**Unit -4 Runtime Environments**

* **Runtime Environment** :The structure of the target computer’s **registers and memory** that serves to manage memory and maintain the information needed **to guide the execution process**
* **Or**
* **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.
* **Three kinds of runtime environments**

(1) **Fully static** environment; FORTRAN77

(2) **Stack-Based** environment; C, C++,Pascal,Ada

(3) **Fully dynamic** environment; LISP

* Main issues will be discussed in more detail in the chapter:
  + For each environment, the language features and their properties

(1) **Scoping and allocation** issues;

(2) Nature of **procedure calls**;

(3) **Parameter passing** mechanisms

* We focus here on the general structure of the environment.
* **Note:**
  + The compiler can only maintain an environment indirectly .It must generate code to perform the necessary maintenance operations during program execution.
  + -An Interpreter on other hand can maintain the environment directly within its own datastructures.

**4.1Memory Organization During Program Execution**

The memory of a typical computer is divided into :

A register area;

Addressable Random access memory (RAM)

(RAM is further divided into):

A code area;

A data area.

The code area is fixed prior to execution, and can be visualized as follows:

|  |  |
| --- | --- |
| Entry pointer to procedure1→ | Code for procedure 1 |
| Entry pointer to procedure2→ | Code for procedure 2 |
|  | … |
| Entry pointer to procedure n→ | Code for procedure n |

In particular, the entry point for each procedure and function is known at compile time.

We focus more on dealing with nonfixed, or dynamic, data allocation

* The global and/or static data of a program can be **fixed in memory prior to execution** ,Data are allocated separately in a fixed area in a similar fashion to the code
  + - In Fortran77, all data are in this class;
    - In Pascal, global variables are in this class;
    - In C, the external and static variables are in this class
    - The **constants** are usually allocated memory **in the global/static area**
  + Constant declarations of C and Pascal;
  + Literal values used in the code,
    - such as “Hello%D\n” and Integer value 12345:
    - Printf(“Hello %d\n”,12345);
* The memory area used for **dynamic data** can be organized in many different ways
  + Typically, this memory can be divided into a stack area and a heap area;
    - **A stack area** used for data whose allocation occurs in LIFO fashion;
    - **A heap area** used for dynamic allocation occurs not in LIFO fashion.
    - Generally, the architecture of the target machine includes a processor stack for procedure calls and returns.
  + Sometimes, a compiler will have to arrange for the **explicit allocation of the processor stack** in an appropriate place in memory.

**The general organization of runtime storage:**

|  |
| --- |
| Code area |
| Global/static area |
| Stack |
| ↓  Free space  ↑ |
| Heap |

Where, the arrows indicate the direction of growth of the stack and heap.**Procedure activation record (An important unit of memory allocation)** Memory allocated for the local data of a procedure or function. An activation record must contains the following sections:

|  |
| --- |
| Space for arguments  ( parameters ) |
| Space for bookkeeping information, including return address |
| Space for local data |
| Space for local temporaries |

Note: this picture only illustrates the general organization of an activation record.



Activation recored ( taken from Ravi Sethi)

* Some parts of an activation record have the **same size for all procedures**
  + Space for bookkeeping information
* Other parts of an activation record may **remain fixed for each individual procedure**
  + Space for arguments and local data
* Some parts of activation record may be **allocated automatically** on procedure calls:
  + Storing the return address
* Other parts of activation record may need to be **allocated explicitly** by instructions generated by the compiler:
  + Local temporary space
* **Depending on the language**, activation records may be allocated in different areas:
  + Fortran77 in the static area;
  + C and Pascal in the stack area; referred to as stack frames
  + LISP in the heap area.
* **Processor registers are also part of the structure of the runtime environment**
  + Registers may be used to store temporaries, local variables, or even global variables;
  + In newer RISC processor, keep entire static area and whole activation records in regiesters;
  + Special-purpose registers to keep track of execution
    - PC program counter;
    - SP stack pointer;
    - FP frame pointer;
    - AP argument pointer
* **The sequence of operations when calling the functions**: ***calling sequence*** 
  + The allocation of memory for the activation record;
  + The computation and storing of the arguments;
  + The storing and setting of necessary registers to affect the call
* **The additional operations when a procedure or function returns: *return sequence (VS call****)*
  + The placing of the return value where the caller can access it;
  + The readjustment of registers;
  + The possible releasing for activation record memory
* **The important aspects of the design of the calling sequence:**

(1) How to **divide the calling sequence** operations between the caller and callee

* + - At a minimum, the caller is responsible for computing the arguments and placing them in locations where they may be found by the callee

(2) To what extent to **rely on processor support for calls** rather that generating explicit code for each step of the calling sequence

**4.2 Fully Static Runtime Environments (FORTAN77)**

**The entire program memory can be visualized as follows:**

|  |  |
| --- | --- |
| **Code for main procedure** | **Code area** |
| **Code for procedure 1** |
| **…** |
| **Code for procedure n** |
|  | **Data area** |
| **Global data area** |
| **Activation record of main procedure** |
| **Activation record of procedure 1** |
| **…** |
| **Activation record of procedure n** |

All data are static, remaining fixed in memory for the duration of program execution.

For a language, such as FORTRAN77, no pointer or dynamic allocation, no recursive procedure calling

* The global variables and all variables are allocated statically.
* Each procedure has only a single activation record.
* All variables, whether local or global, can be accessed directly via fixed address.

Relative little overhead in terms of bookkeeping information to retain in each activation record;

And no extra information about the environment needs to be kept in an activation record;

The calling sequence is simple.

Each argument is computed and stored into its appropriate parameter location in the activation record of callee;

The return address is saved, and jump to the beginning of the code of the callee;

On return, a simple jump is made to the return address.

Example: A FORTRAN77 sample program

PROGRAM TEST

COMMON MAXSIZE

INTEGER MAXSIZE

REAL TABLE(10),TEMP

MAXSIZE = 10

READ \*, TABLE(1),TABLE(2),TABLE(3)

CALL QUADMEAN(TABLE,3,TEMP)

PRINT \*, TEMP

END

SUBROUTINE QUADMEAN(A, SIZE,QMEAN)

COMMON MAXSIZE

INTEGER MAXSIZE,SIZE

REAL A(SIZE),QMEAN,TEMP

INTEGER K

TEMP=0.0

IF ((SIZE .GT. MAXSIZE) .OR. (SIZE .LT. 1) GOTO 99

DO 10 K=1,SIZE

TEMP=TEMP+A(K)\*A(K)

10 CONTINUE

99 QMEAN = SQRT(TEMP/SIZE)

RETURN

END

**A runtime environment for the program above.**

|  |  |
| --- | --- |
| **Global area** | **MAXSIZE** |
| **Activation record of main procedure** | **TABLE (1)**  **(2)**  **…**  **(10)** |
| **TEMP** |
| **3** |
| **Activation record of procedure QUADMEAN** | **A** |
| **SIZE** |
| **QMEAN** |
| **Return address** |
| **TEMP** |
| **K** |
| **Unnamed location** |

A is pointing to TABLE(1),TABLE(2),..

QMEAN is pointing to TEMP

Note: The unnamed location is used to store temporary value during the computation of arithmetic expression.

In Fortan77, parameter values are implicitly memory references, so the locations of arguments of the call(**TABLE**, 3, and **TEMP** ) are copied into the parameter locations of **QUADMEAN** .

This has several consequences.

First, an extra dereference is required to access parameter values.

Second, array parameters do not need to be reallocated and copied (thus, array parameter **A** in **QUADMEAN** is allocated only one space, which points to the base location of **TABLE** during the call).

Third, constant arguments, such as the value 3 in the call, must be stored to a memory location and this location used during the call.