6. LOADER

The loader is program which accepts the object program decks, prepares these programs for execution by the computer, initiates the execution.

In particular, the loader must perform four functions:

1. Allocate space in memory for the programs (Allocation)
2. Resolve symbolic references between object decks (Linking)
3. Adjust all address dependent locations, such as address constants, to correspond to the allocated space (Relocation)
4. Physically place the machine instruction and data into memory (Loading)

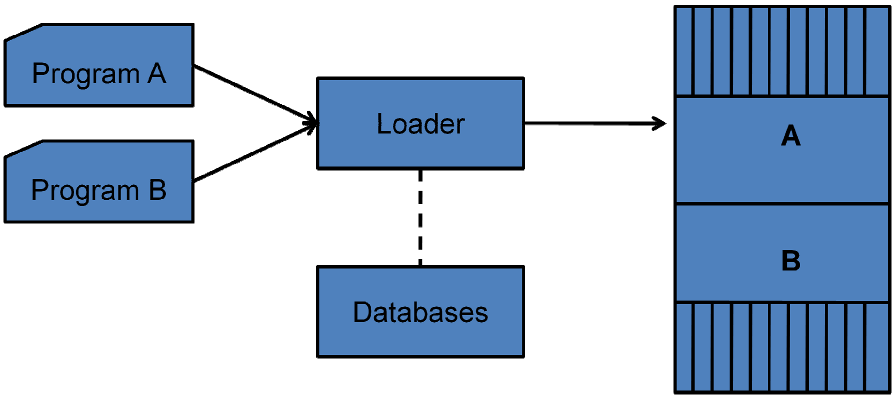

Figure 1: General Loading Scheme

6.1 LOADER SCHEMES

It is desirable to introduce the term segment, which is unit of information that is treated as an entity, be it program or data. Usually a segment corresponds to a source or object deck. It is possible to produce multiple program or data segments in a single source deck by means of the assembly CSECT (control section) pseudo-op, the FORTRAN COMMON statement, or PL/I EXTERNAL STATIC data attribute.

6.1.1 “Compile-and-Go” Loaders

One method of performing the loader functions is to have the assembler run in one part of memory and places the assembled machine instruction and data, as they are assembled, directly into their assigned memory locations. When the assembly is

completed, the assembler causes a transfer to the starting instruction of program. It is used by WATFOR FORTRAN compiler and other language processors.
Such a loading scheme commonly called "compile-and-go" or "assemble-and-go" which is shown in figure 2.

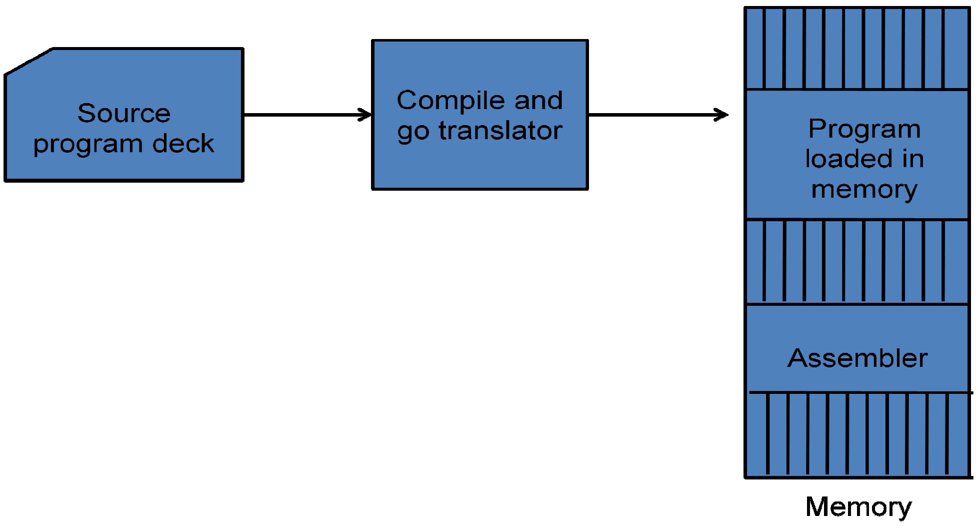

Figure 2: Compile-and-Go Loader

Advantages of Compile-and-Go Loader

- It is relatively easy to implement.
- The assembler simply places the code into core; the "loader" consists of one instruction that transfers to the starting instruction of the newly assembled program.

Disadvantages of Compile-and-Go Loader

However, there are several apparent disadvantages.

- First, a portion of memory is wasted because the core occupied by the assembler is unavailable to the object program.
- Second, it is necessary to retranslate the user's program deck every time it is run.

- Third, it is very difficult to handle multiple segments, especially if the sources are different languages e.g. one subroutine in assembly language and another subroutine in FORTRAN or PL/I.

6.1.2 General Loader Schemes

Outputting the instructions and data as they are assembled circumvents the problem of wasting core for the assembler. Such an output could be saved and loaded whenever the code was to be executed. The assembled programs could be loaded into the same area in core that assembler occupied. This output form, which may be on cards containing a coded form of the instructions, is called an object deck.

The use of an object deck as intermediate data to avoid one disadvantage of the preceding “compile-and-go” scheme requires the addition of a new program to the system, a loader. The loader accepts the assembled machine instructions, data, and other information present in the object format, and places machine instructions and data in core in an executable form.

Advantages of General Loader Scheme

- The loader is assumed to be smaller than the assembler, so that more memory is available to the user.
- Reassembly is no longer necessary to run the program at a later date.
- If all the source program translators produce compatible object program deck formats and use compatible linkage conventions, it is possible to write subroutines in several different languages since the object decks to be processed by the loader will all be in the same “language.”

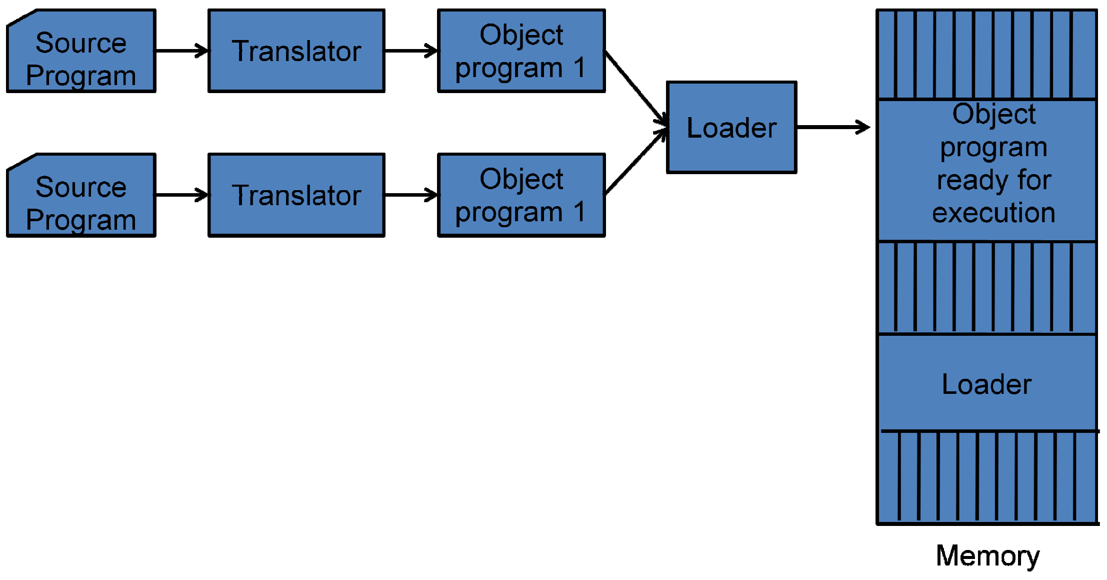

Figure 3: General Loader Scheme

6.1.3 Absolute Loaders

The simplest type of loaders scheme, which fits the general model is called as an absolute loader. In this scheme the assembler outputs the machine language translation of the source program in almost the same form as in the “assemble-and-go” scheme, except that the data is punched on cards instead of being placed directly in memory. The loader in turn simply accepts the machine language text and places it into core at the location prescribed by the assembler. This scheme makes more cores available to the user since the assembler is not in memory at load time.

Advantage of Absolute Loader

- Absolute loaders are simple to implement

Disadvantages of Absolute Loader

- First, the programmer must satisfy to the assembler the address in core where the program is to be loaded.
- Furthermore, if there are multiple subroutines, the programmer must remember the address of each and use that absolute address explicitly in his other subroutines to perform subroutines linkage.

- The programmer must be careful not to assign two subroutines to the same or overlapping locations.

```
MAIN Program                        | Location   Instruction
MAIN      START     100             |
          BALR      12,0            | 100        BALR     12,0
          USING     MAIN+2, 12      | 102        .
          .                         | .          .
          .                         | .          .
          L         15, ASQRT       | 120        L        15, 142(0, 12)
          BALR      14,15           | 124        BALR     14, 15
          .                         | .          .
          .                         | .          .
ASQRT     DC        F'400'          | 244        F'400'
          END                       | 248

SQRT SUBROUTINE                     | Location   Instruction
SQRT      START     400             | 400
          USING     *, 15           | .          .
          .                         | .          .
          .                         | .          .
          BR        14              | 476        BCR      15, 14
```

Figure 4: Absolute Loader Example

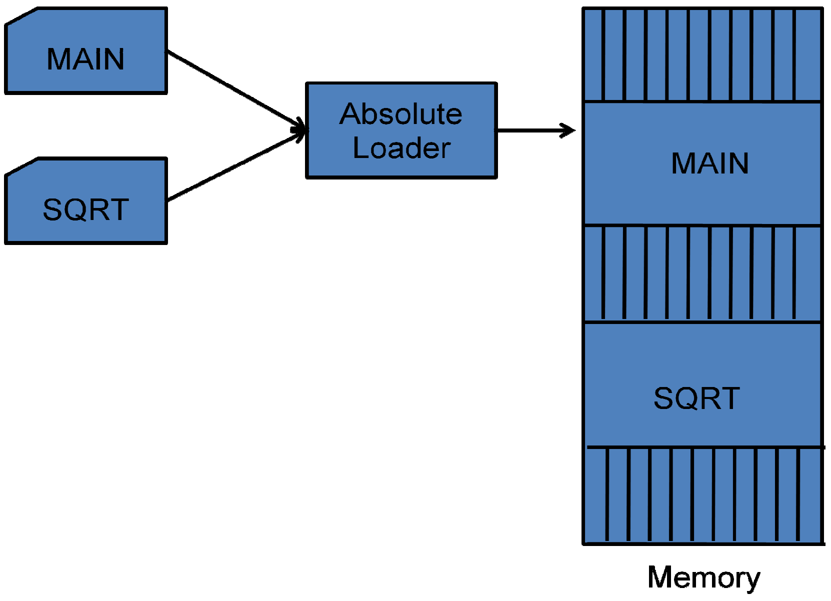

Figure 5: Absolute Loader

The MAIN program is assigned to locations 100-247 and the SQRT subroutine is assigned location 400-477. If changes were made to MAIN that increased its length to more than 300 bytes, the end of MAIN would overlap the start of SQRT. It would then be assign SQRT to a new location by changing its START pseudo-op card and reassembling it. Furthermore, it would also be necessary to modify all other subroutines are being used; this manual “shuffling” can get very complex, tedious, and wasteful of core.

The loader functions are as follows:

1. Allocation-by programmer
2. Linking-by programmer
3. Relocation-by assembler
4. Loading-by loader

6.1.4 Relocating Loaders

To avoid possible reassembling of all subroutines when a single subroutine is changed, and to perform the tasks of allocation and linking for the programmer, the general class of relocating loaders was introduced. An example of a relocating loader is that of the

Binary symbolic routine (BSS) loader such as was used in the IBM 7049, IBM 1130, GE 635, and UNIVAC 1108.

The BSS loader allows many procedures segments, yet only one data segment. The assembler assembles each procedures segment independently and passes on to the loader the text and information as to relocation and intersegment references.

The output of a relocating assembler using a BSS scheme is the object program and information about all other programs it references. In addition, there is information as to location in this program that need to be changed if it is to be loaded in an arbitrary place in core, i.e. the locations which are dependent on the core allocation.

For each source program the assembler outputs a text prefixed by a transfer vector that consists of address containing names of the subroutines referenced by the source program. For example, if SQRT was referenced and was the first subroutine called, the first location in the transfer vector would contain the symbolic name SQRT. The statement calling SQRT would be translated into a transfer instruction indicating a branch to the location of the transfer vector associated with SQRT.

The assembler would also provide the loader with additional information, such as the length of the entire program and the length of transfer vector portion. After loading the text and transfer vector into core, the loader would load each subroutine identified in the transfer vector. It would then place a transfer instruction to the corresponding subroutine in each entry in the transfer vector. Thus, the execution of the call SQRT statement would result in a branch to the first location in the transfer vector, which would contain a transfer instruction to the location of SQRT.

Source Program			**Program length = 48 bytes** **Transfer vector = 8 bytes** **Rel. addr.**	**Relocation**	**Object code**	
MAIN	START					
	EXTRN	SQRT	0	00	‘SQRT’	
	EXTRN	ERR	4	00	‘ERRb’	
	ST	14, SAVE	8	01	ST	14, 36
	L	1, =F’9’	12	01	L	1, 40
	BAL	14, SQRT	16	01	BAL	14, 0
	C	1, =F’3’	20	01	C	1, 44
	BNE	ERR	24	01	BC	7, 4
	L	14, SAVE	28	01	L	14, 36
	BR	14	32	0	BCR	15, 14
SAVE	DS	F	34	0	(Skipped for alignment)	
	END		36	00	(Temporary location)	
			40	00	9	
			44	00	3	

Figure 6: Relocation Loader Example

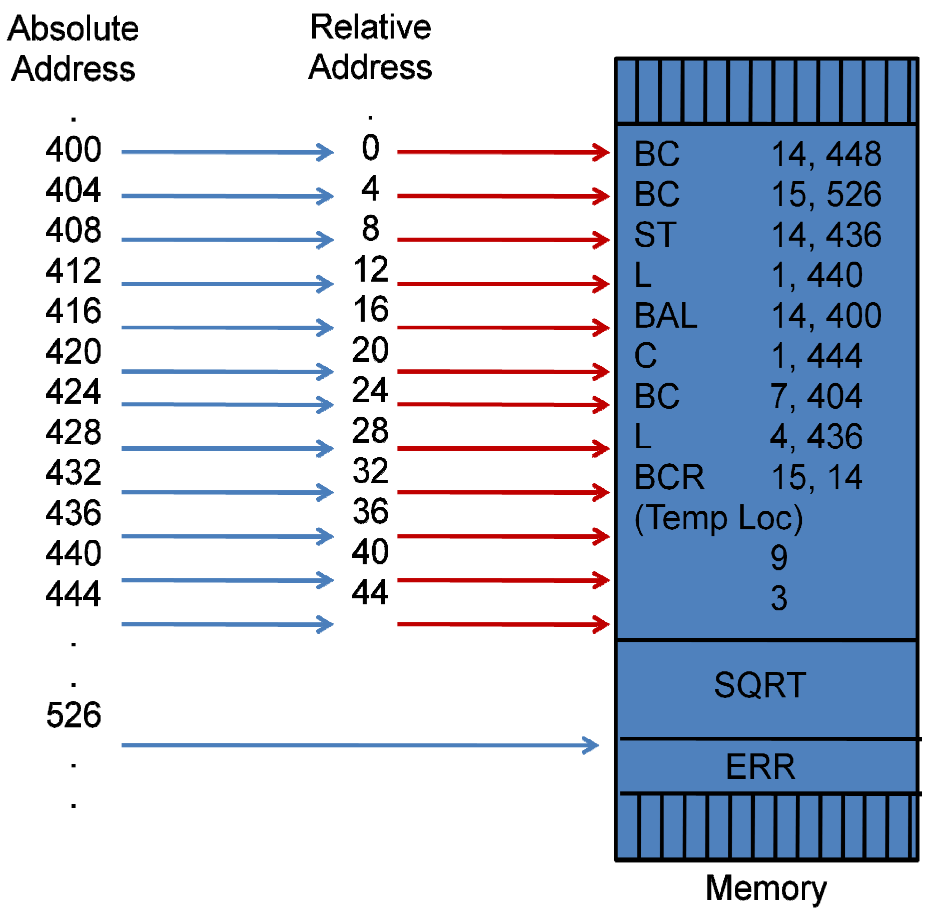

Figure 7: Loading of program in memory

The four functions of the loader were all performed automatically by the BSS loader. It should be noted that

- the relocation bits are used to solve the problem of relocation;
- the transfer vector is used to solve the problem of linking; and
- the program length information is used to solve the problem of allocation.

Disadvantages:

There are several disadvantages to the BSS loader scheme.

- First, the transfer vector linkage is only useful for transfers, and well-suited for loading or storing external data.

- Second, the transfer vector increases the size of the object program in memory.
- Finally, the BSS loader, as described, processes procedure segments but does not facilitate access to data segments that can be shared.

5.1.6 Direct-Linking Loaders

A direct linking loader is general relocatable loader, and is perhaps the most popular loading scheme presently used. The direct linking loader has advantage of the allowing the programmer multiple procedures segments and multiple data segments and of giving him complete freedom in referencing data or instructions contained in other segments. This provides flexible intersegment referencing and accessing ability, while at the same time allowing independent translations of programs.

The assembler must give the loader the following information with each procedure or data segment:

1. The length of segment
2. A list of all symbols in the segment that may be referenced by other segments and their relative within the segment
3. A list of all symbol not defined in segment but referenced in the segment
4. Information as to where address constants are located in the segment and a description of how to revise their values.
5. The machine code translation of the source program and the relative address assigned.

A simple example using a direct linking loading scheme is presented. A source program is translated by an assembler to produce the object depicted in the right column. Mnemonic machine codes have again been used.

Program				Translation		
Card No.				Rel. Loc.		
1	JOHN	START				
2		ENTRY	RESULT			
3		EXTRN	SUM			
4		BALR	12, 0	0	BALR	12, 0
5		USING	*, 10			
6		ST	14, SAVE	2	ST	14, 54(0, 12)
7		L	1, POINTER	6	L	1, 46(0, 12)
8		L	15, ASUM	10	L	15, 58(0, 12)
9		BALR	14, 15	14	BALR	14, 15
10		ST	1, RESULT	16	ST	1, 50(0, 12)
11		L	14, SAVE	20	L	14, 54(0, 12)
12		BR	14	24	BCR	15, 14
				26	----	
13	TABLE	DC	F'1, 7, 9, 10, 3'	28	1	
				32	7	
				36	9	
				40	10	
				44	3	
14	POINTER	DC	A(TABLE)	48	28	
15	RESULT	DS	F	52	----	
16	SAVE	DS	F	56	----	
17	ASUM	DC	A(SUM)	60	?	
18		END		64		

Figure 8: Assembly source program and its translation

The assembler produces four types of cards

1. ESD (External Symbol Directory)
2. TXT (Text Card)
3. RLD (Relocation and Linkage Directory)
4. END

1. ESD (External Symbol Directory)

– This card contain information about
  - all symbols that are defined in this program but that may be referenced elsewhere and
  - all symbols referenced in this program but defined elsewhere

– 'Type' mnemonic in ESD card can be
  - SD (Segment Definition)
  - LD (Local Definition)
  - ER (External Reference)

Reference no.	Symbol	Type	Relative location	Length
1	JOHN	SD	0	64
2	RESULT	LD	52	----
3	SUM	ER	----	----

2. TXT card

– This card contain actual object code translated version of the source program.

Reference number	**Relative location**	**Object code**	
4	0	BALR	12, 0
6	2	ST	14, 54(0, 12)
7	6	L	1, 46(0, 12)
8	10	L	15, 58(0, 12)

9	14	BALR	14, 15
10	16	ST	1, 50(0, 12)
11	20	L	14, 54(0, 12)
12	24	BCR	15, 14
13	28	1	
13	32	7	
13	36	9	
13	40	10	
13	44	3	
14	48	28	
17	60	0	

3. **RLD card**
   - This card contain following information
     - The location of each constant that needs to be changed due to relocation
     - By what it has to be changed
     - The operation to be performed

Reference no.	**Symbol**	**Flag**	**Length**	**Relative location**
14	JOHN	+	4	48
17	SUM	+	4	60

5.1.7 Other Loader Schemes – Binders, linking loaders, overlays, dynamic binders

One disadvantage of the direct linking loader, as presented, is that it is necessary to allocate, relocate, link, and load all of the subroutines each time in order to execute a program. Since there may be tens and often hundreds of subroutines involved, especially when we include utility routines such as SQRT, etc., this loading process can be extremely time-consuming. Furthermore, even though the loader program may be smaller than the assembler, it does absorb a considerable amount of space. These problems can be solved by dividing the loading process into two separate programs: a binder and a module loader.

A binder is a program that performs the same functions as the direct-linking loader in "binding" subroutines together, but rather than placing the relocated and linked text directly into memory, it outputs the text as a file or card deck. This output file is a format ready to be loaded and is typically called load module. The module loader merely has to physically load the module into core. The binder essentially performs the functions of allocation, relocation, and linking; the module loader merely performs the function of loading.

There are two classes of binders. The simplest type produces a load module that looks very much like a single absolute loader deck. This means that the specific core allocation of the program is performed at the time that the subroutines are bound together. Since this kind of module looks like an actual "snapshot" or "image" of section of core, it is called a core image module and the corresponding binder is called a core image builder. A more sophisticated binder, called a linkage editor, can keep track of the relocation information so that the resulting load module, as an ensemble, can be further relocated and thereby loaded anywhere in core. In this case the module loader must perform additional allocation and relocation as well as loading, but it does not have to worry about the complex problems of linking.

In both cases, a program that is to be used repeatedly need only be bound once and then can be loaded whenever required. The core image builder binder is relatively simple and fast. The linking editor binder is somewhat more complex but allows a more flexible allocation and loading scheme.

DYNAMIC LOADING

Usually subroutines of program are needed at different times: for example, pass 1 and pass 2 of an assembler are mutually exclusive. By explicitly recognizing which subroutines call other subroutines it is possible to produce an overlay structure that identifies mutually exclusive subroutines.

In order for the overlay structure to work it is necessary for the module loader to load the various procedures as they are needed. We will not go into their specific details, but there are many binders capable of processing and allocating an overlay structure. The portion of the loader that actually intercepts the "call" and loads the necessary procedure is called the overlay supervisor or simply the flipper. This overlay scheme is called dynamic loading or load-on-call.

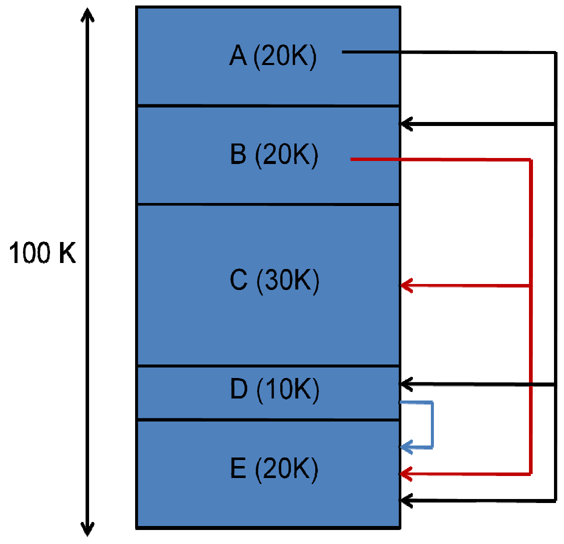

Figure 9: Subroutine calls between the procedures

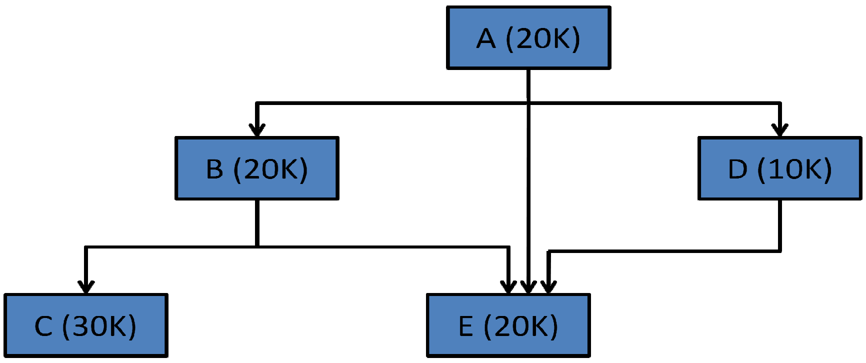

Figure 10: Overlay Structure

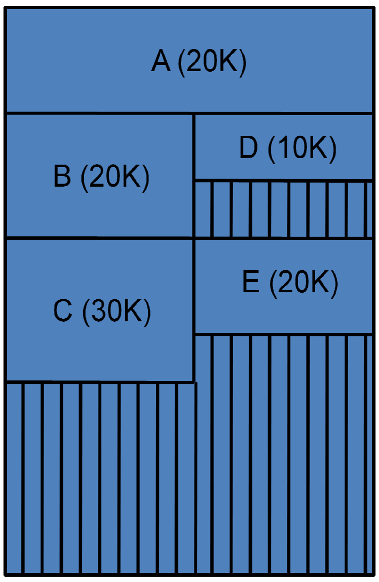

Figure 11: Possible storage assignment of each procedure

DYNAMIC LINKING

A major disadvantage of all of the previous loading schemes is that if subroutines is referenced but never executed, the loader would still incur the overhead of linking the subroutine.

Furthermore, all of these schemes require the programmer to explicitly name all procedures that might be called. It is not possible to write programs as follows:

```
    .
    READ        SUBNAME, ARGUMENT
```

```
        ANSWER = SUBNAME (ARGUMENT)
        PRINT       ANSWER
```

Where the name of the subroutine is an input parameter, SUBNAME, just like the other data.

This is a mechanism by which loading and linking of external references are postponed until execution time. That is, the assembler produces text, binding and relocation information from a source language deck. The loader loads only the main program. If the main program should execute a transfer instruction to an external address, or should reference an external variable, the loader is called. Only then is the segment containing the external reference loaded.

An advantage here is that no overhead is incurred unless the procedure to be called or referenced is actually used. A further advantage is that the system can be dynamically reconfigured. The major drawback to using this type of loading scheme is the considerable overhead and complexity incurred, due to the fact that we have postponed most of the binding process until execution time.