                  sub(a,b)
{                       {
  sum();                  :
  sub(1,2);             }
                        sum1(a,b)
sum()                   {
{                         ..
sum1(a,b)               }
sum2(a,b,c)             sum2(a,b,c)
}                       {
                          :
```



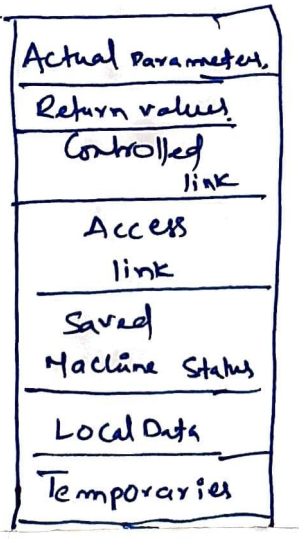
Activation Records.

procedure calls and returns are usually managed by a run time stack known as control stack. Each live activation has an Activation Record (or frame) on control stack. The root of the activation Tree (main) is at the bottom, And Records are added along the path on to stack. The Most recent Activation has record at the top of the stack.

The Activation records for previous example can be given as follows.



The contents of Activation Record vary from language to language. Various data that might appear on Activation Record are as follows.



→

→ Temporaries are the values which arises from evaluation

→ Local Data belonging to the procedure.

→ Saved machine status, Contains info about State of machine before the call procedure. This info includs return address, contents of registers, which must be restored when return occurs.

→ Access link, needed to locate the data in another Activation record.

→ Control link points to the Activation record of caller.

→ Return value, which will be returned by called procedure.

→ Actual parameters used by calling procedure.

Calling Sequences.

procedure calls are implemented by calling Sequences, which consists of code that allocates activation records on stack and enters info into fields.

A return sequence is smaller code, which restores the state of machine so calling procedure can continue its execution after the call.

The Code in Calling Sequence is devided b/w Calling procedure (the "caller") and the procedure it calls (the "callee"). If a procedure is called from n different points, Then portion of Calling sequence is assigned n times

The Calling sequence and its devision b/w Caller and Callee are as follows.

1. The Caller evaluates Actual parameters
2. The caller stores a return address
3. The Callee saves register values and Other status info^n.
4. The Callee initializes its local data and begins execution.

Corresponding return sequnce is

1. Callee places return values next to parameters in Activation Record.
2. Using info^n in machine-status field, Callee restores register and branches Control to return address of Caller.

3.

Access to Non Local Data On the Stack.

Local data of a procedure is declared within body of the procedure. Some times procedures can also access data defined in body of other procedures. This is known as non local data access. Access becomes more complicated when one procedure is declared inside other procedures; known as nested functions. And in functions which can take functions as arguments and returns functions as values.

Data Access Without Nested procedures.

In the family of C languages, all variables are defined within a single function (local) or outside any function (globally). Global variable v can have scope in all functions that follow declaration of v, but not in fnctions which has local declaration of v.

Variables declared within function can have scope within the function and in nested functions

→ MI → For languages that do not allow nested
d procedures, allocation of storage and Access
A to those variables are simple.

→ 1. Global variables are allocated in static storage, The location of these variables is fixed and known at compile time

2. local variables are allocated on stack, Each time new variable will be added on stack top. These can be Access by using top-sp pointer of stack.

A language with Nested procedure Declarations.

→ The C family languages does not support nested procedures.

→ The Algol 60, an ansestor of C had this Capability.

→ Pascal and ML also supports nested procedures.

→ We see nested procedure declarations and usage in ML.

→ ML is a functional language. Variables once declared and initialized can't be changed. Array elements can be changed by special function calls.

→ Variables are defined with values in ML as follows.

Val <name> = <expression>

→ Functions in ML are defined with following Syntax

fun <name> (<arguments>) = <body>

→ Function bodies are defined by Let statements as below.

let <list of definitions> in <statements> end.

The ~~boo~~ scope of each definition consists of all definitions up to in and all statements up to end. Functions can be nested. by For ex. The body of function f, can contain Let statement that includes definitions of another nested function g. Similarly g can have function definitions within its body, leading deep nesting of procedures.

Nesting Depth.

The procedure that are not nested within any other procedures like in C, has a nesting depth 1. If procedure p is defined at nesting depth i, then p has nesting depth i+1.

Concider following example code in ML for quick sort.

```
1) fun sort (input file, Output file) =
   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 ... exchang ...
         in
11)          ... a ... v ... partition ... quicksort
         end
   in
12)  ... a ... readArray ... quicksort ...
   end;
```

⇒ The Outer most function sort has nesting depth 1, which reads an array a of 9 integers and sorts them using the quicksort algorithm.

⇒ Defined within sort, The array a itself. at line 2. The Array's first argument says we have 11 elements, all are initialized to 0.

⇒ Within sort several functions are declared like read array, exchange and quick sort. All of thes have nesting depth 2. readarray and exchange can access array a.

⇒ The function partition is declared within quick sort, and has a depth 3. it can access a, and v which is defined in Quicksort.

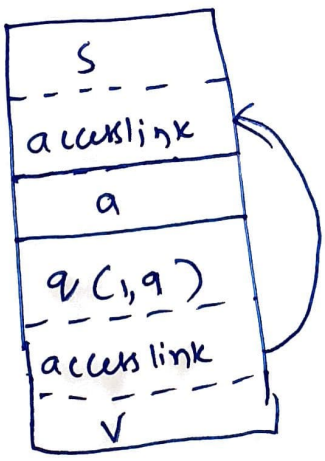
⇒ Quick sort access a, v and partition recursively.

⇒. Outer function sort Accesses a and calls the two procedures read Array and Quick sort.

Access links.

Implementation of normal static scope rule for nested function is obtained by adding a pointer called as access link to each Activation record. If a procedure p is nested immediately within procedure q, Then access link of p points to most recent activation of q. Nesting depth of q is exactly one less than nesting depth of p. Access links forms a chain from activation record at the top of stack to sequence of Activations at progressively lower nesting depths. Along the chain we have data and procedures which are accessible to currently executing procedure.

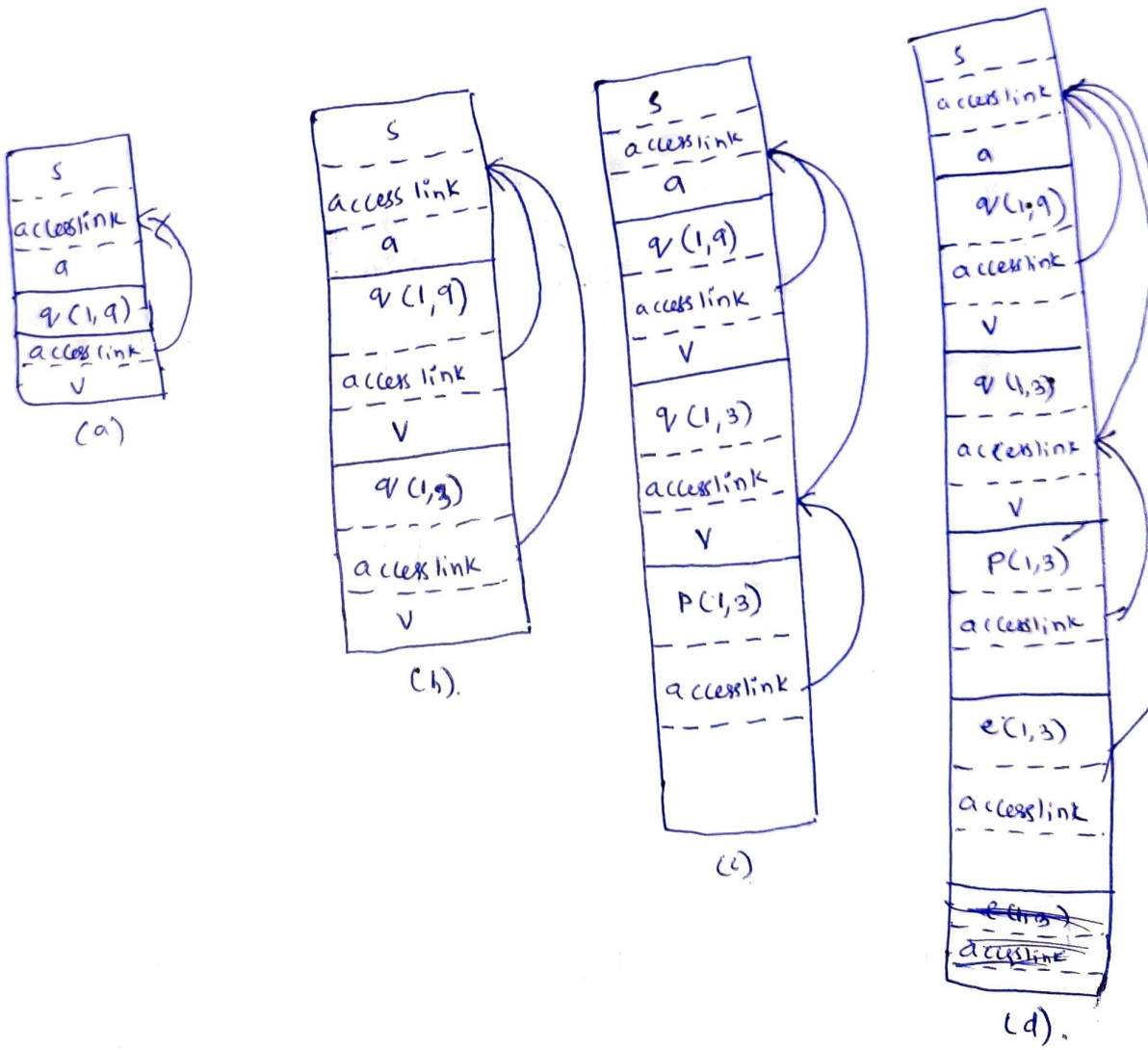
Following figure shows situation after Sort called read Array to load input into array a. The Access link from quicksort q(1,9) points to activation record of Sort, because it is most closely nested function surrounding quick sort.



Following is a recursive call to quicksort(1,3) followed by a call to partition, which calls to exchange.

Note that access links of quicksort(1,3) points to sort, because it is nested under sort.

In fig. d. The access link for exchange bypasses the activation records for quicksort and partition, since exchange is nested immediately within sort.



Heap Management.

The heap is the portion that stores the data which lives indepinitely or until the program terminates ~~and~~ or explicitly deletes it. While local variables typically becomes inaccessible When their procedures end, but many programming languages creates the data or variables in procedures which are independent of procedure life time. For ex, in both C++ and Java the objects created by new, or pointers of them can be passed ~~from other~~ one procedure to other procedure. The objects can continue to exist long after the procedure that created them has go
Such objects are stored on heap.

The Memory Manager.

The memory manager keeps track of free space in heap at all times. It has two functions

1. Allocation - When a program request memory for a variable, The memory manager produces a

Variable allocation - a chunk of contigeous heap memory of requested size. If not possible, it satisfies an allocation request using free space. If freespace is not available, it increase the heap storage space by requesting virtual memory. If no virtual memory exists a message is passed to application program.

Deallocation. - The memory manager returns deallocated space to pool of free space. So it can reuse the space to satisfy other allocation requests.

Memory manager must be prepared to service allocation and deallocation requests of any size.

Properties of memory manager.

Space Efficiency :- Memory manager should minimize total heap space needed by program. Space Efficiency is achieved by minimizing fragmentation

program Efficiency. - Memory manager should make good use of memory to allow program to run faster.

Low overhead.

Overhead is fraction of execution time spent performing allocation - and deallocation.

The memory Hirarchy of a Computer.

The efficeency of program is determined by, number instructions executed and how long it takes to execute each instruction. Time taken to execute ~~exe~~ instructions vary since time taken to access different parts of memory can vary.

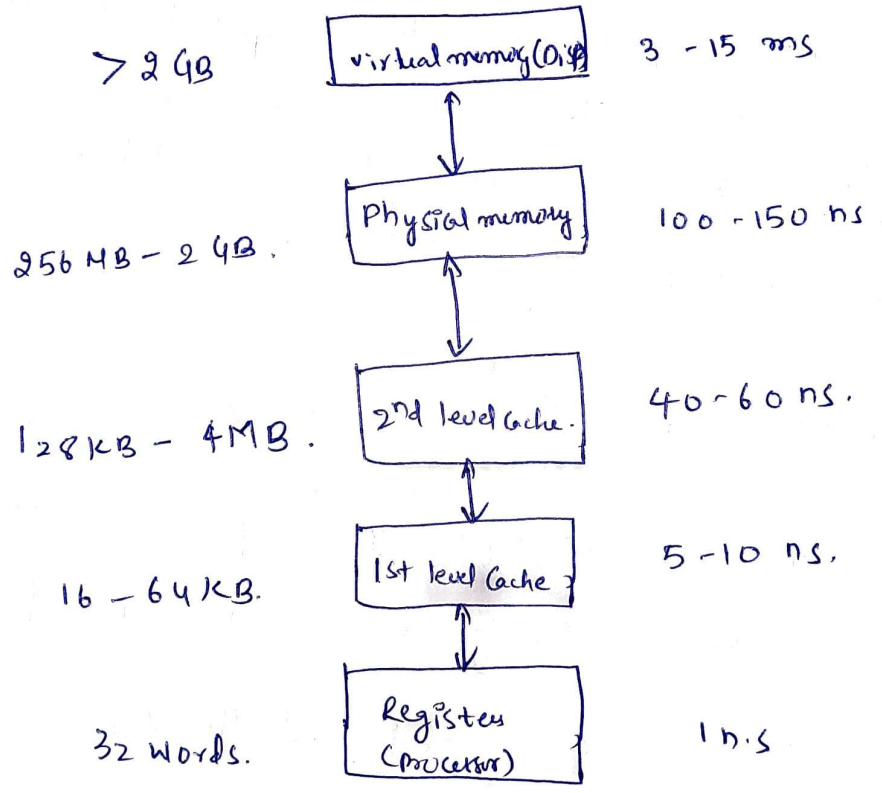
The variation in memory access times is due to limitations in hardware technology We can build small and fast storage, or large and slow storage, but not storage That is both large and fast. Modern Computers arrange their storage as a memory hirarchy as shown below.

→ processor has a small number of registers Whose contents are under software control.

→ Next is one or more levels of cache, usually made by static RAM, That are in kilo bytes to megabytes in size. The next in hirarchy are physical (main) memory, whose capacity is hundreds of megabytes or gigabytes of dynamic RAM. The top in hirarchy is Virtual memory, which is made by gigabytes of disks. For memory Access machine first looks in closest (lower-level) storage, if data is not there, looks in next higher level and so on.

Typical Sizes

Typical Access Times.



Data is transferred as blocks of Continues storage Between main memory and cache data is transferred in blocks known as cache lines, which are typically 32 to 256 bytes long.

Between virtual memory (disk) and main memory data is transferred in blocks known as pages typically between 4K and 64K bytes in size.

Locality in programs.

Programs spends most of the time executing a relatively small fraction of the code, and touching only a small fraction of the data.

temporary locality :- Program has temporal locality if the memory locations it accesses are likely to be accessed again within a short period of time.

Spatial locality :- program has spatial locality if memory locations close to the location accessed are likely also to be accessed within a short period of time.

Reducing Fragmentation.

At the beginning of program execution, The heap is one contigeous unit of free space. As programs allocates and deallocates memory This space is broken up into free and used chunks of memory. We refer to the free chunks of free space as holes. With each

With each allocation request, the memory manager must place requested chunk of memory into a large-enough hole. After allocating a hole, we need to split some hole, creating a yet smaller hole.

With each deallocation, freed chunks of memory are added back to the pool of free space. The contigeous free holes are compacted. If free holes are not contigeous fragments will be created. If holes are not contigeous, request can't be fullfilled.

Best-Fit and Next-Fit Object Placement.

Fragmentation can be controlled how the memory manager places new objects in the heap.

The best fit algorithm tends to spare the large holes to satisfy larger requests. An alternative is first-fit, where an object is placed in first hole in which it fits, takes less time to place objects.

Introduction to Garbage Collection.

Data that can not be referenced is known as garbage. Many high level programming offers automatic garbage collection, which deallocates unreachable data.

Garbage collection initially implemented in Lisp in 1958. Other languages that implements garbage collection include java, Perl, ML, Modula-3, Prolog and Smalltalk.

Garbage collection is reclamation of chunks of storage holding objects that can no longer be accessed by programs.

A user program, known as mutator modifies collection of objects in heap. The mutator creates objects by acquiring memory from memory manager. Mutator creates and drops references to these objects. Objects become garbage when mutator program can not reach them.

The Garbage Collection finds unreachable objects and reclaim their space by handling them to memory manager, which keeps track of free space.

Reachability.

The data which can be accessed directly by a program is known as root set. In Java root set of program consists of all static fields and all variables on stack. A program can reach any root set at any time.

Reachability becomes complex when program has been optimized by compiler. Optimizing compiler uses following things to enable garbage collector to find correct root set.

→ Compiler can restrict invocation of garbage collection to only certain code points in the program, when no "hidden" references exist.

→ Compiler can write out info that garbage collector can use to recover all the references.

→ The Compiler can assure that there is a reference to base address of reachable objects whenever garbage collector may be invoked.

The set of reachable objects changes as a program executes. The set grows as new objects get created and shrinks as objects become unreachable. Once the object unreachable it can not become reachable again.

Mutator does four basic operations to change the set of reachable objects.

1. Object Allocation. → These are performed by memory manager, which returns a reference to each newly created chunk of memory. This operation adds members to set of reachable objects.

2. Parameter passing and Return Values.

References to actual parameters are passed ~~from~~ to formal parameters, and results are returned back. Objects pointed by these references remain reachable.

3. Reference Assignments.

Assignments of the form $u = v$, where u and v are references. has two effects First u is a reference to the object referred by v. As long as u is reachable, the object also reachable. If reference in u is lost the

The Object becomes unreachable.

#Procedure returns.

As a procedure exists, The local Variables are popped off from the stack. The Objects pointed by these Variables becomes unreachable If unreachable Objects holds references to other unreachable Objects they too become unreachable

Reference Counting Garbage Collectors.

Reference counting garbage collector, identifies Garbage as Object changes from reachable to unreachable. When the Object is not referenced its count become zero. Every Object will have field for reference count.

Reference count can be maintained as follows.

1. Object allocation → The reference count for new Object is set to 1.

2. Parameter passing. → The reference count of each Object passed into a procedure is incremented.

3. Reference Assignments.

For statement $u = v$, where u and v are references. The reference count of object referred by v is incremented by 1, and count of old object referred by u decrements by 1.

4. Procedure returns.

As the procedure returns, (exits) objects referred by local variables have their count decremented.

5. Transitive loss of reachability.

Whenever reference count of an object becomes zero, we must decrement each object pointed to by a reference within the object.

Introduction to Trace Based Collection.

Trace-based collectors run periodically to find unreachable objects and reclaim their space. We run Trace-based collector whenever free space is exhausted or amount of free space drops below Threshold.

A Basic Mark-and-Sweep Collector.

Mark-and-sweep garbage collection is straight forward. It finds unreachable objects, and put them on list of free space.

Algorithm first visits and marks reachable objects and the sweeps entire heap to free up unreachable objects.

Algorithm - Mark-and-sweep garbage Collection.

Input → root set of objects, a heap and a free list, called Free, with unallocated chunks of heap. All chunks are marked with free/used status and size tags.

Method :- The algorithm uses several data structures like.

Free → holds objects known to be free.

Unscanned → is a list, which holds the objects that we have determined as reached but whose successors we have not considered.

The Unscanned list is empty initially.

reached-bit → Each object includes a bit to indicate whether it has been reached or not. Before algorithm begins reached-bit is set 0 for all objects.

In Line (1), we initialize the unscanned list by placing there all the objects referred by root-set. Their ~~reference~~ reached-bit is also set to 1.

Line (2) to (7) are a loop, in which we examine each object 'o' placed in unscanned list.

Line (8) to (11), is sweeping phase, reclaim the space of all objects remain unreached at the end of marking phase.

Algorithm

```
/* marking phase */
1) add each object referenced by root set to list
   Unscanned and set its reached-bit to 1.
2). While (unscanned ≠ ϕ)
    {
3)    remove some object O from unscanned;
4)    for (each object O' referenced in O)
        {
5)        if (O' is unreached i.e; its reached-bit is 0)
            {
6)            set the reached-bit of O' to 1;
7)            put O' in unscanned
            }
        }
    }
8).   Free = ϕ
9)    for (each chunk of memory O in heap)
      {
10)       if (O is unreached i.e; its reached-bit is 0)
                add O to free.
11).      else
              set the reached-bit of O to 0;
      }
```