RUN-TIME STORAGE MANAGEMENT

• Information needed during an execution of a procedure is kept in a block of storage called an activation record, which includes storage for names local to the procedure. The two standard storage allocation strategies are:

1. Static allocation 2. Stack allocation

• In static allocation, the position of an activation record in memory is fixed at compile time.
• In stack allocation, a new activation record is pushed onto the stack for each execution of a procedure. The record is popped when the activationends.

• The following three-address statements are associated with the run-time allocation and de allocation of activation records:

1. Call,
2. Return,
3. Halt, and
4. Action, a placeholder for other statements.

• We assume that the run-time memory is divided into areas for:

1. Code
2. Static data
3. Stack

Static allocation

- In this allocation scheme, the compilation data is bound to a fixed location in the memory and it does not change when the program executes.
- As the memory requirement and storage locations are known in advance, runtime support package for memory allocation and de-allocation is not required.
- In a static storage-allocation strategy, it is necessary to be able to decide at compile time exactly where each data object will reside at run time. In order to make such a decision, at least two criteria must be met:
    1. The size of each object must be known at compile time.
    2. Only one occurrence of each object is allowable at a given moment during program execution.
- Because of the first criterion, variable-length strings are disallowed, since their length cannot be established at compile time. Similarly dynamic arrays are disallowed, since their bounds are not known at compile time and hence the size of the data object is unknown.
- Because of the second criterion, nested procedures are not possible in a static storage-allocation scheme. This is the case because it is not known at compile time which or how many nested procedures, and hence their local variables, will be active at execution time.

Implementation of call statement:

The codes needed to implement static allocation are as follows: MOV #here + 20, callee.static_area /*It saves return address*/
GOTO callee.code_area /*It transfers control to the target code for the called procedure */
where,

callee.static_area - Address of the activation record
callee.code_area - Address of the first instruction for called procedure
#here + 20 - Literal return address which is the address of the instruction following GOTO.

Implementation of return statement:

A return from procedure callee is implemented by : GOTO *callee.static_area
This transfers control to the address saved at the beginning of the activation record.

Implementation of action statement:

The instruction ACTION is used to implement action statement.

Implementation of halt statement:

The statement HALT is the final instruction that returns control to the operating system.

Dynamic Allocation

- The allocation can be varied during the execution
- It makes the use of recursive function.
- **In a dynamic storage-allocation strategy**, the data area requirements for a program are not known entirely at compilation time.
- In particular, the two criteria that were given in the previous section as necessary for static storage allocation do not apply for a dynamic storage-allocation scheme.
- The size and number of each object need not be known at compile time; however, they must be known at run time when a block is entered.
- Similarly more than one occurrence of a data object is allowed, provided that each new occurrence is initiated at run time when a block is entered.

Stack Allocation

- Procedure calls and their activations are managed by means of stack memory allocation.
- It works in last-in-first-out (LIFO) method and this allocation strategy is very useful for recursive procedure calls.

Static allocation can become stack allocation by using relative addresses for storage in activation records. In stack allocation, the position of activation record is stored in register so words in activation records can be accessed as offsets from the value in this register.

The codes needed to implement stack allocation are as follows:

Initialization of stack:

```
MOV #stackstart , SP /*
initializes stack */ Code for
the first procedure
HALT /* terminate execution */
```

Implementation of Call statement:

```
ADD #caller.recordsize, SP        /*
increment stack pointer */ MOV #here + 16,
*SP/*Save return address */
GOTO
callee.c
ode_ar
ea
where,
caller.recordsize           - size of the activation record
#here + 16 - address of the instruction following the GOTO
```

Implementation of Return statement:

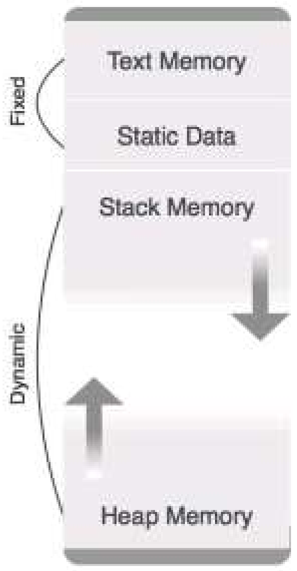
```
GOTO *0 ( SP )             /*return to the caller */
SUB #caller.recordsize, SP         /* decrement SP and restore to previous value */
```

Runtime environment manages runtime memory requirements for the following entities:

- **Code:** It is known as the text part of a program that does not change at runtime. Its memory requirements are known at the compile time.
- **Procedures:** Their text part is static but they are called in a random manner. That is why, stack storage is used to manage procedure calls and activations.
- **Variables:** Variables are known at the runtime only, unless they are global or constant. Heap memory allocation scheme is used for managing allocation and de-allocation of memory for variables in runtime.

Heap Allocation

- Variables local to a procedure are allocated and de-allocated only at runtime.
- Heap allocation is used to dynamically allocate memory to the variables and claim it back when the variables are no more required.
- Except statically allocated memory area, both stack and heap memory can grow and shrink dynamically and unexpectedly.
- Therefore, they cannot be provided with a fixed amount of memory in the system.



- As shown in the image above, the text part of the code is allocated a fixed amount of memory.
- Stack and heap memory are arranged at the extremes of total memory allocated to the program. Both shrink and grow against each other.